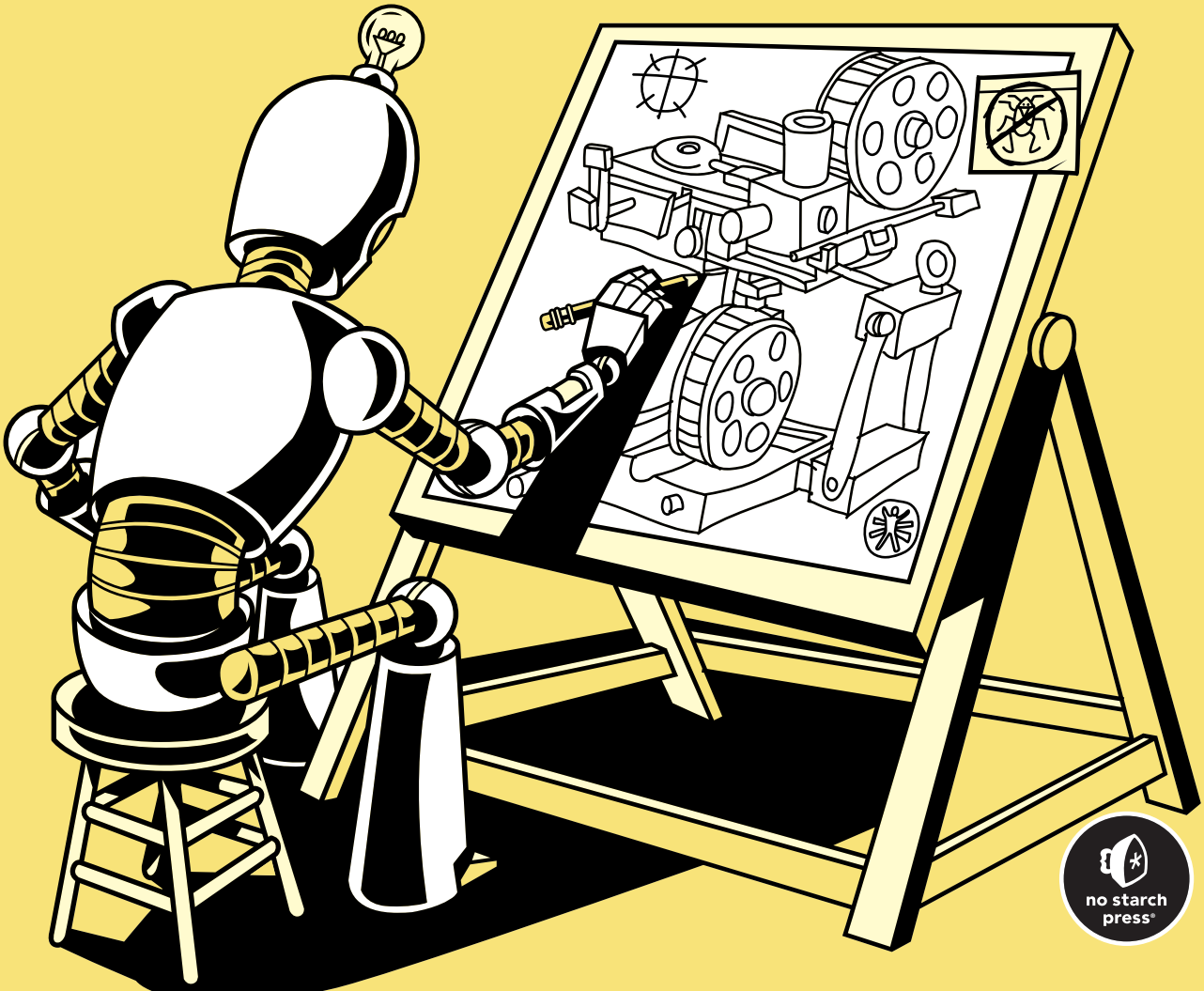
BUILDING A DEBUGGER

WRITE A NATIVE *x64 DEBUGGER* FROM SCRATCH

SY BRAND



no starch press®

BUILDING A DEBUGGER

Write a Native x64 Debugger from Scratch

by Sy Brand

no starch press®

San Francisco

First printing

29 28 27 26 25 1 2 3 4 5

ISBN-13: 978-1-7185-0408-0 (print)
ISBN-13: 978-1-7185-0409-7 (ebook)

Published by No Starch Press®, Inc.
245 8th Street, San Francisco, CA 94103
phone: +1.415.863.9900
www.nostarch.com; info@nostarch.com

Publisher: William Pollock
Managing Editor: Jill Franklin
Production Manager: Sabrina Plomitallo-González
Production Editor: Miles Bond
Developmental Editor: Frances Saux
Cover Illustrator: Josh Kemble
Interior Design: Octopod Studios
Technical Reviewer: Peter Cawley
Copyeditor: George Hale
Proofreader: Rachel Head

Library of Congress Control Number: 2024048685

For customer service inquiries, please contact info@nostarch.com. For information on distribution, bulk sales, corporate sales, or translations: sales@nostarch.com. For permission to translate this work: rights@nostarch.com. To report counterfeit copies or piracy: counterfeit@nostarch.com. The authorized representative in the EU for product safety and compliance is EU Compliance Partner, Pärnu mnt. 139b-14, 11317 Tallinn, Estonia, hello@eucompliancepartner.com, +3375690241.

[E]

To everyone who has ever set a breakpoint in their code and thought to themselves, “I wonder how that works, then?”

About the Author

Sy Brand is Microsoft's C++ Developer Advocate. With more than 10 years' experience in developer tooling, they've worked on profilers, compilers, language runtimes, standard libraries, and, of course, debuggers. They've also contributed to the standards for C++, DWARF, and HSA.

They graduated from the University of St. Andrews with a first-class degree in computer science, where they specialized in compiler implementation. Outside the world of technology, they are a published poet, maker of experimental films and music, activist, and parent to three cats and one entire human.

About the Technical Reviewer

Peter Cawley is an expert in x64 assembly, ELF files, linkers, loaders, and low-level systems programming. He holds a first-class degree in mathematics and computer science from Oxford. After many enjoyable years writing C++ in the high-frequency trading industry, he retired to a quiet corner of Scotland with his wife and their cat. You can often find him online as @corsix.

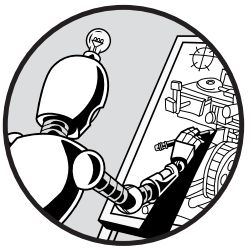
BRIEF CONTENTS

CONTENTS IN DETAIL

ACKNOWLEDGMENTS

This book would not exist without the work and support of many people. Thanks to my editors, Frances Saux and Miles Bond, for tirelessly editing out my Britishisms and propensity for using the passive voice. Thanks also to my technical reviewer, Peter Cawley, for his patience in working through this mammoth book. Thanks to those in my life who have put up with me talking about nothing other than book progress for the past two years, especially Lizzie, Maren, and Quinn. Thanks to Yoshua Wuyts for many conversations that helped shape parts of the book. Thanks to Kyle Loveless for several corrections.

INTRODUCTION



Have you ever set a breakpoint on a line of code and wondered how it worked, or printed the value of a variable and become curious about how the debugger knew where to look for it in memory? Or maybe you want to know what happens in the depths of your computer when you attach to existing processes and bend them to your will.

If so, this is the book for you. *Building a Debugger* will demystify the black boxes that debuggers so often appear to be. It will reveal how they function and walk you through building one of your own, beginning from an empty folder on your filesystem and ending with a functional tool that can step through your programs, set breakpoints, read variables, and more. Along the way, you'll pick up a wealth of knowledge about Linux, operating systems in general, and the low-level operations of computer systems.

What Is a Debugger?

It all started with a moth. In 1947, a team at Harvard University was working on the Harvard Mark II electromechanical computer. This large machine weighed over 20 metric tons and occupied more than 370 m^2 of floor space.

On September 9, the team was diagnosing an issue with the Mark II and traced it back to a rather unusual source: a moth had gotten stuck in one of the electromagnetic relays on which the machine was based, causing it to malfunction. After they removed the minibeast, they taped it to a logbook with the phrase "First actual case of a bug being found." While computer scientists had used the term "bug" before this incident, the poor moth's demise contributed to its widespread use in the lexicon.

Most of the time, however, we're not looking for actual bugs in our systems, so we don't require a screwdriver and magnifying glass to identify issues. Instead, we need a program that can trace, manipulate, and visualize the state of a different program running on our system. Such a program is called a *debugger*.

Developers typically use debuggers for tasks such as tracing the control flow of their code as it runs, inspecting the values of variables at various points of execution, halting the program at predetermined locations, and executing functions inside the running process.

In this book, we're particularly concerned with debuggers for compiled code that runs directly on your central processing unit (CPU), written in a language like C, C++, Rust, or FORTRAN. Debuggers for these programs need to interface directly with the operating system and the underlying hardware, which can lead us to some deep insights into how computers actually work.

What Will We Build?

Over the course of this book, we'll build a command line debugger for native code. I'll call the debugger *sdb*, for *Sy's Debugger*, but you can call yours whatever you like. (You might want to avoid conflicting with the names of existing debuggers, like GDB and LLDB, however.)

When we begin, the project will consist of a single executable, *sdb*, which the user can interact with on the command line. As we progress, we'll build the majority of the debugger as a library, *libsdb*, and then author a command line driver that uses its functionality. This design will make it easier to write automated tests for the code and enable you to develop applications that interact with the debugger programmatically.

The Source Language

We'll write the debugger's source code in C++17. The techniques I'll teach you aren't specific to C++, however, and you can choose a different language if you'd like. While it's easier to handle some of the low-level machinery in a compiled language like C, C++, or Rust, nothing will stop you from translating the book's code into a higher-level language like Python.

The code for *sdb* is available at *https://github.com/TartanLlama/sdb*. Each chapter has its own branch. For example, the code for Chapter 1 is available at *https://github.com/TartanLlama/sdb/tree/chapter-1*.

The book assumes a basic understanding of C++. If you know other C-like languages, you may be able to follow along without much difficulty. To brush up on C++, I recommend the online reference *https://cppreference.com* and *C++ Crash Course* (No Starch Press, 2019) by Josh Lospinoso. I'll also explain certain advanced or modern features of C++ in textual asides.

Limitations

The debugger we'll write in this book won't be thread-safe. We'll assume that only one thread is using it, and I make no guarantee that it will work if you try to use it in a multithreaded context. I chose this design because making the project thread-safe would require significantly more code unrelated to the core activities of the debugger. Feel free to make your debugger thread-safe on your own if you'd like.

What's in This Book?

In each chapter, we'll tackle a debugger feature, explore the background necessary to understand how it works, and add code to implement it:

Chapter 1: Project Setup Walks you through setting up your environment for building the debugger.

Chapter 2: Compilation and Computer Architecture Provides you with the theoretical knowledge you'll need to work through the book, including information about user space and kernel space on Linux, the process of compilation, how the operating system loads programs, and more.

Chapter 3: Attaching to a Process Walks you through launching and attaching to Linux processes and begins the implementation of the debugger library and command line driver.

Chapter 4: Pipes, procfs, and Automated Testing Introduces methods for inter-process communication on Linux and using the process filesystem (procfs) to get information about processes on the system, and helps you begin an automated testing suite for the debugger.

Chapter 5: Registers Details the hardware registers available on x64 systems, including general-purpose, floating-point, and debug registers, and how the debugger can interact with them.

Chapter 6: Testing Registers with x64 Assembly Gives you an introduction to x64 assembly language and shows you how to use it to test the hardware register functionality implemented in the preceding chapter.

Chapter 7: Software Breakpoints Explains how software breakpoints work on x64 systems and walks you through adding support for them to the debugger.

Chapter 8: Memory and Disassembly Helps you add features for reading and manipulating the memory of running processes and disassembling the machine code that is currently executing.

Chapter 9: Hardware Breakpoints and Watchpoints Helps you add support for hardware breakpoints and memory watchpoints to the debugger.

Chapter 10: Signals and Syscalls Deepens your understanding of Unix signals and Linux syscalls and helps you extend the functionality of the debugger to handle them.

Chapter 11: Object Files Introduces ELF, the object file format for Linux, and walks you through writing a parser for it.

Chapter 12: Debug Information Introduces the debug information format for Linux object files and helps you implement a parser for that format.

Chapter 13: Line Tables Describes how debug information encodes the mapping between source code and machine code and details writing a parser for this information.

Chapter 14: Source-Level Breakpoints and Stepping Helps you use the debug information formats described in the preceding chapters to implement breakpoints and code stepping at the level of the source code rather than at the machine code level.

Chapter 15: Call Frame Information Describes the format that Linux uses for encoding the layout of the program call stack and walks you through implementing a parser for it.

Chapter 16: Stack Unwinding Describes how to use the parser created in the preceding chapter to implement a stack unwinder that can generate program backtraces and restore hardware registers to the values that they had before a function call began.

Chapter 17: Shared Libraries Details how Linux systems load and track shared libraries and helps you extend the debugger to support debugging code that is loaded from a shared library.

Chapter 18: Multithreading Describes how Linux uses threads to support programs that execute multiple tasks at the same time and helps you add support for debugging multithreaded programs to the debugger.

Chapter 19: DWARF Expressions Explores how DWARF expressions encode the locations of variables in a running program and walks you through extending the debug information parser to support them.

Chapter 20: Variables and Types Covers the format for variable and type descriptions in debug information and helps you add basic support for visualizing variable contents in the debugger.

Chapter 21: Expression Evaluation Walks you through writing a simple expression evaluation engine that allows debugger users to execute function calls inside the running process.

Chapter 22: Advanced Topics Briefly describes some other debugger features you can research on your own if you want to further extend your debugger.

Most chapters conclude with a set of questions to test your knowledge. You'll find sample answers to these questions in the book's appendix.

The Target Platform

The debugger described in this book will target 64-bit x86 Linux systems. I'll use the term *x64* throughout the book to refer to the 64-bit flavor of x86. I don't make any assumptions about which distribution you're using; the provided code should work as well on Ubuntu as it will on Fedora, Arch, or some Linux-from-scratch monstrosity you built when you were bored.

I do assume that you have a C++ compiler, `binutils`, and a C++ build system like `make` installed. On Debian-based systems, you can install all of these with `apt install build-essential`.

For Windows Users

If you're on Windows, you're best off installing the Windows Subsystem for Linux (WSL). With this setup, you'll be able to develop your application from Windows and run it on Linux almost seamlessly. The installation process for WSL is detailed at *https://aka.ms/wsl*.

At the time of this writing, opening an elevated command prompt (using "Run as administrator") and executing `wsl --install` will install all WSL components and Ubuntu.

For macOS Users

If you're using an x64 macOS system, you can install a Linux virtual machine or, if you want something more lightweight, use a Docker container. To run a virtual machine, you could use UTM, which you can find at *https://mac.getutm.app*. You can find instructions on installing and using Docker at *https://www.docker.com*.

If you're using an ARM-based Mac, such as one with an M1 or M2 chip, you're in a more difficult position, as current x64 emulation technologies don't provide sufficient support for the code in this book. You could install an ARM-based Linux virtual machine or Docker container and attempt to adapt the x64-specific code for ARM, but you'd have to do independent research on the available hardware registers and calling conventions. If you don't feel prepared to do this, I'd recommend finding a secondhand x64 system and using that.

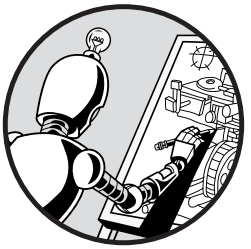
ABBREVIATIONS

ABI	application binary interface
ASLR	address space layout randomization
AVX	Advanced Vector Extensions
CFA	canonical frame address
CIE	common information entry
DAP	Debug Adapter Protocol
DIE	DWARF information entry
ELF	executable and linkable format
FDE	frame description entry
FPR	floating-point register
GDB	GNU debugger
GDB/MI	GDB machine interface
GDB RSP	GDB remote serial protocol
GOT	global offset table
GPR	general-purpose register
LSB	Linux Standard Base
LSM	Linux Security Module
NTFPOC	non-trivial for the purposes of calls
PGID	process group identifier
PID	process identifier
PIE	position-independent executable
PLT	procedure linkage table
RAII	resource acquisition is initialization
RAM	random access memory
SIMD	single instruction multiple data
SSE	Streaming SIMD Extensions
SYSV ABI	System V application binary interface
TGID	thread group identifier
TID	thread identifier
TLS	thread-local storage
vDSO	virtual dynamic shared object

1

PROJECT SETUP

Take my hand
and step with me
into the long grass.



In this chapter, we'll set up our debugger development environment. To build the code, we'll use CMake, an open source family of tools that abstract the compiler and build system used for a given platform, allowing us to focus on the projects we're building and the dependencies between them. We'll also use the `vcpkg` package manager to fulfill a few dependencies.

You can use different compilation and package management tools if you prefer, but all instructions in this book assume you're using CMake and `vcpkg`, so you'll find it easiest to follow suit.

The Directory Structure

Before we write any code, let's create the directory structure for the project. Use the set of files and directories shown in Figure 1-1. (All files can be empty for now.)

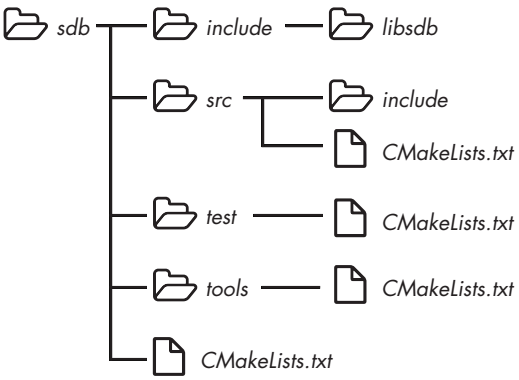


Figure 1-1: The directory structure for the project

CMake instructions live in files called *CMakeLists.txt*. Generally, you'll create one *CMakeLists.txt* file in the root directory of your project, as shown here, and then write additional *CMakeLists.txt* files inside the project's various source-containing directories. These files are responsible for building the code in their respective directories.

The code for `libsdb`, the library that will implement the bulk of the debugger, will be split between the *sdb/include/libsdb* and *sdb/src* directories. The *sdb/include/libsdb* directory will contain all the public headers that a consumer of the library needs to interact with it. The implementation will live in *sdb/src*, which contains a nested *include* directory for any private headers (ones that aren't intended for end users).

Like good little software engineers, we'll write automated tests as we progress through the project. These will live in the *test* directory. The command line utilities we build will go in the *tools* directory. We'll create only one such utility, the main *sdb* command line interface, as part of this book, but if you decide to add the remote debugging support described in Chapter 22, you should place the debug server in this directory as well.

Writing the Top-Level CMake File

We write build instructions for CMake in a domain-specific language also called CMake. Fill in the top-level *CMakeLists.txt* like so:

```
cmake_minimum_required(VERSION 3.19)

project("sdb" LANGUAGES CXX)

include(CTest)

add_subdirectory("src")
add_subdirectory("tools")

if(BUILD_TESTING)
    add_subdirectory("test")
endif()
```

The lines of this file are function calls. We pass arguments to CMake functions in parentheses and delimit them by spaces (not by commas, as in C++). CMake functions support keyword arguments; for example, in the `cmake_minimum_required` call, `VERSION` is a keyword, so `VERSION 3.19` specifies that the value for the `VERSION` parameter is `3.19`.

After declaring a minimum version, we give a name to the project, `sdb`, and tell CMake to enable C++ support. Including the `CTest` module enables CMake's built-in test system and automatically adds a `BUILD_TESTING` variable that users can set to select whether to build the tests when configuring the project. (This option is on by default.) We then load the *CMakeLists.txt* files in the *src* and *tools* directories (although these files are currently empty), and we finish by adding the *test* subdirectory only if the user hasn't explicitly disabled testing.

The CMake language can get complicated. However, we won't use many of its features in this project, and I'll explain the necessary syntax over the course of the book. Now let's write some C++ so we have something to build.

Building the libsdb Library

To test that the build works, add a temporary header to the `libsdb` library. Create a new file at *sdb/include/libsdb/libsdb.hpp* with a function declaration that the *sdb* executable can use later:

```
❶ #ifndef SDB_LIBSDB_HPP
  #define SDB_LIBSDB_HPP

❷ namespace sdb {
      void say_hello();
  }

  #endif
```

This is just a temporary header for ensuring our build works, but it applies principles we'll follow throughout the book. All header guards ❶ in `libsdb` will be of the form `SDB_`*`filename`*`_HPP`, and we'll always use the namespace `sdb` ❷.

Now that we've declared a `say_hello` function, we can implement it in *sdb/src/libsdb.cpp*:

```
  #include <iostream>
❶ #include <libsdb/libsdb.hpp>

  void sdb::say_hello() {
      std::cout << "Hello, sdb!\n";
  }
```

Header files in the public *sdb/include/libsdb* directory should always be included using angle brackets; thus, we use angle brackets to include *libsdb.hpp* ❶. By contrast, header files in the private *sdb/src/include* directory

should be included using quotes. We then implement the `say_hello` function to print a message to `stdout`.

Now we can tell CMake how to build the library. Add the following to *sdb/src/CMakeLists.txt*:

```
❶ add_library(libsdb libsdb.cpp)
❷ add_library(sdb::libsdb ALIAS libsdb)

❸ set_target_properties(
    libsdb
    PROPERTIES OUTPUT_NAME sdb
  )

❹ target_compile_features(libsdb PUBLIC cxx_std_17)

❺ target_include_directories(
    libsdb
    PRIVATE include
    PUBLIC ${CMAKE_SOURCE_DIR}/include
  )
```

First, we tell CMake to create a target called `libsdb` with a single source file, *libsdb.cpp* ❶. It's best practice to use namespaced library targets in CMake, as it can result in more understandable errors if we make mistakes. We do that by creating an alias ❷.

By default, CMake creates libraries called *lib<target name>* on Linux. Since we don't want our compiled library to be called *liblibsdb.a*, we tell CMake that our output library should be called *sdb* ❸. Now it should generate *libsdb.a*.

We specify that the compiler should compile `libsdb` in C++17 mode ❹. Since we specify `PUBLIC` visibility, any targets that compile against `libsdb` will inherit this property. Finally, we specify the *include* directories for `libsdb` ❺. We use `PRIVATE` visibility for the *sdb/src/include* directory, as dependent targets shouldn't transitively add it to their *include* directories. However, we do want dependent targets to inherit *sdb/include*, so we make it `PUBLIC`.

Run CMake to build the library. How to run CMake depends on whether you're building from the command line or from an integrated development environment (IDE), and on which IDE you're using. Refer to your tool's documentation for instructions.

Building the sdb Executable

Now that we've built a library, we can write a simple program that consumes it. Create a file at *sdb/tools/sdb.cpp* that calls `sdb::say_hello`:

```
#include <libsdb/libsdb.hpp>

int main() {
    sdb::say_hello();
}
```

Then, fill in the *sdb/tools/CMakeLists.txt* file with code to build the executable:

```
add_executable(sdb sdb.cpp)
target_link_libraries(sdb PRIVATE sdb::libsdb)
```

We tell CMake to create a target called `sdb` that represents an executable file, also called *sdb*. For now, we have a single source file: *sdb.cpp*. Next, we specify a dependency on `libsdb`. This will handle adding the public *include* directory to the include paths for the build, linking against the library, and ensuring we're building in C++17 mode.

Build and run *sdb*. The exact steps that you'll take to do this will depend on the tools you are using. If you're using CMake from the command line, you can run the following instructions:

```
$ mkdir build && cd build
$ cmake ..
$ cmake --build .
$ ./tools/sdb
Hello, sdb!
```

If you're using an IDE like Visual Studio, refer to the documentation for your IDE.

Making the Dependencies Accessible

The previous section showed you how to build the `libsdb` library and an *sdb* executable that uses it. However, if you want to be able to install both of these components in a way that is easy for other CMake projects to consume, there's some more work you need to do.

Currently, if we installed `libsdb`, the public headers would live in some known installation location rather than in our source tree. As such, we need to tell users to look for them in different locations. We can do this with *generator expressions*, which generate information specific to the current build configuration. Edit the `target_include_directories` call in *sdb/src/CMakeLists.txt* like so:

```
target_include_directories(libsdb
    PUBLIC
    ❶ $<INSTALL_INTERFACE:include>
    ❷ $<BUILD_INTERFACE:${CMAKE_SOURCE_DIR}/include>
```

```
    PRIVATE
        ${CMAKE_SOURCE_DIR}/src/include
)
```

With a couple of generator expressions, we specify the correct header locations for different cases. If the user is accessing an installed version of `libsdb`, the public headers live in the *include* directory relative to the installed location ❶. If they're using a version in the build tree (for example, if they're linking the *sdb* executable), we should use the *include* directory inside the source tree ❷.

Next, we store information about the library target in a format that can later be saved to disk to provide consumers with information about how to use the library and where it is installed:

```
include(GNUInstallDirs)
install(TARGETS libsdb
    EXPORT sdb-targets
    LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
    ARCHIVE DESTINATION ${CMAKE_INSTALL_LIBDIR}
    RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
    INCLUDES DESTINATION ${CMAKE_INSTALL_INCLUDEDIR}
)
```

The `GNUInstallDirs` module contains variables that specify common installation directories: for example, */include* for headers and */lib* for libraries. Using these, we create an export target called `sdb-targets` with information about where to install the different components of the project.

Now we need to install the public headers:

```
install(
    DIRECTORY ${PROJECT_SOURCE_DIR}/include/
    DESTINATION ${CMAKE_INSTALL_INCLUDEDIR}
)
```

The final thing we need to do to make `libsdb` importable by other CMake projects is save its export information to a file:

```
install(
    EXPORT sdb-targets
    FILE sdb-config.cmake
    NAMESPACE sdb::
    DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/sdb
)
```

We save this information to *sdb-config.cmake*. CMake expects this file to be named either *<project>-config.cmake* or *<project>Config.cmake*. We specify that the exported targets should be under the `sdb` namespace so that users can link to the `sdb::libsdb` target.

We also want to install the command line tool, so we'll add instructions for this in *sdb/tools/CMakeLists.txt*:

```
include(GNUInstallDirs)
install(
    TARGETS sdb
    RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
)
```

This code simply installs `sdb` to the default binary directory.

Dependency Management with vcpkg

Dependency management in C++ can become a nightmare. Before moving on, let's use `vcpkg` to make it more pleasant. This tool will automate the process of downloading, building, and installing our dependencies and ease the process of consuming them. You need just two commands to set up `vcpkg`. Run these outside of the *sdb* directory (for example, in the parent directory of *sdb*):

```
$ git clone https://github.com/microsoft/vcpkg.git
$ ./vcpkg/bootstrap-vcpkg.sh
```

When you run CMake, you'll now need to pass one additional command line option on your first configuration to set up `vcpkg` integration. Pass the path to *vcpkg/scripts/buildsystems/vcpkg.cmake* as the `CMAKE_TOOLCHAIN_FILE` argument. For example, from the command line, you can run this:

```
$ cd build
$ cmake .. -DCMAKE_TOOLCHAIN_FILE=/path/to/vcpkg/scripts/buildsystems/vcpkg.cmake
```

If there's already a *CMakeCache.txt* file generated by a previous run of CMake, you'll need to delete it first, which you can do by removing it manually, passing the `--fresh` flag to CMake, or using whatever cache removal facility your IDE has. This is necessary so that CMake uses the new toolchain file to initialize the build environment. The presence of `-- Running vcpkg install` in the CMake output indicates success.

Next, list your project's dependencies in a *vcpkg.json* file located in the root *sdb* directory. We'll pick up new dependencies over the course of the book, but for now, we'll begin with just one: `libedit`, a library that will enable us to quickly set up a decent command line interface. It's actually the same library that LLDB uses to implement its command line interface. Write this in *sdb/vcpkg.json*:

```
{
    "dependencies": ["libedit"]
}
```

Now configuring CMake will build and install `libedit` into a project-local directory. Usually, we would use `find_package` and `target_link_libraries` to locate the dependency and link against it. However, `libedit` does not supply

the necessary CMake modules to enable this. Instead, we can use `pkgconfig`, which is a tool for locating installed libraries. Edit *sdb/CMakeLists.txt* like so:

```
cmake_minimum_required(VERSION 3.19)

project("sdb" LANGUAGES CXX)

find_package(PkgConfig REQUIRED)
pkg_check_modules(libedit REQUIRED IMPORTED_TARGET libedit)
--snip--
```

We find the `PkgConfig` package, then use its `pkg_check_modules` function to locate the `libedit` module. Next, add `PkgConfig::libedit` as an additional target to link against in *sdb/tools/CMakeLists.txt*. Edit the existing `target_link_libraries` call like this:

```
target_link_libraries(sdb PRIVATE sdb::libsdb PkgConfig::libedit)
```

Done! Before we start building a debugger, we'll set up a simple testing environment.

Testing

As with all languages that reach a broad level of adoption, C++ has a myriad of options for automated testing, each with its advantages and disadvantages. This book uses Catch2, which is easy to integrate with the project and allows us to receive useful error messages by writing simple expressions like `REQUIRES(x == y)`, rather than something convoluted like `ASSERT_EQUAL(x, y)`.

Because `vcpkg` has a package (or *port*, in `vcpkg` parlance) for Catch2, we can add it to `sdb` by modifying the *vcpkg.json* file:

```
{
    "dependencies": ["libedit", "catch2"]
}
```

This book uses version 3 of Catch2. We find the package in the *sdb/CMakeLists.txt* file:

```
--snip--
if(BUILD_TESTING)
    find_package(Catch2 CONFIG REQUIRED)
    add_subdirectory ("test")
endif()
```

Then, we can add a new *tests* executable in the *test* directory. We'll create just a single test program to which we'll add all test cases, although you can split them into separate files if you prefer. Edit the *sdb/test/CMakeLists.txt* file like so:

```
add_executable(tests tests.cpp)
target_link_libraries(tests PRIVATE sdb::libsdb Catch2::Catch2WithMain)
```

The `Catch2::Catch2WithMain` target supplies its own `main` function, which deals with command line arguments on its own. We pass it to `target_link_libraries`.

Now we can start writing some tests. Let's write a simple no-op test to validate the environment. In *sdb/test/tests.cpp*, add the following:

```
#include <catch2/catch_test_macros.hpp>

TEST_CASE("validate environment") {
    REQUIRE(false);
}
```

Test cases in Catch2 use the `TEST_CASE` macro, which takes a description of the test case and, optionally, a hierarchical tag, like `[validate.environment]`, which can be used to run subsets of tests. We'll mainly use the `REQUIRE` macro to see if our assertions hold. It immediately aborts the test case if it receives an unexpected value. You could also use `CHECK`, which will fail the test case but allow the rest of it to run.

If you run this test, you should see a report like this:

```
tests is a Catch2 host application.
Run with -? for options

-------------------------------------------------------------------------------
validate environment
-------------------------------------------------------------------------------
../../../test/tests.cpp:3
...............................................................................

../../../test/tests.cpp:4: FAILED:
  REQUIRE( false )

===============================================================================
test cases: 1 | 1 failed
assertions: 1 | 1 failed
```

Hooray! The test failed! This means that the environment is working as expected. Now we can make it pass by changing `false` to `true`:

```
--snip--
    REQUIRE(true);
--snip--
```

Run the test again, and you should see something like this:

```
===============================================================================
All tests passed (1 assertion in 1 test case)
```

Great! Now that we have a working test environment, we can begin building a debugger.

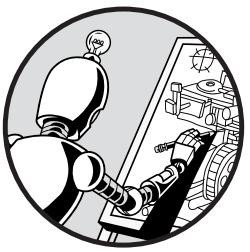
Summary

In this chapter, you set up a development environment and dependencies. Before you start writing the debugger's code, the next chapter covers some of the system components you'll be relying on throughout the book.

2

COMPILATION AND COMPUTER ARCHITECTURE

. . . for the buildings in this place
are strange and cast with rainbows,
like those who dwell here.



Debuggers need to interact with components of the system that are as close to the hardware as you can reasonably get without working on the implementation of the chips themselves. Therefore, you'll require an understanding of how some of these components operate to be able to imbue your debugger with the powers you wish it to have.

In particular, a debugger interfaces closely with a program's compiler, the operating system hosting the target process, and aspects of the computer's architecture. Each of these topics alone could be the subject of an entire book, so we'll focus on the minimum you'll need to know to get your debugger to work.

If you're already familiar with the basics of compilers, computer architecture, and operating systems, take a minute to appreciate the wealth of knowledge you have accrued and move on to the next chapter. By contrast, if you're a beginner, you may find this chapter to be dense with information and want to revisit relevant sections as you encounter their uses in the rest of the book.

Compilation

A *compiler* is a program that translates source code written in one programming language into either binary code or another programming language. In this book, we'll focus on compilers that generate binary code that runs directly on your CPU. This type of binary code is also known as *native code* or *machine code*. When we built the *sdb.cpp* source file into the *sdb* executable, we used a compiler (most likely GCC or Clang). The programs we debug with *sdb* will also have been generated by a compiler.

Compilation is a *lossy*, or *destructive*, process. This means that, generally speaking, it is impossible to reverse the process of compilation; you can't take a compiled executable and translate it back into the source code that produced it. You may be able to produce source code that does the same thing, but it will likely be missing key elements of the original, such as comments and human-readable names. While a human reader might care that you called your variable `database_transaction` rather than `all_mimsy_were_the_borogoves`, your CPU does not. As such, the compiler may discard the name during the compilation process, perhaps along with the fact that the variable existed in the first place. Figure 2-1 shows an example of this loss.

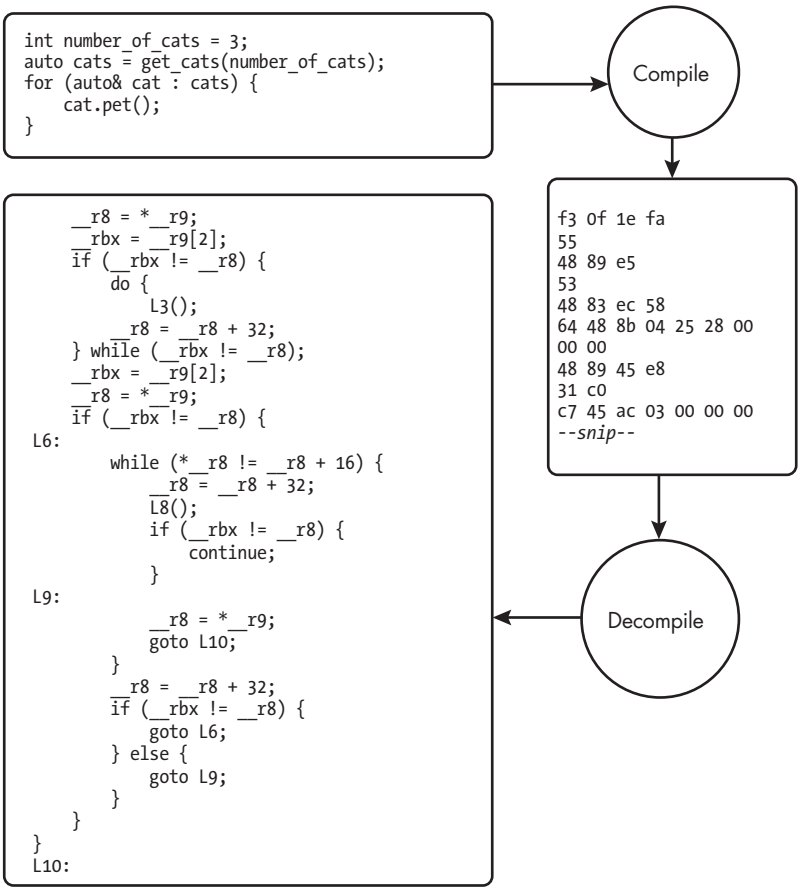


Figure 2-1: Compilation is a lossy process.

As you can see, if we take some beautiful C++ source code and compile it to machine code, we can later decompile it to a C++ representation. But notice how different it is from the original. There are none of the nice variable names, the structure is hard to understand, and we've lost the named function calls. We can't get from the machine code to the C++ used to produce it.

Encoding

When you run a compiler, you usually give it the name of a file that exists on your filesystem. This file consists of bits and bytes, 1s and 0s, like any other file, but by interpreting this binary in a particular way, the compiler can retrieve the text of some source code written in your programming language of choice.

The structure that enables this interpretation is called the file's *encoding*, and it's most likely UTF-8 or ASCII. At the end of the day, text files are binary data, just like anything on your computer. But if we understand how they are encoded, we can read and manipulate them. Figure 2-2 shows an example of a UTF-8-encoded file.

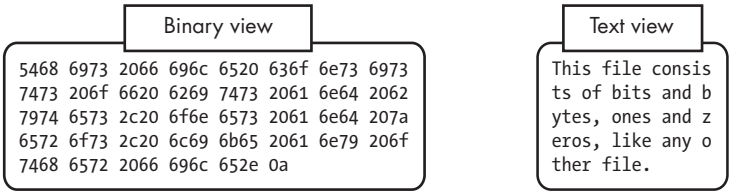


Figure 2-2: All text files are represented in binary using a text encoding scheme.

The view on the left is the hexadecimal representation of the bytes on disk. Given a definition of the UTF-8 encoding scheme, we can interpret those bytes to retrieve the text they represent.

The output of a compiler is also a binary file, but it has a different encoding than a text file. We often call these files *object files*, and as magical as they may appear, they're just like text files; if we understand how they are encoded, we can read and manipulate them.

On Linux systems, executable programs are encoded as *Executable and Linkable Format (ELF)* files, which you'll learn about in Chapter 11. macOS uses a format called Mach-O, and Windows uses one called Portable Executable (PE). While there are many differences between these formats, they exist for a common purpose: to hold the instructions generated by the compiler along with the data that is necessary for the operating system to load and execute them.

Debug Information

How is all of this information relevant to debuggers? I said earlier that compilation is destructive, meaning we can't retrieve the source code from the

machine code. Unfortunately, this is exactly what we want a debugger to do. We want to know which line of the source code we are executing, not where we are in a blob of binary instructions, and we're interested in the value of the variable `do_i_like_green_eggs_and_ham`, not the memory address `0xBABACACA`.

Fortunately, compilers help us out here. When run with the right set of compiler flags, they'll add extra metadata to the object file called *debug information*. Armed with this information, we can take a binary instruction of machine code back in time to when it was a beautiful line of text that could be read by humans.

The debug information format used on Linux systems is DWARF (a reference to the ELF file format; we're all very funny) and you'll learn more about it in Chapter 12. DWARF and other debug information formats provide a veritable gold mine of details about the source code that was used to produce a given object file, including:

- What compiler was used to produce the object file
- The paths to the source files used to compile the program
- The names of functions in the source file and where in memory the executable machine code for them will reside
- The names of variables and how to locate them while the program is running at any given point (variables may move around in memory, so it's important to be able to track this)
- How the data types in the program are structured
- Which line of source code a given machine instruction corresponds to

If you've ever tried to debug a program and not been able to see the source code that corresponds to it, or seen strange behavior while stepping through it, this is likely because the debug information wasn't sufficient for the debugger to be able to locate the source code or fully re-create how it relates to the executable. This might be because the object file was compiled without debug information, or because it was optimized and no longer corresponds exactly to lines of code in the source.

In the next section, we'll look at the parts of the operating system that are relevant to debuggers.

Operating Systems and Debuggers

Debuggers need to be able to alter the execution of a running process, retrieve information about that process, and be notified of events that occur to report these to the user. The operating system provides these facilities. In addition, you need to understand some of the ways in which operating systems load programs for execution and prevent processes from interfering with one another.

Program Loading

I mentioned that an object file is a binary file encoded in some known format. Somewhere in that object file are the instructions that make up the program you have compiled; these, too, are binary data in a special format. We'll talk about how these instructions are encoded in "Assembly and Instruction Encodings" on page 20, but for now, you can consider them amorphous blobs of data.

Somehow, when a user requests that a program be executed, this blob of binary data needs to make it from the filesystem into some place in the computer's working memory (generally, its random access memory, or RAM). The CPU then needs to be told to run the instructions contained within. This process, which is handled by the operating system, is known as *program loading*.

When the operating system is tasked with loading a program, it examines the metadata contained in the program's ELF file to understand what actions it needs to take and then sets up the environment to allow this program to be loaded into memory and run by hardware. Generally, this requires the following:

- Calculating how much memory this program requires and reserving that amount. This memory must hold the code for the program and all of its global variables. The program also needs memory for the *stack*, a structure that holds local variables and information about function calls while the program is running. You'll learn more about the stack in "Stack Frames" on page 22.
- Setting up relevant protections on these areas of memory. For example, we usually don't want to allow the execution of memory that is on the stack, as this can lead to security vulnerabilities.
- Copying the code and any preinitialized global variables from the object file into the reserved memory.
- Finding any dynamic libraries that the binary requires, loading them, and pointing the binary toward the place where these dependencies are loaded.
- Setting up registers, command line arguments, and environment variables.
- Jumping to the program's entry point and letting it continue on its merry way.

Some of these operations are special, privileged operations; you couldn't write a program yourself that did them. For example, you couldn't arbitrarily modify the registers of another process, as these privileged operations must be implemented in a special part of the operating system known as the *kernel*.

User Space vs. Kernel Space

Unless you happen to be a kernel engineer (in which case, why are you reading this chapter?), the programs you write will execute in *user space*. Programs in user space are restricted; for example, they can't read arbitrary locations in memory or write wherever they please on disk. These restrictions protect you from malicious programs and from your own mistakes. (I sleep much better at night knowing that I can't irrevocably destroy a computer by accidentally writing past the end of an array.)

But without access to the filesystem or the ability to allocate memory, it's quite difficult to write useful programs. Fortunately, the operating system supplies a way for user-space programs to make requests to the kernel. These are called *system calls (syscalls)*. You may have already encountered some of these, like `open` for opening files, and you'll meet several more while writing the debugger. Figure 2-3 shows how syscalls provide user-space programs with a way to communicate with kernel space.

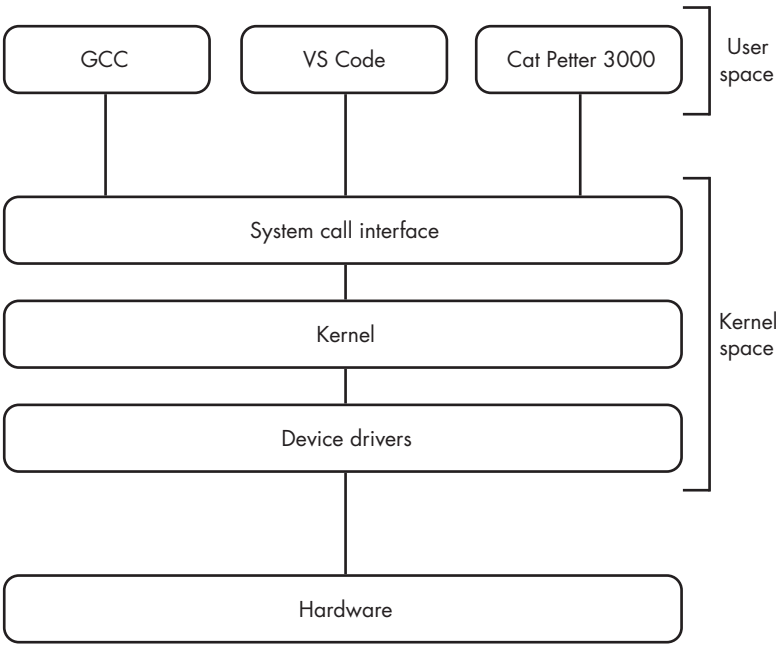


Figure 2-3: The separation of user space and kernel space

User-space programs, like GCC, Visual Studio (VS) Code, and Cat Petter 3000, cannot access the kernel, device drivers, or hardware directly. Instead, they make syscalls that provide controlled access to certain kernel operations.

Another way in which the operating system protects against badly behaved processes is by lying to you about memory.

Virtual Memory

If you've ever printed the value of a pointer in C or C++, you've likely seen some hexadecimal value that looks like `0xBA5EBA11`. This number identifies the location of a byte in memory. The byte before it in memory lives at `0xBA5EBA10`, and the byte after it lives at `0xBA5EBA12`.

However, if you ran two user-space programs and both of them wrote to `0xBA5EBA11`, they would actually be writing to different physical locations in memory. This is because memory accesses made from user space don't go straight to RAM. Instead, the operating system maintains a mapping of which parts of memory it has allocated to each process and, with the help of the hardware, translates memory accesses accordingly. Figure 2-4 shows a visualization of this translation.

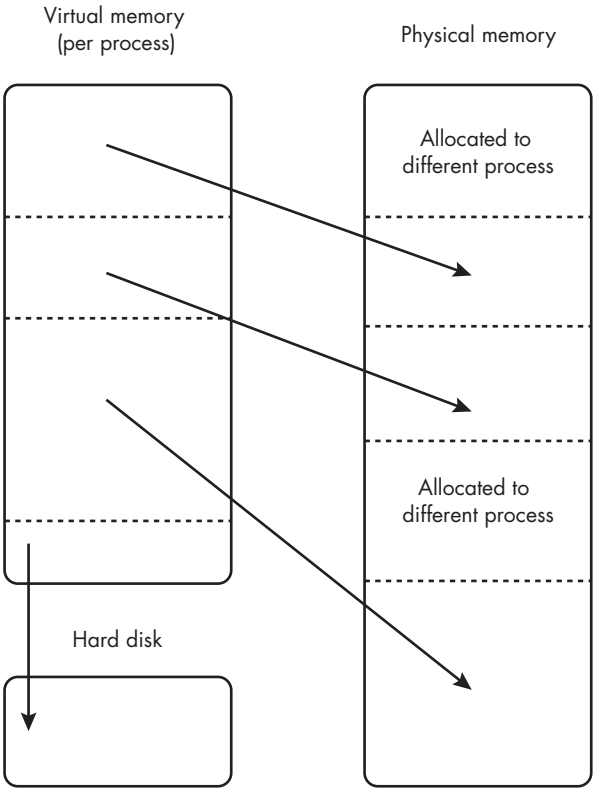


Figure 2-4: Virtual memory translation

The operating system might allocate different blocks of virtual memory for a given process to completely different areas of physical memory. Their physical address space may even be interrupted by blocks of memory allocated to a different process. This not only helps protect processes from

each other but also allows the operating system to pretend the computer has more memory than it actually does, as the real storage for a given virtual memory address might live on disk and be read into RAM only when requested. Sneaky.

If this is the case, how can programs read memory from another process? The answer, once again, involves syscalls.

Debug APIs

Modern operating systems provide syscalls to carry out common debugger tasks. These aren't high-level operations like "set a breakpoint on line 431 of *cat_petter.cpp*." They're close-to-the-hardware activities like "read memory location `0xF005BA11`."

The main debug syscall provided by Linux and macOS to carry out these low-level jobs is `ptrace`, and we'll be using it throughout our debugger to make requests to the operating system. Windows instead exposes different functions for each job a debugger might want to carry out. These functions have descriptive names, like `ReadProcessMemory` and `DebugBreakProcess`.

We'll look at how `ptrace` works under the hood in Chapter 10. In the meantime, let's examine another way for processes to communicate with one another: signals.

Signals

In Unix-like systems, *signals* are simple messages that processes can exchange to trigger specific behaviors, like halting a process or telling it that a network connection was interrupted.

UNIX VS. LINUX VS. POSIX

Unix, Linux, and POSIX are related technologies, but they're not the same. *Unix* is a family of operating systems. First released in 1971 for the PDP-11 minicomputer, it proved incredibly influential, inspiring a slew of "Unix-like" operating systems that follow similar design principles and provide suites of similar programs for interacting with the system.

Linux is a family of Unix-like operating systems that use the Linux kernel. *POSIX* is a family of standards that define common services and utilities for Unix-like operating systems to allow them to interoperate. Some examples of these are signals, process creation facilities, and filesystem operations.

Signals consist of just an integer identifier with an associated name and meaning. You can run `kill -l` at a command line to print out the list of signals available on your machine and their names. For me, this prints the following:

```
1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5)  SIGTRAP
6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL      10)  SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15)  SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20)  SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25)  SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30)  SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37)  SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42)  SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47)  SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52)  SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57)  SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62)  SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

You may be familiar with some of these signals already. For example, a process receives `SIGSEGV` when it attempts to read a virtual memory address that it's not allowed to and `SIGINT` when a user presses CTRL-C at a command line to terminate it. We'll use a handful of these signals over the course of the book, and I'll explain what they do as we come to them.

Signals may originate from another user-space process or from the kernel. A program can install a *signal handler* that calls some function when a specific signal is received. For example, `SIGTERM` terminates a process while allowing it to clean up any resources (such as file handles and database transactions). A process could install a `SIGTERM` handler that does the cleaning up before exiting.

Signals sent by the kernel will often result from *hardware interrupts*. When a process reads memory it's not supposed to, tries to execute an invalid instruction, or performs some other operation that isn't allowed, the hardware notifies the operating system by calling *interrupt handlers*, which in turn may send signals to the offending process. Interrupt handlers are kind of like signal handlers, but they're set up in special regions of memory and interact directly with the hardware. You'll learn more about these in Chapter 10.

To build a debugger, you also need a basic understanding of computer architecture. We'll take a look at that next.

Computer Architecture

Because debuggers interact with the raw memory of processes and must map hardware concepts to source code, you should understand how a program executes at the CPU level, starting with memory.

Registers

As debugger engineers, we're interested in two types of memory: virtual memory, which we've already discussed, and registers. *Registers* are very small regions of memory that are very fast to access. If the hard disk is like a huge, hulking behemoth and RAM is like a human with an average running speed, then registers are like Speedy Gonzales: small and very fast. Registers are around 100 times faster to access than RAM, and x64 has more than 100 registers, each with its own purpose. You'll learn about these in Chapter 5. Some of the common purposes of registers are:

- To hold function arguments and return values
- To hold local variables
- To give information about the status of the CPU
- To keep track of the instruction currently being executed

The register used to keep track of the current instruction is called the *program counter* or *instruction pointer*.

Program Counter

Machine code is nothing more than bytes that live in memory, and the program counter is just a pointer to the place in memory where the current instruction is located. When this instruction is executed, the program counter usually moves to the next instruction in memory. If the instruction is a function call or some instance of control flow (such as instructions resulting from an `if` statement), then the program counter might jump to a different part of memory so it can access the new code to be executed.

Assembly and Instruction Encodings

You might be wondering, if machine code is nothing more than bytes in memory, how does the CPU understand what they mean? For this purpose, the CPU has an *instruction encoding* that specifies how these bytes should be interpreted. Instruction encoding schemes can get very complicated (especially on x64), but in essence, the encoding for a given instruction specifies the following:

- What operation should be carried out, known as the *opcode*
- What data the operation should be executed on, known as the *operands*

Let's look at an example instruction from a computer architecture with a far simpler encoding scheme than x64: the 6502 processor used in BBC Micro computers. Here are a couple of instructions written in 6502 assembly language:

```
LDA     #1
ADC     #2
```

Assembly language is a human-readable textual format that a tool called an *assembler* can convert to a binary representation. Humans use assembly because we generally don't like to write binary machine code by hand, as it's very difficult and error prone. It would be like writing a book by entering the UTF-8 encoding of the text in binary, a process I can assure you I didn't follow for this book.

The 6502 has a register called `A`, or the *accumulator*, for arithmetic instructions. The first instruction in the previous listing, `LDA`, loads a value into the accumulator. In this example, we load the value `1`. The `#` prefix clarifies that this value is a number rather than a memory address. Such numbers expressed in assembly are called *immediate operands*.

The second instruction, `ADC`, adds a value to the accumulator. In this case, we add `2`, leaving the accumulator with the value `3`. When run through an assembler, these text instructions will be converted to their binary encoding. For the 6502, instructions are encoded using 1 byte for the opcode and between 0 and 2 bytes for operands.

Let's take `LDA #1`, for example. The opcode for `LDA` with an immediate operand is `10101001`, according to the 6502 processor's programming manual. The value `1` in binary is `00000001`, so the binary encoding for `LDA #1` is `10101001 00000001`, or `a9 01` in hexadecimal.

Endianness

The binary representation of assembly isn't always stored as you might expect. Consider the following 6502 assembly, which uses the `STA` instruction to store the accumulator in a given memory address:

```
LDA #1
STA $a003
LDA #9
STA $a001
```

Compare these instructions to their encodings in hexadecimal. Can you pick out the different opcodes and operands?

```
a9 01
8d 03 a0
a9 09
8d 01 a0
```

You might notice that the binary representation of the memory addresses flips the bytes of the original; it shows `01 a0` instead of `a0 01`. This is due to the endianness of the processor.

Endianness is a term taken from the book *Gulliver's Travels* by Jonathan Swift. In the book, the people of the fictional island of Lilliput disagree as to whether a hard-boiled egg should be broken from the big end or the little end. Proponents of the former are called "big-endian" and proponents of the latter "little-endian."

In computer architecture, *big-endian* systems store the *biggest*, or initial, bytes first, in the order we write them. So, the hexadecimal number `cafeba11` would be stored in byte order `ca fe ba 11`. In contrast, *little-endian* systems store the biggest bytes last, so `cafeba11` would be stored as `11 ba fe ca`.

The details of why little-endian systems exist are a bit complicated. You'll learn more about them in Chapter 5; for now, it suffices to say that 6502 is little-endian, and so is x64.

Stack Frames

The final architecture topic we'll touch on in this chapter is the *stack frame*. When you create local variables inside of a function, where do they go? Often, they'll be placed into registers, but sometimes they don't fit, or you might call some other function that needs to use those registers. In this case, they go into a special area of memory called the *stack*.

It's called a "stack" because every function in the current chain of calls gets its own little area called a *stack frame*, and operations on these frames occur in a first in, first out manner: when a function is called, a new frame is added to the stack, and when a function returns, the top frame on the stack is removed. Figure 2-5 shows a simplified view of the program stack.

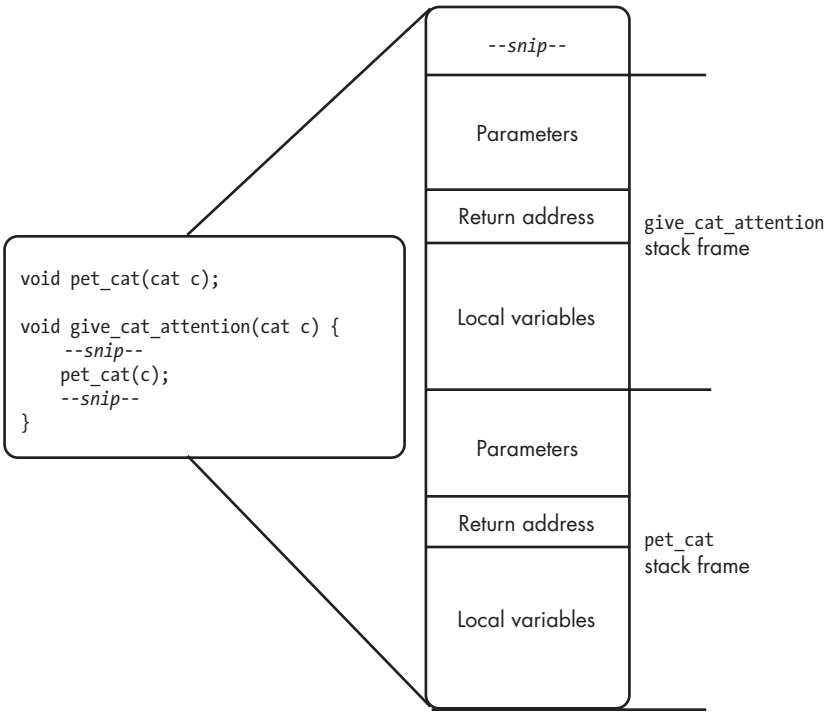


Figure 2-5: A simplified stack

A C++ program begins in the `main` function. When `main` begins, it allocates itself some memory to form its stack frame. It does this by changing

the value in a special register called the *stack pointer*. If you then call some other function (for example, `give_cat_attention`) from `main`, this new function will allocate space for its stack frame on top of the frame for `main`. If `give_cat_attention` calls `pet_cat`, then the stack frame for `pet_cat` goes on top of the one for `give_cat_attention`, and so on. This builds a big stack of local variables. On x64 systems, the stack grows downward, so the "top" of the stack is at the lowest address that's part of the stack.

When a function returns, it deallocates the memory for its stack frame by putting the stack pointer back where it was when the function began. This way, the stack for the process grows and shrinks as functions are called and return. You'll learn more about the stack in Chapters 15 and 16.

This concludes your thousand-miles-per-hour tour of compilers, operating systems, and computer architecture. Don't worry if it seems like a lot of information to absorb; you can always refer back to this chapter when you encounter an unfamiliar concept down the line.

Summary

In this chapter, you got an overview of the compilation process, how compilers output object files, and how debug information can help reverse the lossy process of compilation. You also learned how the operating system kernel can carry out privileged actions, how it uses virtual memory to protect processes from one another, and how it loads programs from disk to be executed. You were introduced to `ptrace` and Unix signals, which we'll use throughout this book, and gained a deeper understanding of how CPU instructions are represented in memory and how registers function as very small, very fast pieces of memory, sometimes with special jobs. Finally, you learned how functions are allocated stack frames to give them working memory while they execute.

Check Your Knowledge

See the appendix for sample answers to the questions in this and the following chapters.

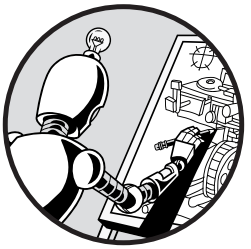
1. What is the name of the object file format used on Linux?
2. What is the name of the debug information format used on Linux?
3. The operating system kernel operates in kernel space. In what space do regular processes run?
4. What is the name for the special functions that the kernel provides so that non-privileged processes can interact with memory and the filesystem?
5. What is the name of the signal that is sent when you press CTRL-C in the console?
6. What is the name of the special register that tracks the current instruction being executed?

7. In instruction encoding, what are the names given to the operation to be carried out and the data this operation should be executed on?
8. What is the name of the tool used to translate assembly code into binary machine code?
9. How is the value `0xba5eba11` stored on little-endian machines?

3

ATTACHING TO A PROCESS

We are all of us
tethered to our guide,
finding somewhere to begin.



What is a debugger without a process to debug? In this chapter, you'll start writing your debugger by creating a program that can attach itself to other processes, either ones that are already running or ones that it launches itself. Users will be able to interact with the debugger through a simple command line interface.

You'll put into practice some of the operating system fundamentals we talked about in Chapter 2 and get your first taste of the `ptrace` system call. You'll also begin to structure your debugger in a way that you can test more easily and use as a library rather than just on the command line.

Process Interaction

Before you write any code, you need to understand the facilities that Linux provides to spawn new processes and trace their execution.

fork and exec

On Linux systems, new processes are spawned using the `fork` and `exec` syscalls. The `fork` syscall divides the running process into two separate processes that are identical save for the return value of `fork` itself; the new, or *child*, process returns `0`, whereas the original, or *parent*, process returns the *process identifier (PID)* of the child. Every process running on your system has a unique PID that you can use in many syscalls to indicate the process on which to operate.

After forking, the child and parent are free to walk their own paths in life. The child process may choose to do something completely different from the parent by executing a different program. It can do so using the `exec*` family of syscalls, which replace the currently executing program with a new one.

Because all spawned tasks follow this common algorithm, no process is born out of nothing on Linux; the links between parents and children form a sort of family tree. The topmost process belongs to whichever tool your flavor of Linux uses as its initialization system, likely `init` or `systemd`, and has the PID `1`. You can visualize this tree using tools like `ps`. Here is some of the output of `ps ax --forest` on my WSL system:

```
  PID TTY      STAT   TIME COMMAND
    1 ?        Sl     0:00 /init
12427 ?        Ss     0:00 /init
12428 ?        S      0:00  \_ /init
12429 pts/0    Ss     0:00      \_ -bash
17351 pts/0    R+     0:00          \_ ps ax --forest
16972 ?        Ss     0:00 /init
16973 ?        S      0:00  \_ /init
16974 pts/2    Ss+    0:00      \_ -bash
```

Note that `ps ax --forest` is not a process born out of the ether; it was spawned by my bash shell, which was in turn spawned by `init`, which was itself spawned by a different `init` process.

ptrace

As mentioned in Chapter 2, `ptrace` is the main debugging interface provided by Linux, so you'll spend a lot of time with it while building your fully fledged debugger. This interface gives you a myriad of tools with which to communicate with a different process. We often refer to such a process as the *inferior process*, or simply the *inferior*.

Unfortunately, as it was first shipped in 1975 with V6 Unix, `ptrace` isn't exactly a shining example of modern API design. The Unix philosophy "do one thing and do it well" seems to have skipped over this particular function, which does rather a lot of things. (Generally 36 things, although its features may vary depending on the Linux kernel version you're using.)

The ptrace interface lives in the <*sys/ptrace.h*> header and looks like this:

```
long ptrace(enum __ptrace_request request, pid_t pid,
            void *addr, void *data);
```

The request parameter indicates the action you would like to perform, which could be anything from reading memory to setting up a process to be traced or sending a SIGKILL. The pid parameter is the PID of the process you'd like to operate on. The addr and data parameters vary in meaning depending on the value you pass for request.

You can examine the tool's manual page for an exhaustive list of available commands, as there are too many to reasonably list here. I'll introduce you to certain commands when you need them, but here are a few examples to give you a taste of what is available:

PTRACE_PEEKDATA Reads 8 bytes of memory at the given address

PTRACE_ATTACH Attaches to the existing process with a given PID

PTRACE_GETREGS Retrieves the current values of all CPU registers

PTRACE_CONT Continues the execution of a process that is currently halted

The return value from ptrace depends on the request, but generally speaking, it returns -1 and sets errno when an error occurs. (This is an example of poor API design; return values whose meanings differ based on the arguments supplied are confusing and difficult to handle correctly.)

While it's tempting to ignore the return value, you should always check if the call to ptrace returned -1 and, if so, report this error to the user. This could save you hours spent tracking down a heisenbug because you forgot to check for ptrace errors. Drink some water. Get enough sleep. Check your return codes.

Launching and Attaching to Processes

With the background out of the way, let's launch a program. We'll support two ways to attach the debugger to a process:

- Launching a named program ourselves and attaching to it by running sdb *<program name>*
- Attaching to an existing process by running sdb -p *<pid>*

To keep the focus on the process interaction code, our command line argument handling will be very basic.

The main Function

Begin by writing the main function in *sdb/tools/sdb.cpp*. A common pattern we'll use in this book is assuming that some function already exists, writing code that uses it, and then implementing that function. This workflow is called *top-down programming*.

Let's assume we have a function called `attach` that launches, attaches to the given program name or PID, and returns the PID of the inferior:

```
#include <iostream>
#include <unistd.h>

❶ namespace {
    pid_t attach(int argc, const char** argv);
}

int main(int argc, const char** argv) {
 ❷ if (argc == 1) {
        std::cerr << "No arguments given\n";
        return -1;
    }

    pid_t pid = attach(argc, argv);
}
```

We'll make a habit of putting symbols inside anonymous namespaces if they're used only in the implementation file in which they're defined. This practice will avoid name collisions if we happen to reuse a name across different files. In this case, for example, we put the `attach` function in an anonymous namespace ❶. Its return type, `pid_t`, is an integral type for storing a process ID.

In `main`, we call `attach` with the command line arguments provided to `sdb`. The first command line argument of a C++ program is always the path to the running executable itself, so the user should supply at least two arguments to specify the program to launch or the PID to attach to. If they've supplied only one argument ❷, we throw an error.

The attach Function

Now write the `attach` function in *sdb.cpp*. Although it's best practice to place this code in `libsdb`, we'll first implement the basic launching and attaching functionality in a single file, then refactor it at the end of the chapter to work in a decent error-handling story.

A high-level view of the structure of `attach` looks like this:

```
#include <string_view>
#include <sys/ptrace.h>

namespace {
    pid_t attach(int argc, const char** argv) {
        pid_t pid = 0;
        // Passing PID
     ❶ if (argc == 3 && argv[1] == std::string_view("-p")) {

        }
```

```
        // Passing program name
        else {

        }

    ❷ return pid;
    }
}
```

In `attach`, we check if the first command line argument passed was `-p` ❶. Note that simply writing `argv[1] == "-p"` would have done the wrong thing, as it would have compared only the pointer values rather than the string contents. Instead, we use C++17's `std::string_view` to avoid relying on old crusty C functions or dynamically allocating memory with `std::string`. Next, we define two branches inside the function, leaving them blank for now. Then, we return `pid` ❷, whose value we'll set inside of those blocks.

Let's implement the block that attaches to an existing process. For this, we use the `PTRACE_ATTACH` request:

```
--snip--
// Passing PID
if (argc == 3 && argv[1] == std::string_view("-p")) {
    pid = std::atoi(argv[2]);
    if (pid <= 0) {
        std::cerr << "Invalid pid\n";
        return -1;
    }
❶ if (ptrace(PTRACE_ATTACH, pid, /*addr=*/nullptr, /*data=*/nullptr) < 0) {
   ❷ std::perror("Could not attach");
        return -1;
    }
}
--snip--
```

We attach to the process by passing `PTRACE_ATTACH` and the process's ID to `ptrace` ❶. As a result, Linux will allow us to send other `ptrace` requests to this process. It will also send the process a `SIGSTOP` to pause its execution. Because we're good citizens, we check if `ptrace` returned an error. Then, we pass `nullptr` as the `addr` and `data` arguments, as they're unused in the `PTRACE_ATTACH` request.

If an error occurs, `ptrace` additionally sets the `errno` variable with an error code describing what went wrong. We use `std::perror` ❷ to print this description to `stderr`, along with a string that we pass to provide more context to the user.

Next, let's implement the launch-and-attach functionality using `fork` and `exec`:

```
--snip--
// Passing program name
else {
    const char* program_path = argv[1];
 ❶ if ((pid = fork()) < 0) {
        std::perror("fork failed");
        return -1;
    }

    if (pid == 0) {
        // In child process
        // Execute debugee
    }
}
--snip--
```

We call `fork` and then ensure that no error occurred ❶. Recall that `fork` returns `0` inside the child process, so we test for this case.

If we're in the child process, we should replace the currently executing program with the program we want to debug. However, before we call `exec`, we must set the process up to be traced using the `PTRACE_TRACEME` request, which will allow us to send more `ptrace` requests to this process in the future:

```
    --snip--
    if (pid == 0) {
        if (ptrace(PTRACE_TRACEME, 0, nullptr, nullptr) < 0) {
            std::perror("Tracing failed");
            return -1;
        }
     ❶ if (execlp(program_path, program_path, nullptr) < 0) {
            std::perror("Exec failed");
            return -1;
        }
    }
    --snip--
```

After enabling process tracing, we call `execlp` ❶, which is one of the flavors of `exec` I mentioned earlier in this chapter. The `l` in `execlp` means that arguments passed to the program should be supplied individually rather than as an array. The `p` means that the facility will search the `PATH` environment variable for the given program name if the supplied path doesn't contain a forward slash (`/`). Here is the signature for `execlp`:

```
int execlp(const char *file, const char *arg, ...);
```

The ... at the end of the argument list means that this function takes a variable number of arguments. Such functions are called *varargs* functions.

After we've attached to the process, we should wait until it has paused execution before we accept any user input. Linux helps us here by stopping the process on a call to `exec` if it's being traced using `ptrace`. We wait for this stop to occur using the `waitpid` function, whose signature is as follows:

```
pid_t waitpid(pid_t pid, int *status, int options);
```

We can pass `waitpid` the PID of a child process to wait until a state change, which occurs when the child is either terminated or stopped by a signal. If the process has already changed state, the function will return immediately; otherwise, the parent process will block until a change occurs.

The `status` output parameter can give us information about the state change that happened. We can call various macros on the returned status to check its properties, such as `WIFSIGNALED(status)`, which checks if the child was terminated by a signal. Consult the manual page for `waitpid` for the complete list of macros.

The `options` parameter allows us to pass various flags to tune the wait, such as `WCONTINUED` to be notified if a `SIGCONT` resumes the child process. For now, we'll simply pass the PID and ignore the other parameters. Extend the `main` function as follows:

```
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, const char** argv) {
    if (argc == 1) {
        std::cerr << "No arguments given\n";
        return -1;
    }

    pid_t pid = attach(argc, argv);

    int wait_status;
    int options = 0;
 ❶ if (waitpid(pid, &wait_status, options) < 0) {
        std::perror("waitpid failed");
    }
}
```

We wait for the process to stop after we attach to it ❶. Now that we've attached to a process (which we may have also launched), we can start reading commands from the user.

Adding a User Interface

We want the user to interact with the debugger through a command line interface. We'll provide such an interface using `libedit`, which lets us

implement history, searching, and the kind of navigation you expect from a console-based debugger. (In fact, the LLDB debugger uses `libedit` as well.) Add basic support for `libedit` in *sdb/tools/sdb.cpp* (we'll shortly replace this code with something more complex; this is just to give you a feel for the process):

```
#include <editline/readline.h>
#include <string>

namespace {
    void handle_command(pid_t pid, std::string_view line);
}

int main(int argc, const char** argv) {
    --snip--
    char* line = nullptr;
 ❶ while ((line = readline("sdb> ")) != nullptr) {
     ❷ handle_command(pid, line);
     ❸ add_history(line);
     ❹ free(line);
    }
}
```

We loop in `main`, reading input from the user until there is nothing left to read ❶. Within the loop, the `readline` function from `libedit` takes a prompt and returns a `char*` representing the line it reads from the user. If it reads an end-of-file (EOF) marker (because the user has entered CTRL-D), it returns `nullptr`.

Next, we call a currently nonexistent function that handles the command ❷. We add the command to the searchable history using `add_history` ❸, which `libedit` provides. Finally, we clean up the memory that `readline` allocated for the line ❹; no one likes memory leaks.

Now we'll go one step further: if the user enters an empty line, we treat this as a shortcut to rerun the last command. Replace the code you just wrote with this:

```
    --snip--
    char* line = nullptr;
    while ((line = readline("sdb> ")) != nullptr) {
     ❶ std::string line_str;

     ❷ if (line == std::string_view("")) {
            free(line);
            if (history_length > 0) {
             ❸ line_str = history_list()[history_length - 1]->line;
            }
       }
```

```
    ❹ else {
          line_str = line;
          add_history(line);
          free(line);
      }

    ❺ if (!line_str.empty()) {
          handle_command(pid, line_str);
      }
  }
```

We add the `std::string` local variable ❶ for holding the command to be executed, regardless of whether it came straight from the user or from the `readline` history.

Next, we check whether the line is empty ❷. If so, we free the memory for it before trying to retrieve the last item in the `readline` history. `libedit` provides a `history_list` function for retrieving the history and a `history_length` global variable that tells us how many entries there are. Using these, we find the most recent line input by the user ❸.

If the line wasn't empty ❹, we save its contents into `line_str`, add it to our history, and free its memory. Finally, we handle the command if we received one ❺.

Handling User Input

Now that we can retrieve textual commands from the user, we need to interpret them and carry out the requested action. Our commands will follow a format similar to that of the GDB and LLDB debuggers. To continue the program, a user can enter `continue`, `cont`, or even just `c`. If they want to set a breakpoint on an address, they'll enter `break set` *`0xcafecafe`*, where *`0xcafecafe`* is the desired address in hexadecimal format.

Add support for continuing the program in *sdb/tools/sdb.cpp*:

```
#include <vector>

❶ namespace {
    std::vector<std::string> split(std::string_view str, char delimiter);
    bool is_prefix(std::string_view str, std::string_view of);
    void resume(pid_t pid);
    void wait_on_signal(pid_t pid);

  ❷ void handle_command(pid_t pid, std::string_view line) {
        auto args = split(line, ' ');
        auto command = args[0];
```

```
        if (is_prefix(command, "continue")) {
            resume(pid);
            wait_on_signal(pid);
        }
        else {
            std::cerr << "Unknown command\n";
        }
    }
}
```

We declare several functions in an anonymous namespace to carry out string handling and process manipulation tasks ❶. We'll implement these shortly. Note that we can't call the `resume` function `continue` because `continue` is a keyword in C++.

We implement a simple command handler ❷ by splitting the command on spaces in case the user provided arguments to the command. If the command is a prefix of `continue`, we continue the process and then wait for it to halt. If we don't recognize the command, we print an error message for the user.

AUTO

The command-handling function uses `auto`, a feature added in C++11 that specifies that the type of a variable should be deduced from its initializer. So, if we write `auto i = 0;`, then `i` will be of type `int`.

Importantly, `auto` doesn't deduce references. If your function returns a reference like `int& get();` and binds it to a variable like `auto i = var.get();`, then `i` will be an `int`, not an `int&`. In other words, it will be a copy of the return, not a reference to it. You need to explicitly ask for references with code like `auto& i = var.get();`.

I'll use `auto` throughout this book to make the code shorter or to save us from having to think about a variable's type if it's not very important.

Next, fill in `split` and `is_prefix`, a couple of small string manipulation helpers:

```
#include <algorithm>
#include <sstream>

namespace {
❶ std::vector<std::string> split(std::string_view str, char delimiter)  {
        std::vector<std::string> out{};
        std::stringstream ss {std::string{str}};
        std::string item;
```

```
        while (std::getline(ss, item, delimiter)) {
            out.push_back(item);
        }

        return out;
    }

❷ bool is_prefix(std::string_view str, std::string_view of) {
        if (str.size() > of.size()) return false;
        return std::equal(str.begin(), str.end(), of.begin());
    }
}
```

The `split` function ❶ uses `std::stringstream` and `std::getline` to read delimited text from the string we give it. The `std::getline` function will read a block of text from the given stream into `item` until it hits the given delimiter. We then collect all of these blocks into a `std::vector` and return it.

The `is_prefix` function ❷ is a small utility function that returns an indication of whether a string is either equal to or a prefix of another string. If you're using C++20, you can simplify this kind of string processing with ranges.

Finally, we use `ptrace` magic to make the inferior process continue:

```
namespace {
❶ void resume(pid_t pid) {
        if (ptrace(PTRACE_CONT, pid, nullptr, nullptr) < 0) {
            std::cerr << "Couldn't continue\n";
            std::exit(-1);
        }
    }

❷ void wait_on_signal(pid_t pid) {
        int wait_status;
        int options = 0;
        if (waitpid(pid, &wait_status, options) < 0) {
            std::perror("waitpid failed");
            std::exit(-1);
        }
    }
}
```

The `resume` function ❶ wraps a call to `ptrace` with the `PTRACE_CONT` request in some error handling. This request causes the operating system to resume the execution of the process. The `wait_on_signal` function ❷ similarly wraps a call to `waitpid`. Now we can evaluate the fruits of our efforts.

Manual Testing

Let's manually test the features we've just added. We'll automate these tests in the next chapter, after we've cleaned up the structure of the code.

We'll start with process launching. You should now be able to launch a process, have it stop once launched, and then resume its execution by entering `continue`. Give it a try by starting one of Linux's most useful programs, `yes`, which does nothing more than print out `y` over and over and over. Run it through *sdb* and continue it like so:

```
$ tools/sdb yes
sdb> continue
```

You should be immediately bombarded by an endless stream of `y`'s. This is good; it means the launching behavior is working.

Now we'll test process attaching. To attach the program to an existing process, you could try targeting a GUI application and noting that it halts execution when you start *sdb*. If you're using the terminal only, you *could* try running it with `yes` if you're brave enough, or you can run the following slightly less exciting test:

```
$ while sleep 5; do echo "I'm alive!"; done&
[1] 1247
$ tools/sdb -p 1247
```

The first command will print "I'm alive!" every five seconds. The ampersand at the end is important; it sends the command to the background.

The command should output the PID of the background process, which will likely be different on your machine. Pass this PID to *sdb* to attach to that process. When you do so, "I'm alive!" should stop printing. When you continue, it should print "I'm alive!" once and then return control back to the debugger because of the call to `waitpid`. Neat!

Depending on your Linux distribution, you may not be allowed to use `PTRACE_ATTACH` on processes that aren't children of *sdb*. This is due to the Yama Linux Security Module (LSM). There are two ways around this. First, you can globally allow attaching to non-child processes by setting LSM to "classic `ptrace` permissions" mode, like so:

```
$ echo 0 > /proc/sys/kernel/yama/ptrace_scope
```

You may need to run this with `sudo`. Second, you can add capabilities to *sdb* to trace non-child processes with `setcap CAP_SYS_PTRACE=+eip sdb`.

Refactoring into a Library

We've written code the quick and dirty way to get going, but now it's time to refactor the project so it will serve us better as we scale up the debugger.

As mentioned in "The Directory Structure" on page 1, we'll place most of the debugger functionality in `libsdb` and treat *sdb* as a command line driver for the library. This will make testing much easier.

As such, `ptrace` shouldn't appear in *sdb*, which shouldn't even be aware of its existence. Let's create some data structures in `libsdb` that represent the components of the system and move the `ptrace` calls into them.

Creating a Process Type

We'll need a type that represents any running process we can launch, attach to, continue, and wait on signals for. We'll call it `sdb::process`. Like many types we'll make over the course of this book, `sdb::process` represents some unique resource. We shouldn't be able to copy an `sdb::process` object, because that would mean creating an entire new process on the system.

Therefore, users of the library will need to interact with `sdb::process` through pointers. To make our lives easier, we'll use *smart pointers*, which are wrappers for pointers that automatically manage the allocated memory, rather than requiring programmers to remember when to free it. Create a new file called *sdb/include/libsdb/process.hpp* with these contents:

```
#ifndef SDB_PROCESS_HPP
#define SDB_PROCESS_HPP

#include <filesystem>
#include <memory>
#include <sys/types.h>

namespace sdb {
    class process {
    public:
     ❶ static std::unique_ptr<process> launch(std::filesystem::path path);
        static std::unique_ptr<process> attach(pid_t pid);

     ❷ void resume();
        /*?*/ wait_on_signal();
        pid_t pid() const { return pid_; }

    private:
        pid_t pid_ = 0;
    };
}

#endif
```

We declare `launch` and `attach` member functions that create `sdb::process` objects ❶. The `launch` function takes the path to the program to launch, whereas `attach` takes the PID of the existing process to attach to.

A user should be able to resume a process that is currently halted, so we declare a member for that ❷. They should also be able to wait for the inferior to be signaled. We should return some information about what signal

was received, but we'll have to do some more thinking about what type to return, so for now, I've left this as an open question.

Finally, an `sdb::process` object needs to keep track of the PID for the process it is tracking and expose this to users, so we add a data member and a member function to retrieve it.

THE STD::UNIQUE_PTR SMART POINTER

Our process type uses the `std::unique_ptr` smart pointer. As an example of its use, if we allocate an `int` and store the pointer in a `std::unique_ptr`, we don't call `delete` when we're done with it because the smart pointer handles this for us:

```
{
    std::unique_ptr<int> i (new int(42));
}
```

Memory is allocated when `new` is called and automatically freed when the `std::unique_ptr` object is destroyed.

In some cases, it's safer to use the helper function `std::make_unique<T>`, as in `auto i = std::make_unique<int>(42);`, because it avoids some tricky problems related to exceptions and evaluation order. I'll prefer this when possible. See Chapter 4 of *Effective Modern C++* by Scott Meyers (O'Reilly, 2014) or the Stack Overflow question at *https://stackoverflow.com/questions/106508/what-is-a-smart-pointer-and-when-should-i-use-one* for more details.

We must make sure that users can't construct a `process` object without going through those static member functions and that they can't accidentally copy it, so we'll disable the default constructor and copy operations:

```
namespace sdb {
    class process {
        --snip--
     ❶ process() = delete;
     ❷ process(const process&) = delete;
        process& operator=(const process&) = delete;
        --snip--
    };
}
```

We delete the default constructor ❶ to force client code to use the static members and then delete the copy constructors ❷ to disable copy and move behavior.

We should clean up the inferior process if we launched it ourselves but leave it running otherwise. Let's add a destructor and a member to track whether we should terminate the process:

```
namespace sdb {
    class process {
    public:
     ❶ ~process();
        --snip--

    private:
        pid_t pid_ = 0;
     ❷ bool terminate_on_end_ = true;
    };
}
```

We declare the destructor as a public member ❶ and add a new private data member to track termination ❷. We should also keep track of the current running state of the process. We'll add an enum for this:

```
namespace sdb {
 ❶ enum class process_state {
        stopped,
        running,
        exited,
        terminated
    };

    class process {
    public:
        --snip--
     ❷ process_state state() const { return state_; }

    private:
        pid_t pid_ = 0;
        bool terminate_on_end_ = true;
     ❸ process_state state_ = process_state::stopped;
    };
}
```

The `process_state` enum ❶ represents the various situations in which a process may find itself. We represent it using a strongly typed enum (`enum class`), which is a C++11 feature that stops enumerator values from implicitly converting to and from integers and automatically qualifies the enumerator names with the name of the enum (like `process_state::stopped`). We add a member to track the state the process is in ❸ and expose this to users ❷. Finally, we need to provide a way for the static members to construct a `process` object. We do this with a private constructor:

```
namespace sdb {
    class process {
    --snip--
```

```
    private:
        process(pid_t pid, bool terminate_on_end)
            : pid_(pid), terminate_on_end_(terminate_on_end) {}
        --snip--
    };
}
```

We make this member private so that client code must use the static `launch` and `attach` functions to construct the `process` object.

Implementing launch and attach

Now we can implement the declared members, starting with `launch` and `attach`. For the most part, we can steal the code from *sdb/tools/sdb.cpp*. Implement them in *sdb/src/process.cpp*:

```
#include <libsdb/process.hpp>
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

std::unique_ptr<sdb::process> sdb::process::launch(
      std::filesystem::path path) {
    pid_t pid;
    if ((pid = fork()) < 0) {
        // Error: fork failed ❶
    }

    if (pid == 0) {
        if (ptrace(PTRACE_TRACEME, 0, nullptr, nullptr) < 0) {
            // Error: Tracing failed
        }
        if (execlp(path.c_str(), path.c_str(), nullptr) < 0) {
            // Error: exec failed
        }
    }

    std::unique_ptr<process> proc (new process(pid, /*terminate_on_end=*/true));
    proc->wait_on_signal();

    return proc;
}

std::unique_ptr<sdb::process> sdb::process::attach(
      pid_t pid) {
    if (pid == 0) {
        // Error: Invalid PID
```

```
    }
    if (ptrace(PTRACE_ATTACH, pid, nullptr, nullptr) < 0) {
        // Error: Could not attach
    }

    std::unique_ptr<process> proc (new process(pid, /*terminate_on_end=*/false));
    proc->wait_on_signal();

    return proc;
}
```

First, `launch` carries out the `fork` and `exec` process you learned about at the start of the chapter; then it creates a new `sdb::process` object using the private constructor we just implemented and waits for the process to halt. Since we're launching the process, we pass `true` as the `terminate_on_end` argument. We'll address the comments about errors ❶ shortly.

Next, `attach` uses `PTRACE_ATTACH` to attach to the running process, then constructs the `sdb::process` and waits for the underlying process to halt. Since we're not launching the process in this case, we pass `false` as the `terminate_on_end` parameter.

You'll need to add this file to *sdb/src/CMakeLists.txt* to get it included in the build. While you're at it, you can remove the test *sdb/src/libsdb.cpp* and *sdb/include/libsdb/libsdb.hpp* files you created in Chapter 1. Replace the existing `add_library` call with this:

```
add_library(libsdb process.cpp)
```

The biggest change we made to the code ported over from *sdb/tools/sdb.cpp* was replacing the printing of error messages and the termination of the program with comments. Force-terminating the program from a library when the program may be able to recover is not good, but code comments aren't a great alternative. Unfortunately, it's time to think about errors.

Handling Errors

There are many different ways to handle errors in C++. The four main ones are ignoring them, using exceptions, using error codes, and relying on result types like `std::expected`.

The first option is unfortunately a common choice, but we can do better for this project. The others are all reasonable options. LLDB disables exceptions, so it uses custom result types and error codes. Projects written in C don't get many options, so they use error codes.

In *sdb*, we'll opt for exceptions. The debugger doesn't have the kind of memory footprint or predictability constraints that often cause projects to eschew exceptions, and exceptions will let us focus on the code's "happy path" while still being robust enough to help us diagnose errors if we make mistakes. We won't try to deal with every possible error case, but we will deal with many common ones so you don't have to spend hours diagnosing issues.

Let's begin by creating an *sdb*-specific exception type that we can use to differentiate our own errors from ones produced within the system, which we'll need to handle properly. We'll put the type in *sdb/include/libsdb/error.hpp*:

```
#ifndef SDB_ERROR_HPP
#define SDB_ERROR_HPP

#include <stdexcept>
#include <cstring>

namespace sdb {
    class error : ❶ public std::runtime_error {
    public:
        [[noreturn]]
        static void send(const std::string& what) { throw error(what); }
        [[noreturn]]
        static void send_errno(const std::string& prefix) {
            throw error(prefix + ": " + ❷ std::strerror(errno));
        }

    private:
        error(const std::string& what) : std::runtime_error(what) {}
    };
}

#endif
```

The `sdb::error` type inherits from `std::runtime_error` ❶, as it's a special kind of runtime error. We provide two ways to create one: `error::send`, which takes a message to use as the error description, and `error::send_errno`, which uses the contents of `errno` as the error description, adding the message we provide as a prefix. We declare both of these with the `[[noreturn]]` attribute, which indicates to the compiler that this function does not return control flow when it exits. This will prevent the compiler from issuing unnecessary warnings in some cases.

While `send` simply throws a new error with the given message, `send_errno` calls `std::strerror` ❷ to get a string representation of `errno`. This function is similar to `std::perror`, but it returns the message as a string rather than printing it to `stderr`.

Finally, we make a private constructor that forwards the error message on to the `std::runtime_error` constructor.

Let's put the new type into practice in the `attach` and `launch` functions in *sdb/src/process.cpp*:

```
#include <libsdb/error.hpp>

std::unique_ptr<sdb::process> sdb::process::launch(std::filesystem::path path) {
    pid_t pid;
    if ((pid = fork()) < 0) {
        error::send_errno("fork failed");
    }

    if (pid == 0) {
        if (ptrace(PTRACE_TRACEME, 0, nullptr, nullptr) < 0) {
            error::send_errno("Tracing failed");
        }
        if (execlp(path.c_str(), path.c_str(), nullptr) < 0) {
            error::send_errno("exec failed");
        }
    }

    std::unique_ptr<process> proc (new process(pid, /*terminate_on_end=*/true));
    proc->wait_on_signal();

    return proc;
}

std::unique_ptr<sdb::process> sdb::process::attach(pid_t pid) {
    if (pid == 0) {
        error::send("Invalid PID");
    }
    if (ptrace(PTRACE_ATTACH, pid, nullptr, nullptr) < 0) {
        error::send_errno("Could not attach");
    }

    std::unique_ptr<process> proc (new process(pid, /*terminate_on_end=*/false));
    proc->wait_on_signal();

    return proc;
}
```

I've replaced all the comments from the old version of the code with calls to either `error::send` or `error::send_errno`, depending on whether `errno` was set by the operation that failed.

You might still notice an issue with the error handling in `process::launch`; I briefly mentioned this when we implemented tracing. We'll fix it later, in Chapter 4. For now, let's move on to the destructor.

Destructing Processes

We can implement the destructor by calling `kill` on the child and waiting until it exits. (I wish the function were named something nicer, like `politely_ask_to_stop`, but oh well.) Here is the process-destructing code, which should go in *sdb/src/process.cpp*:

```
sdb::process::~process() {
    if (pid_ != 0) {
        int status;
        if (state_ == process_state::running) {
         ❶ kill(pid_, SIGSTOP);
            waitpid(pid_, &status, 0);
        }
     ❷ ptrace(PTRACE_DETACH, pid_, nullptr, nullptr);
        kill(pid_, SIGCONT);

        if (terminate_on_end_) {
         ❸ kill(pid_, SIGKILL);
            waitpid(pid_, &status, 0);
        }
    }
}
```

If we have a valid PID when the destructor runs, then we want to detach. For `PTRACE_DETACH` to work, the inferior must be stopped, so if it is currently running, we send it a `SIGSTOP` ❶ and wait for it to stop. We then detach from the process ❷ and let it continue. Finally, if we earlier determined that we should terminate the inferior when the managing `sdb::process` destructs, we send it a `SIGKILL` ❸ and wait for it to terminate.

Note that we don't handle any errors here or throw exceptions. Throwing exceptions from destructors is generally a no-no because if you're calling the destructor in the first place due to an exception being thrown up the stack, there's no way to throw an additional exception, so the program is terminated. We could log errors somewhere, but this will do for our purposes. You'll have to just believe in your destructor.

Next, we'll turn to the last two functions in `process`.

Resuming the Process

We want `sdb::process::resume` to force the process to resume and update its tracked running state. Implement it in *sdb/src/process.cpp*:

```
void sdb::process::resume() {
    if (ptrace(PTRACE_CONT, pid_, nullptr, nullptr) < 0) {
        error::send_errno("Could not resume");
    }
    state_ = process_state::running;
}
```

We simply issue a `PTRACE_CONT` command, check for errors, and update the state to be `running`.

Waiting on Signals

Now let's implement `wait_on_signal`. I said earlier that we should return some information about the signal that occurred. Let's make a type for this purpose in *sdb/include/libsdb/process.hpp*:

```
#include <cstdint>

namespace sdb {
    struct stop_reason {
        stop_reason(int wait_status);

     ❶ process_state reason;
     ❷ std::uint8_t info;
    };

    class process {
    public:
        --snip--
     ❸ stop_reason wait_on_signal();
        --snip--
    };
}
```

The `sdb::stop_reason` type holds the reason for a stop (whether the process exited, terminated, or just stopped) ❶ and some information about the stop (such as the return value of the exit or the signal that caused a stop or termination) ❷. This information will come from the `status` output parameter of `waitpid`, which we'll parse inside the `stop_reason` constructor. We also fill in that question mark from earlier in the return type of `wait_on_signal` with the new `stop_reason` type ❸.

Let's parse the `waitpid` status in the `stop_reason::stop_reason` implementation, found in *sdb/src/process.cpp*. We can use a series of macros to inspect the status that `waitpid` gives us:

```
sdb::stop_reason::stop_reason(int wait_status) {
    if (WIFEXITED(wait_status)) {
        reason = process_state::exited;
        info = WEXITSTATUS(wait_status);
    }
    else if (WIFSIGNALED(wait_status)) {
        reason = process_state::terminated;
        info = WTERMSIG(wait_status);
    }
```

```
    else if (WIFSTOPPED(wait_status)) {
        reason = process_state::stopped;
        info = WSTOPSIG(wait_status);
    }
}
```

First, `WIFEXITED` tells us if a given status represents an exit event; then, `WEXITSTATUS` extracts the exit code. We use `WIFSIGNALED` and `WIFSTOPPED` to figure out whether the stop was due to a termination or a stop and `WTERMSIG` and `WSTOPSIG` to extract the signal codes.

We can now use this type inside `wait_on_signal`:

```
sdb::stop_reason sdb::process::wait_on_signal() {
    int wait_status;
    int options = 0;
    if (waitpid(pid_, &wait_status, options) < 0) {
        error::send_errno("waitpid failed");
    }
    stop_reason reason(wait_status);
    state_ = reason.reason;
    return reason;
}
```

We call `waitpid`, update the state of the process based on the stop reason, and then return the reason to the caller. Now that `sdb::process` has all the `ptrace` functionality we relied on in *sdb/tools/sdb.cpp*, we can go back and substitute it in for the `ptrace` calls. As a result, `attach` becomes this:

```
#include <libsdb/process.hpp>

namespace {
    std::unique_ptr<sdb::process> attach(int argc, const char** argv) {
        // Passing PID
        if (argc == 3 && argv[1] == std::string_view("-p")) {
            pid_t pid = std::atoi(argv[2]);
         ❶ return sdb::process::attach(pid);
        }
        // Passing program name
        else {
            const char* program_path = argv[1];
         ❷ return sdb::process::launch(program_path);
        }
    }
}
```

We replace the `ptrace` calls that attached to the process with a call to `sdb::process::attach` ❶. We then do a similar replacement for the `launch` code ❷. We can get rid of the `resume` and `wait_on_signal` functions in *sdb/tools/sdb.cpp* and replace `handle_command` with this:

```
namespace {
    void handle_command(
          std::unique_ptr<sdb::process>& process,
       ❶ std::string_view line) {
        auto args = split(line, ' ');
        auto command = args[0];

        if (is_prefix(command, "continue")) {
         ❷ process->resume();
            process->wait_on_signal();
        }
        else {
            std::cerr << "Unknown command\n";
        }
    }
}
```

First, we update the signature to take a `std::unique_ptr<sdb::process>&` instead of a PID ❶. We then call our new version of `resume` that lives inside `sdb::process`. If the user issues a `continue` command, we call `resume` ❷ and `wait_on_signal` on the given process.

Even better, we can augment that call to `process->wait_on_signal()` to print out some details for the user:

```
namespace {
    void print_stop_reason(
          const sdb::process& process, sdb::stop_reason reason) {
        std::cout << "Process " << process.pid() << ' ';

        switch (reason.reason) {
        case sdb::process_state::exited:
            std::cout << "exited with status "
                      << static_cast<int>(reason.info);
            break;
        case sdb::process_state::terminated:
            std::cout << "terminated with signal "
                   ❶ <<  sigabbrev_np(reason.info);
            break;
        case sdb::process_state::stopped:
            std::cout << "stopped with signal " << sigabbrev_np(reason.info);
            break;
        }

        std::cout << std::endl;
    }

    void handle_command(
```

```
          std::unique_ptr<sdb::process>& process,
          std::string_view line) {
        --snip--
        if (is_prefix(command, "continue")) {
            process->resume();
            auto reason = process->wait_on_signal();
         ❷ print_stop_reason(*process, reason);
        }
        --snip--
    }
}
```

We introduce a function called `print_stop_reason` that, as you might expect, prints the stop reason. It starts by printing out the inferior's PID for the user, then prints out a message saying why that process stopped. If it exited, we print the exit status, and if it terminated or stopped due to a signal, we print the signal name. Fortunately, there is a function called `sigabbrev_np` ❶ that gets the signal abbreviation for a given signal code, so we use that here. If you're using a toolchain that doesn't supply the `sigabbrev_np` function, you can instead index the `sys_siglist` array, like `sys_siglist[reason.info]`. We also update the implementation of `handle_command` to call `print_stop_reason` ❷.

The last refactoring step is to update `main` to marshal the `sdb::process` around and report exceptions back to the user. We'll place the main loop of the debugger into a `main_loop` function that gets called from `main`:

```
#include <libsdb/error.hpp>

namespace {
    void main_loop(std::unique_ptr<sdb::process>& process) {
        char* line = nullptr;
        while ((line = readline("sdb> ")) != nullptr) {
            std::string line_str;

            if (line == std::string_view("")) {
                free(line);
                if (history_length > 0) {
                    line_str = history_list()[history_length - 1]->line;
                }
            }
            else {
                line_str = line;
                add_history(line);
                free(line);
            }

            if (!line_str.empty()) {
                try {
                 ❶ handle_command(process, line_str);
                }
```

```
            ❷ catch (const sdb::error& err) {
                  std::cout << err.what() << '\n';
              }
          }
      }
  }
}

int main(int argc, const char** argv) {
    if (argc == 1) {
        std::cerr << "No arguments given\n";
        return -1;
    }

    try {
     ❸ auto process = attach(argc, argv);
        main_loop(process);
    }
 ❹ catch (const sdb::error& err) {
        std::cout << err.what() << '\n';
    }
}
```

We extract the main loop in `main` into the new `main_loop` function, which takes the process as an argument. Then we pass that process through to `handle_command` ❶. If we encounter an error while handling a user command, we report the error and continue running so they can issue more commands. We achieve this with a catch handler ❷.

In `main`, we pass the process returned by `attach` ❸ to `main_loop`. If launching or attaching to the initial process fails, we just exit ❹.

I'm sure you'll agree that this code will be much easier to manipulate and test than the old version.

Summary

In this chapter, you built a library that can launch, attach to, and continue Linux processes. You also wrote a command line interface for it that exposes these facilities to the user.

In the next chapter, you'll learn how to use pipes to communicate between the debugger and the launched process, and you'll use them to implement automated tests for the code you just wrote.

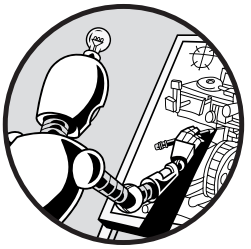
Check Your Knowledge

1. What facilities are used on Linux to launch a new process?
2. What is the name of Linux's debug API?
3. How do we wait for a child process to be signaled on Linux?

4

PIPES, PROCFS, AND AUTOMATED TESTING

I sent vibrations
across the sea
just to hear you smile.



In the last chapter, you implemented an `sdb::process` class that can launch or attach to a process on your computer. In this chapter, you'll write automated tests for this class and implement error communication between the debugger and the inferior process. You'll also learn how to use the Linux process filesystem (procfs) to retrieve information about existing processes.

Test Cases

Until now, our testing has been ad hoc and manual. This is acceptable so long as the debugger consists of two or three simple functions, but as the code gets bigger, manual testing will become infeasible very quickly. We

should create an automated test suite to ensure we don't regress any features as we make changes to the code.

Currently, `sdb::process` has four pieces of functionality: launching a process, attaching to an existing process, continuing a process, and waiting for a process to be signaled. This functionality possesses a few error cases, which we can enumerate as follows:

Launching a process

- Insufficient permissions for `PTRACE_TRACEME`
- Couldn't execute the program

Attaching to an existing process

- Invalid PID
- Insufficient permissions for `PTRACE_ATTACH`

Continuing a process

- Process already terminated

Waiting for a process to be signaled

- Process already terminated

Some of these error conditions are more difficult to test than others (for example, `PTRACE_TRACEME` will only really fail if the system is misconfigured), but we should test as many of them as we can, as well as the case where everything goes well. We'll start by testing `sdb::process::launch`.

Testing Process Launching

Let's write a test to ensure that if we launch a program with `sdb::process`, a process is created on the system. We'll use Catch2, which you set up in Chapter 1. Replace the current contents of *sdb/test/tests.cpp* with the following:

```
#include <catch2/catch_test_macros.hpp>
#include <libsdb/process.hpp>

using namespace sdb;

namespace {
 ❶ bool process_exists(pid_t pid);
}

❷ TEST_CASE("process::launch success", "[process]") {
    auto proc = process::launch("yes");
 ❸ REQUIRE(process_exists(proc->pid()));
}
```

Shortly, we'll write a function to check if a process exists, so we declare it here ❶. We define the test case that ensures that `process::launch` works ❷. We tag the test case with `[process]` so that, once we've written more tests, we can run only the test cases related to `sdb::process` with `./tests [process]`. We launch the process and require that a process with the stored PID exist ❸.

Now we need to implement `process_exists`. One way to do this on Linux is with the `kill` syscall. If you call `kill` with a signal of `0`, it doesn't send a signal to the process but still carries out the existence and permission checks it would make when actually sending a signal:

```
#include <sys/types.h>
#include <signal.h>

namespace {
    bool process_exists(pid_t pid) {
        auto ret = kill(pid, 0);
     ❶ return ret != -1 and errno != ESRCH;
    }
}
```

When `kill` fails because the process doesn't exist, it returns `-1` and sets `errno` to `ESRCH`. To check that a process exists, we return an indication of whether both of those conditions are `false` ❶. When you build and run the test, you should see that it passes.

Let's also write a test to ensure that `sdb::process` throws an `sdb::error` exception when asked to launch a nonexistent program. Catch2 comes with a handy macro for doing exactly this:

```
#include <libsdb/error.hpp>

TEST_CASE("process::launch no such program", "[process]") {
    REQUIRE_THROWS_AS(process::launch("you_do_not_have_to_be_good"), error);
}
```

The `REQUIRE_THROWS_AS` macro ensures that the given expression throws an exception of the given type. Unless you're a massive Mary Oliver fan, you probably don't have a program called `you_do_not_have_to_be_good` sitting in your `PATH` environment variable. (If you do, change the program name to something else.)

When you build and run this test, you'll notice that, contrary to our desires, the test fails because we don't encounter an exception:

```
tests is a Catch2 host application.
Run with -? for options

-------------------------------------------------------------------------------
process::launch no such program
-------------------------------------------------------------------------------
```

```
../../../test/tests.cpp:20
...............................................................................

../../../test/tests.cpp:21: FAILED:
  REQUIRE_THROWS_AS( process::launch("you_do_not_have_to_be_good"), error )
because no exception was thrown where one was expected:

===============================================================================
test cases: 2 | 1 passed | 1 failed
assertions: 2 | 1 passed | 1 failed
```

What gives? Well, we check for `execlp` failures inside `process::launch` as follows:

```
    if (pid == 0) {
        if (ptrace(PTRACE_TRACEME, 0, nullptr, nullptr) < 0) {
            error::send_errno("Tracing failed");
        }
        if (execlp(path.c_str(), path.c_str(), nullptr) < 0) {
            error::send_errno("Exec failed");
        }
    }
```

The issue, as you may have worked out, is that the child process throws the exception but doesn't send this exception to the parent. To correct the problem, let's discuss communications between the parent and child processes.

Pipes for Inter-Process Communication

C++ exceptions don't flow between processes. To communicate errors between the debugger process and the inferior process, we need to use some other mechanism, such as a pipe. In Unix, a *pipe* is a one-way communication channel between two processes whereby data fed into one end of the pipe can be read at the other end. Pipes are a form of *buffered communication*, able to retain up to 64KiB of data by default before being read from.

You can create pipes programmatically with the `pipe` syscall:

```
#include <unistd.h>

int fds[2];
pipe(fds);
```

The `pipe` function creates a pipe, placing a file descriptor for the read end of the pipe into `fds[0]` and a file descriptor for the write end of the pipe into `fds[1]`. A second function, `pipe2`, allows you to set flags on the pipe while creating it, such as to force the pipe to close if the process calls an `exec*` function. We'll use it for our inter-process communication.

Interfacing with pipes is a bit unwieldy and requires us to use older C APIs. Let's write a small wrapper around pipes to clean up the code, as we'll be using them several times throughout the book. In *sdb/include/libsdb/pipe.hpp*, create an `sdb::pipe` type:

```
#ifndef SDB_PIPE_HPP
#define SDB_PIPE_HPP

#include <vector>
#include <cstddef>

namespace sdb {
    class pipe {
    public:
     ❶ explicit pipe(bool close_on_exec);
        ~pipe();

        int get_read() const { return fds_[read_fd]; }
        int get_write() const { return fds_[write_fd]; }
        int release_read();
        int release_write();
        void close_read();
        void close_write();

     ❷ std::vector<std::byte> read();
        void write(std::byte* from, std::size_t bytes);

    private:
     ❸ static constexpr unsigned read_fd = 0;
        static constexpr unsigned write_fd = 1;
        int fds_[2];
    };
}

#endif
```

We perform a bunch of operations here, but none does anything too complicated. The `close_on_exec` parameter for the constructor ❶ tells the pipe whether it should automatically close the file descriptor if the process calls `exec`. This is desirable in cases where we expect to replace the process with another one and don't want duplicate file handles hanging around. We declare a destructor that will do any necessary cleanup and then perform operations to retrieve, release, and close the two ends of the pipe.

We also need `read` and `write` operations that operate on raw bytes ❷. The `std::byte` type added in C++17 represents a single byte. It serves our purposes better than `char`, which has several other uses in C++ and allows operations like integer arithmetic that we should avoid when interacting with raw memory.

Finally, a few private members, `read_fd` and `write_fd`, make it harder to use the wrong index into `fds_`, which holds the actual file descriptors ❸. Note that the wrapper follows a *resource acquisition is initialization (RAII)* approach, as described in the following box.

RESOURCE ACQUISITION IS INITIALIZATION

C++ programmers are great at acronyms, huh? Strange name aside, RAII means that an object acquires all of the resources and invariants it needs to function upon construction and releases all acquired resources upon destruction.

The `std::vector` type is a classic example of RAII, as it internally manages dynamic memory, which gets freed automatically when the vector is destroyed:

```
{
   std::vector<int> is {0,1,2,3};
}
```

In this example, the object acquires sufficient dynamically allocated memory to store four `int` values on construction. When it's destroyed at the end of the scope, the memory is freed automatically; we don't need to remember to call some function to clean it up.

Add *pipe.cpp* to *sdb/src/CMakeLists.txt*:

```
add_library(libsdb process.cpp pipe.cpp)
```

Now implement the members in *sdb/src/pipe.cpp*, starting with the constructor:

```
#include <unistd.h>
#include <fcntl.h>
#include <libsdb/pipe.hpp>
#include <libsdb/error.hpp>

sdb::pipe::pipe(bool close_on_exec) {
 ❶ if (pipe2(fds_, close_on_exec ? O_CLOEXEC : 0) < 0) {
        error::send_errno("Pipe creation failed");
    }
}
```

We pass `O_CLOEXEC` to `pipe2` if `close_on_exec` is `true` ❶, to ensure that the pipe gets closed when we call `execlp`. Otherwise, the process will hang while trying to read from the pipe due to the duplicated file descriptors. Of course, we also remember to check for errors.

Next, implement the destructor and `release_*` and `close_*` functions:

```
#include <utility>

sdb::pipe::~pipe() {
    close_read();
    close_write();
}

int sdb::pipe::release_read() {
 ❶ return std::exchange(fds_[read_fd], -1);
}

int sdb::pipe::release_write() {
    return std::exchange(fds_[write_fd], -1);
}

void sdb::pipe::close_read() {
    if (fds_[read_fd] != -1) {
        close(fds_[read_fd]);
        fds_[read_fd] = -1;
    }
}

void sdb::pipe::close_write() {
    if (fds_[write_fd] != -1) {
        close(fds_[write_fd]);
        fds_[write_fd] = -1;
    }
}
```

The destructor closes both the read and the write end of the pipe. The two `release_*` functions return the current file descriptor for the relevant end of the pipe and set it to some "empty" file descriptor (we use `-1` here). The `std::exchange(a, b)` ❶ convenience helper for `auto tmp = a; a = b; return tmp;` lives in the `<utility>` header. The two `close_*` functions call `close` on the relevant file descriptor so long as it's valid and set it to `-1`.

Finally, add the `read` and `write` functions to read from and write to the pipes. We'll just read a single batch of at most 1,024 bytes when the user requests a read, as this is enough for our needs:

```
std::vector<std::byte> sdb::pipe::read() {
 ❶ char buf[1024];
    int chars_read;
 ❷ if ((chars_read = ::read(fds_[read_fd], buf, sizeof(buf))) < 0) {
        error::send_errno("Could not read from pipe");
    }
```

```
    auto bytes = reinterpret_cast<std::byte*>(buf);
    return std::vector<std::byte>(bytes, bytes + chars_read);
}

❸ void sdb::pipe::write(std::byte* from, std::size_t bytes) {
    if (::write(fds_[write_fd], from, bytes) < 0) {
        error::send_errno("Could not write to pipe");
    }
}
```

Inside `sdb::pipe::read`, we declare an array of 1,024 characters ❶ to use as a buffer. We use `char` instead of `std::byte` because `read` expects this type. We call `::read` to fill the buffer from the read end of the pipe ❷, saving the number of characters that were actually read, as there are likely to be fewer than 1,024 characters in the pipe. We use `::read` with two colons at the start to ensure that we definitely call the `read` function in the global namespace rather than some other one. Finally, we fill a `std::vector<std::byte>` with the resulting data and return it.

Implementing `sdb::pipe::write` is considerably simpler; we call `::write` with the data we're given and check for errors ❸.

Let's now add a pipe to `sdb::process` for communicating errors when spawning a new process. At the start of `launch`, we create the pipe and then call `fork`. Modify *sdb/src/process.cpp* like this:

```
#include <libsdb/pipe.hpp>

std::unique_ptr<process> process::launch(std::filesystem::path path) {
 ❶ pipe channel(/*close_on_exec=*/true);
    --snip--
}
```

Note that it's important to call `pipe` before `fork`, or your pipes won't function; the pipes in the two processes will be completely distinct. Also, we'll call `execlp`, and we want the pipe to be closed when this happens so we don't leave stale file descriptors sitting around. Therefore, we pass `true` to the `sdb::pipe` constructor ❶.

If we're the child process, we call `ptrace` and `execlp`, and if we get any errors, we write them back to the parent process using the pipe:

```
namespace {
 ❶ void exit_with_perror(
        sdb::pipe& channel, std::string const& prefix) {
     ❷ auto message = prefix + ": " + std::strerror(errno);
        channel.write(
            reinterpret_cast<std::byte*>(message.data()), message.size());
        exit(-1);
    }
}
```

```
std::unique_ptr<process> process::launch(std::filesystem::path path) {
    --snip--
    if (pid == 0) {
     ❸ channel.close_read();
        if (ptrace(PTRACE_TRACEME, 0, nullptr, nullptr) < 0) {
            exit_with_perror(channel, "Tracing failed");
        }
        if (execlp(path.c_str(), path.c_str(), nullptr) < 0) {
            exit_with_perror(channel, "exec failed");
        }
    }
    --snip--
}
```

We introduce a new function here called `exit_with_perror` that writes a representation of `errno` to a pipe with a given message prefix ❶. As we did when we wrote `sdb::error`, we use `std::strerror` to get a message based on `errno` and append this to the given prefix ❷. We send this message over the pipe and exit with an error code. On Linux, the error code `-1` is idiomatic for any "something generally went wrong" error.

After we ensure we're inside the child process, we close its read end of the pipe ❸, as we won't be using it. The parent will close its write end. On the parent side, we read the other end of the pipe and throw an exception if the child wrote anything to it:

```
std::unique_ptr<process> process::launch(std::filesystem::path path) {
    --snip--
    channel.close_write();
    auto data = channel.read();
 ❶ channel.close_read();

 ❷ if (data.size() > 0) {
        waitpid(pid, nullptr, 0);
        auto chars = reinterpret_cast<char*>(data.data());
        error::send(std::string(chars, chars + data.size()));
    }

    std::unique_ptr<process> proc (
        new process(pid, /*terminate_on_end=*/true));
    proc->wait_on_signal();

    return proc;
}
```

We close the write end of the pipe, read any message that may have been sent over the read end, and close the read end ❶. Always make sure you close file descriptors as soon as you don't need them anymore. The versions of the file descriptors in the child process will close automatically because

we passed `true` as the `close_on_exec` argument when we created `channel`. If we got something over the read pipe ❷, we wait for the child process to terminate and issue an error with the given message. Now, whenever there is an error in launching the child process, the parent process will throw an exception. We use the `std::string` constructor that takes a begin pointer and end pointer to copy string data from in order to construct the argument to `error::send`.

Now that we've added pipes the the code, all tests in the current test suite should pass, as `sdb::process` correctly issues an exception if launching fails. Next, we'll test the `attach` and `resume` functions. To achieve this, we'll need a way to ask the operating system for information about existing processes.

The Linux procfs

When we tested `sdb::process::launch`, we wanted to make sure that a real process got created for the program we tried to launch. When testing `attach`, however, the process already exists; we instead want to check that we've actually attached to the program and halted its execution. Fortunately for us, Linux provides a way to do exactly this as part of the *process filesystem (procfs)*, which lets us examine the processes running on a system through files.

We'll encounter the procfs several times on our quest to write a functional debugger. At its most basic, it's a virtual filesystem located at */proc* that presents a veritable treasure trove of information about the activities taking place on the system, organized in what look like regular files on the hard drive. For our current purposes, we're interested in *process stat* files, which live in */proc/<pid>/stat* and give high-level information about the state of a given process, such as the name of the executable the process is running, the PID of its parent, the amount of time for which it has been running, and, most importantly for us, the current process execution state.

As an example, here is the state of the `init` process on my machine at the time of writing, accessible through */proc/1/stat*:

```
1 (init) S 0 0 0 0 -1 4194560 3081 315 3 66 10 39 1 0 20 0 2 0 71 1224704
201 18446744073709551615 1 1 0 0 0 0 65536 2147090174 0 0 0 0 17 0 0 0 4
0 0 0 0 0 0 0 0 0 0
```

The first value, `1`, is the PID of the process; the second value, `(init)`, is the name of the executable, enclosed in parentheses; and the third value, `S`, is the execution state. It's this third entry we care about. In a process stat file, it can be one of these options:

- `R` Running
- `S` Sleeping in an interruptible wait
- `D` Waiting in uninterruptible disk sleep
- `Z` Zombie
- `T` Stopped on a signal or, before Linux 2.6.33, trace stopped

- `t` Tracing stop (from Linux 2.6.33 onward)
- `W` Paging (before Linux 2.6.0 only)
- `X` Dead (from Linux 2.6.0 onward)
- `x` Dead (from Linux 2.6.33 to 3.13 only)
- `K` Wakekill (from Linux 2.6.33 to 3.13 only)
- `W` Waking (from Linux 2.6.33 to 3.13 only)
- `P` Parked (from Linux 3.9 to 3.13 only)

This list varies depending on the kernel version you're using. Even so, there are rather many possible states (and some mixed metaphors, given that a process can be dead, a zombie, or waking, but also parked). The one we care about is `t`, the tracing stop. Let's write a function to retrieve this status in *sdb/test/tests.cpp*:

```
#include <fstream>

namespace {
    char get_process_status(pid_t pid) {
     ❶ std::ifstream stat("/proc/" + std::to_string(pid) + "/stat");
        std::string data;
     ❷ std::getline(stat, data);
     ❸ auto index_of_last_parenthesis = data.rfind(')');
     ❹ auto index_of_status_indicator = index_of_last_parenthesis + 2;
        return data[index_of_status_indicator];
    }
}
```

We open the *stat* file for the given PID ❶ and read the line of data it contains ❷. We assume that the file exists, as we'll be using only this function in contexts where we know there's a running process with the given PID. An easy, safe way to get the status indicator we want is to find the index of the last parenthesis ❸ in the line and move two characters to the right ❹.

You might think that we could parse this file by splitting on spaces and then retrieving the third entry. However, filenames can contain spaces, so this approach could fail. You also can't perform parenthesis matching, because a filename may contain unmatched parentheses. For example, if a user named their file *pathological)*, the */proc/<pid>/stat* file would look like this:

```
11259 (pathological )) R 10381 11259 10381 34816 11260 4194304 191 0 0 0
241 404 0 0 20 0 1 0 9608606 3276800 250 18446744073709551615 93841231659008
93841231670534 140730348743840 0 0 0 0 0 0 0 0 17 6 0 0 0 0 0 93841231682592
93841231683604 93841253511168 140730348745569 140730348745586 140730348745586
140730348748775 0
```

Note the space and unmatched parenthesis. We must worry about these kinds of things when writing system tooling; users can be wild. With this support in place, we can test the `attach` function.

Testing Process Attaching

As usual, we implement the test for the `attach` function in *sdb/test/tests.cpp*:

```
TEST_CASE("process::attach success", "[process]") {
    auto pid = /* launch program without attaching */;
    auto proc = process::attach(pid);
    REQUIRE(get_process_status(pid) == 't');
}
```

We could write a function to launch some test program without attaching to it. But `process::launch` already has the code to do the launching. It would serve us better to make attaching optional in `process::launch` so we can reuse it. We'll add an option to `launch` that determines whether we should debug the launched process and track it with a new member in the object. In *sdb/include/libsdb/process.hpp*, make the following modifications:

```
namespace sdb {
    class process {
    public:
        --snip--
        static std::unique_ptr<process> launch(
            std::filesystem::path path, ❶ bool debug = true);
        --snip--

    private:
        process(pid_t pid, bool terminate_on_end, ❷ bool is_attached)
            : pid_(pid),
              terminate_on_end_(terminate_on_end),
              is_attached_(is_attached)
        {}

        --snip--
     ❸ bool is_attached_ = true;
    };
}
```

We add a `debug` parameter ❶ to `sdb::process::launch` with which a caller can specify whether to attach to the launched process. We also add an `is_attached` parameter to the private constructor ❷ and an `is_attached_` member ❸. Now *sdb/src/process.cpp* needs some supporting edits:

```
std::unique_ptr<process> process::launch(
      std::filesystem::path path, ❶ bool debug) {
    --snip--
```

```
    ❷ if (debug and ptrace(PTRACE_TRACEME, 0, nullptr, nullptr) < 0) {
            exit_with_perror(channel, "Tracing failed");
        }
    --snip--

    std::unique_ptr<process> proc (
        new process(pid, /*terminate_on_end=*/true, ❸ debug));

❹ if (debug) {
        proc->wait_on_signal();
    }

    return proc;
}
```

We update the signature of `launch` ❶ and guard the `PTRACE_TRACEME` call so it runs only if the caller requested debugging ❷. We pass the new parameter to the private constructor of `process` ❸. Finally, if we don't start tracing we shouldn't wait for a signal after forking ❹ because the process won't stop automatically after the `exec` call. We also update the private constructor call in `process::attach`:

```
std::unique_ptr<process> process::attach(pid_t pid) {
    --snip--
    std::unique_ptr<process> proc (
        new process(pid, /*terminate_on_end=*/false, ❶ /*attached=*/true));

    proc->wait_on_signal();

    return proc;
}
```

Since we're attaching to a process, we pass `true` as the `is_attached` parameter of the `process` constructor ❶.

We should check if we need to call `PTRACE_DETACH` in `~process` before doing so:

```
process::~process() {
    if (pid_ != 0) {
        int status;
     ❶ if (is_attached_) {
            if (state_ == process_state::running) {
                kill(pid_, SIGSTOP);
                waitpid(pid_, &status, 0);
            }
            ptrace(PTRACE_DETACH, pid_, nullptr, nullptr);
            kill(pid_, SIGCONT);
        }
        --snip--
```

```
    }
}
```

We don't want to detach if we're not already attached, so we check `is_attached_` first ❶. Now we can update the test:

```
TEST_CASE("process::attach success", "[process]") {
    auto target = ❶ process::launch("targets/run_endlessly", false);
    auto proc = process::attach(target->pid());
    REQUIRE(get_process_status(target->pid()) == 't');
}
```

We replace the old comment with a call to `process::launch` ❶ and ensure that we pass `false` as the second argument so we don't attach to the new process.

We'll write a small program that loops endlessly without writing anything to the console and integrate it into the test build. You can think of it as an artisanal version of `yes`. Create a new directory called *sdb/test/targets* and create a new file called *run_endlessly.cpp* in that directory with these contents:

```
int main() {
    volatile int i;
    while (true) i = 42;
}
```

Why not just run `while (true) {}`? Empty infinite loops are technically undefined behavior in C++17, and we want to avoid encountering any weird issues, so we write to a `volatile` variable, which keeps us in the safe realm of defined behavior.

Create a new *CMakeLists.txt* inside *sdb/test/targets* that creates a test executable:

```
add_executable(run_endlessly run_endlessly.cpp)
```

Then, include this subdirectory in *sdb/test/CMakeLists.txt*:

```
--snip--
add_subdirectory("targets")
```

This test should pass, letting us move on to testing the case where we pass an invalid PID:

```
TEST_CASE("process::attach invalid PID", "[process]") {
    REQUIRE_THROWS_AS(process::attach(0), error);
}
```

Attaching to PID `0` should fail, causing us to get an exception and passing the test.

We've finished testing `process::attach`. Let's move on to the last function we'll test in this chapter, `sdb::process::resume`. We can consider `wait_on_signal` to be tested already, as we've relied on it for most of the other tests.

Testing Process Resuming

To evaluate the process-resuming code, we'll test cases in which we continue both valid and invalid processes. For each of these cases, we'll test processes launched by *sdb* as well as preexisting ones. Let's begin with the successful case, in *sdb/test/tests.cpp*:

```
TEST_CASE("process::resume success", "[process]") {
 ❶ {
        auto proc = process::launch("targets/run_endlessly");
        proc->resume();
        auto status = get_process_status(proc->pid());
     ❷ auto success = status == 'R' or status == 'S';
        REQUIRE(success);
    }

    {
     ❸ auto target = process::launch("targets/run_endlessly", false);
        auto proc = process::attach(target->pid());
        proc->resume();
        auto status = get_process_status(proc->pid());
        auto success = status == 'R' or status == 'S';
        REQUIRE(success);
    }
}
```

To reuse names within the same test case while avoiding collisions, we create blocks to scope their uses. Start a new block ❶, launch the *run_endlessly* program, and then resume it. The status of this process should be either `R` (running) or `S` (sleeping) if it is waiting to be scheduled by the operating system ❷.

We also test the launching of the process without attaching to it until it's already running. The only difference is that we pass `false` as the second parameter to `process::launch` ❸ and then call `process::attach` with its PID.

To test that we get an error if we try to continue an already terminated program, let's make a small program that immediately exits in *sdb/test/targets/end_immediately.cpp*:

```
int main() {}
```

Not even a "Hello, world!" Let's add this program to *sdb/test/target/CMakeLists.txt*:

```
--snip--
add_executable(end_immediately end_immediately.cpp)
```

Then, we can write the tests:

```
TEST_CASE("process::resume already terminated", "[process]") {
    auto proc = process::launch("targets/end_immediately");
    proc->resume();
    proc->wait_on_signal();
    REQUIRE_THROWS_AS(proc->resume(), error);
}
```

This test launches the process, resumes it, and then waits for it to terminate with `wait_on_signal`. After termination, we ensure that calling `resume` throws an exception. We don't test `attach` here, because by the time we call `process::attach`, the process may have already completed. With all the tests written, we've completed the chapter.

Summary

In this chapter, you learned how to use pipes to communicate between processes. You wrote a type to manage the write and read ends of a pipe and used it to implement automated tests for the process-launching code from the previous chapter. You also learned how to retrieve information about existing processes from the Linux procfs.

In the next chapter, you'll build upon this work to extend your debugger with the ability to read from and write to CPU registers.

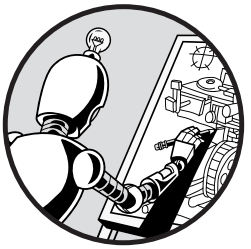
Check Your Knowledge

1. What kind of communication do pipes facilitate?
2. What is the default data pipe buffer size?
3. What does the `pipe2` function provide that the `pipe` function doesn't?
4. Which part of the Linux filesystem can be read to retrieve information about processes on your system?
5. Which specific file can be read to retrieve high-level information about a process, such as its execution state and PID?

5

REGISTERS

Tiny fragments whiz by,
carried by lightning
headed straight for you.



Registers offer us one of the best ways to inspect, poke, prod, manipulate, tinker, and generally mess around with the state of a running process. In this chapter, we'll augment the debugger by granting it the ability to read from and write to registers. This will pave the way for more advanced techniques, like reading the values of program variables, single-stepping around the code, and setting data watchpoints.

x64 Registers

The history of the x64 architecture begins in the 16-bit era of computing. It has picked up many pieces of functionality over the years and needs to maintain backward compatibility with almost all of these. Because of this,

there is a very large set of registers available, some of which have multiple names or are addressable using a variety of sizes.

In this section, we'll consider the following sets of registers: general purpose, x87, MMX, SSE, SSE2, AVX, AVX-512, and debug.

General Purpose

The x64 architecture has 16 64-bit general-purpose registers (GPRs) for things like storing integers and pointers, keeping track of the stack frame, and communicating the state of the processor.

The system's *application binary interface (ABI)* determines how these registers get used. Linux uses the System V ABI (SYSV ABI), a series of documents that describe the calling conventions for functions, the file format of object (ELF) files, how dynamic linking works, and more. It's the veritable bible of binary.

The SYSV ABI consists of a base document, which specifies the general format of object files, and architecture-specific supplements, which describe the ABIs for those systems. You can find links to these on the System V ABI page of the OSDev.org wiki (*https://wiki.osdev.org/System_V_ABI*). The SYSV ABI defines the following x64 general-purpose registers:

`rax` Caller-saved general register; used for return values

`rbx` Callee-saved general register

`rcx` Used to pass the fourth argument to functions

`rdx` Used to pass the third argument to functions

`rsp` Stack pointer

`rbp` Callee-saved general register; optionally used pointer to the top of the stack frame

`rsi` Used to pass the second argument to functions

`rdi` Used to pass the first argument to functions

`r8` Used to pass the fifth argument to functions

`r9` Used to pass the sixth argument to functions

`r10–r11` Caller-saved general registers

`r12–r15` Callee-saved general registers

Caller-saved registers can be safely written over inside a function without causing chaos. *Callee-saved* registers must be saved at the start of the function and restored at the end of the function if they're to be modified.

You can access these registers in 32-bit, 16-bit, and 8-bit mode by referencing their subregister names. For the `a`, `b`, `c`, and `d` registers, you can also access the high 8 bits of the 16-bit registers. You can find the naming schemes of these registers in Table 5-1.

Table 5-1: x64 Register Naming Scheme

Bits	First 8 naming	Second 8 naming
64	`r` prefix (for example, `rax`)	Bare (for example, `r8`)
32	`e` prefix (for example, `eax`)	`d` suffix (for example, `r8d`)
16	Bare (for example, `ax`)	`w` suffix (for example, `r8w`)
High 8	`ah`, `bh`, `ch`, and `dh` only	N/A
Low 8	Remove `x` suffix, add `l` suffix (for example, `al`)	`b` suffix (for example, `r8b`)

In addition, x64 has *segment registers* named `es`, `cs`, `ss`, `ds`, `fs`, and `gs`. For the most part, these are relics of addressing schemes used in 16-bit mode, though compilers still use the `fs` and `gs` registers to support *thread-local variables*, variables that each thread of a process has its own copy of. We can read and write these registers as 64-bit registers only.

Two other 64-bit registers to be aware of are `rip`, which stores the instruction pointer (or program counter), and `rflags`, which tracks various pieces of information about the state of the processor. For example, code could check it to see whether an arithmetic instruction overflowed or whether a logical comparison instruction returned true or false.

Floating Point and Vector

In addition to GPRs, x64 has floating-point registers (FPRs) and registers for vector operations, which execute on many pieces of data in a single instruction.

In early versions, x86 chips didn't support floating-point operations. Intel initially added them using external floating-point units (FPUs), the first of which was the Intel 8087. Since the release of i486 in 1989, most Intel processors have had built-in FPUs, but you'll still see the instructions and registers involved referred to as *x87*. They include eight 80-bit registers, named `st0` through `st7`. Today, compilers use *Streaming SIMD Extensions (SSE)* instructions instead of the x87 instructions and registers for general floating-point operations, but x87 registers are still used for `long double` support because they enable higher precision than SSE.

Introduced in 1997, MMX extended x86 with instructions that could operate on multiple pieces of data at the same time, aptly called single instruction multiple data (SIMD) instructions. MMX uses eight 64-bit registers, named `mm0` through `mm7`. However, these map to the same memory as the x87 registers, so they can't be used at the same time as the x87 floating-point operations.

Added in 1999, SSE improved on MMX by providing its own dedicated registers and supporting 32-bit floating-point operations. The set added eight 128-bit registers, named `xmm0` through `xmm7`. SSE2 later expanded this set, adding the registers `xmm8` to `xmm15`.

Continuing the development of SIMD support, Advanced Vector Extensions (AVX) and AVX-512 extended the SSE registers, adding `ymm` and `zmm` registers. These grew the existing set of `xmm` registers, first to 256 bytes and

then to 512 bytes. While all x64 processors support SSE2, you can be less sure that a given piece of hardware supports AVX and AVX-512. Therefore, we won't support them in *sdb* to avoid the extra code overhead.

Debug

Debug registers support hardware breakpoints and data watchpoints. We'll consider these in detail in Chapter 9. They give us another eight registers to worry about.

For those who haven't been counting, that's 124 registers we have to somehow describe in our project. Don't worry, though; I've got some tricks to make this easier. First, let's look at how to communicate register values to the debugger.

Interactions with ptrace

We can use several `ptrace` requests to interact with registers:

`PTRACE_GETREGS` and `PTRACE_SETREGS` For reading and writing all general-purpose registers at once

`PTRACE_GETFPREGS` and `PTRACE_SETFPREGS` For reading and writing x87, MMX, and SSE registers

`PTRACE_PEEKUSER` For reading debug registers

`PTRACE_POKEUSER` For writing debug registers or a single general-purpose register

We can't use `PTRACE_POKEUSER` for writing single floating-point registers. The reasons for this are a bit complicated; I'll discuss them in "Writing" on page 85.

`PTRACE_GETREGS` and `PTRACE_SETREGS` communicate through the `user_regs` structure, defined in the *<sys/user.h>* header. It looks like this:

```
struct user_regs_struct
{
    unsigned long long int r15;
    unsigned long long int r14;
    unsigned long long int r13;
    unsigned long long int r12;
    unsigned long long int rbp;
    unsigned long long int rbx;
    unsigned long long int r11;
    unsigned long long int r10;
    unsigned long long int r9;
    unsigned long long int r8;
    unsigned long long int rax;
    unsigned long long int rcx;
    unsigned long long int rdx;
```

```
    unsigned long long int rsi;
    unsigned long long int rdi;
    unsigned long long int orig_rax;
    unsigned long long int rip;
    unsigned long long int cs;
    unsigned long long int eflags;
    unsigned long long int rsp;
    unsigned long long int ss;
    unsigned long long int fs_base;
    unsigned long long int gs_base;
    unsigned long long int ds;
    unsigned long long int es;
    unsigned long long int fs;
    unsigned long long int gs;
};
```

Each of those members stores the value of a single 64-bit register. `PTRACE_GETFPREGS` and `PTRACE_SETFPREGS` use the `user_fpregs_struct` from the same header:

```
struct user_fpregs_struct
{
    unsigned short int        cwd;
    unsigned short int        swd;
    unsigned short int        ftw;
    unsigned short int        fop;
    unsigned long long int    rip;
    unsigned long long int    rdp;
    unsigned int              mxcsr;
    unsigned int              mxcr_mask;
 ❶ unsigned int               st_space[32];
 ❷ unsigned int               xmm_space[64];
    unsigned int              padding[24];
};
```

The `st_space` variable ❶ holds the eight `st/mm` registers, `xmm_space` ❷ holds the 16 `xmm` registers, and `padding` is unused. The other members each hold the value of a single register. `PTRACE_PEEKUSER` and `PTRACE_POKEUSER` operate on the `user` type, which contains members of both `user_regs_struct` and `user_fpregs_struct`:

```
struct user
{
 ❶ struct user_regs_struct       regs;
    int                          u_fpvalid;
 ❷ struct user_fpregs_struct     i387;
    unsigned long long int       u_tsize;
    unsigned long long int       u_dsize;
    unsigned long long int       u_ssize;
```

```
    unsigned long long int        start_code;
    unsigned long long int        start_stack;
    long long int                 signal;
    int                           reserved;
    union
    {
        struct user_regs_struct*  u_ar0;
        unsigned long long int    __u_ar0_word;
    };
    union
    {
        struct user_fpregs_struct* u_fpstate;
        unsigned long long int    __u_fpstate_word;
    };
    unsigned long long int        magic;
    char                          u_comm [32];
 ❸ unsigned long long int         u_debugreg [8];
};
```

The members for GPRs ❶ and FPRs ❷ are near the top of the structure. You don't need to worry about most of the other variables, but note the `u_debugreg` field at the bottom of the type ❸; it holds the debug registers. Now that you understand how to access the registers, you can build a way to describe them in the debugger.

Describing Registers

Our code should capture several features of each register: a unique enumerator to identify it in the code; its name; the DWARF register number it is assigned in the SYSV ABI (which will become useful when we implement DWARF expressions in Chapter 19); its size in bytes; its byte offset into the `user` struct, so we can write to and read from it; its register category, such as GPR or FPR; and a default type for the value.

We'll describe these details in an `sdb::register_info` type. Then, we'll use a global array called `sdb::g_register_infos` to store the information for every register in the system. In a new file called *sdb/include/libsdb/register_info.hpp*, enter the following:

```
#ifndef SDB_REGISTER_INFO_HPP
#define SDB_REGISTER_INFO_HPP

#include <cstdint>
#include <cstddef>
#include <string_view>
#include <sys/user.h>

namespace sdb {
 ❶ enum class register_id {
```

```
        // ?
    };

❷ enum class register_type {
        gpr, sub_gpr, fpr, dr
    };

❸ enum class register_format {
        uint, double_float, long_double, vector
    };

❹ struct register_info {
        register_id id;
        std::string_view name;
        std::int32_t dwarf_id;
        std::size_t size;
        std::size_t offset;
        register_type type;
        register_format format;
    };

❺ inline constexpr const register_info g_register_infos[] = {
        // ?
    };
}
#endif
```

The `register_id` type will give each register in the system its own unique enumerator value ❶. I've left it blank for now, as we'll have to address some complexity here shortly. The `register_type` enum specifies whether a given register is a GPR, a subregister of a GPR (like how `eax` is the 32-bit version of `rax`), an FPR, or a debug register ❷. Next, `register_format` enumerates the different ways of interpreting a register ❸. The `register_info` type collects all of the information we need about a single register ❹. Finally, `g_register_infos` will be a global array of the information for every register in the system ❺. The `inline` keyword lets us define this array in the header so we can deduce the number of registers automatically from the initializer.

There is a problem here. We want data about all 124 registers in two separate places, `register_id` and `g_register_infos`. Keeping this much data in sync can become a nightmare, as it's easy to make a mistake that we can't find at compile time without more machinery. Unless we have adequate test coverage, these issues can hide in the darkness, hitting production when we least expect it. Fortunately, tools called *X-Macros* solve exactly this problem without the need for an external code generator.

X-Macros

X-Macros allow us to maintain independent data structures whose members or operations rely on the same underlying data and must be kept in sync. The basic idea is to create a file that defines the data and wraps each element in a macro, sometimes called `X` (hence the name). We could define data about my cats in a file called *data.inc* like so:

```
X(marshmallow, ginger, 3)
X(milkshake, tortoiseshell, 3)
X(lexa, black, 6)
```

To use this data, we temporarily define the `X` macro to format the data however we want it and then include the file. For example, we could use the cat data to generate enumerators for each of these cats and generate code that prints the names of all cats known to the system:

```
❶ #define X(name, color, age) name,
  enum class cat {
   ❷ #include "data.inc"
  };
❸ #undef X

  void print_known_cats() {
      #define X(name, color, age)\
       ❹ fmt::print("{} the {} cat, age {}\n", #name, #color, age);
      #include "data.inc"
      #undef X
  }
```

To define the enum, we define `X` to be `name`, and leave the other arguments unused ❶. Note the trailing comma, which we need to produce the list of enumerators. (It's valid to have a trailing comma for the final enumerator value.) We then include the data ❷, which will populate the enum block with the macro invocations from the file. After closing the block, we undefine `X` to avoid leaking macro definitions ❸.

Inside `print_known_cats`, we define `X` to format the cat's different attributes and print them to the console ❹. Prefixing the macro parameters with `#` turns them into strings. As we did earlier, we include the data and then clean up. This macro generates the following code (minus the formatting, which I added myself):

```
enum class cat {
    marshmallow,
    milkshake,
    lexa,
};

void print_known_cats() {
    fmt::print("{} the {} cat, age {}\n", "marshmallow", "ginger", 3);
    fmt::print("{} the {} cat, age {}\n", "milkshake", "tortoiseshell", 3);
```

```
    fmt::print("{} the {} cat, age {}\n", "lexa", "black", 6);
}
```

It might be macro-based, but it's pretty nifty. Armed with X-Macros, let's define our registers.

Register Tables

We'll create an X-Macro called `DEFINE_REGISTER` and give it arguments to describe the register's name, DWARF ID, size, offset, type, and format. Create an *sdb/include/libsdb/detail/registers.inc* file for this purpose. We place the file in a *detail* subdirectory because the public headers require it but users shouldn't directly include it. Start the file with a guard against regular inclusion:

```
#ifndef DEFINE_REGISTER
#error "This file is intended for textual inclusion with the\
DEFINE_REGISTER macro defined"
#endif
```

Now, including this file without first defining the `DEFINE_REGISTER` macro should cause a compiler error. Next, let's define the GPRs, for which we'll write some helper macros. First, we need a way to get the offset of a GPR inside the `user` struct. For this, we can use the `offsetof` macro:

```
#define GPR_OFFSET(reg) (offsetof(user, regs) + offsetof(user_regs_struct, reg))
```

We give `GPR_OFFSET` a register name. It gets the offset of the `regs` member inside `user` and then adds to it the offset of the given register in the `user_regs_struct` type. Next, we can make a helper macro to define a 64-bit GPR:

```
#define DEFINE_GPR_64(name,dwarf_id) \
    DEFINE_REGISTER(name, dwarf_id, 8, GPR_OFFSET(name),\
     register_type::gpr, register_format::uint)
```

We'll give `DEFINE_GPR_64` just a register name and a DWARF register ID; it fills in `DEFINE_REGISTER` for us by setting the size to 8 bytes (64 bits), getting the offset, and specifying the register type and print format. We similarly define helpers for 32-bit, 16-bit, 8-bit (high), and 8-bit (low) subregisters:

```
#define DEFINE_GPR_32(name,super) \
    DEFINE_REGISTER(name, -1, 4, GPR_OFFSET(super),\
     register_type::sub_gpr, register_format::uint)

#define DEFINE_GPR_16(name,super) \
    DEFINE_REGISTER(name, -1, 2, GPR_OFFSET(super),\
     register_type::sub_gpr, register_format::uint)

#define DEFINE_GPR_8H(name,super) \
❶ DEFINE_REGISTER(name, -1, 1, GPR_OFFSET(super) + 1,\
     register_type::sub_gpr, register_format::uint)
```

```
#define DEFINE_GPR_8L(name,super) \
    DEFINE_REGISTER(name, -1, 1, GPR_OFFSET(super),\
     register_type::sub_gpr, register_format::uint)
```

These macros all take the name of the subregister and the name of the register that they're a part of (`super`). Subregisters don't have DWARF IDs, so we use a dummy value of `-1` in this field. In most cases, we use the offset of the superregister as the register offset. This approach is correct because x64 is little-endian, so the first byte of the register will always be stored at the lowest address. However, for the high 8-bit subregisters, we need to add 1 to this value to access the second byte of the superregister ❶.

Equipped with these helper macros, we can define the 64-bit general-purpose registers:

```
DEFINE_GPR_64(rax, 0),
DEFINE_GPR_64(rdx, 1),
DEFINE_GPR_64(rcx, 2),
DEFINE_GPR_64(rbx, 3),
DEFINE_GPR_64(rsi, 4),
DEFINE_GPR_64(rdi, 5),
DEFINE_GPR_64(rbp, 6),
DEFINE_GPR_64(rsp, 7),
DEFINE_GPR_64(r8, 8),
DEFINE_GPR_64(r9, 9),
DEFINE_GPR_64(r10, 10),
DEFINE_GPR_64(r11, 11),
DEFINE_GPR_64(r12, 12),
DEFINE_GPR_64(r13, 13),
DEFINE_GPR_64(r14, 14),
DEFINE_GPR_64(r15, 15),
DEFINE_GPR_64(rip, 16),
DEFINE_GPR_64(eflags, 49),
DEFINE_GPR_64(cs, 51),
DEFINE_GPR_64(fs, 54),
DEFINE_GPR_64(gs, 55),
DEFINE_GPR_64(ss, 52),
DEFINE_GPR_64(ds, 53),
DEFINE_GPR_64(es, 50),
```

The SYSV ABI defines the DWARF IDs for these registers. We'll also define one for `orig_rax`, which `ptrace` provides as a way to get the ID of a syscall. We'll use this value in Chapter 10. It has no DWARF ID, so we use `-1`:

```
DEFINE_GPR_64(orig_rax, -1),
```

Next, we define the subregisters in a similar fashion:

```
DEFINE_GPR_32(eax, rax), DEFINE_GPR_32(edx, rdx),
DEFINE_GPR_32(ecx, rcx), DEFINE_GPR_32(ebx, rbx),
```

```
DEFINE_GPR_32(esi, rsi), DEFINE_GPR_32(edi, rdi),
DEFINE_GPR_32(ebp, rbp), DEFINE_GPR_32(esp, rsp),
DEFINE_GPR_32(r8d, r8), DEFINE_GPR_32(r9d, r9),
DEFINE_GPR_32(r10d, r10), DEFINE_GPR_32(r11d, r11),
DEFINE_GPR_32(r12d, r12), DEFINE_GPR_32(r13d, r13),
DEFINE_GPR_32(r14d, r14), DEFINE_GPR_32(r15d, r15),

DEFINE_GPR_16(ax, rax), DEFINE_GPR_16(dx, rdx),
DEFINE_GPR_16(cx, rcx), DEFINE_GPR_16(bx, rbx),
DEFINE_GPR_16(si, rsi), DEFINE_GPR_16(di, rdi),
DEFINE_GPR_16(bp, rbp), DEFINE_GPR_16(sp, rsp),
DEFINE_GPR_16(r8w, r8), DEFINE_GPR_16(r9w, r9),
DEFINE_GPR_16(r10w, r10), DEFINE_GPR_16(r11w, r11),
DEFINE_GPR_16(r12w, r12), DEFINE_GPR_16(r13w, r13),
DEFINE_GPR_16(r14w, r14), DEFINE_GPR_16(r15w, r15),

DEFINE_GPR_8H(ah, rax), DEFINE_GPR_8H(dh, rdx),
DEFINE_GPR_8H(ch, rcx), DEFINE_GPR_8H(bh, rbx),

DEFINE_GPR_8L(al, rax), DEFINE_GPR_8L(dl, rdx),
DEFINE_GPR_8L(cl, rcx), DEFINE_GPR_8L(bl, rbx),
DEFINE_GPR_8L(sil, rsi), DEFINE_GPR_8L(dil, rdi),
DEFINE_GPR_8L(bpl, rbp), DEFINE_GPR_8L(spl, rsp),
DEFINE_GPR_8L(r8b, r8), DEFINE_GPR_8L(r9b, r9),
DEFINE_GPR_8L(r10b, r10), DEFINE_GPR_8L(r11b, r11),
DEFINE_GPR_8L(r12b, r12), DEFINE_GPR_8L(r13b, r13),
DEFINE_GPR_8L(r14b, r14), DEFINE_GPR_8L(r15b, r15),
```

This might look like a lot of code to write, but imagine how much work we've saved ourselves by using these macro helpers and X-Macros. You can try running the preprocessor on your code to witness the wall of text it spits out at you (and be happy you didn't have to type it all yourself).

Next are the FPRs. These have different sizes, so in addition to an offset helper macro, we'll write a size helper macro:

```
#define FPR_OFFSET(reg) (offsetof(user, i387) + offsetof(user_fpregs_struct, reg))
#define FPR_SIZE(reg) (sizeof(user_fpregs_struct::reg))
```

These macros look up the offsets and sizes of the given register in the `i387` member of the `user` struct. Now we create a helper macro to define the x87 control and status registers:

```
#define DEFINE_FPR(name,dwarf_id,user_name) \
    DEFINE_REGISTER(name, dwarf_id, FPR_SIZE(user_name), FPR_OFFSET(user_name),\
     register_type::fpr, register_format::uint)
```

This macro is very similar to the GPR ones, but it uses the FPR-specific macros we defined earlier and sets the register type to `register_type::fpr`. It might seem strange to use `uint` for the format of floating-point registers, but

remember that these registers don't actually contain floats, but control and status values.

Next, we create some helpers for the `st`, `mm`, and `xmm` registers:

```
#define DEFINE_FP_ST(number) \
 ❶ DEFINE_REGISTER(st ## number, (33 + number), 16,\
     (FPR_OFFSET(st_space) + number*16), register_type::fpr,\
     register_format::long_double)

#define DEFINE_FP_MM(number) \
    DEFINE_REGISTER(mm ## number, (41 + number), 8,\
     (FPR_OFFSET(st_space) + ❷ number*16), register_type::fpr,\
     register_format::vector)

#define DEFINE_FP_XMM(number) \
    DEFINE_REGISTER(xmm ## number, (17 + number), 16,\
     (FPR_OFFSET(xmm_space) + number*16), register_type::fpr,\
     register_format::vector)
```

Again, those DWARF IDs come from the SYSV ABI. The register numbers for the consecutive `st`, `mm`, and `xmm` registers are also consecutive, so we calculate them by adding the register number to the DWARF ID for the lowest register. Note that while the `st` registers are 10 bytes wide in hardware, they're stored as 16-byte values in the `user_fpregs_struct` type, so we supply 16 as their size ❶. Similarly, while the `mm` registers are 8 bytes wide, they have an additional 8 bytes of padding in `user_fpregs_struct`, so we multiply their offset by 16 rather than 8 ❷. We can now define the registers themselves:

```
DEFINE_FPR(fcw, 65, cwd),
DEFINE_FPR(fsw, 66, swd),
DEFINE_FPR(ftw, -1, ftw),
DEFINE_FPR(fop, -1, fop),
DEFINE_FPR(frip, -1, rip),
DEFINE_FPR(frdp, -1, rdp),
DEFINE_FPR(mxcsr, 64, mxcsr),
DEFINE_FPR(mxcsrmask, -1, mxcr_mask),

DEFINE_FP_ST(0), DEFINE_FP_ST(1), DEFINE_FP_ST(2), DEFINE_FP_ST(3),
DEFINE_FP_ST(4), DEFINE_FP_ST(5), DEFINE_FP_ST(6), DEFINE_FP_ST(7),

DEFINE_FP_MM(0), DEFINE_FP_MM(1), DEFINE_FP_MM(2), DEFINE_FP_MM(3),
DEFINE_FP_MM(4), DEFINE_FP_MM(5), DEFINE_FP_MM(6), DEFINE_FP_MM(7),

DEFINE_FP_XMM(0), DEFINE_FP_XMM(1), DEFINE_FP_XMM(2), DEFINE_FP_XMM(3),
DEFINE_FP_XMM(4), DEFINE_FP_XMM(5), DEFINE_FP_XMM(6), DEFINE_FP_XMM(7),
DEFINE_FP_XMM(8), DEFINE_FP_XMM(9), DEFINE_FP_XMM(10), DEFINE_FP_XMM(11),
DEFINE_FP_XMM(12), DEFINE_FP_XMM(13), DEFINE_FP_XMM(14), DEFINE_FP_XMM(15),
```

Finally, we'll define the debug registers:

```
#define DR_OFFSET(number) (offsetof(user, u_debugreg) + number * 8)
#define DEFINE_DR(number)\
    DEFINE_REGISTER(dr ## number, -1, 8, DR_OFFSET(number),\
     register_type::dr, register_format::uint)

DEFINE_DR(0), DEFINE_DR(1), DEFINE_DR(2), DEFINE_DR(3),
DEFINE_DR(4), DEFINE_DR(5), DEFINE_DR(6), DEFINE_DR(7)
```

We've defined all 124 registers, plus the fake `orig_rax` register. Now we can plug this data into *sdb/include/libsdb/register_info.hpp*:

```
namespace sdb {
 ❶ enum class register_id {
        #define DEFINE_REGISTER(name,dwarf_id,size,offset,type,format) name
        #include <libsdb/detail/registers.inc>
        #undef DEFINE_REGISTER
    };

    --snip--

 ❷ inline constexpr const register_info g_register_infos[] = {
        #define DEFINE_REGISTER(name,dwarf_id,size,offset,type,format) \
            { register_id::name, #name, dwarf_id, size, offset, type, format }
        #include <libsdb/detail/registers.inc>
        #undef DEFINE_REGISTER
    };
}
```

First, we define the `register_id` enum ❶ using just the name of the register. Later, we define the `g_register_infos` array, providing each element as an initializer list with all the members in the right order ❷. We don't need trailing commas because we already wrote commas between all of the macro invocations in *registers.inc*.

Register Interactions

Now that we've defined the registers, we must write code that enables us to read data from registers and write data to them. Let's begin by creating helper functions for looking up registers by name, DWARF ID, or register ID. Make the following modifications to *sdb/include/libsdb/register_info.hpp*:

```
#include <algorithm>
#include <libsdb/error.hpp>

namespace sdb {
    template <class F>
```

```
    const register_info& register_info_by(F f) { ❶
        auto it = std::find_if( ❷
            std::begin(g_register_infos),
            std::end(g_register_infos), f);

        if (it == std::end(g_register_infos)) ❸
            error::send("Can't find register info");

        return *it;
    }
    inline const register_info& register_info_by_id(register_id id) { ❹
        return register_info_by([id](auto& i) { return i.id == id; });
    }
    inline const register_info& register_info_by_name(std::string_view name) {
        return register_info_by([name](auto& i) { return i.name == name; });
    }
    inline const register_info& register_info_by_dwarf(std::int32_t dwarf_id) {
        return register_info_by([dwarf_id](auto& i) { return i.dwarf_id == dwarf_id; });
    }
}
```

We write a `register_info_by` helper function ❶, which takes a comparator function and uses it to find a specific register info entry. We use `std::find_if` to locate the register info that fulfills the function supplied ❷, and, if none exists, we throw an error ❸.

Using this helper function, we define three functions that find a register info entry by ID, name, or DWARF ID. These all defer to `register_info_by`, supplying it with the relevant predicate. Note the importance of the `inline` keyword here ❹. This keyword allows multiple definitions of one function to exist; the linker will throw away all but one of them. Therefore, we can define the function in the header rather than in an implementation file.

Now that we've captured information about the registers in the system, we can create a type to represent the state of registers in a given process. We'll call this simply `sdb::registers`. Create a new file called *sdb/include/libsdb/registers.hpp* with the following code:

```
#ifndef SDB_REGISTERS_HPP
#define SDB_REGISTERS_HPP

#include <sys/user.h>
#include <libsdb/register_info.hpp>

namespace sdb {
    class process;
    class registers {
    public:
     ❶ registers() = delete;
        registers(const registers&) = delete;
```

```
        registers& operator=(const registers&) = delete;

      ❷ using value = /*?*/;
        value read(const register_info& info) const;
        void write(const register_info& info, value val);

    private:
      ❸ friend process;
        registers(process& proc) : proc_(&proc) {}

      ❹ user data_;
        process* proc_;
    };
}

#endif
```

The `sdb::registers` instance should be unique, so we disallow default construction and copying ❶. We declare `read` and `write` functions that operate on some as-yet-unknown `value` type ❷. Only `sdb::process` should be able to construct an `sdb::registers` object, so we declare it as a `friend` ❸. We also hold a pointer back to our parent `sdb::process` so we can ask it to read memory for us. The `data_` member ❹ uses the `user` struct from the *<sys/user.h>* header and is where we'll store the register values.

We still have to figure out what type to use to represent register values. Here are a few options:

- Use a blob of memory, like a `std::byte[16]`.
- Write a `register_value` type that gracefully handles all the various types we could represent in the registers.
- Use `std::variant`.

The first option is too unwieldy, and the second option would require a lot of code, so we'll opt for using `std::variant`, which is a *discriminated union*, meaning it holds one of a specified set of types, along with information about the type currently stored. For example, a `std::variant<bool,int,std::string>` variable could hold either a `bool`, an `int`, or a `std::string` at any given moment. We want to represent the following types:

- 8-, 16-, 32-, and 64-bit signed and unsigned integers
- `float`, `double`, and `long double`
- `std::array<std::byte, 8>` and `std::array<std::byte, 16>`, for which we'll make `byte64` and `byte128` aliases, for brevity

The `std::array` type is a wrapper around a C-style array that behaves in a less surprising way. For example, C-style arrays very easily decay to pointers to their first elements, sometimes resulting in unwanted behavior. Create a

new file called *sdb/include/libsdb/types.hpp* and populate it with those `byte64` and `byte128` aliases:

```
#ifndef SDB_TYPES_HPP
#define SDB_TYPES_HPP

#include <array>
#include <cstddef>

namespace sdb {
    using byte64 = std::array<std::byte, 8>;
    using byte128 = std::array<std::byte, 16>;
}

#endif
```

Then, plug all those types into `registers::value`:

```
#include <variant>
#include <libsdb/types.hpp>

namespace sdb {
    class registers {
    public:
        --snip--
        using value = std::variant<
            std::uint8_t, std::uint16_t, std::uint32_t, std::uint64_t,
            std::int8_t, std::int16_t, std::int32_t, std::int64_t,
            float, double, long double,
            byte64, byte128>;
        --snip--
    };
}
```

We'll also add a couple of convenience functions for the main uses of the `read` and `write` functions in the API, namely using an ID to retrieve the register value as a specific type and writing to a register given its ID:

```
namespace sdb {
    class registers {
    public:
        --snip--
        template <class T>
     ❶ T read_by_id_as(register_id id) const {
            return std::get<T>(read(register_info_by_id(id)));
        }
     ❷ void write_by_id(register_id id, value val) {
            write(register_info_by_id(id), val);
        }
```

```
    };
}
```

To read, we first retrieve the register info that matches the ID we're given and then call `std::get` to retrieve the given type from the returned `std::variant` ❶. This will let us write code like `my_registers.read_by_id_as<std::uint64_t>(register_id::rax)` to read the contents of `rax` as a `std::uint64_t`. Writing is more simple: we just retrieve the relevant register info from the ID and then defer to `write` ❷.

Now we can write the `read` and `write` functions. We'll assume that the `sdb::process` parent will read all the registers of the process into the `data_` member when the inferior is halted (we'll implement this mass reading of registers after we're done with these two functions). Let's start with `read`.

Reading

To approach reading register data, recall that the `offset` member of an `sdb::register_info` tells us the byte offset inside the `user` struct at which the register data lives. Thus, we'll follow this algorithm: first, we get a pointer into the raw bytes of `data_`; next, we figure out where the bytes of the register data live by adding the offset; and lastly, we convert those bytes into a type specified by the `size` and `format` fields.

To get the raw bytes of an object, we can use `reinterpret_cast<std::byte*>(&object)`, and to convert those bytes into an object of some type, we can use `memcpy`. Let's make some helper functions for these tasks in a new file called *sdb/include/libsdb/bit.hpp* so we can call `from_bytes<some_type>(bytes)` and `as_bytes(object)`:

```
#ifndef SDB_BIT_HPP
#define SDB_BIT_HPP

#include <cstring>

namespace sdb {
    template <class To>
    To from_bytes(const std::byte* bytes) {
        To ret;
        std::memcpy(&ret, bytes, sizeof(To));
        return ret;
    }

 ❶ template <class From>
    std::byte* as_bytes(From& from) {
        return reinterpret_cast<std::byte*>(&from);
    }

 ❷ template <class From>
    const std::byte* as_bytes(const From& from) {
```

```
        return reinterpret_cast<const std::byte*>(&from);
    }
}

#endif
```

The `from_bytes` function takes a type as a template parameter and some bytes. It constructs an object of the given type and copies the memory from the given bytes into the new object before returning it.

The `as_bytes` function simply casts its argument into a pointer to `std::bytes`. Although this is a template, we don't need to explicitly supply a template argument when we call it because we can deduce the type from the object we pass. We provide two overloads of this function: one for `const` ❷ and one for non-`const` ❶ arguments.

Let's also add some convenience helpers to cast to `byte64` and `byte128`:

```
#include <libsdb/types.hpp>

namespace sdb {
    template <class From>
    byte128 to_byte128(From src) {
     ❶ byte128 ret{};
     ❷ std::memcpy(&ret, &src, sizeof(From));
        return ret;
    }

    template <class From>
    byte64 to_byte64(From src) {
        byte64 ret{};
        std::memcpy(&ret, &src, sizeof(From));
        return ret;
    }
}
```

These are similar to `from_bytes`, with a couple of differences. The {} in the definitions of `ret` ❶ are important, as they initialize all elements to `0` rather than leaving them uninitialized. We'll rely on this behavior elsewhere. We also copy only up to the size of `From` ❷, letting us cast types smaller than 128 or 64 bytes to these arrays and leaving the rest of the bytes as `0`.

With those helper functions in place, we can implement `read` in a new file called *sdb/src/registers.cpp*:

```
#include <libsdb/registers.hpp>
#include <libsdb/bit.hpp>

sdb::registers::value sdb::registers::read(const register_info& info) const {
 ❶ auto bytes = as_bytes(data_);
```

```
    if (info.format == register_format::uint) {
        switch (info.size) {
        case 1: return from_bytes<std::uint8_t>(bytes + info.offset);
        case 2: return from_bytes<std::uint16_t>(bytes + info.offset);
        case 4: return from_bytes<std::uint32_t>(bytes + info.offset);
        case 8: return from_bytes<std::uint64_t>(bytes + info.offset);
        default: sdb::error::send("Unexpected register size");
        }
    }
    else if (info.format == register_format::double_float) {
        return from_bytes<double>(bytes + info.offset);
    }
    else if (info.format == register_format::long_double) {
        return from_bytes<long double>(bytes + info.offset);
    }
    else if (info.format == register_format::vector and info.size == 8) {
        return from_bytes<byte64>(bytes + info.offset);
    }
    else {
        return from_bytes<byte128>(bytes + info.offset);
    }
}
```

We start by retrieving a pointer to the raw bytes of the register data ❶. The `register_info` objects have an `offset` member that will tell us where in this bunch of bytes we can find the data we need. We then check the format and size of the given register and use the stored offset to retrieve the correct data in the correct format.

Add *registers.cpp* to *sdb/src/CMakeLists.txt*:

```
add_library(libsdb ... registers.cpp)
```

Let's move on to `write`.

Writing

Writing to the registers is a bit more complicated than reading, so we'll take it one step at a time. We can begin in the same way as with `read`, by getting the raw bytes of `data_`:

```
#include <iostream>
#include <libsdb/process.hpp>

void sdb::registers::write(const register_info& info, value val) {
    auto bytes = as_bytes(data_);
```

Now we want to copy the value the user passed into `bytes + info.offset`. We could write a bunch of `if...then...else` blocks that call `std::get_if` on `val` for each of the possible types, but this would get very boring very fast.

Instead, we can use `std::visit`, which takes a function and a `std::variant` and calls the given function with the value stored in the `std::variant`.

Importantly, it preserves the type of the currently stored value, so if we passed a generic lambda to `std::visit`, the lambda would get passed a `double` if a `double` were stored, a `std::uint16_t` if a `std::uint16_t` were stored, and so on. Let's give it a try:

```
--snip--
std::visit([&](auto& v) {
 ❶ if (sizeof(v) == info.size) {
        auto val_bytes = as_bytes(v);
     ❷ std::copy(val_bytes, val_bytes + sizeof(v),
            bytes + info.offset);
    }
    else {
        std::cerr << "sdb::register::write called with "
            "mismatched register and value sizes";
        std::terminate();
    }
}, val);
```

We call `std::visit` with a generic lambda and the `std::variant`. (See the following box for more information about lambdas.) The `std::visit` function will pass the lambda the value currently stored in `val`, preserving its type, and `sizeof(v)` will give us the size of this type. We compare the size against `info.size` to make sure the user passed an object of the correct size ❶. We then get the bytes of the stored value and copy them into the correct place in the structure of register values we created ❷.

LAMBDAS

Lambdas are a C++11 feature that provides a way to write small functions with a compact syntax. These functions can also capture variables from the enclosing scope, making them a powerful way to write little comparator functions. Let's break down the lambda we just wrote:

```
[&]  (auto& v)  { ... }
```

The `[&]` is the *capture specifier*. It says that if we use any variables in the enclosing scope (`info` and `bytes` in this case), we should capture them by reference rather than by value. We could also have explicitly captured these variables by reference by writing `[&info,&bytes]` or captured them by value with `[=]` or `[info,bytes]`.

The `(auto& v)` is the parameter for the lambda. Lambda parameters are mostly the same as parameters for a regular function. However, since C++14, we can additionally write `auto` to deduce the type of the parameter from the function's arguments. Lambdas with `auto` in the parameter list are known as *generic lambdas*.

Finally comes the function body, which acts just like a regular function body.

Note that we didn't have to specify a return type for the lambda. A lambda's return type defaults to `auto`, so it is deduced from `return` statements inside the function body. We could have explicitly given this lambda a `void` return type with `[=](auto& v) -> void { ... }`.

See Chapter 6 of Scott Meyers's *Effective Modern C++* (O'Reilly, 2014) or this Stack Overflow question for more details: *https://stackoverflow.com/questions/7627098/what-is-a-lambda-expression-in-c11*.

This code updates our own records, but we also need to tell the operating system what changes we want made to the inferior's registers. Luckily, `ptrace` provides an area of memory in the same format as the `user` struct, called the *user area*, that we can write into to update a single register value. Later, we'll add a `write_user_area` function to `sdb::process` to expose this functionality, but for now, we'll just assume it exists and that it allows us to supply an offset to write to and a value to write there:

```
void sdb::registers::write(const register_info& info, value val) {
    --snip--

    proc_->write_user_area(info.offset,
        from_bytes<std::uint64_t>(bytes + info.offset));
}
```

We call `write_user_area` with the register value offset and the data we just wrote into our own `user` struct. (If you see some issues with this code relating to register sizes, well done; we'll fix them in the next section.)

For now, let's extend `sdb::process` in *sdb/include/libsdb/process.hpp* with a new `registers_` member, functions to retrieve the registers, the `write_user_area` function, and a private member to read all registers, which we'll call when the inferior halts:

```
#include <libsdb/registers.hpp>

namespace sdb {
    class process {
    public:
        --snip--
     ❶ registers& get_registers() { return *registers_; }
        const registers& get_registers() const { return *registers_; }

     ❷ void write_user_area(std::size_t offset, std::uint64_t data);

    private:
        process(pid_t pid, bool terminate_on_end, bool is_attached)
            : pid_(pid), terminate_on_end_(terminate_on_end),
```

```
            is_attached_(is_attached), ❸ registers_(new registers(*this))
        {}

        void read_all_registers();

        --snip--
    ❹ std::unique_ptr<registers> registers_;
    };
}
```

We add a `registers_` member ❹, being sure to make it a `std::unique_ptr` to the new `registers` type. We make `const` and non-`const` getter functions for this member ❶ and declare `write_user_area` ❷. We initialize the new member in the constructor ❸ and add a `read_all_registers` member, which we'll use to populate `registers_` when the process halts.

We'll use `get_registers()` inside `sdb::process` rather than directly reading the `registers_` member, as this will make refactoring easier when we come to implementing support for multithreading in Chapter 18.

Now we can use `ptrace` to implement `read_all_registers` and `write_user_area` in *sdb/src/process.cpp*:

```
void sdb::process::read_all_registers() {
    if (ptrace(PTRACE_GETREGS, pid_, nullptr, &get_registers().data_.regs) < 0) {
        error::send_errno("Could not read GPR registers");
    }
    if (ptrace(PTRACE_GETFPREGS, pid_, nullptr, &get_registers().data_.i387) < 0) {
        error::send_errno("Could not read FPR registers");
    }
    for (int i = 0; i < 8; ++i) {
        // Read debug registers
    }
}

void sdb::process::write_user_area(std::size_t offset, std::uint64_t data) {
    if (ptrace(PTRACE_POKEUSER, pid_, offset, data) < 0) {
        error::send_errno("Could not write to user area");
    }
}
```

In `read_all_registers`, we read the GPRs using `PTRACE_GETREGS` and the FPRs using `PTRACE_GETFPREGS`. In `write_user_area`, we call `PTRACE_POKEUSER` to write the given data to the user area at the given offset.

Reading debug registers is a bit more complicated, so I left this functionality as a comment in the previous listing. Let's fill it in now. To do so, we need to call `PTRACE_PEEKUSER` once for each of the eight debug registers:

```
--snip--
for (int i = 0; i < 8; ++i) {
    auto id = static_cast<int>(register_id::dr0) + i; ❶
    auto info = register_info_by_id(static_cast<register_id>(id)); ❷

    errno = 0;
    std::int64_t data = ptrace(PTRACE_PEEKUSER, pid_, info.offset, nullptr); ❸
    if (errno != 0) error::send_errno("Could not read debug register");

    get_registers().data_.u_debugreg[i] = data; ❹
}
--snip--
```

Looping over scoped enum values is a bit unwieldy, but we can do it by casting them to and from integers. We retrieve the ID for the ith debug register by adding `i` to the integral representation of the 0th debug register ❶. We then retrieve the `register_info` for that register by casting the ID we computed back to a `register_id` and calling `register_info_by_id` ❷. Now that we have the info for the desired register, we call `PTRACE_PEEKUSER` to read the data from the correct offset ❸. The `PTRACE_PEEKUSER` request sets `errno` to signal errors rather than using the return value. Finally, we write the retrieved data into the correct place in the `registers_` member ❹.

To hook this code together, we can call `read_all_registers` inside of `wait_on_signal`. Update the existing `wait_on_signal` function like this:

```
sdb::stop_reason sdb::process::wait_on_signal() {
    int wait_status;
    int options = 0;
    if (waitpid(pid_, &wait_status, options) < 0) {
        error::send_errno("waitpid failed");
    }
    stop_reason reason(wait_status);
    state_ = reason.reason;

 ❶ if (is_attached_ and state_ == process_state::stopped) {
        read_all_registers();
    }

    return reason;
}
```

After setting `state_`, we read all the registers so long as we're attached to this process and it is stopped ❶.

This covers the bulk of the implementation, but we still need to iron out some details.

Troubleshooting

If you play around with the code we've just written, you may notice three issues with our `write` function:

- The `write_user_area` function throws an exception for `ah`, `bh`, `ch`, and `dh` registers.
- The `write_user_area` function throws an exception for x87 registers.
- Users can't pass types smaller than the register data (for example, a `std::uint32_t` used to write a 64-bit register).

The first problem occurs because `PTRACE_PEEKUSER` and `PTRACE_POKEUSER` require the addresses to align to 8 bytes, meaning they should be divisible by eight with no remainder. The high 8-bit registers are offset by a byte into the superregister, so they aren't aligned. A quick way to align addresses in this way is to bitwise AND them with `~0b111`. This will set the lowest 3 bits in the address to 0, forcing it to be 8-byte-aligned. Modify the last function call in the `write` function in *sdb/src/registers.cpp* like this:

```
void sdb::registers::write(const register_info& info, value val) {
    --snip--

    auto aligned_offset = info.offset & ~0b111;
    proc_->write_user_area(aligned_offset,
        from_bytes<std::uint64_t>(bytes + aligned_offset));
}
```

The second issue occurs because `PTRACE_POKEUSER` and `PTRACE_PEEKUSER` simply don't support writing and reading from the x87 area on x64. This behavior isn't documented anywhere but is likely due to the fact that many of the registers in this struct are smaller than 64 bits, so reading and writing could cover more than one register at a time. We'll look at the actual code that causes this behavior in the Linux kernel in Chapter 10. We can deal with this problem by writing and reading all x87 registers at once. Let's add such a function to *sdb/include/libsdb/process.hpp*, along with one for writing all of the GPRs at once, as we'll need this ability later to restore registers after evaluating function calls:

```
namespace sdb {
    class process {
    public:
        --snip--
        void write_fprs(const user_fpregs_struct& fprs);
        void write_gprs(const user_regs_struct& gprs);

        --snip--
    };
}
```

Define the functions in *sdb/src/process.cpp*:

```
void sdb::process::write_fprs(const user_fpregs_struct& fprs) {
    if (ptrace(PTRACE_SETFPREGS, pid_, nullptr, &fprs) < 0) {
        error::send_errno("Could not write floating point registers");
    }
}

void sdb::process::write_gprs(const user_regs_struct& gprs) {
    if (ptrace(PTRACE_SETREGS, pid_, nullptr, &gprs) < 0) {
        error::send_errno("Could not write general purpose registers");
    }
}
```

In `write_fprs`, we use the `PTRACE_SETFPREGS` request to write all the FPRs. In `write_gprs`, we instead use `PTRACE_SETREGS`.

Then, instead of just writing to the user area in `sdb::registers::write`, we'll vary our behavior depending on whether the register is an FPR. Again, modify the end of the `write` function in *sdb/src/registers.cpp*:

```
void sdb::registers::write(const register_info& info, value val) {
    --snip--

    if (info.type == register_type::fpr) {
        proc_->write_fprs(data_.i387);
    }
    else {
        auto aligned_offset = info.offset & ~0b111;
        proc_->write_user_area(aligned_offset,
            from_bytes<std::uint64_t>(bytes + aligned_offset));
    }
}
```

If the register is an FPR, we write all the FPRs at once. Otherwise, we write the single GPR or debug register value.

The final issue, not being able to pass values smaller than the registers we're writing to, is a bit more subtle. We can't just write the bytes of the smaller type into the low bytes of the larger type. This would work for unsigned integers, but not for signed integers or floats. To see why, take a look at Table 5-2, which shows the byte representations of some values of different types.

Table 5-2: Byte Representations by Type

Type	Value	32-bit representation	64-bit representation
Unsigned integer	0x42	0x42000000	0x42000000 0x00000000
Signed integer	-0x42	0xbeffffff	0xbeffffff 0xffffffff
Floating point	42.42	0x14ae2942	0xf6285c8f 0xc2354540

For unsigned integers, we could write a 32-bit `int` into a 64-bit register by just copying over the 32 bits and writing 0s into the rest of the register. This practice is called *zero extension*. The fact that smaller unsigned integers have the same bit representation as their larger counterparts is one of the reasons systems tend to be little-endian rather than big-endian. But for signed integers, we'd need to determine whether to write 0s or 1s depending on whether the integer is positive or negative. This is called *sign extension*.

The obvious way of encoding signed integers is using one bit to represent the sign: 0 would mean positive and 1 would mean negative. This scheme has a couple of issues, though: you would need different circuitry for dealing with negative numbers, and there would be two ways of encoding the number 0. Thus, the most common way of representing signed integers in binary is a system called *two's complement*.

In two's complement, the most significant bit expresses the most negative number possible with the number of bits available, and the rest of the bits operate as usual. For example, for 8-bit signed integers, the most significant bit represents −128 rather than 128. See Table 5-3 for some examples.

Table 5-3: Byte Representations for 8-Bit Signed and Unsigned Integers

Bits	Unsigned value	Signed value
00000000	0	0
00000001	1	1
00000002	2	2
01111110	126	126
01111111	127	127
10000000	128	-128
10000001	129	-127
10000010	130	-126
11111110	254	-2
11111111	255	-1

You can change a positive integer to a negative two's complement integer by flipping all the bits and adding 1. For example, here's how you would change an 8-bit integer representing 3 into −3:

00000011 3 in binary

11111100 All bits flipped

11111101 Add 1

For floating-point numbers, positive and negative values look almost completely different in hexadecimal, and performing an extension manually would take a bit of additional math. The operation doesn't have a commonly used name like zero extension or sign extension. The SSE2 instructions call

it cvtss2sd, for "convert scalar single-precision floating-point value to scalar double-precision floating-point value," which positively flows off the tongue.

There are a variety of different floating-point representation formats, but the one most commonly used is the IEEE 754 standard. For 64-bit floats, IEEE 754 stores 1 bit for the sign, 11 bits for the *exponent* (the position of the decimal point), and 52 bits for the *mantissa* (the digits of the number).

We can get C++ to do these conversions for us, but we'll have to write either a lot of boring, repetitive code or a short amount of somewhat complex template magic. Let's try the template magic. We'll write a `widen` function template that takes the register info and the value to store and then does the relevant casting to make the given value wide enough to store in the register. Implement this in *sdb/src/registers.cpp*:

```
#include <type_traits>
#include <algorithm>

namespace {
    template <class T>
    sdb::byte128 widen(const sdb::register_info& info, T t) {
        using namespace sdb;
     ❶ if constexpr (std::is_floating_point_v<T>) {
            if (info.format == register_format::double_float)
                return to_byte128(static_cast<double>(t));
            if (info.format == register_format::long_double)
                return to_byte128(static_cast<long double>(t));
        }
     ❷ else if constexpr (std::is_signed_v<T>) {
            if (info.format == register_format::uint) {
                switch (info.size) {
                case 2: return to_byte128(static_cast<std::int16_t>(t));
                case 4: return to_byte128(static_cast<std::int32_t>(t));
                case 8: return to_byte128(static_cast<std::int64_t>(t));
                }
            }
        }

        return to_byte128(t);
    }
}
```

If the type is a floating-point type ❶, we cast it up to the widest relevant floating-point type before casting it to a `byte128`. The `if constexpr` statement is like a regular `if` statement, but it runs at compile time, and `std::is_floating_point_v` is a *type trait* that checks if the given type, `T`, is a floating-point type. If we used a regular `if` statement, the code wouldn't compile because some of the `static_cast`s would be invalid. Using `if constexpr` statements allows for certain types of invalid code inside branches whose conditions aren't met.

If the given type is a signed integer ❷, we sign-extend it to the size of the register before casting it to a `byte128`. Otherwise, the type is an unsigned integer, and we cast it to a `byte128` and return it.

We can then use this function template to widen the given value before copying it into the register data in the `write` function. Modify the call to `std::visit` like this:

```
void sdb::registers::write(const register_info& info, value val) {
    --snip--
    std::visit([&](auto& v) {
     ❶ if (sizeof(v) <= info.size) {
         ❷ auto wide = widen(info, v);
           auto val_bytes = as_bytes(wide);
           std::copy(val_bytes, val_bytes + info.size, bytes + info.offset);
        }
        else {
            std::cerr << "sdb::register::write called with mismatched"
                         "register and value sizes";
            std::terminate();
        }
    }, val);
    --snip--
}
```

We change the size check from `sizeof(v) == info.size` to `sizeof(v) <= info.size` because we can now write smaller-sized values into registers safely ❶. Then, we call `widen` ❷ on the given value. The rest of the code is the same, except we replace `val_bytes + sizeof(v)` with `val_bytes + info.size` so that the widened value gets written.

With this, we've fixed all the issues with the `write` function and can move on.

Summary

In this chapter, you learned about the many, many registers featured in x64 and added facilities to the debugger to read from them and write to them. In the next chapter, you'll test these facilities by writing some x64 assembly code. You'll then expose register reading and writing to command line users. The tests will take us down something of a rabbit hole, however, so now would be a great time to take a break.

Check Your Knowledge

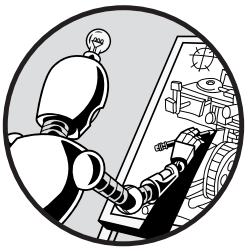
1. What are the main sets of registers that x64 offers?
2. Which register stores the program counter?
3. How do you refer to the low 32 bits of a 64-bit general-purpose register?

4. Which sets of logical registers are mapped to the same memory?
5. Where are the DWARF identifiers for registers defined?
6. What are the names of my cats?
7. The `st` registers are still used to support which type on x64?
8. Which set of registers cannot be read using `PTRACE_PEEKUSER` on x64?
9. What standard does x64 follow for its floating-point binary representation?

6

TESTING REGISTERS WITH X64 ASSEMBLY

We find ourselves
in the lowest levels
of what we wish we knew
and reach upwards to the light.



The debugger library now has facilities for modeling CPU registers and reading and writing their values. Before adding more features, we should craft automated tests for these. In this chapter, you'll learn about x64 assembly and use it to test the code you added in the previous chapter. We'll also add commands to read and write registers to the command line driver.

Why Test with Assembly?

To test register writes, we could write some registers, read them back with `ptrace`, and make sure we received the same values. But it would be better to check the registers from inside the inferior itself, to make sure we wrote

the registers in a way that actually affects the behavior of the code we're debugging.

One way to do this would be to write a program that uses `printf` to output the values of its registers. We could set a breakpoint on the `main` function, hit it, write the registers, and make sure the program prints what we expect.

Unfortunately, we don't yet have the ability to set a breakpoint on `main`. As an easy alternative, we could call `kill(getpid(), SIGTRAP)` or the synonymous `raise(SIGTRAP)` inside the debugged program to force it to stop and trap back into the debugger. This approach has one problem, though: if we call `kill` or `raise`, the program will halt while inside the code for `kill` or `raise`, rather than inside our `main` function.

We could solve this problem by issuing a syscall from inside `main` using inline assembly, but getting inline assembly to work when changing registers from outside the process is tricky. Instead, we'll write this test directly in x64 assembly. This approach requires going down a bit of a rabbit hole, but you'll emerge with knowledge of x64 assembly and a better understanding of what these registers are and how they can be used. I'll guide you through the code step by step, so you should be able to follow along even if you've never written assembly code before.

An x64 Assembly Primer

Assembly languages are human-readable programming languages that map very closely to the binary machine code of the processor that they target. Generally, a single assembly instruction maps to a single machine instruction. Assembly instructions look something like this:

```
movq    %rsp, %rbp
```

The `movq` is the *opcode*, which represents the kind of instruction we want to carry out. In this case, it's "move quadword," which is used to move around 64 bits of data at a time. The x64 processor has hundreds of different opcodes for carrying out various tasks, whereas other, smaller processors might have just a dozen. The human-readable name given to each opcode is called a *mnemonic*.

The other part of the instruction (`%rsp, %rbp`) gives the *operands*, or the data on which to operate. In this case, we're moving 64 bits of data from the `rsp` register to the `rbp` register.

Somewhat confusingly, x64 assembly language has two main competing formats: Intel and AT&T. The previous example uses AT&T syntax; the equivalent Intel instruction is `mov rbp, rsp`. Note that this syntax infers the size of the data to transfer from the register operands and flips the order of the source and destination registers. This book will use AT&T syntax, as GCC defaults to it.

In addition to instructions, assembly language has labels and directives. *Labels* give us a way to mark portions of the code with names so we can reference pieces of data or blocks of instructions elsewhere. We form them by

writing the label name followed by a colon, like `my_label:`. For example, the following code would loop forever, continually adding 1 to the `rax` register:

```
my_label:
    add $1, %rax
    jmp my_label
```

The `jmp` instruction is an unconditional jump; when control flow reaches it, the program counter will set it to the location marked by the given label (`my_label`, in this case).

Directives aren't translated into machine code, but they tell the assembler tool to carry out some action. For example, the `.global <symbol name>` directive instructs the assembler to add the given symbol to the object file's symbol table, enabling the linker to find it during linking. For more information about assembly language, see the manual for the GNU assembler.

Test Setup

Before we get started, we need a way to intercept the output of a child process so that we can communicate with our test programs. As we did in "Pipes for Inter-Process Communication" on page 54, we can use Unix pipes for this. These should allow us to replace the `stdout` of the launched process in `sdb::process::launch`. Modify the `launch` function in *sdb/include/libsdb/process.hpp*:

```
#include <optional>

--snip--
namespace sdb {
    class process {
    public:
    --snip--
        static std::unique_ptr<process> launch(std::filesystem::path path,
            bool debug = true,
            std::optional<int> stdout_replacement = std::nullopt);
    --snip--
    };
}
```

Add an optional parameter to `sdb::process::launch` with which the user can supply a file descriptor to use for `stdout`. In the implementation in *sdb/src/process.cpp*, carry out the stream replacement:

```
std::unique_ptr<process> process::launch(std::filesystem::path path,
      bool debug,
      std::optional<int> stdout_replacement) {
    --snip--
    if (pid == 0) {
        channel.close_read();
```

```
        if (stdout_replacement) {
         ❶ if (dup2(*stdout_replacement, STDOUT_FILENO) < 0) {
                exit_with_perror(channel, "stdout replacement failed");
            }
        }
        --snip--
    }
    --snip--
}
```

If the user supplied a replacement for `stdout`, we use the `dup2` syscall ❶, which closes the second file descriptor argument and then duplicates the first file descriptor argument to the second. This means that anything that goes to `stdout` now goes through whatever `*stdout_replacement` points to, which could be a file or a pipe.

STD::OPTIONAL

C++17 added `std::optional` as a way of expressing values that could be empty. It lets us construct and assign `std::optional<T>` for some `T` directly from a `T` object, so it's quite ergonomic to use. The variable `std::nullopt` represents an empty optional.

We can also convert `std::optional` to a `bool` in some contexts to indicate whether it's empty. So, writing `if (my_optional)` is the same as writing `if (my_optional.has_value())`.

To get the value of an `optional`, you can either use `*` as if it were a pointer or call `.get()`. The difference between these two approaches is that `*my_optional` triggers undefined behavior if `my_optional` is empty, whereas `my_optional.get()` will throw an exception.

Unlike many languages, C++17's `std::optional` doesn't support a monadic interface, like `map` and `flat_map` or `and_then`. C++20 added one, however. In fact, I wrote the standards paper to propose it! It's paper number P0798. If you'd like to use this interface in C++17, you can use my `tl::optional` library that lives on GitHub at *https://github.com/TartanLlama/optional*.

See Casey Carter's blog post "`std::optional`: How, When, and Why" for more details: *https://devblogs.microsoft.com/cppblog/stdoptional-how-when-and-why/*.

Next, we'll write a skeleton program in x64 assembly that we can then expand into a test program. Create an *sdb/test/targets/reg_write.s* file, then add the following to it, which is equivalent to `int main(){}` in C++:

```
.global main

.section .data

.section .text
```

```
❶ main:
      push    %rbp
      movq    %rsp, %rbp

   ❷ popq    %rbp
   ❸ movq    $0, %rax
      ret
```

First, we declare the symbol `main` so that the linker can find the `main` function when we link an executable from this code. We then declare the `.data` section, which is where initialized global data, such as strings, will go. We leave it empty for now. After the `.data` section comes the `.text` section, which is where code goes.

We begin the code with a label that declares the start of the `main` function ❶. We then move on to the *function prologue*, which carries out any necessary setup, like initializing the stack frame. The `rbp` register points to the start of the stack frame for the caller. When `main` is called, we save the old value of `rbp` so we can change the contents of this register without losing track of the caller's stack frame. The `push` instruction takes the value from a register and puts it at the top of the stack.

After we've saved the old frame base, we set up the frame for `main`. The start of its stack will be where the stack pointer currently points; `movq` means "move quadword (64 bits)" and here copies the value from `rsp` to `rbp`. We leave an empty space where the body of the `main` function will go.

Following the function body is the *function epilogue*, which carries out any necessary cleanup. Since we pushed our callee's frame base pointer onto the stack at the start of the function, we restore it to `rbp` using the `popq` (pop quadword) instruction ❷.

We want to return `0` from `main`, indicating that the function ended successfully. We do this by storing `0` in `rax` ❸ and then issuing `ret` to return to our caller. Before we fill in the function body, let's integrate this code into our build. Place the following lines in *sdb/test/targets/CMakeLists.txt*:

```
--snip--
add_executable(reg_write reg_write.s)
target_compile_options(reg_write PRIVATE -pie)
```

We add a command to create a *reg_write* executable and then add the `-pie` option to the compiler flags with `target_compile_options`. You'll learn more about this option in Chapter 7, but for now, it suffices to know that it lets the code be positioned in memory in a more flexible way.

If you configure this program, it will fail because assembly isn't enabled in our CMake project. We can fix that in *sdb/CMakeLists.txt* as follows:

```
--snip--
project ("sdb" LANGUAGES CXX ASM)
--snip--
```

Add ASM to the end of the project command. This enables assembly for the CMake project. Now we can add some code to main. First, let's work out how to force the program to halt and return control to the debugger, which is referred to as *trapping*. For this, we'll use the kill syscall.

Issuing Syscalls

To issue the kill syscall, we need to know how to invoke syscalls in x64 assembly more generally. We'll look more at the anatomy of syscalls on Linux in Chapter 10, but here is the short version of how they're invoked with the System V ABI: the syscall ID goes in rax; subsequent arguments go in rdi, rsi, rdx, r10, r8, and r9; and the return value of the syscall is stored in rax. In order to send a SIGTRAP to ourselves, we need to do the following:

1. Put the syscall for kill in rax, which is 62, given by the SYS_kill value in the <*sys/syscall.h*> header.
2. Put the PID for the running process in rdi.
3. Put the signal ID for SIGTRAP (5) in rsi.
4. Issue the syscall instruction.

The easiest way to get the PID of the running process is to use the getpid syscall, which has the syscall ID 39. Let's start by issuing this syscall and saving the PID into some register that won't be accidentally overwritten. I'll use r12 for this purpose:

```
--snip--
main:
    push    %rbp
    movq    %rsp, %rbp

    # Get PID
❶   movq    $39, %rax
    syscall
    movq    %rax, %r12
    --snip--
```

We store 39 in the rax register ❶ and then issue the syscall instruction to call getpid. The system will store the result back in rax, which we then save into r12 for future use.

Now we can perform the trapping. We issue the kill syscall with SIGTRAP as an argument:

```
main:
    --snip--
    # Trap
    movq    $62, %rax
    movq    %r12, %rdi
```

```
    movq    $5, %rsi
    syscall
    --snip--
```

The syscall ID for `kill` is `62`, which we store in `rax`. Syscalls expect the first argument to be in `rdi`, and in the case of `kill`, the argument is the PID of the process to signal. We stored the PID in `r12` earlier, so we copy the contents of `r12` to `rdi`. The second argument—the signal ID we want to send—goes in `rsi`. The signal ID for `SIGTRAP` is `5`, so we store this value in `rsi`.

We'll use these four lines of code a lot, so let's put them in a macro to clean up `main`. Move the lines we just wrote to the top of the `.text` section and surround them with the `.macro` *<macro name>* and `.endm` directives:

```
--snip--
.section .text

.macro trap
    movq $62, %rax
    movq %r12, %rdi
    movq $5, %rsi
    syscall
.endm

main:
    push    %rbp
    movq    %rsp, %rbp

    # Get PID
    movq    $39, %rax
    syscall
    movq    %rax, %r12

 ❶ trap

    popq    %rbp
    movq    $0, %rax
    ret
```

Now we can write the name of the macro as if it were an assembly instruction in the code ❶.

printf

Recall that our goal is to test the writing of registers. Let's assume that, after we exit from the trap, the debugger has written a value into the `rsi` register, which is where the second argument to `printf` would go (the argument after

the format string). We'll call `printf` to send the text of this register back to the debugger. First, we need to store the format string in the `.data` section:

```
.global main

.section .data

hex_format: .asciz "%#x"
--snip--
```

The `.asciz` directive encodes the string you give it into ASCII with a null terminator. We label it `hex_format` so we can reference it later in the code. The value `%#x` is the `printf` format string for hexadecimal integers.

Now we can call `printf`, followed by `fflush` to ensure the output goes straight to the debugger without being buffered. Add this code after the call to `trap`:

```
    --snip--
    # Print contents of rsi
❶ leaq    hex_format(%rip), %rdi
❷ movq    $0, %rax
❸ call    printf@plt
❹ movq    $0, %rdi
    call    fflush@plt
    trap
    --snip--
```

We use `leaq`, which stands for "load effective address quadword," to put the address of `hex_format` into the `rdi` register, where `printf` expects it ❶. The *`label`*`(%rip)` syntax uses *RIP-relative addressing*, a requirement when we pass that `-pie` flag.

The `printf` function is a varargs function. It is declared like `int printf (const char *format, ...);`, where `...` signifies "any number of arguments." With the SYSV ABI, varargs functions expect `rax` to contain the number of vector registers that are involved in passing arguments. In our case this is `0`, so we store `0` in `rax` ❷.

Next, we call `printf` ❸. The `@plt` part here references the *procedure linkage table (PLT)*, which is used to call functions defined in shared libraries. We'll find out more about this in Chapter 17.

Finally, we call `fflush`, passing `0` to in the `rdi` register to make it flush all streams ❹.

Now we can write our test.

General-Purpose Registers

Let's write a test to ensure that writing to a general-purpose register works. Add the following to *sdb/test/tests.cpp*:

```
#include <libsdb/pipe.hpp>
#include <libsdb/bit.hpp>

TEST_CASE("Write register works", "[register]") {
    bool close_on_exec = false;
 ❶ sdb::pipe channel(close_on_exec);

 ❷ auto proc = process::launch(
        "targets/reg_write", true, channel.get_write());
    channel.close_write();

 ❸ proc->resume();
    proc->wait_on_signal();

    auto& regs = proc->get_registers();
 ❹ regs.write_by_id(register_id::rsi, 0xcafecafe);

 ❺ proc->resume();
    proc->wait_on_signal();

    auto output = channel.read();
 ❻ REQUIRE(to_string_view(output) == "0xcafecafe");
}
```

We need to read the output from the test, so set up a pipe ❶. Launch *reg_write* ❷, resume it until the first trap ❸, and then write a value to `rsi` ❹. Resume again until the next trap ❺, by which point the process should have written the value you just put in `rsi` to `stdout`. Read this value and ensure it matches what you wrote ❻. The `to_string_view` function we use to do so doesn't exist yet, but we'll write it now. Add these helpers to convert `std::byte` sequences into `std::string_views` to *sdb/include/libsdb/bit.hpp*:

```
#include <vector>
#include <string_view>

namespace sdb {
    inline std::string_view to_string_view(
          const std::byte* data, std::size_t size) {
        return { reinterpret_cast<const char*>(data), size };
    }

    inline std::string_view to_string_view(
          const std::vector<std::byte>& data) {
        return to_string_view(data.data(), data.size());
    }
}
```

These construct `std::string_views` out of either a pointer to `std::bytes` and a size or a `std::vector<std::byte>`.

Now, if you run the test, it should pass! However, we have a bunch of other register types and sizes. We won't exhaustively test them, but we should at least try to test one x87, MMX, and SSE register. The x87 test is a bit tricky, so let's do MMX next.

MMX

For MMX, we'll write into `mm0` from the debugger, then move this value into `rsi` on the assembly side and call `printf` with the same format specifier as before. Add the following after the last call to `trap`:

```
--snip--
# Print contents of mm0
movq    %mm0, %rsi
leaq    hex_format(%rip), %rdi
movq    $0, %rax
call    printf@plt
movq    $0, %rdi
call    fflush@plt
trap
--snip--
```

The code is the same as the GPR version, but it copies `mm0` to `rsi` first. Add the following test code to the end of the same test case:

```
--snip--
regs.write_by_id(register_id::mm0, 0xba5eba11);

proc->resume();
proc->wait_on_signal();

output = channel.read();
REQUIRE(to_string_view(output) == "0xba5eba11");
```

Again, the test is the same as the GPR version, except it sets `mm0` instead of `rsi`. Let's move on to SSE.

SSE

SSE's `xmm` registers are used to perform floating-point operations, so we can write a floating-point number into one and format it using `printf`. We'll use the format specifier `%.2f`, which prints out the number with two decimal places. Add this specifier to the `.data` section:

```
--snip--
.section .data
```

```
hex_format:         .asciz "%#x"
float_format:       .asciz "%.2f"
--snip--
```

Then, make the `printf` call in the function body, after the last `trap`:

```
    --snip--
    # Print contents of xmm0
    leaq    float_format(%rip), %rdi
❶   movq    $1, %rax
    call    printf@plt
    movq    $0, %rdi
    call    fflush@plt
    trap
```

Following the approach taken in the other assembly examples, we load the float format specifier into `rdi`. Since the `xmm` registers are vector registers, however, the approach we need to take is a little different than before. According to the SYSV ABI, instead of moving the value to print into `rsi`, we keep it in `xmm0` and write `1` into `rax` to tell `printf` that there is a vector argument ❶. Then, we can call `printf` and `fflush` like before.

Here is the corresponding test code to add at the end of the test case:

```
    --snip--
    regs.write_by_id(register_id::xmm0, 42.24);

    proc->resume();
    proc->wait_on_signal();

    output = channel.read();
    REQUIRE(to_string_view(output) == "42.24");
```

Write a value into `xmm0`, resume to the next trap, and ensure that the value prints. Finally, we'll test the x87 registers.

x87

In Chapter 5, I mentioned that x87 registers provide `long double` support, so we can make use of them by getting `printf` to format a `long double` value. Unfortunately, this isn't as easy as writing a register and calling `printf`.

The x87 instructions operate on a stack of values within the `st0` through `st7` registers. I'll call this the *FPU stack* (floating-point unit stack) to distinguish it from the regular stack region of memory. A register indicates the current top of the FPU stack and provides instructions for manipulating it, such as `fld` and `fstp` for pushing and popping values from the top. Other instructions carry out arithmetic operations, like `faddp`, which pops the top two values from the FPU stack, adds them, and pushes the result.

To make things more complicated, the SYSV ABI says that `long double` arguments must be passed on the function's stack frame rather than using registers, and you can't just use `mov` to transfer a value out of an `st` register.

We'll do the following: manually push a value to the FPU stack from the debugger, call `fstp` in our assembly code to pop the value from the FPU stack onto our function stack, and then call `printf` with the `long double` specifier.

First, we add the new specifier to the `.data` section:

```
--snip--
.section .data

hex_format:         .asciz "%#x"
float_format:       .asciz "%.2f"
❶ long_float_format:  .asciz "%.2Lf"
```

The `Lf` in the format specifier stands for "long float" ❶. Then, we add code to pop a value from the FPU stack and call `printf` after the last `trap` call in the `main` function. We'll also need to allocate some space on the function stack to store the data:

```
    # Print contents of st0
  ❶ subq    $16, %rsp
  ❷ fstpt   (%rsp)
    leaq    long_float_format(%rip), %rdi
    movq    $0, %rax
    call    printf@plt
    movq    $0, %rdi
    call    fflush@plt
  ❸ addq    $16, %rsp
    trap
```

We start by allocating 16 bytes on the stack to store the contents of `st0` ❶. The stack on x64 grows downward, so we do this by subtracting from the `rsp` register. We pop `st0` from the top of the FPU stack with the `fstpt` instruction ❷ and then follow the same pattern we've used for the previous tests. After calling `fflush`, we clean up the space we allocated on the stack by incrementing the stack pointer to its original position ❸.

To push a value to the FPU stack from the debugger, we first need to understand how the stack is represented. There are three registers involved: `st0` (where we want to store the number), `fsw` (the FPU status word), and `ftw` (the FPU tag word).

The `st0` register is simple enough; we just write our float data to it. Add this to the bottom of the test:

```
regs.write_by_id(register_id::st0, 42.24l);
```

Note the `l` suffix to the floating-point number that makes it a `long double`.

The status word tracks the current size of the FPU stack and reports errors like stack overflows, precision errors, or divisions by zero. It's 16 bits

wide, and bits 11 through 13 track the top of the stack. The top of the stack starts at index 0 and actually goes down instead of up, wrapping around up to 7. So, to push our value to the stack, we set bits 11 through 13 to 7 (`0b111`):

```
regs.write_by_id(register_id::fsw,
    std::uint16_t{0b0011100000000000});
```

Finally, the 16-bit tag register tracks which of the `st` registers are valid, empty, or special (meaning they contain NaNs or infinity). A tag of `0b11` means empty, `0b00` means valid. So, we want to set the first tag to `0b00` and the rest to `0b11`:

```
regs.write_by_id(register_id::ftw,
    std::uint16_t{0b0011111111111111});
```

Then we can finish the test case:

```
TEST_CASE("Write register works", "[register]") {
    --snip--
    proc->resume();
    proc->wait_on_signal();

    output = channel.read();
    REQUIRE(to_string_view(output) == "42.24");
}
```

With that, our test case is done and should pass. This might seem like a lot of work for a test, but I hope it's given you deeper insight into the various subparts of x64.

We can also test reads in a similar way. Now that we know what we're doing, it should be much faster.

Register Reads

To test the debugger's ability to read from registers, we'll start with the same structure we did with *reg_write.s*. Create this at *sdb/test/targets/reg_read.s*:

```
.global main

.section .data

.section .text

.macro trap
    movq $62, %rax
    movq %r12, %rdi
    movq $5, %rsi
    syscall
.endm
```

```
main:
    push    %rbp
    movq    %rsp, %rbp

    # Get PID
    movq    $39, %rax
    syscall
    movq    %rax, %r12

    popq    %rbp
    movq    $0, %rax
    ret
```

This is exactly the same as what we started with for the previous test case, except we've removed the `trap` call after we get the PID for the process since we're not going to need to wait on the debugger at that point. Add a directive to store some floating-point data in the `.data` section, as we'll need this to test x87 and MMX registers:

```
--snip--
.section .data
my_double: .double 64.125
--snip--
```

Then, add the following assembly after the the `movq %rax, %r12` instruction to test GPR, MMX, SSE, and x87 registers. We don't need to print any data to a pipe; we can just write to the registers and read those registers from the debugger:

```
--snip--
    # Store to r13
 ❶ movq    $0xcafecafe, %r13
    trap

    # Store to r13b
 ❷ movb    $42, %r13b
    trap

    # Store to mm0
 ❸ movq    $0xba5eba11, %r13
    movq    %r13, %mm0
    trap

    # Store to xmm0
 ❹ movsd   my_double(%rip), %xmm0
    trap
```

```
    # Store to st0
❺ emms
❻ fldl     my_double(%rip)
    trap
--snip--
```

To test GPRs, we write a value into `r13` and then trap so we can read it from the debugger ❶. We do the same with `r13b` to test subregisters ❷.

To test MMX registers, we need to do a bit more work. We can't move directly from a 64-bit immediate to an MMX register, so we first store the value to `r13`, then transfer it to `mm0` ❸. For XMM, we copy the floating-point number we embedded in the binary to `xmm0` ❹.

Finally, for the x87 registers, we first need to issue the `emms` instruction ❺ (which stands for "empty MMX technology state") to clear the MMX state, as MMX and x87 share registers. We then use the `fldl` instruction ("load floating-point value") to push a floating-point number to the FPU stack ❻.

Add code to build this test to the *sdb/test/targets/CMakeLists.txt* file:

```
--snip--
add_executable(reg_read reg_read.s)
target_compile_options(reg_read PRIVATE -pie)
```

Then add a test to *sdb/test/tests.cpp* to ensure that we can read all of the values we've written into the registers:

```
TEST_CASE("Read register works", "[register]") {
    auto proc = process::launch("targets/reg_read");
    auto& regs = proc->get_registers();

    proc->resume();
    proc->wait_on_signal();

❶ REQUIRE(regs.read_by_id_as<std::uint64_t>(register_id::r13) ==
        0xcafecafe);

    proc->resume();
    proc->wait_on_signal();

❷ REQUIRE(regs.read_by_id_as<std::uint8_t>(register_id::r13b) == 42);

    proc->resume();
    proc->wait_on_signal();

❸ REQUIRE(regs.read_by_id_as<byte64>(register_id::mm0)
        == to_byte64(0xba5eba11ull));

    proc->resume();
    proc->wait_on_signal();
```

```
❹ REQUIRE(regs.read_by_id_as<byte128>(register_id::xmm0) ==
    ❺ to_byte128(64.125));

  proc->resume();
  proc->wait_on_signal();

❻ REQUIRE(regs.read_by_id_as<long double>(register_id::st0) ==
      64.125L);
}
```

We launch *reg_read* and resume to the first trap. At this point, it should have written `0xcafecafe` to `r13`, so we check this ❶ and go on to test subregisters ❷, MMX ❸, XMM ❹, and x87 ❻. Note that we use the `ull` suffix ❸ to ensure that the integer is an `unsigned long long` (that is, a `std::uint64_t`) and the `L` suffix ❻ to ensure the floating-point number is a `long double`.

Generally, you shouldn't compare floating-point values by their bit representations due to precision and rounding issues. However, binary can exactly represent floating-point values whose denominators are powers of two (such as 0.5, 0.25, 0.125, and so on), so it's safe for us to do so here ❺. We call such floating-point numbers *dyadic rationals*.

Exposing Registers

The last thing to do is expose the register reading and writing operations to the debugger's user. We'll support four types of commands: `register read` to read GPRs, `register read all` to read all registers, `register read` *<register name>* to read a given register, and `register write` *<register name> <value>*: to write a given register.

Before we add this new set of commands, let's implement a quick and easy way of getting help inside the debugger. This will enable users to see what commands are available and how to use them. It'll also help remind you of the syntax for these commands as their number grows.

The Help Command

Add a new command to `handle_command` in *sdb/tools/sdb.cpp*:

```
void handle_command(
      std::unique_ptr<sdb::process>& process,
      std::string_view line) {
    --snip--
    else if (is_prefix(command, "help")) {
        print_help(args);
    }
    --snip--
}
```

Then, implement the `print_help` function. You can put as much or as little detail in here as you like. I'll support subcommands so users can enter things like `help register` to get information about all register commands:

```
namespace {
    void print_help(const std::vector<std::string>& args) {
        if (args.size() == 1) {
         ❶ std::cerr << R"(Available commands:
    continue    - Resume the process
    register    - Commands for operating on registers
)";
        }

        else if (is_prefix(args[1], "register")) {
            std::cerr << R"(Available commands:
    read
    read <register>
    read all
    write <register> <value>
)";
        }
        else {
            std::cerr << "No help available on that\n";
        }
    }
}
```

We call this `R"(`*text*`)"` syntax ❶ a *raw string literal*. It lets us include newlines and other characters that normally require escape sequences in the string and have them be considered part of the string. Now we can implement register commands.

Register Reading

We'll start by implementing `handle_register_command`. Add a new branch to the `handle_command` function in *sdb/tools/sdb.cpp* for the register commands:

```
    void handle_command(
          std::unique_ptr<sdb::process>& process,
          std::string_view line) {
        --snip--
        else if (is_prefix(command, "register")) {
            handle_register_command(*process, args);
        }
        --snip--
    }
```

We'll defer most of the work to `handle_register_read` and `handle_register_write` functions:

```
namespace {
    void handle_register_command(
         sdb::process& process,
         const std::vector<std::string>& args) {
        if (args.size() < 2) {
            print_help({ "help", "register" });
            return;
        }

        if (is_prefix(args[1], "read")) {
            handle_register_read(process, args);
        }
        else if (is_prefix(args[1], "write")) {
            handle_register_write(process, args);
        }
        else {
            print_help({ "help", "register" });
        }
    }
}
```

If the user passes the wrong number of arguments, or a subcommand that wasn't expected, we print the help information. Otherwise, we call the relevant `handle_register_*` function.

The `handle_register_read` function must be able to display registers in a visually pleasing format. This requires doing some string formatting, which can be deeply unpleasant if you're using the facilities built into C++17. Fortunately, there's a library called `fmtlib`, which is a fast, modern alternative to iostreams. Facilities based on `fmtlib` were added to C++20, so if you're using C++20, you can use them instead. Let's add `fmtlib` as a dependency of the project so we can make use of the niceties it offers. First, we add the dependency to *sdb/vcpkg.json*:

```
{
    "dependencies": ["libedit", "catch2", "fmt"]
}
```

Then, we find the package in *sdb/CMakeLists.txt*:

```
--snip--
find_package(PkgConfig REQUIRED)
pkg_check_modules(libedit REQUIRED IMPORTED_TARGET libedit)
find_package(fmt CONFIG REQUIRED)
--snip--
```

Finally, we link against `fmt::fmt` in *sdb/tools/CMakeLists.txt*:

```
--snip--
target_link_libraries(sdb PRIVATE sdb::libsdb PkgConfig::libedit fmt::fmt)
--snip--
```

Now we can start writing the `handle_register_read` function by creating a formatter for register values:

```
#include <fmt/format.h>
#include <fmt/ranges.h>

namespace {
    void handle_register_read(
         sdb::process& process,
         const std::vector<std::string>& args) {
        auto format = [](auto t) {
         ❶ if constexpr (std::is_floating_point_v<decltype(t)>) {
             ❷ return fmt::format("{}", t);
            }
         ❸ else if constexpr (std::is_integral_v<decltype(t)>) {
             ❹ return fmt::format("{:#0{}x}", t, sizeof(t) * 2 + 2);
            }
            else {
             ❺ return fmt::format("[{:#04x}]", fmt::join(t, ","));
            }
        };
    }
}
```

We include *fmt/format.h* for `fmt::format` and *fmt/ranges.h* for `fmt::join`. The formatter is a generic lambda function, so we can easily use it with `std::variant` later. It has three branches, for floating-point, integral, and vector registers. We use `decltype(expression)` to get the type of an expression, allowing us to check whether `t` is floating point ❶ or integral ❸. We then format floating-point numbers using the default format ({}) and `fmt::format`, which resembles `fmt::print` but returns a `std::string` rather than printing to `stdout` ❷.

The integral formatter ❹ is a bit more complicated. As when printing breakpoint site addresses, `{:#x}` prints a hexadecimal value with a leading `0x`, and `{:#04x}` pads the output with `0`s to make it four characters wide. By writing the nested specifier `{:#0{}x}`, we enable one of the `fmt::format` arguments to determine the amount of padding to use. In this case, `sizeof(t) * 2 + 2` adds two characters per byte of the integer, plus two characters for the leading `0x`. This padding ensures that registers line up and have the correct size when we print them.

To format vector registers ❺, we use `fmt::join` to join arrays using a comma and then format the internal bytes as hexadecimal with a leading `0x`, padded to four characters.

Next, we'll write branches for the two main reading cases: reading many registers and reading one register. The first branch will print either all registers (if the user entered `register read all`) or just the GPRs (if the user entered `register read`). The second branch will print a single requested register:

```
void handle_register_read(
     sdb::process& process,
     const std::vector<std::string>& args) {
    --snip--
    if (args.size() == 2 or
        (args.size() == 3 and args[2] == "all")) {
     ❶ for (auto& info : sdb::g_register_infos) {
            auto should_print = (❷ args.size() == 3 or
              ❸ info.type == sdb::register_type::gpr)
                and info.name != "orig_rax";
            if (!should_print) continue;
            auto value = process.get_registers().read(info);
            fmt::print("{}:\t{}\n", info.name, std::visit(format, value));
        }
    }
    else if (args.size() == 3) {
        try {
         ❹ auto info = sdb::register_info_by_name(args[2]);
            auto value = process.get_registers().read(info);
         ❺ fmt::print("{}:\t{}\n", info.name, std::visit(format, value));
        }
     ❻ catch (sdb::error& err) {
            std::cerr << "No such register\n";
            return;
        }
    }
    else {
     ❼ print_help({ "help", "register" });
    }
}
```

We loop over the register information in the system ❶. If the user specified all registers ❷ or wants just GPRs and the current register info is a GPR ❸, we get the value of that register and print its value using the formatter we defined earlier. We don't print out the `orig_rax` register if all registers are requested, as it's not a real register, just something that `ptrace` uses to communicate information about syscalls.

For the second branch, we search for the `sdb::register_info` value with the given name ❹. If this fails, `sdb::register_info_by_name` will throw an exception, which we can catch; we then print out an error message ❻. If the retrieval succeeds, we read the value and print it out using our formatter ❺.

If none of these options match, we print a help message ❼.

Register Writing

To write to registers, we must write a small parser, which we'll put in a function named `parse_register_value`. For now, here's the usage code, which goes in *sdb/tools/sdb.cpp*:

```
namespace {
    void handle_register_write(
         sdb::process& process,
         const std::vector<std::string>& args) {
     ❶ if (args.size() != 4) {
            print_help({ "help", "register" });
            return;
        }
        try {
         ❷ auto info = sdb::register_info_by_name(args[2]);
         ❸ auto value = parse_register_value(info, args[3]);
         ❹ process.get_registers().write(info, value);
        }
        catch (sdb::error& err) {
         ❺ std::cerr << err.what() << '\n';
            return;
        }
    }
}
```

First, we print out some help if the user provides the wrong number of arguments ❶. Then, we try to look up the register info using the supplied register name ❷, parse the supplied value with the `parse_register_value` function we'll write next ❸, and carry out the register write ❹. If any of these actions fail, we'll print out the error message ❺.

Inside `parse_register_value`, we'll defer to parsers for integral, floating-point, and vector types. Again, we'll write the usage code first, pretending those functions already exist:

```
#include <libsdb/parse.hpp>

namespace {
    sdb::registers::value parse_register_value(
         sdb::register_info info, std::string_view text) {
        try {
            if (info.format == ❶
                 sdb::register_format::uint) {
                switch (info.size) {
                case 1: return sdb::to_integral<std::uint8_t>(text, 16).value(); ❷
                case 2: return sdb::to_integral<std::uint16_t>(text, 16).value();
                case 4: return sdb::to_integral<std::uint32_t>(text, 16).value();
                case 8: return sdb::to_integral<std::uint64_t>(text, 16).value();
                }
```

```
            }
            else if (info.format == ❸
                 sdb::register_format::double_float) {
                return sdb::to_float<double>(text).value(); ❹
            }
            else if (info.format == ❺
                 sdb::register_format::long_double) {
                return sdb::to_float<long double>(text).value();
            }
            else if (info.format == ❻
                 sdb::register_format::vector) {
                if (info.size == 8) {
                    return sdb::parse_vector<8>(text); ❼
                }
                else if (info.size == 16) {
                    return sdb::parse_vector<16>(text);
                }
            }
        }
        catch (...) {} ❽
        sdb::error::send("Invalid format");
    }
}
```

If the register we're trying to write to expects an unsigned integer ❶, we'll parse one of the correct size using a `to_integral` function ❷ that we'll write shortly. If a `double` ❸ or `long double` ❺ is expected, we'll call out to another helper that we'll call `to_float` ❹. Finally, if the register is a vector register ❻, we'll use yet another helper, which we'll call `parse_vector` ❼. We'll just catch any exceptions that are thrown in this process ❽, then throw one of our own if we haven't already returned a valid object. The `std::optional::value` function throws an exception if there is no stored value, so this `catch` block will be triggered if any of our parsers returns an empty optional.

C++ comes with several options for converting strings to integers. Unfortunately, the interfaces can be unwieldy, and they report success even if they don't use the whole string in their conversion. We'll write a small wrapper around `std::from_chars` to solve this problem. Eventually, we'll use these functions elsewhere in *libsdb*, so create a new file at *sdb/include/libsdb/parse.hpp* with these contents:

```
#ifndef SDB_PARSE_HPP
#define SDB_PARSE_HPP

#include <charconv>
#include <cstdint>
#include <optional>
#include <string_view>
```

```
namespace sdb {
    template <class I>
 ❶ std::optional<I> to_integral(std::string_view sv, int base = 10) {
        auto begin = sv.begin();
     ❷ if (base == 16 and sv.size() > 1 and
            begin[0] == '0' and begin[1] == 'x') {
            begin += 2;
        }

        I ret;
     ❸ auto result = std::from_chars(begin, sv.end(), ret, base);

     ❹ if (result.ptr != sv.end()) {
            return std::nullopt;
        }
        return ret;
    }
}

#endif
```

We return a `std::optional<I>` ❶, which represents "either `I` or nothing." If the base is `16` (meaning hexadecimal), we want to allow an `0x` prefix, so we skip past it ❷. We then call `from_chars` to do the main parsing ❸. The call to `std::from_chars` will succeed even if the entire string hasn't been read, so we return an empty optional if some input remains ❹.

We write `to_float` in a manner similar to `to_integral`:

```
namespace sdb {
    template <class F>
    std::optional<F> to_float(std::string_view sv) {
        F ret;
        auto result = std::from_chars(sv.begin(), sv.end(), ret);

        if (result.ptr != sv.end()) {
            return std::nullopt;
        }
        return ret;
    }
}
```

This function simply calls `std::from_chars` and checks that all the input was used.

NOTE *Although `std::from_chars` is part of C++17, GCC's implementation only shipped in version 11. If you're stuck with an older version of GCC, you can use the `fast_float` library:* https://github.com/fastfloat/fast_float. *This library is available in `vcpkg` in the `fast-float` port.*

The parse_vector function is going to parse text in the same format we output vectors in: a comma-separated list of hexadecimal integers with a leading 0x, surrounded with square brackets. We won't allow spaces between the elements.

We'll start by checking that [is the first character:

```
#include <array>
#include <cstddef>

namespace sdb {
 ❶ template <std::size_t N>
   auto parse_vector(std::string_view text) {
    ❷ auto invalid = [] { sdb::error::send("Invalid format"); };

    ❸ std::array<std::byte, N> bytes;
      const char* c = text.data();

    ❹ if (*c++ != '[') invalid();
```

Yes, templates can take compile-time integers as well as types ❶! It's pretty useful for cases like this. We write a simple lambda that we can use to signal there was an error just by calling invalid() ❷. We pass our template parameter to std::array so we have an array of the correct number of bytes requested ❸, then ensure that we got the expected first character ❹.

Next, we'll loop over the first $N - 1$ elements, parsing an element and a comma each time (we'll do the Nth element after, since it won't have a trailing comma):

```
    --snip--
    for (auto i = 0; i < N - 1; ++i) {
        bytes[i] = to_integral<std::byte>({ c, 4 }, 16).value();
        c += 4;
        if (*c++ != ',') invalid();
    }
```

We first parse the four characters from c into a std::byte. The to_integral function doesn't quite support std::byte, but we'll add support for this after we're done with this function. Then we skip c over the integer we just parsed and make sure there's a comma.

Finally, we parse the last value, then make sure we ended up with a] and that we reached the end of the input:

```
    --snip--
    bytes[N - 1] = to_integral<std::byte>({ c, 4 }, 16).value();
    c += 4;

    if (*c++ != ']') invalid();
    if (c != text.end()) invalid();
```

```
        return bytes;
    }
}
```

As mentioned, we need to extend `to_integral` with support for `std::byte`. We can add a *specialization* of `to_integral` for `std::byte` that just calls the `std::uint8_t` version:

```
namespace sdb {
    template<>
    inline std::optional<std::byte> to_integral(
          std::string_view sv, int base) {
        auto uint8 = to_integral<std::uint8_t>(sv, base);
        if (uint8) return static_cast<std::byte>(*uint8);
        return std::nullopt;
    }
}
```

This version of `to_integral` will be called only if `std::byte` is specified as the template argument. Make sure that you write it below the implementation of the primary template for `to_integral`, but above the implementation of `parse_vector` so that `parse_vector` will use this specialization when needed. Also make sure that you declare it as `inline` so that this header can be safely included in multiple translation units.

With all that written, you should be able to read and write registers from the command line!

Now that we can read the program counter, we can print this out when the process stops. The program counter is a virtual address, and we currently don't have a type for representing virtual addresses. Let's add one to *sdb/include/libsdb/types.hpp*. We'll call it `sdb::virt_addr`. When we add support for ELF and DWARF parsing, we'll also have to deal with other kinds of addresses, so having a specific type for virtual addresses rather than a simple type alias will make our code more robust and prevent us from mixing up our address types:

```
#include <cstdint>

namespace sdb {
    class virt_addr {
    public:
        virt_addr() = default;
        explicit virt_addr(std::uint64_t addr)
            : addr_(addr) {}

        std::uint64_t addr() const {
            return addr_;
        }
```

```
    private:
        std::uint64_t addr_ = 0;
    };
}
```

This is essentially a wrapper for a `std::uint64_t`. We mark the constructor as `explicit` to disallow implicit conversions; this will prevent us from accidentally constructing a `virt_addr` from something that isn't one. We write a getter function for the stored address. Finally, we declare a member to hold the wrapped address.

We want to be able to compare virtual addresses and change them by a given offset. The code to do so is a lot of repetitive boilerplate, but someone's got to write it:

```
namespace sdb {
    class virt_addr {
    public:
        --snip--
        virt_addr operator+(std::int64_t offset) const {
            return virt_addr(addr_ + offset);
        }
        virt_addr operator-(std::int64_t offset) const {
            return virt_addr(addr_ - offset);
        }
        virt_addr& operator+=(std::int64_t offset) {
            addr_ += offset;
            return *this;
        }
        virt_addr& operator-=(std::int64_t offset) {
            addr_ -= offset;
            return *this;
        }
        bool operator==(const virt_addr& other) const {
            return addr_ == other.addr_;
        }
        bool operator!=(const virt_addr& other) const {
            return addr_ != other.addr_;
        }
        bool operator<(const virt_addr& other) const {
            return addr_ < other.addr_;
        }
        bool operator<=(const virt_addr& other) const {
            return addr_ <= other.addr_;
        }
        bool operator>(const virt_addr& other) const {
            return addr_ > other.addr_;
        }
        bool operator>=(const virt_addr& other) const {
            return addr_ >= other.addr_;
```

```
        }
        --snip--
    };
}
```

These functions are all fairly self-explanatory; they just wrap operations on the underlying address but maintain some type safety, so that we can't, for example, compare a virtual address with some other kind of address.

Next, add a helper for getting the program counter in *sdb/include/libsdb/process.hpp*:

```
#include <libsdb/types.hpp>

namespace sdb {
    class process {
    public:
        --snip--
        virt_addr get_pc() const {
            return virt_addr{
                get_registers().read_by_id_as<std::uint64_t>(register_id::rip)
            };
        }
        --snip--
    };
}
```

Then refactor `print_stop_reason` in *sdb/tools/sdb.cpp* to use `fmtlib` and include the program counter if stopped due to a signal:

```
void print_stop_reason(const sdb::process& process, sdb::stop_reason reason) {
    std::string message;
    switch (reason.reason) {
    case sdb::process_state::exited:
        message = fmt::format("exited with status {}",
                              static_cast<int>(reason.info));
        break;
    case sdb::process_state::terminated:
        message = fmt::format("terminated with signal {}",
                              sigabbrev_np(reason.info));
        break;
    case sdb::process_state::stopped:
        message = fmt::format("stopped with signal {} at {:#x}",
                              sigabbrev_np(reason.info), process.get_pc().addr());
        break;
    }

    fmt::print("Process {} {}\n", process.pid(), message);
}
```

Register Interactions

Here's a short conversation I had with my debugger that shows the commands working:

```
$ tools/sdb test/targets/reg_read
sdb> c
Process 15811 stopped with signal TRAP at 0x558a749e114c
sdb> reg read rax
rax:    0x0000000000000000
sdb> reg write ah 0x42
sdb> reg read rax
rax:    0x0000000000004200
sdb> reg read st0
st0:    0
sdb> reg read mm0
mm0:    [0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00]
sdb> reg write st0 65.42
sdb> reg read st0
st0:    65.42
sdb> reg read mm0
mm0:    [0x0a,0xd7,0xa3,0x70,0x3d,0x0a,0xd7,0x82]
sdb> reg read
rax:    0x0000000000004200
rdx:    0x00007fffffffe158
rcx:    0x0000555555555156
rbx:    0x0000000000000000
rsi:    0x0000000000000005
rdi:    0x00000000000036cf
rbp:    0x00007fffffffe030
rsp:    0x00007fffffffe030
r8:     0x00007ffff7fa9f10
r9:     0x00007ffff7fc9040
r10:    0x00007ffff7fc3908
r11:    0x0000000000000246
r12:    0x00000000000036cf
r13:    0x00000000cafecafe
r14:    0x0000555555557df8
r15:    0x00007ffff7ffd040
rip:    0x0000555555555156
eflags: 0x0000000000000246
cs:     0x0000000000000033
fs:     0x0000000000000000
gs:     0x0000000000000000
ss:     0x000000000000002b
ds:     0x0000000000000000
es:     0x0000000000000000
```

I started by reading the `rax` register, which stored the value `0`, and then I wrote `0x42` into the `ah` subregister. Reading `rax` again shows that `0x42` was correctly stored into the upper byte of the lowest 16 bits of the register. I then read `st0` and `mm0`, which use the same memory and are both filled with 0 bits. After writing `65.42` into `st0`, reading `st0` returns the same value and reading `mm0` returns the byte pattern used to encode that floating-point number. Both of these behaviors are correct. I then read all GPRs, and it prints out a long list of registers with their names all correctly aligned.

Summary

In this chapter, you learned the basics of x64 assembly language and used it to write tests for the debugger's register interaction functions. In the next chapter, you'll learn about software breakpoints and add support for them to your debugger.

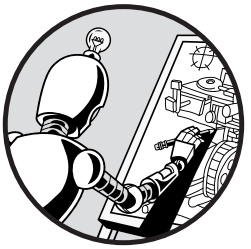
Check Your Knowledge

1. What does `q` stand for in the `movq` instruction?
2. Which characters mark x64 assembler directives and labels?
3. Machine instructions go in which ELF object file section?
4. The system call ID you wish to execute goes in which register before invoking the `syscall` instruction?
5. What part of the x64 instruction set is used for `long double` support?
6. Which assembler directive is used to output a null-terminated ASCII string?

7

SOFTWARE BREAKPOINTS

All beautiful things in the world
stop
to look back at you, softly.



Breakpoints allow debugger users to specify a code location at which execution of the program should halt, allowing for further inspection of the program's state. Breakpoints are one of the most important features of any debugger, but to users, they often feel like magic. Fortunately, if you look past the smoke and mirrors, you'll find that breakpoints are quite simple (though rather ingenious) in their implementation.

In this chapter, you'll learn about software breakpoints and implement them in *sdb*, and then you'll add the usual automated tests. You'll also implement stepping over single machine instructions.

Hardware vs. Software Breakpoints

A debugger can support two main kinds of breakpoints: hardware and software. Hardware breakpoints involve setting architecture-specific registers to produce breaks for you, whereas software breakpoints involve modifying the machine instructions in the process's memory.

While a tool can set an unlimited number of software breakpoints, the number of hardware breakpoints is limited by the number of debug registers in the system. On x64, there are only four hardware breakpoint registers.

Hardware breakpoints have the powerful ability to trigger breaks if a given address is executed, written to, or read from. Software breakpoints can trigger breaks on execution only. Hardware breakpoints also enable the debugging of program exploits that involve overwriting memory with executable code, so they can be useful in security and reverse engineering situations. We'll focus on software breakpoints in this chapter and then implement hardware breakpoints in Chapter 9.

Implementing Software Breakpoints

I mentioned that we set software breakpoints by modifying the executing code on the fly. You might now be asking yourself exactly how to modify the code, and how the debugger can tell that the execution has hit the breakpoint.

To modify a process's code, we can use `ptrace`. We previously relied on this utility to read and write data, but we can also use it on the machine instructions held in memory. (If only `ptrace` could also make coffee, it could solve all of our low-level debugging problems for us.)

We must make a modification that causes the processor to halt and signal the program when executing the code at the breakpoint address. On x64, we accomplish this by overwriting the instruction at that address with the `int3` instruction.

The x64 architecture has an *interrupt vector table* that the operating system can use to register handlers for various events, such as dividing by zero, accessing protected memory, or executing invalid opcodes. You can think of these handlers as error handling callbacks at the hardware level, and we'll look at them in more detail in Chapter 10. When the processor executes the `int3` instruction, it passes control to the breakpoint interrupt handler, which—in the case of Linux—signals the process with a `SIGTRAP`.

The last piece of the puzzle is how the debugger gets notified of the break. We already wrote a `wait_on_signal` function that can listen for signals being sent to the child process. We can use that function here as well; we'll set the breakpoint, continue the program, call `wait_on_signal`, and block the program until the `SIGTRAP` occurs. We can then communicate this breakpoint to the user, perhaps by printing the source location the debugger has reached or changing the focused line in a GUI debugger.

With the theory out of the way, we can start implementing breakpoints.

Representing Locations

Let's create a type to represent and manage a single breakpoint location. Importantly, when a user sets a single logical breakpoint, that breakpoint may be associated with several locations. Say, for example, they set a breakpoint in the C++ function `to_string`. There are many overloads of `to_string` with different types, so we'll need to create multiple physical breakpoints for each overload. We'll differentiate between the logical, user-level breakpoint and the physical, hardware-level breakpoint by calling the former `sdb::breakpoint` and the latter `sdb::breakpoint_site`. In this chapter, we're only implementing setting a breakpoint on a memory address, so we only need to implement `sdb::breakpoint_site`. We'll implement `sdb::breakpoint` in Chapter 14.

Instances of `sdb::process` should maintain a list of all breakpoint sites set on that process. As such, users of *libsdb* should be able to create a breakpoint only by making a request to a `process` object. We'll implement this behavior by making the constructor for `breakpoint_site` private and making `process` a `friend`, granting it access to the `private` members of `breakpoint_site`. We should also disallow copying and moving of `breakpoint_site` objects. Create a new file at *sdb/include/libsdb/breakpoint_site.hpp* with these contents:

```
#ifndef SDB_BREAKPOINT_SITE_HPP
#define SDB_BREAKPOINT_SITE_HPP

#include <cstdint>
#include <cstddef>
#include <libsdb/types.hpp>

namespace sdb {
 ❶ class process;

    class breakpoint_site {
    public:
     ❷ breakpoint_site() = delete;
        breakpoint_site(const breakpoint_site&) = delete;
        breakpoint_site& operator=(const breakpoint_site&) = delete;

     ❸ using id_type = std::int32_t;
     ❹ id_type id() const { return id_; }

     ❺ void enable();
        void disable();

     ❻ bool is_enabled() const { return is_enabled_; }
        virt_addr address() const { return address_; }

        bool at_address(virt_addr addr) const {
            return address_ == addr;
        }
```

```
        bool in_range(virt_addr low, virt_addr high) const {
            return low <= address_ and high > address_;
        }

    private:
     ❼ breakpoint_site(
            process& proc, virt_addr address);
        friend process;

        id_type id_;
        process* process_;
        virt_addr address_;
        bool is_enabled_;
     ❽ std::byte saved_data_;
    };
}

#endif
```

We forward-declare `sdb::process` ❶ so we can use references and pointers to it in this file without including *process.hpp*. Then we begin the implementation of `breakpoint_site` by explicitly disabling default construction and all copy and move behaviors ❷. Breakpoints should have unique identifiers, referenced either from code or on the command line, so we create an `id` type alias ❸ for the actual integral type so that we don't have to care about it elsewhere. We also write a member function that retrieves the breakpoint site's ID ❹.

We'll need member functions to enable and disable the breakpoint site, but these will be complex to write, so for now, we just declare them ❺; we'll implement them later. We implement the functions for retrieving whether the site is enabled and the address on which it's set inline, because they're very short ❻. We also write functions for checking whether a breakpoint site is at a given address or lies in an address range. These functions may appear unnecessary, but they'll come in handy when we add support for source-level breakpoints in Chapter 14.

We declare a private constructor ❼ and declare `sdb::process` as a `friend` so that it can access this constructor. Finally, we declare members for all the information that a breakpoint site needs to track, including a member to hold the data we replace with the `int3` instruction when setting a breakpoint ❽.

Most of this code merely tracks state information; the real magic happens in the `enable` and `disable` functions. Before we implement those, let's enable the creation and management of the breakpoint sites.

Creating Sites

To create a breakpoint site, we first need a constructor in *sdb/src/breakpoint_site.cpp* that initializes members and generates a unique ID:

```
#include <libsdb/breakpoint_site.hpp>

namespace {
    auto get_next_id() {
     ❶ static sdb::breakpoint_site::id_type id = 0;
     ❷ return ++id;
    }
}

sdb::breakpoint_site::breakpoint_site(
    process& proc, virt_addr address)
    : process_{ &proc }, address_{ address }, is_enabled_{ false },
    saved_data_{} {
    id_ = get_next_id();
}
```

We implement `get_next_id` with a function-local static variable ❶ that is initialized exactly once, on the first call to the function. Subsequent calls will use the same object. We increment and return the result on each call ❷, and because we use the pre-increment version of `++`, the first ID will be `1`. The constructor for `breakpoint_site` calls this function and initializes the `id_` member with the result.

Add *breakpoint_site.cpp* to your *sdb/src/CMakeLists.txt* file to add it to the library:

```
add_library(libsdb ... breakpoint_site.cpp)
```

Now we'll add support to `sdb::process` for creating and tracking breakpoint sites. Add the following to *sdb/include/libsdb/process.hpp*:

```
#include <vector>
#include <libsdb/breakpoint_site.hpp>

namespace sdb {
    class process {
    public:
        --snip--

     ❶ breakpoint_site& create_breakpoint_site(virt_addr address);
        // Some way to iterate over and remove breakpoint sites
        --snip--

    private:
        --snip--
     ❷ std::vector<std::unique_ptr<breakpoint_site>> breakpoint_sites_;
    };
}
```

We add a new `breakpoint_sites_` member ❷. We can't store a `std::vector <breakpoint_site>` because we can't copy or move a breakpoint site, but we get around this by dynamically allocating them and storing `std::unique_ptr` values instead. We also declare a function to create a breakpoint site at a given virtual address ❶.

Managing Sites

We're still missing a way for client code to iterate over breakpoint sites, retrieve a site given an ID or an address, and perform other interactions. We have a few options for implementing these operations:

- Write a function that returns a reference to `breakpoint_sites_`. This is pretty icky, as it's very much an implementation detail that we're storing these as `std::unique_ptr`s, and we don't want to let users modify the vector itself.
- Write a custom type, such as `breakpoint_site_collection`, that exposes functions for all of the operations we want to support, like `find_by_id(id)`, `for_each(callback)`, and so on. This involves quite a lot of code overhead.
- Return an `indirect_range` that dereferences each `std::unique_ptr` before returning it on iteration. This requires either pulling in some other dependency, like Boost, or writing some somewhat gnarly template code.

To keep the C++ relatively simple and clean, we'll choose the second option. Here are some operations we should support:

- Adding a new breakpoint site to the collection
- Checking whether a site exists based on an ID or address
- Getting a site by ID or address
- Removing a site by ID or address
- Iterating over sites
- Retrieving the size of the collection
- Checking whether a site is empty

However, there are other kinds of stopping points that the debugger should support. We'll use the name *stop point* to refer to breakpoint sites, source-level breakpoints, and watchpoints. We don't want to have to write three separate types, called `breakpoint_site_collection`, `watchpoint_collection`, and `breakpoint_collection`, that do essentially the same thing. As such, we'll write a class template called `sdb::stoppoint_collection` that can cover all of our needs. Create a new file at *sdb/include/libsdb/stoppoint_collection.hpp* with these contents:

```
#ifndef SDB_STOPPOINT_COLLECTION_HPP
#define SDB_STOPPOINT_COLLECTION_HPP

#include <vector>
#include <memory>
#include <libsdb/types.hpp>

namespace sdb {
    template <class Stoppoint>
    class stoppoint_collection {
    public:
        Stoppoint& push(std::unique_ptr<Stoppoint> bs);

        bool contains_id(typename Stoppoint::id_type id) const; ❶
        bool contains_address(virt_addr address) const;
        bool enabled_stoppoint_at_address(virt_addr address) const;

        Stoppoint& get_by_id(typename Stoppoint::id_type id); ❷
        const Stoppoint& get_by_id(typename Stoppoint::id_type id) const; ❸
        Stoppoint& get_by_address(virt_addr address);
        const Stoppoint& get_by_address(virt_addr address) const;

        void remove_by_id(typename Stoppoint::id_type id);
        void remove_by_address(virt_addr address);

        template <class F>
        void for_each(F f); ❹
        template <class F>
        void for_each(F f) const;

        std::size_t size() const { return stoppoints_.size(); }
        bool empty() const { return stoppoints_.empty(); }

    private:
        using points_t = std::vector<std::unique_ptr<Stoppoint>>;
        points_t stoppoints_;
    };
}
#endif
```

We declare several functions here, but most are fairly straightforward. The `push` function adds a new stop point to the collection. The `contains_id` and `contains_address` functions return whether the collection contains a stop point matching the given ID or virtual address ❶, and `enabled_stoppoint_at_address` is a convenience function for determining if there's an enabled stop point in the collection with the given address. We need the `typename`

keyword for any types that depend on the `Stoppoint` template parameter; it tells the compiler that this name refers to a type rather than a value. The `get_by_id` and `get_by_address` functions will do the same but also return the matching stop point, if there is one. Note that we implement both `const` ❸ and non-`const` ❷ overloads to maintain `const`-correctness. The `remove_by_id` and `remove_by_address` functions will (you guessed it) remove the stop point matching the ID of the address.

The `for_each` functions ❹ are a little more complicated; users will pass them a function or lambda to be called with a reference to each breakpoint site. Finally, `size` and `empty` return the relevant information about the collection state, and we add a private member that stores pointers to the actual stop points.

Before we implement these functions, we can plug the `stoppoint_collection` type into `sdb::process` in *sdb/include/libsdb/process.hpp*:

```
--snip--
#include <libsdb/stoppoint_collection.hpp>

namespace sdb {
    class process {
    public:
        --snip--

        stoppoint_collection<breakpoint_site>&
        breakpoint_sites() { return breakpoint_sites_; }

        const stoppoint_collection<breakpoint_site>&
        breakpoint_sites() const { return breakpoint_sites_; }

    private:
        --snip--
        stoppoint_collection<breakpoint_site> breakpoint_sites_;
    };
}
```

We add functions for retrieving the breakpoint site collection, keeping in mind `const`-correctness by implementing two overloads. We also replace the type of `breakpoint_sites_` with the new `stoppoint_collection` type, supplying `breakpoint_site` as the template argument.

Let's implement `create_breakpoint_site` in *sdb/src/process.cpp*:

```
sdb::breakpoint_site&
sdb::process::create_breakpoint_site(virt_addr address)
{
    if (breakpoint_sites_.contains_address(address)) { ❶
        error::send("Breakpoint site already created at address " +
            std::to_string(address.addr()));
    }
```

```
    return breakpoint_sites_.push(
        std::unique_ptr<breakpoint_site>(new breakpoint_site(*this, address)));
}
```

If two breakpoints point to the same site, they could very easily overwrite each other and cause data loss. As such, we disallow this ❶. After performing this check, we create a new breakpoint site and push it into the `breakpoint_sites_` collection.

We can then implement the functions for `stoppoint_collection`. The code is fairly repetitive, but it does the job. Because `stoppoint_collection` is a template, the implementations must go in the header file rather than in a *.cpp* file so that the compiler can generate code from those member functions whenever it sees a new specialization of the `stoppoint_collection` template. Let's start by implementing `push` in *sdb/include/libsdb/stoppoint_collection.hpp*:

```
namespace sdb {
    template <class Stoppoint>
    Stoppoint& stoppoint_collection<Stoppoint>::push(
          std::unique_ptr<Stoppoint> bs) {
        stoppoints_.push_back(std::move(bs));
        return *stoppoints_.back();
    }
}
```

We push the argument into the `stoppoints_` vector and return a reference to that element. Remember to move unique pointers with `std::move` when transferring ownership; otherwise, the code won't compile. If you're not familiar with move semantics, see the following box.

MOVE SEMANTICS

Move semantics, a feature added in C++11, provides a way to transfer ownership of resources from one object to another. When objects manage resources, such as dynamically allocated memory or file handles, it can sometimes be costly or impossible to copy these resources to other objects. Types that should never be copied are called *move-only* types. Types that are instead costly to copy can use move semantics to avoid those copies in certain circumstances. Consider, for example, the following `std::vector` type, which stores a few MB of data. Say we want to store this vector somewhere:

```
void store(std::vector<big_type>);
{
    std::vector<big_type> copy_is_expensive;
    fill_with_data(copy_is_expensive);
    store(copy_is_expensive);
}
```

(continued)

Copying all of this data to store it somewhere else is unnecessary, because `copy_is_expensive` is about to go out of scope, so its memory will be reclaimed. It would be better to somehow signal to `store` that it can steal, or *move*, the data instead of having to copy it. We do this using `std::move`:

```
void store(std::vector<big_type>);
{
    std::vector<big_type> copy_is_expensive;
    fill_with_data(copy_is_expensive);
    store(std::move(copy_is_expensive));
}
```

In many cases, you'll get move semantics for free. If you store types that are move-aware, like `std::vector`, the compiler will generate the necessary operations for you to ensure that moving an object of your type will also move its subobjects. If the types you're storing aren't move-aware but represent some resource, you can implement these move operations yourself by writing a move constructor and move assignment operator.

Rather than the copy versions of the resource, which generally take references to `const` like `t(const t& rhs)` and `t& operator=(const t& rhs)`, the move versions take *rvalue references*. These look like `t(t&& rhs)` and `t& operator (t&& rhs)`. Inside them, you can take the resource from `rhs` and put `rhs` into an empty state.

For more details, see Chapter 5 of Scott Meyers's *Modern Effective C++* (O'Reilly, 2014) or this Stack Overflow question: *https://stackoverflow.com/questions/3106110/what-is-move-semantics*.

Now we can move on to `contains_id` and `contains_address`. These functions, along with many other functions in this type, will need to find stop points by ID or virtual address. To minimize repeated code, let's perform this task in two private members called `find_by_id` and `find_by_address`. Declare them in the type definition:

```
namespace sdb {
    template <class Stoppoint>
    class stoppoint_collection {
        --snip--
    private:
        using points_t = std::vector<std::unique_ptr<Stoppoint>>;

        typename points_t::iterator find_by_id(typename Stoppoint::id_type id);
        typename points_t::const_iterator find_by_id(typename Stoppoint::id_type id) const;
        typename points_t::iterator find_by_address(virt_addr address);
        typename points_t::const_iterator find_by_address(virt_addr address) const;

        points_t stoppoints_;
    };
```

The `find_by_id` and `find_by_address` functions return iterators to the stop point's vector. We make `const` and non-`const` overloads and vary the return type between `iterator` and `const_iterator`, depending on whether the member function itself is declared `const`. Implement them in the same file:

```
#include <algorithm>

namespace sdb {
    template <class Stoppoint>
    auto stoppoint_collection<Stoppoint>::find_by_id(typename Stoppoint::id_type id) ❶
        -> typename points_t::iterator { ❷
        return std::find_if(begin(stoppoints_), end(stoppoints_),
            [=](auto& point) { return point->id() == id; });
    }

    template <class Stoppoint>
    auto stoppoint_collection<Stoppoint>::find_by_id(typename Stoppoint::id_type id) const
        -> typename points_t::const_iterator {
        return const_cast<stoppoint_collection*>(this)->find_by_id(id); ❸
    }

    template <class Stoppoint>
    auto stoppoint_collection<Stoppoint>::find_by_address(virt_addr address)
        -> typename points_t::iterator {
        return std::find_if(begin(stoppoints_), end(stoppoints_),
            [=](auto& point) { return point->at_address(address); });
    }

    template <class Stoppoint>
    auto stoppoint_collection<Stoppoint>::find_by_address(virt_addr address) const
        -> typename points_t::const_iterator {
        return const_cast<stoppoint_collection*>(this)->find_by_address(address);
    }
}
```

We use a few tricks here to minimize duplication and large type names. As `points_t::iterator` is a member type of `sdb::stoppoint_collection<Stoppoint>`, `typename sdb::stoppoint_collection<Stoppoint>::points_t::iterator` would be the normal return type, which is quite the mouthful. Instead, we use *trailing return types* by specifying the return type as `auto` ❶ and writing the actual return type at the end of the signature ❷. Doing this lets us omit the `sdb::stoppoint_collection<Stoppoint>::` part. We do this for all four functions.

We implement the non-`const` version of `find_by_id` with `std::find_if` ❶, an algorithm that returns an iterator to the first element that satisfies the given predicate. We include the header for `std::find_if` at the top of the file. If it can't find such an element, it returns the end iterator it's given. We pass a lambda that returns an indication of whether a given stop point's ID is equal to `id`.

We could implement the `const` version by copying and pasting the code from the non-`const` version, but I've chosen to implement the former in terms of the latter instead. We cast away the `const`ness of `this` in order to call the non-`const` overload ❸. This is fine because the function will end by converting `iterator` to a `const_iterator`, so we're not in danger of breaking `const`-correctness.

The `find_by_address` functions are very similar, except they check for a given address rather than an ID.

Now we can implement the rest of the functions using these private members, starting with the functions `contains_id`, `contains_address`, and `enabled_stoppoint_at_address`:

```
namespace sdb {
    template <class Stoppoint>
    bool stoppoint_collection<Stoppoint>::contains_id(typename Stoppoint::id_type id) const {
        return find_by_id(id) != end(stoppoints_);
    }

    template <class Stoppoint>
    bool stoppoint_collection<Stoppoint>::contains_address(virt_addr address) const {
        return find_by_address(address) != end(stoppoints_);
    }

    template <class Stoppoint>
    bool stoppoint_collection<Stoppoint>::enabled_stoppoint_at_address(
          virt_addr address) const {
        return contains_address(address) and get_by_address(address).is_enabled();
    }
}
```

The `contains_id` and `contains_address` functions now call the relevant `find_by_*` function and indicate whether it returned the end iterator (meaning it found no matching stop point). Next are the `get_by_id` overloads:

```
#include <libsdb/error.hpp>

namespace sdb {
    template <class Stoppoint>
    Stoppoint& stoppoint_collection<Stoppoint>::get_by_id(
          typename Stoppoint::id_type id) {
        auto it = find_by_id(id);
        if (it == end(stoppoints_))
            error::send("Invalid stoppoint id");
        return **it;
    }

    template <class Stoppoint>
    const Stoppoint& stoppoint_collection<Stoppoint>::get_by_id(
          typename Stoppoint::id_type id) const {
```

```
            return const_cast<stoppoint_collection*>(this)->get_by_id(id);
        }
    }
```

We include the header for `sdb::error`. The non-`const` version finds the relevant iterator using `find_by_id`. If it returns the end iterator, the function throws an exception. Otherwise, it returns a reference to the found element. Note that we must dereference the iterator twice: once for the iterator and once for the `std::unique_ptr` it references. We implement the `const` version in terms of the non-`const` one following the approach we took previously.

The `get_by_address` overloads follow the same pattern as the previous functions:

```
namespace sdb {
    template <class Stoppoint>
    Stoppoint& stoppoint_collection<Stoppoint>::get_by_address(
          virt_addr address) {
        auto it = find_by_address(address);
        if (it == end(stoppoints_))
            error::send("Stoppoint with given address not found");
        return **it;
    }

    template <class Stoppoint>
    const Stoppoint& stoppoint_collection<Stoppoint>::get_by_address(
          virt_addr address) const {
        return const_cast<stoppoint_collection*>(this)->get_by_address(address);
    }
}
```

These are essentially the same as the `get_by_id` functions, but they call `find_by_address` instead of `find_by_id`. Next are the `remove_by_*` functions:

```
namespace sdb {
    template <class Stoppoint>
    void stoppoint_collection<Stoppoint>::remove_by_id(typename Stoppoint::id_type id) {
        auto it = find_by_id(id);
        (**it).disable();
        stoppoints_.erase(it);
    }

    template <class Stoppoint>
    void stoppoint_collection<Stoppoint>::remove_by_address(virt_addr address) {
        auto it = find_by_address(address);
        (**it).disable();
        stoppoints_.erase(it);
    }
}
```

For both functions, we find the relevant stop point, disable it so we don't have any leaky stop points, and then erase them from the container.

The last function we need is `for_each`. This loops over the stop points in the collection, calling the `f` parameter with each one:

```
namespace sdb {
    template <class Stoppoint>
 ❶ template <class F>
    void stoppoint_collection<Stoppoint>::for_each(❷ F f) {
        for (auto& point : stoppoints_) {
         ❸ f(*point);
        }
    }
    template <class Stoppoint>
    template <class F>
    void stoppoint_collection<Stoppoint>::for_each(F f) const {
        for (const auto& point : stoppoints_) {
            f(*point);
        }
    }
}
```

Having two `template` lines in a row might look strange, but this is a how we define a member function template of a class template. The template argument for `for_each` ❶, `F`, is the type of the function to call once for every element in the collection. The `for_each` function takes a single regular function parameter ❷: the function to be called. By making `for_each` a template that takes the function type as a template parameter, we allow users to pass a regular function, a lambda, or some custom function object.

The body of the function is simpler. It loops over all of the stop points in the collection, passing each one as an argument to the supplied function ❸. The `const` version is a copy and paste job. We can't do the `const_cast` trick we did for the other functions because we don't want to accidentally pass a non-`const` stop point to the given function and break correctness.

These breakpoint site management functions involved a fair amount of work and hairy template code, but they'll save us a lot of pain in the future. Let's test the breakpoint site machinery to make sure we're happy with it and then move on to actually setting some breakpoints.

Testing Site Management

In *sdb/test/tests.cpp*, we'll test that we can create breakpoint sites and that their IDs increase as expected:

```
TEST_CASE("Can create breakpoint site", "[breakpoint]") {
    auto proc = process::launch("targets/run_endlessly");
 ❶ auto& site = proc->create_breakpoint_site(virt_addr{ 42 });
    REQUIRE(site.address().addr() == 42);
}
```

```
TEST_CASE("Breakpoint site ids increase", "[breakpoint]") {
    auto proc = process::launch("targets/run_endlessly");

    auto& s1 = proc->create_breakpoint_site(virt_addr{ 42 });
    REQUIRE(s1.address().addr() == 42);

    auto& s2 = proc->create_breakpoint_site(virt_addr{ 43 });
❷   REQUIRE(s2.id() == s1.id() + 1);

    auto& s3 = proc->create_breakpoint_site(virt_addr{ 44 });
    REQUIRE(s3.id() == s1.id() + 2);

    auto& s4 = proc->create_breakpoint_site(virt_addr{ 45 });
    REQUIRE(s4.id() == s1.id() + 3);
}
```

First, we test the creation of breakpoint sites. Because we're not enabling the breakpoints upon creation, it's okay for us to write dummy values for the addresses. It's also reasonable for `create_breakpoint_site` not to check for address validity, as the user may request a breakpoint on an address that isn't yet writable. As such, we create a breakpoint site at address 42 ❶ and ensure that the created site has the correct address.

To test that breakpoint site IDs increase, we create a series of breakpoints and ensure they're allocated subsequent numbers. Recall that IDs are globally unique within a run of the program because we use a function-local static variable. We don't want our tests to depend on the order in which they're run, so we check the IDs relative to each other ❷ rather than considering absolute values. Next, we test searching for sites:

```
TEST_CASE("Can find breakpoint site", "[breakpoint]") {
    auto proc = process::launch("targets/run_endlessly");
❶   const auto& cproc = proc;

❷   proc->create_breakpoint_site(virt_addr{ 42 });
    proc->create_breakpoint_site(virt_addr{ 43 });
    proc->create_breakpoint_site(virt_addr{ 44 });
    proc->create_breakpoint_site(virt_addr{ 45 });

    auto& s1 = proc->breakpoint_sites().get_by_address(virt_addr{ 44 });
    REQUIRE(proc->breakpoint_sites().contains_address(virt_addr{ 44 }));
    REQUIRE(s1.address().addr() == 44);

    auto& cs1 = cproc->breakpoint_sites().get_by_address(virt_addr{ 44 });
    REQUIRE(cproc->breakpoint_sites().contains_address(virt_addr{ 44 }));
    REQUIRE(cs1.address().addr() == 44);

    auto& s2 = proc->breakpoint_sites().get_by_id(s1.id() + 1);
```

```
    REQUIRE(proc->breakpoint_sites().contains_id(s1.id() + 1));
    REQUIRE(s2.id() == s1.id() + 1);
    REQUIRE(s2.address().addr() == 45);

    auto& cs2 = proc->breakpoint_sites().get_by_id(cs1.id() + 1);
    REQUIRE(cproc->breakpoint_sites().contains_id(cs1.id() + 1));
    REQUIRE(cs2.id() == cs1.id() + 1);
    REQUIRE(cs2.address().addr() == 45);
}

TEST_CASE("Cannot find breakpoint site", "[breakpoint]") {
    auto proc = process::launch("targets/run_endlessly");
    const auto& cproc = proc;

    REQUIRE_THROWS_AS(
        proc->breakpoint_sites().get_by_address(virt_addr{ 44 }), error);
    REQUIRE_THROWS_AS(proc->breakpoint_sites().get_by_id(44), error);
    REQUIRE_THROWS_AS(
        cproc->breakpoint_sites().get_by_address(virt_addr{ 44 }), error);
    REQUIRE_THROWS_AS(cproc->breakpoint_sites().get_by_id(44), error);
}
```

We write two tests: one for successful searches and one for unsuccessful searches. We also test `const` ❶ and non-`const` processes to ensure that both work. To test successful searches, we create a series of breakpoint sites ❷ and ensure that the `get_by_*` and `contains_*` member functions retrieve breakpoints with the expected IDs and addresses.

To test for failure, we ensure that both `get_by_*` functions throw errors if we pass a breakpoint site ID that doesn't correspond to a valid site. You can expand these tests if you'd like, but the ones shown here should be enough to give you a sense of security when you make changes to your codebase.

Next, we test retrieving the size and emptiness of a site collection:

```
TEST_CASE("Breakpoint site list size and emptiness", "[breakpoint]") {
    auto proc = process::launch("targets/run_endlessly");
    const auto& cproc = proc;

    REQUIRE(proc->breakpoint_sites().empty());
    REQUIRE(proc->breakpoint_sites().size() == 0);
    REQUIRE(cproc->breakpoint_sites().empty());
    REQUIRE(cproc->breakpoint_sites().size() == 0);

    proc->create_breakpoint_site(virt_addr{ 42 });
    REQUIRE(!proc->breakpoint_sites().empty());
    REQUIRE(proc->breakpoint_sites().size() == 1);
    REQUIRE(!cproc->breakpoint_sites().empty());
    REQUIRE(cproc->breakpoint_sites().size() == 1);
```

```
    proc->create_breakpoint_site(virt_addr{ 43 });
    REQUIRE(!proc->breakpoint_sites().empty());
    REQUIRE(proc->breakpoint_sites().size() == 2);
    REQUIRE(!cproc->breakpoint_sites().empty());
    REQUIRE(cproc->breakpoint_sites().size() == 2);
}
```

To test the `size` and `empty` functions, we first ensure that the size is zero and that the list is empty. We then create a couple of breakpoint sites and ensure that the size increments as expected and that `empty` now returns `true`.

Finally, we test iteration, which requires a little more explaining:

```
TEST_CASE("Can iterate breakpoint sites", "[breakpoint]") {
    auto proc = process::launch("targets/run_endlessly");
    const auto& cproc = proc;

    proc->create_breakpoint_site(virt_addr{ 42 });
    proc->create_breakpoint_site(virt_addr{ 43 });
    proc->create_breakpoint_site(virt_addr{ 44 });
    proc->create_breakpoint_site(virt_addr{ 45 });

    proc->breakpoint_sites().for_each(
     ❶ [addr = 42](auto& site) mutable {
            REQUIRE(site.address().addr() == addr++);
        });

    cproc->breakpoint_sites().for_each(
        [addr = 42](auto& site) mutable {
            REQUIRE(site.address().addr() == addr++);
        });
}
```

As in several of the other tests, we create a process and several breakpoint sites that we can then iterate over. To make the test concise, we use an advanced feature of C++ lambdas: init-captures ❶. This creates a capture with an initial value (`42`). We use the `mutable` keyword to let us change the value of this capture during iteration. (Otherwise, the capture would be declared `const`.) Using this feature, we increment the captured variable on each iteration and check against the site address. We do this for both `const` and non-`const` processes, as usual.

All of these tests should pass.

That was quite a lot of machinery to get through before we get to the real subject of our chapter. But really, it's this kind of machinery and careful thought to the architecture of our program that will help us extend it without getting completely lost later on. It's as much a part of building a debugger as setting a breakpoint is.

Speaking of setting breakpoints, we can start doing this now by implementing the `enable` function.

Enabling Breakpoints

To enable a breakpoint site, we need to replace the instruction at the given address with an `int3` instruction, which we encode as `0xcc`. We must also save the replaced instruction so we can restore the code later; we don't want to forget to execute the user's code! In *sdb/src/breakpoint_site.cpp*, add the following:

```
#include <sys/ptrace.h>
#include <libsdb/breakpoint_site.hpp>
#include <libsdb/process.hpp>
#include <libsdb/error.hpp>

void sdb::breakpoint_site::enable() {
    if (is_enabled_) return;

    errno = 0;
    std::uint64_t data = ptrace(PTRACE_PEEKDATA, process_->pid(), address_, nullptr); ❶
    if (errno != 0) {
        error::send_errno("Enabling breakpoint site failed");
    }

    saved_data_ = static_cast<std::byte>(data & 0xff); ❷

    std::uint64_t int3 = 0xcc;
    std::uint64_t data_with_int3 = ((data & ~0xff) | int3); ❸

    if (ptrace(PTRACE_POKEDATA, process_->pid(), address_, data_with_int3) < 0) { ❹
        error::send_errno("Enabling breakpoint site failed");
    }

    is_enabled_ = true;
}
```

If the breakpoint site is already enabled, we do nothing. Carrying out the rest of the process on an already set breakpoint will likely lead to corrupted code.

We read 64 bits of data from the address at which we need to set the breakpoint using `PTRACE_PEEKDATA` ❶, which communicates errors in a different way from most other `ptrace` requests: it merely sets `errno` rather than communicating an error via its return value. Therefore, to check for an error, we set `errno` to `0`, then check whether `ptrace` set `errno` after the call. If so, then an error occurred, and we report it with `error::send_errno`.

As we received 64 bits of data and need only the first 8 bits, we bitwise AND the data with `0xff` ❷. We then need to replace the bits we just saved with `0xcc`. First, we zero out those bits by bitwise ANDing the data with `~0xff`; then, we bitwise OR it with `0xcc` to set the correct bits ❸. I'll describe this bitwise operation in more detail shortly.

Now that we've replaced the first byte of the data we read, we write it to memory using `PTRACE_POKEDATA` ❹, which is the opposite of `PTRACE_PEEKDATA`. Finally, we set `is_enabled_` to `true` to track the state of the breakpoint site.

Let's now walk through the code's bitwise operations in more detail. (Feel free to skip to the next section if the code already makes sense to you.) On my machine, the memory for the start of the `main` function of *run_endlessly* looks like this:

```
$ objdump -d run_endlessly
--snip--
0000000000001129 <main>:
    1129:       f3 0f 1e fa             endbr64
    112d:       55                      push   %rbp
    112e:       48 89 e5                mov    %rsp,%rbp
--snip--
```

On the left side are memory addresses in hexadecimal. These tell us that `0x1129` is the start address of the function. In the middle, also in hexadecimal, are the bytes stored at those locations. On the right are the assembly instructions corresponding to those bytes. Because x64 is little-endian, we won't get `0xf30f1efa554889e5` if we load this data into a 64-bit integer. The swapped bytes instead give us `0xe5894855fa1e0ff3`.

Our aim is to replace `f3` with `cc`, first saving `f3` aside. We get `f3` by bitwise ANDing with `0xff`. This looks like the following:

```
    e5 89 48 55 fa 1e 0f f3
AND 00 00 00 00 00 00 00 ff
=   00 00 00 00 00 00 00 f3
```

Here is the equivalent of the last 4 bytes of that calculation in binary:

```
    11111010 00011110 0001111 11110011
AND 00000000 00000000 0000000 11111111
=   00000000 00000000 0000000 11110011
```

To replace the last byte with `cc`, we bitwise AND the data with `~0xff` to zero those bits and then bitwise OR it with `cc` to set the correct bits. The `~` in `~0xff` is the complement operator. It flips all of the bits in the integer; so, `~0xff` for a 64-bit number is just a shorter way of writing `0xffffffffffffff00`. (I had to triple-check that I used the correct number of `f` characters, so you can see why this syntax is useful.)

In binary, this process for the last 4 bytes looks like the following:

```
    1111010 00011110 0001111 11110011
AND 1111111 11111111 1111111 00000000
=   1111010 00011110 0001111 00000000
OR  0000000 00000000 0000000 11001100
=   1111010 00011110 0001111 11001100
```

After this brief aside, let's go back to implementing `breakpoint_site::disable`.

Disabling Breakpoints

To disable a breakpoint site, we'll implement a `disable` function. While easier to write than `enable`, it has a subtlety to it: since `ptrace` memory reads and writes operate on whole words rather than bytes, we can't simply write the saved byte back to memory. We need to first read the word at the location to restore, then overwrite the low byte with the original data and write it back to memory:

```
void sdb::breakpoint_site::disable() {
    if (!is_enabled_) return;

    errno = 0;
    std::uint64_t data = ptrace(PTRACE_PEEKDATA, process_->pid(), address_, nullptr);
    if (errno != 0) {
        error::send_errno("Disabling breakpoint site failed");
    }

    auto restored_data =
        ((data & ~0xff) | static_cast<std::uint8_t>(saved_data_));
    if (ptrace(PTRACE_POKEDATA, process_->pid(), address_, restored_data) < 0) {
        error::send_errno("Disabling breakpoint site failed");
    }

    is_enabled_ = false;
}
```

Like in the `enable` function, we return immediately if the site is already disabled. We then read a word of data using `PTRACE_PEEKDATA` and check the return code. We restore the original instruction by masking the first byte with `data & ~0xff` and then bitwise ORing the result with the saved data. After this calculation, we write the result back to memory with `PTRACE_POKEDATA` and record that the site is disabled in the `is_enabled_` member.

Adding Breakpoints to the Debugger

Now we need to expose this breakpoint functionality to users of the debugger. In *sdb/tools/sdb.cpp*, add a new case to `handle_command` for handling breakpoint commands:

```
namespace {
    void handle_command(
         std::unique_ptr<sdb::process>& process,
         std::string_view line) {
        --snip--
```

```
        else if (is_prefix(command, "breakpoint")) {
            handle_breakpoint_command(*process, args);
        }
        --snip--
    }
}
```

This case defers to the `handle_breakpoint_command` function, which we'll write next. We'll also add commands for listing, setting, enabling, disabling, and deleting breakpoints, in that order.

Listing

An operation to list breakpoints should output something like this:

```
Current breakpoints:
1: address = 0xcafe, disabled
2: address = 0xacab, enabled
```

Each entry has the site's ID, followed by its address and whether it's enabled. Let's implement breakpoint listing in *src/tools/sdb.cpp*:

```
namespace {
    void handle_breakpoint_command(sdb::process& process,
          const std::vector<std::string>& args) {
        if (args.size() < 2) {
            print_help({ "help", "breakpoint" });
            return;
        }

        auto command = args[1];

        if (is_prefix(command, "list")) {
         ❶ if (process.breakpoint_sites().empty()) {
                fmt::print("No breakpoints set\n");
            }
         ❷ else {
                fmt::print("Current breakpoints:\n");
                process.breakpoint_sites().for_each(❸[](auto& site) {
                    fmt::print(❹"{}: address = {:#x}, {}\n",
                        site.id(), site.address().addr(),
                        site.is_enabled() ? "enabled" : "disabled");
                    });
            }
            return;
        }
    }
}
```

We start, as usual, by printing a help message if the user provides an incorrect number of arguments. We then implement the `list` subcommand. If no breakpoints are set, we print a message to that effect ❶. Otherwise, we print the list in the format I specified earlier ❷. We pass a lambda to `for_each` to call for each breakpoint site, printing the information for that site ❸.

The format string for the site information ❹ has a fair bit going on. The {} characters are placeholders to fill with the arguments we pass to `fmt::print`. The first placeholder corresponds to the first additional argument, the second placeholder to the second, and so on. The `{:#x}` specifier formats the number as a lowercase hexadecimal number with leading `0x`. The blank {} specifiers format the argument using a default format that depends on the argument's type. This will print the site ID as an integer and the enabled/disabled text as a string.

For the rest of the breakpoint subcommands, we'll have to parse strings to integers. For example, `breakpoint set` should take an address written in hexadecimal. The `breakpoint enable` command should take a site ID as an integer.

Setting

Armed with the `to_integral` helper we wrote in the last chapter, we can implement breakpoint setting:

```
namespace {
  void handle_breakpoint_command(sdb::process& process,
          const std::vector<std::string>& args) {
        --snip--
        if (args.size() < 3) {
            print_help({ "help", "breakpoint" });
            return;
        }

        if (is_prefix(command, "set")) {
            auto address = sdb::to_integral<std::uint64_t>(args[2], 16); ❶

            if (!address) {
                fmt::print(stderr,
                    "Breakpoint command expects address in "
                    "hexadecimal, prefixed with '0x'\n");
                return;
            }

            process.create_breakpoint_site(sdb::virt_addr{ *address }).enable();
            return;
        }
    }
}
```

All breakpoint subcommands other than `list` take an additional argument, so we start by ensuring that the user passed at least three arguments ("breakpoint," the subcommand, and the subcommand arguments).

If the subcommand passed was `set`, we parse the given address using the `to_integral` helper ❶ we wrote earlier. We pass `16` to `to_integral` to make it parse hexadecimal input. If a parsing error occurs, we report it to the user and return. Otherwise, we create a breakpoint site at the parsed address, enable it, and then return.

Enabling, Disabling, and Deleting

The last three commands, to enable, disable, and delete breakpoint sites, are more straightforward:

```
namespace {
  void handle_breakpoint_command(sdb::process& process,
          const std::vector<std::string>& args) {
        --snip--
        auto id = sdb::to_integral<sdb::breakpoint_site::id_type>(args[2]);
        if (!id) {
            std::cerr << "Command expects breakpoint id";
            return;
        }

        if (is_prefix(command, "enable")) {
            process.breakpoint_sites().get_by_id(*id).enable();
        }
        else if (is_prefix(command, "disable")) {
            process.breakpoint_sites().get_by_id(*id).disable();
        }
        else if (is_prefix(command, "delete")) {
            process.breakpoint_sites().remove_by_id(*id);
        }
    }
}
```

The three commands all take a breakpoint site ID as their argument, so we parse this, returning an error if it can't be parsed. We enable or disable breakpoints by calling `get_by_id` to retrieve the site and then calling either `enable` or `disable`, depending on the command passed. To delete the site, we call `remove_by_id`.

Providing Help

Let's also add a help message to `print_help` to remind you of how to work with breakpoints:

```
    --snip--
    if (args.size() == 1) {
```

```
            std::cerr << R"(Available commands:
    breakpoint  - Commands for operating on breakpoints
    continue    - Resume the process
    register    - Commands for operating on registers
)";
    }
    --snip--
    else if (is_prefix(args[1], "breakpoint")) {
        std::cerr << R"(Available commands:
    list
    delete <id>
    disable <id>
    enable <id>
    set <address>
)";
    }
    --snip--
```

The format you use here doesn't really matter; it's mostly to help you remember the command syntax down the road, after you've implemented many more functions. I used raw string literals to avoid having to escape the newlines.

Determining Where to Set Breakpoints

I'm sure you're itching to test your breakpoints. But the ability to set a breakpoint isn't very useful if you don't know what address to set it at. In Chapter 14, we'll add the ability to set breakpoints on function names or source code lines, but for now, we can discover the site address manually.

One way to test the debugger is to write a function that prints "Hello, world!" and set a breakpoint on the function call. If we launch the inferior, it should halt on entry. If we then resume execution, we should see the process stop without printing anything because it will have hit the breakpoint before printing a message.

To perform this test, we need a binary to debug and an address at which to set the breakpoint. Create an *sdb/test/targets/hello_sdb.cpp* file with these contents:

```
#include <cstdio>

int main() {
    std::puts("Hello, sdb!");
}
```

Remember to add the file to *sdb/test/targets/CMakeLists.txt*:

```
--snip--
add_executable(hello_sdb hello_sdb.cpp)
```

Now run a build.

To find the address we need, we can use the handy *objdump* utility, a program that is likely already installed on your system and can display information about object files. Open a shell and execute **`objdump -d hello_sdb`**. You should see a representation of the binary data for your code, alongside the disassembled instructions that correspond to that data.

Somewhere in there, you'll find the code for the `main` function. Mine looks like this:

```
$ objdump -d hello_sdb
--snip--
0000000000001149 <main>:
    1149:       f3 0f 1e fa             endbr64
    114d:       55                      push   %rbp
    114e:       48 89 e5                mov    %rsp,%rbp
    1151:       48 8d 05 ac 0e 00 00    lea    0xeac(%rip),%rax
    1158:       48 89 c7                mov    %rax,%rdi
❶ 115b:       e8 f0 fe ff ff          call   1050 <puts@plt>
    1160:       b8 00 00 00 00          mov    $0x0,%eax
    1165:       5d                      pop    %rbp
    1166:       c3                      ret
--snip--
```

On the left side are memory addresses in hexadecimal. These tell us that `0x1149` is the start address of the function. In the middle, also in hexadecimal, are the bytes stored at those locations. On the right side are the assembly instructions corresponding to those bytes. Here, we can see that the call to `puts` starts at address `0x115b` ❶.

So, can we just set a breakpoint on `0x115b`? Unfortunately, it's a bit more complicated than that.

Position-Independent Executables

Today, many compilers produce *position-independent executables (PIEs)* by default. These executables don't expect to be loaded at a specific memory address; they can be loaded anywhere and still work. As such, memory addresses within PIEs aren't absolute virtual addresses, they're offsets from the start of the final load address of the binary. In other words, `0x115b` doesn't mean "virtual address `0x115b`," it means "`0x115b` bytes away from where the binary was loaded." We'll refer to these addresses as *file addresses*. We'll also sometimes need to refer to offsets from the start of the object file on disk, which we'll call *file offsets*. File offsets and file addresses are not necessarily the same, because different parts of the ELF file may be mapped into different areas of virtual memory.

PIEs exist to support *address space layout randomization (ASLR)*, a protection against malicious code that takes advantage of known virtual addresses to attack programs. By randomizing the virtual addresses at which a program

is loaded, the system keeps attackers from making predictions that could help them compromise it.

You can check whether a given executable is a PIE by running *file* on it from a command line. For my *hello_sdb* executable, I get this:

```
$ file test/targets/hello_sdb
test/targets/hello_sdb: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=8a16c5368b08a5763a9820dd330d8817bd70bcb6, for GNU/Linux 3.2.0,
with debug_info, not stripped
```

The important part here is "ELF 64-bit LSB pie executable," which tells us we're dealing with a PIE. Your version of the *file* program may instead say "ELF 64-bit LSB shared object," which is equivalent.

To deal with PIEs in the debugger, we have two options. First, we could compile our test programs with `-no-pie` to force them to commit to absolute load addresses. Second, we could parse information in the procfs to find the real load address of the executable.

Option one isn't very appealing, as it would limit the set of programs the debugger could debug. Option two, or some equivalent of it, is the choice that production debuggers take. We'll also disable ASLR for processes we launch ourselves so the addresses remain stable between program runs, making it easier to write tests.

Disabling ASLR is quite easy. You can disable ASLR globally by running **`echo 0 > /proc/sys/kernel/randomize_va_space`** on a command line, but it's better to do it on a per-process basis. The `personality` syscall allows you to change some execution properties for processes, like whether they're limited to 32-bit addresses or whether ASLR is enabled. We can call this right after we call `fork` in `sdb::process::launch` in *sdb/src/process.cpp*:

```
#include <sys/personality.h>

std::unique_ptr<process> process::launch
(std::filesystem::path path, bool debug) {
    --snip--
    if (pid == 0) {
        personality(ADDR_NO_RANDOMIZE);
        --snip--
    }
    --snip--
}
```

Passing the `ADDR_NO_RANDOMIZE` option to `personality` disables address space randomization for the current process. Be sure to call this function in the child process, where the PID returned by `fork` is 0.

We can find information about where the system loaded programs in memory at */proc/<pid>/maps*. For me, the start of this file looks like this when running with ASLR enabled:

```
556e2e531000-556e2e532000 r--p 00000000 08:10 129448  /path/to/run_endlessly
556e2e532000-556e2e533000 r-xp 00001000 08:10 129448  /path/to/run_endlessly
556e2e533000-556e2e534000 r--p 00002000 08:10 129448  /path/to/run_endlessly
556e2e534000-556e2e535000 r--p 00002000 08:10 129448  /path/to/run_endlessly
556e2e535000-556e2e536000 rw-p 00003000 08:10 129448  /path/to/run_endlessly
```

Each line is a region of mapped memory, and they correspond to different segments of the object file. The first part is the address range, followed by the read/write/execute permissions of the region. Next comes the file offset of the segment, some more detailed information about where the file is on disk, and the path to the file. From the permissions, you can kind of guess what the different segments are. The second one, with executable (`x`) permissions, is the code. The last segment, with writable (`w`) permissions, contains non-constant global variables.

Due to ASLR, running the program again produces different addresses. However, if you turn off ASLR, you should get the same addresses every time you run the program. This stability makes the binary much easier to deal with when testing the debugger.

We'll add one more bit of help before we try to actually set a breakpoint: making *sdb* print the PIDs of programs it launches. In *sdb/tools/sdb.cpp*, add the following:

```
std::unique_ptr<sdb::process> attach(int argc, const char** argv) {
    --snip--
    // Passing program name
    else {
        auto program_path = argv[1];
        auto proc = sdb::process::launch(program_path);
        fmt::print("Launched process with PID {}\n", proc->pid());
        return proc;
    }
}
```

We're finally ready to test whether the breakpoint code works.

Breakpoint Tests

Let's use the debugger to set a breakpoint on the "Hello, world" program.

The algorithm for finding where to set a breakpoint depends on whether the program is compiled as a PIE or a non-PIE. The PIE case is a bit more involved, but we'll follow that algorithm here. We'll force all test executables to compile as PIEs, in case your compiler doesn't do this by default. We'll also force them to compile without optimizations and with debug information, even in release CMake builds, as we'll need this information when we start

parsing DWARF in Chapter 12. We can do this by adding CMake functions in *sdb/test/targets/CMakeLists.txt*. Rewrite this file like so:

```
❶ function(add_test_cpp_target name)
      add_executable(${name} "${name}.cpp")
      target_compile_options(${name} PRIVATE -g -O0 -pie)
   ❷ add_dependencies(tests ${name})
  endfunction()

  function(add_test_asm_target name)
      add_executable(${name} "${name}.s")
      target_compile_options(${name} PRIVATE -pie)
      add_dependencies(tests ${name})
  endfunction()

  add_test_cpp_target(run_endlessly)
  add_test_cpp_target(end_immediately)
  add_test_cpp_target(hello_sdb)

  add_test_asm_target(reg_write)
  add_test_asm_target(reg_read)
```

First, we define a CMake function with the name `add_test_cpp_target` ❶, which takes a single parameter called `name`. It calls `add_executable` to create a target with the given name that has a single source file whose path is the given name plus *.cpp*. We add three flags to the target compile options: `-g` for generating debug info, `-O0` for disabling optimizations (note the capital letter O followed by the number 0), and `-pie` for generating a PIE. We also call `add_dependencies` to ensure that whenever we build the `tests` target, we also build the test programs ❷.

The `add_test_asm_target` function is much the same, but it affects assembly sources. We use the `.s` suffix when creating the executable. The `-g` and `-O0` flags aren't necessary because we won't use debug information for assembly code and the assembler doesn't carry out optimizations. Finally, we call `add_test_cpp_target` or `add_test_asm_target` for each of the existing test targets to build them.

Now that we've built the test programs, we move to the next step of setting a breakpoint: finding the address on which to set it. We can find the address of the `call` instruction in the `main` function of *hello_sdb* using **`objdump -d hello_sdb`**. This is a file address rather than a virtual address because we built the program as a PIE. In my case, the file address is `0x115b`:

```
115b:       e8 f0 fe ff ff          call   1050 <puts@plt>
```

For me, the `.text` section is mapped into memory at the same offset at which it exists in the ELF file. We can see this with the `readelf` utility:

```
$ readelf -S test/targets/hello_sdb
There are 36 section headers, starting at offset 0x4558:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
--snip--
  [16] .text             PROGBITS         0000000000001060  00001060
       0000000000000107  0000000000000000  AX       0     0     16
--snip--
```

Because the `Address` and `Offset` fields are the same, file addresses and file offsets are equivalent for the `.text` section. If they're different for your executable, you'll need to subtract the file offset from the file address to calculate the *section load bias* for that section and subtract this load bias from the file address of the `call` instruction to calculate where that instruction lives inside the ELF file.

We'll use the procfs to find the load address of the instruction. Run *sdb* on *hello_sdb*. It should launch the program, halt it on entry, and print the PID of the inferior:

```
$ tools/sdb test/targets/hello_sdb
Launched process with PID 11786
sdb>
```

Open a new terminal or press CTRL-Z to put *sdb* in the background. Read the memory map file for the process in the procfs:

```
$ cat /proc/11786/maps
555555554000-555555555000  r--p 00000000 08:10 129448  /path/to/hello_sdb
555555555000-555555556000  r-xp 00001000 08:10 129448  /path/to/hello_sdb
555555556000-555555557000  r--p 00002000 08:10 129448  /path/to/hello_sdb
555555557000-555555558000  r--p 00002000 08:10 129448  /path/to/hello_sdb
555555558000-555555559000  rw-p 00003000 08:10 129448  /path/to/hello_sdb
```

We're looking for an instruction, and we know it will be in the second entry because it's the only one marked as executable (`x`). However, you could also identify it by looking at the file offset in column three and the mapped ranges in column one.

The second entry corresponds to file offset `0x1000`. By subtracting the low range (`0x555555555000`) from the high range (`0x555555556000`), we can see that it spans `0x1000` bytes. This means that it covers the file offset regions `0x1000` through `0x2000`, and the address we're looking for (`0x115b`) is within that range.

We calculate the load address of the instruction by adding the load address of the segment to the instruction's file address, then subtracting the segment's start file address. In this case, this is `0x115b + 0x555555555000 - 0x1000 = 0x55555555515b`. This is where we need to set the breakpoint.

Now set the breakpoint and continue the program:

```
sdb> break set 0x55555555515b
sdb> c
Process 344 stopped with signal TRAP at 0x55555555515c
sdb>
```

If this worked correctly, you should see no output because the program should have stopped before the call. Congratulations! You've just set a breakpoint.

Continuing

You might notice that if you continue from the breakpoint, you hit an illegal instruction. Don't worry, you're not in trouble, it's just because the CPU executed the `int3` instruction and is now trying to execute the remainder of the bytes of the instruction we scribbled over. These bytes don't form a valid instruction, so the operating system sends the process a `SIGILL`.

We could bypass this issue by disabling the breakpoint, moving the program counter back 1 byte, and then continuing, but this would cause problems if we should hit the same breakpoint more than once. Instead, the solution is to set the program counter back by 1 byte if we stop at a breakpoint. Then, to resume the process, we can disable the breakpoint, step over a single instruction, re-enable the breakpoint, and resume.

In order to move the program counter back, we should add another little helper function to *sdb/include/libsdb/process.hpp* that allows us to set the program counter:

```
namespace sdb {
    class process {
    public:
        --snip--
        void set_pc(virt_addr address) {
            get_registers().write_by_id(register_id::rip, address.addr());
        }
        --snip--
    };
}
```

This function writes the given address to the `rip` register.

Next, if the process stopped due to a `SIGTRAP` and the address 1 byte below the program counter is an enabled breakpoint, we should fix up the program counter to point to the breakpoint. Add the following to `wait_on_signal` in *sdb/src/process.cpp*:

```
    --snip--
    if (is_attached_ and state_ == process_state::stopped) {
        read_all_registers();
```

```
        auto instr_begin = get_pc() - 1;
        if (reason.info == SIGTRAP and
            breakpoint_sites_.enabled_stoppoint_at_address(instr_begin)) {
            set_pc(instr_begin);
        }
    }
    --snip--
```

Finally, when we resume the process, if the process is currently stopped at a breakpoint, we should step over the breakpoint. Add the following code to the beginning of *sdb/src/process.cpp*:

```
void sdb::process::resume() {
    auto pc = get_pc();
    if (breakpoint_sites_.enabled_stoppoint_at_address(pc)) {
        auto& bp = breakpoint_sites_.get_by_address(pc);
        bp.disable();
        if (ptrace(PTRACE_SINGLESTEP, pid_, nullptr, nullptr) < 0) {
            error::send_errno("Failed to single step");
        }
        int wait_status;
        if (waitpid(pid_, &wait_status, 0) < 0) {
            error::send_errno("waitpid failed");
        }
        bp.enable();
    }
    --snip--
}
```

First, we check if we're at an enabled breakpoint. If so, we disable it and use `PTRACE_SINGLESTEP` to execute a single instruction. We call `waitpid` to wait until the inferior has executed the instruction and halted, and then we re-enable the breakpoint before continuing the process and setting the state to running.

Now you should be able to set the same breakpoint you did in your initial test, hit it, continue, and see the process print `Hello, sdb!` and then exit:

```
$ tools/sdb test/targets/hello_sdb
Launched process with PID 1408
sdb> break set 0x55555555515b
sdb> c
Process 1408 stopped with signal TRAP at 0x55555555515b
sdb> c
Hello, sdb!
Process 1408 exited with status 0
sdb>
```

As we've just discovered `PTRACE_SINGLESTEP`, let's also use it to add support for stepping over machine instructions to the debugger's command line. Add a `step_instruction` function to *sdb/include/libsdb/process.hpp*:

```
namespace sdb {
    class process {
    public:
        --snip--
        sdb::stop_reason step_instruction();
        --snip--
    };
}
```

The function returns a stop reason that describes why the process halted after stepping. For example, it could step directly into a breakpoint, in which case the breakpoint stop should be returned.

Implement the function in *sdb/src/process.cpp*. If a breakpoint is enabled at the program counter, we should disable it before stepping and enable it afterward:

```
sdb::stop_reason sdb::process::step_instruction() {
    std::optional<breakpoint_site*> to_reenable;
    auto pc = get_pc();
    if (breakpoint_sites_.enabled_stoppoint_at_address(pc)) {
     ❶ auto& bp = breakpoint_sites_.get_by_address(pc);
        bp.disable();
        to_reenable = &bp;
    }

    if (ptrace(PTRACE_SINGLESTEP, pid_, nullptr, nullptr) < 0) {
        error::send_errno("Could not single step");
    }
    auto reason = wait_on_signal();

    if (to_reenable) {
        to_reenable.value()->enable();
    }
    return reason;
}
```

This function is very similar to the code we wrote for continuing the process. We declare a `std::optional<breakpoint_site*>` to track the breakpoint site at which the process is currently stopped, if there is one. If there is indeed a breakpoint at the current program counter location, we retrieve it, disable it, and store a pointer to it for re-enabling later. Note that we must declare `bp` as a reference ❶ so that we can safely store a pointer to it in a variable declared in the enclosing scope.

After potentially disabling a breakpoint, we call `PTRACE_SINGLESTEP` to step over an instruction and `wait_on_signal` to wait until that is completed and

retrieve a reason for stopping. If there is a breakpoint to enable, we enable it, then finally return the stop reason to the caller.

Next, we'll add a stepping command. We'll write a simple `step` command for now and expand on it when we add support for stepping at the source code level in Chapter 14. The command should print the stop reason, just like `continue`. In *sdb/tools/sdb.cpp*, add the following:

```
namespace {
    void handle_command(std::unique_ptr<sdb::process>& process,
          std::string_view line) {
        --snip--
        else if (is_prefix(command, "step")) {
            auto reason = process->step_instruction();
            print_stop_reason(*process, reason);
        }
        --snip--
    }
}
```

We add a "step" branch to `handle_command` that steps over an instruction, then prints the stop reason. We'll also add some help text to `print_help`:

```
            std::cerr << R"(Available commands:
    breakpoint  - Commands for operating on breakpoints
    continue    - Resume the process
    register    - Commands for operating on registers
    step        - Step over a single instruction
)";
```

Let's move on to testing.

Automated Testing

We performed a lot of manual work to derive the correct address on which to set a breakpoint. We should find a way to automate this process so we can include it in our test suite. We want to implement the following algorithm:

1. Find the file address at which we'd like to set a breakpoint.
2. Find the section load bias for the section that contains that file address.
3. Convert the breakpoint file address to a file offset using the section load bias.
4. Calculate the load address of that file offset in the current process.
5. Set a breakpoint on the load address.
6. Continue, and ensure that the process stops at the breakpoint.
7. Continue again, and ensure the process exits with the desired output.

Let's start by implementing a function for finding the section load bias for a given file address. In Chapter 11, we'll deal with file addresses, file offsets, and virtual addresses in a more robust way, but for now, we'll just parse the output of `readelf`. Add the following to *sdb/test/tests.cpp*:

```
namespace {
    std::int64_t get_section_load_bias(
          std::filesystem::path path, Elf64_Addr file_address) {
        auto command = std::string("readelf -WS ") + path.string();
        auto pipe = popen(command.c_str(), "r");

        std::regex text_regex(R"(PROGBITS\s+(\w+)\s+(\w+)\s+(\w+))");
        char* line = nullptr;
        std::size_t len = 0;
        while (getline(&line, &len, pipe) != -1) {
            std::cmatch groups;
            if (std::regex_search(line, groups, text_regex)) {
                auto address = std::stol(groups[1], nullptr, 16);
                auto offset = std::stol(groups[2], nullptr, 16);
                auto size = std::stol(groups[3], nullptr, 16);
                if (address <= file_address and
                    file_address < (address + size)) {
                    free(line);
                    pclose(pipe);
                    return address - offset;
                }
            }
            free(line);
            line = nullptr;
        }
        pclose(pipe);
        sdb::error::send("Could not find section load bias");
    }
}
```

The command we'll execute is `readelf -WS <program name>`, which prints out the section headers with the entry for each section on a single line. We use `popen` to run this command and open a pipe from which we can read its output. We create a regular expression, which I'll explain in detail shortly, that captures the file address, file offset, and size of the section. We then loop over the lines of the process output using the `getline` function from the C standard library, because the C++ one operates on streams rather than file descriptors. This function dynamically allocates a string that stores a line and stores it in the first argument. We match our regular expression against this line, and if it matches, we parse the address, offset, and size of the section into integers. If the given file address lies in the range of the section address plus the section size, we clean up and return the section load bias. If not, we free the line and set `line` to `nullptr`, because otherwise `getline` will try

to reuse the freed memory. If we don't find a matching section, we throw an exception.

Let's look at that regular expression. This is the text that we're trying to match:

```
[16] .text             PROGBITS        0000000000001060 001060 000107 00  AX  0   0 16
```

The `PROGBITS` part of the regular expression matches that string in the line. The `\s+` parts match one or more pieces of whitespace. The `(\w+)` parts match one or more alphanumeric characters and capture them in a capture group, due to the enclosing parentheses. As such, the three capture groups will capture the three whitespace-separated numbers that succeed `PROGBITS`.

Now we can find the file address at which to set the breakpoint in the binary. Rather than automating disassembling the machine code and locating the `call` instruction, we'll get the file address of the binary's *entry point*: the function called at the start of the program. On Linux, this is called `_start`, and it does some environment setup before eventually calling the `main` function that the user defines. Once we have the file address, we'll subtract the section load bias to get the file offset of the entry point. Again, in Chapter 11, we'll write our own ELF parser that can retrieve the entry point, but for now, we can write a small procedure that retrieves the entry point file address using the *elf.h* header that comes with Linux. Add the following to *sdb/test/tests.cpp*:

```
#include <elf.h>

namespace {
    std::int64_t get_entry_point_offset(std::filesystem::path path) {
        std::ifstream elf_file(path);

        Elf64_Ehdr header;
        elf_file.read(reinterpret_cast<char*>(&header), sizeof(header));

        auto entry_file_address = header.e_entry;
        auto load_bias = get_section_load_bias(path, entry_file_address);
        return entry_file_address - load_bias;
    }
}
```

First, we open the given file and ready it for reading by creating a `std::ifstream`. The `Elf64_Ehdr` type defines the binary format for the ELF header, which gives metadata about the object file, such as its entry point. We declare an instance of one and then read the data from the beginning of the object file into it using the `std::istream::read` function. Note that we first need to cast the address of `header` into a `char*` because this is what the `read` function expects. The `e_entry` field gives us the entry point file address for the object file. We subtract the section load bias for the relevant section to compute the entry point file offset and return it.

So far, so good. To get the load address, we can do some simple parsing of the */proc/<pid>/maps* file. Refer to "Position-Independent Executables" on page 151 if you need a refresher on what this looks like.

In Chapter 11, we'll implement a more robust version of this parsing, but for now, we can write a brittle solution that gets just the information we want. We'll rely on the fact that our binary's entries will appear first in the file and that they have only a single segment marked as executable:

```
#include <regex>

namespace {
    virt_addr get_load_address(pid_t pid, std::int64_t offset) {
        std::ifstream maps("/proc/" + std::to_string(pid) + "/maps");
     ❶ std::regex map_regex (R"((\w+)-\w+ ..(.). (\w+))");

        std::string data;
        while (std::getline(maps, data)) {
            std::smatch groups;
            std::regex_search(data, groups, map_regex);

            if (groups[2] == 'x') {
                auto low_range = std::stol(groups[1], nullptr, 16);
                auto file_offset = std::stol(groups[3], nullptr, 16);
                return virt_addr(offset - file_offset + low_range);
            }
        }
        sdb::error::send("Could not find load address");
    }
}
```

We open the memory map file for the given PID as a `std::ifstream` and create a regular expression (which I'll explain in detail soon) that extracts the low address range, the executable flag, and the file offset from the line of text. We then use `std::getline` to loop over the lines of text in that file. We call the `std::regex_search` function to match the line against the regex we wrote and populate `groups` with the extracted data. We're looking for the line that is marked as executable, so the second matched group should be `x` rather than a dot. If so, we calculate the load address of the file offset by subtracting the file offset of the loaded segment within the original object file and adding the low virtual address of the mapped segment.

Again, let's break down the regex ❶. We want to match this part of the text:

```
555555555000-555555556000  r-xp 00001000
```

We handle the low range with `(\w+)`. Next is `-`, which just matches the `-` in the text. The sequence `\w+` matches one or more characters but doesn't capture it, as we don't care about this part of the text. After this, `..(.)`.

matches four characters, capturing the third one. This extracts whether the segment is executable. Finally, we capture the file offset with (\w+) again.

Now we can hook the get_load_address function into our test. We'll set a breakpoint on entry, continue, and make sure it's hit. Then we'll continue from the breakpoint and make sure that the process exited normally and printed "Hello, sdb!":

```
TEST_CASE("Breakpoint on address works", "[breakpoint]") {
    bool close_on_exec = false;
    sdb::pipe channel(close_on_exec);

    auto proc = process::launch("targets/hello_sdb", true, channel.get_write());
    channel.close_write();

    auto offset = get_entry_point_offset("targets/hello_sdb");
    auto load_address = get_load_address(proc->pid(), offset);

    proc->create_breakpoint_site(load_address).enable();
    proc->resume();
    auto reason = proc->wait_on_signal();

    REQUIRE(reason.reason == process_state::stopped);
    REQUIRE(reason.info == SIGTRAP);
    REQUIRE(proc->get_pc() == load_address);

    proc->resume();
    reason = proc->wait_on_signal();

    REQUIRE(reason.reason == process_state::exited);
    REQUIRE(reason.info == 0);

    auto data = channel.read();
    REQUIRE(to_string_view(data) == "Hello, sdb!\n");
}
```

We create a pipe so we can read the output of the program, then launch the process. Next, we get the file offset of the entry point for *hello_sdb* from the ELF file and calculate its load address in the current process. We then set a breakpoint on that load address and resume. This should cause the process to stop due to a SIGTRAP, with the program counter set to the address on which we enabled a breakpoint, so we check for all of those cases. Finally, we resume again and ensure the process exited and printed out "Hello, sdb!"

Let's also test that we can remove breakpoint sites from the process's collection:

```
TEST_CASE("Can remove breakpoint sites", "[breakpoint]") {
    auto proc = process::launch("targets/run_endlessly");
```

```
    auto& site = proc->create_breakpoint_site(virt_addr{ 42 });
    proc->create_breakpoint_site(virt_addr{ 43 });
    REQUIRE(proc->breakpoint_sites().size() == 2);

    proc->breakpoint_sites().remove_by_id(site.id());
    proc->breakpoint_sites().remove_by_address(virt_addr{ 43 });
    REQUIRE(proc->breakpoint_sites().empty());
}
```

We create two breakpoint sites and then remove them using the two `remove_by_*` functions, ensuring that we end up with an empty list. Run the tests; they should pass.

Summary

In this chapter, you learned how software breakpoints work. You extended the debugger library with a type (`sdb::breakpoint_site`) for representing physical breakpoints and added management functions to the `sdb::process` type. To `sdb::breakpoint_site`, you added the power of setting and disabling software breakpoints.

You also learned about position-independent executables (PIEs) and address space layout randomization (ASLR), and how they can make locating the virtual addresses of a loaded program difficult. In spite of these challenges, you implemented an algorithm for locating a binary instruction inside of a loaded program and setting a breakpoint on it. You automated this process and added it to your test suite. You also added support for single-stepping over single machine instructions.

In the next chapter, you'll build upon this work by adding support for reading and writing memory and disassembling the code of the inferior.

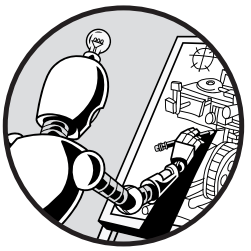
Check Your Knowledge

1. What is the difference between a hardware breakpoint and a software breakpoint? What are the benefits of each?
2. Which `ptrace` commands can we use for reading and writing memory?
3. What instruction triggers software breakpoints on x86?
4. How can we check whether an executable is position-independent?
5. How can we disable ASLR?
6. Which part of the procfs contains information about memory maps for a process?

8

MEMORY AND DISASSEMBLY

. . . and I wrote to the waves
words that vanished
with the tide
to be forgotten.



Memory corruption errors are one of the trickiest issues to debug in software. A memory write can invalidate a data structure without crashing the program, only for the corruption to hide in the shadows until it causes unexpected behavior a few thousand function calls later.

Debuggers usually provide a means to inspect and modify the memory of the inferior process so the user can track down these sneaky problems. In this chapter, you'll add commands to your debugger for achieving this.

By reading the memory of the code that is currently being executed, the debugger can also show the user a representation of the code. Most people won't understand the instructions by looking at the raw memory dump, however, so debuggers must first *disassemble* the machine code, turning it back into human-readable assembly, before showing it to the user. You'll add disassembly support to your debugger in the memory manipulation features you implement.

Memory Operations

When we implemented breakpoint site enabling in Chapter 7, we read from and wrote to memory with `PTRACE_PEEKDATA` and `PTRACE_POKEDATA`. To interact with memory, we could just write wrappers around these functions and expose them to the user. However, these operations work on 64 bits at a time, so if we wanted to read a large amount of data out of the process (maybe to visualize some large data structure), we would have to make a lot of system calls, meaning many trips to and from kernel space, causing bad performance.

A better option might be to use the `process_vm_readv` and `process_vm_writev` syscalls and the */proc/pid>/mem* file, both of which support reading and writing larger chunks. Unfortunately, the memory gods aren't on our side: `process_vm_writev` doesn't support writing to protected areas of memory like code segments, and WSL has poor support for */proc/<pid>/mem*. So, we'll use `process_vm_readv` to read memory and grit our teeth while using `ptrace` to write memory one word at a time.

Reading and Writing

Let's add two new member functions to `sdb::process` for reading and writing memory. For reading, we should take a virtual address and an amount to read and return a `std::vector<std::byte>` that stores the requested memory. For writing, we could take an address and a `std::vector<std::byte>`, but using `std::vector` here might force additional memory allocations where they're not needed. As such, we'll write a small `span` type that represents a view of an existing region of memory. Add the following to *sdb/include/libsdb/types.hpp*:

```
#include <vector>
namespace sdb {
    template <class T>
    class span {
    public:
        span() = default;
        span(T* data, std::size_t size) : data_(data), size_(size) {}
        span(T* data, T* end) : data_(data), size_(end-data) {}
        template <class U>
        span(const std::vector<U>& vec) : data_(vec.data()), size_(vec.size()) {}


        T* begin() const { return data_; }
        T* end() const { return data_ + size_; }
        std::size_t size() const { return size_; }
        T& operator[](std::size_t n) { return *(data_ + n); }

    private:
        T* data_ = nullptr;
        std::size_t size_ = 0;
```

```
    };
}
```

The `span` type has four constructors: a default one that leaves the span empty, one that takes a start pointer and a size, one that takes start and end pointers, and one that takes a `std::vector`. We make the `std::vector` constructor a template with its own template parameter rather than using `std::vector<T>` because, for instance, it will be common for us to initialize an `sdb::span<const std::byte>` with a `std::vector<std::byte>`, and those types don't exactly match. The rest of the type is fairly straightforward; it stores the necessary information and allows users to retrieve the elements, start and end pointers, and size of the span.

Now we can declare `read_memory` and `write_memory` in *sdb/include/libsdb/process.hpp*:

```
namespace sdb {
    class process {
    public:
        --snip--
        std::vector<std::byte> read_memory(virt_addr address, std::size_t amount) const;
        void write_memory(virt_addr address, span<const std::byte> data);
        --snip--
    };
}
```

The `read_memory` function takes a virtual address to read from and a number of bytes to read and returns the requested memory in a `std::vector`. The `write_memory` function takes a virtual address to write to and a `span<const std::byte>` representing the data to write. The `const` part in `span<const std::byte>` ensures that we cannot accidentally modify the data to which the span is pointing.

Implement these in *sdb/src/process.cpp*, beginning with `read_memory`:

```
#include <sys/uio.h>

std::vector<std::byte>
sdb::process::read_memory(
      virt_addr address, std::size_t amount) const {
    std::vector<std::byte> ret(amount);

    iovec local_desc{ ret.data(), ret.size() };
    std::vector<iovec> remote_descs;
    while (amount > 0) {
        auto up_to_next_page = 0x1000 - (address.addr() & 0xfff);
        auto chunk_size = std::min(amount, up_to_next_page);
        remote_descs.push_back({ reinterpret_cast<void*>(address.addr()), chunk_size });
        amount -= chunk_size;
        address += chunk_size;
    }
```

```
    if (process_vm_readv(pid_, &local_desc, /*liovcnt=*/1,
        remote_descs.data(), /*riovcnt=*/remote_descs.size(), /*flags=*/0) < 0) {
        error::send_errno("Could not read process memory");
    }
    return ret;
}
```

We initialize a `std::vector` with enough space to fit the data we'll read. The `process_vm_readv` function performs the memory read and takes descriptions of the memory involved in the data transfer as arguments of type `iovec`. This type wraps a pointer to the buffer and the size of the buffer. The descriptor for the memory into which the data should be copied (the *local descriptor*) is simply the data and size of the `ret` vector.

The descriptor for the memory from which the data should be copied (the *remote descriptor*) is a bit more complicated. We want client code to be able to request large amounts of data, potentially more than is possible to read from the inferior, and have whatever data is accessible returned. For example, if we want to read a Linux filepath from the inferior process, this could be up to 4KiB in size, but maybe the path is only 12 bytes in size and directly after the path in memory is a page that is inaccessible to the inferior. We should still return those 12 valid bytes. The `process_vm_readv` function does support partial memory transfers, but if any part of the read for a single `iovec` descriptor fails, then no data is copied for that entire descriptor. As such, we create a vector of descriptors and split the range of data to be copied on memory page boundaries (assuming 4KiB pages, which is the default on x64). We pass this vector to `process_vm_readv`.

We also pass the PID of the process from which we want to read, the number of local and remote descriptors we're passing (one, in both cases), and an argument for additional flags. As none exist at the time of writing, we pass `0`. If `process_vm_readv` fails, it returns `-1` and sets `errno`, so we make sure to handle this potential error.

The `write_memory` function is a bit more complicated. Because `ptrace` can only write exactly 8 bytes at a time, we might accidentally overwrite some data if the amount we want to write isn't divisible by eight. In such cases, we'll need to read 8 bytes from the desired address, copy the data over the start of these bytes, then write them back:

```
#include <libsdb/bit.hpp>

void sdb::process::write_memory(
    virt_addr address, span<const std::byte> data) {
    std::size_t written = 0;
    while (written < data.size()) { ❶
        auto remaining = data.size() - written;
        std::uint64_t word;
        if (remaining >= 8) { ❷
            word = from_bytes<std::uint64_t>(data.begin() + written);
```

```
        }
        else { ❸
            auto read = read_memory(address + written, 8);
            auto word_data = reinterpret_cast<char*>(&word);
            std::memcpy(word_data, data.begin() + written, remaining);
            std::memcpy(word_data + remaining, read.data() + remaining, 8 - remaining);
        }
        if (ptrace(PTRACE_POKEDATA, pid_, address + written, word) < 0) {
            error::send_errno("Failed to write memory");
        }
        written += 8;
    }
}
```

We loop until we've written all of the data that the caller gave us ❶. Each iteration of this loop writes 8 bytes to the inferior process. We calculate the amount of remaining data based on how much we've written so far and declare a `word` variable that will hold the data to be written on this iteration. If at least 8 bytes remain, we'll write the next 8 bytes from the start of the given buffer ❷. Otherwise, we need to handle the special case of doing a partial memory write ❸. For this, we read the 8 bytes we'll be writing to and cast a pointer to `word` to a `char*` so we can use `memcpy` on it. We then copy the remaining data into the start of `word_data`, followed by the bytes we're trying not to overwrite. Finally, we call `ptrace` with the `PTRACE_POKEDATA` request to write the next 8 bytes to the inferior, handle any errors, and record that we've written another 8 bytes, so that the loop now considers the next 8 bytes.

We'll also write a convenience helper called `read_memory_as` that reads a block of memory as an object of a given type. Add this code to *sdb/include/libsdb/process.hpp*:

```
#include <libsdb/bit.hpp>

namespace sdb {
    class process {
    public:
        --snip--
        template <class T>
        T read_memory_as(virt_addr address) const {
            auto data = read_memory(address, sizeof(T));
            return from_bytes<T>(data.data());
        }
        --snip--
    };
}
```

This function template has one template parameter that gives the type to which the memory should be converted. It reads an amount of memory

equal to the size of that type from the given address and then converts the resulting bytes to the given type. Because this is a template, we must implement it in the header rather than in the implementation file.

Exposing Memory to the User

Now we'll support memory operations on the command line. Add a new command to `handle_command` in *sdb/tools/sdb.cpp*:

```
--snip--
else if (is_prefix(command, "memory")) {
    handle_memory_command(*process, args);
}
--snip--
```

We'll support three subcommands: `memory read` *`<address>`*, to read an unspecified amount of data from that address (we'll default to 32 bytes); `memory read` *`<address> <number of bytes>`*, to read the specified number of bytes from that address; and `memory write` *`<address> <values>`*, to write the given values to the given address.

Implement `handle_memory_command` as a simple wrapper around the functions that will do the actual reading and writing:

```
namespace {
    void handle_memory_command(
          sdb::process& process,
          const std::vector<std::string>& args) {
        if (args.size() < 3) {
            print_help({ "help", "memory" });
            return;
        }
        if (is_prefix(args[1], "read")) {
            handle_memory_read_command(process, args);
        }
        else if (is_prefix(args[1], "write")) {
            handle_memory_write_command(process, args);
        }
        else {
            print_help({ "help", "memory" });
        }
    }
}
```

As usual, we call `print_help` if the user supplies unexpected arguments. Otherwise, we forward the arguments to one of the `handle_memory_read_command` or `handle_memory_write_command` functions, which we'll write next.

Let's start with `handle_memory_read_command`. Implementing this function requires many steps, so we'll do it piece by piece, beginning with argument parsing:

```
namespace {
    void handle_memory_read_command(
          sdb::process& process,
          const std::vector<std::string>& args) {
        auto address = sdb::to_integral<std::uint64_t>(args[2], 16);
        if (!address) sdb::error::send("Invalid address format");

        auto n_bytes = 32;
        if (args.size() == 4) {
            auto bytes_arg = sdb::to_integral<std::size_t>(args[3]);
            if (!bytes_arg) sdb::error::send("Invalid number of bytes");
            n_bytes = *bytes_arg;
        }
```

We parse the address the user wants to read from, throwing an error if this fails. We then try to parse the number of bytes the user wants to read. If they don't specify a number, we default to 32. To perform the data read, we call into `sdb::process::read_memory`:

```
        --snip--
        auto data = process.read_memory(sdb::virt_addr{ *address }, n_bytes);
```

Finally, we print out the memory. We'll print the data in batches of 16 bytes in hexadecimal, like this:

```
sdb> mem read 0x0000555555555515b
0x0055555555515b: cc f0 fe ff ff b8 00 00 00 00 5d c3 00 f3 0f 1e
0x0055555555516b: fa 48 83 ec 08 48 83 c4 08 c3 00 00 00 00 00 00
```

We write a loop that batches up the memory and uses `fmt::print` to write the batches in the desired format:

```
        --snip--
        for (std::size_t i = 0; i < data.size(); i += 16) {
            auto start = data.begin() + i;
            auto end = data.begin() + std::min(i + 16, data.size());
            fmt::print("{:#016x}: {:02x}\n",
                *address + i, fmt::join(start, end, " "));
        }
    }
}
```

We loop over the data 16 bytes at a time and compute start and end locations for the range we'll print. When computing the end of the data, we offset the start of the data by the smaller of `i+16` and the total data size. This ensures that `end` doesn't point past the end of the data if the number of bytes isn't divisible by 16. We then call `fmt::print`. The `{:#016x}` specifier prints the address in hexadecimal with a leading `0x` and pads it to 16 characters. The

{:02x} specifier prints the value of each byte in hexadecimal without a leading 0x and pads it to two characters. Lastly, fmt::join causes that second specifier to apply to every element of the given range with the given delimiter (in our case, a blank space) between them.

Now we implement handle_memory_write_command. When writing memory, we'll expect input in the form of hexadecimal values, comma-separated and surrounded in square brackets, as in mem write 0x555555555156 [0xff,0xff]. This is essentially the approach of the parse_vector function, with one key difference: we should keep reading values until we find a] character and then return a std::vector<std::byte>. Let's write a parse_vector overload that does this in *sdb/libsdb/include/parse.hpp*:

```
namespace sdb {
    inline auto parse_vector(
          std::string_view text) {
        auto invalid = [] { sdb::error::send("Invalid format"); };

        std::vector<std::byte> bytes;
        const char* c = text.data();

        if (*c++ != '[') invalid();

        while (*c != ']') {
            auto byte = sdb::to_integral<std::byte>({ c, 4 }, 16);
            bytes.push_back(byte.value());
            c += 4;

            if (*c == ',') ++c;
            else if (*c != ']') invalid();
        }

        if (++c != text.end()) invalid();

        return bytes;
    }
}
```

We define a simple lambda that we can call to throw an error in a concise way. We then define a vector to hold the parsed bytes and grab the first character of the data. If it's not an opening square bracket, the input is invalid, so we throw an error. After this check, we loop, reading bytes and commas until we hit the closing square bracket. On each iteration, we parse four characters (for example, 0xff) as a byte, move the c pointer past this data, and then ensure that we have either a comma or a closing square bracket. We check that we've consumed all of the text in the argument and then return the parsed bytes.

The `handle_memory_write_command` function then parses the input and calls `process.write_memory`:

```
namespace {
    void handle_memory_write_command(
          sdb::process& process,
          const std::vector<std::string>& args) {
        if (args.size() != 4) {
            print_help({ "help", "memory" });
            return;
        }

        auto address = sdb::to_integral<std::uint64_t>(args[2], 16);
        if (!address) sdb::error::send("Invalid address format");

        auto data = sdb::parse_vector(args[3]);
        process.write_memory(
            sdb::virt_addr{ *address }, { data.data(), data.size() });
    }
}
```

If the caller passed an incorrect number of arguments, we print a help message. Otherwise, we parse the address to write to and the vector of data, then call `process.write_memory` to write the data to the inferior.

Finally, we add some help text to `print_help`:

```
        --snip--
        if (args.size() == 1) {
            std::cerr << R"(Available commands:
    breakpoint  - Commands for operating on breakpoints
    continue    - Resume the process
    memory      - Commands for operating on memory
    register    - Commands for operating on registers
    step        - Step over a single instruction
)";
        }
        --snip--
        else if (is_prefix(args[1], "memory")) {
            std::cerr << R"(Available commands:
    read <address>
    read <address> <number of bytes>
    write <address> <bytes>
)";
        }
        --snip--
```

You can test these memory operations with the *reg_read* program. Launch the program, resume until it traps, and write some random bytes to the place

where the program counter is pointing. Then resume and watch the process terminate due to an illegal instruction:

```
$ tools/sdb test/targets/reg_read
Launched process with PID 24233
sdb> c
Process 22092 stopped with signal TRAP at 0x555555555156
sdb> mem read 0x555555555156
0x00555555555156: 41 b5 2a 48 c7 c0 3e 00 00 00 4c 89 e7 48 c7 c6
0x00555555555166: 05 00 00 00 0f 05 49 bd 11 ba 5e ba 00 00 00 00
sdb> mem write 0x555555555156 [0xff,0xff]
sdb> mem read 0x555555555156
0x00555555555156: ff ff 2a 48 c7 c0 3e 00 00 00 4c 89 e7 48 c7 c6
0x00555555555166: 05 00 00 00 0f 05 49 bd 11 ba 5e ba 00 00 00 00
sdb> c
Process 24233 stopped with signal ILL at 0x555555555156
sdb>
```

Bask in the glory of your heinous memory shenanigans for a minute, then get ready to write some tests.

Testing Memory Operations

Let's a write a small test application whose memory we can read and write. To test reads, we'll create a local integer variable with known contents and write the address of this variable to `stdout`. In the debugger, we'll then read the memory at that address and ensure that the value we get matches the value in the code. Create an *sdb/test/targets/memory.cpp* file with these contents:

```
#include <cstdio>
#include <sys/signal.h>
#include <unistd.h>

int main() {
    unsigned long long a = 0xcafecafe;
    auto a_address = &a;

    write(STDOUT_FILENO, &a_address, sizeof(void*));
    fflush(stdout);

    raise(SIGTRAP);
}
```

We define an integer variable with the value `0xcafecafe` and retrieve its address; then we write it to `stdout` with the `write` syscall. We flush the stream so the debugger will definitely get the message and trap to signal the debugger.

As usual, add this to the file *sdb/test/targets/CMakeLists.txt*:

```
--snip--
add_test_cpp_target(memory)
--snip--
```

Then, add a test to *sdb/test/tests.cpp*:

```
TEST_CASE("Reading and writing memory works", "[memory]") {
    bool close_on_exec = false;
    sdb::pipe channel(close_on_exec);
    auto proc = process::launch("targets/memory", true, channel.get_write());
    channel.close_write();

    proc->resume();
    proc->wait_on_signal();

    auto a_pointer = from_bytes<std::uint64_t>(channel.read().data());
    auto data_vec = proc->read_memory(virt_addr{ a_pointer }, 8);
    auto data = from_bytes<std::uint64_t>(data_vec.data());
    REQUIRE(data == 0xcafecafe);
}
```

We launch the test program and replace its `stdout` with a pipe so we can read the pointer the program prints. Next, we resume the process and wait for it to trap; then we parse the pointer the program printed to the pipe, read the data at the memory address given by that pointer, and ensure that it's the value we expect (`0xcafecafe`). This test should pass.

To test writes, we'll create a local character array, print the address of this array to `stdout`, and, in the debugger, write a string to this address. The test will print the string to `stdout` and make sure we receive the same string in the debugger.

Add the following to the existing *sdb/test/targets/memory.cpp* file:

```
int main() {
    --snip--
    char b[12] = { 0 };
    auto b_address = &b;
    write(STDOUT_FILENO, &b_address, sizeof(void*));
    fflush(stdout);
    raise(SIGTRAP);

    printf("%s", b);
}
```

We declare an array of zeros that the debugger will write into. As in the previous test, we print the address of this array, flush `stdout`, and then trap. Finally, we print the contents of the array, which the debugger will have written to before resuming the process.

Next, edit the test to carry out the rest of the steps required:

```
TEST_CASE("Reading and writing memory works", "[memory]") {
    --snip--
    proc->resume();
    proc->wait_on_signal();

    auto b_pointer = from_bytes<std::uint64_t>(channel.read().data());
    proc->write_memory(
        virt_addr{ b_pointer }, { as_bytes("Hello, sdb!"), 12 });

    proc->resume();
    proc->wait_on_signal();

    auto read = channel.read();
    REQUIRE(to_string_view(read) == "Hello, sdb!");
}
```

We resume the process and wait for it to trap after writing to `stdout`. We then read the address of the `b` variable from the pipe and write `Hello, sdb!` into that memory. Finally, we resume the process again and ensure it prints out the string.

If you run the tests, you should see this:

```
All tests passed (66 assertions in 17 test cases)
```

Congratulate yourself on successfully implementing so many features! Maybe throw yourself a little party. When you're done celebrating, it's time to move on to disassembly.

Disassembly

Now that we can read from memory and read the instruction pointer, it's possible to print out a textual representation of the assembly code of the inferior when the user stops at a breakpoint. We can feed the raw binary instructions, from the current instruction pointer onward, to a tool that decodes them, spitting out assembly code. We call such a tool a *disassembler*.

Writing an x64 disassembler is way out of scope for this book; the topic could fill an entire book itself. The instruction encoding for x64 is very complex and includes a huge number of instructions. If you're interested in the encoding format, you can read the "X86-64 Instruction Encoding" article on the OSDev.org wiki at *https://wiki.osdev.org/X86-64_Instruction_Encoding*.

This book uses the Zydis library for disassembly as it's very lightweight and also supports encoding instructions. Zydis has several versions with rather different interfaces; I'll use version 4. If you don't want to add any more dependencies to the project, you could instead use `libopcodes`, which comes with `binutils`. Be warned, however, that it's almost completely undocumented.

Add Zydis to *sdb/vcpkg.json*:

```
{
    "dependencies": ["libedit", "catch2", "fmt", "zydis"]
}
```

Find the package in *sdb/CMakeLists.txt*, underneath the call to `find_package` for `fmt`:

```
--snip--
find_package(fmt CONFIG REQUIRED)
find_package(zydis CONFIG REQUIRED)
--snip--
```

Then link against it in *sdb/src/CMakeList.txt* and add *disassembler.cpp* as a source file (as we'll be writing this file shortly):

```
add_library(libsdb ... disassembler.cpp)
target_link_libraries(libsdb PRIVATE Zydis::Zydis)
```

If you followed along with "Making the Dependencies Accessible" on page 5 and want to maintain the ability to install `libsdb` and allow dependent libraries to easily consume it through CMake, you'll need to add a CMake configuration file for `libsdb` and make Zydis a dependency. For details on this process, see the "Installing" chapter of *An Introduction to Modern CMake* (*https://cliutils.gitlab.io/modern-cmake/chapters/install/installing.html*).

Add *disassembler.cpp* to the existing `add_library` call and write a new call to `target_link_libraries` to link against Zydis. We won't expose any code from Zydis in the public headers; we need it only for building, so we use `PRIVATE` visibility.

We'll encapsulate disassembling in an `sdb::disassembler` type. Create a new *sdb/include/libsdb/disassembler.hpp* file with these contents:

```
#ifndef SDB_DISASSEMBLER_HPP
#define SDB_DISASSEMBLER_HPP

#include <libsdb/process.hpp>
#include <optional>

namespace sdb {
    class disassembler {
        struct instruction {
            virt_addr address;
            std::string text;
        };

    public:
        disassembler(process& proc) : process_(&proc) {}

        std::vector<instruction> disassemble(
```

```
            std::size_t n_instructions,
            std::optional<virt_addr> address = std::nullopt);

    private:
        process* process_;
    };
}

#endif
```

Our new `sdb::disassembler` type has a nested `instruction` type, which holds a string representation of the instruction and the memory address where the binary instruction it corresponds to is stored. The constructor takes a reference to an `sdb::process` and stores a pointer to it. The `disassemble` function takes a number of instructions to disassemble and, optionally, an address from which to disassemble; by default, it uses the current program counter value.

Implement `disassemble` in a new *sdb/src/disassembler.cpp* file. Begin the function by creating a `std::vector` with enough memory reserved for all of the instructions:

```
#include <Zydis/Zydis.h>
#include <libsdb/disassembler.hpp>

std::vector<sdb::disassembler::instruction> sdb::disassembler::disassemble(
      std::size_t n_instructions,
      std::optional<virt_addr> address) {
    std::vector<instruction> ret;
    ret.reserve(n_instructions);
```

If the user didn't supply an address to disassemble from, we'll default to the location where the program counter is pointing. We'll need to read a chunk of memory from that address, but x64 instructions aren't all the same size, so we don't know how much memory we'll have to read. We could choose to read a small chunk at a time, reading more if we run out of bytes to disassemble. However, the largest x64 instruction is 15 bytes, so if we read `n_instructions * 15`, we're guaranteed to have enough memory to disassemble `n_instructions`, so long as there are that many instructions left in memory:

```
    --snip--
    if (!address) {
        address.emplace(process_->get_pc());
    }
    auto code = process_->read_memory(*address, n_instructions * 15);
```

Then, until we either run out of instructions to decode or have decoded `n_instructions`, we decode another instruction and push it back to `ret`:

```
    --snip--
    ZyanUSize offset = 0;
    ZydisDisassembledInstruction instr;

    while (ZYAN_SUCCESS(ZydisDisassembleATT(
        ZYDIS_MACHINE_MODE_LONG_64, address->addr(),
        code.data() + offset, code.size() - offset, &instr))
        and n_instructions > 0)
    {
        ret.push_back(instruction{ *address, std::string(instr.text) });
        offset += instr.info.length;
        *address += instr.info.length;
        --n_instructions;
    }

    return ret;
}
```

The `ZYAN_SUCCESS` macro checks whether the call to `ZydisDisassembleATT` succeeded. We loop, calling `ZydisDisassembleATT` to populate `instr` with the disassembled instruction information. We then push back a new `sdb::disassembler::instruction` into the return vector, which records the address of the instruction as well as its text as a string. Finally, we update our bookkeeping variables.

Now we can expose this feature to users. We'll do this in two ways: by printing out the disassembly when the inferior stops and by providing an explicit `disassemble` command. In *sdb/tools/sdb.cpp*, add a `print_disassembly` function to print some code to the console:

```
#include <libsdb/disassembler.hpp>

namespace {
    void print_disassembly(sdb::process& process,
          sdb::virt_addr address, std::size_t n_instructions) {
        sdb::disassembler dis(process);
        auto instructions = dis.disassemble(n_instructions, address);
        for (auto& instr : instructions) {
         ❶ fmt::print("{:#018x}: {}\n", instr.address.addr(), instr.text);
        }
    }
}
```

We create a `disassembler` object, disassemble the requested number of instructions from the given address, and then print all of these instructions, along with their addresses. The `:#018x` specifier ❶ prints a hexadecimal representation of the address with enough padding to ensure that the output stays vertically aligned.

Both the `continue` and the `step` command should disassemble instructions after `wait_on_signal` returns. Let's factor this behavior into a `handle_stop` function:

```
namespace {
    void handle_stop(sdb::process& process, sdb::stop_reason reason) {
        print_stop_reason(process, reason);
        if (reason.reason == sdb::process_state::stopped) {
            print_disassembly(process, process.get_pc(), 5);
        }
    }
}
```

We print the stop reason and then, if the process stopped due to a signal, print five lines of disassembly, starting from the program counter. We call this function from the `continue` and `step` command handlers inside of `handle_command`:

```
        --snip--
        if (is_prefix(command, "continue")) {
            process->resume();
            auto reason = process->wait_on_signal();
            handle_stop(*process, reason);
        }
        else if (is_prefix(command, "step")) {
            auto reason = process->step_instruction();
            handle_stop(*process, reason);
        }
        --snip--
```

Then, we add a new command:

```
        --snip--
        else if (is_prefix(command, "disassemble")) {
            handle_disassemble_command(*process, args);
        }
        --snip--
```

We'll support two optional flags:

```
disassemble -c <n_instructions> -a <address>
```

We'll default these to five instructions and the current program counter value. Start implementing the `handle_disassemble_command` function with these defaults:

```
namespace {
    void handle_disassemble_command(
     sdb::process& process, const std::vector<std::string>& args) {
```

```
        auto address = process.get_pc();
        std::size_t n_instructions = 5;
```

We'll then loop over the rest of the arguments, filling in `address` and `n_instructions` with the user's specifications:

```
        --snip--
        auto it = args.begin() + 1;
        while (it != args.end()) {
            if (*it == "-a" and it + 1 != args.end()) {
                ++it;
                auto opt_addr = sdb::to_integral<std::uint64_t>(*it++, 16);
                if (!opt_addr) sdb::error::send("Invalid address format");
                address = sdb::virt_addr{ *opt_addr };
            }
            else if (*it == "-c" and it + 1 != args.end()) {
                ++it;
                auto opt_n = sdb::to_integral<std::size_t>(*it++);
                if (!opt_n) sdb::error::send("Invalid instruction count");
                n_instructions = *opt_n;
            }
            else {
                print_help({ "help", "disassemble" });
                return;
            }
        }
```

We loop over the arguments, checking which flag the user passed and parsing the relevant type. If we get an argument we weren't expecting or fail to parse an integer, we throw an exception. Next, we call into the disassembler:

```
        --snip--
        print_disassembly(process, address, n_instructions);
    }
```

Finally, we add help information for this command to `print_help`:

```
        --snip--
        if (args.size() == 1) {
            std::cerr << R"(Available commands:
    breakpoint  - Commands for operating on breakpoints
    continue    - Resume the process
    disassemble - Disassemble machine code to assembly
    memory      - Commands for operating on memory
    register    - Commands for operating on registers
    step        - Step over a single instruction
)";
        }
        --snip--
```

```
        else if (is_prefix(args[1], "disassemble")) {
            std::cerr << R"(Available options:
    -c <number of instructions>
    -a <start address>
)";
        }
        --snip--
```

If you test this new code with breakpoints, however, you may encounter a problem:

```
$ tools/sdb test/targets/hello_sdb
Launched process with PID 3479
sdb> break set 0x55555555515b
sdb> c
Process 3479 stopped with signal TRAP at 0x55555555515b
0x000055555555515b: int3
```

The only instruction the debugger has disassembled is the `int3` instruction used for breakpoints. Before we disassemble, we need to replace each of those `int3` instructions with the original byte. We could instead disable all of the breakpoints in the region before reading memory, but this is rather wasteful as it may incur many syscalls. It's better to first read the memory and then fix the bytes.

We'll add a function called `read_memory_without_traps` to `sdb::process`. Declare it in *sdb/include/libsdb/process.hpp*, next to the regular `read_memory` function:

```
        --snip--
        std::vector<std::byte> read_memory(
            virt_addr address, std::size_t amount) const;
        std::vector<std::byte> read_memory_without_traps(
            virt_addr address, std::size_t amount) const;
        --snip--
```

Implement the function in *sdb/src/process.cpp*. We'll pretend we already have a function called `sdb::stoppoint_collection::get_in_region` that gets us the list of stop points between two addresses:

```
std::vector<std::byte>
sdb::process::read_memory_without_traps(
      virt_addr address, std::size_t amount) const {
    auto memory = read_memory(address, amount);
    auto sites = breakpoint_sites_.get_in_region(
        address, address + amount);
    for (auto site : sites) {
        if (!site->is_enabled()) continue;
        auto offset = site->address() - address.addr();
        memory[offset.addr()] = site->saved_data_;
```

```
    }
    return memory;
}
```

We read the requested memory, then get a list of the breakpoint sites in that region. For each enabled site, we replace the `int3` instruction at the memory address on which the breakpoint is set with the address's original data.

Next, add the `get_in_region` function to *sdb/include/libsdb/stoppoint_collection.hpp*:

```
namespace sdb {
    template <class Stoppoint>
    class stoppoint_collection {
    public:
        --snip--
        std::vector<Stoppoint*> get_in_region(
            virt_addr low, virt_addr high) const;
        --snip--
    };
}
```

Implement the function in the same header:

```
namespace sdb {
    template <class Stoppoint>
    std::vector<Stoppoint*> stoppoint_collection<Stoppoint>::get_in_region(
          virt_addr low, virt_addr high) const {
        std::vector<Stoppoint*> ret;
        for (auto& site : stoppoints_) {
            if (site->in_range(low, high)) {
                ret.push_back(&*site);
            }
        }
        return ret;
    }
}
```

We loop over the stop points, pushing a pointer to points whose addresses fall in the given range into a vector, which we then return.

Finally, replace the call to `read_memory` in `sdb::disassembler::disassemble` with a call to `read_memory_without_traps`:

```
    --snip--
    auto code = process_->read_memory_without_traps(
        *address, n_instructions * 15);
    --snip--
```

Now, if you hit the same breakpoint you did earlier, the disassembler should pretend that the breakpoint isn't there and return the disassembly:

```
$ tools/sdb test/targets/hello_sdb
Launched process with PID 4077
sdb> break set 0x55555555515b
sdb> c
Process 4077 stopped with signal TRAP at 0x55555555515b
0x000055555555515b: call 0x0000555555555050
0x0000555555555160: mov $0x00, %eax
0x0000555555555165: pop %rbp
0x0000555555555166: ret
0x0000555555555167: add %dh, %bl
```

Sometimes, lying to users is okay.

Summary

In this chapter, you augmented the debugger with the ability to read and write memory and disassemble machine instructions. In the next chapter, you'll implement hardware breakpoints and data watchpoints.

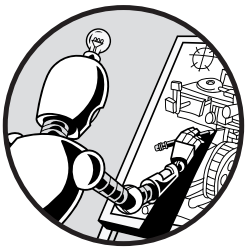
Check Your Knowledge

1. What are the disadvantages of the `PTRACE_PEEKDATA` and `PTRACE_POKEDATA` commands compared to other options for reading and writing memory?
2. What is the job of a disassembler?
3. What is the size of the largest x64 instruction, in bytes?

9

HARDWARE BREAKPOINTS AND WATCHPOINTS

Looking out from the lighthouse
I see water crash on the rocks
and breathe the salt air
as time stops.



In Chapter 7, you learned about software breakpoints, which we create by replacing instructions in the currently executing code with traps. Software breakpoints have a cousin that lives even closer to your CPU: hardware breakpoints, which we set by writing values directly into registers. In this chapter, you'll learn how x64 manages hardware breakpoints and add support for them to your debugger.

We can repurpose the same registers used for creating hardware breakpoints to create *watchpoints*, which are like breakpoints but get triggered when a specific address is read from or written to rather than executed. This chapter also walks you through adding support for watchpoints to your debugger.

Debug Registers

On x64, setting and tracing hardware breakpoints requires interfacing with the debug registers, which are named `DR0` through `DR15`:

`DR0` Breakpoint address #0

`DR1` Breakpoint address #1

`DR2` Breakpoint address #2

`DR3` Breakpoint address #3

`DR4` Obsolete alias for `DR6`

`DR5` Obsolete alias for `DR7`

`DR6` Debug status register

`DR7` Debug control register

`DR8–15` Reserved for processor use

We can set only four hardware breakpoints at a time, and we must write the addresses at which to break to the `DR0` through `DR3` registers. The least significant 4 bits of the status register (`DR6`) correspond to each of these registers in turn. If code triggers a hardware breakpoint for one of those addresses, the relevant bit in `DR6` will be set to 1. For example, triggering the breakpoint for the address held in `DR2` would set the third bit in `DR6`.

Let's next consider the bit layout of `DR7`, the control register, starting with the least-significant bit:

0 Local `DR0` breakpoint enabled

1 Global `DR0` breakpoint enabled

2 Local `DR1` breakpoint enabled

3 Global `DR1` breakpoint enabled

4 Local `DR2` breakpoint enabled

5 Global `DR2` breakpoint enabled

6 Local `DR3` breakpoint enabled

7 Global `DR3` breakpoint enabled

8–15 Reserved/not relevant to us

16–17 Conditions for `DR0`

18–19 Byte size of `DR0` breakpoint

20–21 Conditions for `DR1`

22–23 Byte size of `DR1` breakpoint

24–25 Conditions for `DR2`

26–27 Byte size of `DR2` breakpoint

28–29 Conditions for `DR3`

30–31 Byte size of `DR3` breakpoint

Those "local" and "global" breakpoint bits differ in their behavior across task switches. Your operating system is constantly switching between different processes to create the illusion that all the processes on your system are running at the same time. The local hardware breakpoints are supposed to affect only the currently running process, whereas global hardware breakpoints should apply to all processes. On Linux, however, the local and global breakpoints actually do the same thing and work in "local" mode. This choice makes sense when you consider that setting a global hardware breakpoint when debugging, say, your cat-petting tool could cause your word processor to hang (undesirable behavior).

The condition bits enable us to halt the process when a program reads from or writes to an address as well as when the address is executed. We'll use just the instruction execution option for setting breakpoints, but we'll use write and read breakpoints to implement watchpoints. Here are the meanings of the condition bits:

`00b` Instruction execution only

`01b` Data writes only

`10b` I/O reads and writes (generally unsupported)

`11b` Data reads and writes

Finally, the size field allows us to define how much data to watch in a watchpoint. For instruction execution breakpoints, we must set this field to 1 byte in size (`00b`). Here are the possible values:

`00b` 1 byte

`01b` 2 bytes

`10b` 8 bytes

`11b` 4 bytes

With the theory out of the way, we can start to implement hardware breakpoints.

Implementing Hardware Breakpoints

We need to support both setting and clearing hardware breakpoints. These operations will require reading and writing values from and to the debug registers. First, however, we need to augment the `sdb::breakpoint_site` type to track hardware breakpoints.

Tracking

Under the hood, we need a way to track whether a given breakpoint site should use hardware or software breakpoints. Let's add a field that does this to `sdb::breakpoint_site`. While we're at it, we'll also add a field for tracking whether a breakpoint is for internal usage, as the debugger will use

breakpoints itself to implement functionality like source-level stepping and shared library tracing. Add these to *sdb/include/libsdb/breakpoint_site.hpp*:

```
namespace sdb {
    class breakpoint_site {
    public:
        --snip--
        bool is_hardware() const { return is_hardware_; }
        bool is_internal() const { return is_internal_; }

    private:
        breakpoint_site(
            process& proc, virt_addr address,
            bool is_hardware = false, bool is_internal = false);

        --snip--
        bool is_hardware_;
        bool is_internal_;
     ❶ int hardware_register_index_ = -1;
    };
}
```

We add data members to track whether we should set a breakpoint site using hardware breakpoints and if the breakpoint is for internal use. We also add member functions for retrieving these members and add parameters (both defaulted to `false`) for them to the constructor. Finally, we add a `hardware_register_index` member that will track the index of the debug register a hardware breakpoint is using ❶.

In *sdb/src/breakpoint_site.cpp*, update the constructor call to initialize these new fields:

```
sdb::breakpoint_site::breakpoint_site(
    process& proc, virt_addr address, bool is_hardware, bool is_internal)
    : process_{ &proc }, address_{ address }, is_enabled_{ false },
    saved_data_{}, is_hardware_{ is_hardware },
    is_internal_{ is_internal } {
    id_ = is_internal_ ? -1 : get_next_id();
}
```

We initialize `is_hardware_` and `is_internal_`. We also set the site ID depending on whether `is_internal_` is `true`. Internal breakpoints get the ID `-1`.

The `sdb::process::read_memory_without_traps` function should ignore hardware breakpoints. Modify the function like this:

```
std::vector<std::byte>
sdb::process::read_memory_without_traps(
      virt_addr address, std::size_t amount) const {
    --snip--
    for (auto site : sites) {
```

```
        if (!site->is_enabled() or site->is_hardware()) continue;
        --snip--
    }
    --snip--
}
```

We add a condition to the existing `if` statement so that hardware breakpoints are ignored as well as disabled ones.

Next, we'll modify the `breakpoint_site::enable` and `disable` functions so they can handle hardware breakpoints. We'll implement the details in functions we'll add to `sdb::process` later in the chapter, but for now, we'll imagine that `sdb::process` has these `set_hardware_breakpoint` and `clear_hardware_stoppoint` functions. The `set_hardware_breakpoint` function will return the index of the debug register at which we set the breakpoint, and `clear_hardware_stoppoint` will take this index and clear the breakpoint at that register. Modify *sdb/src/breakpoint_site.cpp* like this:

```
void sdb::breakpoint_site::enable() {
    if (is_enabled_) return;

    if (is_hardware_) {
        hardware_register_index_ =
            process_->set_hardware_breakpoint(id_, address_);
    }
    else {
     ❶ --snip--
    }

    is_enabled_ = true;
}

void sdb::breakpoint_site::disable() {
    if (!is_enabled_) return;

    if (is_hardware_) {
        process_->clear_hardware_stoppoint(hardware_register_index_);
        hardware_register_index_ = -1;
    }
    else {
     ❷ --snip--
    }

    is_enabled_ = false;
}
```

After the check to `is_enabled_` in `enable`, we add an `if...else` block that checks `is_hardware_`. If this breakpoint site represents a hardware breakpoint, we call `set_hardware_breakpoint` with the breakpoint site ID and breakpoint

address and save the returned register index. We move the rest of the code currently in the function, apart from the `is_enabled_ = true;` statement, into the `else` block ❶.

In `disable`, we make a similar change, adding an `if...else` block after the check to `is_enabled_` that checks whether this site represents a hardware breakpoint. If so, we pass the saved register index to `clear_hardware_stoppoint` and reset `hardware_register_index_`. Again, we move the rest of the code except for the last store to `is_enabled_` into the `else` block ❷.

In *sdb/include/libsdb/process.hpp*, `create_breakpoint_site` should also take an argument for whether a hardware or software breakpoint should be created:

```
    --snip--
    breakpoint_site& create_breakpoint_site(
        virt_addr address,
        bool hardware = false,
        bool internal = false);
    --snip--
```

The implementation in *sdb/src/process.cpp* should forward this parameter to the `breakpoint_site` constructor:

```
sdb::breakpoint_site& sdb::process::create_breakpoint_site(
      virt_addr address, bool hardware, bool internal) {
    --snip--
    return breakpoint_sites_.push(
        std::unique_ptr<breakpoint_site>(
            new breakpoint_site(*this, address, hardware, internal)));
}
```

Then, `handle_breakpoint_command` in *sdb/tools/sdb.cpp* should optionally take a new `-h` argument, which specifies that the function should set a hardware breakpoint. Modify the call to `create_breakpoint_site` like so:

```
        if (is_prefix(command, "set")) {
            --snip--
            bool hardware = false;
            if (args.size() == 4) {
                if (args[3] == "-h") hardware = true;
                else sdb::error::send("Invalid breakpoint command argument");
            }
            process.create_breakpoint_site(
                sdb::virt_addr{ *address }, hardware).enable();
            --snip--
        }
```

You can also update the help info with explanations of the new features:

```
        --snip--
        else if (is_prefix(args[1], "breakpoint")) {
            std::cerr << R"(Available commands:
    list
    delete <id>
    disable <id>
    enable <id>
    set <address>
    set <address> -h
)";
        }
        --snip--
```

Finally, if the breakpoint site is internal, we shouldn't include it in the `breakpoint list` command, so add a check to `handle_breakpoint_command`:

```
namespace {
    void handle_breakpoint_command(sdb::process& process,
        --snip--
        if (is_prefix(command, "list")) {
            if (process.breakpoint_sites().empty()) {
                --snip--
            }
            else {
                fmt::print("Current breakpoints:\n");
                process.breakpoint_sites().for_each([](auto& site) {
                 ❶ if (site.is_internal()) return;
                    --snip--
                });
            }
            return;
        }
    }
}
```

At the start of the lambda that we pass to the `for_each` function, we return early if the breakpoint site is marked as internal ❶.

Now the debugger is keeping track of a new property for breakpoints, from the command line to the data model. It still doesn't do anything, though. Let's implement the functions we referenced earlier, starting with `set_hardware_breakpoint`.

Setting

To set a hardware stop point, we need to do the following:

1. Find a free space among the `DR` registers for the new stop point by locating one that isn't yet enabled.
2. Write the desired address to the correct `DR` register.
3. Encode the stop point mode and size into the form expected by the control register.
4. Clear the enable bit, mode bits, and size bits in the control register corresponding to the chosen `DR` register.
5. Mask in the new bits.
6. Write the new contents of the control register back to the system.

To facilitate this process, we'll write a private `set_hardware_stoppoint` function, which we'll reuse for both hardware breakpoints and watchpoints. Add the following to *sdb/include/libsdb/process.hpp*:

```
namespace sdb {
    class process {
    public:
        --snip--
        int set_hardware_breakpoint(
            breakpoint_site::id_type id, virt_addr address);

    private:
        --snip--
        int set_hardware_stoppoint(
            virt_addr address, /*?*/ mode, std::size_t size);
    };
}
```

The `mode` parameter should express whether to trigger the stop point on writing, on both reading and writing, or on the execution of the given address. Let's make an enum for this parameter in *sdb/include/libsdb/types.hpp*:

```
namespace sdb{
    --snip--
    enum class stoppoint_mode {
        write, read_write, execute
    };
}
```

The x64 architecture doesn't support stopping only on reads. We could simulate this behavior by keeping track of the value at the address, setting a read/write stop point, and then swallowing any traps at which the value at the address has changed. However, this is a very niche use case, so I won't

bother implementing it here. Plug the enum into the `set_hardware_stoppoint` function signature:

```
        int set_hardware_stoppoint(
            virt_addr address, stoppoint_mode mode, std::size_t size);
```

The implementation of `set_hardware_breakpoint` in *sdb/src/process.cpp* should simply forward to `set_hardware_stoppoint`. Recall that the size for execution-only hardware breakpoints must be `1`:

```
int sdb::process::set_hardware_breakpoint(
     breakpoint_site::id_type id, virt_addr address) {
    return set_hardware_stoppoint(address, stoppoint_mode::execute, 1);
}
```

Now that we've done the bookkeeping and data transfer, the fun can begin. Let's write the `set_hardware_stoppoint` function. First, read the control register (`DR7`) and find a free spot:

```
int sdb::process::set_hardware_stoppoint(
      virt_addr address, stoppoint_mode mode, std::size_t size) {
    auto& regs = get_registers();
    auto control = regs.read_by_id_as<std::uint64_t>(
        register_id::dr7);

 ❶ int free_space = find_free_stoppoint_register(control);
```

We read the control register, then use the `find_free_stoppoint_register` function, which we'll write later ❶. The function will return `0`, `1`, `2`, or `3`, depending on which register is free, or throw an exception if there is no free space.

Next, write the given address to the `DR` register corresponding to the free space you found:

```
    --snip--
    auto id = static_cast<int>(register_id::dr0) + free_space;
    regs.write_by_id(static_cast<register_id>(id), address.addr());
```

The debug registers' IDs all fall one after another; the ID of `DR1` is the one directly following `DR0`, and so on. So, we can calculate the correct ID by casting the ID of `DR0` to an integer, adding the index we were given by `find_free_stoppoint_register`, then casting it back to a `register_id`.

We'll call a couple more helper functions to encode the bits for the mode and size:

```
    --snip--
    auto mode_flag = encode_hardware_stoppoint_mode(mode);
    auto size_flag = encode_hardware_stoppoint_size(size);
```

Now, to set the correct bits, we must perform some serious bit twiddling. Take a look at the DR7 encoding in "Debug Registers" on page 186 to remind yourself of what calculations you'll need to perform. To set the correct bits, we'll calculate the following bit locations:

- Enable bit location: `free_space * 2`
- Mode bits location: `free_space * 4 + 16`
- Size bits location: `free_space * 4 + 18`

To set these bits with the correct content, we'll reset the relevant bits in the control register to 0, shift the new bits we'd like to write into the correct place, and bitwise OR everything together. For example, say we want to set a read/write stop point of size 8 on debug register 2. This means we'd want to set the enable flag to `1`, the mode flag to `11`, and the size flag to `10`.

Let's expand these values to 32 bits to make the operations easier to visualize:

```
Enable flag:   00000000 00000000 00000000 00000001
Mode flag:     00000000 00000000 00000000 00000011
Size flag:     00000000 00000000 00000000 00000010
```

Now let's shift the enable flag left by `2 * 2` (4), the mode flag by `2 * 4 + 16` (24), and the size flag by `2 * 4 + 18` (26):

```
Enable flag:   00000000 00000000 00000000 01000000
Mode flag:     00000011 00000000 00000000 00000000
Size flag:     00001000 00000000 00000000 00000000
```

If we bitwise OR these values together, we get the following:

```
Result:        00001011 00000000 00000000 01000000
```

We want to bitwise OR this result with the control register, but first we need to zero out the relevant bits to leave no old data behind. We can do this by generating a binary sequence that imagines that the enable, mode, and size bits were all set to 1, flipping all the bits in this sequence, and then bitwise ANDing this result with the control register. Here's an example, with completely random contents for the control register:

```
Bitmask:       00001111 00000000 00000000 01000000
Flipped:       11110000 11111111 11111111 10111111

Control:       10001110 10111110 01011110 10101111

Reset control: 10000000 10111110 01011110 00101111
```

Finally, we bitwise OR the two results together:

```
Desired bits:  00001011 00000000 00000000 01000000
Control:       10000000 10111110 01011110 00101111
Result:        10001011 10111110 01011110 01101111
```

The following listing translates this process into code in `set_hardware_stoppoint`:

```
    --snip--
    auto enable_bit = (1 << (free_space * 2));
    auto mode_bits = (mode_flag << (free_space * 4 + 16));
    auto size_bits = (size_flag << (free_space * 4 + 18));

    auto clear_mask = (0b11 << (free_space * 2))
                    | (0b1111 << (free_space * 4 + 16));
    auto masked = control & ~clear_mask;

    masked |= enable_bit | mode_bits | size_bits;
```

Finally, we write the new contents of the control register and return:

```
    --snip--
    regs.write_by_id(register_id::dr7, masked);

    return free_space;
}
```

To encode the mode and size, we use a couple of `switch` statements:

```
namespace {
    std::uint64_t encode_hardware_stoppoint_mode(
          sdb::stoppoint_mode mode) {
        switch (mode) {
        case sdb::stoppoint_mode::write: return 0b01;
        case sdb::stoppoint_mode::read_write: return 0b11;
        case sdb::stoppoint_mode::execute: return 0b00;
        default: sdb::error::send("Invalid stoppoint mode");
        }
    }

    std::uint64_t encode_hardware_stoppoint_size(
          std::size_t size) {
        switch (size) {
        case 1: return 0b00;
        case 2: return 0b01;
        case 4: return 0b11;
        case 8: return 0b10;
        default: sdb::error::send("Invalid stoppoint size");
        }
    }
}
```

These functions translate between `sdb::stoppoint_mode` or size values and the bit patterns that you saw in “Debug Registers” on page 187. To find the

first free DR register, we'll check the two enable bits in the control register that correspond to each DR register until we find one that has no bits set:

```
namespace {
    int find_free_stoppoint_register(
          std::uint64_t control_register) {
        for (auto i = 0; i < 4; ++i) {
            if ((control_register & (0b11 << (i * 2))) == 0) {
                return i;
            }
        }
        sdb::error::send("No remaining hardware debug registers");
    }
}
```

We've covered everything we need to do to set the stop point. Now we'll implement hardware breakpoint clearing.

Clearing

The function to clear a breakpoint is much simpler, and we can reuse some of the calculations we just made. Declare it in *sdb/include/libsdb/process.hpp*:

```
namespace sdb {
    class process {
    public:
        --snip--
        void clear_hardware_stoppoint(int index);
    };
}
```

Define this function in *sdb/src/process.cpp*:

```
void sdb::process::clear_hardware_stoppoint(int index) {
    auto id = static_cast<int>(register_id::dr0) + index;
    get_registers().write_by_id(static_cast<register_id>(id), 0);

    auto control = get_registers().
         read_by_id_as<std::uint64_t>(register_id::dr7);

    auto clear_mask = (0b11 << (index * 2))
                    | (0b1111 << (index * 4 + 16));
    auto masked = control & ~clear_mask;

    get_registers().write_by_id(register_id::dr7, masked);
}
```

We write `0` to the `DR` register at the given index, calculate a clearing mask for the control register (the same one we used in the previous section), bitwise AND the control register with the complement of the clear mask, and write the value back the control register.

The debugger now supports hardware breakpoints! To test them, let's go on a little detour into the world of reverse engineering and malware.

Testing Hardware Breakpoints

Reverse engineering contexts make heavy use of hardware breakpoints. Many malware specimens modify their own code while running, rendering software breakpoints useless because the malware can just scribble over the `0xcc` byte that the debugger puts in memory to trigger the break. Other kinds of malware detect whether they're being traced by a debugger and change their activities to hide their evil deeds.

Let's write some simulated malware. While the program won't do anything truly bad, it will try to detect whether it's being debugged with software breakpoints and, if so, act as if it's a regular, well-behaved program. Create a file called *sdb/test/targets/anti_debugger.cpp* containing a function called `an_innocent_function`. Despite the function's name, we'll have it do something evil:

```
#include <cstdio>

void an_innocent_function() {
    std::puts("Putting pineapple on pizza...");
}
```

If you like putting pineapple on pizza, that's fine; this book is a judgment-free zone. Feel free to replace the function's "malicious" behavior with something else. The `main` function will loop forever, calling `an_innocent_function` every second. However, if it detects a software breakpoint, it will put pepperoni on the pizza instead of pineapple.

The question is, how do we detect a software breakpoint? We could scan the code of `an_innocent_function` for `0xcc`, but `0xcc` could be part of data written to a register or something else not involving the `int3` instruction, so that solution is imperfect. We could also disassemble the function and look for `int3` instructions, but I'll choose a simpler option: writing a basic checksum function, running it on the function at startup, and then recalculating the checksum every time we call `an_innocent_function`. If the checksum has changed, we know the code has been modified. This is a simplified version of *section hashing*, the process of hashing the entire contents of the `.text` section of the binary (where the code lives) and embedding the result in the binary to check against at runtime.

We'll put an empty function right under an_innocent_function and use a pointer to it to work out the size of the code for an_innocent_function:

```
#include <cstdio>

void an_innocent_function() {
    std::puts("Putting pineapple on pizza...");
}

void an_innocent_function_end() {}
```

The checksum function then just sums up all the bytes of an_innocent_function into a single integer:

```
#include <numeric>

int checksum() {
    auto start = reinterpret_cast<volatile const char*>(
        &an_innocent_function);
    auto end = reinterpret_cast<volatile const char*>(
        &an_innocent_function_end);
    return std::accumulate(start, end, 0);
}
```

We cast the pointers to the an_innocent_function and an_innocent_function_end functions to volatile const char*s and then sum up all the bytes. We use volatile to tell the compiler that these bytes could change at any time, so it can't optimize the code by removing any loads from the memory. Finally, we write the main function:

```
#include <unistd.h>

int main() {
    auto safe = checksum();

    while (true) {
        sleep(1);
        if (checksum() == safe) {
            an_innocent_function();
        }
        else {
            puts("Putting pepperoni on pizza...");
        }
    }
}
```

The function first calculates the checksum of an_innocent_function and saves this as the "safe" checksum value, representing situations where no

breakpoint is set. It loops infinitely, sleeping for a second and then recalculating the checksum. If nothing has changed, we can keep putting pineapple on the pizza. Otherwise, someone has probably set a breakpoint, so the program acts natural and puts pepperoni on the pizza instead. Add this test target to *sdb/test/targets/CMakeLists.txt*:

```
--snip--
add_test_cpp_target(anti_debugger)
--snip--
```

We've written a program that puts pineapple on pizza, but only if nobody is watching. It's best to test this program with two terminals (though you can make do with one terminal and CTRL-Z). Launch the program in one terminal, and you'll see it endlessly putting pineapple on pizza:

```
$ test/targets/anti_debugger
Putting pineapple on pizza...
Putting pineapple on pizza...
Putting pineapple on pizza...
Putting pineapple on pizza...
```

In another terminal, get the PID of this process using `top` or `ps`:

```
$ ps ax | grep anti_debugger
4327 pts/0    S+     0:00 test/targets/anti_debugger
4335 pts/2    S+     0:00 grep --color=auto anti_debugger
```

The PID is 4327 in my case. Now you'll work out the load address of `an_innocent_function` using the same algorithm we did when setting software breakpoints in Chapter 7. First, get the offset of the code:

```
$ objdump -D test/targets/anti_debugger | grep an_innocent_function
0000000000001169 <_Z20an_innocent_functionv>:
0000000000001183 <_Z24an_innocent_function_endv>:
    119a:       48 8d 05 c8 ff ff ff    lea    -0x38(%rip),%rax
    11a5:       48 8d 05 d7 ff ff ff    lea    -0x29(%rip),%rax
    11f7:       e8 6d ff ff ff          call   1169 <_Z20an_innocent_functionv>
```

Here, the offset is `0x1169`. Then, find the load address:

```
$ cat /proc/4327/maps | grep anti_debugger | grep r-xp
55b0ce514000-55b0ce515000 r-xp 00001000 08:10 139999  anti_debugger
```

In my case, the offset is `55b0ce514000 - 1000 + 1169`, or `55b0ce514169`. Attach to the process and try setting a software breakpoint:

```
$ tools/sdb -p 4327
sdb> break set 0x55b0ce514169
sdb> c
```

In your other terminal, you should see that the program has detected the debugger and changed its behavior:

```
Putting pineapple on pizza...
Putting pineapple on pizza...
Putting pepperoni on pizza...
Putting pepperoni on pizza...
Putting pepperoni on pizza...
```

If you press CTRL-Z in the terminal with `anti_debugger` running, it should trap back into the debugger. You can then delete that breakpoint and reset it as a hardware breakpoint:

```
Process 4327 stopped with signal TSTP at 0x7f5678fca7fa
0x00007f5678fca7fa: mov %rax, %rbp
0x00007f5678fca7fd: mov %ebp, %r15d
0x00007f5678fca800: cmp $-0x16, %ebp
0x00007f5678fca803: jnz 0x00007F5678FCA809
0x00007f5678fca805: test %bl, %bl
sdb> break delete 1
sdb> break set 0x55b0ce514169 -h
```

Now, if you continue, you should successfully stop inside of the devious function:

```
sdb> c
Process 4327 stopped with signal TRAP at 0x55b0ce514169
0x000055b0ce514169: endbr64
0x000055b0ce51416d: push %rbp
0x000055b0ce51416e: mov %rsp, %rbp
0x000055b0ce514171: lea 0x000055b0ce515004, %rax
0x000055b0ce514178: mov %rax, %rdi
```

We'd like to add this process to the automated tests, so let's make it more easily testable. Back in *sdb/test/targets/anti_debugger.cpp*, we'll write the address of `an_innocent_function` at the start of `main` and then raise a `SIGTRAP` after each message instead of sleeping. Modify the `main` function like this:

```
#include <signal.h>

int main() {
    auto safe = checksum();

    auto ptr = reinterpret_cast<void*>(&an_innocent_function);
    write(STDOUT_FILENO, &ptr, sizeof(void*));
    fflush(stdout);

    raise(SIGTRAP);
```

```
    while (true) {
        if (checksum() == safe) {
            an_innocent_function();
        }
        else {
            puts("Putting pepperoni on pizza...");
        }

        fflush(stdout);
        raise(SIGTRAP);
    }
}
```

In the test, we'll essentially automate the same steps we just performed. We'll launch the process, read the address of the function on which to set a breakpoint, set the software breakpoint, and then resume and notice that the program put pepperoni on the pizza. Next, we'll delete the software breakpoint, create a hardware breakpoint at the same address, and continue, noting that we hit the breakpoint. We'll then continue again, noting that we caught the program putting pineapple on the pizza:

```
TEST_CASE("Hardware breakpoint evades memory checksums",
          "[breakpoint]") {
    bool close_on_exec = false;
    sdb::pipe channel(close_on_exec);
    auto proc = process::launch(
        "targets/anti_debugger", true, channel.get_write());
    channel.close_write();

    proc->resume();
    proc->wait_on_signal();

    auto func = virt_addr(
        from_bytes<std::uint64_t>(channel.read().data()));

    auto& soft = proc->create_breakpoint_site(func, false);
    soft.enable();

    proc->resume();
    proc->wait_on_signal();

    REQUIRE(to_string_view(channel.read()) ==
            "Putting pepperoni on pizza...\n");

    proc->breakpoint_sites().remove_by_id(soft.id());
    auto& hard = proc->create_breakpoint_site(func, true);
    hard.enable();
```

```
    proc->resume();
    proc->wait_on_signal();

    REQUIRE(proc->get_pc() == func);

    proc->resume();
    proc->wait_on_signal();

    REQUIRE(to_string_view(channel.read()) ==
            "Putting pineapple on pizza...\n");
}
```

Run the test, and ensure that the debugger catches the program in its illicit act. Now we can move on to watchpoints.

Watchpoints

Watchpoints use the same mechanisms as hardware breakpoints but can make a process stop when reading from and writing to an address as well as when executing it. This can be useful in a host of scenarios, such as advanced reverse engineering, tracking down memory corruption issues, and getting used to an existing codebase.

Start by creating an *sdb/include/libsdb/watchpoint.hpp* file that defines an `sdb::watchpoint` type. This type is fairly similar to `sdb::breakpoint_site`, but it doesn't need to track any saved data, though it must track a stop mode and watchpoint size. As in `sdb::breakpoint_site`, we should create these values only through an `sdb::process` object to track them correctly:

```
#ifndef SDB_WATCHPOINT_HPP
#define SDB_WATCHPOINT_HPP

#include <cstdint>
#include <cstddef>
#include <libsdb/types.hpp>

namespace sdb {
    class process;

    class watchpoint {
    public:
        watchpoint() = delete;
        watchpoint(const watchpoint&) = delete;
        watchpoint& operator=(const watchpoint&) = delete;

        using id_type = std::int32_t;
        id_type id() const { return id_; }

        void enable();
```

```
            void disable();

            bool is_enabled() const { return is_enabled_; }
            virt_addr address() const { return address_; }
            stoppoint_mode mode() const { return mode_; }
            std::size_t size() const { return size_; }

            bool at_address(virt_addr addr) const {
                return address_ == addr;
            }
            bool in_range(virt_addr low, virt_addr high) const {
                return low <= address_ and high > address_;
            }

        private:
            friend process;
            watchpoint(
                process& proc, virt_addr address,
                stoppoint_mode mode, std::size_t size);

            id_type id_;
            process* process_;
            virt_addr address_;
            stoppoint_mode mode_;
            std::size_t size_;
            bool is_enabled_;
            int hardware_register_index_ = -1;
        };
    }

    #endif
```

Like for `sdb::breakpoint_site`, we delete the default constructor and copy behaviors so the watchpoints are unique and can be created only through an `sdb::process` object. We also use the same members for tracking the ID, the address to which the watchpoint applies, whether the watchpoint is enabled, and the assigned hardware register index, and the same functions for checking whether the watchpoint is at a given address or within an address range. (We'll use these last two functions when we add support for source-level breakpoints.) In addition to these members, we create members for the stop point mode and the size of the watchpoint. The implementation defers most of the work to `sdb::process`. Write it in a new *sdb/src/watchpoint.cpp* file:

```
#include <libsdb/watchpoint.hpp>
#include <libsdb/process.hpp>
#include <libsdb/error.hpp>
```

```
namespace {
    auto get_next_id() {
        static sdb::watchpoint::id_type id = 0;
        return ++id;
    }
}

sdb::watchpoint::watchpoint(
    process& proc, virt_addr address, stoppoint_mode mode, std::size_t size)
    : process_{ &proc }, address_{ address }, is_enabled_{ false },
    mode_{ mode }, size_{ size } {
    id_ = get_next_id();
}

void sdb::watchpoint::enable() {
    if (is_enabled_) return;

    hardware_register_index_ = process_->set_watchpoint(id_, address_, mode_, size_);
    is_enabled_ = true;
}

void sdb::watchpoint::disable() {
    if (!is_enabled_) return;

    process_->clear_hardware_stoppoint(hardware_register_index_);
    is_enabled_ = false;
}
```

The `get_next_id` function is the same as the one for breakpoint sites. The constructor stores its parameters in the relevant members and calculates the ID for this watchpoint. The `enable` function calls an `sdb::process::set_watchpoint` function we'll write shortly, and `disable` calls the `clear_hardware_stoppoint` function we wrote earlier in this chapter. Add this new file to *sdb/src/CMakeLists.txt*:

```
add_library(libsdb ... watchpoint.cpp)
```

One common error case we can check for is that watchpoints on x64 must be aligned to their size: 8-byte watchpoints must fall on an 8-byte boundary, 4-byte watchpoints on a 4-byte boundary, and so on. We can check for this requirement with a handy little bitwise trick in the constructor. Edit it like so:

```
sdb::watchpoint::watchpoint(
    process& proc, virt_addr address,
    stoppoint_mode mode, std::size_t size)
    : process_{ &proc }, address_{ address },
    is_enabled_{ false }, mode_{ mode },
    size_{ size } {
```

```
    if ((address.addr() & (size - 1)) != 0) {
        error::send("Watchpoint must be aligned to size");
    }

    id_ = get_next_id();
}
```

If the size of the breakpoint is 8, the least significant 4 bits of the address must be 0 for the address to be aligned. Because `address & (8 - 1)` is `address & 0b1111`, we ensure that this calculation results in 0. Similarly, if the size is 4, the least significant 3 bits of the address should be 0, and `address & (4 - 1)` will be `address & 0b111`. The same approach works for 2-byte and 1-byte watchpoints.

Let's add the `set_watchpoint` function used in `sdb::watchpoint::enable` to `sdb::process`. In *sdb/include/libsdb/process.hpp*, write the following:

```
#include <libsdb/watchpoint.hpp>

namespace sdb {
    class process {
    public:
        --snip--
        int set_watchpoint(
            watchpoint::id_type id, virt_addr address,
            stoppoint_mode mode, std::size_t size);
        --snip--
    };
}
```

Then, implement the function in *sdb/src/process.cpp*, using the `set_hardware_stoppoint` function we already wrote:

```
int sdb::process::set_watchpoint(
      watchpoint::id_type id, virt_addr address,
      stoppoint_mode mode, std::size_t size) {
    return set_hardware_stoppoint(address, mode, size);
}
```

Now we need a way to create watchpoints from an `sdb::process`. Because we did the hard work of making `sdb::stoppoint_collection` a template that can support any kind of stop point, this should be fairly easy; we can simply store an `sdb::stoppoint_collection<watchpoint>`. In *sdb/include/libsdb/process.hpp*, create types and functions for managing watchpoints that mirror the existing breakpoint site functionality:

```
namespace sdb {
    class process {
    public:
        --snip--
```

```
        watchpoint& create_watchpoint(
            virt_addr address, stoppoint_mode mode, std::size_t size);
        stoppoint_collection<watchpoint>& watchpoints() {
            return watchpoints_;
        }
        const stoppoint_collection<watchpoint>& watchpoints() const {
            return watchpoints_;
        }
        --snip--

    private:
        --snip--
        stoppoint_collection<breakpoint_site> breakpoint_sites_;
        stoppoint_collection<watchpoint> watchpoints_;
    };
}
```

We declare a function for creating watchpoints, add a `stoppoint_collection <watchpoint>` member for tracking watchpoints, and add members for retrieving this collection. Implement `create_watchpoint` in *sdb/src/process.cpp*, much in the same way as `create_breakpoint_site`:

```
sdb::watchpoint&
sdb::process::create_watchpoint(virt_addr address, stoppoint_mode mode, std::size_t size) {
    if (watchpoints_.contains_address(address)) {
        error::send("Watchpoint already created at address " +
            std::to_string(address.addr()));
    }
    return watchpoints_.push(
        std::unique_ptr<watchpoint>(new watchpoint(*this, address, mode, size)));
}
```

Since we're using debug registers rather than modifying the running code, we technically could support multiple watchpoints with the same address, but it's not particularly worthwhile to do so. So, we throw an error if a watchpoint with the given address already exists. Otherwise, we create a new watchpoint and add it to the watchpoint collection.

Exposing Watchpoints to the User

The debugger will support the following watchpoint subcommands: `watchpoint list` to display all watchpoints, `watchpoint set <address> <mode> <size>` to set a new watchpoint, `watchpoint enable <id>` for enabling, `watchpoint disable <id>` for disabling, and `watchpoint delete <id>` for deletion. To expose the watchpoint feature, add a new command to *sdb/tools/sdb.cpp* in `handle_command`:

```
        --snip--
        else if (is_prefix(command, "watchpoint")) {
            handle_watchpoint_command(*process, args);
        }
        --snip--
```

To implement most of the subcommands, you can steal the code from `handle_breakpoint_command`, although `set` and `list` require more thought due to the different stop point modes that we support. As such, we'll split those into separate functions:

```
namespace {
    void handle_watchpoint_command(sdb::process& process,
          const std::vector<std::string>& args) {
        if (args.size() < 2) {
            print_help({ "help", "watchpoint" });
            return;
        }

        auto command = args[1];

        if (is_prefix(command, "list")) {
            handle_watchpoint_list(process, args);
            return;
        }

        if (is_prefix(command, "set")) {
            handle_watchpoint_set(process, args);
            return;
        }

        if (args.size() < 3) {
            print_help({ "help", "watchpoint" });
            return;
        }

        auto id = sdb::to_integral<sdb::watchpoint::id_type>(args[2]);
        if (!id) {
            std::cerr << "Command expects watchpoint id";
            return;
        }

        if (is_prefix(command, "enable")) {
            process.watchpoints().get_by_id(*id).enable();
        }
```

```
        else if (is_prefix(command, "disable")) {
            process.watchpoints().get_by_id(*id).disable();
        }
        else if (is_prefix(command, "delete")) {
            process.watchpoints().remove_by_id(*id);
        }
    }
}
```

In `handle_watchpoint_list`, we'll output the watchpoint's ID, address, mode, size, and whether it's enabled:

```
namespace {
    void handle_watchpoint_list(sdb::process& process,
          const std::vector<std::string>& args) {
     ❶ auto stoppoint_mode_to_string = [](auto mode) {
            switch (mode) {
            case sdb::stoppoint_mode::execute: return "execute";
            case sdb::stoppoint_mode::write: return "write";
            case sdb::stoppoint_mode::read_write: return "read_write";
            default: sdb::error::send("Invalid stoppoint mode");
            }
        };

        if (process.watchpoints().empty()) {
            fmt::print("No watchpoints set\n");
        }
        else {
            fmt::print("Current watchpoints:\n");
            process.watchpoints().for_each([&](auto& point) {
                fmt::print("{}: address = {:#x}, mode = {}, size = {}, {}\n",
                point.id(), point.address().addr(),
                stoppoint_mode_to_string(point.mode()), point.size(),
                point.is_enabled() ? "enabled" : "disabled");
            });
        }
    }
}
```

We write a lambda to get a string representation of a stop point mode ❶. If there are no watchpoints, we print a message to inform the user of this. Otherwise, we loop over all the watchpoints, printing their details for the user.

For `handle_watchpoint_set`, we'll expect the mode to be either `write`, `rw`, or `execute`:

```
namespace {
    void handle_watchpoint_set(sdb::process& process,
          const std::vector<std::string>& args) {
```

```
        if (args.size() != 5) {
            print_help({ "help", "watchpoint" });
            return;
        }
        auto address = sdb::to_integral<std::uint64_t>(args[2], 16);
        auto mode_text = args[3];
        auto size = sdb::to_integral<std::size_t>(args[4]);

        if (!address or !size or
            !(mode_text == "write" or
              mode_text == "rw" or
              mode_text == "execute")) {
            print_help({ "help", "watchpoint" });
            return;
        }

        sdb::stoppoint_mode mode;
        if (mode_text == "write") mode = sdb::stoppoint_mode::write;
        else if (mode_text == "rw") mode = sdb::stoppoint_mode::read_write;
        else if (mode_text == "execute") mode = sdb::stoppoint_mode::execute;

        process.create_watchpoint(
            sdb::virt_addr{ *address }, mode, *size).enable();
    }
}
```

If the user supplies an incorrect number of arguments, we throw an error. Otherwise, we parse the address as a hexadecimal integer, grab the string of the stop point mode argument, and parse the size as a decimal integer. If parsing either of those integers fails, or if the user passes an unexpected mode, we throw an error. Otherwise, we translate the stop point mode string into the relevant enumerator value, create a watchpoint with all of this data, and enable it.

As always, add a help message. We've implemented a sizeable number of commands at this point, so the help feature should be starting to pay off:

```
        if (args.size() == 1) {
            std::cerr << R"(Available commands:
    breakpoint  - Commands for operating on breakpoints
    continue    - Resume the process
    disassemble - Disassemble machine code to assembly
    memory      - Commands for operating on memory
    register    - Commands for operating on registers
    step        - Step over a single instruction
    watchpoint  - Commands for operating on watchpoints
)";
        }
        --snip--
```

```
        else if (is_prefix(args[1], "watchpoint")) {
            std::cerr << R"(Available commands:
    list
    delete <id>
    disable <id>
    enable <id>
    set <address> <write|rw|execute> <size>
)";
        }
        --snip--
```

Now we can move on to testing.

Testing Watchpoints

We'll reuse the `anti_debugger` example from the hardware breakpoint test to check whether we can use watchpoints to detect the checksum function's reading of the program's code. Then, we'll set a software breakpoint right after the read to evade the protection mechanism. First, launch the program with the usual pipe to read its output:

```
TEST_CASE("Watchpoint detects read", "[watchpoint]") {
    bool close_on_exec = false;
    sdb::pipe channel(close_on_exec);
    auto proc = process::launch(
        "targets/anti_debugger", true, channel.get_write());
    channel.close_write();
```

Continue it until you reach the first trap, then read the pointer that the program prints to `stdout` (where we want to eventually set a breakpoint):

```
    --snip--
    proc->resume();
    proc->wait_on_signal();

    auto func = virt_addr(
        from_bytes<std::uint64_t>(channel.read().data()));
```

Set a watchpoint on that address so the program will halt when the checksum function tries to read the code of the function on which we'll set a breakpoint:

```
--snip--
auto& watch = proc->create_watchpoint(func, sdb::stoppoint_mode::read_write, 1);
watch.enable();
```

Resume the process until the watchpoint is hit. Step over a single instruction so that the anti-debugger reads the function's original code, then sneakily set a software breakpoint:

```
    --snip--
    proc->resume();
    proc->wait_on_signal();

    proc->step_instruction();
    auto& soft = proc->create_breakpoint_site(func, false);
    soft.enable();
```

Resume the process. At this point, the software breakpoint should have been hit, so require that the process stop due to a `SIGTRAP`, then resume once more:

```
    --snip--
    proc->resume();
    auto reason = proc->wait_on_signal();

    REQUIRE(reason.info == SIGTRAP);

    proc->resume();
    proc->wait_on_signal();

    REQUIRE(to_string_view(channel.read()) == "Putting pineapple on pizza...\n");
}
```

The program should print `Putting pineapple on pizza...`, indicating that we successfully used watchpoints to evade the protection. If all of the tests have passed, the processes should consider themselves well and truly watched.

Summary

In this chapter, you learned how to use the x64 debug registers to set hardware breakpoints and watchpoints. You added support to your debugger for setting both of these and wrote automated tests to ensure that these features work. In the next chapter, you'll improve the signal handling of your debugger and add the ability to trace syscalls.

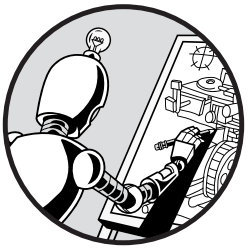
Check Your Knowledge

1. How many hardware breakpoints can you set at the same time on x64?
2. Which register is the debug control register?
3. How could you simulate read-only watchpoints?
4. In which situations should users use hardware breakpoints rather than software breakpoints?

10

SIGNALS AND SYSCALLS

A flashlight
across the street
calling us to warmth.



Now that the debugger implements breakpoints, stepping, and watchpoints, it's close to being a useful tool for everyday use. In this chapter, you'll extend its signal handling and add the ability to trace syscalls. You'll also learn how syscalls work on Linux and how the operating system sends signals to processes when certain events occur.

Signal Handlers

One common debugger feature we don't yet support is the ability to press CTRL-C on the command line to interrupt the inferior process. Pressing CTRL-C sends a `SIGINT` signal to the currently executing process, and at the moment, this causes *sdb* to terminate. Ideally, the *sdb* process should send a `SIGSTOP` to the inferior, then continue reading input from the user. This will force the inferior to halt and allow the user to examine the program state.

We can receive notifications of `SIGINT` signals sent to *sdb* by installing a *signal handler*, a function that a process registers with the operating system to deal with a specific signal. When the process receives a signal, the operating system calls the corresponding signal handler.

Importantly, a call to your signal handler could happen at any time. Your program could be in the middle of performing a calculation, running some other signal handler, or sending an email. As such, you need to be mindful of the kinds of operations you carry out in a signal handler. The handler should be *reentrant*, or able to be safely interrupted and resumed without issue. Depending on the settings used when installing the handler, it may also need to allow multiple concurrent calls. This means that signal handlers can't perform I/O operations, acquire a mutex, or take actions that rely on external state.

POSIX defines a list of functions deemed *async-signal-safe*, meaning you can safely call them from inside a signal handler. The list is too long to include here, but you can find it by running `man signal-safety`.

You can install signal handlers using the `signal` syscall, which takes a signal ID on which to install a handler and a pointer to the handler function. This handler function can be either `SIG_IGN`, which ignores the signal; `SIG_DFL`, which invokes the default handler; or a pointer to a function of type `void (*sighandler_t) (int)`, which installs that function as the handler. (For those who don't like reading C function pointer syntax, that's a pointer to a function returning `void` and taking a single `int`, the received signal.)

Let's install a `SIGINT` signal handler that sends a `SIGSTOP` to the inferior. We have no way to pass custom data to a signal handler, so we must add the current process as a global variable. (I know, mutable global variables are evil. Let this be a reminder to tread extra carefully inside of signal handlers.) In *sdb/tools/sdb.cpp*, add a `g_sdb_process` variable:

```
namespace {
    sdb::process* g_sdb_process = nullptr;
}
```

Then, create a signal handler that calls `kill` with the PID of the inferior. The `kill` syscall is async-signal-safe, so we shouldn't encounter any issues:

```
namespace {
    void handle_sigint(int) {
        kill(g_sdb_process->pid(), SIGSTOP);
    }
}
```

Now install this signal handler and update `g_sdb_process` from `main`:

```
#include <csignal>

int main(int argc, const char* argv[]) {
    --snip--
```

```
    try {
        auto process = attach(argc, argv);
        g_sdb_process = process.get();
        signal(SIGINT, handle_sigint);
        main_loop(process);
    }
    --snip--
}
```

After the call to `attach`, save a pointer to the process in the `g_sdb_process` global variable and then install the `handle_sigint` function as a signal handler for `SIGINT` signals before starting the main loop.

This code may seem fine and dandy, but fine and dandy it is not. You may notice that if you test it, the child process receives both a `SIGINT` and a `SIGSTOP`:

```
$ tools/sdb test/targets/run_endlessly
Launched process with PID 414
sdb> c
^CProcess 414 stopped with signal INT at 0x555555555138
0x0000555555555138: jmp 0x0000555555555131
0x000055555555513a: add %al, (%rax)
0x000055555555513c: endbr64
0x0000555555555140: sub $0x08, %rsp
0x0000555555555144: add $0x08, %rsp
sdb> c
Process 414 stopped with signal STOP at 0x555555555138
0x0000555555555138: jmp 0x0000555555555131
0x000055555555513a: add %al, (%rax)
0x000055555555513c: endbr64
0x0000555555555140: sub $0x08, %rsp
0x0000555555555144: add $0x08, %rsp
```

What gives? Well, when you press CTRL-C, you don't merely send a `SIGINT` to the running process; you send it to all processes in the same process group. Forked processes run in the same process group as their parent, so when *sdb* gets a `SIGINT`, the inferior gets a `SIGINT`. You can see these process group IDs with the `ps` command:

```
$ ps xao pid,pgid,comm
  PID  PGID COMMAND
    1     0 init
 --snip--
 4600  4600 sdb
 4601  4600 run_endlessly
```

Note that *sdb* and *run_endlessly* have different PIDs but the same process group ID (PGID).

"Okay," I hear you say, "let's just set SIG_IGN as the signal handler on *sdb* to ignore the signal and let the child process get the SIGINT."

Not so fast. If we do that, the inferior will also catch other signals we don't want it to. For example, when the user presses CTRL-Z to suspend *sdb* to the background, they will also suspend the inferior. Even worse, resizing the window of the terminal in which *sdb* is running will stop the inferior with a SIGWINCH.

The correct approach is to keep the SIGINT handler in *sdb*, but change the inferior's process group when we fork it. We can do this with setpgid, which has the following signature:

```
int setpgid(pid_t pid, pid_t pgid);
```

If you call setpgid with two 0s, it automatically applies to the PID of the running process and sets the PGID to the same value as the PID. Modify sdb::process::launch in *sdb/src/process.cpp* like so:

```
std::unique_ptr<sdb::process> sdb::process::launch(
      std::filesystem::path path, bool debug,
      std::optional<int> stdout_replacement) {
    --snip--

    if (pid == 0) {
        if (setpgid(0, 0) < 0) {
            exit_with_perror(channel, "Could not set pgid");
        }
        personality(ADDR_NO_RANDOMIZE);
        --snip--
    }
}
```

Before the call to personality, call setpgid and ensure that it causes no error. Now, if you build and restart *sdb*, you can see that the PGID of the inferior differs from that of *sdb*:

```
$ ps xao pid,pgid,comm
  PID  PGID COMMAND
    1     0 init
 #...
 4698  4698 sdb
 4699  4699 run_endlessly
```

Try sending a CTRL-C. You should see a SIGSTOP in the inferior only:

```
$ tools/sdb test/targets/run_endlessly
Launched process with PID 507
sdb> c
^CProcess 507 stopped with signal STOP at 0x555555555131
0x0000555555555131: movl $0x2A, -0x04(%rbp)
0x0000555555555138: jmp 0x0000555555555131
```

```
0x000055555555513a: add %al, (%rax)
0x000055555555513c: endbr64
0x0000555555555140: sub $0x08, %rsp
sdb> c
```

Much better.

Another improvement we could make to the user experience is printing more complete information about why a stop occurred. Right now, if the user hits a breakpoint, the debugger tells them that a `SIGTRAP` occurred, but not whether it was due to a single step, a watchpoint, or a breakpoint. If a breakpoint or a watchpoint caused the stop, the user doesn't know which one was hit, and for watchpoints, they can't know the new value written to the address without running a `memory read` command themselves. Let's fix that.

Printing Stop Information

The `waitpid` function identifies the signal that caused the child to stop, but it doesn't give us much information to report back to the user. Fortunately, `ptrace` comes to the rescue with the `PTRACE_GETSIGINFO` command, which provides the following information about a signal:

```
typedef struct {
    int          si_signo;      /* Signal number */
    int          si_code;       /* Signal code */
    pid_t        si_pid;        /* Sending process ID */
    uid_t        si_uid;        /* Real user ID of sending process */
    void        *si_addr;       /* Address of faulting instruction */
    int          si_status;     /* Exit value or signal */
    union sigval si_value;      /* Signal value */
} siginfo_t;
```

The value we're particularly interested in here is `si_code`, which gives us detailed information about the source of a signal. For example, while `si_signo` indicates that the signal was a `SIGFPE` for floating-point errors, `si_code` can tell us whether it was a division by zero, an overflow, an invalid operation, and so on. In theory, `si_code` may have one of the following values on a `SIGTRAP`:

`SI_KERNEL` Generic trap sent from the kernel

`TRAP_BRKPT` Software breakpoint

`TRAP_HWBKPT` Hardware breakpoint

`TRAP_TRACE` Single step

Somewhat amusingly, however, the Linux kernel actually reports the wrong values on x64: `SI_KERNEL` for software breakpoints and `TRAP_BRKPT` for single-stepping over a syscall. Enough important tools rely on this bug's behavior that it's just not worth fixing anymore.

Let's add support to `sdb::stop_reason` for displaying extended signal information. In *sdb/include/libsdb/process.hpp*, add a new enum called `sdb::trap_type` and add a member to `stop_reason` to optionally hold one:

```
namespace sdb {
    enum class trap_type {
        single_step, software_break,
        hardware_break, unknown
    };

    struct stop_reason {
        --snip--
        std::optional<trap_type> trap_reason;
    };
}
```

The `trap_type` enum describes whether a `SIGTRAP` occurred due to a single step, a software breakpoint, a hardware breakpoint, or an unknown reason. We add a `std::optional<trap_type>` member to `sdb::stop_reason` to store a trap reason if the stop occurred due to a `SIGTRAP`.

Next, add a new private member function to `sdb::process` called `augment_stop_reason`, which we'll call once we know that the process stopped because of a signal rather than because it exited or terminated. Make the following change to *sdb/include/libsdb/process.hpp*:

```
namespace sdb {
    class process{
    private:
        --snip--
        void augment_stop_reason(stop_reason& reason);
        --snip--
    };
}
```

Now implement the function in *sdb/src/process.cpp* by calling `ptrace` with `PTRACE_GETSIGINFO`:

```
void sdb::process::augment_stop_reason(sdb::stop_reason& reason) {
    siginfo_t info;
    if (ptrace(PTRACE_GETSIGINFO, pid_, nullptr, &info) < 0) {
        error::send_errno("Failed to get signal info");
    }

    reason.trap_reason = trap_type::unknown;
    if (reason.info == SIGTRAP) {
        switch (info.si_code) {
        case TRAP_TRACE:
            reason.trap_reason = trap_type::single_step;
            break;
```

```
        case SI_KERNEL:
            reason.trap_reason = trap_type::software_break;
            break;
        case TRAP_HWBKPT:
            reason.trap_reason = trap_type::hardware_break;
            break;
        }
    }
}
```

First, we call `ptrace` to get the signal information and report any error. If the stop occurred due to a `SIGTRAP`, we record the trap reason based on the `si_code` member of the signal information, defaulting to `unknown`. Remember that x64 uses `SI_KERNEL`, not `TRAP_BRKPT`, for software breakpoints.

Call this new function in `sdb::process::wait_on_signal` after reading all registers. We don't rely on the state of registers in `augment_stop_reason` yet, but we will once we implement syscall tracing:

```
sdb::stop_reason sdb::process::wait_on_signal() {
    --snip--
    if (is_attached_ and state_ == process_state::stopped) {
        read_all_registers();
        augment_stop_reason(reason);

        auto instr_begin = get_pc() - 1;
        --snip--
    }
    --snip--
}
```

Before we update the `print_stop_reason` function to expose this new data, we must implement a couple of other missing elements. Notably, we don't have an easy way to identify which watchpoint or hardware breakpoint was hit. Also, we don't keep track of the value previously stored at a watchpoint location so we can print out what change occurred, if any. Let's tackle these in turn.

Tracking Debug Register Assignments

When we receive a trap due to a hardware breakpoint, we can check the debug status register (`DR6`) to see which breakpoint was hit. We can then check the corresponding register in the range `DR0`–`DR3` to retrieve the address at which the debugger set the breakpoint and use this to locate the corresponding breakpoint or watchpoint. We'll add a function to `sdb::process` to achieve this. In *sdb/include/libsdb/process.hpp*, make the following modification:

```
namespace sdb {
    class process {
    public:
        --snip--
```

```
        std::variant<breakpoint_site::id_type, watchpoint::id_type>
        get_current_hardware_stoppoint() const;
        --snip--
    };
}
```

You may notice that `breakpoint_site::id_type` and `watchpoint::id_type` happen to be aliases to the same type. It's acceptable to use multiple options of the same type in a `std::variant`, as the `std::variant::index` function distinguishes them.

Recall from Chapter 9 that the `DR6` status register uses the least significant 4 bits to encode the hit breakpoint. For example, setting bit 0 means the breakpoint described by `DR0` was hit, setting bit 1 means the breakpoint in `DR1` was hit, and so on. GCC and Clang provide a handy function for finding the position of the least significant set bit: `__builtin_ctz`, short for "count trailing zeros." The name is formulated in different words, but if you think about it, you can convince yourself that "count trailing zeros" and "find position of the least significant set bit" are equivalent.

With the help of this function, we can find the hardware stop point that caused the process to trap. Implement `get_current_hardware_stoppoint` in *sdb/src/process.cpp*:

```
std::variant<sdb::breakpoint_site::id_type, sdb::watchpoint::id_type>
sdb::process::get_current_hardware_stoppoint() const {
    auto& regs = get_registers();
    auto status = regs.read_by_id_as<std::uint64_t>(register_id::dr6);
    auto index = __builtin_ctzll(status); ❶

    auto id = static_cast<int>(register_id::dr0) + index;
    auto addr = virt_addr(
        regs.read_by_id_as<std::uint64_t>(static_cast<register_id>(id)));

    using ret = std::variant<sdb::breakpoint_site::id_type, sdb::watchpoint::id_type>;
    if (breakpoint_sites_.contains_address(addr)) {
        auto site_id = breakpoint_sites_.get_by_address(addr).id();
        return ret{ std::in_place_index<0>, site_id }; ❷
    }
    else {
        auto watch_id = watchpoints_.get_by_address(addr).id();
        return ret{ std::in_place_index<1>, watch_id };
    }
}
```

We read the contents of the status register, then find the index of the set bit. Here, we use the `ll` (`unsigned long long`) flavor of the `__builtin_ctz` function ❶. We calculate the ID of the relevant register by using the same trick we used in Chapter 9: casting the ID of `DR0` to an integer, adding the index of the register we want, and then casting it back to the `register_id` type.

Reading the register with this ID gives us the address at which the stop point is set.

We need to return a `std::variant<sdb::breakpoint_site::id_type, sdb::watchpoint::id_type>` value, and we make an alias for it to avoid having to write it twice. If the breakpoint sites collection contains the stop point's address, we find the breakpoint site corresponding to that address and return the breakpoint site ID. We use `std::in_place_index<0>` to specify that we're setting the type at index 0 of the `std::variant`, which is `sdb::breakpoint_site::id_type` ❷. If no breakpoint site with that address exists, this must be a watchpoint, so we find the corresponding watchpoint and return its ID, using `std::in_place_index<1>` to specify that the ID corresponds to a watchpoint.

Tracking Watchpoint Values

Next, we'll enable watchpoints to track the values to which they point. Add two new fields to `sdb::watchpoint` to hold the current value at the watched address and the previously read value. Also add an `update_data` function that rereads the value at the watched memory location. We'll call this function whenever a watchpoint triggers a stop. Edit *sdb/include/libsdb/watchpoint.hpp* like so:

```
namespace sdb {
    class process;

    class watchpoint {
    public:
        --snip--
        std::uint64_t data() const { return data_; }
        std::uint64_t previous_data() const { return previous_data_; }

        void update_data();

    private:
        --snip--
        std::uint64_t data_ = 0;
        std::uint64_t previous_data_ = 0;
    };
}
```

Now implement `update_data` and call it from the constructor in *sdb/src/watchpoint.cpp*. The function should read the new data from `address_` into `data_` and put the old value of `data_` into `previous_data_`:

```
#include <utility>

sdb::watchpoint::watchpoint(
    process& proc, addr_t address, stoppoint_mode mode, std::size_t size)
    : process_{ &proc }, address_{ address }, is_enabled_{ false },
```

```
    mode_{ mode }, size_{ size } {
    --snip--
    update_data();
}

void sdb::watchpoint::update_data() {
    std::uint64_t new_data = 0;
    auto read = process_->read_memory(address_, size_);
    memcpy(&new_data, read.data(), size_);
    previous_data_ = std::exchange(data_, new_data);
}
```

We call `update_data` at the end of the constructor and then implement `update_data`. We declare a `new_data` variable, read the necessary amount of memory from the watched address, copy this data into `new_data`, and store the result in the `data_` member, putting the old value of `data_` in `previous_data_`.

We should also call the `update_data` function whenever a watchpoint triggers a stop. We can do this inside `sdb::process::wait_on_signal`, in *sdb/src/process.cpp*. Modify the file like this:

```
    --snip--
    if (is_attached_ and state_ == process_state::stopped) {
        --snip--
        auto instr_begin = get_pc() - 1;
        if (reason.info == SIGTRAP) {
            if (reason.trap_reason == trap_type::software_break and
                breakpoint_sites_.contains_address(instr_begin) and
                breakpoint_sites_.get_by_address(instr_begin).is_enabled()) {
                set_pc(instr_begin);
            }
            else if (reason.trap_reason == trap_type::hardware_break) {
                auto id = get_current_hardware_stoppoint();
             ❶ if (id.index() == 1) {
                    watchpoints_.get_by_id(std::get<1>(id)).update_data();
                }
            }
        }
    }
    --snip--
```

We change the `if` statement that begins with the `reason.info == SIGTRAP` test. When we get a `SIGTRAP`, we check the reason for it. If a software breakpoint caused the stop, we walk the program counter back 1 byte to the start of the `int3` instruction. If, instead, a hardware stop point caused the stop and the current hardware stop point is a watchpoint (which we check by examining the index of the variant returned by `get_current_hardware_stoppoint` ❶), we update the watchpoint's data. Note that multiple watchpoints could be triggered by the execution of a single instruction; for example, 16 bytes could

be watched with two contiguous 8-byte watchpoints, which are then simultaneously triggered by a 16-byte write. We don't deal with this edge case here, but you can choose to add handling for it if you desire.

Displaying Stop Information

Now we can put these pieces together to greatly improve the debugger's reporting about stop reasons. In `print_stop_reason`, we currently print a generic `stopped with signal <signal> at <pc>` message. If the signal was a `SIGTRAP`, we should augment this message with information about the specific stop point or indicate that the stop occurred due to a single step. For watchpoints, we should also print out the value stored at that address.

We'll do this work in a `get_sigtrap_info` function, which we'll implement shortly. First, though, let's call it from `print_stop_reason` in *sdb/tools/sdb.cpp*. Modify the `sdb::process_state::stopped` case of the `switch` statement as follows:

```
        case sdb::process_state::stopped:
            message = fmt::format("stopped with signal {} at {:#x}",
                sigabbrev_np(reason.info), process.get_pc().addr());
            if (reason.info == SIGTRAP) {
                message += get_sigtrap_info(process, reason);
            }
            break;
```

We begin the message with the same text as before. If the process stopped because of a `SIGTRAP`, we also call `get_sigtrap_info` and append the result to the message.

Let's start implementing `get_sigtrap_info` by adding the breakpoint ID to the message for software breakpoints. We can look up the breakpoint site using the current program counter. Write the following in *sdb/tools/sdb.cpp*:

```
namespace {
    std::string get_sigtrap_info(
          const sdb::process& process, sdb::stop_reason reason) {
        if (reason.trap_reason == sdb::trap_type::software_break) {
            auto& site = process.breakpoint_sites().get_by_address(process.get_pc());
            return fmt::format(" (breakpoint {})", site.id());
        }
```

If the trap occurred due to a software breakpoint, we find the breakpoint site corresponding to the current program counter and return a string giving the site's ID.

For hardware breakpoints, we should find the current hardware stop point. In the case of a hardware breakpoint, we print just the ID, but for a watchpoint, we should also print the current value, along with the previous value if it differs:

```
        --snip--
        if (reason.trap_reason == sdb::trap_type::hardware_break) {
```

```
            auto id = process.get_current_hardware_stoppoint();

            if (id.index() == 0) {
                return fmt::format(" (breakpoint {})", std::get<0>(id));
            }

            std::string message;
            auto& point = process.watchpoints().get_by_id(std::get<1>(id));
            message += fmt::format(" (watchpoint {})", point.id());

            if (point.data() == point.previous_data()) {
                message += fmt::format("\nValue: {:#x}", point.data());
            }
            else {
                message += fmt::format("\nOld value: {:#x}\nNew value: {:#x}",
                    point.previous_data(), point.data());
            }
            return message;
        }
```

For traps caused by hardware stop points, we retrieve the ID of the current hardware stop point. If the index of the returned variant is 0, we're looking at a hardware breakpoint, so we return a message containing that breakpoint's ID. Otherwise, we're looking at a watchpoint. We grab the relevant watchpoint, start a message with the ID of the watchpoint, and then handle the watchpoint's data. If the current data matches the previous data, we just report the current data. Otherwise, we print both the old value and the new value.

For traps caused by a single step, we simply return `(single step)`. Otherwise, we don't know the reason for the stop, so we return an empty string:

```
        --snip--
        if (reason.trap_reason == sdb::trap_type::single_step) {
            return " (single step)";
        }

        return "";
    }
}
```

We can test this new reporting feature with the *anti_debugger* program. Launch this program under *sdb* and set a read/write watchpoint at the start of `an_innocent_function`, whose file address you can find with *objdump* and whose load address is at */proc/<pid>/maps*. Then, resume the process. You should see the watchpoint ID and value printed out:

```
$ tools/sdb test/targets/anti_debugger
Launched process with PID 2815
sdb> watch set 0x5555555551a9 rw 1
```

```
sdb> c
Process 2815 stopped with signal TRAP at 0x5555555552c3 (watchpoint 1)
Value: 0xf3
0x00005555555552c3: movsx %al, %eax
0x00005555555552c6: add %eax, -0x14(%rbp)
0x00005555555552c9: addq $0x01, -0x08(%rbp)
0x00005555555552ce: mov -0x08(%rbp), %rax
0x00005555555552d2: cmp -0x10(%rbp), %rax
```

The debugger is starting to look like the real thing! You can repeat this test with hardware breakpoints if you'd like (remembering to delete the watchpoint first).

Soon, we'll do a deep dive into how the inferior actually receives those `SIGTRAP` signals, but first, let's add support for catching syscalls.

Catchpoints

In this section, we'll introduce a new concept to the debugger: catchpoints. A *catchpoint* stops the process when a specific event occurs. We'll implement only one catchpoint, for syscalls, but you may choose to add additional catchpoints for C++ exceptions and shared library loading or unloading (see Chapter 22 for more on this).

Tracing Syscalls

We won't need to create an `sdb::catchpoint` type to implement catchpoints for syscalls, as we can adjust our use of `ptrace` to trace them. By enabling the `PTRACE_O_TRACESYSGOOD` option, we can extend the signal information that `ptrace` provides so it becomes much easier to distinguish `SIGTRAP` signals that come from syscalls. We'll do this in `sdb::process::launch` and `sdb::process::attach`. Make the following changes to *sdb/src/process.cpp*:

```
namespace {
    void set_ptrace_options(pid_t pid) {
        if (ptrace(PTRACE_SETOPTIONS, pid, nullptr, PTRACE_O_TRACESYSGOOD) < 0) {
            sdb::error::send_errno("Failed to set TRACESYSGOOD option");
        }
    }
}
std::unique_ptr<sdb::process>
sdb::process::launch(std::filesystem::path path, bool debug,
      std::optional<int> stdout_replacement) {
    --snip--
    if (debug) {
        proc->wait_on_signal();
        set_ptrace_options(proc->pid());
    }
```

```
    return proc;
}

std::unique_ptr<sdb::process>
sdb::process::attach(pid_t pid) {
    --snip--
    set_ptrace_options(proc->pid());

    return proc;
}
```

We introduce a new `set_ptrace_options` function that calls `ptrace` and sets the `PTRACE_O_TRACESYSGOOD` option, being sure to handle potential errors. We call this function from both `launch` and `attach`, right before returning.

Now that we've enabled this option, the signal number will have its eighth bit set if the `SIGTRAP` came from a syscall. This means we can use `signal == (SIGTRAP|0x80)` to check whether we're trapped by a syscall. To actually receive traps from syscalls, we use the `PTRACE_SYSCALL` request instead of `PTRACE_CONT` when resuming the process.

Let's add a new member to `sdb::process` to track which syscalls we're tracing. We may want to trace only a subset of them, as a multitude of syscalls can occur. We'll create a new `sdb::syscall_catch_policy` type to capture this information. In *sdb/include/libsdb/process.hpp*, make the following changes:

```
namespace sdb {
    class syscall_catch_policy {
    public:
        enum mode {
            none, some, all
        };

        static syscall_catch_policy catch_all() {
            return { mode::all, {} };
        }

        static syscall_catch_policy catch_none() {
            return { mode::none, {} };
        }

        static syscall_catch_policy catch_some(std::vector<int> to_catch) {
            return { mode::some, std::move(to_catch) };
        }

        mode get_mode() const { return mode_; }
        const std::vector<int>& get_to_catch() const { return to_catch_; }

    private:
        syscall_catch_policy(mode mode, std::vector<int> to_catch) :
            mode_(mode), to_catch_(std::move(to_catch)) {}
```

```
        mode mode_ = mode::none;
        std::vector<int> to_catch_;
    };
}
```

We'll either catch no syscalls, all syscalls, or certain syscalls, so we create an enum called `mode` that can express this. We add two data members: one for the mode (which defaults to catching no syscalls) and a `std::vector<int>` that lists the syscall IDs we should be catching in the case that `mode_` is `mode::some`. We also add a private constructor that fills these in. We make it private rather than public because we use *named constructors* to make creating a `syscall_catch_policy` less verbose. These are the static `catch_all`, `catch_none`, and `catch_some` functions. Finally, we add members to retrieve the stored data.

Add a member of this type to `sdb::process`, along with a member to set the current policy:

```
namespace sdb {
    class process {
    public:
        --snip--
        void set_syscall_catch_policy(syscall_catch_policy info) {
            syscall_catch_policy_ = std::move(info);
        }

    private:
        --snip--
        syscall_catch_policy syscall_catch_policy_ =
            syscall_catch_policy::catch_none();
    };
}
```

We default the catching policy to catch no syscalls.

In `sdb::process::resume`, we tune the `ptrace` request we send based on this member. In *sdb/src/process.cpp*, make the following change to the `PTRACE_CONT` call:

```
void sdb::process::resume() {
    --snip--
    auto request =
        syscall_catch_policy_.get_mode() == syscall_catch_policy::mode::none ?
        PTRACE_CONT : PTRACE_SYSCALL;
    if (ptrace(request, pid_, nullptr, nullptr) < 0) {
        error::send_errno("Could not resume");
    }
    --snip--
}
```

If the current syscall catch policy is to catch no syscalls, we pass `PTRACE_CONT` to `ptrace`. Otherwise, we'll want to catch at least some syscalls, so we pass `PTRACE_SYSCALL`. Now the inferior will trap whenever a syscall is entered or exited. We should communicate data about these stops in `sdb::process::wait_on_signal`.

If we request a `PTRACE_SYSCALL`, the inferior will halt twice for each syscall: once on entry and once on exit. This enables the tracer to check the arguments to the syscall before it's executed and then check the return value on exit. Let's add a type, `sdb::syscall_information`, to express this. Define it in *sdb/include/libsdb/process.hpp*:

```
namespace sdb {
    struct syscall_information {
        std::uint16_t id;
        bool entry;
        union {
            std::array<std::uint64_t, 6> args;
            std::int64_t ret;
        };
    };
}
```

We create a member for the syscall number: for example, `0` for `read` and `1` for `write`. We also store whether this stop occurred due to an entry or exit event. The `args` member stores the arguments to the syscall, and we'll use it only for entry events. The `ret` member stores the return code, and we'll use it only for exit events. We put these last two members in a `union` because they can share storage.

Add `sdb::syscall_information` as an optional member to `sdb::stop_reason`, and add `syscall` as an enumerator value for `sdb::trap_type`:

```
namespace sdb {
    enum class trap_type {
        single_step, software_break, hardware_break, syscall, unknown
    };

    struct stop_reason {
        stop_reason(int wait_status);

        process_state reason;
        std::uint8_t info;
        std::optional<trap_type> trap_reason;
        std::optional<syscall_information> syscall_info;
    };
}
```

If the stop occurred due to a syscall, we will fill in the `syscall_info` member with information about the event.

Next, let's further augment the augmented stop reason. In newer versions of the Linux kernel, a `PTRACE_GET_SYSCALL_INFO` call can get us all the information we need. However, some of you may be stuck using older kernel versions, so we'll calculate these details manually.

We need a way to tell whether we're stopped due to a syscall entry or exit. To record this detail, we'll add a simple Boolean member to `sdb::process` to track whether we're waiting for an exit event:

```
namespace sdb {
    class process {
        --snip--

    private:
        --snip--
        bool expecting_syscall_exit_ = false;
    };
}
```

Now we need to extend the `sdb::process::augment_stop_reason` function to handle syscalls:

```
void sdb::process::augment_stop_reason(sdb::stop_reason& reason) {
    siginfo_t info;
    if (ptrace(PTRACE_GETSIGINFO, pid_, nullptr, &info) < 0) {
        error::send_errno("Failed to get signal info");
    }

    if (reason.info == (SIGTRAP | 0x80)) {
        // Fill in syscall information

     ❶ reason.info = SIGTRAP;
        reason.trap_reason = trap_type::syscall;
        return;
    }

    expecting_syscall_exit_ = false;

    reason.trap_reason = trap_type::unknown;
    if (reason.info == SIGTRAP) {
        --snip--
    }
}
```

After the `PTRACE_GETSIGINFO` call, we check whether we stopped due to a syscall. Recall that the signal communicates this information by being `SIGTRAP | 0x80`. If we stopped due to a syscall, we fill in the syscall information. I've left this as a comment for now; we'll write this code next. After filling in this information, we change `reason.info` from `SIGTRAP | 0x80` to just `SIGTRAP` so it's easier to check later ❶, and we record that the trap occurred

due to a syscall. If we didn't stop due to a syscall, we clear `expecting_syscall_exit_` and then handle the rest of the `SIGTRAP` possibilities, as done previously.

To fill in the syscall information, we'll check the `expecting_syscall_exit_` flag. If it's set, we can assume that a syscall exit caused the stop. In this case, `ptrace` stores the syscall number in the fake `orig_rax` register and the return value in `rax`:

```
--snip--
if (reason.info == (SIGTRAP | 0x80)) {
    auto& sys_info = reason.syscall_info.emplace();
    auto& regs = get_registers();

    if (expecting_syscall_exit_) {
        sys_info.entry = false;
        sys_info.id = regs.read_by_id_as<std::uint64_t>(
            register_id::orig_rax);
        sys_info.ret = regs.read_by_id_as<std::uint64_t>(
            register_id::rax);
     ❶ expecting_syscall_exit_ = false;
    }
    else {
        // Handle entry
    }

    reason.info = SIGTRAP;
    reason.trap_reason = trap_type::syscall;
    return;
}
--snip--
```

We default-construct a `syscall_info` object in the `std::optional<syscall_info>syscall_info` member with the `emplace` function. If we're expecting an exit event, we set `entry` to false, then read the syscall number from `orig_rax` and the return code from `rax`. Finally, we clear `expecting_syscall_exit_` so the next syscall event will be interpreted as an entry event ❶.

For an entry event, we once again store the syscall number in `orig_rax`. According to the SYSV ABI, the system stores the arguments to the syscall in registers `rdi`, `rsi`, `rdx`, `r10`, `r8`, and `r9`, in that order:

```
--snip--
if (reason.info == (SIGTRAP | 0x80)) {
    auto& sys_info = reason.syscall_info.emplace();
    auto& regs = get_registers();

    if (expecting_syscall_exit_) {
        --snip--
    }
    else {
```

```
            sys_info.entry = true;
            sys_info.id = regs.read_by_id_as<std::uint64_t>(
                register_id::orig_rax);

            std::array<register_id, 6> arg_regs = {
                register_id::rdi, register_id::rsi, register_id::rdx,
                register_id::r10, register_id::r8, register_id::r9
            };
            for (auto i = 0; i < 6; ++i) {
                sys_info.args[i] = regs.read_by_id_as<std::uint64_t>(
                    arg_regs[i]);
            }

            expecting_syscall_exit_ = true;
        }

        reason.info = SIGTRAP;
        reason.trap_reason = trap_type::syscall;
        return;
    }
    --snip--
```

If `expecting_syscall_exit_` is `false`, we know we have an entry event. We set `entry` to `true`, read the syscall number from `orig_rax`, and then set up an array of the register IDs that store the arguments for the syscall. We loop over these, reading each register into the `args` member of `sys_info`. Finally, we set `expecting_syscall_exit_` to `true` to interpret the next syscall event as an exit event.

After we've augmented the stop information, we may need to resume the process if the current syscall at which we're stopped isn't one for which the user requested tracing. Add a `maybe_resume_from_syscall` private member to `sdb::process`. In *sdb/include/libsdb/process.hpp*, make the following modifications:

```
namespace sdb {
    class process {
        --snip--
    private:
        --snip--
        sdb::stop_reason maybe_resume_from_syscall(const stop_reason& reason);
        --snip--
    };
}
```

Now implement the member in *sdb/src/process.cpp*:

```
sdb::stop_reason sdb::process::maybe_resume_from_syscall(
      const stop_reason& reason) {
    if (syscall_catch_policy_.get_mode() ==
```

```
         syscall_catch_policy::mode::some) {
        auto& to_catch = syscall_catch_policy_.get_to_catch();
        auto found = std::find(
            begin(to_catch), end(to_catch), reason.syscall_info->id);

        if (found == end(to_catch)) {
            resume();
            return wait_on_signal();
        }
    }

    return reason;
}
```

First, we check the active syscall catch policy. If we're only catching some syscalls and the current one isn't in the list, we resume and call `wait_on_signal`. Note that we don't need to check `syscall_catch_policy::mode::none`, because in that case, we don't continue the process with `PTRACE_SYSCALL` and therefore don't trigger a stop in the first place.

NOTE *The recursive call to `wait_on_signal` could result in the debugger's stack getting very large in the case that a catchpoint is set for only some syscalls and the inferior makes many syscalls that don't match the catch policy before halting for a reason that should be reported back to the user. I've chosen this recursive solution for simplicity for now, but we'll eventually rewrite this function in a way that eliminates this problem.*

Call this function from `sdb::process::wait_on_signal`:

```
sdb::stop_reason sdb::process::wait_on_signal() {
    --snip--
    if (state_ == process_state::stopped) {
        --snip--

        if (reason.info == SIGTRAP) {
            if (reason.trap_reason == trap_type::software_break and ...) {
                --snip--
            }
            else if (reason.trap_reason == trap_type::hardware_break) {
                --snip--
            }
          ❶ else if (reason.trap_reason == trap_type::syscall) {
                reason = maybe_resume_from_syscall(reason);
            }
        }
    }
    --snip--
}
```

We add a branch that checks whether the trap occurred due to a syscall ❶. If so, we call `maybe_resume_from_syscall` and replace the current stop reason with the one it returns. With this work complete, we can add user-facing commands to enable syscall tracing.

Exposing Syscall Catchpoints

We'll support a few different catchpoint commands: `catchpoint syscall` should trace all syscalls, `catchpoint syscall none` should trace no syscalls, and `catchpoint syscall` *`<list of syscalls>`* should trace the syscalls given by the list. The supplied list of syscalls should be comma-separated and allow both syscall numbers and syscall names. To support this command, we'll need functions to map syscall numbers to names. We'll also want to support the reverse operation so we can report user-readable syscall names when the inferior traps.

You might expect the operating system to provide functions for these conversions, but unfortunately, it doesn't. We have to implement them ourselves. This might seem like a lot of work (for example, my system has 360 syscalls), but we don't have to write each mapping manually. With a bit of ingenuity and X-Macros, we can automate it.

The operating system does provide a header file at *asm/unistd_64.h* that contains syscall macros, which look like this:

```
--snip--
#define __NR_read 0
#define __NR_write 1
#define __NR_open 2
#define __NR_close 3
--snip--
```

We can manipulate this file slightly to get what we need. Our goal is to produce something like the following:

```
DEFINE_SYSCALL(read,0)
DEFINE_SYSCALL(write,1)
DEFINE_SYSCALL(open,2)
DEFINE_SYSCALL(close,3)
--snip--
```

We can convert the header file into the desired format with `sed`, a program available on all Unix systems for filtering and transforming text, often by using regular expressions. Here is the command we'll use:

```
$ sed -n -r 's/^#define __NR_(.+) (.+)/DEFINE_SYSCALL(\1,\2)/p' \
/usr/include/x86_64-linux-gnu/asm/unistd_64.h
```

The `-n` flag keeps the command from printing anything. This will let us print only the lines that match the regular expression. Next, `-r` specifies the

use of extended regular expressions, which means we won't need to escape certain special characters.

The form `s/pattern/replacement/p` will substitute text matching the pattern with the given replacement and then print it out. In the regular expression `^#define __NR_(.+) (.+)`, the `^` matches the beginning of the line, `#define __NR_` matches that exact text, and `(.+) (.+)` matches the syscall name and number, capturing them for use in the replacement string. We replace the tokens `\1` and `\2` with the data captured by those parenthesized groups. Finally, we pass the path to *asm/unistd_64.h*, which may differ on your platform.

You can paste the text printed by this command into a file or redirect it to one using `>filename`. Either way, place these `DEFINE_SYSCALL` lines into *sdb/src/include/syscalls.inc* and add a guard at the top. Note that we save this file in the private *include* directory, inside the *sdb/src* directory, and not in the public *include* directory. The file's contents should look like this:

```
#ifndef DEFINE_SYSCALL
#error "This file is intended for textual inclusion with the\
DEFINE_SYSCALL macro defined"
#endif

DEFINE_SYSCALL(read,0)
DEFINE_SYSCALL(write,1)
DEFINE_SYSCALL(open,2)
DEFINE_SYSCALL(close,3)
--snip--
```

This regular expression technique saved us a huge amount of error-prone typing. Now we'll expose syscalls to users with a new header in *sdb/include/libsdb/syscalls.hpp*. This header will declare functions to convert syscall IDs to syscall names and vice versa:

```
#ifndef SDB_SYSCALLS_HPP
#define SDB_SYSCALLS_HPP

#include <string_view>

namespace sdb {
    std::string_view syscall_id_to_name(int id);
    int syscall_name_to_id(std::string_view name);
}

#endif
```

Implement these functions in a new file called *sdb/src/syscalls.cpp*. We'll create a big `switch` statement for `syscall_id_to_name` and use `std::unordered_map` for `syscall_name_to_id`. Start with the former:

```
#include <libsdb/syscalls.hpp>
#include <libsdb/error.hpp>

std::string_view sdb::syscall_id_to_name(int id) {
    switch (id) {
        #define DEFINE_SYSCALL(name,id) case id: return #name;
        #include "include/syscalls.inc"
        #undef DEFINE_SYSCALL
    default: sdb::error::send("No such syscall");
    }
}
```

We define the `DEFINE_SYSCALL` macro to be `case id: return #name;` and include the syscall file inside a `switch` statement. That hash character is the *stringification operator*, which turns the macro parameter into a string to produce code like this:

```
switch (id) {
case 0: return "read";
case 1: return "write";
case 2: return "open";
case 3: return "close";
--snip--
default: sdb::error::send("No such syscall");
}
```

To implement `syscall_name_to_id`, we'll initialize a `std::unordered_map` at startup and then look up IDs in this map when they're requested:

```
#include <unordered_map>

namespace {
    const std::unordered_map<std::string_view, int> g_syscall_name_map = {
        #define DEFINE_SYSCALL(name,id) { #name, id },
        #include "include/syscalls.inc"
        #undef DEFINE_SYSCALL
    };
}

int sdb::syscall_name_to_id(std::string_view name) {
    if (g_syscall_name_map.count(name) != 1)
        sdb::error::send("No such syscall");
    return g_syscall_name_map.at(name);
}
```

We define the `DEFINE_SYSCALL` macro to be `{ #name, id }`, and then include the syscalls file inside the initializer for a `std::unordered_map`. Note the trailing comma, which will separate the map entry initializers. The function

`syscall_name_to_id` looks up the given name in this map and returns the corresponding ID, if one exists. The code generated by that macro looks like this:

```
    const std::unordered_map<std::string_view, int> g_syscall_name_map = {
        { "read", 0 },
        { "write", 1 },
        { "open", 2 },
        { "close", 3},
        --snip--
    };
```

Add *syscalls.cpp* to *sdb/src/CMakeLists.txt*:

```
add_library(libsdb ... syscalls.cpp)
```

Now add a quick test for syscalls to *sdb/test/tests.cpp*:

```
#include <libsdb/syscalls.hpp>

TEST_CASE("Syscall mapping works", "[syscall]") {
    REQUIRE(sdb::syscall_id_to_name(0) == "read");
    REQUIRE(sdb::syscall_name_to_id("read") == 0);
    REQUIRE(sdb::syscall_id_to_name(62) == "kill");
    REQUIRE(sdb::syscall_name_to_id("kill") == 62);
}
```

We ensure that we can translate between the names and IDs of the `read` and `kill` syscalls in both directions. This test should pass.

Now that we can translate between syscall numbers and names, we can add the new `catchpoint` command to *sdb/tools/sdb.cpp* and report the syscall information when the process stops. Add the new option to `handle_command`:

```
        --snip--
        else if (is_prefix(command, "catchpoint")) {
            handle_catchpoint_command(*process, args);
        }
        --snip--
```

We'll defer the syscall catch handling to a `handle_syscall_catchpoint_command` function, as this will make it easier to handle other types of catchpoints later:

```
namespace {
    void handle_catchpoint_command(
          sdb::process& process, const std::vector<std::string>& args) {
        if (args.size() < 2) {
            print_help({ "help", "catchpoint" });
            return;
        }
```

```
            if (is_prefix(args[1], "syscall")) {
                handle_syscall_catchpoint_command(process, args);
            }
        }
    }
```

This function will parse the arguments and set the syscall catch policy on the process:

```
#include <libsdb/syscalls.hpp>

namespace {
    void handle_syscall_catchpoint_command(
          sdb::process& process, const std::vector<std::string>& args) {
        sdb::syscall_catch_policy policy =
            sdb::syscall_catch_policy::catch_all();

        if (args.size() == 3 and args[2] == "none") {
            policy = sdb::syscall_catch_policy::catch_none();
        }
        else if (args.size() >= 3) {
            auto syscalls = split(args[2], ',');
            std::vector<int> to_catch;
            std::transform(begin(syscalls), end(syscalls),
                std::back_inserter(to_catch),
                [](auto& syscall) {
                    return isdigit(syscall[0]) ?
                        sdb::to_integral<int>(syscall).value() :
                        sdb::syscall_name_to_id(syscall);
                });
            policy = sdb::syscall_catch_policy::catch_some(std::move(to_catch));
        }

        process.set_syscall_catch_policy(std::move(policy));
    }
}
```

We define a `policy` variable to store the new syscall catch policy and default to catching all syscalls. If the user passed `none` as an argument, we change this policy to catching no syscalls.

If the user passed additional arguments, those arguments represent syscalls the user wants to trace. We split the arguments on commas and then use the `std::transform` algorithm to process them. This algorithm calls a function on every element of a range, storing the result in some other range. In this case, we call a lambda on each argument to check whether the argument starts with a digit. If so, it must be a syscall number, so we parse it to an integer. Otherwise, it must be a syscall name, so we call `syscall_name_to_id` on it to retrieve the corresponding syscall number.

We also pass a `std::back_inserter(to_catch)` value to `std::transform` to call `push_back` on `to_catch` with the result of every call to the lambda. We then set the policy to catch this list of syscalls. Finally, we call `set_syscall_catch_policy` to set the process's catch policy.

Add some help info to `print_help` about catchpoints:

```
        --snip--
        if (args.size() == 1) {
            std::cerr << R"(Available commands:
    --snip--
    catchpoint  - Commands for operating on catchpoints
    --snip--
)";
        }
        --snip--
        else if (is_prefix(args[1], "catchpoint")) {
            std::cerr << R"(Available commands:
    syscall
    syscall none
    syscall <list of syscall IDs or names>
)";
        }
```

Finally, we can print extended stop information for syscalls. Add a new branch to `get_sigtrap_info` that prints out this new data:

```
        --snip--
        if (reason.trap_reason == sdb::trap_type::syscall) {
            const auto& info = *reason.syscall_info;
            std::string message = " ";
            if (info.entry) {
                message += "(syscall entry)\n";
                message += fmt::format("syscall: {}({:#x})",
                    sdb::syscall_id_to_name(info.id),
                    fmt::join(info.args, ","));
            }
            else {
                message += "(syscall exit)\n";
                message += fmt::format("syscall returned: {:#x}", info.ret);
            }
            return message;
        }
        --snip--
```

If a trap occurs due to a syscall, we check whether it's an entry or an exit event. For entry events, we print out the syscall's name, followed by all of the arguments to the syscall. For exit events, we print out the return code of the syscall. Now let's test syscall catchpoints.

Testing Syscall Catchpoints

The `anti_debugger` program we wrote in the previous chapter is a good target on which to test syscall tracing. Let's use the new catchpoints to try to catch it in the act of putting pineapple on pizza! We can launch the program and trace the `write` syscall to halt the program just before it does so.

Launch the program, set the syscall catch policy, and continue past the first few `write` calls, which aren't part of the main program. Then, you should encounter two calls whose first argument to the syscall is `1` (the specifier for `stdout`). If you continue past these, you'll see that they correspond to the printing of the function pointer and the forbidden pineapple placing:

```
$ tools/sdb test/targets/anti_debugger
Launched process with PID 2812
sdb> catch sys write
sdb> c
--snip--
Process 3499 stopped with signal TRAP at 0x7ffff7ea3a77 (syscall entry)
syscall: write(0x1,0x7fffffffdf30,0x8,0x7ffff7fc3908,0x7ffff7fa9f10,0x7ffff7fc9040)
0x00007ffff7ea3a77: cmp $-0x1000, %rax
0x00007ffff7ea3a7d: jnbe 0x00007FFFF7EA3AD0
0x00007ffff7ea3a7f: ret
0x00007ffff7ea3a80: sub $0x28, %rsp
0x00007ffff7ea3a84: mov %rdx, 0x18(%rsp)
sdb> c
QUUUUProcess 3499 stopped with signal TRAP at 0x7ffff7ea3a77 (syscall exit)
syscall returned: 0x8
0x00007ffff7ea3a77: cmp $-0x1000, %rax
0x00007ffff7ea3a7d: jnbe 0x00007FFFF7EA3AD0
0x00007ffff7ea3a7f: ret
0x00007ffff7ea3a80: sub $0x28, %rsp
0x00007ffff7ea3a84: mov %rdx, 0x18(%rsp)
sdb> c
Process 3499 stopped with signal TRAP at 0x7ffff7e259fc
0x00007ffff7e259fc: mov %eax, %r13d
0x00007ffff7e259ff: neg %r13d
0x00007ffff7e25a02: cmp $-0x1000, %eax
0x00007ffff7e25a07: mov $0x00, %eax
0x00007ffff7e25a0c: cmovbe %eax, %r13d
sdb> c
Process 3499 stopped with signal TRAP at 0x7ffff7ea3a77 (syscall entry)
syscall: write(0x1,0x5555555592a0,0x1e,0x77,0x0,0x5555555592a0)
0x00007ffff7ea3a77: cmp $-0x1000, %rax
0x00007ffff7ea3a7d: jnbe 0x00007FFFF7EA3AD0
0x00007ffff7ea3a7f: ret
0x00007ffff7ea3a80: sub $0x28, %rsp
0x00007ffff7ea3a84: mov %rdx, 0x18(%rsp)
```

```
sdb> c
Putting pineapple on pizza...
Process 3499 stopped with signal TRAP at 0x7ffff7ea3a77 (syscall exit)
syscall returned: 0x1e
0x00007ffff7ea3a77: cmp $-0x1000, %rax
0x00007ffff7ea3a7d: jnbe 0x00007FFFF7EA3AD0
0x00007ffff7ea3a7f: ret
0x00007ffff7ea3a80: sub $0x28, %rsp
0x00007ffff7ea3a84: mov %rdx, 0x18(%rsp)
```

Let's write a simple test to ensure that the process halts for the catchpoints. We'll check that both entry and exit breaks work. Add this case to *sdb/test/tests.cpp*:

```
#include <fcntl.h>

TEST_CASE("Syscall catchpoints work", "[catchpoint]") {
    auto dev_null = open("/dev/null", O_WRONLY);
    auto proc = process::launch("targets/anti_debugger", true, dev_null);

    auto write_syscall = sdb::syscall_name_to_id("write");
    auto policy = sdb::syscall_catch_policy::catch_some({ write_syscall });
    proc->set_syscall_catch_policy(policy);

    proc->resume();
    auto reason = proc->wait_on_signal();

    REQUIRE(reason.reason == sdb::process_state::stopped);
    REQUIRE(reason.info == SIGTRAP);
    REQUIRE(reason.trap_reason == sdb::trap_type::syscall);
    REQUIRE(reason.syscall_info->id == write_syscall);
    REQUIRE(reason.syscall_info->entry == true);

    proc->resume();
    reason = proc->wait_on_signal();

    REQUIRE(reason.reason == sdb::process_state::stopped);
    REQUIRE(reason.info == SIGTRAP);
    REQUIRE(reason.trap_reason == sdb::trap_type::syscall);
    REQUIRE(reason.syscall_info->id == write_syscall);
    REQUIRE(reason.syscall_info->entry == false);

    close(dev_null);
}
```

We don't want the output of this program to pollute the output of our test runner. The usual way to ensure that is to redirect the output of the program to */dev/null*, which is like a black hole that swallows anything sent to

it. To do this, we open a file descriptor for */dev/null* and pass it as the `stdout` descriptor when we launch the process. We then find the ID of the `write` syscall, set the process's catch policy, and resume twice, ensuring that we stop due to syscalls both times, that the syscall is the same ID as the one we supplied, and that the `entry` field is set correctly. Finally, we close the file descriptor for */dev/null*.

Signal and Interrupt Internals

We've improved the debugger's signal handling and implemented syscall catchpoints. Let's take a deep dive into how these features work under the hood on Linux. Take, for example, `ptrace` with `PTRACE_PEEKUSER`: how does this call actually enter kernel space, save the current registers, and communicate them to the user? Well, the Linux kernel defines `ptrace` as a syscall at *linux/kernel/ptrace.c*:

```
SYSCALL_DEFINE4(ptrace, long, request, long, pid, unsigned long, addr,
                unsigned long, data) {
    --snip--
}
```

This rather ugly and complex macro is responsible for setting up metadata about the syscall, such as its name and parameter types, and then defining a function called `sys_<name>`. In this case, the macro defines `sys_ptrace`.

This isn't the function we call when we call `ptrace`, however. We're in user space, not kernel space, so we call a function from the C standard library. In the GNU implementation, `glibc`, this function lives at *glibc/sysdeps/unix/sysv/linux/ptrace.c*. It's actually fairly short:

```
long int
ptrace (enum __ptrace_request request, ...)
{
  long int res, ret;
  va_list ap;
  pid_t pid;
  void *addr, *data;

  va_start (ap, request);
  pid = va_arg (ap, pid_t);
  addr = va_arg (ap, void *);
  data = va_arg (ap, void *);
  va_end (ap);

  if (request > 0 && request < 4)
    data = &ret;

❶ res = INLINE_SYSCALL (ptrace, 4, request, pid, addr, data);
```

```
  if (res >= 0 && request > 0 && request < 4)
    {
      __set_errno (0);
      return ret;
    }

  return res;
}
```

The magic happens at that `INLINE_SYSCALL` macro invocation ❶. Through a bunch of code generation machinery in the `glibc` build system, it ends up executing a `syscall` instruction with `rax` set to `101`.

Why `101`? Essentially, in the Linux kernel, some code generation machinery in the build system defines the numbers for syscalls, whether they're specific to 32-bit or 64-bit architectures or common to both, what the name of each is, and which C function to call when that syscall is executed. You can find the Linux kernel source code at *https://github.com/torvalds/linux*. You'll see an entry in *arch/x86/entry/syscalls/syscall_64.tbl* that looks like this:

```
101    64    ptrace    sys_ptrace
```

This is the `sys_ptrace` function defined as part of the `SYSCALL_DEFINE4` call.

Let's explore the link between the execution of the `syscall` instruction and the call of `sys_ptrace`. The kernel initialization calls a function named `syscall_init`, which lives in *arch/x86/kernel/cpu/common.c*. The first few lines of `syscall_init` look like this:

```
void syscall_init(void)
{
    wrmsr(MSR_STAR, 0, (__USER32_CS << 16) | __KERNEL_CS);
    wrmsrl(MSR_LSTAR, (unsigned long)entry_SYSCALL_64);
    --snip--
}
```

That `wrmsrl` function stands for "write model specific 64-bit register" and refers to registers that allow the configuration of key operating system tasks. `MSR_LSTAR` is the register for configuring system calls. When the processor executes a `syscall` instruction, it executes the function at the address stored in the register. This essentially sets up a callback function in the hardware to call `entry_SYSCALL_64` when the CPU executes a `syscall` instruction.

The `entry_SYSCALL_64` function, in *arch/x86/entry/entry_64.S*, isn't written in C, but in assembly. Its definition is long and detailed, but let's look at a few of its lines:

```
SYM_CODE_START(entry_SYSCALL_64)
    --snip--
    PUSH_AND_CLEAR_REGS rax=$-ENOSYS
    --snip--
    call    do_syscall_64
```

```
    --snip--
SYM_CODE_END(entry_SYSCALL_64)
```

The PUSH_AND_CLEAR_REGS macro is responsible for saving the current state of system registers onto the stack in a format that exactly matches the pt_regs C structure:

```
.macro PUSH_AND_CLEAR_REGS rdx=%rdx rcx=%rcx rax=%rax save_ret=0
    PUSH_REGS rdx=\rdx, rcx=\rcx, rax=\rax, save_ret=\save_ret
    CLEAR_REGS
.endm

.macro PUSH_REGS rdx=%rdx rcx=%rcx rax=%rax save_ret=0
    .if \save_ret
    pushq    %rsi          /* pt_regs->si */
    movq     8(%rsp), %rsi /* Temporarily store the return address in %rsi */
    movq     %rdi, 8(%rsp) /* pt_regs->di (overwriting original return address) */
    .else
    pushq    %rdi          /* pt_regs->di */
    pushq    %rsi          /* pt_regs->si */
    .endif
    pushq    \rdx          /* pt_regs->dx */
    pushq    \rcx          /* pt_regs->cx */
    pushq    \rax          /* pt_regs->ax */
    pushq    %r8           /* pt_regs->r8 */
    pushq    %r9           /* pt_regs->r9 */
    pushq    %r10          /* pt_regs->r10 */
    pushq    %r11          /* pt_regs->r11 */
    pushq    %rbx          /* pt_regs->rbx */
    pushq    %rbp          /* pt_regs->rbp */
    pushq    %r12          /* pt_regs->r12 */
    pushq    %r13          /* pt_regs->r13 */
    pushq    %r14          /* pt_regs->r14 */
    pushq    %r15          /* pt_regs->r15 */
    UNWIND_HINT_REGS

    .if \save_ret
    pushq    %rsi          /* Return address on top of stack */
    .endif
.endm
```

The entry_SYSCALL_64 function then calls do_syscall_64, which is hooked up to the code generated from that syscall table and calls sys_ptrace. So, we've traveled from user space, through glibc, and into kernel space through the CPU syscall handler, where we saved the state of the registers and called sys_ptrace.

The sys_ptrace function itself handles some architecture-independent commands, like PTRACE_ATTACH and PTRACE_INTERRUPT, and carries out security

checks to make sure the caller is actually permitted to call ptrace. Then, it dispatches to the architecture-specific handler, which in our case is arch_ptrace, inside *arch/x86/kernel/ptrace.c*. The code for PTRACE_POKEUSER looks like this:

```
case PTRACE_POKEUSR: /* Write the word at location addr in the USER area */
    ret = -EIO;
    if ((addr & (sizeof(data) - 1)) || addr >= sizeof(struct user))
        break;

    if (addr < sizeof(struct user_regs_struct))
        ret = putreg(child, addr, data);
    else if (addr >= offsetof(struct user, u_debugreg[0]) &&
             addr <= offsetof(struct user, u_debugreg[7])) {
        addr -= offsetof(struct user, u_debugreg[0]);
        ret = ptrace_set_debugreg(child,
                addr / sizeof(data), data);
    }
    break;
```

The PTRACE_POKEUSER command gives the illusion that it's writing a solid mapped region of memory, but on x64, this is actually a total lie. If you try to read anything that isn't a GPR or debug register, it returns an error code. Remember how, in Chapter 5, we had to use PTRACE_SETFPREGS instead of PTRACE_POKEUSER for FPU registers? This is why.

The putreg function contains several levels of indirection but ends up writing to the pt_regs struct shown earlier, which was constructed on the stack when the syscall handler was invoked.

Control passes back to the assembly code in *arch/x86/entry/entry_64.S* when the syscall exits. It can choose from a few different methods to get back to the user-mode process, but in all of them, it first calls POP_REGS:

```
.macro POP_REGS pop_rdi=1
    popq %r15
    popq %r14
    popq %r13
    popq %r12
    popq %rbp
    popq %rbx
    popq %r11
    popq %r10
    popq %r9
    popq %r8
    popq %rax
    popq %rcx
    popq %rdx
    popq %rsi
    .if \pop_rdi
    popq %rdi
```

```
    .endif
.endm
```

This function restores all the registers from the stack. Because `putreg` wrote all the changes into this structure inside the call to `sys_ptrace`, this restoration writes the desired values into the system registers, completing the process.

Signals touch very similar parts of the Linux kernel. In particular, let's see how a `SIGTRAP` gets sent to a process when it executes an `int3` instruction. The `int3` instruction is a special version of the `int <n>` instruction. When the operating system starts, it registers a bunch of interrupt handlers with the hardware that get called in various circumstances, and `int <n>` generates a call to the interrupt handler at index *n*. In x64, `int 3` is given its own instruction so it can be encoded as a single byte, which makes setting breakpoints easier.

The kernel sets up interrupt handlers in *arch/x86/kernel/idt.c*, where it defines `asm_exc_int3` as the handler for interrupt 3:

```
static const __initconst struct idt_data early_idts[] = {
    --snip--
    SYSG(X86_TRAP_BP, asm_exc_int3),
    --snip--
};

void __init idt_setup_early_traps(void)
{
    idt_setup_from_table(
        idt_table, early_idts, ARRAY_SIZE(early_idts), true);
    load_idt(&idt_descr);
}
```

You'll find `x86_TRAP_BP` defined as `3` in *arch/x86/include/asm/trapnr.h*, so this file sets the handler for interrupt 3 to be `asm_exc_int3`. In *arch/x86/kernel/traps.c*, this function sends a `SIGTRAP` to the process:

```
DEFINE_IDTENTRY_RAW(exc_int3)
{
    --snip--
        do_int3_user(regs);
    --snip--
}

static void do_int3_user(struct pt_regs *regs)
{
    if (do_int3(regs))
        return;

    cond_local_irq_enable(regs);
    do_trap(X86_TRAP_BP, SIGTRAP, "int3", regs, 0, 0, NULL);
```

```
    cond_local_irq_disable(regs);
}
```

That `do_trap` call triggers the `SIGTRAP`. We've successfully traced the execution of the `int3` instruction all the way to the kernel sending `SIGTRAP` to the process!

Summary

In this chapter, you extended your debugger to handle signals from the command line gracefully and report detailed information about signals to the user. You also added support for syscall tracing to the debugger. Finally, we performed a deep dive into how the Linux kernel handles signals and interrupts, and you saw the real code that gets executed in the operating system to facilitate these tools.

In the next chapter, you'll implement an ELF parser, which is the first step to adding source-level debugging support to your debugger.

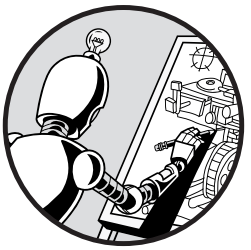
Check Your Knowledge

1. Which function installs signal handlers on a process?
2. Which signal gets sent to a process when you press CTRL-C at a command line?
3. What are reentrant functions?
4. Which function sets the process group of a process?
5. Which debug register on x64 is the status register?
6. Which `ptrace` request causes it to halt on syscall entry and exit?

11

OBJECT FILES

The doom of the Elves is to be immortal,
to love the beauty of the world,
to bring it to full flower with their gifts of delicacy and perfection
–J.R.R. Tolkien, *The Silmarillion*



It's time to read the sacred text of ELF. I'm not talking about *The Silmarillion*, but rather the System V ABI base specification, which describes the ELF object format, and the x64 supplement that specifies the x64-specific parts.

The debugger needs to parse the ELF files relevant to the running process so it can retrieve debug information, locate the code for a given function, and more. To extract information from ELF files, we could rely on one of the many existing open source ELF parsers. However, if we're reinventing the wheel, we may as well reinvent the chisel used to carve the rock as well. In this chapter, we'll implement a new class, `sdb::elf`, that will be able to parse ELF files and expose the data contained within to the rest of the library.

What Is an ELF?

ELF files can be executable programs, shared libraries, static libraries (called *relocatable files* in the specifications), and *core dumps* (snapshots of memory and registers taken to debug a process that has crashed). To accommodate all of these flavors, ELF files use sections and segments to communicate information relevant to both linking and executing binaries.

Sections and Segments

ELF communicates link-time information in *sections*, named regions of the binary accompanied by relevant flags and attributes. These flags contain information such as whether a section holds code or strings and whether a linker can merge the section with identical ones found in other ELF files it's linking together.

ELF files communicate execution-time information in *segments*. These don't have names but contain flags and attributes relevant to program execution, such as where to load the program into memory and what the permissions on that block of memory should be.

Importantly, sections and segments provide different views of the same data. Let me explain with an example. If you run `readelf --sections --segments test/targets/anti_debugger`, you'll get something like this:

```
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000000318  00000318
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.gnu.pr[...] NOTE             0000000000000338  00000338
       0000000000000030  0000000000000000   A       0     0     8
  [ 3] .note.gnu.bu[...] NOTE             0000000000000368  00000368
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .note.ABI-tag     NOTE             000000000000038c  0000038c
       0000000000000020  0000000000000000   A       0     0     4
  [ 5] .gnu.hash         GNU_HASH         00000000000003b0  000003b0
       0000000000000024  0000000000000000   A       6     0     8

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
```

```
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x00000000000005f0 0x00000000000005f0  R      0x1000
  LOAD           0x0000000000001000 0x0000000000001000 0x0000000000001000
                 0x0000000000000149 0x0000000000000149  R E    0x1000
  LOAD           0x0000000000002000 0x0000000000002000 0x0000000000002000
                 0x00000000000000c0 0x00000000000000c0  R      0x1000

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.gnu.build-id .gnu.hash .dynsym .dynstr
   03     .text
   04     .eh_frame
   05     .dynamic
```

I've shortened the output to the first five sections and segments for brevity. In it, you can see three blocks of information: the *section headers*, which describe the sections in the file; the *program headers*, which describe the segments in the file; and the mapping between sections and segments.

Consider, for example, the bytes from `0x318` to `0x334`. These bytes are part of both the `.interp` section and segment 1:

```
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 1] .interp           PROGBITS         0000000000000318  00000318
       000000000000001c  0000000000000000   A       0     0     1

  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1

 Section to Segment mapping:
   01     .interp
```

The `.interp` section spans the file offsets between `0x318` and `0x334`. Segment 1 is of type `INTERP` and spans the same file offsets. The section-to-segment map shows that segment 1 and the `.interp` section relate to the same data.

Note that the mapping of sections and segments isn't one-to-one. Segment 0, the program header, doesn't map to a section. Segment 2 covers multiple sections, which will be loaded into memory in one block. The `.interp` section appears in multiple segments, as it gives the path to the dynamic loader (in the `INTERP` segment type) and is loaded into memory (in the `LOAD` segment type).

In this chapter, we'll focus entirely on sections, rather than segments. You'll learn more about segments and program headers in Chapter 17, when we consider program loading in more detail. First, though, let's look at the overall structure of an ELF file.

File Structure

Every ELF file begins with an ELF header that gives high-level information about the file, such as where to find the section and program headers, what architecture the file is for, and what flavor of object file it is (executable, shared library, relocatable, or core dump). Usually the program header comes next, which, as we've seen, describes the segments. Following the program header is the file's main data, which may be viewed as either sections or segments. Finally comes the section header, which describes the sections. As such, we can take two views of an ELF file: a linking view and an execution view, as illustrated in Figure 11-1.

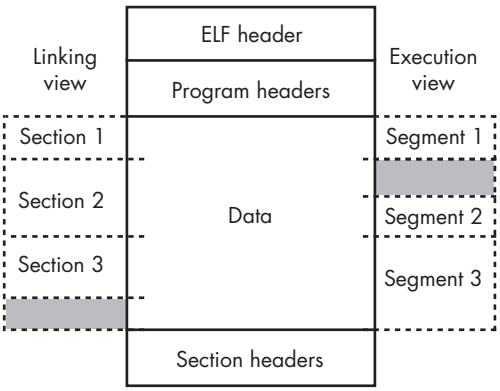


Figure 11-1: The layout of an ELF file

Some sections are special, and the ELF specification details their encoding. Examples include *string tables*, used to hold textual data, and *symbol tables*, used to describe entities like functions and variables for linking purposes. We'll look at these formats in detail later in this chapter.

ELF Header Parsing

Let's begin our parser by parsing the ELF header. Throughout this code, we'll rely on the types already specified in the *elf.h* header provided by Linux. This will minimize the amount of boring busywork we'll need to do. The *elf.h* header specifies data types used throughout the header:

```
typedef uint64_t    Elf64_Addr;
typedef uint16_t    Elf64_Half;
typedef int16_t     Elf64_SHalf;
typedef uint64_t    Elf64_Off;
typedef uint32_t    Elf64_Sword;
```

```
typedef uint32_t    Elf64_Word;
typedef uint64_t    Elf64_Xword;
typedef int64_t     Elf64_Sxword;
```

These convenience typedefs save you from having to remember the specific integral types used for things like addresses. The header file also defines the `Elf64_Ehdr` type for 64-bit ELF headers:

```
typedef struct elf64_hdr {
    unsigned char e_ident[EI_NIDENT]; // ELF "magic number"
    Elf64_Half e_type;          // Executable, shared lib, relocatable, core
    Elf64_Half e_machine;       // Architecture, e.g., EM_X86_64 for x64
    Elf64_Word e_version;       // Version, must be 1
    Elf64_Addr e_entry;         // Entry point virtual address
    Elf64_Off e_phoff;          // Program header table file offset
    Elf64_Off e_shoff;          // Section header table file offset
    Elf64_Word e_flags;         // Processor-specific flags
    Elf64_Half e_ehsize;        // ELF header size
    Elf64_Half e_phentsize;     // Program header size
    Elf64_Half e_phnum;         // Number of program headers
    Elf64_Half e_shentsize;     // Section header size
    Elf64_Half e_shnum;         // Number of section headers
    Elf64_Half e_shstrndx;      // Section that holds the string table
} Elf64_Ehdr;
```

The *magic number* is a set sequence of bytes used to identify that this blob of binary is indeed an ELF file. Consult the inline comments for the meanings of the other fields.

Create a new file at *sdb/include/libsdb/elf.hpp* to begin the implementation of `sdb::elf`:

```
#ifndef SDB_ELF_HPP
#define SDB_ELF_HPP

#include <filesystem>
#include <elf.h>

namespace sdb {
    class elf {
    public:
        elf(const std::filesystem::path& path);
        ~elf();

        elf(const elf&) = delete;
        elf& operator=(const elf&) = delete;

        std::filesystem::path path() const { return path_; }
        const Elf64_Ehdr& get_header() const { return header_; }
```

```
    private:
        int fd_;
        std::filesystem::path path_;
        std::size_t file_size_;
        std::byte* data_;
        Elf64_Ehdr header_;
    };
}

#endif
```

The constructor takes the path to an ELF file on disk. We declare a destructor because we'll need to do some cleanup when destroying an `elf` object. These `elf` objects should be unique, so we delete the copy operations, then we define members to store an open file descriptor for the ELF file, the path to and size of the file, a pointer to the file's bytes, and the header information.

ELF files can be megabytes or even gigabytes in size. One convenient way to handle large files is to use the `mmap` syscall to map them into the virtual memory of our process, letting us pretend we've read the file completely into memory. The operating system can then handle loading necessary parts of the file from disk when required. Create an *sdb/src/elf.cpp* file and implement the constructor such that it `mmap`s the ELF file and copies the header bytes into the `header_` member:

```
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <libsdb/elf.hpp>
#include <libsdb/error.hpp>
#include <libsdb/bit.hpp>

sdb::elf::elf(const std::filesystem::path& path) {
    path_ = path;

    if ((fd_ = open(path.c_str(), O_RDONLY)) < 0) {
        error::send_errno("Could not open ELF file");
    }

    struct stat stats;
    if (fstat(fd_, &stats) < 0) {
        error::send_errno("Could not retrieve ELF file stats");
    }
    file_size_ = stats.st_size;

    void* ret;
```

```
    if ((ret = mmap(0, file_size_, PROT_READ, MAP_SHARED, fd_, 0)) == MAP_FAILED) {
        close(fd_);
        error::send_errno("Could not mmap ELF file");
    }
    data_ = reinterpret_cast<std::byte*>(ret);

    std::copy(data_, data_ + sizeof(header_), as_bytes(header_));
}
```

First, we open the file with the `open` syscall. We then find the size of the file with `fstat`. This function takes a file descriptor and a pointer to a `stat` object and fills in the `stat` object with a bunch of metadata about the file. Note that when you declare `stats`, you must write `struct stat stats` rather than just `stat stats` because a function named `stat` already exists in the global namespace; the `struct` keyword disambiguates the declaration.

After opening the file and getting its size, we map it into virtual memory. We pass `0` as the first argument, which instructs `mmap` to choose where to map this memory. We pass the size of the file and tell `mmap` to map it as read-only (`PROT_READ`). We also need to pass a flag indicating how to deal with writes, but the flag we choose doesn't matter in this case because the memory is mapped as read-only, so we pass `MAP_SHARED`. If the underlying file is modified while we have it mapped, the memory map will be updated, but we won't attempt to handle this case. Finally, we pass the file descriptor for the file we want to map and an offset of `0` to map the entire file.

When the mapping completes, we cast the returned pointer to a pointer to bytes, store it in the `data_` member, and copy the header into the `header_` member. Now we can read from `data_` to retrieve information from any location inside the ELF file without having to copy the data ourselves. Neat!

Make sure you close the file and unmap the memory in the destructor:

```
sdb::elf::~elf() {
    munmap(data_, file_size_);
    close(fd_);
}
```

Also add *elf.cpp* to *sdb/src/CMakeLists.txt*:

```
add_library(libsdb ... elf.cpp)
```

Next, we can parse the section header.

Section Header Parsing

The section header table lives at the offset of `header_.e_shoff` from the start of the file. In other words, the `header_.e_shoff` field gives the file offset of the section header table. Each entry is of type `Elf64_Shdr`:

```
typedef struct elf64_shdr {
    Elf64_Word sh_name;        // Section name as string table index
    Elf64_Word sh_type;        // Type, e.g., code, string/symbol table
```

```
    Elf64_Xword sh_flags;      // Section attributes, e.g., writable during execution
    Elf64_Addr sh_addr;        // Virtual load address
    Elf64_Off sh_offset;       // File offset
    Elf64_Xword sh_size;       // Section size in bytes
    Elf64_Word sh_link;        // Index of an associated section
    Elf64_Word sh_info;        // Additional info, e.g., section group info
    Elf64_Xword sh_addralign;  // Section alignment
    Elf64_Xword sh_entsize;    // Entry size if the section holds a table
} Elf64_Shdr;
```

We'll add a new member to `sdb::elf` to hold section headers and a private member function to parse them:

```
#include <vector>

namespace sdb {
    class elf {
    --snip--

    private:
        void parse_section_headers();

        --snip--
        std::vector<Elf64_Shdr> section_headers_;
    };
}
```

Implement this member in *sdb/src/elf.cpp* by copying the data from `header_.e_shoff`. You'll find the number of section headers at `header_.e_shnum`. Also add a call to `parse_section_headers` at the end of the `elf` constructor:

```
sdb::elf::elf(const std::filesystem::path& path) {
    --snip--
    parse_section_headers();
}

void sdb::elf::parse_section_headers() {
    section_headers_.resize(header_.e_shnum);
    std::copy(data_ + header_.e_shoff,
        data_ + header_.e_shoff + sizeof(Elf64_Shdr) * header_.e_shnum,
        reinterpret_cast<std::byte*>(section_headers_.data()));
}
```

We resize the `section_headers_` member depending on how many sections the ELF header says there are. We then copy all of these headers into the vector. To determine the number of bytes to copy, we start at the offset given in the header and multiply the size of a section header by the number of sections there are.

We must handle one special case, which occurs when a file contains many sections. ELF reserves a range of section indices for special uses, and if the number of sections reaches that limit, the file stores the number of sections elsewhere. More concretely, if a file has `0xff00` sections or more, it sets `e_shnum` to `0` and stores the number of sections in the `sh_size` field of the first section header. To accommodate this fact, modify the code you just wrote like this:

```
void sdb::elf::parse_section_headers() {
    auto n_headers = header_.e_shnum;
    if (n_headers == 0 and header_.e_shentsize != 0) {
        n_headers = from_bytes<Elf64_Shdr>(data_ + header_.e_shoff).sh_size;
    }

    section_headers_.resize(n_headers);
    std::copy(data_ + header_.e_shoff,
        data_ + header_.e_shoff + sizeof(Elf64_Shdr) * n_headers,
        reinterpret_cast<std::byte*>(section_headers_.data()));
}
```

If the file specifies the number of headers to be zero but also specifies the section header size element, there must be more than `0xff00` sections, so we read the `sh_size` field of the first section header to get the real number of headers. Then, we continue as before.

Now that we've parsed the section headers, we can write functions to access the symbol table and string table, which are special sections. Let's start with the string table.

String Tables

A string table is a list of null-terminated strings. It allows the rest of the ELF file to refer to a string using a byte offset into the string table. Generally, an ELF file has two string tables: the *general* string table, which lives in the `.strtab` section, and the *section name* string table, which lives in the `.shstrtab` section. Certain files may also have a `.dynstr` table containing strings used for dynamic linking. Table 11-1 shows an example of a string table.

Table 11-1: An Example String Table

Index	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	\0	M	i	l	k	s	h	a	k	e
10	\0	s	h	a	k	e	\0	n	o	\0

String tables always begin with a null terminator, which means that references to index 0 encode empty strings. As you can see, the table can duplicate strings and lets us reference substrings. For example, some strings we could reference in this table include "Milkshake" (index 1), "shake" (index 5),

"shake" (index 11), and "no" (index 17). Let's begin by parsing the section names string table.

Getting Section Names

To add support for retrieving section names, add a `get_section_name` function to `sdb::elf` in *sdb/include/libsdb/elf.hpp*:

```
#include <string_view>

namespace sdb {
    class elf {
    public:
        --snip--
        std::string_view get_section_name(std::size_t index) const;
        --snip--
    };
}
```

Now implement this function in *sdb/src/elf.cpp* by finding the section name string table with the index given in the ELF header, then getting the string at the given offset in that section:

```
std::string_view sdb::elf::get_section_name(std::size_t index) const {
    auto& section = section_headers_[header_.e_shstrndx];
    return { reinterpret_cast<char*>(data_) + section.sh_offset + index };
}
```

The `e_shstrndx` field of the ELF header identifies the section that stores the string table for section names (usually `.shstrtab`). We grab the section header for this section, then return the string that starts at the given index into that section.

Building a Section Map

Next, let's build a map from section names to section headers. This will make it much easier to look up sections. Add a new private member to hold the section map to `sdb::elf` and some member functions to retrieve section information by section name. Also add a private member function to build the map. Make the modifications in *sdb/include/libsdb/elf.hpp*:

```
#include <unordered_map>
#include <optional>
#include <libsdb/types.hpp>

namespace sdb {
    class elf {
    public:
        --snip--
```

```
        std::optional<const Elf64_Shdr*>
        get_section(std::string_view name) const;
        span<const std::byte> get_section_contents(std::string_view name) const;

    private:
        --snip--
        void build_section_map();
        --snip--
        std::unordered_map<std::string_view, Elf64_Shdr*> section_map_;
    };
}
```

The `get_section` function will return a pointer to the section header for the section with the given name, if one exists, while the `get_section_contents` function will return a span of bytes with the data for that section. We'll call the `build_section_map` private member from the constructor and it will initialize the `section_map_` member, which maps the section names to the section headers.

Implement `build_section_map` by getting the section name for each section from the string table and updating the private mapping. Also call this function from the `elf` constructor:

```
sdb::elf::elf(const std::filesystem::path& path) {
    --snip--
    build_section_map();
}

void sdb::elf::build_section_map() {
    for (auto& section : section_headers_) {
        section_map_[get_section_name(section.sh_name)] = &section;
    }
}
```

We loop over all of the section headers and add an entry in the map that goes from the name of that section to a pointer to the section's header. Then we implement `get_section`, which merely performs a checked map lookup:

```
std::optional<const Elf64_Shdr*>
sdb::elf::get_section(std::string_view name) const {
    if (section_map_.count(name) == 0) {
        return std::nullopt;
    }
    return section_map_.at(name);
}
```

If the section map doesn't contain the given name, we return an empty optional. Otherwise, we return a pointer to the relevant section header. The `get_section_contents` function can defer to `get_section`:

```
sdb::span<const std::byte>
sdb::elf::get_section_contents(std::string_view name) const {
```

```
    if (auto sect = get_section(name); sect) {
        return { data_ + sect.value()->sh_offset, sect.value()->sh_size};
    }
    return { nullptr, std::size_t(0) };
}
```

We try to get the section with the given name. The `if (auto var = init; var)` syntax allows us to initialize a variable and test its value without having to declare the variable outside the scope of the `if` statement. If we find a section with the given name, we return a span representing the region contained in the section. Otherwise, we return an empty span.

Now that we have the ability to perform section lookups, we'll locate the `.strtab` or `.dynstr` sections and add support for looking up strings in the general string table.

Parsing the General String Table

The general string table lives in the `.strtab` section, though some ELF files may have only an abbreviated `.dynstr` section instead of a `.strtab` section. To retrieve a string from the table given an index, add a `get_string` function to `sdb::elf`:

```
namespace sdb {
    class elf {
    public:
        --snip--
        std::string_view get_string(std::size_t index) const;
        --snip--
    };
}
```

The function will resemble `get_section_name`, but it will look up the string in the `.strtab` or `.dynstr` section instead of the `.shstrtab` section. Implement it in *sdb/src/elf.cpp*:

```
std::string_view sdb::elf::get_string(std::size_t index) const {
    auto opt_strtab = get_section(".strtab");
    if (!opt_strtab) {
        opt_strtab = get_section(".dynstr");
        if (!opt_strtab) return "";
    }
    return {
        reinterpret_cast<char*>(data_) + opt_strtab.value()->sh_offset + index
    };
}
```

We grab the section header corresponding to the `.strtab` or `.dynstr` section and then return the string that begins at the given offset into that section.

NOTE *Although most ELF files have a general string table, in some cases they may allocate different string tables to different sections. The more robust way to handle string tables is to read the* `sh_link` *field of the section header to which the string table index belongs, which provides the section index of the string table for that section. I've opted to assume there's a general string table for simplicity.*

Now, before we move on to parsing symbol tables, we need to think more carefully about the kinds of addresses we have in our system.

File Addresses, File Offsets, and Virtual Addresses

Currently, we have a type that stores virtual addresses (`sdb::virt_addr`) but no type to store file addresses or file offsets. Because the symbol table gives us a function's address information in the form of file addresses, we'll require a type to represent them. We'll also need a type to represent file offsets, as the DWARF information we'll parse in the next chapter uses them heavily.

Let's write an `sdb::file_addr` type that stores a file address and implement routines for translating between file addresses and virtual addresses. We'll also write an `sdb::file_offset` type that stores a file offset. Add a function to `sdb::virt_addr` in *sdb/include/libsdb/types.hpp* that converts the virtual address to an `sdb::file_addr` type in a given ELF file:

```
namespace sdb {
    class file_addr;
    class elf;
    class virt_addr {
    public:
        --snip--
        file_addr to_file_addr(const elf& obj) const;
        --snip--
    };
}
```

We forward-declare `sdb::file_addr` and `sdb::elf` so that we can use them in the function declaration for `to_file_addr`. Now define `sdb::file_addr` in the same file:

```
namespace sdb {
    class file_addr {
    public:
        file_addr() = default;
        file_addr(const elf& obj, std::uint64_t addr)
            : elf_(&obj), addr_(addr) {}

        std::uint64_t addr() const {
            return addr_;
        }
        const elf* elf_file() const {
            return elf_;
        }
```

```
        virt_addr to_virt_addr() const;

    private:
        const elf* elf_ = nullptr;
        std::uint64_t addr_ = 0;
    };
}
```

This type resembles `sdb::virt_addr` but additionally stores a pointer to the ELF file. The address is relative to the load address of this file.

We'll also write operator overloads to make using this type more ergonomic. This time, we need to handle the pointer to the ELF file as well. For the equality operators, we'll test the equality of the ELF pointers. However, the relative comparison operators don't make sense if the ELF files for the addresses don't match; this mismatch would indicate a programming error, so we'll assert that the ELF pointers are equal:

```
#include <cassert>

namespace sdb {
    class file_addr {
    public:
        --snip--
        file_addr operator+(std::int64_t offset) const {
            return file_addr(*elf_, addr_ + offset);
        }
        file_addr operator-(std::int64_t offset) const {
            return file_addr(*elf_, addr_ - offset);
        }
        file_addr& operator+=(std::int64_t offset) {
            addr_ += offset;
            return *this;
        }
        file_addr& operator-=(std::int64_t offset) {
            addr_ -= offset;
            return *this;
        }
        bool operator==(const file_addr& other) const {
            return addr_ == other.addr_ and elf_ == other.elf_;
        }
        bool operator!=(const file_addr& other) const {
            return addr_ != other.addr_ or elf_ != other.elf_;
        }
        bool operator<(const file_addr& other) const {
            assert(elf_ == other.elf_);
            return addr_ < other.addr_;
        }
```

```
        bool operator<=(const file_addr& other) const {
            assert(elf_ == other.elf_);
            return addr_ <= other.addr_;
        }
        bool operator>(const file_addr& other) const {
            assert(elf_ == other.elf_);
            return addr_ > other.addr_;
        }
        bool operator>=(const file_addr& other) const {
            assert(elf_ == other.elf_);
            return addr_ >= other.addr_;
        }
        --snip--
    };
}
```

The `file_offset` type is much simpler, as we won't need to do many operations on this type; we'll use it to make sure we don't mix up file offsets and file addresses in our code:

```
namespace sdb {
    class file_offset {
    public:
        file_offset() = default;
        file_offset(const elf& obj, std::uint64_t off)
            : elf_(&obj), off_(off) {}

        std::uint64_t off() const {
            return off_;
        }
        const elf* elf_file() const {
            return elf_;
        }

    private:
        const elf* elf_ = nullptr;
        std::uint64_t off_ = 0;
    };
}
```

The type is essentially the same as `sdb::file_addr`, but it stores an offset instead of an address and doesn't need the comparison operators or conversion functions.

To convert between file addresses and virtual addresses, we need to know where the ELF file is loaded in virtual memory. Determining this location requires some nuance, so read the next few paragraphs very carefully.

We must consider three different kinds of addresses: absolute offsets from the start of the object file (corresponding to the `sdb::file_offset` type),

virtual addresses specified in the ELF file (corresponding to the `sdb::file_addr` type), and the actual virtual addresses in the executing program (corresponding to the `sdb::virt_addr` type).

Contiguous sections in the ELF file don't necessarily map contiguously into memory; gaps could exist between them. This commonly occurs when the segments corresponding to those sections have different memory permissions assigned to them. For example, the segment holding the `.data` section should be mapped as read/write, but the segment holding the `.text` section is usually mapped as read/execute. Linux assigns memory permissions to *pages* of memory, which are 4,096 bytes in size on x64, so if sections with different permissions aren't aligned to 4,096 bytes, the system must load them with gaps between them.

Parts of the ELF file that reside in different segments may refer to each other using file addresses, and we need these addresses to remain valid no matter where the segments are loaded in memory. So, the ELF file specifies the file address at which each segment gets loaded, and to ensure that cross-references between segments work, the file addresses and the real virtual addresses will only ever differ by a single offset for the entire ELF file, called the *load bias*. That is, the gaps specified in the ELF file and the gaps between the loaded segments will always be the same. See Figure 11-2 for a visualization of this principle.

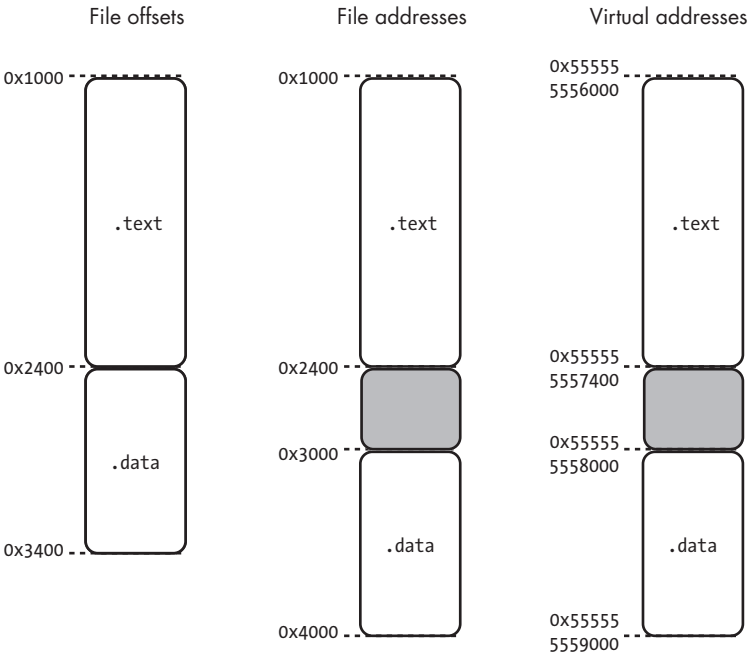


Figure 11-2: Possible layouts of ELF sections in memory

As such, to translate a file address into a virtual address, we calculate the load bias of the ELF file and add that to the file address. Let's add

functionality to `sdb::elf` for tracking the ELF load bias. Add these members to *sdb/include/libsdb/elf.hpp*:

```
namespace sdb {
    class elf {
    public:
        --snip--
        virt_addr load_bias() const {
            return load_bias_;
        }
        void notify_loaded(virt_addr address) {
            load_bias_ = address;
        }

    private:
        --snip--
        virt_addr load_bias_;
    };
}
```

We'll deal with actually finding the load bias when we hook the ELF parser into the rest of the debugger. For now, we can assume that `load_bias_` is correct and use it to translate between virtual addresses and file addresses.

We'll need a helper function to retrieve the section to which a given file or virtual address belongs. Declare two overloads for `get_section_containing_address` in *sdb/include/libsdb/elf.hpp*:

```
namespace sdb {
    class elf {
    public:
        --snip--
        const Elf64_Shdr* get_section_containing_address(
            file_addr addr) const;
        const Elf64_Shdr* get_section_containing_address(
            virt_addr addr) const;
    };
}
```

Then, define them in *sdb/src/elf.cpp*:

```
const Elf64_Shdr* sdb::elf::get_section_containing_address(
      file_addr addr) const {
    if (addr.elf_file() != this) return nullptr;
    for (auto& section : section_headers_) {
        if (section.sh_addr <= addr.addr() and
            section.sh_addr + section.sh_size > addr.addr()) {
            return &section;
        }
    }
```

```
    return nullptr;
}

const Elf64_Shdr* sdb::elf::get_section_containing_address(virt_addr addr) const {
    for (auto& section : section_headers_) {
        if (load_bias_ + section.sh_addr <= addr and
            load_bias_ + section.sh_addr + section.sh_size > addr) {
            return &section;
        }
    }
    return nullptr;
}
```

The file address helper finds the section at which the given address lies, between the `sh_addr` field of the section header and `sh_addr + sh_size`, which gives the end of the section. The virtual address helper uses the `sh_addr` field, offset by the ELF load bias. Implement `sdb::virt_addr::to_file_addr` and `sdb::file_addr::to_virt_addr` in a new file called *sdb/src/types.cpp*:

```
#include <libsdb/types.hpp>
#include <libsdb/elf.hpp>
#include <cassert>

sdb::virt_addr sdb::file_addr::to_virt_addr() const {
    assert(elf_ && "to_virt_addr called on null address");
    auto section = elf_->get_section_containing_address(*this);
    if (!section) return virt_addr{};
    return virt_addr { addr_ + elf_->load_bias().addr() };
}

sdb::file_addr sdb::virt_addr::to_file_addr(const elf& obj) const {
    auto section = obj.get_section_containing_address(*this);
    if (!section) return file_addr{};
    return file_addr{ obj, addr_ - obj.load_bias().addr()};
}
```

The `to_virt_addr` function first asserts that `elf_` is not null, since this would indicate that something has gone very wrong. It then tries to locate the section that contains the file address to ensure that this file address is valid for the ELF file. If there isn't one, we return an empty virtual address. Otherwise, we add the stored address to the load bias to compute the real virtual address. The `to_file_addr` function does the opposite, subtracting the load bias to compute the file address.

Add this new file to *sdb/src/CMakeLists.txt*:

```
add_library(libsdb ... syscalls.cpp elf.cpp types.cpp)
```

While we're dealing with file addresses, let's add a little helper function to `sdb::elf` to retrieve the start file address of a section by name. Declare it in *sdb/include/libsdb/elf.hpp*:

```
namespace sdb {
    class elf {
    public:
        --snip--
        std::optional<file_addr> get_section_start_address(
            std::string_view name) const;
        --snip--
    };
}
```

Define it in *sdb/src/elf.cpp*:

```
std::optional<sdb::file_addr> sdb::elf::get_section_start_address(
    std::string_view name) const {
    if (auto sect = get_section(name); sect) {
        return file_addr{ *this, sect.value()->sh_addr };
    }
    return std::nullopt;
}
```

If the section exists, we return a file address containing its start address. Otherwise, we return an empty optional. Now that we have a type to represent file addresses, we can implement a parser for ELF symbol tables.

Parsing the Symbol Table

A symbol table contains linking-related information about global program entities such as variables and functions. Rather than the high-level source code details you might get from debug information, the symbol table includes more basic aspects, such as the following:

- The size of the entity (for example, the object size or number of bytes in a function's machine code)
- Whether the entity is available to other ELF files that might link against it
- The category to which this entity belongs (for example, function, variable, ELF section, or file)

An ELF file may have two symbol tables: a *complete* symbol table named `.symtab` with `SHT_SYMTAB` as the section header's `sh_type` member, and an *abbreviated* symbol table named `.dynsym` with `SHT_DYNSYM` as the section type, which contains only the set of symbols needed for dynamic linking. Each ELF file may have at most one of each and might not have a symbol table at

all. Many developers strip their executables of symbols so they don't take up unnecessary space.

In 64-bit ELF files, symbols have the following layout:

```
typedef struct {
    Elf64_Word st_name;     // Symbol name (string table offset)
    unsigned char st_info;  // Symbol's type (e.g., function, variable)
                            // and binding (e.g., local, global)
    unsigned char st_other; // Symbol's visibility
    Elf64_Half st_shndx;    // Which section the symbol is defined in
    Elf64_Addr st_value;    // Value of the symbol (e.g., address)
    Elf64_Xword st_size;    // Size of the symbol in bytes
} Elf64_Sym;
```

Consult the inline comments for descriptions of the members.

Add new private members to `sdb::elf` to hold the symbol table and parse it:

```
namespace sdb {
    class elf {
        --snip--
    private:
        --snip--
        void parse_symbol_table();

        --snip--
        std::vector<Elf64_Sym> symbol_table_;
    };
}
```

Next, implement `parse_symbol_table`. If the ELF file has a `.symtab` table, we should use it; otherwise, we should try to use the `.dynsym` table. Call this function from the `elf` constructor:

```
sdb::elf::elf(const std::filesystem::path& path) {
    --snip--
    parse_symbol_table();
}

void sdb::elf::parse_symbol_table() {
    auto opt_symtab = get_section(".symtab");
    if (!opt_symtab) {
        opt_symtab = get_section(".dynsym");
        if (!opt_symtab) return;
    }

    auto symtab = *opt_symtab;
    symbol_table_.resize(symtab->sh_size / symtab->sh_entsize);
    std::copy(data_ + symtab->sh_offset,
```

```
            data_ + symtab->sh_offset + symtab->sh_size,
            reinterpret_cast<std::byte*>(symbol_table_.data()));
    }
```

We first attempt to get the section header for the symbol table. If there isn't one, we just return, leaving `symbol_table_` empty. Otherwise, we calculate the number of entries in the symbol table by dividing the overall symbol table size by the size of a single entry, resize the `symbol_table_` vector accordingly, then copy the data over.

To parse sections, all we needed was the ability to look up a section header by name. For symbols, we must do a bit more than that. For one thing, multiple symbols may have the same name (for example, if they have different types or are *weak* symbols, which may be overridden). We also want the ability to find a symbol that lives at, or contains, a given file address. This will be useful for actions like finding out which function is currently being executed.

We'll construct two maps to help us perform these operations: one that maps names to multiple potential symbol table entries, and one that maps a single address range to a single symbol. For the former, we can use `std::unordered_multimap<std::string_view, Elf64_Sym*>`. For the latter, we'll use a `std::map<std::pair<file_addr, file_addr>, Elf64_Sym*>` with a special comparator that takes into account only the start of the address range; that way, we can easily look up symbols by their starting address without having to know their size. Add these private members to `sdb::elf` along with a private member to parse them and several public member functions that expose them:

```
#include <map>

namespace sdb {
    class elf {
    public:
        --snip--
        std::vector<const Elf64_Sym*> get_symbols_by_name(
            std::string_view name) const;

        std::optional<const Elf64_Sym*> get_symbol_at_address(
            file_addr addr) const;
        std::optional<const Elf64_Sym*> get_symbol_at_address(
            virt_addr addr) const;

        std::optional<const Elf64_Sym*> get_symbol_containing_address(
            file_addr addr) const;
        std::optional<const Elf64_Sym*> get_symbol_containing_address(
            virt_addr addr) const;

    private:
        --snip--
```

```
        void build_symbol_maps();

        --snip--
        std::unordered_multimap<std::string_view, Elf64_Sym*>
        symbol_name_map_;

        struct range_comparator {
            bool operator()(
                  std::pair<file_addr, file_addr> lhs,
                  std::pair<file_addr, file_addr> rhs) const {
                return lhs.first < rhs.first;
            }
        };
        std::map<std::pair<file_addr, file_addr>, Elf64_Sym*, range_comparator>
        symbol_addr_map_;
    };
}
```

We add functions for retrieving the set of symbols that correspond to the given name, the symbol that starts at a given file address, and the symbol that contains a given file address. For those final two, we declare overloads for both file addresses and virtual addresses for convenience.

The `range_comparator` type is the special comparator I mentioned earlier; it takes two pairs of addresses and compares only the first element of each pair (the low address). We supply this type as a template argument to `std::map`, which sets it as the type to use when comparing keys in the map.

Now let's write the functions, starting with `build_symbol_maps`. For each symbol, this function should make the relevant insertions into the name and address maps. Note that we should make entries in the address map only for symbols that actually have addresses, which excludes various types of symbols, such as file symbols and those that are undefined.

For the symbol name map, we'll add an entry for the symbol name as it is written in the ELF file as well as an entry for the *demangled* name, if any. Demangling addresses a consequence of the C++ language's support for function overloading and templates: that one function name may correspond to many different physical functions. To enable this, C++ symbol names are *mangled*, or transformed to also carry information about the namespace, return type, function arguments, template arguments, and more. For instance, `sdb::elf::build_symbol_maps` gets mangled to become the symbol `_ZN3sdb3elf17build_symbol_mapsEv`.

We won't concern ourselves with exactly how names are mangled; it suffices to say that the scheme is defined in the Itanium ABI for C++ and a function called `abi::__cxa_demangle` in the *<cxxabi.h>* header can demangle symbols for us. Read section 5.1 of the Itanium C++ ABI, "External Names (a.k.a. Mangling)" (*https://itanium-cxx-abi.github.io/cxx-abi/abi.html#mangling*), if you want to know all of the gory details. Implement `build_symbol_maps` in *sdb/src/elf.cpp*:

```
#include <cxxabi.h>

sdb::elf::elf(const std::filesystem::path& path) {
    --snip--
    build_symbol_maps();
}

void sdb::elf::build_symbol_maps() {
    for (auto& symbol : symbol_table_) {
        auto mangled_name = get_string(symbol.st_name);
        int demangle_status;
        auto demangled_name = abi::__cxa_demangle(
            mangled_name.data(), nullptr, nullptr, &demangle_status);
        if (demangle_status == 0) {
            symbol_name_map_.insert({ demangled_name, &symbol });
            free(demangled_name);
        }
        symbol_name_map_.insert({ mangled_name, &symbol });


        if (symbol.st_value != 0 and
            symbol.st_name != 0 and
            ELF64_ST_TYPE(symbol.st_info) != STT_TLS) {
            auto addr_range = std::pair(
                file_addr{ *this, symbol.st_value },
                file_addr{ *this, symbol.st_value + symbol.st_size });
            symbol_addr_map_.insert({ addr_range, &symbol });
        }
    }
}
```

We loop over every symbol in the symbol table, getting the (potentially mangled) name of the symbol from the string table and attempting to demangle it. The `__cxa_demangle` function takes a mangled name and a pointer to a buffer to fill with the mangled name; if this name is `nullptr`, the function dynamically allocates a string and returns it instead. We also pass the function an out-parameter for the demangled name size (if null, the size won't be stored) and an out-parameter for status reporting.

If demangling succeeds, `__cxa_demangle` stores `0` in the status argument, so we check `demangle_status`, store a new map entry if demangling succeeded, and free the allocated pointer. Regardless of the demangling's success, we add a mapping to the symbol name map for the symbol's (potentially mangled) name. If the symbol has an address and a name (meaning the `st_value` and `st_name` fields are not `0`) and doesn't point to thread-local storage (indicated by `ELF64_ST_TYPE(st_info)` being `STT_TLS`), we add an entry to the address map that maps the symbol's address range to a pointer to the symbol entry.

Now we'll turn our attention to `get_symbols_by_name`. We want this function to look up the given name in the symbol name map and return pointers to all symbols that match the name. While we could do some additional string processing so that, for example, `begin` finds the `std::vector<Elf64_Sym, std::allocator<Elf64_Sym>>::begin()` symbol, we won't include this feature here for simplicity. Implement the function in *sdb/src/elf.cpp*:

```
#include <algorithm>

std::vector<const Elf64_Sym*>
sdb::elf::get_symbols_by_name(std::string_view name) const {
    auto [begin, end] = symbol_name_map_.equal_range(name);

    std::vector<const Elf64_Sym*> ret;
    std::transform(begin, end, std::back_inserter(ret),
        [](auto& pair) { return pair.second; });
    return ret;
}
```

The interface for `std::multimap` is a bit odd if you're used to the one-to-one map types in the standard library. We retrieve a range of the map entries whose keys match the given name with `equal_range` and use *structured bindings* to immediately give names to the two elements of the pair that this function returns. (See the "Structured Bindings" box for more details.) We then declare a vector to hold the matching symbols and use `std::transform` to fill it. The multimap iterators provide the key and value, but we only need the values, so we copy the second element of each pair into the return vector.

STRUCTURED BINDINGS

The structured bindings feature of C++17 breaks, or *destructures*, variables into several pieces and gives names to those pieces. For example, we could destructure a `std::pair` to give names to the two elements of the pair, like this:

```
std::pair<int,int> my_pair {0,1};
auto [first, second] = my_pair;
```

Once this code runs, `first` will have the value `0` and `second` will have the value `1`.

Next, we'll implement the version of the `get_symbol_at_address` function that takes file addresses. Because our address map takes into account only the start address of the range, we can locate the relevant symbol, if it exists, by calling `std::map::find` with {`address, <some arbitrary address>`}:

```
std::optional<const Elf64_Sym*>
sdb::elf::get_symbol_at_address(file_addr address) const {
    if (address.elf_file() != this) return std::nullopt;
```

```
    file_addr null_addr;
    auto it = symbol_addr_map_.find({ address, null_addr });
    if (it == end(symbol_addr_map_)) return std::nullopt;

    return it->second;
}
```

If the file address points to the wrong ELF file, we return `std::nullopt`. We then create a null file address and call `find` with `{address, null_addr}` to find the element in the map whose start address matches the given one. If we don't find one, we return an empty optional. Otherwise, we return a pointer to the symbol.

The virtual address version of `get_symbol_at_address` function can defer to the one we just wrote:

```
std::optional<const Elf64_Sym*>
sdb::elf::get_symbol_at_address(virt_addr address) const {
    return get_symbol_at_address(address.to_file_addr(*this));
}
```

Before moving on, we must handle a couple of situations that might arise in the file address case:

- The symbol containing the address begins exactly at the address.
- The symbol containing the address begins earlier than the address and spans past it.

Helpfully, `std::map` comes with a `lower_bound` function that finds the first element that is either greater than or equal to our given key. With this, we can handle both of the necessary situations:

```
std::optional<const Elf64_Sym*>
sdb::elf::get_symbol_containing_address(file_addr address) const {
    if (address.elf_file() != this or symbol_addr_map_.empty())
        return std::nullopt;

    file_addr null_addr;
    auto it = symbol_addr_map_.lower_bound({ address, null_addr });

    if (it != end(symbol_addr_map_)) {
        if (auto [key, value] = *it; key.first == address) {
            return value;
        }
    }

    if (it == begin(symbol_addr_map_)) return std::nullopt;

    --it;
    if (auto [key, value] = *it;
```

```
        key.first < address and key.second > address) {
        return value;
    }

    return std::nullopt;
}
```

If the file address points to the wrong ELF file or the symbol table is empty, then we return `std::nullopt`. Otherwise, we find the first address entry with a starting address greater than or equal to the given one with `lower_bound`. So long as `lower_bound` didn't return the end iterator, we handle the situation in which the symbol containing the given offset begins exactly at that offset. We use structured bindings to give names to the key and value to which the iterator points rather than using `it->first.first`, and we check whether the first element of the address range matches the given address. If so, we return the pointer to the symbol.

If no entry matches the given address exactly, we'll look at the entry immediately preceding the lower bound we computed earlier, because the range could begin before the given address and span beyond it. If the current iterator is the begin iterator, there is no entry preceding it, so we return an empty optional. Otherwise, we decrement the iterator to get the preceding one and check whether the desired address falls within its range. If so, we've found the entry we're looking for, so we return the associated symbol. Otherwise, we've exhausted all possibilities, so we return an empty optional.

Finally, the virtual address version of `get_symbol_containing_address` can defer to the one we just wrote:

```
std::optional<const Elf64_Sym*>
sdb::elf::get_symbol_containing_address(virt_addr address) const {
    return get_symbol_containing_address(address.to_file_addr(*this));
}
```

Hooray! We have a working symbol table. (At least, we hope so; we'll test it once we've hooked the parser into the rest of the debugger.)

Creating a Target Type

Now that we've written an ELF parser, we need somewhere in our debugger state for object files to live. We could add to `sdb::process`, but this type is getting rather large as it is, and an object file isn't really contained in `process`; rather, it's higher-level, symbolic information relevant to the process.

In this section, we'll introduce a new type that represents this more symbolic layer of the program we're interacting with: `sdb::target`. While the `sdb::process` type has facilities for low-level operations such as reading and writing memory, managing breakpoint sites, and stepping over instructions, `sdb::target` will provide high-level operations such as managing

symbolic breakpoints, stepping through source code, and querying debug information.

Auxiliary Vectors

The `sdb::target` type will set the load address of the ELF file's `.text` section. To find the code's load address when constructing the `sdb::elf` object, we could parse */proc/<pid>/maps*, as we did in "Position-Independent Executables" on page 151. In this section, however, we'll find the load address in another way, as it lets us introduce a new concept: the *auxiliary vector*.

Defined in the SYSV ABI's processor supplement document, the auxiliary vector is an array of identifier/value pairs that the operating system kernel uses to provide information about a process to user space. It can encode information such as where the ELF program headers were loaded, the PID of the process, and, most importantly for us, the real entry point of the program. Armed with the real entry point, we can subtract the entry point offset stored in the ELF file header to get the load bias of the ELF file. In the *elf.h* header file, which defines the auxiliary vector IDs, we can see that the ID we want is `AT_ENTRY`.

We can get the auxiliary vector for a process in a couple of ways. When the process is started, the auxiliary vector is put in memory just above the program stack. Helpfully, Linux provides the same data in the */proc/<pid>/auxv* file. Let's add a function to `sdb::process` to get the auxiliary vector for the process. Add a declaration to *sdb/include/libsdb/process.hpp*:

```
#include <unordered_map>

namespace sdb {
    class process {
    public:
        --snip--
        std::unordered_map<int, std::uint64_t> get_auxv() const;
    };
}
```

This function maps the macro values (for example, `AT_ENTRY`) to the value of the corresponding entry in the auxiliary vector. Implement this function in *sdb/src/process.cpp*. It should keep reading entries from the aux vector until it finds the `AT_NULL` entry:

```
#include <elf.h>
#include <fstream>

std::unordered_map<int, std::uint64_t> sdb::process::get_auxv() const {
    auto path = "/proc/" + std::to_string(pid_) + "/auxv";
    std::ifstream auxv(path);

    std::unordered_map<int, std::uint64_t> ret;
```

```
    std::uint64_t id, value;

    auto read = [&](auto& into) {
        auxv.read(reinterpret_cast<char*>(&into), sizeof(into));
    };

    for (read(id); id != AT_NULL; read(id)) {
        read(value);
        ret[id] = value;
    }
    return ret;
}
```

We compute the path to the the auxiliary vector file and open an input stream for it. We declare variables to store the map to return and the ID and value of an entry. Entries in */proc/<pid>/auxv* are binary-encoded pairs of 64-bit integers. We make a small helper to read the data into a given variable and then use it to continue reading pairs of values, storing them in the map, until we read one whose ID is `AT_NULL` (which terminates the list). Finally, we return the map we generated.

Targets

Now we can implement the `sdb::target` type, which will manage the symbolic level of the program that we're debugging, such as storing the `sdb::elf` object for the program, reading debug information, and carrying out debugger operations at the level of the source code. Begin this type in a new *sdb/include/libsdb/target.hpp* file. It should have a process, an object file, and functions for retrieving these. Like `sdb::process`, it should be non-copyable and constructible only with `launch` and `attach` static members:

```
#ifndef SDB_TARGET_HPP
#define SDB_TARGET_HPP

#include <memory>
#include <libsdb/elf.hpp>
#include <libsdb/process.hpp>

namespace sdb {
    class target {
    public:
        target() = delete;
        target(const target&) = delete;
        target& operator=(const target&) = delete;

        static std::unique_ptr<target> launch(
            std::filesystem::path path,
            std::optional<int> stdout_replacement = std::nullopt);
```

```
        static std::unique_ptr<target> attach(pid_t pid);

        process& get_process() { return *process_; }
        const process& get_process() const { return *process_; }
        elf& get_elf() { return *elf_; }
        const elf& get_elf() const { return *elf_; }

    private:
        target(std::unique_ptr<process> proc, std::unique_ptr<elf> obj)
            : process_(std::move(proc)), elf_(std::move(obj))
        {}

        std::unique_ptr<process> process_;
        std::unique_ptr<elf> elf_;
    };
}

#endif
```

The `launch` and `attach` functions should both parse the ELF file for the relevant process and set the load address of the `.text` section. We'll factor this out into a `create_loaded_elf` function. Create a new *sdb/src/target.cpp* file with these contents:

```
#include <libsdb/target.hpp>
#include <libsdb/types.hpp>

namespace {
    std::unique_ptr<sdb::elf> create_loaded_elf(
        const sdb::process& proc, const std::filesystem::path& path) {
        auto auxv = proc.get_auxv();
        auto obj = std::make_unique<sdb::elf>(path);
        obj->notify_loaded(
            sdb::virt_addr(auxv[AT_ENTRY] - obj->get_header().e_entry));
        return obj;
    }
}
```

We'll use this `create_loaded_elf` function inside of both `launch` and `attach`. This function gets the auxiliary vector for the process, creates an `sdb::elf` object for the file at the given path, then sets the load address of its `.text` section by subtracting the load address of the entry point in the ELF header from the actual load address of the entry point. Add this new file to *sdb/src/CMakeLists.txt*:

```
add_library(libsdb ... target.cpp)
```

Implement `sdb::target::launch` and `attach` in the same file. The `launch` function is the easiest to implement; it merely constructs the `process_` and `elf_` members using the given path:

```
std::unique_ptr<sdb::target>
sdb::target::launch(
      std::filesystem::path path, std::optional<int> stdout_replacement) {
    auto proc = process::launch(path, true, stdout_replacement);
    auto obj = create_loaded_elf(*proc, path);
    return std::unique_ptr<target>(
        new target(std::move(proc), std::move(obj)));
}
```

The `attach` function is a bit trickier. We need a way to find the ELF file associated with a given PID. Here, the procfs comes to the rescue once more! The */proc/<pid>/exe* file is a symbolic link to the executable for the given process. We can pass this path to `create_loaded_elf`:

```
std::unique_ptr<sdb::target>
sdb::target::attach(pid_t pid) {
    auto elf_path = std::filesystem::path("/proc") / std::to_string(pid) / "exe";
    auto proc = process::attach(pid);
    auto obj = create_loaded_elf(*proc, elf_path);
    return std::unique_ptr<target>(
        new target(std::move(proc), std::move(obj)));
}
```

We compute the path to the executable by concatenating `/proc`, the PID, and `exe`. Notice that `std::filesystem::path` supports the `/` operator for path concatenation. We then create the `sdb::elf` file with the computed path and return a new `sdb::target` object.

We'll add more features to `sdb::target` in later chapters, but for now, let's refactor the command line driver to create an `sdb::target` instead of an `sdb::process`, then use the symbolic information from the ELF file to work out which function is currently being executed. In *sdb/tools/sdb.cpp*, `attach` should create the target:

```
#include <libsdb/target.hpp>

namespace {
    std::unique_ptr<sdb::target> attach(int argc, const char** argv) {
        // Passing PID
        if (argc == 3 && argv[1] == std::string_view("-p")) {
            pid_t pid = std::atoi(argv[2]);
            return sdb::target::attach(pid);
        }
```

```
        // Passing program name
        else {
            const char* program_path = argv[1];
            auto target = sdb::target::launch(program_path);
            fmt::print("Launched process with PID {}\n", target->get_process().pid());
            return target;
        }
    }
}
```

We update the return type to be `std::unique_ptr<sdb::target>` and update the calls to `attach` and `launch` to call the versions in `sdb::target` rather than `sdb::process`. We also update the call to `pid` to first get the `process` object from the target and the last `return` statement to return the target.

The `main` function needs updating as well, to pass the target to `main_loop` and set up the `SIGINT` handler:

```
int main(int argc, const char* argv[]) {
    if (argc == 1) {
        std::cerr << "No arguments given\n";
        return -1;
    }

    try {
        auto target = attach(argc, argv);
        g_sdb_process = &target->get_process();
        signal(SIGINT, handle_sigint);
        main_loop(target);
    }
    catch (const sdb::error& err) {
        std::cout << err.what() << '\n';
    }
}
```

The `attach` function now returns an `sdb::target`, so change the name of the `process` variable to `target`. Store `&target->get_process()` in the `g_sdb_process` global variable, and pass `target` to the `main_loop` function.

The `main_loop` function should pass the target to `handle_command`:

```
namespace {
    void main_loop(std::unique_ptr<sdb::target>& target) {
        --snip--
        handle_command(target, line_str);
        --snip--
    }
}
```

For now, `handle_command` will just pass the given target's process to most of the subcommands. The only one that we'll give the target itself is `handle_stop`, so that we can try to print the current function. Grab a reference to the target's process near the top of the function, and then update all of the calls:

```
namespace {
❶ void handle_command(std::unique_ptr<sdb::target>& target,
          std::string_view line) {
        auto args = split(line, ' ');
        auto command = args[0];
    ❷ auto process = &target->get_process();

        if (is_prefix(command, "continue")) {
            process->resume();
            auto reason = process->wait_on_signal();
         ❸ handle_stop(*target, reason);
        }
        --snip--
        else if (is_prefix(command, "step")) {
            auto reason = process->step_instruction();
         ❹ handle_stop(*target, reason);
        }
        --snip--
    }
}
```

The updates you need to make are in the function signature ❶, the creation of a local `process` variable ❷, and both calls to `handle_stop` ❸ ❹. In `handle_stop`, pass the target to `print_stop_reason`:

```
namespace {
    void handle_stop(sdb::target& target, sdb::stop_reason reason) {
        print_stop_reason(target, reason);
        if (reason.reason == sdb::process_state::stopped) {
            print_disassembly(
                target.get_process(), target.get_process().get_pc(), 5);
        }
    }
}
```

Update the parameter type and the arguments to `print_stop_reason` and `print_disassembly`.

Now we can make a nonstructural change. In `handle_stop`, if the process stopped due to a signal, we should look up the symbol corresponding to the current program counter and print out the function name, if there is one. That part of the `switch` statement is getting rather crowded, so let's factor it into a `get_signal_stop_reason` function:

```
namespace {
    std::string get_signal_stop_reason(
          const sdb::target& target, sdb::stop_reason reason) {
        auto& process = target.get_process();
        std::string message = fmt::format("stopped with signal {} at {:#x}",
            sigabbrev_np(reason.info), process.get_pc().addr());

        auto func = target.get_elf().get_symbol_containing_address(process.get_pc());
        if (func and ELF64_ST_TYPE(func.value()->st_info) == STT_FUNC) {
            message += fmt::format(" ({})",
                target.get_elf().get_string(func.value()->st_name));
        }

        if (reason.info == SIGTRAP) {
            message += get_sigtrap_info(process, reason);
        }

        return message;
    }
}
```

At the start of the function, we grab a reference to the process so we don't have to repeat `target.get_process()` in multiple places. We initialize the `message` variable and then retrieve the symbol at the current program counter with `get_symbol_containing_address`. This function returns a pointer to the symbol corresponding to the current function. Getting the type of a symbol is a little unwieldy, as it's packed into the `st_info` member, but we use the `ELF64_ST_TYPE` macro to retrieve the relevant bits and test it against `STT_FUNC` to see if the symbol represents a function. If so, we add the function's name to the message, retrieving the name from the string table.

We now need to modify `print_stop_reason` to call this new function:

```
namespace {
    void print_stop_reason(
          const sdb::target& target, sdb::stop_reason reason) {
        std::string message;
        switch (reason.reason)
        {
        --snip--
        case sdb::process_state::stopped:
            message = get_signal_stop_reason(target, reason);
            break;
        }

        fmt::print("Process {} {}\n", target.get_process().pid(), message);
    }
}
```

First, we update the parameter type from `const sdb::process&` to `const sdb::target&`. We then update the `sdb::process_state::stopped` branch to populate the message with the `get_signal_stop_reason` function we just wrote. Finally, we update the printing of the PID to first extract the process from the target.

Now, if we stop at a breakpoint inside of a function for which we have a symbol range, the debugger will print the function's name when the inferior stops:

```
$ tools/sdb test/targets/hello_sdb
Launched process with PID 1647
sdb> break set 0x55555555515b
sdb> c
Process 1647 stopped with signal TRAP at 0x55555555515b (main) (breakpoint 1)
0x000055555555515b: call 0x0000555555555050
0x0000555555555160: mov $0x00, %eax
0x0000555555555165: pop %rbp
0x0000555555555166: ret
0x0000555555555167: add %dh, %bl
```

The debugger correctly adds `(main)` to the output! Now we can test the debugger's new features.

Testing

To test our new type, we won't account for all possible scenarios; rather, we'll ensure that if we regress some core functionality, we'll know about it immediately.

We'll start by ensuring that the symbol at the entry point offset is the `_start` function. Supplied by the C++ toolchain, this function sets up the support that the program needs to run before calling `main`:

```
#include <libsdb/target.hpp>

TEST_CASE("ELF parser works", "[elf]") {
    auto path = "targets/hello_sdb";
    sdb::elf elf (path);
    auto entry = elf.get_header().e_entry;
    auto sym = elf.get_symbol_at_address(file_addr{ elf, entry });
    auto name = elf.get_string(sym.value()->st_name);
    REQUIRE(name == "_start");
```

We can also look for the `_start` symbol using `get_symbols_by_name`, to ensure we get the right one:

```
    --snip--
    auto syms = elf.get_symbols_by_name("_start");
    name = elf.get_string(syms.at(0)->st_name);
    REQUIRE(name == "_start");
```

Finally, we'll pretend that the ELF file is loaded at `0xcafecafe` and ensure that `0xcafecafe + entry` finds the `_start` symbol if we look up the symbol by address rather than offset:

```
TEST_CASE("ELF parser works", "[elf]") {
    --snip--
    elf.notify_loaded(virt_addr{ 0xcafecafe });
    sym = elf.get_symbol_at_address(virt_addr{ 0xcafecafe + entry });
    name = elf.get_string(sym.value()->st_name);
    REQUIRE(name == "_start");
}
```

These tests should all pass. That will do for testing!

Summary

In this chapter, you learned about the layout of ELF files, which store the code of a program along with necessary data and metadata for linking, loading, and executing. You also implemented a parser for ELF files and tested it.

In the next chapter, you'll begin implementing a parser for the debug information format used on Linux, DWARF. DWARF is considerably more complex than ELF, so we'll dedicate multiple chapters to implementing all of the parser's necessary pieces to facilitate a full debugging experience.

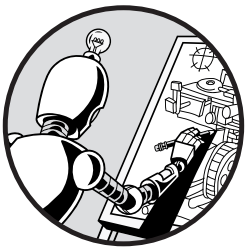
Check Your Knowledge

1. What are the four kinds of object files that ELF files can represent?
2. What is the difference between an ELF section and an ELF segment?
3. Which Linux syscall can map large files into memory?
4. How are ELF string tables represented?
5. What is the auxiliary vector?

12

DEBUG INFORMATION

The mountain smoked beneath the moon;
The dwarves, they heard the tramp of doom.
They fled their hall to dying fall
Beneath his feet, beneath the moon.
–J.R.R Tolkien, *The Hobbit*



Now that you've mastered the ELF format, it's time to turn your attention to the land of the DWARFs. *DWARF* is the main debug information format used on Linux. It relates a binary back to the source code that produced it, allowing us to match machine instructions to lines of source code, locate source functions and variables in the running process, describe the types that exist in the program, and more.

Debuggers must be able to interpret DWARF so they can carry out operations that require understanding the relationship between source code and machine code, such as setting breakpoints on source files and single-stepping through lines of source code. In this chapter, you'll learn about the information encoded in the DWARF format and how you can decode it.

Then, you'll write a parser that you'll use in future chapters to power your debugger.

An Introduction to DWARF

At the time of writing, DWARF information comes in two different formats: 32-bit and 64-bit. You might assume that a debugger for a 64-bit system would need to use 64-bit DWARF, but 32-bit DWARF is perfectly capable of representing debug info for 64-bit systems, and most compilers produce 32-bit DWARF by default, as it has a more compact representation. So, I'll focus on 32-bit DWARF in this book.

The different versions of the DWARF standard aren't backward compatible, meaning a parser for DWARF 5 can't parse DWARF 4, 3, 2, or 1. This lets the DWARF standard evolve without taking on representational overhead but is a trade-off that requires consumers to change their behavior (sometimes quite substantially) depending on the version of DWARF with which a given file is encoded. This book focuses on DWARF 4, which is recent enough to teach modern techniques but old enough that the compiler installed on your system should already support it.

If you'd like to support 64-bit DWARF and other DWARF versions, take a look at the DWARF specification at *https://dwarfstd.org*. Sections 1.4 through 1.7 and section 7.4 of the DWARF 5 standard list the differences between the formats. It's very possible to write a DWARF consumer that supports multiple formats, but doing so makes parsing and data representation more complex. This chapter focuses instead on how debug information is encoded.

DWARF Sections

By design, DWARF works well with ELF files, but nothing stops a compiler from encoding DWARF information in a different object file format. The object file need only provide a way to store data in a set of sections with predefined names and cross-reference between them. DWARF 4 specifies the following set of sections:

`.debug_info` The core DWARF data containing DWARF information entries (DIEs)

`.debug_abbrev` Abbreviations used to simplify the `.debug_info` section

`.debug_aranges` A lookup table to speed up locating debug information relevant to an address range

`.debug_frame` Information about how to unwind the stack; usually empty, as the SYSV ABI specifies the use of the `.eh_frame` section instead

`.debug_line` Mappings from machine instructions to source-level instructions

`.debug_loc` Descriptions of where variables live

`.debug_macinfo` Macro descriptions

`.debug_ranges` Address ranges referenced by DIEs

`.debug_str` A string table used by `.debug_info`; uses the same format as ELF string tables

`.debug_types` Type descriptions; may instead live in the `.debug_info` section

`.debug_pubnames` A lookup table for global objects and functions; legacy, essentially unused

`.debug_pubtypes` A lookup table for global types; legacy, essentially unused

This chapter focuses on the `.debug_info`, `.debug_abbrev`, and `.debug_ranges` sections. Parsing `.debug_info` will enable us to carry out operations such as setting a breakpoint on a function name and looking up which function we're currently in. The symbol tables we parsed in the previous chapter could get us most of the way there for these operations, but they wouldn't support important tasks like function inlining. Some elements of the `.debug_info` section rely on information stored in the `.debug_abbrev` or `.debug_ranges` section, so we'll parse those as well to get the necessary data.

Let's begin with an explanation of the `.debug_info` section, as it's the main repository of knowledge about your program's source-level entities and how they map to the machine code. For example, if you'd like to know what source files were used to compile an executable, where the variable `had_existential_crisis` is stored, or where to find the code for the `pet_cat` function, you should check the `.debug_info` section.

This section divides information into categories for each compile unit involved in the program. In C++, a *compile unit* is an implementation file (usually ending in the *.cpp* suffix) along with the header files included in that implementation file. For each compile unit, `.debug_info` represents the relevant debugging information for all source-level entities as a graph of *DWARF information entries (DIEs)*. Each DIE represents a single program entity, such as a namespace, a variable, a function, or a parameter. You can see a textual representation of their format using the `dwarfdump` command. Here's an example:

```
$ dwarfdump test/targets/hello_sdb
< 0><0x0000000b>  DW_TAG_compile_unit
                    DW_AT_producer         GNU C++17 11.3.0 -mtune=generic -march=x86-64 -g
                                           -g -gdwarf-4 -O0 -fasynchronous-unwind-tables
                                           -fstack-protector-strong -fstack-clash-protection
                                           -fcf-protection
                    DW_AT_language         DW_LANG_C_plus_plus
                    DW_AT_name             ../../../test/targets/hello_sdb.cpp
                    DW_AT_comp_dir         /home/tartanllama/.vs/sdb_new/out/build/linux-debug
                    DW_AT_low_pc           0x00001149
                    DW_AT_high_pc          <offset-from-lowpc> 30 <highpc: 0x00001167>
                    DW_AT_stmt_list        0x00000000
< 1><0x0000002d>    DW_TAG_namespace
```

```
                    DW_AT_name              std
                    DW_AT_decl_file         0x00000003 /usr/include/x86_64-linux-gnu/c++/
                                                       11/bits/c++config.h
                    DW_AT_decl_line         0x00000116
                    DW_AT_decl_column       0x0000000b
                    DW_AT_sibling           <0x00000125>
< 2><0x0000003a>    DW_TAG_namespace
                      DW_AT_name              __cxx11
                      DW_AT_decl_file         0x00000003 /usr/include/x86_64-linux-gnu/c++/
                                                         11/bits/c++config.h
                      DW_AT_decl_line         0x0000012e
                      DW_AT_decl_column       0x00000041
                      DW_AT_export_symbols    yes(1)
< 2><0x00000043>    DW_TAG_imported_module
                      DW_AT_decl_file         0x00000003 /usr/include/x86_64-linux-gnu/c++/
                                                         11/bits/c++config.h
                      DW_AT_decl_line         0x0000012e
                      DW_AT_decl_column       0x00000041
                      DW_AT_import            <0x0000003a>
```

DWARF organizes DIEs in a tree structure; the more deeply nested DIEs are logical children of the preceding DIE with a shallower nesting. You can identify the start of each DIE by the numbers in angle brackets in the left-most column. The first number is the nesting level, and the second is the DIE's byte offset from the start of the `.debug_info` section. Each DIE has a *tag*, such as `DW_TAG_compile_unit` or `DW_TAG_namespace`, that identifies the type of source program entity that the DIE represents. The DIE stores its information as a list of attributes. An attribute's type specifies the kind of information being expressed, such as `DW_AT_name` or `DW_AT_language`, and is followed by a value.

Using this information, we can understand details about the compiled program. For example, from the compile unit DIE and its `DW_AT_name` attribute, we can see that the compilation involved a file that lives at *../../../test/targets/hello_sdb.cpp*. Further, the `DW_AT_language` attribute tells us that this file was written in C++.

In the nested DIEs, we learn that the program has a nested namespace called `std::__cxx11` containing a `DW_TAG_imported_module` entity (usually a `using namespace` statement) that imports the module represented by the DIE at an offset of `0x0000003a`. Curiously, these last two DIEs refer to the exact same source location, as the `DW_AT_decl_file`, `DW_AT_decl_line`, and `DW_AT_decl_column` attributes are all the same. This suggests that the programmer used the C++11 *inline namespace* feature to declare a namespace and make its contents visible to the containing namespace. If we look at the file specified in the DIE, we can see that our deduction is correct:

```
namespace std
{
    inline namespace __cxx11 __attribute__((__abi_tag__ ("cxx11"))) { }
}
```

Consider a few more details about this textual representation: the DW_AT_low_pc, DW_AT_high_pc, and DW_AT_sibling attributes. The DW_AT_low_pc and DW_AT_high_pc attributes indicate the range of addresses containing the code for the compilation unit, while DW_AT_sibling provides a DWARF parser with a quick way to locate the sibling of a given DIE without having to parse its children. In this case, the sibling is the DIE at offset 0x00000125.

You should now understand how you might use the contents of the .debug_info section to reconstruct information about the source program. Next, we'll consider how the compiler encodes DIEs.

DIE Binary Encoding

If you were designing a binary encoding system for the textual representation you just saw, you might opt to store an integer representing the tag, followed by a list of pairs: an integer representing the attribute type and the attribute value. The actual DIE encoding is more complicated than that, for a few good reasons.

The first reason is that the compiler can encode the same attribute type in different ways. For example, it can encode a DW_AT_high_pc type as a constant address or as an offset from the DW_AT_low_pc attribute of the same DIE. Furthermore, it could encode that offset as a 1-, 2-, 4-, 8-, or multibyte integer. If an address range is very small, the compiler could choose to use 8 bytes for the low PC, then a single byte for the high PC. Doing so shaves off just a few bytes, but if thousands of DIEs take advantage of this optimization, you've suddenly gained a few thousand bytes of savings.

Because the compiler can encode attributes in multiple ways, each attribute value also has an associated *form* that specifies its encoding. You can see the form associated with each attribute value by passing dwarfdump the -M flag in addition to the usual -a flag. Here is a subset of the examples from earlier with form annotations (and the leftmost column removed to save space):

```
DW_TAG_namespace
  DW_AT_name                 std <form DW_FORM_string> ❶
  DW_AT_decl_file            0x00000003 /usr/include/x86_64-linux-gnu/c++/
                                        11/bits/c++config.h <form DW_FORM_data1>
  DW_AT_decl_line            0x00000116 <form DW_FORM_data2>
  DW_AT_decl_column          0x0000000b <form DW_FORM_data1>
  DW_AT_sibling              <0x00000125> <form DW_FORM_ref4>
DW_TAG_namespace
  DW_AT_name                 __cxx11 <form DW_FORM_strp> ❷
  DW_AT_decl_file            0x00000003 /usr/include/x86_64-linux-gnu/c++/
                                        11/bits/c++config.h <form DW_FORM_data1>
  DW_AT_decl_line            0x0000012e <form DW_FORM_data2>
  DW_AT_decl_column          0x00000041 <form DW_FORM_data1>
  DW_AT_export_symbols       yes(1) <form DW_FORM_flag_present>
```

Most of the integers in this example are small, so the compiler has chosen to use 1- and 2-byte integers to represent them (`DW_FORM_data1` and `DW_FORM_data2`). Interestingly, it has chosen to represent the string of `std` directly in the DIE (`DW_FORM_string`) ❶ but encoded `__cxx11` as a string table reference (`DW_FORM_strp`) ❷. This is because `std`, when encoded as a null-terminated string, takes up 4 bytes (one for each character, plus the null terminator), whereas `__cxx11` would take up 8. The compiler has decided to use a 4-byte string table offset for `__cxx11` to save 4 bytes of storage in the `.debug_info` section, at the cost of requiring consumers to have to reference the string table. DWARF is full of trade-offs like these.

So, instead of merely storing each attribute type and attribute value as a pair, we must add an integer to represent the form, right? Well, that would be one approach, but that's not what DWARF does. To understand why, take a look at this example:

```
DW_TAG_imported_declaration
  DW_AT_decl_file             0x00000002 /usr/include/c++/11/cstdio <form DW_FORM_data1>
  DW_AT_decl_line             0x00000062 <form DW_FORM_data1>
  DW_AT_decl_column           0x0000000b <form DW_FORM_data1>
  DW_AT_import                <0x000003d6> <form DW_FORM_ref4>
DW_TAG_imported_declaration
  DW_AT_decl_file             0x00000002 /usr/include/c++/11/cstdio <form DW_FORM_data1>
  DW_AT_decl_line             0x00000063 <form DW_FORM_data1>
  DW_AT_decl_column           0x0000000b <form DW_FORM_data1>
  DW_AT_import                <0x0000043d> <form DW_FORM_ref4>
DW_TAG_imported_declaration
  DW_AT_decl_file             0x00000002 /usr/include/c++/11/cstdio <form DW_FORM_data1>
  DW_AT_decl_line             0x00000065 <form DW_FORM_data1>
  DW_AT_decl_column           0x0000000b <form DW_FORM_data1>
  DW_AT_import                <0x00000477> <form DW_FORM_ref4>
```

Notice that these three DIEs all have the same tag, attribute types, and forms; only their attribute values differ. This is a very common pattern, and in a large `.debug_info` section it would lead to a huge amount of unnecessary repetition. A ridiculous quantity of unnecessary repetition. Boatloads more repetition than is necessary.

To address this issue, DWARF uses *abbreviation tables*. Each entry of an abbreviation table contains a tag, a bit that encodes whether the DIE has children, a list of attribute types, and the form used to encode each attribute. The DIEs then store only an index into the abbreviation table and the attribute values for the DIE. DWARF places abbreviation tables in the `.debug_abbrev` section, and we'll see their encodings in more detail very soon.

Now that you understand DWARF at a high level, we can begin writing the parser by solving the issue of integer constants.

Fetching a Constants File

Because DWARF has a myriad of DIE tags, attribute types, and forms, it needs a compact way to represent these elements in binary. The obvious solution is to assign each one an integral value using enums. Unfortunately, DWARF requires hundreds of these constants, and if I asked you to enter hundreds of integral constants into your text editor without error, you'd be quite justified in closing this book forever.

Unlike for ELF, the operating system doesn't provide a handy *dwarf.h* file. Fortunately, I've written one for you. Fetch the file at *https://github.com/TartanLlama/sdb/blob/chapter-12/include/libsdb/detail/dwarf.h* and place it at the matching location in your codebase.

You can now begin building the base of the debugger's DWARF parser. Create a new file at *sdb/include/libsdb/dwarf.hpp* with these contents:

```
#ifndef SDB_DWARF_HPP
#define SDB_DWARF_HPP

❶ #include <libsdb/detail/dwarf.h>

namespace sdb {
 ❷ class elf;
    class dwarf {
    public:
        dwarf(const elf& parent);
        const elf* elf_file() const { return elf_; }

    private:
        const elf* elf_;
    };
}

#endif
```

Include the *dwarf.h* file you just prepared ❶. You'll need references or pointers to the parent `sdb::elf` object only, so instead of including the *libsdb/elf.hpp* header, forward-declare `sdb::elf` to avoid giving more work to the compiler ❷. Finally, create a simple `dwarf` type that stores a pointer to its object file and provides a way to retrieve it. The constructor must do a bit of work, so we'll define it later.

Now that we've performed this setup, we can move on to parsing. Because the parsing of DIEs relies on the abbreviation tables, we'll start with the `.debug_abbrev` section.

Parsing Abbreviation Tables

The `.debug_abbrev` section contains several abbreviation tables. Each compile unit in the `.debug_info` section uses exactly one abbreviation table, but different compile units may share the same table.

DWARF identifies each abbreviation table by its byte offset from the start of the `.debug_abbrev` section and gives each table entry an *abbreviation code*: an integer that starts at 1 and increases for each subsequent entry in the abbreviation table. As such, we can store the `.debug_abbrev` section as a nested map, represented by the following type:

```
std::unordered_map<std::size_t,
                   std::unordered_map<std::uint64_t, abbrev>>
```

This type maps offsets to another map, of integers to abbreviation entries. We'll discuss the contents of the `abbrev` type soon. While other types might store this data in a more cache-friendly manner, we use `std::unordered_map` for simplicity.

The `.debug_abbrev` section can be large, so we shouldn't parse it in its entirety when we create the `sdb::dwarf` object. Instead, we'll parse each abbreviation table when it's first requested and then cache it for future lookups. Declare the interface for retrieving an abbreviation table in *sdb/include/libsdb/dwarf.hpp*:

```
#include <unordered_map>

namespace sdb {
    class dwarf {
    public:
        --snip--
        const std::unordered_map<std::uint64_t, abbrev>& get_abbrev_table(
            std::size_t offset);
    private:
        --snip--
        std::unordered_map<std::size_t,
            std::unordered_map<std::uint64_t, abbrev>> abbrev_tables_;
    };
}
```

We add a new member called `abbrev_tables_` that will store the parsed tables, then add a member function that will parse the table or return one that has already been parsed. The `get_abbrev_table` function will offload most of the work to a helper `parse_abbrev_table` function. Create a new *sdb/src/dwarf.cpp* file and implement `get_abbrev_table`:

```
#include <libsdb/dwarf.hpp>

const std::unordered_map<std::uint64_t, sdb::abbrev>&
sdb::dwarf::get_abbrev_table(std::size_t offset) {
```

```
    if (!abbrev_tables_.count(offset)) {
        abbrev_tables_.emplace(offset, parse_abbrev_table(*elf_, offset));
    }
    return abbrev_tables_.at(offset);
}
```

If we haven't yet parsed the abbreviation table at the requested offset, we do so using the currently nonexistent `parse_abbrev_table` function and store the result in `abbrev_tables_`. If we've already called `get_abbrev_table` with this offset, we return the previously parsed table. Add the *dwarf.cpp* file to *sdb/src/CMakeLists.txt*:

```
add_library(libsdb ... dwarf.cpp)
```

Now, before you can understand exactly what abbreviation entries store and how they encode this data, you first need to understand a somewhat strange encoding scheme that DWARF uses for some integers: Little Endian Base 128 (`LEB128`).

Integer Encoding

Because DWARF takes memory consumption very seriously, it aims to encode the debug information as efficiently as is reasonable. One common source of difficulty is that most of the integers it must store are small, but some are larger.

Take, for example, DWARF DIE tags such as `DW_TAG_compile_unit`. The DWARF standard specifies `0x43` as the highest value for encoding tags, but it reserves the numbers `0x4080` to `0xffff` for compiler implementers to use for their own purposes. So, tags usually fit in a single byte, but once in a while, they might need 2 bytes. Does this mean we should always encode tags with 2 bytes? Not according to DWARF. Instead, the format uses `LEB128`, a variable-length encoding scheme for integers that attempts to minimize the storage required for small integers while allowing the encoding of larger integers.

For unsigned integers (`ULEB128`), the scheme splits the bit representation into groups of 7 bits and adds a 1 bit to the front of each group, except for the most significant group, to which it adds a 0 bit. As such, an `LEB128`-encoded integer requires an overhead of 1 bit for every 7 bits required to encode the integer normally. As an example, let's encode the integer 624,485 in `LEB128`. Here are the steps:

1. Encode the integer in binary: 10011000011101100101
2. Pad the value to a multiple of 7 bits: 010011000011101100101
3. Split the value into groups of seven: 0100110 0001110 1100101
4. Prefix the groups with 0 or 1: 00100110 10001110 11100101

Encoding signed integers (`SLEB128`) is a bit more complicated and requires familiarity with two's complement encoding, discussed in Chapter 5.

The scheme begins by encoding the integer using two's complement at a bit width that is exactly divisible by seven. Then, it follows the same splitting and prefixing procedure as for unsigned integers. For example, here is how we'd encode the number −123,456:

1. Encode 123,456 in binary: 11110001001000000
2. Pad the value to a multiple of 7 bits: 000011110001001000000
3. Flip all bits: 11110000111011011111
4. Add 1 to the value: 11110000111011000000
5. Split the value into groups of seven: 1111000 0111011 1000000
6. Prefix the groups with 0 or 1: 01111000 10111011 11000000

You'll find `LEB128` used across the DWARF standard, so it pays to spend some time getting used to it.

Entry Structure

Each abbreviation entry starts with the abbreviation code used to reference the table, encoded as a `ULEB128` integer. Directly following the abbreviation code is another `ULEB128` that encodes the entry's tag. You can find the mappings between tags and integer values in the DWARF standard and in the *dwarf.h* file you prepared earlier. (For example, `DW_TAG_compile_unit` maps to `0x11`.) After the tag comes a single bit that states whether the DIE has child DIEs. This bit facilitates the representation of the DIE tree structure.

Last are the attribute specifications. Each specification is a `ULEB128` that encodes the attribute type, followed by a `ULEB128` for the form. Again, you can find these mappings in *dwarf.h*. The list of attribute specifications terminates with an entry whose type and form are both `0`.

Attributes use the form `DW_FORM_indirect` in cases where many DIEs have mostly the same sets of attributes and forms but have one or two that differ. For example, say most of the attribute values fit in a single-byte integer, but some require 2 bytes. The `DW_FORM_indirect` form lets the compiler keep a single abbreviation entry for these DIEs and requires the DIE itself to supply the form for any outlier attribute as a `ULEB128`-encoded prefix.

The DWARF Cursor

We'll begin the parser by writing a type to help us parse forms from various locations. This `cursor` type will point to a location in the DWARF information, allowing us to easily access information from that location. As a result, if we wanted to parse a `ULEB128`, a 64-bit integer, and then a string, we could do something like the following:

```
auto the_uleb = cur.uleb128();
auto the_int = cur.u64();
auto the_string = cur.string();
```

The cursor will handle the parsing of the data and advance the location being pointed to. Only the DWARF parser's code will use this `cursor` type, so we'll implement it in *sdb/src/dwarf.cpp*. Begin its implementation like this:

```
#include <libsdb/types.hpp>

namespace {
    class cursor {
    public:
     ❶ explicit cursor(sdb::span<const std::byte> data)
            : data_(data), pos_(data.begin()) {}

     ❷ cursor& operator++() { ++pos_; return *this; }
        cursor& operator+=(std::size_t size) { pos_ += size; return *this; }

     ❸ const std::byte* position() const { return pos_; }

     ❹ bool finished() const {
            return pos_ >= data_.end();
        }

    private:
        sdb::span<const std::byte> data_;
        const std::byte* pos_;
    };
}
```

The cursor stores an `sdb::span` representing the data range it's looking at and a pointer to a `std::byte` representing the cursor's current position. We initialize these values in the constructor ❶, setting the initial position to the start of the span.

We'll want to be able to manually walk the cursor forward to skip over bytes, so we add some helpers to achieve this ❷. The `operator++` and `operator+=` helpers will enable us to write code such as `++cur` and `cur += n` to move the cursor forward. We also add a way to get the cursor's current position ❸ and a way to check whether the cursor has read all of the data (meaning the current position is at or past the end of the span) ❹.

Now we can add some member functions to perform the parsing. Let's start with the parsing of fixed-width integers:

```
#include <libsdb/bit.hpp>

namespace {
    class cursor {
    public:
        --snip--
        template <class T>
```

```
        T fixed_int() {
            auto t = sdb::from_bytes<T>(pos_);
            pos_ += sizeof(T);
            return t;
        }
        --snip--
    };
}
```

We write a function template that can parse any fixed-width integer. We use `sdb::from_bytes` to parse the data, advance the current position by the byte size of the integer we're parsing, and then return the result.

Let's add a bunch of convenience functions that call this function template for specific types:

```
namespace {
    class cursor {
    public:
        --snip--
        std::uint8_t u8() { return fixed_int<std::uint8_t>(); }
        std::uint16_t u16() { return fixed_int<std::uint16_t>(); }
        std::uint32_t u32() { return fixed_int<std::uint32_t>(); }
        std::uint64_t u64() { return fixed_int<std::uint64_t>(); }
        std::int8_t s8() { return fixed_int<std::int8_t>(); }
        std::int16_t s16() { return fixed_int<std::int16_t>(); }
        std::int32_t s32() { return fixed_int<std::int32_t>(); }
        std::int64_t s64() { return fixed_int<std::int64_t>(); }
        --snip--
    };
}
```

This will keep the usage of the cursor type concise.

Only slightly more complex is the parsing of strings:

```
#include <string_view>
#include <algorithm>

namespace {
    class cursor {
    public:
        --snip--
        std::string_view string() {
            auto null_terminator = std::find(pos_, data_.end(), std::byte{0});
            std::string_view ret(reinterpret_cast<const char*>(pos_),
                                 null_terminator - pos_);
            pos_ = null_terminator + 1;
            return ret;
        }
        --snip--
```

```
    };
}
```

We find the first 0 byte (the null terminator) between the current position and the end of the data. We then construct a `std::string_view` that starts at the current position and whose size is the distance to the null terminator. We set the new cursor position to one byte past the null terminator, then return the result.

Now we get to the harder types, starting with `ULEB128`. Recall that encoding these values requires splitting the integer into groups of 7 bits, then adding a bit to the front of each group. To reverse this process, we read the value one byte at a time, remove the first bit of the byte, then shift the remaining 7 bits into their correct place. We stop when we get to a byte that starts with a 0. In C++, we can perform these steps like so:

```
namespace {
    class cursor {
    public:
        --snip--
        std::uint64_t uleb128() {
            std::uint64_t res = 0;
            int shift = 0;
            std::uint8_t byte = 0;
            do {
                byte = u8();
                auto masked = static_cast<uint64_t>(byte & 0x7f);
                res |= masked << shift;
                shift += 7;
            } while ((byte & 0x80) != 0);
            return res;
        }
        --snip--
    };
}
```

The `res` variable holds the result we're computing, `shift` holds the amount by which to shift the next byte to the left, and `byte` holds the current byte we're looking at. We loop, reading a byte, masking off the first bit, and then shifting the masked byte into the right position and bitwise ORing it with `res` to set the bits. Before continuing the loop, we add 7 to `shift` to shift the next 7 bits into position. We terminate the loop when the topmost bit of the read byte is 0.

The algorithm for `SLEB128` integers is similar, but if we haven't filled the entire result integer with bits and the number is negative, we must perform a sign extension by filling the remaining bits with 1s:

```
namespace {
    class cursor {
    public:
        --snip--
```

```
        std::int64_t sleb128() {
            std::uint64_t res = 0;
            int shift = 0;
            std::uint8_t byte = 0;
            do {
                byte = u8();
                auto masked = static_cast<uint64_t>(byte & 0x7f);
                res |= masked << shift;
                shift += 7;
            } while ((byte & 0x80) != 0);

         ❶ if ((shift < sizeof(res) * 8) and (byte & 0x40)) {
             ❷ res |= (~static_cast<std::uint64_t>(0) << shift);
            }

            return res;
        }
        --snip--
    };
}
```

You'll find the difference between the functions in the last `if` block ❶. We check whether we filled the result integer by comparing `shift` to the size of the result storage, and we determine whether the number should be negative by checking whether the last byte read has a 1 in its second-highest position. If so, we fill the remaining high bits of the result by bit-flipping 0 to obtain an integer with all bits set and then left-shifting the unnecessary ones off the end ❷.

Be careful about the types in this code. The `res` and `byte` variables are unsigned rather than signed integers. It's good practice to always use unsigned integers when doing bitwise operations, because doing them on signed integers may yield unexpected behavior. For example, in C++17, shifting a negative signed integer left is undefined behavior.

Extraction

Let's create an `sdb::abbrev` type to store the values held in an abbreviation table entry: an abbreviation code, a tag, a flag indicating whether the DIE has children, and a list of attribute specifications. Add the following to *sdb/include/libsdb/dwarf.hpp*:

```
#include <vector>
#include <cstdint>

namespace sdb {
    struct attr_spec {
        std::uint64_t attr;
        std::uint64_t form;
    };
```

```
    struct abbrev {
        std::uint64_t code;
        std::uint64_t tag;
        bool has_children;
        std::vector<attr_spec> attr_specs;
    };
    --snip--
}
```

These structures give us an easy way to collect all of the necessary data. Now we'll implement `parse_abbrev_table`. Let's build the function's main structure in *sdb/src/dwarf.cpp*:

```
#include <libsdb/elf.hpp>

namespace {
    std::unordered_map<std::uint64_t, sdb::abbrev>
    parse_abbrev_table(const sdb::elf& obj, std::size_t offset) {
        cursor cur(obj.get_section_contents(".debug_abbrev"));
        cur += offset;

        std::unordered_map<std::uint64_t, sdb::abbrev> table;
        std::uint64_t code = 0;
        do {
            // Parse one entry.
        } while (code != 0);

        return table;
    }
}
```

When we call `parse_abbrev_table`, we pass it the offset of the table to parse. We create a cursor from the contents of that section and then move that cursor forward the required number of bytes. Afterward, we declare an unordered map to use as the return value and a `code` variable that will store the parsed abbreviation code for use in our loop termination condition. We then loop, parsing entries, until we reach an entry whose code is `0`. Finally, we return the parsed table.

With the main structure out of the way, we can do the parsing:

```
std::unordered_map<std::uint64_t, sdb::abbrev>
parse_abbrev_table(const sdb::elf& obj, std::size_t offset) {
    --snip--
    do {
        code = cur.uleb128();
        auto tag = cur.uleb128();
        auto has_children = static_cast<bool>(cur.u8());

        std::vector<sdb::attr_spec> attr_specs;
```

```
        std::uint64_t attr = 0;
        do {
            attr = cur.uleb128();
            auto form = cur.uleb128();
            if (attr != 0) {
                attr_specs.push_back(sdb::attr_spec{ attr, form });
            }
        } while (attr != 0);

        if (code != 0) {
            table.emplace(code,
                sdb::abbrev{ code, tag, has_children, std::move(attr_specs) });
        }
    } while (code != 0);
    --snip--
}
```

Because it makes use of the `cursor` type we wrote, the parsing code reads like a straight description of the structure we're parsing. We extract the `ULEB128` for the code, the `ULEB128` for the tag, the 1-byte unsigned integer for the children flag (which will be either 1 or 0), and the list of `ULEB128` pairs of attribute types and forms, terminated by a pair of 0s.

Now that we've handled the abbreviation tables, we can parse the `.debug_info` section, beginning with the compile unit headers.

Parsing Compile Unit Headers

As we've discussed, the `.debug_info` section is split into information for each compile unit involved in the compilation of the program. Every compile unit begins with a *compile unit header*, which describes four important characteristics of that unit:

- A 4-byte unsigned integer representing the byte size of the information for this compile unit (excluding this field itself, but including the rest of the header)
- A 2-byte unsigned integer representing the DWARF version for this compile unit information (4, in our case)
- A 4-byte unsigned integer representing the offset into the `.debug_abbrev` section at which the abbreviation table for this compile unit begins (we'll pass this value to `sdb::dwarf::get_abbrev_table`)
- A 1-byte unsigned integer representing the byte size of an address on the system (8, in our case)

Let's make a type to store this information. Add the following to *sdb/include/libsdb/dwarf.hpp*:

```
#include <libsdb/types.hpp>

namespace sdb {
 ❶ class dwarf;
    class compile_unit {
    public:
     ❷ compile_unit(dwarf& parent,
                    span<const std::byte> data,
                    std::size_t abbrev_offset)
           : parent_(&parent)
           , data_(data)
           , abbrev_offset_(abbrev_offset) {}

     ❸ const dwarf* dwarf_info() const { return parent_; }
        span<const std::byte> data() const { return data_; }

     ❹ const std::unordered_map<std::uint64_t, sdb::abbrev>&
        abbrev_table() const;

    private:
     ❺ dwarf* parent_;
        span<const std::byte> data_;
        std::size_t abbrev_offset_;
    };
    --snip--
}
```

We start by forward-declaring `sdb::dwarf` ❶ so we can store a pointer to one inside of the compile unit. The constructor takes a reference to the parent `sdb::dwarf`, a span representing its data inside the `.debug_info` section, and the offset to the abbreviation table for that compile unit ❷. It has data members to store these ❺ and member functions to retrieve the parent and the data range ❸. Instead of exposing a member function to retrieve the abbreviation table offset, we declare a convenience function that directly retrieves the abbreviation table for this compile unit ❹.

Implement `abbrev_table` in *sdb/src/dwarf.cpp*:

```
const std::unordered_map<std::uint64_t, sdb::abbrev>&
sdb::compile_unit::abbrev_table() const {
    return parent_->get_abbrev_table(abbrev_offset_);
}
```

This function simply calls `get_abbrev_table` on the parent with the stored offset. Now we can add some members to `sdb:dwarf` to manage compile units. Make the following changes to *sdb/include/libsdb/dwarf.hpp*:

```
#include <memory>

namespace sdb {
```

```
    class dwarf {
    public:
        --snip--
        const std::vector<std::unique_ptr<compile_unit>>&
        compile_units() const { return compile_units_; }

    private:
        --snip--
        std::vector<std::unique_ptr<compile_unit>> compile_units_;
    };
}
```

Add a data member storing a vector of `sdb::compile_units` and a member function to retrieve them. We'll carry out the actual parsing in the constructor of `sdb::dwarf`. Implement it in *sdb/src/dwarf.cpp*:

```
sdb::dwarf::dwarf(const sdb::elf& parent) : elf_(&parent) {
    compile_units_ = parse_compile_units(*this, parent);
}
```

The constructor stores a pointer to the supplied parent and then calls a `parse_compile_units` function, which we'll write next. This function should keep trying to parse compile units until it hits the end of the `.debug_info` section:

```
namespace {
    std::vector<std::unique_ptr<sdb::compile_unit>> parse_compile_units(
          sdb::dwarf& dwarf, const sdb::elf& obj) {
        auto debug_info = obj.get_section_contents(".debug_info");
        cursor cur(debug_info);

        std::vector<std::unique_ptr<sdb::compile_unit>> units;
        while (!cur.finished()) {
            auto unit = parse_compile_unit(dwarf, obj, cur);
            cur += unit->data().size();
            units.push_back(std::move(unit));
        }

        return units;
    }
}
```

We start by obtaining a cursor for the `.debug_info` section. Until we hit the end of that section, we call a `parse_compile_unit` function that we'll write next to do the actual parsing, move the cursor forward by the size of that compile unit, and store the parsed unit. When we're done, we return a vector of all the compile units we parsed.

The `parse_compile_unit` function will use the `cursor` type to read the fields of the compile unit header and check for DWARF formats we don't support:

```
#include <libsdb/error.hpp>

namespace {
    std::unique_ptr<sdb::compile_unit> parse_compile_unit(
          sdb::dwarf& dwarf, const sdb::elf& obj, cursor cur) {
        auto start = cur.position();
        auto size = cur.u32();
        auto version = cur.u16();
        auto abbrev = cur.u32();
        auto address_size = cur.u8();

        if (size == 0xffffffff) {
            sdb::error::send("Only DWARF32 is supported");
        }
        if (version != 4) {
            sdb::error::send("Only DWARF version 4 is supported");
        }
        if (address_size != 8) {
            sdb::error::send("Invalid address size for DWARF");
        }

        size += sizeof(std::uint32_t);

        sdb::span<const std::byte> data = { start, size };
        return std::make_unique<sdb::compile_unit>(dwarf, data, abbrev);
    }
}
```

We start by saving the current cursor position, as we'll use it again later, then parse all the fields of the compile unit header one by one. If the first 4 bytes of the size field are `0xffffffff`, we're looking at a DWARF64 compile unit and should throw an error, as we don't support DWARF64. Likewise, we throw an error if the address size isn't 8 bytes and if the DWARF version is anything other than 4. Before returning the compile unit information, we add the size of a `std::uint32_t` to the compile unit size, because the reported size in the compile unit header doesn't include the size field itself.

We've now completed the parsing of compile unit headers and can move on to the most important part: parsing DIEs themselves.

Parsing DIEs

Recall that DWARF encodes DIEs in a tree structure. The root node of each compile unit is the DIE representing the compile unit itself. If a DIE has children, the abbreviation entry for that DIE has its children bit set to 1 (denoted by the enumerator `DW_CHLIDREN_yes`); otherwise, the bit is set to 0 (`DW_CHILDREN_no`). If a DIE doesn't have children, the DIE immediately following it in the `.debug_info` section is its sibling. If a DIE does have children, the

DIE immediately following it is its first child, and additional children are siblings of the first child. To terminate a chain of sibling entries, DWARF uses a null entry, which consists solely of the abbreviation code `0`. See Figure 12-1 for a simplified example of a tree's encoding.

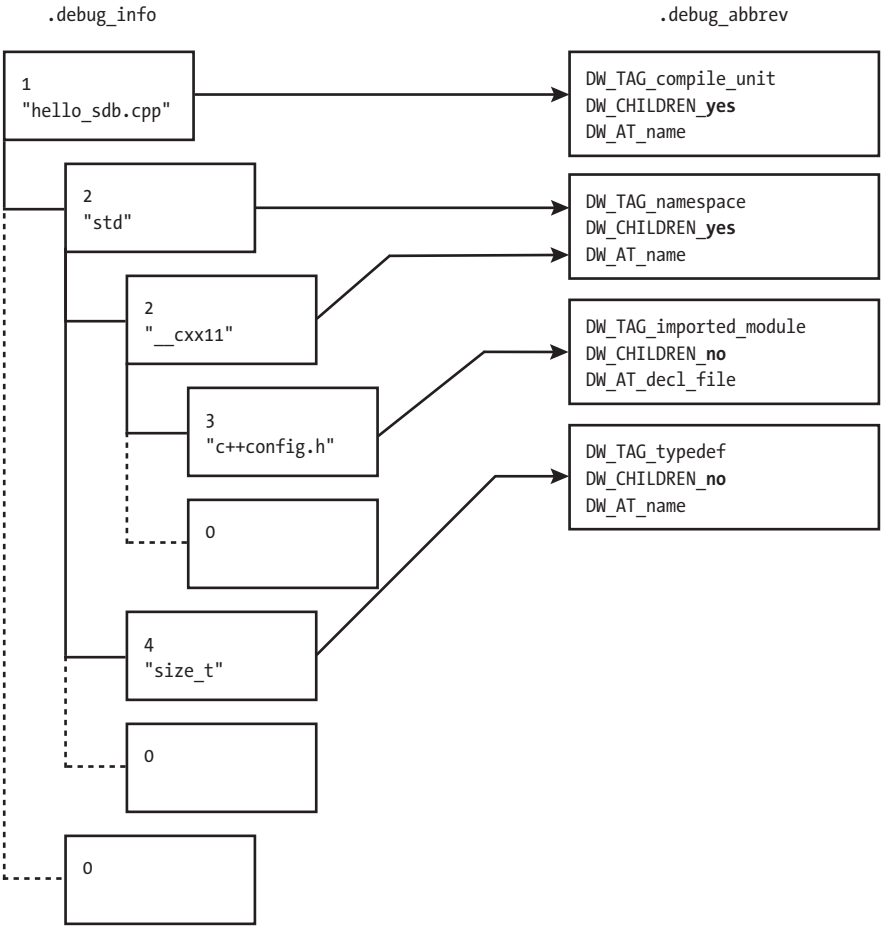


Figure 12-1: A simplified visualization of the DIE tree's encoding

The root node, or the DIE representing the compile unit, consists of the abbreviation code `1`, which points to the first entry of the `.debug_abbrev` section, and an attribute value that corresponds to the `DW_AT_name` attribute in the abbreviation entry. Because its abbreviation entry stores `DW_CHILDREN_yes`, the DIE has children, and the following DIE is its first child.

The second DIE points to the second abbreviation entry and represents the `std` namespace. It, too, has children. Its first child also points to the second abbreviation entry, as its structure is the same, and it represents the `__cxx11` namespace. This DIE's only child is the `DW_TAG_imported_module` DIE for the *c++config.h* header. The list of children for the `__cxx11` namespace DIE terminates with a DIE consisting solely of the abbreviation code `0`.

Following the termination of the __cxx11 DIE's children list is a DIE representing the size_t typedef. Because a 0 entry hasn't yet terminated the children list for the std DIE, the size_t DIE is a child of that DIE. Finally, two 0 entries terminate the children lists of the std DIE and the compile unit DIE, respectively.

Because the DIE trees for a program begin at the root nodes for each compile unit, we should expose the DIEs to users from the sdb::compile_unit type. Add a function to this class in *sdb/include/libsdb/dwarf.hpp* to obtain the root DIE for that compile unit:

```
namespace sdb {
    class die;
    class compile_unit {
    public:
        die root() const;
        --snip--
    };
}
```

You'll likely notice that we immediately find ourselves in need of an sdb::die type. Let's write the start of one. For now, let's forget about exposing attribute values and children to users and focus on what data the type needs to store:

- A pointer to the compile unit to which it belongs
- A pointer to its abbreviation table entry
- A pointer to the DIE immediately after this one (whether it be a child or a sibling)

We must store one more piece of data. We don't want to have to compute the location of the DIE's attributes every time one is requested, so we'll compute those locations once and store them in sdb::die. Define this type in *sdb/include/libsdb/dwarf.hpp*, below the definition for sdb::compile_unit:

```
namespace sdb {
    class die {
    public:
        explicit die(const std::byte* next) : next_(next) {}
        die(const std::byte* pos, const compile_unit* cu, const abbrev* abbrev,
            std::vector<const std::byte*> attr_locs, const std::byte* next) :
            pos_(pos), cu_(cu), abbrev_(abbrev),
            attr_locs_(std::move(attr_locs)), next_(next) {}

        const compile_unit* cu() const { return cu_; }
        const abbrev* abbrev_entry() const { return abbrev_; }
        const std::byte* position() const { return pos_; }
        const std::byte* next() const { return next_; }
```

```
    private:
        const std::byte* pos_ = nullptr;
        const compile_unit* cu_ = nullptr;
        const abbrev* abbrev_ = nullptr;
        const std::byte* next_ = nullptr;
        std::vector<const std::byte*> attr_locs_;
    };
}
```

We create two constructors: one for null DIEs that points `next_` to the next DIE in the section and one for non-null DIEs that stores the DIE's information in the object. As usual, we then create member functions to retrieve the information and data members to store it. Note that we don't expose the attribute locations to users; we'll merely use this optimization when we add support for retrieving attribute values to `sdb::die`.

Now we can implement the `sdb::compile_unit::root` function. We'll hand off much of the work to a `parse_die` function, which we'll reuse in several places. Implement `root` in *sdb/src/dwarf.cpp*:

```
sdb::die sdb::compile_unit::root() const {
    std::size_t header_size = 11;
    cursor cur({ data_.begin() + header_size, data_.end() });
    return parse_die(*this, cur);
}
```

The size of a compile unit header is 11 bytes, so the DIE for the compile unit is 11 bytes past the beginning of its `data_` member. We create a cursor for the range spanning the start of the compile unit DIE up until the end of the data for this compile unit, then call `parse_die` with a reference to this compile unit and the cursor we just created.

Now comes what you've all been waiting for: the function to actually parse a DIE. Recall that DWARF encodes a DIE as a `ULEB128` representing its abbreviation code, followed by a list of attribute values. In *sdb/src/dwarf.cpp*, begin the implementation by finding the abbreviation code and dealing with null DIEs:

```
namespace {
    sdb::die parse_die(const sdb::compile_unit& cu, cursor cur) {
        auto pos = cur.position();
        auto abbrev_code = cur.uleb128();

        if (abbrev_code == 0) {
            auto next = cur.position();
            return sdb::die{ next };
        }
```

We save the current position of the cursor so that we can store it in the resulting `sdb::die`. We then read a `ULEB128`. If it's 0, the DIE is a null DIE. The next DIE will be at the position currently pointed to by the cursor, so we save

this value and return a DIE with this pointer stored. If the DIE isn't null, we'll need to grab its abbreviation entry:

```
        --snip--
        auto& abbrev_table = cu.abbrev_table();
        auto& abbrev = abbrev_table.at(abbrev_code);
```

We get the abbreviation table for the compile unit containing this DIE by calling `cu.abbrev_table`. We then look up the nested table using the abbreviation code we just parsed to find the entry for the DIE.

The next portion of the function is a bit more complicated. We need to find the location of the next DIE, and we also want to precompute the locations for each attribute of this DIE. To do this, we need to know the size of each of its attributes. We can reference the attribute specifications held in the abbreviation entry we just found to get the forms used for each attribute value, but we need a way to get the size of a given form so we can skip over it.

We'll add a `skip_form` function to `cursor` that will advance the cursor by the required number of bytes. For now, let's pretend that this function already exists; we'll implement it when we're finished with `parse_die`. With this new function in our imaginations, we can write code to skip over each attribute in the DIE, saving their locations:

```
        --snip--
        std::vector<const std::byte*> attr_locs;
        attr_locs.reserve(abbrev.attr_specs.size());
        for (auto& attr : abbrev.attr_specs) {
            attr_locs.push_back(cur.position());
            cur.skip_form(attr.form);
        }
```

We first declare a `std::vector` to hold the locations. Because we know it's the same size as the attribution specification vector inside the abbreviation entry, we call `.reserve` to reserve the memory ahead of time, avoiding some unnecessary dynamic allocations. We then loop over each attribute specification, saving the current position of the cursor to the attribute locations vector and skipping over the current attribute using its form information.

Now all that remains is to return an `sdb::die` object with the values we just computed:

```
        --snip--
        auto next = cur.position();
        return sdb::die(pos, &cu, &abbrev, std::move(attr_locs), next);
    }
}
```

Because we skipped over all of the attributes in this DIE, the cursor should point to the next DIE, if there is one. We bundle all of the data we need into an `sdb::die` and return it. Now we can implement `cursor::skip_form`.

Implementing Form Skipping

There's no getting around it: skipping over attribute values requires handling every possible DWARF form in a big `switch` statement. While writing this code is inconvenient, I promise it will be worth the pain. It will also help you learn more about the available DWARF forms.

First, we'll write the function's main structure in *sdb/src/dwarf.cpp*:

```
namespace {
    class cursor {
    public:
        --snip--
        void skip_form(std::uint64_t form) {
            switch (form) {
            // Many things go here.
            default: sdb::error::send("Unrecognized DWARF form");
            }
        }
    };
}
```

We can roughly group DWARF forms based on their encodings, so we'll tackle them in these batches. Fixed-width forms are the simplest; they're always the same size, no matter their contents. This group includes the following:

Flags (`DW_FORM_flag` and `DW_FORM_flag_present`)
: Indicate the presence or absence of an attribute. The former flag is encoded as a single byte, where 1 means the attribute is present and 0 means it's not, while the latter flag always indicates presence, so its value requires no bytes of storage.

Fixed-width constants (`DW_FORM_data1,2,4,8`)
: An integer of the specified byte width. For example, `DW_FORM_data8` is 8 bytes wide.

Addresses (`DW_FORM_addr`)
: An integer of the byte width for addresses on the system (8 for x64).

Section offsets (`DW_FORM_sec_offset`)
: A 4-byte offset into a section other than `.debug_info` or `.debug_str`.

Fixed-width, unit-local references (`DW_FORM_ref1,2,4`)
: A 1-, 2-, or 4-byte offset from the start of the current compile unit header.

Global references (`DW_FORM_ref_addr`)
: A 4-byte integer offset to anywhere in the `.debug_info` section.

String references (`DW_FORM_strp`)
: A 4-byte integer offset into the `.debug_str` section.

This code can handle all of these forms:

```
--snip--
case DW_FORM_flag_present:
    break;

case DW_FORM_data1:
case DW_FORM_ref1:
case DW_FORM_flag:
    pos_ += 1; break;

case DW_FORM_data2:
case DW_FORM_ref2:
    pos_ += 2; break;

case DW_FORM_data4:
case DW_FORM_ref4:
case DW_FORM_ref_addr:
case DW_FORM_sec_offset:
case DW_FORM_strp:
    pos_ += 4; break;

case DW_FORM_data8:
case DW_FORM_addr:
    pos_ += 8; break;
--snip--
```

As you can see, nothing particularly interesting is happening here. We merely map each form to movements of the current cursor position. Let's move on to the next group.

DWARF encodes `LEB128` forms with either `ULEB128` or `SLEB128` integers. This group covers the `LEB128` constants `DW_FORM_udata` and `DW_FORM_sdata`, which are integers encoded in `ULEB128` and `SLEB128`, respectively, and the variable-width, unit-local references (`DW_FORM_ref_udata`), which are `ULEB128` offsets from the start of the current compile unit header. We can handle these with the following code:

```
--snip--
case DW_FORM_sdata:
    sleb128(); break;
case DW_FORM_udata:
case DW_FORM_ref_udata:
    uleb128(); break;
--snip--
```

We parse either an `SLEB128` or a `ULEB128` and throw away the result.

Next, we tackle explicitly sized forms, whose size is encoded in the attribute value itself. This group covers data block forms (`DW_FORM_block` and `DW_FORM_block{1,2,4}`), which are blocks of data whose the size is encoded

as a ULEB128 or fixed-sized integer, respectively, and DWARF expressions (DW_FORM_exprloc) for computing the locations of variables, prefixed with the size of the expression as a ULEB128. To skip over these forms, we must parse the relevant integer and advance the current position by that number of bytes:

```
--snip--
case DW_FORM_block1:
    pos_ += u8();
    break;
case DW_FORM_block2:
    pos_ += u16();
    break;
case DW_FORM_block4:
    pos_ += u32();
    break;
case DW_FORM_block:
case DW_FORM_exprloc:
    pos_ += uleb128();
    break;
--snip--
```

The final group covers the as yet unaddressed forms, namely strings (DW_FORM_string), or sequences of contiguous non-null bytes followed by one null byte, and indirect forms (DW_FORM_indirect), a value encoded as a ULEB128 that gives its actual form, followed by an attribute encoded in that form. For strings, we simply keep reading bytes until we get one that is 0:

```
case DW_FORM_string:
    while (!finished() && *pos_ != std::byte(0)) {
        ++pos_;
    }
    ++pos_;
    break;
```

Ensure that you remember to include the second ++pos_ to advance past the null terminator. Lastly, indirect forms might sound complicated to handle, but we can actually just read the form from the cursor and call skip_form recursively with the result:

```
case DW_FORM_indirect:
    skip_form(uleb128());
    break;
```

All done! Now we can move on to a somewhat more interesting engineering problem: traversing the tree of DIEs.

Traversing the DIE Tree

Users of the DWARF parser can now retrieve the root DIEs for each compile unit. However, this isn't very useful on its own; we need to be able to visit the children of those nodes, and those children's children, and so forth. In the `sdb::stoppoint_collection` class, we solved this problem by writing a `for_each` function template. You could do this here too, but due to the deeply nested structure of DIEs, this code would become very ugly very fast. Instead, we'll write a small range type that we can use to iterate over the children of DIEs.

Add a nested class declaration and member function to `sdb::die` to facilitate this in *sdb/include/libsdb/dwarf.hpp*:

```
namespace sdb {
    class die {
    public:
        --snip--
        class children_range;
        children_range children() const;
        --snip--
    };
}
```

We won't worry about making the range type conform to the expectations of C++20 ranges and will instead err on the side of simplicity. The `sdb::die::children_range` type will wrap a DIE and provide `begin` and `end` member functions that we can call to retrieve iterators to the children. This will allow us to write code like the following:

```
for (auto child : my_die.children()) {
    do_something(child);
}
```

Here is the implementation, omitting the code for the `iterator` type for now:

```
namespace sdb {
    class die::children_range {
    public:
        children_range(die die) : die_(std::move(die)) {}
        class iterator {
            // Blank for now
        };
        iterator begin() const {
            if (die_.abbrev_->has_children) {
                return iterator{ die_ };
            }
            return end();
        }
        iterator end() const { return iterator{}; }
```

```
    private:
        die die_;
    };
}
```

The constructor takes an `sdb::die` and stores it. The `begin` function returns an iterator constructed with the stored DIE, so long as that DIE has children. Otherwise, it returns the end iterator. The `end` function returns an empty iterator that marks the end of the range.

In order to return child DIEs, we'll need to parse them at some point. We could parse the entire tree ahead of time and store our own representation in memory. However, these trees can get very large. We'll be much better off parsing them on demand whenever the iterator type of a `children_range` is incremented.

The implementation of the iterator type should support these operations: default construction, leaving the iterator empty (generally expected of iterators in C++); construction with an `sdb::die`; copy construction and assignment, so we can easily copy iterators; `operator*` and `operator->`, to access the wrapped DIE; `operator++`, for advancing to the next DIE; and `operator==` and `operator!=`, for determining when we've finished iterating.

The tricky behaviors to get right are the increment and equality operators; the rest are pretty straightforward. Here is the declaration:

```
#include <optional>

namespace sdb {
    class die::children_range {
    public:
        --snip--
        class iterator {
        public:
         ❶ using value_type = die;
           using reference = const die&;
           using pointer = const die*;
           using difference_type = std::ptrdiff_t;
           using iterator_category = std::forward_iterator_tag;

         ❷ iterator() = default;
           iterator(const iterator&) = default;
           iterator& operator=(const iterator&) = default;

         ❸ explicit iterator(const die& die);

         ❹ const die& operator*() const { return *die_; }
           const die* operator->() const { return &die_.value(); }

         ❺ iterator& operator++();
```

```
            iterator operator++(int);

          ❻ bool operator==(const iterator& rhs) const;
          ❼ bool operator!=(const iterator& rhs) const {
                return !(*this == rhs);
            }

        private:
          ❽ std::optional<die> die_;
        };
        --snip--
    };
}
```

To use iterator types with algorithms from the `<algorithm>` or `<numeric>` headers, the types must expose a few pieces of information about themselves. We achieve this by adding five type aliases ❶ to the iterator type: `value_type` is the type being iterated over, `reference` is the type returned by `operator*`, `pointer` is the type returned by `operator->`, `difference_type` is the type returned by subtracting two iterators, and `iterator_category` is the iterator's category. The category can be one of the following:

`std::output_iterator_tag` Iterators that can be written to (such as `std::ostream_iterator`)

`std::input_iterator_tag` One-pass iterators

`std::forward_iterator_tag` Multipass iterators

`std::bidirectional_iterator_tag` Iterators that can be both incremented and decremented

`std::random_access_iterator_tag` Iterators that let you move to an arbitrary element in constant time

In our case, we default to setting `difference_type` to the difference between pointers (`std::ptrdiff_t`) because we won't support subtracting iterators. The iterator category is `std::forward_iterator_tag` because the iterators can move forward only, but you can pass over the range multiple times by saving an iterator before incrementing it.

We use the default and copy constructors and assignment operators ❷, because those operations will work as they are. The constructor that takes an `sdb::die` needs a bit more work, so we just declare it for now ❸, being sure to mark it as `explicit` to avoid sneaky implicit conversions. The indirection operators return a reference or pointer to the stored `die` ❹. The increment operators ❺ and equality operator ❻ also require more work, so we just declare them for now.

In case you're not familiar with the `operator++(int)` function, note that the `int` parameter isn't a real parameter; it's just a way to distinguish the pre-increment (`++value`) and post-increment (`value++`) operators. The post-increment operator takes a dummy `int` parameter. We implement the

inequality operator ❼ in terms of the equality operator. Finally, we store the `sdb::die` wrapped in a `std::optional` to support default construction ❽.

We've declared three other functions that we need to implement, so let's do so now, starting with the default constructor, which should parse the first child DIE of the one that is given. Implement it in *sdb/src/dwarf.cpp*:

```
sdb::die::children_range::iterator::iterator(const sdb::die& d) {
    cursor next_cur({ d.next_, d.cu_->data().end() });
    die_ = parse_die(*d.cu_, next_cur);
}
```

We create a cursor that starts at the first child of the DIE and can go up to the end of the compile unit that owns it. We then call `parse_die` to retrieve an `sdb::die` representing the contents.

We'll write the equality operator next, as it's simpler than the increment one. This operator should test whether two iterators are either pointing to the same DIE or both null:

```
bool sdb::die::children_range::iterator::operator==(
      const iterator& rhs) const {
    auto lhs_null = !die_.has_value() or !die_->abbrev_entry();
    auto rhs_null = !rhs.die_.has_value() or !rhs.die_->abbrev_entry();
    if (lhs_null and rhs_null) return true;
    if (lhs_null or rhs_null) return false;

    return die_->abbrev_ == rhs->abbrev_ and die_->next() == rhs->next();
}
```

An iterator is null if it doesn't have a DIE stored or the stored DIE has an abbreviation code of `0`. As such, we first check if both iterators are null. If so, they're equal. If only one of them is null, they're not equal. If both are valid, we return whether their abbreviation codes and next DIE pointers match.

Let's move on to the increment operator. This one is a bit trickier. If the DIE we're looking at has no children, the next sibling is just the next DIE, so we parse the DIE at `next_`. Otherwise, we'll need to skip over all the children of this DIE to find the sibling. We can do so using a nested iterator. Here's the code:

```
sdb::die::children_range::iterator&
sdb::die::children_range::iterator::operator++() {
 ❶ if (!die_.has_value() or !die_->abbrev_) return *this;

 ❷ if (!die_->abbrev_->has_children) {
        cursor next_cur({ die_->next_, die_->cu_->data().end() });
        die_ = parse_die(*die_->cu_, next_cur);
    }
```

```
    else {
     ❸ iterator sub_children(*die_);
     ❹ while (sub_children->abbrev_) ++sub_children;
        cursor next_cur({ sub_children->next_, die_->cu_->data().end() });
     ❺ die_ = parse_die(*die_->cu_, next_cur);
    }
    return *this;
}
```

If the iterator is null, we exit early ❶. If the DIE is valid and doesn't have children, we build a cursor from `next_` to the end of the compile unit and parse the DIE there ❷. If the DIE does have children, we build an iterator to iterate over them ❸ and advance it until we hit a null entry ❹. The next DIE after the null entry is the sibling we're looking for, so we parse that ❺.

We implement the post-increment operator in terms of the pre-increment operator:

```
sdb::die::children_range::iterator
sdb::die::children_range::iterator::operator++(int) {
    auto tmp = *this;
    ++(*this);
    return tmp;
}
```

We've followed a common C++ pattern: implementing a pre-increment function and then writing the post-increment equivalent by copying the current iterator, calling pre-increment on `*this`, and returning the unmodified copy. With all of this work done, we can duct tape it together by implementing `sdb::die::children`:

```
sdb::die::children_range sdb::die::children() const {
    return children_range(*this);
}
```

Note that DWARF provides a quick way for compilers to get the sibling of a DIE: they can encode a reference to the sibling using a `DW_AT_sibling` attribute. However, this requires us to be able to get the values of attributes, which we can't do yet. Let's add support for attributes and then come back to our iterator type.

Reading Attributes

Every DIE has a set of attributes, each with a type, a form, and a value. Importantly, a DIE can't have multiple attributes of the same type defined; it can't have two different names, two different locations, two different siblings, and so on. This means we can create a very simple interface for our DIEs' attributes: one function to check whether an attribute exists and one function to retrieve its value.

We'll add a `contains` function for the former and an `operator[]` for the latter, so that users can write something like `my_die[DW_AT_name]` to get the name of the DIE. We'll also need a type representing the value of an attribute, which we'll call `attr`. Add these declarations to *sdb/include/libsdb/dwarf.hpp*:

```
namespace sdb {
    class die {
    public:
        --snip--
        bool contains(std::uint64_t attribute) const;
        attr operator[](std::uint64_t attribute) const;
        --snip--
    };
}
```

Both of these functions take an attribute identifier, like `DW_AT_sibling`. The `contains` function returns whether the DIE has an attribute of that type, whereas `operator[]` retrieves the value. Implement them in *sdb/src/dwarf.cpp*, starting with `contains`. This function should look through the attribute specifications in this DIE's abbreviation entry and check for one matching the given attribute type:

```
bool sdb::die::contains(std::uint64_t attribute) const {
    auto& specs = abbrev_->attr_specs;
    return std::find_if(begin(specs), end(specs),
        [=](auto spec) { return spec.attr == attribute; }) != end(specs);
}
```

We grab the attribute specifications from the abbreviation entry and then use `std::find_if` to find a matching specification. If one exists, `std::find_if` will return an iterator that points to it; otherwise, it will return the end iterator. So to check whether this DIE contains a matching attribute, we return whether `std::find_if` returned an iterator not matching the end iterator.

Let's move on to `operator[]`. Here, the `attr` type will need the following information to interpret the bytes of the attribute value correctly:

- A pointer to the compilation unit it belongs to, as some forms encode offsets from the start of the current compilation unit
- The attribute's type
- The attribute's form
- Where the bytes for the attribute's value begin

Let's write the code:

```
sdb::attr sdb::die::operator[](std::uint64_t attribute) const {
    auto& specs = abbrev_->attr_specs;
    for (std::size_t i = 0; i < specs.size(); ++i) {
```

```
        if (specs[i].attr == attribute) {
            return { cu_, specs[i].attr, specs[i].form, attr_locs_[i] };
        }
    }

    error::send("Attribute not found");
}
```

We loop through the attribute specifications, looking for one that matches the attribute we're looking for. If we find one, we return an `sdb::attr` constructed from the compile unit, the attribute type and form, and the location of the start of the relevant attribute we precomputed earlier. Because we order the `attr_locs_` list the as the `attr_specs` list, we use the same index to access the matching elements from each. If no attribute of the requested type exists, we throw an exception.

Now the `sdb::attr` type should store all of the data that we passed to the constructor in `operator[]` and expose functions for retrieving the attribute value depending on what the form is. For now, we'll support getting the value as an address, a block of data, an integer, a string, or a reference to a DIE. We'll need to add more options in the future, when we implement DWARF expressions and range lists, but these options will do for now. Define the type in *sdb/include/libsdb/dwarf.hpp*, right at the top of the `sdb` namespace:

```
#include <string_view>

namespace sdb {
 ❶ class compile_unit;
    class die;
    class attr {
    public:
     ❷ attr(const compile_unit* cu, std::uint64_t type,
             std::uint64_t form, const std::byte* location) :
            cu_(cu), type_(type), form_(form), location_(location) {}

     ❸ std::uint64_t name() const { return type_; }
        std::uint64_t form() const { return form_; }

     ❹ file_addr as_address() const;
        std::uint32_t as_section_offset() const;
        span<const std::byte> as_block() const;
        std::uint64_t as_int() const;
        std::string_view as_string() const;
        die as_reference() const;

    private:
     ❺ const compile_unit* cu_;
        std::uint64_t type_;
```

```
        std::uint64_t form_;
        const std::byte* location_;
    };
    --snip--
}
```

We forward-declare `sdb::compile_unit` ❶ and `sdb::die` so we can refer to them. In the implementation of `sdb::attr`, we write a constructor ❷ that simply takes all of the data we passed in `sdb::die::operator[]` and stores it in data members we define next ❺. We expose the attribute name and form ❸ but not the compile unit or attribute location, which are for internal use only. We then declare all of those value-reading functions I mentioned ❹. Let's implement them now.

We'll start with `as_address`. Because we're limiting ourselves to x64, we can just read a single 64-bit integer from the start of the attribute bytes and return it. Add the following in *sdb/src/dwarf.cpp*:

```
sdb::file_addr sdb::attr::as_address() const {
    cursor cur({ location_, cu_->data().end() });
    if (form_ != DW_FORM_addr) error::send("Invalid address type");
    auto elf = cu_->dwarf_info()->elf_file();
    return file_addr{ *elf, cur.u64() };
}
```

We create a cursor for the data range and read a `std::uint64_t` with it. If the form isn't `DW_FORM_addr`, we throw an exception, because addresses are always of this form. This address is a file address in the ELF file to which this DWARF information belongs, so we return an `sdb::file_addr` with the relevant `sdb::elf` object.

The `as_section_offset` function is similar, except that section offsets are 32 bits in 32-bit DWARF and use the form `DW_FORM_sec_offset`:

```
std::uint32_t sdb::attr::as_section_offset() const {
    cursor cur({ location_, cu_->data().end() });
    if (form_ != DW_FORM_sec_offset) error::send("Invalid offset type");
    return cur.u32();
}
```

The other functions are slightly more complicated because we must vary their behavior depending on what form the attribute is encoded with. Let's do `as_int` next. This function should check the form for the size of integer to parse:

```
std::uint64_t sdb::attr::as_int() const {
    cursor cur({ location_, cu_->data().end() });
    switch (form_) {
    case DW_FORM_data1:
        return cur.u8();
```

```
    case DW_FORM_data2:
        return cur.u16();
    case DW_FORM_data4:
        return cur.u32();
    case DW_FORM_data8:
        return cur.u64();
    case DW_FORM_udata:
        return cur.uleb128();
    default:
        error::send("Invalid integer type");
    }
}
```

Again, we create a cursor for the data range. We then switch on the form of the attribute and parse the relevant integer type. If the form isn't one we expect, we throw an exception.

Next, we'll handle blocks, which DWARF encodes as a size followed by data. The size can be an integer of 1, 2, or 4 bytes, or a `ULEB128`. As such, we can read the size and then return a span that begins immediately after the integer we just read:

```
sdb::span<const std::byte> sdb::attr::as_block() const {
    std::size_t size;
    cursor cur({ location_, cu_->data().end() });
    switch (form_) {
    case DW_FORM_block1:
        size = cur.u8();
        break;
    case DW_FORM_block2:
        size = cur.u16();
        break;
    case DW_FORM_block4:
        size = cur.u32();
        break;
    case DW_FORM_block:
        size = cur.uleb128();
        break;
    default:
        error::send("Invalid block type");
    }
    return { cur.position(), size };
}
```

This is a pretty similar process to the one for integers: get a cursor, parse the values we want, throw an exception on surprises, and return the requested data.

The code for DIE references will also look similar. We'll parse an offset of the correct size for the attribute's form, then parse the DIE that exists at that offset from the start of the compile unit and return it. Parsing `DW_FORM_ref_addr` is a bit more complicated, so I've left that out for now:

```
sdb::die sdb::attr::as_reference() const {
    cursor cur({ location_, cu_->data().end() });
    std::size_t offset;
    switch (form_) {
    case DW_FORM_ref1:
        offset = cur.u8(); break;
    case DW_FORM_ref2:
        offset = cur.u16(); break;
    case DW_FORM_ref4:
        offset = cur.u32(); break;
    case DW_FORM_ref8:
        offset = cur.u64(); break;
    case DW_FORM_ref_udata:
        offset = cur.uleb128(); break;
    case DW_FORM_ref_addr: {
        // Something complicated
    }
    default:
        error::send("Invalid reference type");
    }

    cursor ref_cur({ cu_->data().begin() + offset, cu_->data().end() });
    return parse_die(*cu_, ref_cur);
}
```

At the end of the function, we make a new cursor based on the offset we just read and parse the DIE at that position. All of these forms' offsets are based on the start of the current DIE's compile unit, so we add that offset to `cu_->data().begin()` to calculate the position of the referenced DIE. The `DW_FORM_ref_addr` form, on the other hand, can reference data in other compile units, so its offset is relative to the start of the `.debug_info` section. I'll show you the code to calculate this, then walk through it:

```
case DW_FORM_ref_addr: {
    offset = cur.u32();
    auto section = cu_->dwarf_info()->elf_file()->get_section_contents(".debug_info");
    auto die_pos = section.begin() + offset;
    auto& cus = cu_->dwarf_info()->compile_units();
    auto cu_finder = [=](auto& cu) {
        return cu->data().begin() <= die_pos and cu->data().end() > die_pos;
    };
```

```
        auto cu_for_offset = std::find_if(begin(cus), end(cus), cu_finder);
        cursor ref_cur({ die_pos, cu_for_offset->get()->data().end() });
        return parse_die(**cu_for_offset, ref_cur);
    }
```

First, we read the offset from the start of the `.debug_info` section and ask the parent ELF file for the contents of that section. Given these contents, we add the offset to the beginning of the section to the parsed offset to calculate the position of the referenced DIE.

The difficult part is that, to parse a DIE, we need to know to which compile unit it belongs, but all we have is an offset. As such, we look through all of the compile units contained in the debug information for a compile unit whose data range contains the DIE we're looking for. We do this by grabbing the list of compile units from the debug information; defining a lambda that, given a compile unit, returns whether `die_pos` is in the data range of that compile unit; and then using `std::find_if` to locate the correct compile unit. With the correct unit found, we create a cursor for the referenced DIE position and parse the DIE there.

That was a bit tough, but we can put references aside for now and move on to strings, which are easier. DWARF encodes strings either as `DW_FORM_string`, a null-terminated string directly embedded in the DIE, or as `DW_FORM_strp`, a 4-byte offset into the `.debug_str` section that indicates where the string lives. Here is the code to retrieve a string attribute as a `std::string_view`:

```
std::string_view sdb::attr::as_string() const {
    cursor cur({ location_, cu_->data().end() });
    switch (form_) {
    case DW_FORM_string:
        return cur.string(); ❶
    case DW_FORM_strp: {
        auto offset = cur.u32();
        auto stab = cu_->dwarf_info()->elf_file()->get_section_contents(".debug_str");
        cursor stab_cur({ stab.begin() + offset, stab.end() });
        return stab_cur.string(); ❷
    }
    default:
        error::send("Invalid string type");
    }
}
```

We create a cursor. If we're looking at a string embedded in the DIE, we just parse it and return it ❶. If instead we're looking at a string reference, we parse the offset into the `.debug_str` section, get the contents of that section, and then parse the string from there ❷. As usual, if we get a form we didn't expect, we throw an error.

Hooray! We can now read attributes from DIEs. This means we can go back to the child iterator and add support for the `DW_AT_sibling` attribute,

which will speed up iteration significantly for large DIE trees. We'll also add some other extensions based on this new ability we've added to the parser.

Augmenting DIEs with Attribute Support

In this section, we'll round out the facilities provided by `sdb::die`. We'll optimize the child iterator type and add helper functions to retrieve the DIE's address range. These features will enable us to locate the compile unit to which an address belongs, which will be useful for finding the function we're in when the inferior halts.

Optimizing Child Iterators

Let's add support for `DW_AT_sibling` to the `sdb::die::children_range::iterator` type. (Yes, it's quite a mouthful.) Because we already added reference support to the cursor, supporting this type is actually rather straightforward. When we increment an iterator on a DIE that has children, we should first check whether the DIE has a `DW_AT_sibling` attribute. If so, we should retrieve the referenced DIE rather than creating a nested iterator and skipping over every child and nested child individually. Edit the implementation of `operator++` in *sdb/src/dwarf.cpp*:

```
sdb::die::children_range::iterator&
sdb::die::children_range::iterator::operator++() {
    if (!die_.has_value() or !die_->abbrev_) return *this;

    if (!die_->abbrev_->has_children) {
        --snip--
    }
 ❶ else if (die_->contains(DW_AT_sibling)) {
        die_ = die_.value()[DW_AT_sibling].as_reference();
    }
    else {
        --snip--
    }
    return *this;
}
```

Between the existing branches, we add a new `else if` branch ❶ that checks whether the DIE contains a sibling attribute. If so, we set `die_` to the result of interpreting that attribute value as a DIE reference. This can be a huge optimization, potentially saving us from parsing hundreds of nested DIEs and allowing us to skip directly to the sibling.

The last thing we need to add before testing the parser is support for retrieving the address ranges of an `sdb::die`.

Extracting DIE Address Ranges

DWARF can encode address ranges in two ways. To express contiguous address ranges, it uses a pair of `DW_AT_low_pc` and `DW_AT_high_pc` attributes (where PC stands for *program counter*). For noncontiguous address ranges, it uses a `DW_AT_ranges` attribute, which references a range list entry in the `.debug_ranges` section. We'll handle contiguous ranges first.

Contiguous

The `DW_AT_high_pc` attribute is a bit tricky, because DWARF can express it either as a `DW_FORM_addr`, in which case it's a direct virtual address, or as any of the integer forms, in which case it's an offset from the the `DW_AT_low_pc` attribute, which is always encoded as a `DW_FORM_addr`. In *sdb/include/libsdb/dwarf.hpp*, let's add a couple of member functions to `sdb::die` for handling these attributes:

```
namespace sdb {
    class die {
    public:
        --snip--
        file_addr low_pc() const;
        file_addr high_pc() const;
        --snip--
    };
}
```

As usual, implement the functions in *sdb/src/dwarf.cpp*:

```
sdb::file_addr sdb::die::low_pc() const {
    return (*this)[DW_AT_low_pc].as_address();
}

sdb::file_addr sdb::die::high_pc() const {
    auto attr = (*this)[DW_AT_high_pc];
    std::uint64_t addr;
    if (attr.form() == DW_FORM_addr) {
        addr = attr.as_address();
    }
    else {
        addr = low_pc() + attr.as_int();
    }
    return file_addr{
        *cu_->dwarf_info()->elf_file(),
        addr
    };
}
```

The low program counter value is always a file address, so we just grab the relevant `sdb::attr` and call the `as_address` function on it. We return an `sdb::file_addr` that points to the ELF file that contains this DWARF information.

For the high program counter value, we check the form. If the form is an address, we extract it. Otherwise, the form must be an offset from the low program counter, so we extract the low program counter and then offset it with the high program counter attribute as an integer. Now we can move on to range lists.

Noncontiguous

Range lists are more complicated than simple high/low pairs, and we'll need to add a new `sdb::range_list` type to make them easily usable. First, let's look at the encoding of the `.debug_range` section.

The section consists of a series of entries of three possible kinds: regular range list entries; base address selectors, which change how we should interpret regular entries; and end-of-list indicators. All range list entries consist of two integers with a byte size identical to the address size of the machine (8 bytes, on x64). For base address selectors, these integers are the following, respectively:

1. An integer with all bits set, which indicates that this entry is a base address selector
2. An integer that sets the base address from which all future range list entries should be considered an offset (until the base address is changed again or the list ends)

Regular entries instead use the following integers:

1. A beginning address offset relative to the current base address
2. An ending address offset relative to the current base address

Each pair of offsets, along with the base address, gives a range of addresses covered by this DIE. These ranges can't overlap.

If no base address selector entry precedes the current one, the base address gets encoded as the `DW_AT_low_pc` attribute in the DIE that is referencing the range list. Such a DIE will have a `DW_AT_low_pc` attribute but not a matching `DW_AT_high_pc` attribute.

An end-of-list indicator has both integers set to 0.

Let's parse this information with an `sdb::range_list` type. As with DIE trees, range lists can become very large, so we'll parse them lazily. We should be able to construct the range list type with an `sdb::span` from the start of the range list and up to the end of the `.debug_range` section (because we don't know how long a range list will be). It should optionally take the base address stored in the DIE's `DW_AT_low_pc` attribute, if there is one. Also, it should have a way to check whether any range in the list contains a given address, as well as a way to iterate over the list. As we did for DIE children, we'll write a small iterator type to achieve all of this. Define the type in *sdb/include/libsdb/dwarf.hpp*, above `sdb::attr`:

```
namespace sdb {
    class compile_unit;
    class range_list {
    public:
     ❶ range_list(
        const compile_unit* cu, span<const std::byte> data,
        file_addr base_address)
            : cu_(cu), data_(data), base_address_(base_address) {}

     ❷ struct entry {
            file_addr low;
            file_addr high;

            bool contains(file_addr addr) const {
                return low <= addr and addr < high;
            }
        };

     ❸ class iterator;
        iterator begin() const;
        iterator end() const;

     ❹ bool contains(file_addr address) const;

    private:
     ❺ const compile_unit* cu_;
        span<const std::byte> data_;
        file_addr base_address_;
    };
}
```

We write a constructor ❶ that fills in the data members we supply at the bottom of the class ❺. For range list entries, we define a small nested type ❷ that wraps a low and a high address and supplies a member function to check whether a given address is within that range. We merely declare the iterator type for now ❸, alongside `begin` and `end` functions; we'll implement these next. We also declare a function that will check whether a given address is in any entry of the range list ❹.

Next is the iterator type, which should present a view of the logical list of range entries, with base address entries resolved internally to the iterator and not exposed to users. We'll implement almost all of this work in `operator++`. Here is the type definition:

```
namespace sdb {
    class range_list::iterator {
    public:
     ❶ using value_type = entry;
        using reference = const entry&;
```

```
        using pointer = const entry*;
        using difference_type = std::ptrdiff_t;
        using iterator_category = std::forward_iterator_tag;

    ❷ iterator(
            const compile_unit* cu,
            span<const std::byte> data,
            file_addr base_address);

    ❸ iterator() = default;
        iterator(const iterator&) = default;
        iterator& operator=(const iterator&) = default;

    ❹ const entry& operator*() const { return current_; }
        const entry* operator->() const { return &current_; }

    ❺ bool operator==(iterator rhs) const { return pos_ == rhs.pos_; }
        bool operator!=(iterator rhs) const { return pos_ != rhs.pos_; }

    ❻ iterator& operator++();
        iterator operator++(int);

    private:
    ❼ const compile_unit* cu_ = nullptr;
        span<const std::byte> data_{ nullptr,nullptr };
        file_addr base_address_;
        const std::byte* pos_ = nullptr;
        entry current_;
    };
}
```

Like we did for the child iterators, we expose type aliases that enable us to use the iterator type with standard algorithms ❶. Also, like child iterators, range list iterators are multipass, meaning we'll use `std::forward_iterator_tag` as the iterator category. The constructor ❷ takes the same data and base address as the `range_list` itself. Iterators should be default-constructible and copyable, and the default versions will work fine, so we default them ❸. The iterator holds the entry currently being pointed to, so the `operator*` and `operator->` functions simply return it ❹. We determine equality using the `pos_` member, which we set as `nullptr` for the end-of-range iterator ❺. We then declare the increment operators ❻ and all of the member data ❼, making sure to give the members default values so that the default constructor works.

Now let's implement the member functions, starting with the constructor. This function should initialize the data members and call `operator++` to prime the first element of the range. Implement it in *sdb/src/dwarf.cpp*:

```
sdb::range_list::iterator::iterator(
        const compile_unit* cu,
        sdb::span<const std::byte> data,
        file_addr base_address)
    : cu_(cu), data_(data)
    , base_address_(base_address)
    , pos_(data.begin()) {
    ++(*this);
}
```

We initialize the members and then call `operator++` using `++(*this)`.

The `operator++` function is where the actual parsing work happens. Recall that each entry consists of two 64-bit integers, and we have three types of entries to handle: regular entries, base address entries (whose first integer is the integer with all bits set to 1), and end-of-list indicators (where both integers are 0). Here's the code:

```
sdb::range_list::iterator&
sdb::range_list::iterator::operator++() {
    auto elf = cu_->dwarf_info()->elf_file();
    constexpr auto base_address_flag = ~static_cast<std::uint64_t>(0);

 ❶ cursor cur({ pos_, data_.end() });
    while (true) {
        current_.low = file_addr{ *elf, cur.u64() };
        current_.high = file_addr { *elf, cur.u64() };

     ❷ if (current_.low.addr() == base_address_flag) {
            base_address_ = current_.high;
        }
     ❸ else if (current_.low.addr() == 0 and current_.high.addr() == 0) {
            pos_ = nullptr;
            break;
        }
     ❹ else {
            pos_ = cur.position();
            current_.low += base_address_.addr();
            current_.high += base_address_.addr();
            break;
        }
    }

    return *this;
}
```

We begin by grabbing a pointer to the ELF file for this range list and computing the flag used to indicate base address entries. This flag is a 64-bit integer with all bits set to 1, which we can produce by taking an integer with all bits set to 0 and using the complement (~) operator. We then initialize a cursor with which to read the integers ❶.

After the setup code, we parse the list in a loop, where we continue to read pairs of integers. If the first of the pair is the base address entry flag we computed earlier ❷, we set the base address to the value of the second integer. If both integers are 0, we have an end-of-list indicator ❸, so we set `pos_` to `nullptr` to indicate that the iteration is complete and break out of the loop. Otherwise, the entry is regular ❹, so we save the current position of the cursor, increment the current addresses by the base address to calculate the correct values for this entry, and break out of the loop.

Now we can implement the post-increment operator in terms of the pre-increment operator:

```
sdb::range_list::iterator
sdb::range_list::iterator::operator++(int) {
    auto tmp = *this;
    ++(*this);
    return tmp;
}
```

This operator is similar to the one we wrote for the child iterators, except it uses a different iterator type.

With the parser written, we can implement `sdb::attr::as_range_list` to parse an attribute as an `sdb::range_list`. First, declare it in *sdb/include/libsdb/dwarf.hpp*:

```
namespace sdb {
    class attr {
    public:
        --snip--
        range_list as_range_list() const;
        --snip--
    };
}
```

Then, define it in *sdb/src/dwarf.cpp*:

```
sdb::range_list sdb::attr::as_range_list() const {
 ❶ auto section = cu_->dwarf_info()->elf_file()->get_section_contents(
    ".debug_ranges");
 ❷ auto offset = as_section_offset();
 ❸ span<const std::byte> data(section.begin() + offset, section.end());

    auto root = cu_->root();
 ❹ file_addr base_address = root.contains(DW_AT_low_pc)
        ? root[DW_AT_low_pc].as_address()
```

```
        : file_addr{};

    return { cu_, data, base_address };
}
```

DWARF encodes range list attributes as offsets into the `.debug_ranges` section. We grab the contents of that section from the ELF file ❶ and then parse the offset stored at the current attribute position ❷. With these two elements, we compute the potential data range for the range list ❸, which might run up to the end of the `.debug_ranges` section.

The last piece of information that `sdb::range_list` needs is the initial base address, if there is one. Recall that we can find this value in the `DW_AT_low_pc` attribute, if present. As such, if the `DW_AT_low_pc` attribute exists in the root compile unit DIE, we use it; otherwise, we use an empty address ❹. Finally, we return an `sdb::range_list` constructed with the data we just calculated.

To finish `sdb::range_list`, we need to implement `begin`, `end`, and `contains`:

```
sdb::range_list::iterator
sdb::range_list::begin() const {
    return { cu_, data_, base_address_ };
}

sdb::range_list::iterator
sdb::range_list::end() const {
    return {};
}

bool sdb::range_list::contains(file_addr address) const {
    return std::any_of(begin(), end(),
        [=](auto& e) { return e.contains(address); });
}
```

The `begin` function constructs the iterator with the stored data range and base address, whereas `end` constructs an empty iterator. The `contains` function uses the `std::any_of` algorithm to check whether any of the entries in the list contain the given address.

Now that we support range lists, we can add a helper function to `sdb::die` to check contained file addresses and then extend `sdb::die::low_pc()` and `sdb::die::high_pc()` to also handle range lists.

Checking Offsets and Supporting Range Lists

We'll add a member function to `sdb::die`, `contains_address`, to check whether the given file address lies within the ranges for the given DIE. This feature will allow us to find the function containing a given program counter value. We'll also add support for range lists to `low_pc` and `high_pc`, which will be especially useful for setting function breakpoints and dealing with certain compiler optimizations.

Declare `contains_address` in *sdb/include/libsdb/dwarf.hpp*:

```
namespace sdb {
    class die {
    public:
        --snip--
        bool contains_address(file_addr address) const;
        --snip--
    };
}
```

Implement the function in *sdb/src/dwarf.cpp*:

```
bool sdb::die::contains_address(file_addr address) const {
    if (address.elf_file() != this->cu_->dwarf_info()->elf_file()) {
        return false;
    }

    if (contains(DW_AT_ranges)) {
        return (*this)[DW_AT_ranges].as_range_list().contains(address);
    }
    else if (contains(DW_AT_low_pc)) {
        return low_pc() <= address and high_pc() > address;
    }
    return false;
}
```

First, we ensure that the ELF file for the given address is the same as the ELF file to which this DIE belongs. We then check whether the DIE contains a `DW_AT_ranges` attribute. If so, we return an indication of whether the range list contains the given address. If, instead, the DIE contains a `DW_AT_low_pc` attribute, we return an indication of whether the given address lies between that attribute and the high PC attribute.

Following a similar pattern, extend `low_pc` like so:

```
sdb::file_addr sdb::die::low_pc() const {
    if (contains(DW_AT_ranges)) {
        auto first_entry = (*this)[DW_AT_ranges].as_range_list().begin();
        return first_entry->low;
    }
    else if (contains(DW_AT_low_pc)) {
        return (*this)[DW_AT_low_pc].as_address();
    }
    error::send("DIE does not have low PC");
}
```

If the DIE contains a `DW_AT_ranges` attribute, we return the low address of the first range, because address ranges are always in ascending order of

address. If it has a DW_AT_low_pc attribute, we continue to interpret the value as an address. Otherwise, we throw an exception. Make similar changes to high_pc:

```
sdb::file_addr sdb::die::high_pc() const {
    if (contains(DW_AT_ranges)) {
        auto ranges = (*this)[DW_AT_ranges].as_range_list();
        auto it = ranges.begin();
        while (std::next(it) != ranges.end()) ++it;
        return it->high;
    }
    else if (contains(DW_AT_high_pc)) {
        auto attr = (*this)[DW_AT_high_pc];
        file_addr addr;
        if (attr.form() == DW_FORM_addr) {
            return attr.as_address();
        }
        else {
            return low_pc() + attr.as_int();
        }
    }
    error::send("DIE does not have high PC");
}
```

If we encounter a DW_AT_ranges attribute, we get the high address of the highest pair of addresses. To do this, we get an iterator to the first range, increment it until it points to the element before the end iterator (that is, the last element of the list), and return the high range of that pair. If we encounter a DW_AT_high_pc attribute, we do the same as we used to, interpreting the attribute as either an address or an offset from the low program counter. Otherwise, we throw an exception.

The sdb::die type now has a host of utilities to make interacting with its information faster and easier. Let's also add lookup utilities to the main dwarf type.

Augmenting the dwarf Type

Some debugger actions will require querying all available DWARF information at once. For example, to set a breakpoint on a function name or find the function DIE corresponding to the code currently being executed, the debugger may need to examine all function DIEs it has access to. When the amount of debug information available is large, these operations become prohibitively expensive; the debugger might need to parse every single DIE to find what it wants.

To make these operations feasible, we'll add a function index to sdb::dwarf and some query functions to interact with it. This index will record the file

addresses of all function DIEs in the `dwarf` object and map function names to them. Add the following functions to *sdb/include/libsdb/dwarf.hpp*:

```
namespace sdb {
    class dwarf {
    public:
        --snip--
        const compile_unit* compile_unit_containing_address(
            file_addr address) const;
        std::optional<die> function_containing_address(
            file_addr address) const;

        std::vector<die> find_functions(std::string name) const;
        --snip--
    }
}
```

We add three functions: two for retrieving the compile unit or function to which a given file address belongs and one for retrieving the function DIEs for functions that match the given name. Next, we'll add private members for indexing behavior:

```
#include <string>

namespace sdb {
    class dwarf {
        --snip--

    private:
        --snip--
        void index() const;
        void index_die(const die& current) const;
        --snip--
        struct index_entry {
            const compile_unit* cu;
            const std::byte* pos;
        };
        mutable std::unordered_multimap<std::string, index_entry>
        function_index_;
    };
}
```

We add two private member functions: one for indexing the entire set of DIEs in the `dwarf` object and one for indexing a single DIE. Index entries consist of a pointer to the compile unit to which the function belongs (because we'll require this information to parse the DIE later) and a pointer to the beginning of the DIE. The `function_index_` member maps function names to index entries. Because multiple functions can share the same name, we make this member a multimap rather than a one-to-one map. We also mark

it as mutable so that const member functions can modify it. The mutable keyword in C++ allows us to do exactly this: implement caches that don't affect the logical state of the type, but speed up certain operations.

Let's implement these two functions in *sdb/src/dwarf.cpp*, starting with compile_unit_containing_address:

```
const sdb::compile_unit*
sdb::dwarf::compile_unit_containing_address(file_addr address) const {
    for (auto& cu : compile_units_) {
        if (cu->root().contains_address(address)) {
            return cu.get();
        }
    }
    return nullptr;
}
```

We loop over the compile units for this sdb::dwarf object. If we find one whose root DIE contains the given address, we return a pointer to it. If we don't find one, we return a null pointer. Make sure to declare the loop variable as a reference so that return cu.get(); doesn't return a dangling pointer. Next, we'll implement function_containing_address:

```
std::optional<sdb::die>
sdb::dwarf::function_containing_address(file_addr address) const {
    index();
    for (auto& [name, entry] : function_index_) {
        cursor cur({ entry.pos, entry.cu->data().end() });
        auto d = parse_die(*entry.cu, cur);
        if (d.contains_address(address) and
            d.abbrev_entry()->tag == DW_TAG_subprogram) {
            return d;
        }
    }
    return std::nullopt;
}
```

We begin by indexing the DWARF information to ensure that we populate function_index_. (We'll index the DWARF information exactly once in the implementation of index.) We then loop over all entries in the function index, parsing the DIE at the relevant position and determining whether that DIE contains the given address and is a regular function. If the DIE does contain the address, we return the DIE. If we find no relevant entry in the index, we return std::nullopt. This may occur, for example, if the code at that address belongs to some other shared library.

We can now implement the final user-facing function, find_functions:

```
std::vector<sdb::die> sdb::dwarf::find_functions(std::string name) const {
    index();

    std::vector<die> found;
```

```
    auto [begin, end] = function_index_.equal_range(name);
    std::transform(begin, end, std::back_inserter(found), [](auto& pair) {
        auto [name, entry] = pair;
        cursor cur({ entry.pos, entry.cu->data().end() });
        return parse_die(*entry.cu, cur);
    });
    return found;
}
```

Once again, we begin by indexing the DWARF information. We'll return a vector of the DIEs that matched the given name, so we declare an empty vector to begin with. The interface for multimaps in C++ is a bit different from the one for one-to-one maps: instead of using `.at` or `operator[]`, we use `.equal_range` to retrieve a pair of iterators corresponding to the range of entries matching the given key. We then use `std::transform` to call a lambda on every element of this range, pushing the results into the `found` vector. The iterators returned by `std::unordered_multimap` dereference to key-value pairs, so the lambda starts by extracting the function name and index entry from this pair. It then constructs a cursor from the data in the index entry and returns a parsed DIE. Finally, we return the vector of DIEs we just parsed.

With the public members implemented, we can tackle the top-level `index` function:

```
void sdb::dwarf::index() const {
    if (!function_index_.empty()) return;
    for (auto& cu : compile_units_) {
        index_die(cu->root());
    }
}
```

Here, we guard the indexing behavior by checking whether the index has already been populated. If it has, we immediately return. This check ensures that we index the DWARF information only once. We then loop through all of the compile units and index their root DIEs. The `index_die` function will recursively index all children of this DIE.

Indexing a DIE will require retrieving the name of that DIE, however, which can be a little more complicated than just reading the `DW_AT_name` attribute. Let's write a function to handle the complexity.

Retrieving DIE Names

Most function DIEs encode the name of the function as a `DW_AT_name` attribute, but two special types of function encode the name indirectly. DIEs that represent out-of-line definitions (which occur, for example, when we declare a member function in a header file and define it in a source file) contain a `DW_AT_specification` attribute that points to the DIE representing the original declaration. Also, inlined functions (those whose body the compiler has copy-pasted into the body of another function) contain a `DW_AT_abstract_origin` attribute that points to the DIE representing the copied function.

We can handle these cases easily with the existing support we've added to the `sdb::die` type. Declare an `sdb::die::name` function in *sdb/include/libsdb/dwarf.hpp*:

```
namespace sdb {
    class die {
    public:
        --snip--
        std::optional<std::string_view> name() const;
        --snip--
    };
}
```

This function returns the name of the DIE, if there is one. Implement it in *sdb/src/dwarf.cpp*:

```
std::optional<std::string_view> sdb::die::name() const {
    if (contains(DW_AT_name)) {
        return (*this)[DW_AT_name].as_string();
    }
    if (contains(DW_AT_specification)) {
        return (*this)[DW_AT_specification].as_reference().name();
    }
    if (contains(DW_AT_abstract_origin)) {
        return (*this)[DW_AT_abstract_origin].as_reference().name();
    }
    return std::nullopt;
}
```

If the DIE has a `DW_AT_name` attribute, we return that attribute as a string. If it has a `DW_AT_specification` or a `DW_AT_abstract_origin` attribute, we resolve that attribute as a reference to another DIE and then call `.name` on it. We call `.name` rather than just grabbing the `DW_AT_name` attribute on the result to account for chains of references (for example, out-of-line definitions that were inlined). If none of these attributes exist, we return `std::nullopt`. We can now implement DIE indexing.

Indexing DIEs

The `sdb::dwarf::index_die` function should add the given DIE to the function index so long as it has address range data, then recursively index all of that DIE's children. Let's implement it in *sdb/src/dwarf.cpp*:

```
void sdb::dwarf::index_die(const die& current) const {
    bool has_range = current.contains(DW_AT_low_pc) or current.contains(DW_AT_ranges);
    bool is_function = current.abbrev_entry()->tag == DW_TAG_subprogram or
        current.abbrev_entry()->tag == DW_TAG_inlined_subroutine;
    if (has_range and is_function) {
        if (auto name = current.name(); name) {
```

```
            index_entry entry{ current.cu(), current.position() };
            function_index_.emplace(*name, entry);
        }
    }
    for (auto child : current.children()) {
        index_die(child);
    }
}
```

A DIE has an address range if it contains a `DW_AT_low_pc` or a `DW_AT_ranges` attribute. DWARF specifies functions with the `DW_TAG_subprogram` tag or, if the DIE represents a function whose body the compiler has copied from elsewhere, the `DW_TAG_inlined_subroutine` tag. If a DIE has an address range and represents a function, we try to get its name. If it has a name, we create an index entry for this DIE and add it to the index. Regardless of whether this DIE was a function or has address information, we recursively index its children.

We've completed our implementation of DIE parsing. While it took a significant amount of work, you should have learned quite a bit about how the DWARF format represents programs. Before we test this new debugger feature, let's hook the parser into `sdb::elf`. Add an `sdb::dwarf` member in *sdb/include/libsdb/elf.hpp*:

```
namespace sdb {
    class dwarf;
    class elf {
    public:
        --snip--
        dwarf& get_dwarf() { return *dwarf_; }
        const dwarf& get_dwarf() const { return *dwarf_; }

    private:
        --snip--
        std::unique_ptr<dwarf> dwarf_;
    };
}
```

The new member holds a unique pointer to a `dwarf` object. We also add member functions to retrieve it. Initialize this member at the end of the constructor in *sdb/src/elf.cpp*:

```
#include <libsdb/dwarf.hpp>

sdb::elf::elf(const std::filesystem::path& path) {
    --snip--
    dwarf_ = std::make_unique<dwarf>(*this);
}
```

Now let's test the DWARF parser.

Testing the Parser

Exhaustively testing the DWARF parser is a big task, and not one we'll aim to achieve in this section. Instead, we'll focus on writing a small set of tests that should give you a sense of safety as you make changes to your codebase.

Before we begin writing the tests, we must ensure that our compiler outputs DWARF 4 information. To do this, add `-gdwarf-4` to the compiler flags for the C++ test targets in *sdb/test/targets/CMakeLists.txt*:

```
function(add_test_cpp_target name)
    --snip--
    target_compile_options(${name} PRIVATE -g -O0 -pie -gdwarf-4)
    --snip--
endfunction()
```

Now we'll start with a simple test to ensure that we can iterate over compile units and read their metadata. Let's parse the *hello_sdb* test program's DWARF and ensure that the compile unit information we read says that it was written in C++. Add the following to *sdb/test/tests.cpp*:

```
#include <libsdb/dwarf.hpp>

TEST_CASE("Correct DWARF language", "[dwarf]") {
    auto path = "targets/hello_sdb";
    sdb::elf elf(path);
    auto& compile_units = elf.get_dwarf().compile_units();
    REQUIRE(compile_units.size() == 1);

    auto& cu = compile_units[0];
    auto lang = cu->root()[DW_AT_language].as_int();
    REQUIRE(lang == DW_LANG_C_plus_plus);
}
```

We create an `sdb::elf` object for the program and an `sdb::dwarf` object for the ELF file. There should be only one compile unit, so we require that this be the case. We then interpret the `DW_AT_language` attribute of the root compile unit DIE as an integer and ensure that the value corresponds to C++.

Next, we'll make sure that iterating over DIEs produces DIEs whose abbreviation entries have nonzero codes. This test covers both the DIE parsing code and the abbreviation parsing code:

```
TEST_CASE("Iterate DWARF", "[dwarf]") {
    auto path = "targets/hello_sdb";
    sdb::elf elf(path);
    auto& compile_units = elf.get_dwarf().compile_units();
    REQUIRE(compile_units.size() == 1);

    auto& cu = compile_units[0];
    std::size_t count = 0;
```

```
    for (auto& d : cu->root().children()) {
        auto a = d.abbrev_entry();
        REQUIRE(a->code != 0);
        ++count;
    }
    REQUIRE(count > 0);
}
```

We should also test programs that have more than one compile unit. Let's write a simple program with two compile units and check whether our parser can find the DIE for the `main` function. Create an *sdb/test/targets/multi_cu_main.cpp* file with these contents:

```
void do_something();

int main() {
    do_something();
}
```

Also create an *sdb/test/targets/multi_cu_other.cpp* file that defines the `do_something` function:

```
void do_something() {
    // Surprise! It's nothing.
}
```

Modify the *sdb/test/targets/CMakeLists.txt* file to build this target. The `add_test_cpp_target` function doesn't support multiple source files, so we'll specify the compile options manually:

```
--snip--
add_executable(multi_cu multi_cu_main.cpp multi_cu_other.cpp)
target_compile_options(multi_cu PRIVATE -g -O0 -pie -gdwarf-4)
add_dependencies(tests multi_cu)
```

Now, in *sdb/test/tests.cpp*, write a test to find the `main` function DIE of this program:

```
TEST_CASE("Find main", "[dwarf]") {
    auto path = "targets/multi_cu";
    sdb::elf elf(path);
    sdb::dwarf dwarf(elf);

    bool found = false;
    for (auto& cu : dwarf.compile_units()) {
        for (auto& die : cu->root().children()) {
            if (die.abbrev_entry()->tag == DW_TAG_subprogram
                and die.contains(DW_AT_name)) {
                auto name = die[DW_AT_name].as_string();
```

```
                if (name == "main") {
                    found = true;
                }
            }
        }
    }

    REQUIRE(found);
}
```

We parse the DWARF information for the program, then loop through the compile units and the children of the root compile unit DIEs. If we find a DIE that represents a function (`DW_TAG_subprogram`) and has a `DW_AT_name` attribute whose content is `main`, we've found what we're looking for. At the end of the test, we add a `REQUIRES` statement to ensure that we found the correct DIE.

Let's also test range lists. We'll manually create some range list data and ensure that the interpreter correctly deciphers it. We'll begin by parsing DWARF information so that we have a compile unit to pass to the range list constructor. We won't actually use this compile unit, but `sdb::range_list` requires it:

```
TEST_CASE("Range list", "[dwarf]") {
    auto path = "targets/hello_sdb";
    sdb::elf elf(path);
    sdb::dwarf dwarf(elf);
    auto& cu = dwarf.compile_units()[0];
```

Create a range list that consists of a regular entry, a base address selector, another regular entry, and an end-of-list indicator:

```
    --snip--
    std::vector<std::uint64_t> range_data {
        0x12341234, 0x12341236,
        ~0ULL, 0x32,
        0x12341234, 0x12341236,
        0x0, 0x0
    };
```

Cast a pointer to this list's data to a `std::byte*`, and create an `sdb::range_list` with it:

```
    --snip--
    auto bytes = reinterpret_cast<std::byte*>(range_data.data());
    sdb::range_list list(cu.get(),
        { bytes, bytes + range_data.size() }, file_addr{});
```

The first entry should have the low address `0x12341234` and the high address `0x12341236`. This means it should contain `0x12341234` and `0x12341235`, but not `0x12341236`. Let's test all of these conditions:

```
    --snip--
    auto it = list.begin();
    auto e1 = *it;
    REQUIRE(e1.low.addr() == 0x12341234);
    REQUIRE(e1.high.addr() == 0x12341236);
    REQUIRE(e1.contains(file_addr{ elf, 0x12341234 }));
    REQUIRE(e1.contains(file_addr{ elf, 0x12341235 }));
    REQUIRE(!e1.contains(file_addr{ elf, 0x12341236 }));
```

Let's do the same with the second regular entry. The values in this entry are the same as those in the first, but because of the base address selector of `0x32`, they should be offset by `0x32`:

```
    --snip--
    ++it;
    auto e2 = *it;
    REQUIRE(e2.low.addr() == 0x12341266);
    REQUIRE(e2.high.addr() == 0x12341268);
    REQUIRE(e2.contains(file_addr{ elf, 0x12341266 }));
    REQUIRE(e2.contains(file_addr{ elf, 0x12341267 }));
    REQUIRE(!e2.contains(file_addr{ elf, 0x12341268 }));
```

Finally, we can ensure that advancing the iterator again reaches the end of the list and that the `range_list::contains` function works:

```
    --snip--
    ++it;
    REQUIRE(it == list.end());

    REQUIRE(list.contains(file_addr{ elf, 0x12341234 }));
    REQUIRE(list.contains(file_addr{ elf, 0x12341235 }));
    REQUIRE(!list.contains(file_addr{ elf, 0x12341236 }));
    REQUIRE(list.contains(file_addr{ elf, 0x12341266 }));
    REQUIRE(list.contains(file_addr{ elf, 0x12341267 }));
    REQUIRE(!list.contains(file_addr{ elf, 0x12341268 }));
}
```

We've written a small, yet good enough, set of minimal tests. Feel free to expand it yourself if you desire, but in the meantime, ensure that all of these tests pass.

Summary

In this chapter, you learned about the information that the DWARF format encodes and how to decode it. You then wrote a parser for compile unit headers, abbreviation entries, DIEs, and range lists. In the next chapter, you'll learn about the DWARF line table format and implement a parser for it.

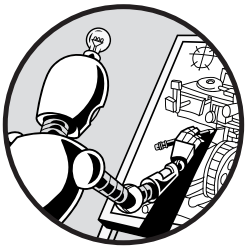
Check Your Knowledge

1. What does DIE stand for?
2. Are new versions of the DWARF format backward compatible with older versions?
3. In which ELF section does the main bulk of the DWARF information live?
4. What is the purpose of abbreviation entries?
5. What is the purpose of the `LEB128` format?
6. What three kinds of entries can be found within a range list?

13

LINE TABLES

Connections through space, dreamed
and then forgotten
but for the feeling.



Some debugger concepts appear very complex at first but later turn out to be simple. Take, for example, reading a general-purpose register from another process, which you can do by calling `ptrace`. Other concepts may seem very simple but are actually complex, such as handling DWARF line tables, which allow you to match machine code instructions to lines of source code.

You might expect line tables to be straightforward binary-encoded matrices that you can easily read into memory. Unfortunately, DWARF uses a domain-specific language designed purely for expressing line tables, so to read them, you must write an interpreter. In this chapter, we'll do exactly that. This interpreter will provide the support we'll need to implement source-level breakpoints in the next chapter.

Line Table Contents

You might be wondering why DWARF defines a line-to-line mapping as a domain-specific language. After all, the information that a line table encodes is fairly straightforward. Here is the start of a line table, represented in a textual format by `dwarfdump`:

```
0x0003386c  [  64, 5] NS uri: "/usr/include/c++/11/bits/exception.h"
0x00033878  [  64,34] NS
0x00033886  [  64,36] NS
```

The first column is the address of the machine instruction to which the mapping corresponds. Between the square brackets are the line number and column of the source code that machine instruction corresponds to. The `NS` specifier means this entry represents the start of a new statement in the source code, and the `uri` specifier defines the file the code belongs to.

The information being encoded isn't all that complex. The problem is that the line table gets really big. For example, if you dump a textual representation of the *sdb* line table, you should notice that it's a bit over 2MB. For a debug build of Clang, the line table is 640MB. This is impractically large, so DWARF compresses the table by expressing it as a binary-encoded program in a language designed specifically for this purpose.

The definition for this program in the DWARF specification consists of two main parts: one for the abstract machine on which the program runs, which we must model in our parser, and one for the language and its encoding, which we must interpret to manipulate the abstract machine and produce the rows of the line table.

Interpreting the Line Table Program

In this section, we'll split up the line table program definitions into smaller chunks and implement them as we go, beginning with the program header.

The Program Header

Like compile units, the line table program begins with a header that contains metadata about the table. This header allows producers to encode high-level information about the table, such as its size and version, and tune the way in which the table is encoded so they can minimize its memory footprint. The line table program header consists of 12 fields:

`unit_length` (`uint32_t`) The byte size of the line number information for this compile unit, not including the `unit_length` field itself.

`version` (`uint16_t`) The version of the line number information. For DWARF 4, this value is `4`.

`header_length` (`uint32_t`) The number of bytes from the end of the `header_length` field until the beginning of the line number program.

`minimum_instruction_length` (`uint8_t`) The byte size of the smallest machine instruction. On x64, this is `1`.

`maximum_operations_per_instruction` (`uint8_t`) The maximum number of operations that may be encoded in an instruction. For architectures that are not *very long instruction word (VLIW)* architectures, this will always be `1`. The x64 architecture is not VLIW, so it always has the value `1` for this field.

`default_is_stmt` (`uint8_t`) Whether rows in the matrix should be interpreted as the beginning of source code statements by default. This allows the producer to save space if most machine instructions are ordered in the same way as the source code statements, which is usually true for unoptimized code.

`line_base` (`int8_t`) The minimum value that special opcodes can add to the `line` register. You'll learn about special opcodes soon.

`line_range` (`uint8_t`) The range of values that special opcodes can add to the `line` register.

`opcode_base` (`uint8_t`) The number assigned to the first special opcode.

`standard_opcode_lengths` (an array of `uint8_t` values) The number of operands that each standard opcode takes. The first element of this array corresponds to the first standard opcode, the second element to the second opcode, and so on. This field allows producers to describe any additional standard opcodes they've used to consumers.

`include_directories` (a sequence of null-terminated strings) The paths that were searched for included files. Each entry is either an absolute path or a path relative to the compilation directory (specified with the `DW_AT_comp_dir` attribute on the root compile unit DIE). The sequence ends with a single null byte.

`file_names` (a sequence of file entries) The source files involved in this compilation. Each entry contains the following: a null-terminated string representing the filename, either as an absolute path, a path relative to the compilation directory, or a path relative to one of the directories specified in the `include_directories` field; a `ULEB128` representing the directory to which this path is relative, if it is a relative path (a value of `0` indicates that the compilation directory contains the path, while a value greater than `0` represents an index into the `include_directories` field, which numbers its entries starting at 1); a `ULEB128` representing the file's last modification time; and a `ULEB128` representing the byte size of the file. A single null byte terminates the sequence of entries.

Let's add a type to store this information and represent the entire line table. We'll call it `sdb::line_table`. Let's start by filling in just the data members. Define it in *sdb/include/libsdb/dwarf.hpp*, above the definitions for `sdb::die` and `sdb::compile_unit`:

```
#include <filesystem>

namespace sdb {
    class line_table {
    public:
     ❶ struct file {
            std::filesystem::path path;
            std::uint64_t modification_time;
            std::uint64_t file_length;
        };

    private:
     ❷ sdb::span<const std::byte> data_;
        const compile_unit* cu_;
        bool default_is_stmt_;
        std::int8_t line_base_;
        std::uint8_t line_range_;
        std::uint8_t opcode_base_;
        std::vector<std::filesystem::path> include_directories_;
     ❸ mutable std::vector<file> file_names_;
    };
}
```

We define a nested type, `sdb::line_table::file`, to represent file entries ❶. For the data members, we'll require a data range (for reading the table) and a pointer back to the compile unit (for retrieving elements such as the compilation directory) ❷. We'll store only a subset of the 12 fields in the line table program header; as we're limiting ourselves to x64 and DWARF 4 and not handling DWARF extensions, we can assume that the `version`, `header_length`, `opcode_base`, `minimum_instruction_length`, `maximum_operations_per_instruction`, and `standard_opcode_lengths` fields are fixed. We also don't need to store `unit_length`, as the `data_` member will record the end of the line table.

You might notice that we do something a bit icky for the `file_names_` member ❸. We want to allow users of our library to get a `const sdb::line_table&` and iterate over its entries. However, the line table program instruction (`DW_LNE_define_file`) adds entries to the `file_names` field as an alternative way of encoding the set of files used in the compilation. Rather than forcing a large part of the debugger library to remove `const` qualifications, we mark the `file_names_` member as `mutable`, which allows it to be modified even when the `line_table` is marked `const`. This is a bit of a hack, but it's a trade-off worth making for our purposes.

Let's move on to the constructor and getters for the compile unit and filenames. The constructor is a bit long and tedious, but it won't write itself. Be sure to define it underneath the definition for the `file` type:

```
namespace sdb {
    class line_table {
    public:
        --snip--

        line_table(sdb::span<const std::byte> data,
            const compile_unit* cu,
            bool default_is_stmt, std::int8_t line_base,
            std::uint8_t line_range, std::uint8_t opcode_base,
            std::vector<std::filesystem::path> include_directories,
            std::vector<file> file_names)
            : data_(data), cu_(cu)
            , default_is_stmt_(default_is_stmt)
            , line_base_(line_base)
            , line_range_(line_range)
            , opcode_base_(opcode_base)
            , include_directories_(std::move(include_directories))
            , file_names_(std::move(file_names))
        {}

        const compile_unit& cu() const { return *cu_; }
        const std::vector<file>& file_names() const { return file_names_; }
        --snip--
    };
}
```

The constructor simply fills in the members with the given arguments. The `sdb::line_table` values should be unique, so we'll disable copy behaviors for them:

```
namespace sdb {
    class line_table {
    public:
        line_table(const line_table&) = delete;
        line_table& operator=(const line_table&) = delete;
        --snip--
    };
}
```

With this class created, we can add a field to `sdb::compile_unit` for holding the line table:

```
namespace sdb {
    class compile_unit {
    public:
        --snip--
```

```
        const line_table& lines() const { return *line_table_; }

    private:
        --snip--
        std::unique_ptr<line_table> line_table_;
    };
}
```

We add a `std::unique_ptr<line_table>` member and a function to retrieve the line table. We'll need to initialize the `line_table_` member in the constructor for `sdb::compile_unit`, which will involve parsing the line table, so we should move the definition of the constructor out of the header file and into the implementation file. Change the current constructor definition to a declaration:

```
namespace sdb {
    class compile_unit {
    public:
        compile_unit(dwarf& parent,
            span<const std::byte> data,
            std::size_t abbrev_offset);
        --snip--
    };
}
```

Now move the old definition into *sdb/src/dwarf.cpp* and add a call to a `parse_line_table` function, which we'll write shortly:

```
sdb::compile_unit::compile_unit(
    dwarf& parent,
    span<const std::byte> data,
    std::size_t abbrev_offset)
    : parent_(&parent)
    , data_(data)
    , abbrev_offset_(abbrev_offset) {
    line_table_ = parse_line_table(*this);
}
```

The only change to the constructor is the call to `parse_line_table` in the function body, which fills in the `line_table_` member. Let's implement this function now, also in *sdb/src/dwarf.cpp*. Because it's quite lengthy, we'll write it a few lines at a time.

Line table programs live in the `.debug_line` section of the object file, and the `DW_AT_stmt_list` attribute of the root compile unit DIE identifies the specific line table program for a given compile unit. This attribute gives an offset into the `.debug_line` section at which the program header begins. Start the implementation by retrieving this information and creating a parsing cursor:

```
namespace {
    std::unique_ptr<sdb::line_table>
    parse_line_table(const sdb::compile_unit& cu) {
        auto section = cu.dwarf_info()->elf_file()->get_section_contents(".debug_line");
        if (!cu.root().contains(DW_AT_stmt_list)) return nullptr;
        auto offset = cu.root()[DW_AT_stmt_list].as_section_offset();
        cursor cur({ section.begin() + offset, section.end() });
```

The next two pieces of data are the program's size and version. Using the size, we can compute the program's end position, which we'll use later. For the version, we'll throw an error if it's something other than the expected value (4):

```
        --snip--
        auto size = cur.u32();
        auto end = cur.position() + size;

        auto version = cur.u16();
        if (version != 4) sdb::error::send("Only DWARF 4 is supported");
```

Some older versions of GCC will output DWARF 3 line table information even when outputting DWARF 4. If your version does this, you can support version 3 by changing the version number check you just wrote and skipping over the `maximum_operations_per_instruction` field in the next listing.

Now we can continue parsing fields:

```
        --snip--
        (void)cur.u32();  // Header length

        auto minimum_instruction_length = cur.u8();
        if (minimum_instruction_length != 1)
            sdb::error::send("Invalid minimum instruction length");

        auto maximum_operations_per_instruction = cur.u8();
        if (maximum_operations_per_instruction != 1)
            sdb::error::send("Invalid maximum operations per instruction");
```

We don't really care about the header length, so we parse it and throw away the result. The cast to `void` is to avoid compiler warnings. For the minimum instruction length and maximum operations per instruction, we expect both values to be `1`, so we throw an error if that's not the case. We then parse four more fields:

```
        --snip--
        auto default_is_stmt = cur.u8();
        auto line_base = cur.s8();
        auto line_range = cur.u8();
        auto opcode_base = cur.u8();
```

We'll need these fields to interpret the line table program, so we save them for later.

Next are the standard opcode lengths. We won't support DWARF extensions, but we will support producers that decide not to use all of the standard opcodes. Line table program opcodes get encoded as `uint8_t` values. Each standard opcode is assigned a number, beginning at 1 and incrementing. DWARF 4 has 12 standard opcodes, which we'll look at in "Standard Opcodes" on page 356.

Immediately after the standard opcodes comes `opcode_base`, the number of the first special opcode. For example, if the producer used all the standard opcodes in this line table program, `opcode_base` would be 13, which is the highest possible value for `opcode_base`. The rest of the special opcodes use the numbers ranging from `opcode_base` to 255, the highest number a `uint8_t` can represent. A producer can therefore choose not to use some of the standard opcodes, lowering `opcode_base` and giving more encoding space to special opcodes. Producers may make this trade-off if they decide that having more special opcodes would reduce the space required to encode the line table.

To parse the standard opcode lengths, we'll read a number of values equal to `opcode_base - 1` and then ensure that these lengths match the opcode lengths specified in the DWARF standard (you'll find out exactly what standard opcodes do when we move on to parsing the line table program itself):

```
--snip--
std::array<std::uint8_t, 12> expected_opcode_lengths {
    0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1
};
for (auto i = 0; i < opcode_base - 1; ++i) {
    if (cur.u8() != expected_opcode_lengths[i]) {
        sdb::error::send("Unexpected opcode length");
    }
}
```

If `opcode_base` is less than 13, the producer hasn't used all the standard opcodes, so we won't check the matching elements of `expected_opcode_lengths`. We then parse the include directories:

```
--snip--
std::vector<std::filesystem::path> include_directories;
std::filesystem::path compilation_dir (cu.root()[DW_AT_comp_dir].as_string()); ❶
for (auto dir = cur.string();
    !dir.empty();
    dir = cur.string()) {
    if (dir[0] == '/') { ❷
        include_directories.push_back(std::string(dir));
    }
    else { ❸
```

```
            include_directories.push_back(compilation_dir / std::string(dir));
        }
    }
```

Recall that include directories can be either absolute paths or paths relative to the compilation directory for this compile unit. We retrieve this compilation directory by reading the `DW_AT_comp_dir` attribute of the root compile unit DIE ❶. We loop until we read an empty string (a single null byte). If the directory name begins with a `/`, it's an absolute path, so we store the path as is ❷. Otherwise, we first prefix the path with the compilation directory ❸.

The last field contains the filename entries. We may also need to parse these entries from `DW_LNE_define_file` instructions, so we'll factor this work into a `parse_line_table_file` function:

```
        --snip--
        std::vector<sdb::line_table::file> file_names;
        while (*cur.position() != std::byte(0)) {
            file_names.push_back(
                parse_line_table_file(
                    cur, compilation_dir, include_directories));
        }
        cur += 1;
```

We keep reading file entries until we read a null byte. Once we're done, we increment the cursor to move past the null byte, bringing the cursor to the beginning of the line table program itself. We'll now construct the return value:

```
        --snip--
        sdb::span<const std::byte> data {cur.position(), end};
        return std::make_unique<sdb::line_table>(data, &cu,
            default_is_stmt,
            line_base, line_range, opcode_base,
            std::move(include_directories), std::move(file_names));
    }
}
```

First, we construct the data span. This span will range from the current cursor position to the end of the line table program we saved at the top of the function. Then, we create a `std::unique_ptr<sdb::line_table>` with all of the data we just parsed and return it.

We've wrapped up the long constructor function. Before we move on, let's implement `parse_line_table_file`. This function should parse the filepath, parent directory index, modification time, and file length, then make any relative paths absolute:

```
namespace {
    sdb::line_table::file parse_line_table_file(cursor& cur,
          std::filesystem::path compilation_dir,
```

```
          const std::vector<std::filesystem::path>& include_directories) {
      ❶ auto file = cur.string();
        auto dir_index = cur.uleb128();
        auto modification_time = cur.uleb128();
        auto file_length = cur.uleb128();

        std::filesystem::path path = file;
      ❷ if (file[0] != '/') {
          ❸ if (dir_index == 0) {
                path = compilation_dir / std::string(file);
            }
          ❹ else {
                path = include_directories[dir_index - 1] / std::string(file);
            }
        }
      ❺ return {path.string(), modification_time, file_length};
    }
}
```

We parse the data in the entry ❶ before fixing the paths. Recall that relative paths are relative to the compilation directory if `dir_index` is `0`; otherwise, they're relative to the `include_directories` entry at the specified index, beginning with 1. We first check whether the path is relative by checking that it doesn't begin with a slash (`/`) ❷, then handle the special cases of being relative to the compilation directory ❸ and being relative to an include directory ❹. Be sure to subtract 1 from `dir_index` when indexing `include_directories`, as the indices begin at 1 rather than 0. Finally, we return the entry ❺.

We've now successfully parsed the line table program header and can move on to the abstract machine on which the line table program runs.

The Abstract Machine

The line table's abstract machine includes storage for a single row of the line table. For example, it contains fields for the program counter value, the source line and column, and the source file. The specification calls these fields *registers*, as an analogy to CPU registers. The line table program's instructions manipulate these registers, perhaps by incrementing the current source line or changing the source file involved in the entry. Putting the registers into the correct state may require several instructions, so the registers won't necessarily hold a complete line; to do so, the program will use certain specific instructions. We'll look at an example of such an instruction in the next section.

Here are the registers the abstract machine stores, along with their uses and the values to which they're initialized at the start of the interpretation process:

address The program counter value, which marks the start of the machine instruction for which a mapping is being provided. Initialized to `0`.

op_index The index of an operation within a VLIW instruction. For non-VLIW architectures, this will always be 0. Initialized to `0`.

file The source file that stores the code for the mapped address; represented as an index into the `file_names` field of the line table program header. Initialized to `1`.

line The source line number. The abstract machine numbers lines beginning at 1 and uses 0 for instructions that don't map to a source line. Initialized to `1`.

column The column number within a source line. The abstract machine numbers columns beginning at 1. Initialized to `0`.

is_stmt Whether this instruction marks the beginning of a statement, making it a recommended location for a breakpoint. The initialization value is determined by the `default_is_stmt` field in the line number program header.

basic_block Whether this instruction marks the beginning of a *basic block*, which is a series of instructions with no branches. Initialized to `false`.

end_sequence Whether this entry is special and marks the byte immediately following a sequence of instructions. Initialized to `false`.

prologue_end Whether this instruction marks the end of the function prologue and thus should be used for function entry breakpoints. Initialized to `false`.

epilogue_begin Whether this instruction marks the beginning of the function epilogue and thus should be used for function exit breakpoints. Initialized to `false`.

isa The instruction set used for this instruction, as defined by the ABI in use. Because we're assuming x64, we can ignore this field. Initialized to `0`. Commonly used on ARM processors that can also use the smaller Thumb instruction set.

discriminator An identifier to give performance profilers information about which basic block a given instruction belongs to. We won't need to do anything with this field in the debugger. Initialized to `0`.

Let's write a type that stores all of this data, called `sdb::line_table::entry`. Define it in *sdb/include/libsdb/dwarf.hpp*:

```
namespace sdb {
    class line_table {
    public:
        struct entry;
        --snip--
```

```
    };

    struct line_table::entry {
        file_addr address;
        std::uint64_t file_index = 1;
        std::uint64_t line = 1;
        std::uint64_t column = 0;
        bool is_stmt;
        bool basic_block_start = false;
        bool end_sequence = false;
        bool prologue_end = false;
        bool epilogue_begin = false;
        std::uint64_t discriminator = 0;
        file* file_entry = nullptr;
    };
}
```

We declare the `entry` type inside the definition of `sdb::line_table` and then define `sdb::line_table::entry` below that definition to keep the code organized, as the definition is a bit long. The type's members should match the fields I previously described. We add a `file_entry` member in which to store a pointer to the actual file data for this entry, as this information is much more useful to users than the `file_index` member. We initialize all but the `is_stmt` and `address` members. We'll need to fill in the former with information based on the `default_is_stmt` field of the line table program header, and we set the latter to `0` automatically.

Let's also add a way to compare the equality of two entries:

```
    struct line_table::entry {
        --snip--
        bool operator==(const entry& rhs) const {
            return address == rhs.address and
                file_index == rhs.file_index and
                line == rhs.line and
                column == rhs.column and
                discriminator == rhs.discriminator;
        }
    };
```

We don't need to check every member of the entry; the address, file index, line, column, and discriminator will do. In a well-formed line table, no entries should have equal values for those members and unequal values for the other members.

As when parsing DIEs and range lists, we'll use an iterator type as the main parsing engine here. Let's add one to `sdb::line_table`:

```
namespace sdb {
    class line_table {
    public:
```

```
        class iterator;
        iterator begin() const;
        iterator end() const;
        --snip--
    };
}
```

We declare the `iterator` type in the `sdb::line_table` definition. The type will require the following members:

- A pointer to the `sdb::line_table` to which it belongs
- An `sdb::line_table::entry` for the entire matrix entry to which it's currently pointing
- An `sdb::line_table::entry` for the current state of the abstract machine registers
- A byte position inside the `.debug_line` section pointing to the data to parse next

Here is the type's definition:

```
namespace sdb {
    class line_table::iterator {
    public:
        using value_type = entry; ❶
        using pointer = const entry*;
        using reference = const entry&;
        using difference_type = std::ptrdiff_t;
        using iterator_category = std::forward_iterator_tag;

        iterator(const line_table* table_); ❷

        iterator() = default; ❸
        iterator(const iterator&) = default;
        iterator& operator=(const iterator&) = default;

        const line_table::entry& operator*() const { return current_; } ❹
        const line_table::entry* operator->() const { return &current_; }

        bool operator==(const iterator& rhs) const { return pos_ == rhs.pos_; } ❺
        bool operator!=(const iterator& rhs) const { return pos_ != rhs.pos_; }

        iterator& operator++(); ❻
        iterator operator++(int);

    private:
        const line_table* table_; ❼
        line_table::entry current_;
        line_table::entry registers_;
```

```
        const std::byte* pos_;
    };
}
```

As in the iterators we wrote in the previous chapter, we expose information about the iterator type through type aliases ❶. Again, we use `std::ptrdiff_t` as a default value for `difference_type` and `std::forward_iterator_tag` for the iterator category, because we're defining a multipass iterator. The constructor will need to parse the first entry of the logical line table matrix, so we merely declare it for now ❷. The default operations for default construction, copy construction, and copy assignment will all work fine ❸. When dereferenced, the iterator should return the current matrix entry ❹. We define iterator equality in terms of the current byte position to which the iterator is pointing ❺. We'll handle the bulk of the parsing in `operator++`, so we simply declare it and the post-increment operator for now ❻. Finally, we declare the data members ❼.

We must still define a few members. Let's begin implementing the iterator constructor in *sdb/src/dwarf.cpp*:

```
sdb::line_table::iterator::iterator(const sdb::line_table* table)
     : table_(table), pos_(table->data_.begin()) {
    registers_.is_stmt = table->default_is_stmt_;
    ++(*this);
}
```

We initialize the table pointer with the one we're given and set the current position to the beginning of the table. Given the table, we can now initialize `registers_.is_stmt` based on the `default_is_stmt` field of the program header. We then call the increment operator on the iterator to parse the matrix's first row.

We'll implement `operator++` in the next section, where we discuss the line table program itself. For now, we can implement `sdb::line_table::begin` and `end`:

```
sdb::line_table::iterator
sdb::line_table::begin() const {
    return iterator(this);
}
sdb::line_table::iterator
sdb::line_table::end() const {
    return {};
}
```

These functions simply call iterator constructors that take a pointer to an `sdb::line_table` and the default constructor, respectively. Now we can move on to the line table program's instructions.

The Program Instructions

Following the header are the instructions that make up the line table program. Each instruction belongs to one of three categories:

Standard opcode A `uint8_t` opcode that names an operation, such as `DW_LNS_advance_line`, for advancing the current line by a given amount, or `DW_LNS_set_basic_block`, for setting the `basic_block` register to `true`. The number and type of operands that a standard opcode takes depends on which opcode it is. For example, `DW_LNS_advance_line` takes one operand indicating the number of lines to advance by, whereas `DW_LNS_set_basic_block` takes no operands.

Extended opcode Used to encode more complex instructions. Extended opcodes begin with a null byte, followed by a `ULEB128` giving the size of the next instruction. After this size comes a `uint8_t` providing the extended opcode and then the operands.

Special opcode Used to advance the current line and address and emit a matrix row, all in a single opcode. Special opcodes consist of a single `uint8_t` with no operands.

To implement the interpreter for these instructions, we'll first write a skeleton of `operator++`, which advances the line table iterator, and then go through each of these categories in turn to add the necessary support.

Incrementing the iterator should run the line table program until it reaches an instruction that emits a new matrix row. This means we may need to run an arbitrary number of instructions before returning from `operator++`. To handle this, we'll write an `execute_instruction` function that executes a single line table program instruction and returns an indication of whether that instruction should emit a new row of the matrix. Add a declaration for a private member to `sdb::line_table::iterator` in *sdb/include/libsdb/dwarf.hpp*:

```
namespace sdb {
    class line_table::iterator {
        --snip--
    private:
        bool execute_instruction();
        --snip--
    };
}
```

Implement `operator++` in terms of this function in *sdb/src/dwarf.cpp*:

```
sdb::line_table::iterator&
sdb::line_table::iterator::operator++() {
    if (pos_ == table_->data_.end()) {
        pos_ = nullptr;
        return *this;
    }
```

```
    bool emitted = false;
    do {
        emitted = execute_instruction();
    } while (!emitted);

    current_.file_entry = &table_->file_names_[current_.file_index-1];
    return *this;
}
```

We begin by checking whether we've reached the end of the table. If so, we set the current position to `nullptr` to signal this fact and return. Otherwise, we continue executing instructions until one of them emits a new row of the matrix. At this point, `current_` should hold the row data. We then resolve the `file_entry` member, filling it in with a pointer to the correct entry in the table's `file_names_` field based on the `file_index` member of the matrix row. Remember to subtract 1 from this index, as file indices begin at 1 rather than 0. Finally, we return `*this`, which is idiomatic behavior for `operator++`.

As we did with DIE child iterators and range list iterators, we implement the post-increment operator in terms of the pre-increment one:

```
sdb::line_table::iterator
sdb::line_table::iterator::operator++(int) {
    auto tmp = *this;
    ++(*this);
    return tmp;
}
```

We make a copy of the current iterator, increment `*this`, and return the unmodified copy. Having completed the interpreter's main structure, we can move on to implementing `execute_instruction`, starting with standard opcodes.

Standard Opcodes

DWARF 4 specifies 12 standard opcodes, each of which has a specific meaning. Here are the opcodes, their operands, and what they signify:

`DW_LNS_copy` (no operands) Emits a matrix row using the current values of the abstract machine registers and then resets the `discriminator` register to `0` and the `basic_block`, `prologue_end`, and `epilogue_begin` registers to `false`.

`DW_LNS_advance_pc` (one `ULEB128` operand) Adds the operand to the `address` register.

`DW_LNS_advance_line` (one `SLEB128` operand) Adds the operand to the `line` register.

`DW_LNS_set_file` (one `ULEB128` operand) Stores the operand in the `file` register.

`DW_LNS_set_column` (one `ULEB128` operand) Stores the operand in the `column` register.

`DW_LNS_negate_stmt` (no operands) Sets the `is_stmt` register to the negation of its current value.

`DW_LNS_set_basic_block` (no operands) Sets the `basic_block` register to `true`.

`DW_LNS_const_add_pc` (no operands) Advances the `address` register by the same amount as special opcode 255 would. We'll cover exactly how special opcodes work later, but for now, you can trust that the formula is `(255 - opcode_base) / line_range`.

`DW_LNS_fixed_advance_pc` (one `uint16_t` operand) Adds the operand to the `address` register. This standard opcode is the only one that takes a fixed-sized operand. It exists to support assemblers that don't support `LEB128` encoding.

`DW_LNS_set_prologue_end` (no operands) Sets the `prologue_end` register to `true`.

`DW_LNS_set_epilogue_begin` (no operands) Sets the `epilogue_begin` register to `true`.

`DW_LNS_set_isa` (one `ULEB128` operand) Stores the operand in the `isa` register.

The `execute_instruction` function's implementation is more or less a straight translation of these opcodes into a `switch` statement, with some bookkeeping at either end. The function's structure looks like this:

```
bool sdb::line_table::iterator::execute_instruction() {
    auto elf = table_->cu_->dwarf_info()->elf_file();
    cursor cur({ pos_, table_->data_.end() });
    auto opcode = cur.u8();
    bool emitted = false;

    if (opcode > 0 and opcode < table_->opcode_base_) {
        // Handle standard opcode
    }

    pos_ = cur.position();
    return emitted;
}
```

We grab a pointer to the ELF file for this line table, create a cursor for the relevant data range, and read the opcode from the current position. Most instructions don't emit matrix rows, so we make a variable called `emitted` and default it to `false`. Standard opcodes are between 1 and `opcode_base` − 1, so we check for this condition. Finally, we update the current position to the location at which the cursor ended up and return whether we emitted a matrix row.

Now comes the big ol' `switch` statement inside of the `if` block:

```
--snip--
if (opcode > 0 and opcode < table_->opcode_base_) {
    switch (opcode) {
    case DW_LNS_copy:
        current_ = registers_;
        registers_.basic_block_start = false;
        registers_.prologue_end = false;
        registers_.epilogue_begin = false;
        registers_.discriminator = 0;
     ❶ emitted = true;
        break;
    case DW_LNS_advance_pc:
        registers_.address += cur.uleb128();
        break;
    case DW_LNS_advance_line:
        registers_.line += cur.sleb128();
        break;
    case DW_LNS_set_file:
        registers_.file_index = cur.uleb128();
        break;
    case DW_LNS_set_column:
        registers_.column = cur.uleb128();
        break;
    case DW_LNS_negate_stmt:
        registers_.is_stmt = !registers_.is_stmt;
        break;
    case DW_LNS_set_basic_block:
        registers_.basic_block_start = true;
        break;
    case DW_LNS_const_add_pc:
        registers_.address +=
            (255 - table_->opcode_base_) / table_->line_range_;
        break;
    case DW_LNS_fixed_advance_pc:
        registers_.address += cur.u16();
        break;
    case DW_LNS_set_prologue_end:
        registers_.prologue_end = true;
        break;
    case DW_LNS_set_epilogue_begin:
        registers_.epilogue_begin = true;
        break;
    case DW_LNS_set_isa:
     ❷ break;
    default:
     ❸ error::send("Unexpected standard opcode");
```

```
        }
    }
    --snip--
```

This code matches the textual description of the opcodes shown earlier as closely as possible. Note that `DW_LNS_copy` is the only standard opcode that emits a matrix row, so it sets `emitted` to `true` ❶, and because we ignore the `isa` register, `DW_LNS_set_isa` does nothing ❷. We also issue an error if we get an unexpected opcode ❸.

We've handled all of the standard opcodes. The extended opcodes will be easier to implement and require less code, even if they're a bit more difficult to understand.

Extended Opcodes

We use extended opcodes for instructions that require more space to encode or that don't occur often enough to merit taking up one of the 255 slots allocated to standard or special opcodes.

Recall that extended opcodes begin with a 0 byte, followed by a `ULEB128` value that indicates the length of the instruction and a byte containing the actual extended opcode. Much like the standard opcodes, extended opcodes each have a specific meaning and set of operands. Here are their names, operands, and descriptions:

`DW_LNE_end_sequence` (no operands) Sets the `end_sequence` register to `true` and emits a matrix row using the current values of the abstract machine registers, then resets the registers to their default values. Every line table program ends with an instruction of this type.

`DW_LNE_set_address` (one `uint64_t` operand) Stores the operand in the `address` register.

`DW_LNE_define_file` (one file entry) Appends a file entry to the `file_names` line table program header field. This file entry has the same format as the `file_names` line table program header field.

`DW_LNE_set_discriminator` (one `ULEB128` operand) Stores the operand in the `discriminator` register.

Much like the standard opcodes, we'll implement extended opcodes in a `switch` statement:

```
bool sdb::line_table::iterator::execute_instruction() {
    --snip--
    if (opcode > 0 and opcode < table_->opcode_base_) {
        --snip--
    }
    else if (opcode == 0) { ❶
        auto length = cur.uleb128();
        auto extended_opcode = cur.u8();

        switch (extended_opcode) {
```

```
        case DW_LNE_end_sequence: ❷
            registers_.end_sequence = true;
            current_ = registers_;
            registers_ = entry{};
            registers_.is_stmt = table_->default_is_stmt_;
            emitted = true;
            break;
        case DW_LNE_set_address: ❸
            registers_.address = file_addr(
                *elf, cur.u64());
            break;
        case DW_LNE_define_file: { ❹
            auto compilation_dir =
                table_->cu_->root()[DW_AT_comp_dir].as_string();
            auto file = parse_line_table_file(
                cur, std::string(compilation_dir), table_->include_directories_);
            table_->file_names_.push_back(file);
            break;
        }
        case DW_LNE_set_discriminator: ❺
            registers_.discriminator = cur.uleb128();
            break;
        default:
            error::send("Unexpected extended opcode");
        }
    }
    --snip--
}
```

Some of these conditions are more involved than those for the standard opcodes, so we'll walk through them in more detail. First, extended opcodes begin with a 0 byte, so if the original opcode we read is 0, we know we're looking at an extended opcode ❶. We start by reading the length of the instruction, followed by the extended opcode itself. We won't actually use the length, but it could be useful for skipping over extended opcodes that a given DWARF producer has added as a vendor extension, to give one example.

The instructions for setting the address ❸ and discriminator ❺ are very simple; we read the relevant type with our cursor and store the result in the correct register. For `DW_LNE_end_sequence` ❷, we reset the current registers by assigning a new `entry` to them. We must also be sure to set the `is_stmt` field based on the default value from the line table program header and then set `emitted` to `true` to signal that a new matrix row is available.

For `DW_LNE_define_file` ❹, we need to first grab the compilation directory from the root compile unit DIE, then call out to the `parse_line_table_file` function we wrote earlier before pushing the result into the `file_names_` member. Now we can move on to special opcodes.

Special Opcodes

Special opcodes are strange. If you read the DWARF specification, you should find that it clearly explains almost all of the relevant concepts, with the exception of special opcodes. As such, I'll introduce them in a roundabout way that should hopefully help you understand them in a fraction of the time it took me.

Line tables are generally most useful for unoptimized code, as they can match the source code lines and machine code lines without too much difficulty. In this situation, each subsequent row of the line table matrix is usually similar to the previous line. The address may have gone up a bit, and the line number has likely advanced slightly, but it might also have gone a couple of lines back (for example, in the case of `if` statements with conditions split on multiple lines).

Because we're working with so much data, we can make huge storage optimizations by finding ways to encode the most common cases in the least amount of space. This is the aim of special opcodes. Every special opcode will adjust the current address and line by a small amount, reset a few flags, and emit a new row of the matrix, all within a single byte of data. Most importantly, the special opcode indicates how much to adjust the address and line by.

Special opcodes comes with a few nuances. First, a DWARF producer must specify a range of line adjustments that can be expressed as special opcodes. For example, in a program where the line numbers in the matrix generally go up or down by just two or three lines at a time, it could pick a range of −3 to 3. For one where the jumps are often larger and the largest jumps tend to go forward, it might pick a range of −4 to 10.

Once that producer has made that decision, we can construct a table whose rows describe how much to adjust the line number by and whose columns specify how much to adjust the address by. We number each cell of this table incrementally. See Table 13-1 for an example.

Table 13-1: Modeling the Range –3 to 3

	Line advance						
Address advance	**–3**	**–2**	**–1**	**0**	**1**	**2**	**3**
0	0	1	2	3	4	5	6
1	7	8	9	10	11	12	13
2	14	15	16	17	18	19	20
3	21	22	23	24	25	26	27
4	28	29	30	31	32	33	34
. . .				. . .			

Now imagine that instead of numbering those cells starting from 0 with no fixed end, we number them from `operand_base` to 255, the maximum value representable in a `uint8_t`, as in Table 13-2.

Table 13-2: A Special Opcode Table for the Range –3 to 3

	Line advance						
Address advance	**–3**	**–2**	**–1**	**0**	**1**	**2**	**3**
0	13	14	15	16	17	18	19
1	20	21	22	23	24	25	26
2	27	28	29	30	31	32	33
3	34	35	36	37	38	39	40
4	41	42	43	44	45	46	47
5	48	49	50	51	52	53	54
6	55	56	57	58	59	60	61
7	62	63	64	65	66	67	68
8	69	70	71	72	73	74	75
9	76	77	78	79	80	81	82
10	83	84	85	86	87	88	89
11	90	91	92	93	94	95	96
12	97	98	99	100	101	102	103
13	104	105	106	107	108	109	110
14	111	112	113	114	115	116	117
15	118	119	120	121	122	123	124
16	125	126	127	128	129	130	131
17	132	133	134	135	136	137	138
18	139	140	141	142	143	144	145
19	146	147	148	149	150	151	152
20	153	154	155	156	157	158	159
21	160	161	162	163	164	165	166
22	167	168	169	170	171	172	173
23	174	175	176	177	178	179	180
24	181	182	183	184	185	186	187
25	188	189	190	191	192	193	194
26	195	196	197	198	199	200	201
27	202	203	204	205	206	207	208
28	209	210	211	212	213	214	215
29	216	217	218	219	220	221	222
30	223	224	225	226	227	228	229
31	230	231	232	233	234	235	236
32	237	238	239	240	241	242	243
33	244	245	246	247	248	249	250
34	251	252	253	254	255	—	—

Special opcodes are those cell values. For example, in Table 13-2, special opcode 13 advances the address by 0 and the line by –3. Special opcode 106 advances the address by 13 and the line by –1. We encode the range for the

special opcode table with the `line_base` and `line_range` fields in the line table program header. In this case, `line_base` would be `-3` and `line_range` would be `7` because there are seven columns in the table.

With the theory out of the way, we can define the implementation of special opcodes in the interpreter. Most of the work involves performing a bit of math to calculate the line and address advances from a given cell of the table. Where the `adjusted_opcode` is `opcode - opcode_base`, each special opcode will do the following:

1. Add `adjusted_opcode / line_range` to the `address` register.
2. Add `line_base + (adjusted_opcode % line_range)` to the `line` register.
3. Emit a row of the matrix using the current values of the abstract machine register.
4. Set the `basic_block` register to `false`.
5. Set the `prologue_end` register to `false`.
6. Set the `epilogue_begin` register to `false`.
7. Set the `discriminator` register to `0`.

Let's write the code to perform these operations:

```
bool sdb::line_table::iterator::execute_instruction() {
    --snip--
    if (opcode > 0 and opcode < table_->opcode_base_) {
        --snip--
    }
    else if (opcode == 0) {
        --snip--
    }
    else {
        auto adjusted_opcode = opcode - table_->opcode_base_;
        registers_.address += adjusted_opcode / table_->line_range_;
        registers_.line +=
            table_->line_base_ + (adjusted_opcode % table_->line_range_);
        current_ = registers_;
        registers_.basic_block_start = false;
        registers_.prologue_end = false;
        registers_.epilogue_begin = false;
        registers_.discriminator = 0;
     ❶ emitted = true;
    }
    --snip--
}
```

This code is a direct translation of the algorithm I just described. Remember to set `emitted` to `true` ❶ to signal that a new matrix row is available. Congratulations! You've written an entire interpreter for the DWARF line table program language. Certain lookup tasks will be quite tedious with

the current interface, however, so let's add functions to make them more ergonomic.

Retrieving Entries by Line or File Address

We'll need the ability to find the line table entry that corresponds to a given source code line or program counter value when implementing source-level stepping and breakpoints in Chapter 14. Let's add functions for these tasks to `sdb::line_table`. First, add the declarations to *sdb/include/libsdb/dwarf.hpp*:

```
namespace sdb {
    class line_table {
    public:
        --snip--
        iterator get_entry_by_address(file_addr address) const;
        std::vector<iterator> get_entries_by_line(
            std::filesystem::path path, std::size_t line) const;
    };
}
```

The `get_entry_by_address` function takes a file address and returns an iterator to the line table entry that corresponds to that address. We return an iterator rather than the entry itself because user code may want to examine nearby entries to, for example, skip over function prologues while stepping.

The `get_entries_by_line` function returns a vector of iterators because a single line of source code may correspond to multiple line table entries. This could happen, for instance, if multiple function calls occur within a single expression, as in `pet_cat(get_cat("Marshmallow"));`. Let's implement these functions in *sdb/src/dwarf.cpp*, starting with `get_entry_by_address`:

```
sdb::line_table::iterator
sdb::line_table::get_entry_by_address(file_addr address) const {
    auto prev = begin();
 ❶ if (prev == end()) return prev;

    auto it = prev;
 ❷ for (++it; it != end(); prev = it++) {
        if (prev->address <= address and
            it->address > address and
         ❸ !prev->end_sequence) {
            return prev;
        }
    }
    return end();
}
```

We can't simply look for the entry whose address is equal to the given one because the given address may lie between two entries. Instead, we look

for the first entry whose address is less than or equal to the given one, and where the subsequent entry's address is greater than the given one.

We begin by checking whether the line table is empty, in which case we return the end iterator ❶. We then loop over the entries in the table ❷, returning an entry if it's the one containing the given address. Note that we make sure we don't return entries marked as the end of a sequence, as those aren't true entries in the table ❸.

Let's next consider `get_entries_by_line`, as it's a little trickier. We want to support both absolute paths and subpaths, which will make setting breakpoints on lines of source code fairly user-friendly. For example, if a user wants to set a breakpoint at line 42 of */home/marshmallow/find_treats.cpp*, they should be able to enter `find_treats.cpp:42`, `marshmallow/find_treats.cpp:42`, `home/marshmallow/find_treats.cpp:42`, or `/home/marshmallow/find_treats.cpp:42`.

First, we'll implement a `path_ends_with` helper function that returns an indication of whether a path ends in another path. Write it in *sdb/src/dwarf.cpp*:

```
namespace {
    bool path_ends_in(const std::filesystem::path& lhs,
                      const std::filesystem::path& rhs) {
        auto lhs_size = std::distance(lhs.begin(), lhs.end());
        auto rhs_size = std::distance(rhs.begin(), rhs.end());
        if (rhs_size > lhs_size) return false;
        auto start = std::next(lhs.begin(), lhs_size - rhs_size);
        return std::equal(start, lhs.end(), rhs.begin());
    }
}
```

We begin by computing the number of elements in each path (that is, the number of directories, plus one for the filename). If the second argument has more elements than the first, then it cannot be a suffix of the first, so we return `false`. Otherwise, we calculate the element in the first path at which we should start comparing the two paths. This element is the one at an offset of `lhs_size - rhs_size` from the start of the path. Finally, we use `std::equal` to check if the subpath from `start` to `lhs.end()` matches `rhs`.

Now we can implement `get_entries_by_line` in *sdb/src/dwarf.cpp*:

```
std::vector<sdb::line_table::iterator>
sdb::line_table::get_entries_by_line(
      std::filesystem::path path, std::size_t line) const {
    std::vector<iterator> entries;

    for (auto it = begin(); it != end(); ++it) {
        auto& entry_path = it->file_entry->path;
        if (it->line == line) {
            if ((path.is_absolute() and entry_path == path) or
                (path.is_relative() and path_ends_in(entry_path, path))) {
                entries.push_back(it);
            }
```

```
        }
    }

    return entries;
}
```

We start by creating an empty vector of iterators that we'll populate as we iterate over the entries in the table. We loop over the entries. If an entry's line matches the one we're looking for, then we examine the entry's path. We push the entry's iterator to the results vector in two cases: if the given path is absolute and matches the entry's path, or if the given path is relative and is a suffix of the entry's path. Finally, we return the entries we found.

These functions should make it much easier to look up entries. Let's also add some DIE attribute helpers to return line information.

DIE Line Attribute Helpers

DIEs may contain file and line information. However, the attributes used to encode this information differ depending on whether the function is inlined. (We'll discuss inlining in more detail in the next chapter.) Let's add helper functions to ease the handling of these special cases.

Add `file` and `line` functions to `sdb::die` in *sdb/include/libsdb/dwarf.hpp*. Also add a type and member function that wraps these values in a single call, which will come in handy for stack unwinding in Chapter 16:

```
namespace sdb {
    struct source_location {
        const line_table::file* file;
        std::uint64_t line;
    };
    class die {
    public:
        --snip--
        source_location location() const;
        const line_table::file& file() const;
        std::uint64_t line() const;
        --snip--
    };
}
```

Ensure that you define `sdb::line_table` before `sdb::die`, or this code won't compile due to the direction of the dependency. Implement these functions in *sdb/src/dwarf.cpp*:

```
sdb::source_location
sdb::die::location() const {
    return { &file(), line() };
}
```

```
const sdb::line_table::file&
sdb::die::file() const {
    std::uint64_t idx;
    if (abbrev_->tag == DW_TAG_inlined_subroutine) {
        idx = (*this)[DW_AT_call_file].as_int();
    }
    else {
        idx = (*this)[DW_AT_decl_file].as_int();
    }
    return this->cu_->lines().file_names()[idx-1];
}

std::uint64_t sdb::die::line() const {
    if (abbrev_->tag == DW_TAG_inlined_subroutine) {
        return (*this)[DW_AT_call_line].as_int();
    }
    return (*this)[DW_AT_decl_line].as_int();
}
```

The line table encodes the file for a DIE as a 1-based index into its filenames list. If the function was inlined, the line table encodes this index as a `DW_AT_call_file` attribute; otherwise, it uses a `DW_AT_decl_file` attribute. It encodes lines as integers, and it similarly uses `DW_AT_call_line` for inlined functions and `DW_AT_decl_line` for non-inlined functions.

Let's also add a small helper function to *sdb/include/libsdb/dwarf.hpp* for retrieving the line entry corresponding to a file address:

```
namespace sdb {
    class dwarf {
    public:
        --snip--
        line_table::iterator line_entry_at_address(file_addr address) const {
            auto cu = compile_unit_containing_address(address);
            if (!cu) return {};
            return cu->lines().get_entry_by_address(address);
        }
        --snip--
    };
}
```

This code finds the compile unit corresponding to the given file address, retrieves the line table for that compile unit, and then returns the relevant entry. If it finds no compile unit for the given address, it returns an empty iterator.

We've finished implementing the support we'll give to users of the line table library. But, of course, no implementation is complete without testing.

Testing the Interpreter

As we did with the main DWARF parser, we'll write a small test for the line table interpreter to make sure its core behavior works. We'll parse *hello_sdb*, then check that the filename is correct and that the interpreter generates line entries for lines 2, 3, and 4:

```
TEST_CASE("Line table", "[dwarf]") {
    auto path = "targets/hello_sdb";
    sdb::elf elf(path);
    sdb::dwarf dwarf(elf);

    REQUIRE(dwarf.compile_units().size() == 1);

    auto& cu = dwarf.compile_units()[0];
    auto it = cu->lines().begin();

    REQUIRE(it->line == 2);
    REQUIRE(it->file_entry->path.filename() == "hello_sdb.cpp");

    ++it;
    REQUIRE(it->line == 3);

    ++it;
    REQUIRE(it->line == 4);

    ++it;
    REQUIRE(it->end_sequence);
    ++it;
    REQUIRE(it == cu->lines().end());
}
```

We parse the file and ensure that it contains only one compile unit. We then grab an iterator to the first entry. The line number should be 2 and the filename should be *hello_sdb.cpp*. We increment the iterator twice, each time ensuring that we parse the correct line number. Finally, we increment the iterator past the end sequence marker and then one more time, and ensure that we've reached the end of the list.

Summary

In this chapter, you learned how DWARF encodes line tables and wrote a parser that can interpret the line table program to re-create a program's full line table information bit by bit. In the next chapter, you'll finally reap the rewards of this parsing work by adding support for source-level breakpoints and stepping to your debugger.

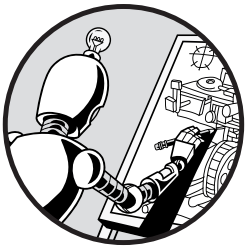
Check Your Knowledge

1. Why is the DWARF line table information encoded in a program rather than a simple binary table?
2. What are the three main categories of line table program opcodes?
3. What is the purpose of special opcodes?
4. What information does an entry in the line table provide?

14

SOURCE-LEVEL BREAKPOINTS AND STEPPING

A higher plane, above the world
where we live and dance
in pure thought.



In this chapter, the time you spent writing a DWARF parser will finally pay off, as you're ready to add source-level breakpoints and stepping support to the debugger. These features will use almost all of the functionality you've implemented up to this point and are perhaps the most-used features of a debugger.

The debugger already supports these operations for machine instructions, but it's much more useful for users of the debugger to operate at the level of the source code, as it's much easier to see how the execution relates to the code you wrote rather than what the compiler generated.

You'll use the DWARF parser to support three source-level stepping algorithms and three kinds of source-level breakpoints. You'll also learn how function inlining complicates these facilities and how debuggers hide inlining from the user to present a consistent debugging environment. We'll start by discussing inlining, because it affects both stepping and breakpoints.

Function Inlining

Function inlining is a compiler optimization that replaces a call to a function with the body of that function. In other words, instead of pointing to the code to jump to, the compiler copies the body of the function to the call site.

Inlining increases the code size, but as a tradeoff, it avoids the overhead of a function call, which requires the compiler to generate a function prologue or epilogue (the parts that perform necessary setup and cleanup). Removing the function call also enables the compiler to aggressively optimize the code for the callee in the context of the enclosing function. Lastly, inlining can interact with cache performance.

Even in debug builds, which usually aren't optimized, compilers inline some functions (for instance, ones marked `always_inline`), as well as certain function template specializations. For a concrete example of what inlining does, consider the following code:

```
#include <cstdio>

void call_puts() {
    std::puts("Hello");
}

int main() {
    call_puts();
}
```

This simple program defines a `call_puts` function that prints a message and then calls this function inside `main`. After we compile this code, the resulting assembly should look something like this:

```
call_puts:
    endbr64                 ;start of prologue
    push   %rbp
    mov    %rsp,%rbp        ;end of prologue
    lea    0xeac(%rip),%rax
    mov    %rax,%rdi
    call   1050 <puts@plt>
    nop
    pop    %rbp             ;epilogue
    ret

main:
    endbr64
    push   %rbp
    mov    %rsp,%rbp
    call   1149 <call_puts>
    mov    $0x0,%eax
    pop    %rbp
    ret
```

The important things to note here are that the `call_puts` function has four instructions that are part of the prologue or epilogue, and that inside `main` is a call to the `call_puts` function. To enforce inlining, mark `call_puts` as `always_inline`, like so:

```
--snip--
__attribute__((always_inline))
void call_puts() {
    std::puts("Hello");
}
--snip--
```

The code should then generate something similar to this assembly:

```
main:
    endbr64
    push   %rbp
    mov    %rsp,%rbp
 ❶ lea    0xe92(%rip),%rax
    mov    %rax,%rdi
    call   1050 <puts@plt>
    nop
    mov    $0x0,%eax
    pop    %rbp
    ret
```

Notice that the `call_puts` prologue and epilogue have disappeared and that the rest of the instructions in that function have moved directly into `main`, replacing the call to `call_puts`.

Inlining creates problems for debuggers because it blurs the lines between functions. Imagine that the program counter points to the call to the first line of the inlined `call_puts` function ❶. Should we report to the user that we're inside `main` or inside `call_puts`? Logically, that machine code instruction belongs to both functions.

Here is the model we'll follow: when the program halts on an instruction at the top of an inlined function, the debugger should maintain the illusion that a real function call instruction exists there. It should display the source code for the caller and then, if the user steps into function call, display the source code for the callee. To illustrate this rule, let's consider an example with multiple levels of inlining:

```
#include <cstdio>

__attribute__((always_inline))
void a() {
    std::puts("Hello");
}

__attribute__((always_inline))
```

```
void b() {
    a();
}

__attribute__((always_inline))
void c() {
    b();
}

int main() {
    c();
}
```

In this example, we have a chain of calls: `main` calls `c`, which calls `b`, which calls `a`, which calls `puts`, and `a`, `b`, and `c` are all marked as `always_inline`. If the user sets a breakpoint on `main` and continues, the debugger should report that the program is halted inside `main`, like this:

```
sdb> break set main
sdb> c
Process 937 stopped with signal TRAP at 0x5555555551a2,
force_inline.cpp:4 (main) (breakpoint 1)
  15 }
  16
  17 int main() {
> 18     c();
  19 }
```

Because the compiler has inlined the `a`, `b`, and `c` functions, the first instruction after the `main` function's prologue technically also belongs to all of those functions. However, the debugger should pretend that execution is inside `main` alone. If the user then performs a source-level step inside the function call, the debugger should pretend that the program state was changed and that execution is now inside `c`, even though it needs to execute zero instructions to get there:

```
sdb> step
Process 937 stopped with signal TRAP at 0x5555555551a2,
force_inline.cpp:4 (main) (single step)
  11
  12 __attribute__((always_inline))
  13 void c() {
> 14     b();
  15 }
  16
  17 int main() {
  18     c();
  19
```

Note that the program counter (`0x5555555551a2`) hasn't changed, but that the debugger now shows that execution is inside `c`. If the user steps again, execution should be inside `b`:

```
sdb> step
Process 937 stopped with signal TRAP at 0x5555555551a2,
force_inline.cpp:4 (main) (single step)
   6
   7 __attribute__((always_inline))
   8 void b() {
>  9     a();
  10 }
  11
  12 __attribute__((always_inline))
  13 void c() {
  14
```

Again, the program counter hasn't changed. The debugger has created the illusion of regular calls where there are none.

Inlining affects other operations as well, such as stepping out of functions or placing breakpoints on function entry, so it's important that we build facilities to recognize inlined functions and track their state as the debugger tries to maintain the facade of normality.

To implement all of this, you must understand how DWARF represents inlining. DWARF does so using DIEs with the `DW_TAG_inlined_subroutine` tag. For instance, the DWARF for the `call_puts` example looks like this:

```
DW_TAG_subprogram
  DW_AT_external  (true)
  DW_AT_name      ("main")
  DW_AT_decl_file ("test/targets/force_inline.cpp")
  DW_AT_decl_line (8)
  DW_AT_decl_column       (0x05)
  DW_AT_type      (0x0000017d "int")
  DW_AT_low_pc    (0x0000000000001163)
  DW_AT_high_pc   (0x0000000000001182)
  DW_AT_frame_base        (DW_OP_call_frame_cfa)
  DW_AT_GNU_all_tail_call_sites   (true)
  DW_AT_sibling   (0x00000744)

  DW_TAG_inlined_subroutine
    DW_AT_abstract_origin (0x00000744 "call_puts")
    DW_AT_low_pc  (0x000000000000116b)
    DW_AT_high_pc (0x000000000000117b)
    DW_AT_call_file       ("test/targets/force_inline.cpp")
    DW_AT_call_line       (9)
    DW_AT_call_column     (0x0e)
```

The DIE for the `main` function has a child that represents the inlined function. This DIE has a `DW_AT_abstract_origin` attribute, which is a reference to the `DW_TAG_subprogram` DIE for the function that was inlined. The `DW_AT_low_pc` and `DW_AT_high_pc` values tell us the program counter ranges for the inlined code.

Retrieving Inlined Function Stacks

We need a way to retrieve the inlined function stack for a given program counter value. For example, if we're at an offset of `0x116b`, we should be able to work out that we're inside the `call_puts` function inlined into `main`. To account for deeper chains of inlining, the debugger must be able to represent an arbitrarily deep inline stack.

Tracking Inlining

Let's add inline stack tracking to `sdb::target`. Whenever the process stops, the `target` object should recompute the current inline stack. Enabling this will require adding a way for `sdb::process` to notify the `target` of stops. Let's implement this now. Add a `target*` member to `sdb::process` and a member function for setting it in *sdb/include/libsdb/process.hpp*:

```
namespace sdb {
    class target;
    class process {
    public:
        --snip--
        void set_target(target* tgt) { target_ = tgt; }

    private:
        --snip--
        target* target_ = nullptr;
    };
}
```

We forward-declare `target` so we can form pointers to a `target` without introducing a circular dependency by including its header. We then add a new member variable that points to the parent target and a member function that modifies it.

We now need to notify the `target` when a stop occurs. Add a new member function for this purpose to `sdb::target` in *sdb/include/libsdb/target.hpp*:

```
namespace sdb {
    class target {
    public:
        --snip--
        void notify_stop(const sdb::stop_reason& reason);
        --snip--
```

```
    };
}
```

When we call `sdb::target::launch`, we should initialize the `target*` inside the child `process`. Modify `sdb::target::launch` in *sdb/src/target.cpp*:

```
std::unique_ptr<sdb::target>
sdb::target::launch(
    std::filesystem::path path, std::optional<int> stdout_replacement) {
    --snip--
 ❶ auto tgt = std::unique_ptr<target>(
        new target(std::move(proc), std::move(obj)));
    tgt->get_process().set_target(tgt.get());
    return tgt;
}
```

Rather than immediately returning the `std::unique_ptr` we create, we store it in a variable ❶. Then we call `set_target` on the stored `process` before returning the created `target`. We should do the same in `sdb::target::attach`:

```
std::unique_ptr<sdb::target>
sdb::target::attach(pid_t pid) {
    --snip--
    auto tgt = std::unique_ptr<target>(
        new target(std::move(proc), std::move(obj)));
    tgt->get_process().set_target(tgt.get());
    return tgt;
}
```

We need to call `sdb::target::notify_stop` when the process is stopped. Do this inside `sdb::process::wait_on_signal`, in *sdb/src/process.cpp*:

```
#include <libsdb/target.hpp>

sdb::stop_reason sdb::process::wait_on_signal() {
    --snip--

    if (is_attached_ and state_ == process_state::stopped) {
        --snip--

        if (reason.info == SIGTRAP) {
            --snip--
        }
        if (target_) target_->notify_stop(reason);
    }

    return reason;
}
```

We include the header for `sdb::target`. Inside the branch in which the process stops due to a signal, we call `notify_stop` on the `target`, if one is set.

Now that the `target` gets notified of stops, we can calculate the inline stack. In *sdb/include/libsdb/dwarf.hpp*, let's add a function to `sdb::dwarf` that calculates the inline stack at a given file address:

```
namespace sdb {
    class dwarf {
    public:
        --snip--
        std::vector<die> inline_stack_at_address(file_addr address) const;
        --snip--
    };
}
```

This function takes a file address and returns a list of DIEs representing the inline function stack at that address. The first element is the outer, non-inlined function. Subsequent elements represent functions inlined into the preceding function that contain the given address. Implement this function in *sdb/src/dwarf.cpp*:

```
std::vector<sdb::die> sdb::dwarf::inline_stack_at_address(file_addr address) const {
    auto func = function_containing_address(address);
    std::vector<sdb::die> stack;
    if (func) {
        stack.push_back(*func); ❶
        while (true) {
            const auto& children = stack.back().children();
            auto found = std::find_if(children.begin(), children.end(),
                [=](auto& child) {
                    return child.abbrev_entry()->tag == DW_TAG_inlined_subroutine and
                        child.contains_address(address);
                });
            if (found == children.end()) { ❷
                break;
            }
            else { ❸
                stack.push_back(*found);
            }
        }
    }
    return stack;
}
```

We start by looking for a non-inlined function containing the given address. If we find one, we begin the stack with this function ❶. Then we loop, looking for child DIEs that represent inlined functions and contain the given address. If we don't find any ❷, we stop looking. If we do find one ❸, we push it to the end of the stack and search that DIE's children.

Finding the Current Stack

Now that we can track the inline stack, we can add functionality to `sdb::target` to indicate which frame of the inline stack the debugger should present to the user as the current one. For this purpose, we'll make an `sdb::stack` type. We'll augment this type in Chapter 16, when we implement full stack unwinding.

The type should contain an `inline_height_` member that represents the location at which we'll consider the process to have stopped. Specifically, it should indicate a number of frames above the deepest inlined function. Whenever the process halts and execution is at the top of one or more inlined functions, we should reset this height to point to the outermost of those functions. Define this in a new file at *sdb/include/libsdb/stack.hpp*:

```
#ifndef SDB_STACK_HPP
#define SDB_STACK_HPP

#include <vector>
#include <libsdb/dwarf.hpp>

namespace sdb {
    class target;
    class stack {
    public:
        stack(target* tgt) : target_(tgt) {}
        void reset_inline_height();
        std::vector<sdb::die> inline_stack_at_pc() const;
        std::uint32_t inline_height() const { return inline_height_; }
        const target& get_target() const { return *target_; }

    private:
        target* target_ = nullptr;
        std::uint32_t inline_height_ = 0;
    };
}

#endif
```

The `sdb::stack` type's constructor takes the target to which this stack belongs. We declare a `reset_inline_height` function that we'll call whenever the process halts. We also declare some helper functions to retrieve the inline stack at the current program counter value, the current inline height, and the frame of the inline stack we're examining. Finally, we define two data members to hold a pointer to the target and the current inline height.

We'll implement these members in a new file at *sdb/src/stack.cpp*, starting with `inline_stack_at_pc`. To calculate the inline stack at the current program counter, we'll need to convert the program counter from a virtual address to a file address. As we'll perform this operation multiple times in this chapter,

let's add a function to `sdb::target` that implements it. Declare this function in *sdb/include/libsdb/target.hpp*:

```
namespace sdb {
    class target {
    public:
        --snip--
        file_addr get_pc_file_address() const;
        --snip--
    };
}
```

Implement the function in *sdb/src/target.cpp*:

```
sdb::file_addr sdb::target::get_pc_file_address() const {
    return process_->get_pc().to_file_addr(*elf_);
}
```

This function grabs the program counter virtual address from the process and converts it to a file address. Now we can implement `inline_stack_at_pc` in *sdb/src/stack.cpp*:

```
#include <libsdb/stack.hpp>
#include <libsdb/target.hpp>

std::vector<sdb::die>
sdb::stack::inline_stack_at_pc() const {
    auto pc = target_->get_pc_file_address();
    if (!pc.elf_file()) return {};
    return pc.elf_file()->get_dwarf().inline_stack_at_address(pc);
}
```

We grab the program counter as a file address, make sure we received a valid address, and then compute the inline stack at that address. Add this new file to the `add_library` call in *sdb/src/CMakeLists.txt*:

```
add_library(libsdb ... stack.cpp)
```

Let's move on to the `reset_inline_height` function. This function should walk backward over every inline stack frame for which execution is at the beginning, incrementing the current inline height each time:

```
void sdb::stack::reset_inline_height() {
    auto stack = inline_stack_at_pc();

    inline_height_ = 0;
    auto pc = target_->get_pc_file_address();
```

```
    for (auto it = stack.rbegin();
        it != stack.rend() and it->low_pc() == pc;
        ++it) {
        ++inline_height_;
    }
}
```

First, we calculate the inline stack at the current program counter. We initialize `inline_height_` to `0`, pointing to the deepest inline function. We then get the file address of the program counter so we can check it against the start address of each frame. Using reverse iterators (`rbegin` and `rend`), we iterate backward through the frames, starting at the deepest one, until we hit either the beginning or a frame of which execution isn't at the start. On each iteration, we increment the current inline height.

Let's hook this code into the rest of the system. Add a `sdb::stack` member to `sdb::target`:

```
#include <libsdb/stack.hpp>

namespace sdb {
    class target {
    public:
        --snip--
        stack& get_stack() { return stack_; }
        const stack& get_stack() const { return stack_; }

    private:
        target(std::unique_ptr<process> proc, std::unique_ptr<elf> obj)
            : process_(std::move(proc)), elf_(std::move(obj)),
              stack_(this)
        {}
        --snip--
        stack stack_;
    };
}
```

We include *stack.hpp*, add the data member to the bottom of the type definition, initialize this member in the constructor, and add members to retrieve the stack. Lastly, we implement `sdb::notify_stop` to call `stack_.reset_inline_height` when the process stops:

```
void sdb::target::notify_stop(const sdb::stop_reason& reason) {
    stack_.reset_inline_height();
}
```

Now, every time the process halts, we'll recalculate the inline height.

Source-Level Stepping

Source-level stepping operations walk through statements at the level of the original source code, rather than at the level of the machine code. Debuggers provide several kinds of steps. We'll support the following three:

Step in Steps to the next line of source code and into function calls

Step over Steps to the next line of source code, skipping over function calls

Step out Steps out of the current function call and back to the caller

I'll introduce the algorithms for these three step types as we implement them. Begin by declaring functions for them in `sdb::target`. Add the following to *sdb/include/libsdb/target.hpp*:

```
namespace sdb {
    class target {
    public:
        --snip--
        sdb::stop_reason step_in();
        sdb::stop_reason step_out();
        sdb::stop_reason step_over();
        --snip--
    };
}
```

Each of these functions will carry out the requested step and return an explanation of why the process halted. Stops can occur for reasons other than single-step requests; for example, if the user sets a breakpoint on a line of code inside a function whose execution the user steps over, the debugger should still hit that breakpoint and report it back to the user. Let's start by implementing step in.

Step In

The basic idea behind the step in operation is to step through single machine instructions until the program counter lands on an instruction that belongs to a different line of source code from the one at which it began. When the program counter arrives at a new source line, it might have entered a new function. In that case, we also skip over the prologue of that function, which sets up the stack. If we don't, other debugger features (like the version of step out we'll write shortly) might not operate properly, as they assume that the stack is correctly set up for the current function.

We also need to handle the situation in which execution is at the beginning of an inlined function. In that case, we pretend to step into the function by decrementing the current inline height and returning. Let's add a function to `sdb::stack` that supports this behavior:

```
namespace sdb {
    class stack {
    public:
        --snip--
        void simulate_inlined_step_in() {
            --inline_height_;
        }
    };
}
```

The `step_in` algorithm has a few parts to it, so we'll implement it in pieces. Add the following to *sdb/src/target.cpp*:

```
#include <csignal>

sdb::stop_reason sdb::target::step_in() {
    auto& stack = get_stack();
    if (stack.inline_height() > 0) {
        stack.simulate_inlined_step_in();
        return stop_reason(process_state::stopped, SIGTRAP, trap_type::single_step);
    }
```

We begin by getting a reference to the target's stack. Doing so might seem unnecessary, as we could just name `stack_` directly, but it will save us some refactoring when we add support for multiple threads in Chapter 18. If the current inline height is greater than 0, the debugger is representing the program state as stopped in a caller. To maintain the illusion of no inlining, we decrement the height. As a result, the debugger should tell the user that the program is now in the callee. We then return a `stop_reason` that indicates that the process stopped due to a single step. The `stop_reason` type doesn't currently support such manual construction, but we'll add this support soon.

Next, we handle the process of stepping to a new line of source code:

```
    --snip--
❶ auto orig_line = line_entry_at_pc();
    do {
     ❷ auto reason = process_->step_instruction();
     ❸ if (reason.reason != process_state::stopped
            or reason.info != SIGTRAP
            or reason.trap_reason != trap_type::single_step) {
            return reason;
        }
    } while ((line_entry_at_pc() == orig_line
        or line_entry_at_pc()->end_sequence)
        and line_entry_at_pc() != line_table::iterator{});
```

To work out when we're finished stepping, we need to know the execution's line entry location when the operation began. We'll find this location with a `line_entry_at_pc` function we'll write shortly. For now, we assume this function returns the line entry to which the program counter is currently pointing ❶.

Once we've received the line entry, we loop. On each iteration, we step over a single instruction and store the reason why execution stopped ❷. If the reason is something other than a single step ❸, the program may have stopped at a breakpoint or terminated completely. In this case, we stop trying to step and return the reason to the user.

Otherwise, we continue looping. The loop terminates when the line entry corresponding to the current program counter differs from the one we stored at the start of the operation. If this line entry is a special end-of-sequence marker, however, we keep stepping, as the marker doesn't correspond to an actual line of source code; it just terminates a sequence of line entries. We also terminate the loop if there's no line table entry for the new program counter value. (Note that if there's no line table entry for the program counter value at which `step_in` began, we still step over a single instruction.)

After this loop terminates, execution will have reached a new line of source code. However, we may still need to step over the function prologue if we've entered a new function:

```
--snip--
auto pc = get_pc_file_address();
if (pc.elf_file() != nullptr) {
    auto& dwarf = pc.elf_file()->get_dwarf();
    auto func = dwarf.function_containing_address(pc);
    if (func and func->low_pc() == pc) {
        auto line = line_entry_at_pc();
        if (line != line_table::iterator{}) {
            ++line;
            return run_until_address(line->address.to_virt_addr());
        }
    }
}
```

To check whether we're within the prologue of a function, we get the program counter value as a file address and find the function containing the program counter offset. If the program counter is at the start of that function's range, we know we've entered the prologue of a function.

DWARF supports marking line entries to indicate the end of the function prologue, but GCC doesn't output this information. Instead, the first line table entry for a function marks the start of the prologue, and the second entry marks the start of the function body. As such, we get the current line table entry (the entry for the prologue) and increment it by one to retrieve the entry marking the start of the function body. The program should then run until it reaches the address given by that second line table

entry, for which we defer to a `run_until_address` function we'll write shortly. If we can't find the ELF file for the program counter value, the function information, or a line table entry, we bail out of prologue skipping.

The last step is to return a stop reason explaining that we stopped due to a single step:

```
    --snip--
    return stop_reason(
        process_state::stopped, SIGTRAP, trap_type::single_step);
}
```

The `sdb::stop_reason` type doesn't currently support manual construction, so let's add a constructor to it in *sdb/include/libsdb/process.hpp*. While we're at it, let's also add a default constructor and functions to easily check whether a stop occurred due to a single step or a breakpoint at a given address, as a way to clean up some of the code we just wrote and make the rest of the step algorithms easier to implement:

```
#include <csignal>

namespace sdb {
    struct stop_reason {
        --snip--
        stop_reason() = default;
        stop_reason(process_state reason, std::uint8_t info,
            std::optional<trap_type> trap_reason = std::nullopt,
            std::optional<syscall_information> syscall_info = std::nullopt)
            : reason(reason)
            , info(info)
            , trap_reason(trap_reason)
            , syscall_info(syscall_info)
        {}

        bool is_step() const {
            return reason == process_state::stopped
                and info == SIGTRAP
                and trap_reason == trap_type::single_step;
        }
        bool is_breakpoint() const {
            return reason == process_state::stopped
                and info == SIGTRAP
                and (trap_reason == trap_type::software_break
                    or trap_reason == trap_type::hardware_break);
        }
        --snip--
    };
}
```

The constructor accepts the various pieces of data that a `stop_reason` holds and stores them in the data members. The `is_step` and `is_breakpoint` functions check that the fields are set as expected. Now we can replace that check from earlier, changing

```
if (reason.reason != process_state::stopped
            or reason.info != SIGTRAP
            or reason.trap_reason != trap_type::single_step) {
    return reason;
}
```

into this:

```
if (!reason.is_step()) return reason;
```

Next, we'll implement the `line_entry_at_pc` and `run_until_address` helper functions. Declare them both in *sdb/include/libsdb/target.hpp*:

```
#include <libsdb/dwarf.hpp>

namespace sdb {
    class target {
    public:
        --snip--
        sdb::line_table::iterator line_entry_at_pc() const;
        sdb::stop_reason run_until_address(virt_addr address);
        --snip--
    };
}
```

Write the functions in *sdb/src/target.cpp*, starting with `line_entry_at_pc`:

```
sdb::line_table::iterator
sdb::target::line_entry_at_pc() const {
    auto pc = get_pc_file_address();
    if (!pc.elf_file()) return line_table::iterator();
    auto cu = pc.elf_file()->get_dwarf().compile_unit_containing_address(pc);
    if (!cu) return line_table::iterator();
    return cu->lines().get_entry_by_address(pc);
}
```

We grab the current program counter as a file address. We might get an empty file address (for example, if the function currently being executed belongs to a shared library), so we check it and return an empty iterator if it's invalid. We then locate the compile unit for this address. Again, if we get a null pointer, we return an empty iterator. Otherwise, we return the entry corresponding to the current program counter in the correct compile unit.

Next, we'll implement `run_until_address`. This function will set a temporary breakpoint on the given address, continue until it reaches the breakpoint, and then remove the breakpoint. Add it to *sdb/src/target.cpp*:

```
sdb::stop_reason sdb::target::run_until_address(virt_addr address) {
    breakpoint_site* breakpoint_to_remove = nullptr;
 ❶ if (!process_->breakpoint_sites().contains_address(address)) {
        breakpoint_to_remove = &process_->create_breakpoint_site(
            address, false, true);
        breakpoint_to_remove->enable();
    }

 ❷ process_->resume();
    auto reason = process_->wait_on_signal();
    if (reason.is_breakpoint()
        and process_->get_pc() == address) {
        reason.trap_reason = trap_type::single_step;
    }

 ❸ if (breakpoint_to_remove) {
        process_->breakpoint_sites().remove_by_address(
            breakpoint_to_remove->address());
    }

    return reason;
}
```

There may already be a breakpoint set on the given address, and we can't set a new breakpoint there if that is the case. To deal with this situation, we track a pointer to the breakpoint that we'll remove after continuing, but allow it to be null if a breakpoint already exists there. If no breakpoint exists at the given address ❶, we create a new one, mark it as internal, store a pointer to it, and enable it. We then resume the process ❷ and wait until it halts.

Note that execution may halt for a reason other than hitting the breakpoint at the desired address: the process may terminate, or it may hit some other breakpoint before execution reaches the one we expect. As such, we check the reason for the halt. If it's due to the breakpoint we expect, we set the trap reason to single step, which the debugger will report to the user. Otherwise, we return the same reason given by `wait_on_signal`. Before returning the stop reason, we remove any temporary breakpoint we may have set ❸.

Now that we've implemented step in, we can move on to step over.

Step Over

Step over is a bit more complicated than step in. The step types share the same basic idea: they keep stepping over instructions until execution reaches a different line table entry than the one at which it began. However, if execution reaches a function call, we must step over the call, which we'll achieve by placing a breakpoint on the instruction immediately succeeding the call

and continuing the process. Furthermore, if we're currently above an inlined function, we should step over that entire block. Again, we'll implement this function step by step. Add the following in *sdb/src/target.cpp*:

```
#include <optional>
#include <libsdb/disassembler.hpp>

sdb::stop_reason sdb::target::step_over() {
    auto orig_line = line_entry_at_pc();
    disassembler disas(*process_);
    sdb::stop_reason reason;
    auto& stack = get_stack();
    do {
        // Blank for now
    } while ((line_entry_at_pc() == orig_line
        or line_entry_at_pc()->end_sequence)
        and line_entry_at_pc() != line_table::iterator{});
    return reason;
}
```

As with step in, we start by saving the line entry to which the program counter currently points. Next, we create a disassembler for the process, which we'll use to determine whether the next instruction to be executed is a function call. We take a reference to the stack and loop until execution reaches a new line table entry that isn't an end sequence marker, as we did for step in.

Now we'll tackle the loop body. If we're at the start of an inlined function, we jump over it. If we're at a call instruction, we jump to the instruction directly succeeding it. Otherwise, we step over a single instruction:

```
sdb::stop_reason sdb::target::step_over() {
    --snip--
    do {
        auto inline_stack = stack.inline_stack_at_pc(); ❶
        auto at_start_of_inline_frame = stack.inline_height() > 0;
        if (at_start_of_inline_frame) {
            auto frame_to_skip = inline_stack[inline_stack.size() - stack.inline_height()]; ❷
            auto return_address = frame_to_skip.high_pc().to_virt_addr();
            reason = run_until_address(return_address);
            if (!reason.is_step() ❸
                or process_->get_pc() != return_address) {
                return reason;
            }
        }
        else if (auto instructions = disas.disassemble(2, process_->get_pc()); ❹
            instructions[0].text.rfind("call") == 0) {
            reason = run_until_address(instructions[1].address);
```

```
            if (!reason.is_step()
                or process_->get_pc() != instructions[1].address) {
                return reason;
            }
        }
        else { ❺
            reason = process_->step_instruction();
            if (!reason.is_step()) return reason;
        }
    } while (line_entry_at_pc() == orig_line or
             line_entry_at_pc()->end_sequence);
    --snip--
}
```

We compute the inline stack ❶ and work out whether it contains any inline frames and, if so, whether we're at the start of one of them. In that case, we jump over the inlined frame directly below the one that the debugger is tracking as the current frame. We figure out which frame to skip ❷, calculate the end of the inlined code, and run up to that address. If the process stops for some other reason ❸, such as a breakpoint being set inside the inlined function, we report this.

This implementation of skipping inline frames will work in most cases with unoptimized code, but it's not necessarily the case that control flow will hit the instruction directly after the inlined block after completing its execution. Production debuggers will look for any branch instructions in the inlined block and, if they find any, will step through the inlined block by putting breakpoints at each branch instruction until the program makes it out of the program counter ranges for the inlined function. This approach is significantly more complex, so I've opted for the simpler solution here.

If we're not at the start of an inline frame, we check whether we instead need to step over a call instruction ❹. We disassemble two instructions at the current program counter. We need two instructions because we need to know two things: whether the current instruction is a function call and the address of the subsequent instruction, so we can set a breakpoint on it.

If the first instruction starts with "call," we run up until the subsequent instruction's address. Again, if the process halted for some other reason, we immediately stop stepping and return that reason. If the first instruction doesn't start with "call" and we're not at an inlined frame ❺, we step over a single instruction. At this point, we've completed step over and can move on to step out.

Step Out

Step out is a bit different from the other two step algorithms. Instead of continually stepping, it should make the process run up until the return address of the currently executing function. How do we calculate the return address? Well, that's a bit tricky.

Recall the simplified x64 stack we discussed in Chapter 2, in which each stack frame contains the return address of the current function. However, while the program is running, it's not clear how to locate exactly where on the stack the return address is. We can figure this out in two main ways.

The more robust way of determining this location is to parse the DWARF information for details about the call frame. While parsing the DWARF is the robust way to solve this problem, it's also quite complex. We'll do this in Chapter 16, when we implement full stack unwinding.

In this section, I'll teach you the easy way, which relies on using specific compiler options when building the program to debug. If you compile the debugger program with `-fno-omit-frame-pointer`, the frame base of the currently executing function gets stored in the `rbp` register, and the frame base for the caller function gets stored on the stack, just after the return address and just before the local variables. The *frame base* for a stack frame is the memory address directly after the stored return address. Figure 14-1 shows a simplified x64 stack for a function when compiled with `-fno-omit-frame-pointer`.

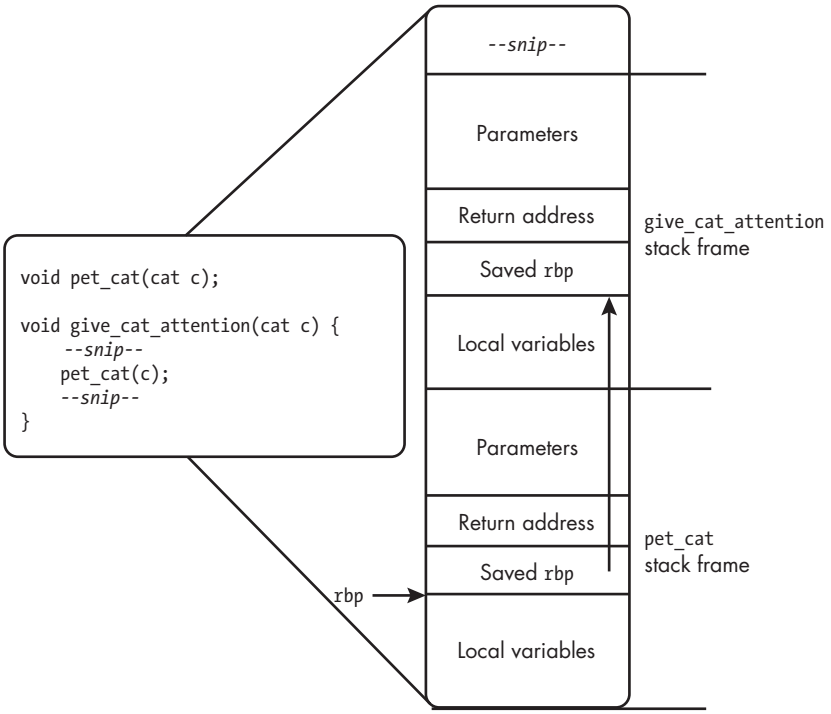


Figure 14-1: A simplified x64 stack with base pointers

The `rbp` register points to the memory location directly following the return address for the currently executing function, so to figure out where to set a breakpoint to step out, we just need to read the memory 8 bytes

above the current value of `rbp`. If we're inside an inlined function, we instead find the end address of the inlined block. With this knowledge, we can now implement `step_out`:

```
#include <libsdb/bit.hpp>

sdb::stop_reason sdb::target::step_out() {
    auto& stack = get_stack();
    auto inline_stack = stack.inline_stack_at_pc();
    auto has_inline_frames = inline_stack.size() > 1;
    auto at_inline_frame = stack.inline_height() < inline_stack.size() - 1;

    if (has_inline_frames and at_inline_frame) {
        auto current_frame = inline_stack[inline_stack.size() - stack.inline_height() - 1];
        auto return_address = current_frame.high_pc().to_virt_addr();
        return run_until_address(return_address);
    }

    auto frame_pointer = process_->get_registers()
        .read_by_id_as<std::uint64_t>(register_id::rbp);

    auto return_address = process_->read_memory_as<std::uint64_t>(
        virt_addr{ frame_pointer + 8 });

    return run_until_address(virt_addr{ return_address });
}
```

We begin by retrieving the current inline stack. If we find more than one frame and the current inline height isn't pointing to the outermost frame, we're looking at an inline frame and should step to the end of it. To do this, we retrieve the current frame, calculate the end of its range as a virtual address, and then run up until that address, returning the stop reason. Note that the same caveats apply here as to skipping over inlined functions in our `step_out` implementation: the approach doesn't work in cases where control flow doesn't hit the instruction directly after an inlined block.

If we're not at an inline frame, we instead read the return address from the stack. We read the current value of the `rbp` register, read the memory that is 8 bytes above it, and convert the result to an integer. This integer gives us the return address of the current function, which we then pass to `run_until_address` to complete the step out operation.

One place where this algorithm fails to work properly is if the program is halted inside a recursive function at least one level deep. In that case, trying to step out may actually step into another function call, because the return address can be hit by a recursive call. We'll address this case in Chapter 16, where we'll implement stack unwinding.

Exposing Stepping to the User

With all three step operations complete, we can add corresponding commands to the debugger's user interface. We'll support the following stepping commands: `next`, for performing a step over; `finish`, for performing a step out; `step`, for performing a step in; and `stepi`, for instruction-level stepping.

Modify the existing `step` command and add the new ones to *sdb/tools/sdb.cpp*:

```
namespace {
    void handle_command(std::unique_ptr<sdb::target>& target,
          std::string_view line) {
        --snip--
        else if (is_prefix(command, "next")) {
            auto reason = target->step_over();
            handle_stop(*target, reason);
        }
        else if (is_prefix(command, "finish")) {
            auto reason = target->step_out();
            handle_stop(*target, reason);
        }
        else if (is_prefix(command, "step")) {
            auto reason = target->step_in();
            handle_stop(*target, reason);
        }
        else if (is_prefix(command, "stepi")) {
            auto reason = process->step_instruction();
            handle_stop(*target, reason);
        }
        --snip--
}
```

These commands call the relevant function on the target or process and then call `handle_stop` to report the stop to the user. Let's also update the help information to document these commands:

```
namespace {
    void print_help(const std::vector<std::string>& args) {
        if (args.size() == 1) {
            std::cerr << R"(Available commands:
    breakpoint  - Commands for operating on breakpoints
    catchpoint  - Commands for operating on catchpoints
    continue    - Resume the process
    disassemble - Disassemble machine code to assembly
    finish      - Step-out
    memory      - Commands for operating on memory
    next        - Step-over
```

```
    register    - Commands for operating on registers
    step        - Step-in
    stepi       - Single instruction step
    watchpoint  - Commands for operating on watchpoints
)";
        }
        --snip--
    }
}
```

We've finished implementing stepping, so let's move on to breakpoints.

Source-Level Breakpoints

Setting source-level breakpoints generally requires us to do two things: resolve the source-level request into a set of memory addresses and set breakpoint sites on all of those addresses.

For example, if the user requests a breakpoint on the function `pet_cat`, we must first find the addresses for the start of each `pet_cat` function in the current program and then set breakpoint sites on each of those addresses. If, on the other hand, the user requests a breakpoint on *cat_petting.cpp* line 42, we should find the memory address that corresponds to that line and create a breakpoint site for it.

Let's create a new `sdb::breakpoint` type to represent breakpoints and then use inheritance to create types for the different possible kinds of breakpoints:

`sdb::function_breakpoint` A breakpoint set on a function name

`sdb::line_breakpoint` A breakpoint set on a line of a source file

`sdb::address_breakpoint` A breakpoint set on a virtual memory address

That last type essentially corresponds to an `sdb::breakpoint_site`, but having a unified interface for dealing with breakpoints will make our lives easier. Begin by implementing `sdb::breakpoint` in *sdb/include/libsdb/breakpoint.hpp*:

```
#ifndef SDB_BREAKPOINT_HPP
#define SDB_BREAKPOINT_HPP

#include <cstdint>
#include <cstddef>
#include <string>
#include <libsdb/stoppoint_collection.hpp>
#include <libsdb/breakpoint_site.hpp>
#include <libsdb/types.hpp>

namespace sdb {
    class target;
```

```
    class breakpoint {
    public:
     ❶ virtual ~breakpoint() = default;

     ❷ breakpoint() = delete;
        breakpoint(const breakpoint&) = delete;
        breakpoint& operator=(const breakpoint&) = delete;

     ❸ using id_type = std::int32_t;
        id_type id() const { return id_; }

        void enable();
        void disable();

        bool is_enabled() const { return is_enabled_; }
        bool is_hardware() const { return is_hardware_; }
        bool is_internal() const { return is_internal_; }

     ❹ virtual void resolve() = 0;

     ❺ /*?*/ breakpoint_sites() { return breakpoint_sites_; }

    protected:
        friend target;
     ❻ breakpoint(
            target& tgt, bool is_hardware = false, bool is_internal = false);

        id_type id_;
        target* target_;
        bool is_enabled_ = false;
        bool is_hardware_ = false;
        bool is_internal_ = false;
        /*?*/ breakpoint_sites_;
        breakpoint_site::id_type next_site_id_ = 1;
    };
}
#endif
```

Because `sdb::breakpoint` is designed for inheritance, we declare a virtual destructor for it ❶. Now, any time a derived type gets deleted through a pointer to an `sdb::breakpoint`, it will call the derived destructor as well.

Like breakpoint sites, breakpoints should be unique and constructible only through their managing type (in this case, `sdb::target`). As such, we delete the default constructor and copy operations ❷. As with breakpoint sites, we define an ID type ❸, a function to retrieve the breakpoint ID, functions for enabling and disabling the breakpoint, and queries for whether the breakpoint is enabled and whether it's a hardware breakpoint or an internal breakpoint.

To customize the `sdb::breakpoint` subtypes, we use the `resolve` function ❹, which creates breakpoint sites based on the breakpoint type and the arguments passed to it. We declare it as a virtual function using the `virtual` keyword and add `= 0` at the end of the declaration so the subtypes are forced to override it. We'll also need a function called `breakpoint_sites` to return the set of breakpoint sites resolved by a breakpoint, but we need to put more thought into what type to return, so we leave it as a question mark for now ❺.

The IDs for breakpoint sites should be relative to each breakpoint. That is, if a breakpoint has three sites, they'll have the IDs 1, 2, and 3, regardless of how many other breakpoint sites there are. To achieve this design, we store a `next_site_id_` member that determines the ID of the next breakpoint site resolved for this breakpoint.

We declare a protected constructor (not a private one, because we want derived types to be able to access all members) ❻ and declare `sdb::target` as a `friend` so that it can construct new `sdb::breakpoint` objects. Finally, we define all of the members we'll need.

Expanding Breakpoint Sites

Breakpoint sites can now belong to breakpoints, so they should track their parent and allow the parent to supply them with IDs. In *sdb/include/libsdb/breakpoint_site.hpp*, add a `parent_` member and a new constructor that takes a parent breakpoint and an ID for the site:

```
namespace sdb {
    class breakpoint;
    class breakpoint_site {
        --snip--
    private:
        --snip--
        breakpoint_site(
            breakpoint* parent, id_type id,
            process& proc, virt_addr address,
            bool is_hardware = false, bool is_internal = false);
        --snip--
        breakpoint* parent_ = nullptr;
    };
}
```

Implement the constructor in *sdb/src/breakpoint_site.cpp*:

```
sdb::breakpoint_site::breakpoint_site(
    breakpoint* parent, id_type id,
    process& proc, virt_addr address,
    bool is_hardware, bool is_internal)
    : parent_{ parent }, id_(id),
      process_{ &proc }, address_{ address },
```

```
      is_enabled_{ false }, saved_data_{},
      is_hardware_{ is_hardware }, is_internal_{ is_internal } {
}
```

Also add a new overload for `sdb::process::create_breakpoint_site` to *sdb/include/libsdb/process.hpp*:

```
namespace sdb {
    class process {
    public:
        --snip--
        breakpoint_site& create_breakpoint_site(
            breakpoint* parent, breakpoint_site::id_type id,
            virt_addr address,
            bool hardware = false, bool internal = false);
        --snip--
    };
}
```

Implement this overload in *sdb/src/process.cpp*:

```
sdb::breakpoint_site&
sdb::process::create_breakpoint_site(
      breakpoint* parent, breakpoint_site::id_type id, virt_addr address,
      bool hardware, bool internal) {
    if (breakpoint_sites_.contains_address(address)) {
        error::send("Breakpoint site already created at address " +
            std::to_string(address.addr()));
    }
    return breakpoint_sites_.push(
        std::unique_ptr<breakpoint_site>(
            new breakpoint_site(
                parent, id, *this, address, hardware, internal)));
}
```

This function ensures that no breakpoint site already exists at that address and then creates a new one, forwards on all the parameters, and pushes it into the `breakpoint_sites_` collection. Next, we'll decide what type to use for `breakpoint_sites_` inside `sdb::breakpoint`.

Determining the Breakpoint Site Type

To store the breakpoint sites resolved by a breakpoint, you might initially want to use the `sdb::stoppoint_collection` type you wrote in Chapter 9. However, doing so causes an issue: `sdb::stoppoint_collection` owns the stop points that it stores, holding them as `std::unique_ptr` objects. We want to instead hold non-owning pointers to existing `sdb::breakpoint_site` objects stored

inside the sdb::process object. This will allow us to centrally manage all breakpoint sites and make sure that we don't, for example, try to create two breakpoints that refer to the same breakpoint site, which would be a recipe for data corruption.

We don't need to write an entirely new type to accomplish this; we can extend sdb::stoppoint_collection to support both owning and non-owning policies. We'll add another template parameter to sdb::stoppoint_collection that indicates whether it should own its stop points or simply refer to stop points stored elsewhere. Edit the *sdb/include/libsdb/stoppoint_collection.hpp* file:

```
namespace sdb {
    template <class Stoppoint, bool Owning = true>
    class stoppoint_collection {
        --snip--
    };
}
```

We add a template parameter called Owning, which defaults to true. Now we need to make some changes to the stored type and to the type that the push function accepts:

```
#include <type_traits>

namespace sdb {
    template <class Stoppoint, bool Owning = true>
    class stoppoint_collection {
    public:
     ❶ using pointer_type = std::conditional_t<Owning,
            std::unique_ptr<Stoppoint>,
            Stoppoint*>;

     ❷ Stoppoint& push(pointer_type bs);
        --snip--

    private:
     ❸ using points_t = std::vector<pointer_type>;
        --snip--
    };
}
```

We start by introducing a member type called pointer_type. This type uses std::conditional_t, which is basically an if statement that runs at compile time ❶. If the first argument (Owning) is true, then pointer_type is an alias for std::unique_ptr<Stoppoint>. Otherwise, it's an alias for Stoppoint*. We update the push function to take an argument with the correct pointer type ❷ and update points_t to store a vector of this pointer type ❸.

The member function implementations will work with no additional changes thanks to the power of templates, but we do need to update their signatures. For example, `contains_id` used to have the following signature:

```
template <class Stoppoint>
bool stoppoint_collection<Stoppoint>::contains_id(
      typename Stoppoint::id_type id) const {
    --snip--
}
```

Because we added a new template parameter, we need to update the template parameters to this:

```
template <class Stoppoint, bool Owning>
bool stoppoint_collection<Stoppoint,Owning>::contains_id(
      typename Stoppoint::id_type id) const {
    --snip--
}
```

Add a `bool Owning` template parameter and supply `Owning` as a template argument to `stoppoint_collection`. You'll need to make the same adjustment for every member function. It's a bit tedious, but it can be handled quickly by running a search and replace in your editor. For example, replace `<class Stoppoint>` and `stoppoint_collection<Stoppoint>` with `<class Stoppoint, bool Owning>` and `stoppoint_collection<Stoppoint,Owning>` across the whole file. Also be sure to change the `push` parameter type in its definition, like so:

```
template <class Stoppoint, bool Owning>
Stoppoint&
stoppoint_collection<Stoppoint,Owning>::push(
     ❶ pointer_type bs) {
    --snip--
}
```

We update the parameter ❶ to be `pointer_type` instead of `std::unique_ptr <Stoppoint>`. The `sdb::stoppoint_collection` type is now ready for use inside `sdb::breakpoint`.

Update the `breakpoint_sites` member function and `breakpoint_sites_` member data in *sdb/include/libsdb/breakpoint.hpp*. Also add versions of the `at_address` and `in_range` member functions used by `stoppoint_collection` so that users can store a `stoppoint_collection<breakpoint>`. Finally, add a `const` overload for `breakpoint_sites`:

```
namespace sdb {
    class breakpoint {
    public:
        --snip--
        stoppoint_collection<breakpoint_site, false>&
        breakpoint_sites() { return breakpoint_sites_; }
        const stoppoint_collection<breakpoint_site, false>&
        breakpoint_sites() const { return breakpoint_sites_; }
```

```
        bool at_address(virt_addr addr) const {
            return breakpoint_sites_.contains_address(addr);
        }
        bool in_range(virt_addr low, virt_addr high) const {
            return !breakpoint_sites_.get_in_region(low, high).empty();
        }

    protected:
        --snip--
        stoppoint_collection<breakpoint_site, false> breakpoint_sites_;
    };
}
```

We use `stoppoint_collection<breakpoint_site, false>` as the `breakpoint_sites_` type, passing `false` to make the collection non-owning. No other uses of `sdb::stoppoint_collection` in the codebase need modification, because we defaulted the second argument to `true`. The implementations of `at_address` and `in_range` check whether at least one of the breakpoint sites resolved by this breakpoint is at the requested address or in the requested range.

Implementing Breakpoint Functions

Now we can implement the `sdb::breakpoint` member functions. These should handle constructing, enabling, disabling, and resolving breakpoints. Create a new file, *sdb/src/breakpoint.cpp*, with the following contents:

```
#include <libsdb/breakpoint.hpp>
#include <libsdb/target.hpp>

namespace {
 ❶ auto get_next_id() {
        static sdb::breakpoint::id_type id = 0;
        return ++id;
    }
}

sdb::breakpoint::breakpoint(
    target& tgt, bool is_hardware, bool is_internal)
    : target_{ &tgt }, is_hardware_{ is_hardware },
    is_internal_{ is_internal } {
 ❷ id_ = is_internal ? -1 : get_next_id();
}

void sdb::breakpoint::enable() {
    is_enabled_ = true;
    breakpoint_sites_.for_each([](auto& site) { site.enable(); });
}
```

```
void sdb::breakpoint::disable() {
    is_enabled_ = false;
    breakpoint_sites_.for_each([](auto& site) { site.disable(); });
}
```

We handle the ID in the same way as in `sdb::breakpoint_site`, by writing a `get_next_id` function ❶ with a local static ID that increments on every call and then calling this function in the constructor so long as the breakpoint isn't internal ❷. The `enable` and `disable` functions resemble each other: they both set the `is_enabled_` member to the relevant value and then call either `.enable` or `.disable` on each breakpoint site in the stored collection. Add this file to *sdb/src/CMakeLists.txt*:

```
add_library(libsdb ... breakpoint.cpp)
```

The base breakpoint type is complete. The complexity lies in the subtypes.

Creating Breakpoint Subtypes

Recall that we'll support three types of breakpoints, for functions, lines, and addresses. Let's create types for each of these, beginning with `function_breakpoint`. Define all three types in *sdb/include/libsdb/breakpoint.hpp*:

```
namespace sdb {
    class function_breakpoint : public breakpoint {
    public:
        void resolve() override;
        std::string_view function_name() const { return function_name_; }

    private:
        friend target;
        function_breakpoint(
            target& tgt, std::string function_name,
            bool is_hardware = false, bool is_internal = false)
            : breakpoint(tgt, is_hardware, is_internal)
            , function_name_(std::move(function_name)) {
            resolve();
        }
        std::string function_name_;
    };
}
```

We declare `sdb::function_breakpoint` as a derived type of `breakpoint` using the `: public breakpoint` syntax. We need to implement a custom `resolve` function, so we declare it and mark it as `override`. Marking it as `override` isn't entirely necessary, but doing so helps the compiler diagnose any mistakes that we make more easily and signals our intent.

The `function_breakpoint` type stores the function name given to it on construction, so we write a member function to retrieve the name and a

member to store it. Because `sdb::target` will always be responsible for constructing this type, we declare the constructor as `private` and make `target` a `friend`. The constructor calls the base class constructor for `breakpoint` with the given target, stores the function name, and then calls `resolve` to initialize the breakpoint sites for the breakpoint.

Before we implement `resolve`, let's define the other two breakpoint types. They're very similar, except they provide different members depending on the data they need to store:

```
#include <filesystem>

namespace sdb {
    class line_breakpoint : public breakpoint {
    public:
        void resolve() override;
        const std::filesystem::path file() const { return file_; }
        std::size_t line() const { return line_; }

    private:
        friend target;
        line_breakpoint(target& tgt,
            std::filesystem::path file,
            std::size_t line,
            bool is_hardware = false,
            bool is_internal = false)
            : breakpoint(tgt, is_hardware, is_internal), file_(std::move(file)), line_(line) {
            resolve();
        }
        std::filesystem::path file_;
        std::size_t line_;
    };

    class address_breakpoint : public breakpoint {
    public:
        void resolve() override;
        virt_addr address() const { return address_; }
    private:
        friend target;
        address_breakpoint(
            target& tgt, virt_addr address,
            bool is_hardware = false, bool is_internal = false)
            : breakpoint(tgt, is_hardware, is_internal), address_(address) {
            resolve();
        }
        virt_addr address_;
    };
}
```

The `line_breakpoint` type stores a path to the file on which it should be set, as well as the desired line number. The `address_breakpoint` type stores an address instead.

Now we'll tackle the `resolve` functions, which create the breakpoint sites for a breakpoint. The `address_breakpoint` version is the easiest, so let's start with that. Implement it in *sdb/src/breakpoint.cpp*:

```
void sdb::address_breakpoint::resolve() {
    if (breakpoint_sites_.empty()) {
        auto& new_site = target_->get_process()
            .create_breakpoint_site(
                this, next_site_id_++, address_, is_hardware_, is_internal_);
        breakpoint_sites_.push(&new_site);
        if (is_enabled_) new_site.enable();
    }
}
```

If we haven't already resolved the breakpoint, we create a new breakpoint site at the given address and store a pointer to it in the `breakpoint_sites_` member. We also increment the `next_site_id_` member so that subsequent breakpoint sites will receive higher IDs. Currently, we resolve breakpoints only once, upon construction, but in Chapter 17, we'll implement shared library tracking, which will require us to resolve breakpoints whenever a new shared library is loaded. As such, if this breakpoint is already marked as enabled, we enable the new breakpoint site.

Let's move on to `function_breakpoint`, which is more complex. We need to find the DIEs for the functions we're interested in and then create breakpoint sites following those functions' prologues. If the DWARF information for the ELF file isn't sufficient, we'll fall back to the ELF symbol table to resolve the breakpoint. Let's add a `find_functions` function to `sdb::target` that implements this behavior. Declare the function in *sdb/include/libsdb/target.hpp*:

```
namespace sdb {
    class target {
    public:
        --snip--
        struct find_functions_result {
            std::vector<die> dwarf_functions;
            std::vector<std::pair<const elf*, const Elf64_Sym*>> elf_functions;
        };
        find_functions_result find_functions(std::string name) const;
        --snip--
    };
}
```

This function returns a struct that wraps the DWARF DIEs and ELF symbols found for the given name. For now, we fill in only one of these options on each call, but in Chapter 17, we'll support shared libraries, where

some ELF files may have DWARF information and others may not. For that reason, we make this value a struct rather than a union or a `std::variant`. Implement the function in *sdb/src/target.cpp*:

```
sdb::target::find_functions_result
sdb::target::find_functions(std::string name) const {
    find_functions_result result;

    auto dwarf_found = elf_->get_dwarf().find_functions(name);
    if (dwarf_found.empty()) {
        auto elf_found = elf_->get_symbols_by_name(name);
        for (auto sym : elf_found) {
            result.elf_functions.push_back(std::pair{ elf_.get(), sym });
        }
    }
    else {
        result.dwarf_functions.insert(
            result.dwarf_functions.end(),
            dwarf_found.begin(), dwarf_found.end());
    }

    return result;
}
```

We initialize an instance of `find_functions_result` to hold the DWARF data we retrieve. Within that information, we then try to locate functions, inserting them into the `result` object. If we don't find any functions, we look them up in the ELF symbol table instead and then push them into the `result` object. Finally, we return the result.

With this function written, we can implement the `sdb::function_breakpoint::resolve` function in *sdb/src/breakpoint.cpp*. We'll start by implementing resolution for DIEs and then handle ELF symbols:

```
void sdb::function_breakpoint::resolve() {
    auto found_functions = target_->find_functions(function_name_);
    for (auto die : found_functions.dwarf_functions) {
        if (die.contains(DW_AT_low_pc) or die.contains(DW_AT_ranges)) {
            file_addr addr;
            if (die.abbrev_entry()->tag == DW_TAG_inlined_subroutine) { ❶
                addr = die.low_pc();
            }
            else { ❷
                auto function_line = die.cu()->lines()
                    .get_entry_by_address(die.low_pc());
                ++function_line;
                addr = function_line->address;
            }
            auto load_address = addr.to_virt_addr(); ❸
```

```
            if (!breakpoint_sites_.contains_address(load_address)) {
                auto& new_site = target_->get_process()
                    .create_breakpoint_site(
                        this, next_site_id_++, load_address, is_hardware_, is_internal_);

                breakpoint_sites_.push(&new_site);
                if (is_enabled_) new_site.enable();
            }
        }
    }
```

We loop over all of the DIEs that match the given name. If we can retrieve the function's start address, we try to resolve the first instruction after the prologue. If the function was inlined ❶, it has no prologue, so we set a breakpoint at the start of the function. Otherwise, we find the corresponding line entry for the start of the function and then advance the resulting iterator to find the entry pointing to the first instruction after the prologue ❷. Because the address we just worked out is a file address rather than a virtual address, we calculate where the instruction was loaded ❸ and then create a breakpoint site if one doesn't already exist.

Now we can implement resolution for ELF symbols, which is simpler:

```
    --snip--
    for (auto sym : found_functions.elf_functions) {
        auto file_address = file_addr{ *sym.first, sym.second->st_value };
        auto load_address = file_address.to_virt_addr();
        if (!breakpoint_sites_.contains_address(load_address)) {
            auto& new_site = target_->get_process().create_breakpoint_site(
                this, next_site_id_++, load_address, is_hardware_, is_internal_);
            breakpoint_sites_.push(&new_site);
            if (is_enabled_) new_site.enable();
        }
    }
}
```

We loop over the functions found in the ELF symbol table, retrieving their file addresses and converting them to load addresses. If no breakpoint exists there yet, we create a new one, push it into the tracked breakpoint sites for this breakpoint, and enable it, if the breakpoint is enabled.

Finally, let's implement line breakpoint resolution. This function should find the memory address of the instruction corresponding to the given line, potentially skip over a function prologue, and then create a breakpoint site at the correct load address:

```
void sdb::line_breakpoint::resolve() {
    auto& dwarf = target_->get_elf().get_dwarf();
    for (auto& cu : dwarf.compile_units()) {
        auto entries = cu->lines().get_entries_by_line(file_, line_);
```

```
        for (auto entry : entries) {
            auto& dwarf = entry->address.elf_file()->get_dwarf(); ❶
            auto stack = dwarf.inline_stack_at_address(entry->address);

            auto no_inline_stack = stack.size() == 1; ❷
            auto should_skip_prologue = no_inline_stack and ❸
                (stack[0].contains(DW_AT_ranges) or stack[0].contains(DW_AT_low_pc)) and
                stack[0].low_pc() == entry->address;
            if (should_skip_prologue) {
                ++entry; ❹
            }
            auto load_address = entry->address.to_virt_addr();
            if (!breakpoint_sites_.contains_address(load_address)) {
            auto& new_site = target_->get_process()
                .create_breakpoint_site(
                    this, next_site_id_++, load_address, is_hardware_, is_internal_);

                breakpoint_sites_.push(&new_site);
                if (is_enabled_) new_site.enable();
            }
        }
    }
}
```

The line we're looking for could be in any compile unit, so we loop over the compile units to which we have access. Multiple entries could be associated with a single line of code (if the function is inlined somewhere, for example), so we resolve breakpoint sites for all entries. Note that we grab the DWARF file from the line table entry rather than using the one we got from the target ❶; this is to support shared libraries in Chapter 17. If the given line exists inside a non-inlined function ❷, we may need to skip over the prologue. We do so as long as we find no inline stack, the function DIE has address range information, and the address for the line table entry we found is at the start of the function ❸. As for function name breakpoints, we skip the prologue by selecting the line table entry after the one for the start of the function ❹. Once we've found the correct line table entry, we compute the load address for the relevant file address and create a new breakpoint site if one doesn't already exist.

Creating Breakpoints

Now that all of our breakpoint types can resolve addresses, we need the ability to create them through an `sdb::target`. In the same way that `sdb::process` manages breakpoint sites, `sdb::target` should manage source-level breakpoints.

Add a collection of breakpoints to `sdb::target`, along with functions to create each type of breakpoint, to *sdb/include/libsdb/target.hpp*:

```
#include <libsdb/breakpoint.hpp>

namespace sdb {
    class target {
    public:
        --snip--
        breakpoint& create_address_breakpoint(
            virt_addr address,
            bool hardware = false, bool internal = false);
        breakpoint& create_function_breakpoint(
            std::string function_name,
            bool hardware = false, bool internal = false);
        breakpoint& create_line_breakpoint(
            std::filesystem::path file, std::size_t line,
            bool hardware = false, bool internal = false);


        stoppoint_collection<breakpoint>&
        breakpoints() { return breakpoints_; }
        const stoppoint_collection<breakpoint>&
        breakpoints() const { return breakpoints_; }

    private:
        --snip--
        stoppoint_collection<breakpoint> breakpoints_;
    };
}
```

We include the header for `sdb::breakpoint`. The `create_*_breakpoint` functions will create a breakpoint of the requested type and add it to the set of managed breakpoints for this `target`. We also add member functions for retrieving the breakpoint collection. Implement the breakpoint creation functions in *sdb/src/target.cpp*:

```
sdb::breakpoint&
sdb::target::create_address_breakpoint(
      virt_addr address, bool hardware, bool internal) {
    return breakpoints_.push(
        std::unique_ptr<address_breakpoint>(
            new address_breakpoint(
                *this, address, hardware, internal)));
}

sdb::breakpoint&
sdb::target::create_function_breakpoint(
```

```
      std::string function_name, bool hardware, bool internal) {
    return breakpoints_.push(
        std::unique_ptr<function_breakpoint>(
            new function_breakpoint(
                *this, function_name, hardware, internal)));
}

sdb::breakpoint&
sdb::target::create_line_breakpoint(
      std::filesystem::path file, std::size_t line,
      bool hardware, bool internal) {
    return breakpoints_.push(
        std::unique_ptr<line_breakpoint>(
            new line_breakpoint(
                *this, file, line, hardware, internal)));
}
```

You shouldn't see anything surprising in these functions; they just create the requested breakpoint types wrapped in a `std::unique_ptr` and push them into the stored breakpoint collection. We've fully supported breakpoints! Let's expose these to the user.

Exposing Breakpoints to the User

We've already written commands that debugger users can run to interact with breakpoints, but so far, these commands have operated on breakpoint sites. We must change them so they operate on breakpoints instead so that the user can create source-level breakpoints.

In *sdb/tools/sdb.cpp*, modify `handle_command` so it passes the `target` rather than the `process` to `handle_breakpoint`:

```
namespace {
    void handle_command(std::unique_ptr<sdb::target>& target,
        std::string_view line) {
        --snip--
        else if (is_prefix(command, "breakpoint")) {
            handle_breakpoint_command(*target, args);
        }
        --snip--
    }
}
```

Modify the `handle_breakpoint_command` signature to take an `sdb::target&`:

```
void handle_breakpoint_command(sdb::target& target,
      const std::vector<std::string>& args) {
    --snip--
}
```

Now we'll update the `list` command so that it produces output like the following:

```
sdb> break set 0x5555555551ac
sdb> break set main
sdb> break set force_inline.cpp:15
sdb> break list
Current breakpoints:
1: address = 0x5555555551ac, enabled:
    .1: address = 0x5555555551ac, enabled
2: function = main, enabled:
    .2: address = 0x5555555551a2, enabled
3: file = force_inline.cpp, line = 15, enabled:
    .3: address = 0x555555555197, enabled
    .4: address = 0x5555555551b3, enabled
```

The `list` command should print each breakpoint, as well as the list of breakpoint sites resolved for that breakpoint.

The code we must write is longer than ideal for placing inside the existing `handle_breakpoint_command` function, so we'll implement it inside a new one, `handle_breakpoint_list_command`:

```
namespace {
    void handle_breakpoint_list_command(sdb::target& target) {
        if (target.breakpoints().empty()) { ❶
            fmt::print("No breakpoints set\n");
        }
        else {
            fmt::print("Current breakpoints:\n");
            target.breakpoints().for_each([](auto& bp) { ❷
                if (bp.is_internal()) return;
                fmt::print("{}: ", bp.id());
                if (auto func_bp = dynamic_cast<sdb::function_breakpoint*>(&bp)) { ❸
                    fmt::print("function = {}", func_bp->function_name());
                }
                else if (auto line_bp = dynamic_cast<sdb::line_breakpoint*>(&bp)) { ❹
                    fmt::print("file = {}, line = {}",
                        line_bp->file().string(), line_bp->line());
                }
                else if (auto addr_bp = dynamic_cast<sdb::address_breakpoint*>(&bp)) { ❺
                    fmt::print("address = {:#x}", addr_bp->address().addr());
                }
                fmt::print(", {}:\n", bp.is_enabled() ? "enabled" : "disabled");
                bp.breakpoint_sites().for_each([&](auto& site) { ❻
                    fmt::print("    .{}: address = {:#x}, {}\n",
                    site.id(), site.address().addr(),
                    site.is_enabled() ? "enabled" : "disabled");
                    });
```

```
            });
        }
    }
}
```

If no breakpoints are set ❶, we print a message to inform the user of this fact. Otherwise, we print details about each non-internal breakpoint ❷, starting with its ID. We then use `dynamic_cast` to work out the breakpoint's dynamic type. When we dynamically cast a pointer to an `sdb::breakpoint*` into a derived type, it should return either a pointer to the derived type or `nullptr`, if the dynamic type isn't the one we requested.

If the breakpoint is a function breakpoint ❸, we print the function name. If it's a line breakpoint ❹, we print the file and the line. If it's an address breakpoint ❺, we print the address in hexadecimal. After handling the derived breakpoint type, we print an indication of whether the breakpoint is enabled. Finally, for each breakpoint site resolved by that breakpoint ❻, we print its ID, address, and whether it's enabled or disabled. We prefix the ID with a dot; users will be able to use this value to refer to individual breakpoint sites when using the `enable`, `disable`, and `delete` commands.

Next is the `set` command. We should support three forms for this command: `break set` *<function name>*, `break set` *<file>*`:`*<line>*, and `break set 0x`*<address>*. Implement the code inside a new `handle_breakpoint_set_command` function:

```
namespace {
    void handle_breakpoint_set_command(
          sdb::target& target, const std::vector<std::string>& args) {
        bool hardware = false;
        if (args.size() == 4) {
            if (args[3] == "-h") hardware = true;
            else sdb::error::send("Invalid breakpoint command argument");
        }

     ❶ if (args[2].find("0x") == 0) {
            auto address = to_integral<std::uint64_t>(args[2], 16);

            if (!address) {
                fmt::print(stderr,
                    "Breakpoint command expects address in "
                    "hexadecimal, prefixed with '0x'\n");
                return;
            }

            target.create_address_breakpoint(
                sdb::virt_addr{ *address }, hardware).enable();
        }
```

```
    ❷ else if (args[2].find(':') != std::string::npos) {
          auto data = split(args[2], ':');
          auto path = data[0];
          auto line = to_integral<std::uint64_t>(data[1]);
          if (!line) {
              fmt::print(stderr,
                  "Line number should be an integer\n");
              return;
          }
          target.create_line_breakpoint(path, *line, hardware).enable();
      }
    ❸ else {
          target.create_function_breakpoint(args[2]).enable();
      }
  }
}
```

We start by recording whether the user specified the `-h` flag to set a hardware breakpoint. If the subcommand argument's first two characters are `0x` ❶, this must be an address breakpoint. We attempt to convert this hexadecimal string to an integer. If the conversion fails, we print an error. Otherwise, we create an address breakpoint and enable it.

If the argument doesn't start with `0x`, but it does contain a colon (`:`), it must be a line number breakpoint ❷. We split the argument on the colon, convert the line number to an integer, and then create the breakpoint and enable it. If the line number conversion fails, we report it to the user. You could also check if that file actually exists, but I omitted this check for simplicity.

If the breakpoint isn't an address or line breakpoint, it must be a function breakpoint ❸, so we pass the argument to `create_function_breakpoint`. Again, you might want to check whether this function actually exists.

Update the existing `list` and `set` subcommand branches to call the new functions we just wrote:

```
    --snip--
    if (is_prefix(command, "list")) {
        handle_breakpoint_list_command(target);
        return;
    }
    --snip--
    if (is_prefix(command, "set")) {
        handle_breakpoint_set_command(target, args);
        return;
    }
    --snip--
```

Replace the current contents of those branches with calls to the relevant functions.

Now we'll tackle the `enable`, `disable`, and `delete` commands. We'd like the `enable` and `disable` commands to apply to both breakpoints and breakpoint sites, allowing users to disable specific breakpoint sites as well as entire breakpoints. For example, a user could enter `break enable 1` to enable all the breakpoint sites for the breakpoint with an ID of 1, or they could enter `break enable 1.2` to enable only the breakpoint site with an ID of 2 inside the breakpoint with an ID of 1. We won't support `delete` for breakpoint sites, however, because this would require notifying the parent breakpoint and performing additional bookkeeping for breakpoints that get resolved multiple times (which will happen when we implement shared library support in Chapter 17).

We'll introduce a new function that deals with these commands, as they involve a fair amount of code. Replace the rest of the existing code in `handle_breakpoint_command` with a call to a function named `handle_breakpoint_toggle`, which we'll implement next:

```
    --snip--
    handle_breakpoint_toggle(target, args);
```

Implement `handle_breakpoint_toggle` in the same file:

```
namespace {
    void handle_breakpoint_toggle(
          sdb::target& target,
          const std::vector<std::string>& args) {
        auto command = args[1];

        auto dot_pos = args[2].find('.');
        auto id_str = args[2].substr(0, dot_pos);
        auto id = to_integral<sdb::breakpoint::id_type>(id_str);
        if (!id) {
            std::cerr << "Command expects breakpoint id";
            return;
        }
        auto& bp = target.breakpoints().get_by_id(*id);

        if (dot_pos != std::string::npos) {
            auto site_id_str = args[2].substr(dot_pos + 1);
            auto site_id = to_integral<sdb::breakpoint_site::id_type>(site_id_str);
            if (!site_id) {
                std::cerr << "Command expects breakpoint site id";
                return;
            }
            if (is_prefix(command, "enable")) {
                bp.breakpoint_sites().get_by_id(*site_id).enable();
            }
            else if (is_prefix(command, "disable")) {
                bp.breakpoint_sites().get_by_id(*site_id).disable();
```

```
            }
        }
        else if (is_prefix(command, "enable")) {
            bp.enable();
        }
        else if (is_prefix(command, "disable")) {
            bp.disable();
        }
        else if (is_prefix(command, "delete")) {
            bp.breakpoint_sites().for_each([&](auto& site) {
                target.get_process().breakpoint_sites().remove_by_address(site.address());
            });
            target.breakpoints().remove_by_id(*id);
        }
    }
}
```

First, we grab the breakpoint subcommand from the supplied arguments. We look for a dot character in the given ID, extract the breakpoint ID from it, print an error if this failed, and then find the corresponding breakpoint. If the dot position is not `std::string::npos`, the supplied ID must be for a breakpoint site. We extract the breakpoint site ID (the part after the dot) and print an error if this fails. If the command was `enable`, we enable the breakpoint site. If the command was `disable`, we disable the site. If there was no dot character, we should operate on the breakpoint itself. Depending on what the supplied command was, we either enable, disable, or delete the breakpoint. When we delete breakpoints, we also remove all their breakpoint sites from the `sdb::process`.

Users of the debugger can now interact with source-level breakpoints! Before we test breakpoints and stepping, let's augment the debugger's user interface so it can print out the source code currently being executed.

Printing Currently Executing Source Code

Printing the disassembly for the current code is fine, but it's much easier for users to debug their code if they can see the source code that corresponds to the instructions being executed. Fortunately, with the line table and inlining support we've implemented, we can easily achieve this.

At the moment, `handle_stop` calls `print_disassembly` after it prints the stop reason. We'll add a `print_source` function and call it instead of `print_disassembly` to print the source code if we have enough information to do so. This information will come from one of two places. If the debugger is currently simulating regular function calls based on an inline stack, we'll query the inline stack for the relevant file and line. Otherwise, we'll look up the line table entry corresponding to the current program counter value. In *src/tools/sdb.cpp*, modify `handle_stop` like so:

```
namespace {
    void handle_stop(sdb::target& target, sdb::stop_reason reason) {
        print_stop_reason(target, reason);
        if (reason.reason == sdb::process_state::stopped) {
            if (target.get_stack().inline_height() > 0) { ❶
                auto stack = target.get_stack().inline_stack_at_pc();
                auto frame = stack[stack.size() - target.get_stack().inline_height()];
                print_source(frame.file().path, frame.line(), 3);
            }
            else if (auto entry = target.line_entry_at_pc(); ❷
                entry != sdb::line_table::iterator()) {
                print_source(entry->file_entry->path, entry->line, 3);
            }
            else { ❸
                print_disassembly(
                    target.get_process(), target.get_process().get_pc(), 5);
            }
        }
    }
}
```

As before, we first print information about the stop reason we received. If the process stopped due to a signal, we print the current execution location. If the inline height is greater than 0, we're currently simulating regular function calls ❶, so we print the source information for the current frame. We retrieve the current inline stack and then locate the frame below the current one. This frame's file and line information will point to the exact source location at which the function was inlined. We then print the source code around that location. The third argument to `print_source` is the number of lines of context to print. I chose three, as it gives the user a reasonable amount of context without flooding their console space.

If we're not currently simulating non-inline function calls ❷, we get the line entry corresponding to the current program counter value. If we don't receive the end iterator, this entry is valid, so we print the source code around that location. Otherwise ❸, we don't have enough information to print source code, so we print disassembly instead.

Now we can implement `print_source`. The function is fairly long, so we'll implement it step by step. Begin it in *sdb/tools/sdb.cpp*:

```
#include <fstream>
#include <filesystem>
#include <cmath>

namespace {
    void print_source(
          const std::filesystem::path& path, std::uint64_t line,
        std::uint64_t n_lines_context) {
        std::ifstream file{ path.string() };
```

```
auto start_line = line <= n_lines_context ? 1 : line - n_lines_context;
auto end_line = line + n_lines_context + 1;
```

We start by creating a `std::ifstream` from the given path, which opens the file and readies it for reading. We then calculate the start and end line numbers to print based on the given line and amount of context requested. We assign source code lines numbers starting at 1. The start line should never be smaller than 1, so if the line at which to start is less than or equal to the number of lines of context to give, we simply set the start line to 1. For the end line, we'll use the line one after the current line, plus the requested lines of context. Later, we'll deal with the possibility that the end line falls outside of the file's range.

After determining the start and end lines, we discard any source code that comes before the starting line we're looking for:

```
--snip--
char c{};
auto current_line = 1u;
while (current_line != start_line && file.get(c)) {
    if (c == '\n') {
        ++current_line;
    }
}
```

We loop, continually reading characters. Every time we get a newline character, we increment the current line. When the current line is equal to the desired start line, we terminate the loop because we're ready to print the source.

We'd like to print the line number before each line of source code, as well as a small arrow (>) at the beginning of the currently executing line. We'll need to repeat this process in a couple of places, so we factor the code to do so into a lambda function:

```
--snip--
auto print_line_start = [&](auto current_line) {
    auto fill_width = static_cast<int>(
        std::floor(std::log10(end_line))) + 1;
    auto arrow = current_line == line ? ">" : " ";
    fmt::print("{} {:>{}} ", arrow, current_line, fill_width);
};
```

The calculations here are a bit nuanced. The line numbers should all be the same width so they don't misalign the source code. For example, the output should look like this:

```
 8  pet_cat();
 9  put_cat_to_bed();
10  contemplate_life();
```

We'd like to avoid the following, which improperly indents the code:

```
8  pet_cat();
9  put_cat_to_bed();
10  contemplate_life();
```

We shouldn't set the width of the line numbers arbitrarily, however. If we end up printing a source file that is 10,000 lines long and align the numbers to a width of four characters, the code will end up misaligned. On the other hand, if we decide to align the numbers to a width of six characters but only print source files of up to 100 lines, we've wasted a lot of space.

So, we calculate a fill width by figuring out the length in characters of the end line when formatted as a string. The `std::floor(std::log10(end_line)) + 1` expression calculates this. If you're not familiar with logarithms used this way, you can think of it as "How many times can you divide this number by 10?"

If the current line in the source code is equal to the current line supplied to `print_source`, we should output an arrow; otherwise, we output a blank space. Next, we use a nested format specifier to generate the text. In `"{} {:>{}} "`, an arrow or blank space will replace the first pair of braces, the fill width will replace the inner set of the second pair, and the current line number will replace the outer set of the second pair, along with the necessary whitespace. This specifier should correctly format the line number, padded to a sensible width.

With this helper function defined, we can print out the source:

```
        --snip--
        print_line_start(current_line);
        while (current_line <= end_line && file.get(c)) {
            std::cout << c;
            if (c == '\n') {
                ++current_line;
                print_line_start(current_line);
            }
        }

        std::cout << std::endl;
    }
}
```

We print the line start information for the current line. Then, we continue reading characters until either we reach the end line we computed earlier or nothing remains in the file. We output each character we read. If we read a newline character, we increment the current line and output the line start information for the new line. Finally, we output another newline character and flush the stream with `std::endl` so it's immediately output to the console.

Lastly, we print the filename, line number, and source code function inside `print_stop_reason`. The message should look something like this:

```
Process 2544 stopped with signal TRAP at 0x5555555551a2,
force_inline.cpp:4 (main) (breakpoint 1)
```

Let's add a `function_name_at_address` function to `sdb::target` that retrieves the function name for a given program counter value. Declare it in *sdb/include/libsdb/target.hpp*:

```
namespace sdb {
    class target {
    public:
        --snip--
        std::string function_name_at_address(
            virt_addr address) const;
        --snip--
    };
}
```

We return a `std::string` instead of a `std::string_view` because in Chapter 17, we'll additionally qualify the function name with the ELF file in which it's defined, so we can't just return a view of an existing string. Implement the function in *sdb/src/target.cpp*:

```
#include <cxxabi.h>

std::string sdb::target::function_name_at_address(
      virt_addr address) const {
    auto file_address = address.to_file_addr(*elf_);
    auto obj = file_address.elf_file();
    if (!obj) return "";

    auto func = obj->get_dwarf().function_containing_address(file_address);
    if (func and func->name()) {
        return std::string{*func->name()};
    }
    else if (auto elf_func = obj->get_symbol_containing_address(file_address);
        elf_func and ELF64_ST_TYPE(elf_func.value()->st_info) == STT_FUNC) {
        auto elf_name = std::string{ obj->get_string(elf_func.value()->st_name) };
        return abi::__cxa_demangle(elf_name.c_str(), nullptr, nullptr, nullptr);
    }
    return "";
}
```

We attempt to find the function DIE containing the program counter as a file address. If that DIE exists and has a name attribute, we return the name. Otherwise, we follow the same process we use in `get_signal_stop_reason`; we look for the ELF symbol containing the current program counter as an

offset, and if we find one that is a function symbol, we demangle it and return it. If we don't find a relevant ELF symbol, we don't have enough information to get the function name, so we return an empty string.

Now we can use this function in `get_signal_stop_reason`. Modify the existing code in *sdb/tools/sdb.cpp*:

```
namespace {
    std::string get_signal_stop_reason(
          const sdb::target& target, sdb::stop_reason reason) {
        auto& process = target.get_process();
        auto pc = process.get_pc();
        std::string message = fmt::format("stopped with signal {} at {:#x}",
            sigabbrev_np(reason.info), pc.addr());

        auto line = target.line_entry_at_pc();
        if (line != sdb::line_table::iterator()) {
            auto file = line->file_entry->path.filename().string();
            message += fmt::format(", {}:{}", file, line->line);
        }

        auto func_name = target.function_name_at_address(pc);
        if (func_name != "") {
            message += fmt::format(" ({})", func_name);
        }

        if (reason.info == SIGTRAP) {
            message += get_sigtrap_info(process, reason);
        }

        return message;
    }
}
```

We add a `pc` variable that will enable us to call `get_pc` only once. The "stopped with signal . . ." message remains the same. After computing this message, we find the line table entry at the current program counter value. If we don't get an empty iterator, we append the file and line number from that entry to the message.

Next, we call the `function_name_at_address` function we just wrote to retrieve the function name. We add the function name to the message, so long as we received one. Finally, we append the `SIGTRAP` information to the message, as we used to, and then break. Stopping at a breakpoint should now print out source information, like this:

```
sdb> break set main
sdb> c
Process 2897 stopped with signal TRAP at 0x5555555551a2,
force_inline.cpp:4 (main) (breakpoint 1)
```

```
  15 }
  16
  17 int main() {
> 18     c();
  19 }
sdb>
```

Our tool is starting to behave like a real source-level debugger! Before moving on to stack unwinding, let's test the code.

Testing

We'll test the code in the opposite order of that in which we wrote the features. This is because function breakpoints will make it easier to test the stepping routines.

Breakpoints

To test source-level breakpoints, let's write a short program involving overloaded functions. Create a new file at *sdb/test/targets/overloaded.cpp* with these contents:

```
#include <string>
#include <iostream>

void print_type(int) {
    std::cout << "int";
}

void print_type(double) {
    std::cout << "double";
}

void print_type(std::string) {
    std::cout << "string";
}

int main() {
    print_type(0);
    print_type(1.4);
    print_type("hello");
}
```

We write three overloads of `print_type` that print the type of the argument with which they were called. Add a target for this program to *sdb/test/targets/CMakeLists.txt*:

```
add_test_cpp_target(overloaded)
```

Let's run some manual tests first and then automate them. Launch *overloaded* with *sdb*, set a breakpoint on line 17 of *overloaded.cpp*, and continue. The process should stop on the specified line:

```
sdb> break set overloaded.cpp:17
sdb> c
Process 438 stopped with signal TRAP at 0x5555555564a9,
overloaded.cpp:17 (main) (breakpoint 1)
  14 }
  15
  16 int main() {
> 17     print_type(0);
  18     print_type(1.4);
  19     print_type("hello");
  20 }
```

Set a breakpoint on `print_type`, list the breakpoints, and disable the breakpoint site for the overload with the lowest address (likely the `int` overload). You should have created three breakpoint sites, and the disabling should work:

```
sdb> break set print_type
sdb> break list
Current breakpoints:
1: file = overloaded.cpp, line = 17, enabled:
    .1: address = 0x5555555564a9, enabled
2: function = print_type, enabled:
    .1: address = 0x555555556418, enabled
    .2: address = 0x555555556445, enabled
    .3: address = 0x555555556471, enabled
sdb> break disable 2.1
sdb> break list
Current breakpoints:
1: file = overloaded.cpp, line = 17, enabled:
    .1: address = 0x5555555564a9, enabled
2: function = print_type, enabled:
    .1: address = 0x555555556418, disabled
    .2: address = 0x555555556445, enabled
    .3: address = 0x555555556471, enabled
```

Continue the process three times. The program should halt at two different overloads of `print_type` and then terminate:

```
sdb> c
Process 438 stopped with signal TRAP at 0x555555556445,
overloaded.cpp:9 (print_type) (breakpoint 3)
   6 }
   7
   8 void print_type(double) {
```

```
>  9     std::cout << "double";
  10 }
  11
  12 void print_type(std::string) {
  13     std::cout << "string";
  14
sdb> c
Process 438 stopped with signal TRAP at 0x555555556471,
overloaded.cpp:13 (print_type) (breakpoint 4)
  10 }
  11
  12 void print_type(std::string) {
> 13     std::cout << "string";
  14 }
  15
  16 int main() {
  17     print_type(0);
  18
sdb> c
intdoublestringProcess 438 exited with status 0
```

Now let's automate this test. Create a new test case in *sdb/test/tests.cpp*:

```
#include <libsdb/target.hpp>

TEST_CASE("Source-level breakpoints", "[breakpoint]") {
    auto dev_null = open("/dev/null", O_WRONLY);
    auto target = target::launch("targets/overloaded", dev_null);
    auto& proc = target->get_process();

    target->create_line_breakpoint("overloaded.cpp", 17).enable();

    proc.resume();
    proc.wait_on_signal();

    auto entry = target->line_entry_at_pc();
    REQUIRE(entry->file_entry->path.filename() == "overloaded.cpp");
    REQUIRE(entry->line == 17);

    auto& bkpt = target->create_function_breakpoint("print_type");
    bkpt.enable();

    sdb::breakpoint_site* lowest_bkpt = nullptr;
    bkpt.breakpoint_sites().for_each([&lowest_bkpt](auto& site) {
        if (lowest_bkpt == nullptr or site.address().addr() < lowest_bkpt->address().addr()) {
            lowest_bkpt = &site;
        }
    });
```

```
    lowest_bkpt->disable();

    proc.resume();
    proc.wait_on_signal();

    REQUIRE(target->line_entry_at_pc()->line == 9);

    proc.resume();
    proc.wait_on_signal();

    REQUIRE(target->line_entry_at_pc()->line == 13);

    proc.resume();
    auto reason = proc.wait_on_signal();

    REQUIRE(reason.reason == sdb::process_state::exited);
    close(dev_null);
}
```

We launch the target with `stdout` redirected to */dev/null* and create a breakpoint on line 17 of *overloaded.cpp*. After resuming the process and waiting for it to halt, we ensure that we stopped at the expected location. We then create a breakpoint on the overloaded `print_type` function and disable the breakpoint site with the lowest address, which will most likely be the overload for `int`, as it comes first in the source code. To achieve this, we initialize a pointer to a breakpoint site and then loop over all breakpoint sites that belong to the breakpoint, finding the one with the lowest address and storing it in the pointer. Once the loop terminates, we disable the found breakpoint site. We then resume the process three times, ensuring that we stop at lines 9 and 13 and that the process then exits. Finally, we close the file handle for */dev/null*. This test should pass. That will do for breakpoints. Let's move on to stepping.

Stepping

We'd like to test all three of our stepping functions for both inlined functions and non-inlined functions. To do so, we'll write a program whose call chain is a few functions deep and that includes inlining. Create a new file at *sdb/test/targets/step.cpp* with these contents:

```
#include <cstdio>

__attribute__((always_inline))
inline void scratch_ears() {
    std::puts("Scratching ears");
}

__attribute__((always_inline))
```

```
inline void pet_cat() {
    scratch_ears();
    std::puts("Done petting cat");
}

void find_happiness() {
    pet_cat();
    std::puts("Found happiness");
}

int main() {
    find_happiness();
    find_happiness();
}
```

Add a target for the file to *sdb/test/targets/CMakeLists.txt*:

```
add_test_cpp_target(step)
```

Once again, we'll run some manual tests and then automate them. Launch the program with *sdb*, set a breakpoint on `main`, and continue. The debugger should hit the breakpoint:

```
Launched process with PID 1437
sdb> break set main
sdb> c
Process 1437 stopped with signal TRAP at 0x5555555551cf,
step.cpp:20 (main) (breakpoint 1)
  17 }
  18
  19 int main() {
> 20     find_happiness();
  21     find_happiness();
  22 }
```

Step over the first function call and into the second one. Execution should halt on the first call inside `find_happiness`:

```
sdb> next
Scratching ears
Done petting cat
Found happiness
Process 1437 stopped with signal TRAP at 0x5555555551d4,
step.cpp:21 (main) (single step)
  18
  19 int main() {
  20     find_happiness();
> 21     find_happiness();
  22 }
```

```
sdb> step
Process 1437 stopped with signal TRAP at 0x555555555195,
step.cpp:5 (find_happiness) (single step)
  12 }
  13
  14 void find_happiness() {
> 15     pet_cat();
  16     std::puts("Found happiness");
  17 }
  18
  19 int main() {
  20
```

Step into `pet_cat`. Because this call was inlined, the program counter shouldn't change, but the source location should:

```
sdb> step
Process 1437 stopped with signal TRAP at 0x555555555195,
step.cpp:5 (find_happiness) (single step)
   7
   8 __attribute__((always_inline))
   9 void pet_cat() {
> 10     scratch_ears();
  11     std::puts("Done petting cat");
  12 }
  13
  14 void find_happiness() {
  15
```

Step over the inlined `scratch_ears` call, out of the `pet_cat` function, and out of the `find_happiness` function. Execution should stop at the end of `main`:

```
sdb> next
Scratching ears
Process 1437 stopped with signal TRAP at 0x5555555551a5,
step.cpp:11 (find_happiness) (single step)
   8 __attribute__((always_inline))
   9 void pet_cat() {
  10     scratch_ears();
> 11     std::puts("Done petting cat");
  12 }
  13
  14 void find_happiness() {
  15     pet_cat();
  16
sdb> finish
Done petting cat
Process 1437 stopped with signal TRAP at 0x5555555551b5,
step.cpp:16 (find_happiness) (single step)
```

```
  13
  14 void find_happiness() {
  15     pet_cat();
> 16     std::puts("Found happiness");
  17 }
  18
  19 int main() {
  20     find_happiness();
  21
sdb> finish
Found happiness
Process 1437 stopped with signal TRAP at 0x5555555551d9,
step.cpp:22 (main) (single step)
  19 int main() {
  20     find_happiness();
  21     find_happiness();
> 22 }
```

If the tests worked as expected, you've successfully implemented source-level stepping, and you can convert this manual test into an automated one. Add a new test case to *sdb/test/tests.cpp*:

```
TEST_CASE("Source-level stepping", "[target]") {
    auto dev_null = open("/dev/null", O_WRONLY);
    auto target = target::launch("targets/step", dev_null);
    auto& proc = target->get_process();

    target->create_function_breakpoint("main").enable();
    proc.resume();
    proc.wait_on_signal();

    auto pc = proc.get_pc();
    REQUIRE(target->function_name_at_address(pc) == "main");

    target->step_over();

    auto new_pc = proc.get_pc();
    REQUIRE(new_pc != pc);
    REQUIRE(target->function_name_at_address(pc) == "main");

    target->step_in();

    pc = proc.get_pc();
    REQUIRE(target->function_name_at_address(pc) == "find_happiness");
    REQUIRE(target->get_stack().inline_height() == 2);

    target->step_in();
```

```
    new_pc = proc.get_pc();
    REQUIRE(new_pc == pc);
    REQUIRE(target->get_stack().inline_height() == 1);

    target->step_out();

    new_pc = proc.get_pc();
    REQUIRE(new_pc != pc);
    REQUIRE(target->function_name_at_address(pc) == "find_happiness");

    target->step_out();

    pc = proc.get_pc();
    REQUIRE(target->function_name_at_address(pc) == "main");
    close(dev_null);
}
```

This code essentially walks through the manual test we ran previously. We launch the target, set a breakpoint on `main`, and resume the process. When the process halts, we ensure that we're inside the `main` function. We step over the first call to `find_happiness` and ensure that the program counter has changed but that we're still inside `main`. We then step into `find_happiness`, make sure the debugger reports the correct function, and ensure the inline height is 2 (because we're also at the top of the inlined `pet_cat` and `scratch_head` functions).

Next, we step into the inlined `pet_cat` function and ensure that the program counter hasn't changed but that the inline height has gone down. Checking the function name at that offset wouldn't return `pet_cat` because `sdb::dwarf` reports the outermost non-inlined function for offset queries where there are inlined frames, so we don't bother checking it. We step out of `pet_cat`, ensure that the program counter has changed and that we're still inside `find_happiness`, and then step into `main`. This test should pass.

Summary

In this chapter, you learned how function inlining complicates debugging and then devised a way to create the illusion that there is no inlining. You implemented routines to step into, over, and out of function calls and to set breakpoints at the level of source code rather than in the machine code. You also added the ability to print out the currently executing source code line.

In the next chapter, you'll begin implementing a full stack unwinder and augment your debugger with the ability to produce backtraces and restore registers to their values from before the current function was called.

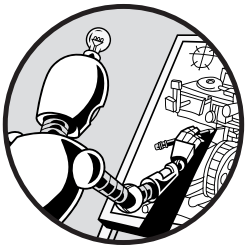
Check Your Knowledge

1. What is function inlining?
2. Which DWARF tag is used to represent an inlined call?
3. Which register sometimes stores the stack frame base for the current function? Why can't we always rely on this register?

15

CALL FRAME INFORMATION

The tower flows endlessly
down through the depths
and we search for the light
that marks its home.



In this chapter, you'll learn about DWARF call frame information, which will enable you to implement the *backtrace*, one of a debugger's most useful tools for finding issues in a program. For any state, it shows the sequence of function calls the program used to arrive there, which is invaluable for determining the actions leading to a bug.

We generate a backtrace by *unwinding* the function call stack: finding the current function's return address, locating the function to which that address belongs, finding the return address of that function, and so on. Unwinding the stack also lets us restore the state of saved registers so the user can examine the values of variables in functions higher up the call stack while the program is stopped inside of a callee.

We'll write parsers for key elements of the call frame information, which will provide the information needed to unwind the stack in the next chapter.

A Backtrace Example

To understand why backtraces are helpful, consider the following program:

```
#include <cstdio>

void say_hello() {
    puts("Hello");
    puts("My name is Marshmallow");
}

void say_hello_twice() {
 ❶ say_hello();
 ❷ say_hello();
}
int main() {
    say_hello_twice();
 ❸ say_hello();
}
```

If we set a breakpoint on the `say_hello` function and keep resuming the program, we'll hit the breakpoint three times: once inside each of the two calls in `say_hello_twice` ❶ ❷, and once inside the call in `main` ❸. When stopped at those three points in the program execution, a debugger may generate the following three backtraces:

```
*[0]: 0x555555555151 say_hello
 [1]: 0x55555555517a say_hello_twice
 [2]: 0x55555555518f main

*[0]: 0x555555555151 say_hello
 [1]: 0x55555555517f say_hello_twice
 [2]: 0x55555555518f main

*[0]: 0x555555555151 say_hello
 [1]: 0x555555555194 main
```

Each line of the backtrace represents a function in the current call stack, numbered beginning with 0. The hexadecimal values are the value of the program counter for each stack frame.

The first two backtraces have three functions in the call stack, because `main` called `say_hello_twice`, which called `say_hello`. In the third case, `main` called `say_hello` directly.

Note that the only difference between the first two backtraces is the value of the program counter inside of `say_hello_twice`. This difference occurs because there are two call instructions in the generated code for each of the two function calls. If we look at the generated assembly, we can see this fact more clearly:

```
say_hello_twice:
    endbr64
    push   %rbp
    mov    %rsp,%rbp
❶   call   say_hello
❷   call   say_hello
    nop
    pop    %rbp
    ret
```

In the first backtrace, the program counter points to the first call instruction ❶. In the second backtrace, it points to the second ❷.

Let's look at how DWARF encodes information about stack unwinding.

DWARF Call Frame Information

Recall from Chapter 2 that the call stack includes a frame to represent each function call. To unwind a single frame of the stack, we must know how to locate the *canonical frame address (CFA)*, which is a specific point in the stack frame that we use as the base address for other computations. We must also know where the return address is stored and how to restore any registers whose values were saved onto the stack as part of the callee's function prologue.

We can obtain these details using DWARF call frame information, which essentially defines a huge table. Each row of the table lists a program counter value, and the rules in that row apply to the instructions ranging from that address up to, but not including, the program counter value in the next table row (or the end of the function, in the case of the last row in the table). Table rows encode two things: the CFA for the start of the current stack frame, and rules for restoring registers to the values that they would have if the current function immediately returned to its caller.

Because storing the entire call frame information table would be infeasible due to its size, object files encode it as a space-efficient binary program for the DWARF parser to run, much like the line table program we interpreted in Chapter 13.

Unfortunately, call frame information is architecture-specific, so the DWARF specification alone isn't sufficient to describe how to unwind the stack for a given platform. Instead, we must consult the platform's ABI. For instance, DWARF specifies that an object file should store its call frame information in the `.debug_frame` section, but compilers targeting Linux and the SYSV ABI tend to leave that section blank, instead storing a similar (but slightly incompatible) format in the `.eh_frame` section. This format is specified in the SYSV ABI and extended by the Linux Standard Base (LSB) specification, which standardizes many features of Linux distributions. We'll focus on this format, which we'll call the EH format.

To complicate matters, the SYSV ABI and LSB documents are outdated and don't reflect what systems use today. They describe a format based on

DWARF 2, but more recent compilers sometimes output formats based on more recent versions of DWARF. Even worse, the version of call frame information they output is not tied to the version of the DWARF information they output outside of the `.eh_frame` section. For example, GCC may output DWARF 4 debug information but use an EH format based on DWARF 2 for call frame information. As such, we'll support call frame information based on DWARF 2, 3, and 4.

The call frame information contains *frame description entries (FDEs)*, which cover ranges of instructions. Usually, you'll find an entry for each function in the program. However, many entries contain the same pieces of information; for example, they may store the return address at the same offset from the CFA. To avoid repetition, the call frame information can place these common parts in *common information entries (CIEs)* to which multiple FDEs can link.

Together, CIEs and FDEs encode a series of call frame information instructions that describe how to compute rows of that huge table we just discussed. A DWARF parser can interpret these instructions for the entry containing the current program counter value to figure out how to unwind the current stack frame. By repeating this process using the address of the call instruction that led to the current function, the parser can unwind another stack frame, and so on, until it reaches the beginning of the program (which for the purposes of this chapter is the `main` function, but in reality is usually the `_start` function provided by your C++ toolchain).

Common Information Entries

In the EH format, CIEs consist of 10 fields:

`length` (`std::uint32_t`) The byte size of this CIE, not including the `length` field itself.

`CIE_id` (`std::uint32_t`) Distinguishes CIEs from FDEs. The DWARF standard specifies this field to be `0xffffffff` for CIEs, but the EH format specifies it to be `0`.

`version` (`std::uint8_t`) The version of the call frame information that this CIE is representing. For formats based on DWARF 2, this is `1`, for formats based on DWARF 3, it is `3`, and for formats based on DWARF 4, it is `4`.

`augmentation_string` (null-terminated string) ABI-specific augmentations. I'll detail the options that the EH format specifies momentarily.

`address_size` (`std::uint8_t`) The byte size of an address on the target machine. On x64, this is `8`. This field is present only in CIEs based on DWARF 4 and up.

`segment_size` (`std::uint8_t`) The byte size of segment selectors on the target machine. On x64, this is `0`. This field is present only in CIEs based on DWARF 4 and up.

`code_alignment_factor` (`ULEB128`) Tunes the behavior of certain CFI instructions.

`data_alignment_factor` (`SLEB128`) Tunes the behavior of certain CFI instructions.

`return_address_register` (`std::uint8_t` in DWARF 2, `ULEB128` otherwise) The DWARF register number for the register that stores the return address. On x64, this will be register 16, which is a made-up register number specifically for return addresses, but which we've assigned to `rip` so that restoring `rip` is like returning from a function.

`augmentation_data` (array of `std::uint8_t` values) Stores data that is outlined in the `augmentation_string` field. This field is present only in the EH format, not in DWARF.

`initial_instructions` (array of `std::uint8_t` values) A sequence of CFI instructions that specify how to unwind the stack and are prepended to all FDEs linked to this CIE.

The EH format can specify *augmentation data*, which provides platform- and language-specific information, and alternate data encoding schemes. The `augmentation_string` field describes the contents and order of the augmentation data, and the `augmentation_data` field contains the data itself. The `augmentation_string` field may have the following specifiers:

`z` Indicates that there is a `ULEB128` specifying the size of the augmentation data (not including the `ULEB128` itself). If there is augmentation data, this must be the first entry in it.

`L` Indicates that there is a byte specifying the encoding of a pointer to additional information for exception handling routines called the *language-specific data area (LSDA)*. We won't use this pointer in our debugger.

`R` Indicates that there is a byte specifying the encoding of FDE code pointers for linked FDEs.

`P` Indicates that there is a byte specifying the encoding of a pointer, followed by the encoded pointer itself, which indicates a personality function used to handle language-specific tasks by exception handling routines. We won't use this pointer in our debugger.

The pointer encoding bytes are constructed from the following values. This scheme is woefully underdocumented, and real implementations don't follow the specifications, so I've included the values that compilers actually output and noted how they deviate from the SYSV ABI and LSB:

`0x0` The value is stored as a pointer-sized integer (8 bytes on x64).

`00x1` The value is stored as an `LEB128`.

`00x2` The value is stored as a 2-byte integer.

`0x3` The value is stored as a 4-byte integer.

`0x4` The value is stored as an 8-byte integer.

`0x8` The value is signed.

`0x10` **(program counter relative)** The value is relative to the current parser position (not the current program counter, as specified in the SYSV ABI and LSB).

`0x20` **(`.text` section relative)** The value is relative to the start of the `.text` section. GCC doesn't generate this encoding for x64, but we'll implement it anyway.

`0x30` **(data section relative)** The trickiest value, as it's completely architecture-dependent and nearly undocumented. The value is relative to the start of the `.eh_frame_hdr` section if the pointer is in the `.eh_frame_hdr` section and to 0 otherwise (on x64). The LSB and SYSV ABI describe this value differently, and neither reflects what real tools output.

`0x40` **(function relative)** The value is relative to the start of the function to which the unwind information applies.

`0x50` **(aligned absolute)** The encoded pointer value has been padded to lie on a word-sized address boundary. Never used for x64; necessary only for platforms whose exception information needs updating at runtime and who don't support unaligned writes.

`0x80` **(indirect)** The value gives a memory address that stores the actual pointer value to use. On x64, this is only ever used to encode the pointer to the personality routine. This routine usually lives in the standard library, which is usually dynamically loaded, meaning the pointer needs updating at runtime. Because we don't use the personality routine in our debugger, we don't need to handle this encoding.

Fun fact: the indirect encoding doesn't come from the DWARF standard, the SYSV ABI, or the LSB; it's an undocumented GCC extension that the entire ecosystem now depends on because GCC is so ubiquitous.

Pointer encoding bytes are constructed by bitwise ORing the integer type identifier (bits 0, 1, and 2), the signedness (bit 3), and the relativity (the remaining bits).

Let's consider an augmentation example to make this information more concrete. If the `augmentation_string` field contains the string `zR`, the `augmentation_data` field contains a `ULEB128` that encodes the size of the rest of the augmentation data, followed by a byte that gives the encoding for code pointers in FDEs linked to this CIE. This byte could specify that pointers are encoded as signed `LEB128` values relative to the start of the `.text` section. In this case, the bit pattern would be `0x1 | 0x8 | 0x20 = 0b00101001`.

Frame Description Entries

FDEs consist of only 6 fields, compared to the 10 fields of CIEs:

`length` (`std::uint32_t`) The byte size of this FDE, not including the `length` field itself.

`CIE_id` (`std::int32_t`) The negative distance from the current parse position to the linked CIE. For example, a value of `40` means that the CIE starts at 40 bytes before the current parse position.

`initial_location` (encoded pointer) A pointer to the first instruction to which this FDE applies. The encoding for this pointer is given by the `R` augmentation of the linked CIE and defaults to 64-bit absolute values if there is no `R` augmentation.

`address_range` (encoded integer) The byte size of the code to which this FDE applies. The encoding for this pointer is given by the `R` augmentation of the linked CIE and defaults to 64-bit absolute values if there is no `R` augmentation. However, this value should always be interpreted as an absolute integer value, even if the `R` augmentation says it should be relative to some base address.

`augmentation_data` (array of `std::uint8_t`) Stores data that is outlined in the linked CIE's `augmentation_string` field. This field is present only in the EH format, not DWARF.

`instructions` (array of `std::uint8_t`) A sequence of call frame information instructions that specify how to unwind the stack.

To implement stack unwinding, we'll begin by parsing the call frame information, then write an interpreter for call frame information instructions that can unwind a single stack frame. We can use this interpreter to unwind the entire stack.

Parsing Common Information Entries

Let's start by writing a type we'll use to manage call frame information and store CIEs. We don't need to store all of a CIE's fields, because `version`, `CIE_id`, and `return_address_register` should always be the same. Also, we'll check whether linked FDEs contain augmentation information (meaning the CIE's augmentation string is non-empty) and, if so, record the encoding for FDE code pointers. Begin defining the type in *sdb/include/libsdb/dwarf.hpp*, above the definition of `sdb::dwarf`:

```
namespace sdb {
    class dwarf;
    class call_frame_information {
    public:
        struct common_information_entry {
            std::uint32_t length;
            std::uint64_t code_alignment_factor;
```

```
            std::int64_t data_alignment_factor;
            bool fde_has_augmentation;
            std::uint8_t fde_pointer_encoding;
            span<const std::byte> instructions;
        };

        call_frame_information() = delete;
        call_frame_information(const call_frame_information&) = delete;
        call_frame_information& operator=(const call_frame_information&) = delete;

        const dwarf& dwarf_info() const { return *dwarf_; }

    private:
        const dwarf* dwarf_;
    };
}
```

We delete the type's copy and default construction operations, but we'll add a constructor to this type soon. We store a pointer to the `sdb::dwarf` object that this call frame information belongs to, as we'll need to use it in a few places in the parser.

FDEs reference CIEs by their offset in the object file. As such, when parsing an FDE, we'll need to be able to quickly retrieve a CIE from this offset. To facilitate this retrieval, we'll parse CIEs when they are required and cache them in a map from offsets to `sdb::call_frame_information` objects. We'll parse FDEs when the stack unwinder requires them. Add this map to `sdb::call_frame_information`:

```
namespace sdb {
    class call_frame_information {
    public:
        --snip--
        const common_information_entry& get_cie(file_offset at) const;

    private:
        --snip--
        mutable std::unordered_map<std::uint32_t, common_information_entry> cie_map_;
    };
}
```

We add a member that stores the map and a function to retrieve a CIE from the offset at which it begins. We declare `cie_map_` as `mutable` because it's used as a cache.

Let's implement `get_cie` and assume that we've already written a `parse_cie` function that parses the CIE at the given offset. Implement `get_cie` in *sdb/src/dwarf.cpp*:

```
const sdb::call_frame_information::common_information_entry&
sdb::call_frame_information::get_cie(file_offset at) const {
    auto offset = at.off();
    if (cie_map_.count(offset)) {
        return cie_map_.at(offset);
    }

    auto section = at.elf_file()->get_section_contents(".eh_frame");
    cursor cur({ at.elf_file()->file_offset_as_data_pointer(at), section.end()});
    auto cie = parse_cie(cur);
    cie_map_.emplace(offset, cie);
    return cie_map_.at(offset);
}
```

We look up the given offset in the CIE cache. If we've already parsed this CIE, we return it. Otherwise, we construct a cursor from the specified offset into the ELF file up to the end of the `.eh_frame` section. We parse the CIE there, emplace it into the map, and return a reference to the stored CIE.

Now we can start writing the parser. Add a `parse_cie` function to *sdb/src/dwarf.cpp* that will parse a single CIE at a given cursor position. We'll write it piece by piece, starting by parsing all the components up to the augmentation string:

```
namespace {
    sdb::call_frame_information::common_information_entry
    parse_cie(cursor cur) {
        auto start = cur.position();
        auto length = cur.u32() + 4;
        auto id = cur.u32();
        auto version = cur.u8();

        if (!(version == 1 or version == 3 or version == 4)) {
            sdb::error::send("Invalid CIE version");
        }
```

We record the start of the CIE for later use. We then parse all the elements of the CIE one by one. We add 4 to the parsed length because the `length` field doesn't include its own size. If the CIE version is a value other than `1`, `3`, or `4`, we throw an error.

Next, we parse up to the augmentation data:

```
        --snip--
        auto augmentation = cur.string();

        if (!augmentation.empty() and augmentation[0] != 'z') {
            sdb::error::send("Invalid CIE augmentation");
        }
```

```
        if (version == 4) {
            auto address_size = cur.u8();
            auto segment_size = cur.u8();
            if (address_size != 8)
                sdb::error::send("Invalid address size");
            if (segment_size != 0)
                sdb::error::send("Invalid segment size");
        }

        auto code_alignment_factor = cur.uleb128();
        auto data_alignment_factor = cur.sleb128();
        auto return_address_register =
            version == 1 ? cur.u8() : cur.uleb128();
```

We first parse the augmentation string. If this string is not empty, it must start with `z`, the specifier for the augmentation data size, to enable consumers to easily skip over the augmentation data if they want to. We throw an error if this is not the case. If the CIE version is `4`, then there are fields for the address and segment sizes. We parse them and throw errors if they have unexpected values. We then parse the code and data alignment factors and the return address register, whose encoding depends on the CIE version.

Next, we'll handle the augmentation data. The data for the `z`, `R`, and `P` specifiers is of fixed length, so these specifiers are easy to parse. The `L` specifier is trickier, as the pointer's length depends on the specified encoding. Because we won't actually use this pointer, we don't care about the relative part of the encoding. We'll assume that a `parse_eh_frame_pointer_with_base` function parses a pointer at the given cursor position with a given encoding and a given base address. Its signature will look like this:

```
std::uint64_t parse_eh_frame_pointer_with_base(
     cursor& cur, std::uint8_t encoding, std::uint64_t base);
```

Whether the base address and return value are file offsets or file addresses depends on the context in which we parse the pointer, so we use bare integers here. We'll implement this function when we've finished writing `parse_cie`, which continues as follows:

```
        --snip--
        std::uint8_t fde_pointer_encoding = DW_EH_PE_udata8 | DW_EH_PE_absptr;
        for (auto c : augmentation) {
            switch (c) {
            case 'z': cur.uleb128(); break;
            case 'R': fde_pointer_encoding = cur.u8(); break;
            case 'L': cur.u8(); break;
            case 'P': {
                auto encoding = cur.u8();
                (void)parse_eh_frame_pointer_with_base(cur, encoding, 0);
                break;
```

```
            }
            default: sdb::error::send("Invalid CIE augmentation");
            }
        }
```

We define an `fde_pointer_encoding` type to hold the encoding. By default, an FDE code pointer's encoding uses an absolute 64-bit address. The `DW_EH_PE_*` macros produce understandable names for the bitmask values you saw in "Common Information Entries" on page 430. We then loop over all specifiers in the augmentation string.

The `z` specifier gives the length of the augmentation data as a `ULEB128`. We don't need this value, so we parse it and throw it away. The `R` specifier gives the pointer encoding for code pointers in linked FDEs as a byte. We do need this information for decoding FDE data, so we save it.

We don't need the `L` specifier, so we throw it away. The `P` specifier gives the pointer encoding for the personality routine as a byte, followed by a pointer encoded using that scheme. We don't need this pointer, so we first parse the encoding, then parse the encoded pointer and throw it away.

Finally, we can save a span representing the call frame information instructions and package the data in a `common_frame_information` object:

```
        --snip--
        sdb::span<const std::byte> instructions = {
            cur.position(), start + length
        };
        bool fde_has_augmentation = !augmentation.empty();
        return { length, code_alignment_factor,
                 data_alignment_factor, fde_has_augmentation,
                 fde_pointer_encoding, instructions };
    }
}
```

The instructions span from the end of the augmentation data to the end of the CIE. We could execute these instructions now and save the state of the interpreter to avoid recomputing the same data, but for simplicity, we'll just store the instruction bytes. Linked FDEs contain augmentation data so long as the CIE also contains augmentation data, so we compute whether there is data. We then return the `common_frame_information` object.

Now we can implement `parse_eh_frame_pointer_with_base`, the pointer parsing function. The least significant 4 bits of the encoding byte tell us how to interpret the pointer:

```
namespace {
    std::uint64_t parse_eh_frame_pointer_with_base(
        cursor& cur, std::uint8_t encoding, std::uint64_t base) {
        switch (encoding & 0x0f) {
        case DW_EH_PE_absptr: return base + cur.u64();
        case DW_EH_PE_uleb128: return base + cur.uleb128();
        case DW_EH_PE_udata2: return base + cur.u16();
```

```
            case DW_EH_PE_udata4: return base + cur.u32();
            case DW_EH_PE_udata8: return base + cur.u64();
            case DW_EH_PE_sleb128: return base + cur.sleb128();
            case DW_EH_PE_sdata2: return base + cur.s16();
            case DW_EH_PE_sdata4: return base + cur.s32();
            case DW_EH_PE_sdata8: return base + cur.s64();
            default: sdb::error::send("Unknown eh_frame pointer encoding");
            }
        }
    }
```

We retrieve the integer encoding specifier from the encoding byte, then compare the result with the binary encoding macros. Using the macro that matches the data, we parse the relevant integral type and add it to the base address. We handle unknown encodings and then return the computed pointer value.

We also need to deal with the most significant 4 bits of the encoding byte, which tell us what address the pointer is relative to. We'll define a `parse_eh_frame_pointer` function that will take the cursor, the encoding, and the four base addresses to use for the four relative encoding schemes we need to handle:

```
namespace {
    std::uint64_t parse_eh_frame_pointer(
          const sdb::elf& elf,
          cursor& cur, std::uint8_t encoding,
          std::uint64_t pc, std::uint64_t text_section_start,
          std::uint64_t data_section_start, std::uint64_t func_start) {
        std::uint64_t base = 0;
        switch (encoding & 0x70) {
        case DW_EH_PE_absptr: break;
        case DW_EH_PE_pcrel:
            base = pc; break;
        case DW_EH_PE_textrel:
            base = text_section_start; break;
        case DW_EH_PE_datarel:
            base = data_section_start; break;
        case DW_EH_PE_funcrel:
            base = func_start; break;
        default: sdb::error::send("Unknown eh_frame pointer encoding");
        }

        return parse_eh_frame_pointer_with_base(cur, encoding, base);
    }
}
```

Like in `parse_eh_frame_pointer_with_base`, whether the base addresses and return value are file offsets or file addresses is context-dependent, so we use

raw integers for them. We initialize `base` to `0`. We then mask out the least significant 4 bits and the most significant bit and determine which of the relevant `DW_EH_PE_*` macros the result corresponds to. We mask out the most significant bit (`0x80`) because it corresponds to the indirect encoding scheme, which we don't need to handle.

If the pointer is absolute, the base address is 0. Otherwise, the base address is one of the four values given as arguments. In each case, we throw an exception if no supplied base address exists for the requested encoding. We then call `parse_eh_frame_pointer_with_base` with the base address we calculated and return the result.

This function completes the CIE parser. We'll hook it into the rest of our code later, after we've implemented FDE parsing. Let's move on to that now.

Parsing Frame Description Entries

FDEs are simpler than CIEs but quite similar in structure, so you don't need to learn any new concepts to parse them. Let's add a nested type to `sdb::call_frame_information` to hold their data. We don't need to store the augmentation information, as the debugger won't use it. Define the type in *sdb/include/libsdb/dwarf.hpp*, below the definition of `common_information_entry`:

```
namespace sdb {
    class call_frame_information {
    public:
        --snip--
        struct frame_description_entry {
            std::uint32_t length;
            const common_information_entry* cie;
            file_addr initial_location;
            std::uint64_t address_range;
            span<const std::byte> instructions;
        };
        --snip--
    };
}
```

We'll implement the FDE parser in *sdb/src/dwarf.cpp*. It will take a reference to the `sdb::call_frame_information` object to which the FDE belongs and a cursor to the beginning of the FDE. As with the CIE parser, we'll implement this function piece by piece, starting with the code to handle the length:

```
namespace {
    sdb::call_frame_information::frame_description_entry
    parse_fde(const sdb::call_frame_information& cfi, cursor cur) {
        auto start = cur.position();
        auto length = cur.u32() + 4;
```

Like in the CIE parser, we save the start location and add 4 to the parsed length, because the `length` field doesn't include its own size.

Next, we parse the offset to the linked CIE, which the FDE expresses as a negative offset from the current parse position. To look up the parsed CIE in the CIE map, we need to convert it into a file offset. No function in `sdb::elf` can achieve this, so let's add a couple of simple helpers to *sdb/include/libsdb/elf.hpp*:

```
namespace sdb {
    class elf {
    public:
        --snip--
        file_offset data_pointer_as_file_offset(const std::byte* ptr) const {
            return file_offset(*this, ptr - data_);
        }
        const std::byte* file_offset_as_data_pointer(file_offset offset) const {
            return data_ + offset.off();
        }
        --snip--
    };
}
```

These functions convert between file offsets and data pointers. Now, back in the implementation of `parse_fde`, we can parse CIE offsets and look them up in the CIE map:

```
        --snip--
        auto elf = cfi.dwarf_info().elf_file();
        auto current_offset = elf->data_pointer_as_file_offset(cur.position());
        sdb::file_offset cie_offset { *elf, current_offset.off() - cur.s32() };
        auto& cie = cfi.get_cie(cie_offset);
```

We retrieve the `sdb::elf` object to which this call frame information belongs and convert the current cursor position into a file offset. We then parse the distance to the linked CIE and subtract it from the current cursor offset to get the offset of the linked CIE from the start of the object file. We use this offset to retrieve the CIE, potentially parsing it and caching it.

The next field to parse is `initial_location`. The FDE encodes this field according to the `R` augmentation in the linked CIE, so we'll make use of the `parse_eh_frame_pointer` function we wrote earlier to decode it:

```
        --snip--
        current_offset = elf->data_pointer_as_file_offset(cur.position());
        auto text_section_start = elf->get_section_start_address(".text")
            .value_or(sdb::file_addr{});
        auto initial_location_addr = parse_eh_frame_pointer(
            *elf, cur, cie.fde_pointer_encoding, current_offset.off(),
            text_section_start.addr(), 0, 0);
        sdb::file_addr initial_location{ *elf, initial_location_addr };
```

We recalculate the current parse offset and retrieve the start of the `.text` section from the ELF file. We then pass this information to `parse_eh_frame_pointer` to parse the encoded pointer. We're not in the `.eh_frame_hdr` section, so we pass `0` for the penultimate argument. No function is currently being executed, so we pass `0` for the final argument as well.

Next, we parse the `address_range` field. While also encoded according to the `R` augmentation in the linked CIE, only the low 4 bits are used for `address_range`, so we can call `parse_eh_frame_pointer_with_base` rather than `parse_eh_frame_pointer` to handle it:

```
        --snip--
        auto address_range = parse_eh_frame_pointer_with_base(
            cur, cie.fde_pointer_encoding, 0);
```

Next is the augmentation data. We've recorded whether this data should exist in the `fde_has_augmentation` section of the linked CIE. If the FDE has augmentation data, the first entry must be the size of the augmentation data. Because we don't need any of this information, we'll just read the length of the data and skip past it:

```
        --snip--
        if (cie.fde_has_augmentation) {
            auto augmentation_length = cur.uleb128();
            cur += augmentation_length;
        }
```

Finally, we create a span that captures the position of the CFI instructions and return all of the data we parsed:

```
        --snip--
        sdb::span<const std::byte> instructions = { cur.position(), start + length };
        return { length, &cie, initial_location, address_range, instructions };
    }
}
```

Before we hook these parsers into our codebase and move on to interpreting the call frame information instructions, we must add one more substantial piece of support: a fast way to look up the FDE that corresponds to a given instruction.

Looking Up Frame Description Entries

If we want to unwind the stack from a given instruction, we need the ability to take the address of that instruction and find the FDE that holds the unwind information for it. However, there may be thousands, tens of thousands, or even hundreds of thousands of FDEs, so performing a linear search is impractical.

Fortunately, a section in the ELF file called `.eh_frame_hdr` contains a fast lookup table for FDEs. You can find a description of this EH frame header

section in section 10.6.2 of the generic part of the LSB core specification (*https://refspecs.linuxbase.org/LSB_5.0.0/LSB-Core-generic/LSB-Core-generic.html*).

By parsing this section, we can perform a binary search on the `initial_location` value to locate the FDE that corresponds to a given instruction. Let's implement such a parser.

Parsing the EH Frame Header

The `.eh_frame_hdr` section contains a binary search table that maps addresses to offsets of the FDEs that store the corresponding unwind information. The section consists of seven fields:

`version` (`std::uint8_t`) The version of the `.eh_frame_hdr` format (must be `1`).

`eh_frame_ptr_enc` (`std::uint8_t`) The encoding format of the `eh_frame_ptr` field.

`fde_count_enc` (`std::uint8_t`) The encoding format of the `fde_count` field.

`table_enc` (`std::uint_8_t`) The encoding format of the entries in the binary search table. This will always be a fixed-size encoding (that is, not `ULEB128`) because otherwise jumping to arbitrary entries would not be possible.

`eh_frame_ptr` (encoded pointer) A pointer to the start of the `.eh_frame` section.

`fde_count` (encoded integer) The number of entries in the binary search table. This will always be an absolute encoding.

`binary_search_table` (array of integers encoded according to the `table_enc` field) A table containing the number of entries given by the `fde_count` field. Each entry of the table corresponds to a single FDE and consists of two encoded integers: the value of the `initial_location` field for the FDE and the offset of the FDE from the start of the object file. The entries are sorted in an ascending order by the `initial_location` value.

Fields that specify encoding formats follow the scheme used in the CIE and FDE augmentations, described in "Common Information Entries" on page 430.

To hold the header information, we'll add a nested `eh_hdr` type to `sdb::call_frame_information` in *sdb/src/dwarf.hpp*:

```
namespace sdb {
    class call_frame_information {
    public:
        --snip--
        struct eh_hdr {
            const std::byte* start;
            const std::byte* search_table;
```

```
            std::size_t count;
            std::uint8_t encoding;
            call_frame_information* parent;
            const std::byte* operator[](file_addr address) const;
        };
        --snip--
    };
}
```

We store a pointer to the start of the `.eh_frame_hdr` section, as we'll need this value to decode certain pointers. We also need a pointer to the start of the search table, the number of entries in the table, the entries' encoding, and a pointer to the parent call frame information. We then declare a function that takes an instruction's object file offset and returns a pointer to the start of the FDE for that instruction.

To perform the parsing, we'll write a `parse_eh_hdr` function in *sdb/src/dwarf.cpp*. We begin by gathering information from the ELF file about the start of various sections, as we'll need it to initialize our cursor and parse encoded pointers:

```
namespace {
    sdb::call_frame_information::eh_hdr
    parse_eh_hdr(sdb::dwarf& dwarf) {
        auto elf = dwarf.elf_file();
        auto eh_hdr_start = *elf->get_section_start_address(".eh_frame_hdr");
        auto text_section_start = *elf->get_section_start_address(".text");
```

We retrieve the `sdb::elf` for the given `sdb::dwarf` and then retrieve start addresses for the `.eh_frame_hdr` and `.text` sections, which we'll need to decode pointers. Next, we initialize the cursor with the data from the `.eh_frame_hdr` section and parse the first four fields:

```
        --snip--
        auto eh_hdr_data = elf->get_section_contents(".eh_frame_hdr");
        cursor cur(eh_hdr_data);

        auto start = cur.position();
        auto version = cur.u8();
        auto eh_frame_ptr_enc = cur.u8();
        auto fde_count_enc = cur.u8();
        auto table_enc = cur.u8();
```

Now we parse the encoded pointer to the `.eh_frame` section. We don't really need this value, as we can easily retrieve the pointer from `sdb::elf`, so we throw away the encoded pointer:

```
        --snip--
        (void)parse_eh_frame_pointer_with_base(cur, eh_frame_ptr_enc, 0);
```

Next, we parse the FDE count. This value should always be absolute, so we use `parse_eh_frame_pointer_with_base`:

```
        --snip--
        auto fde_count = parse_eh_frame_pointer_with_base(
            cur, fde_count_enc, 0);
```

Finally, we save the current cursor position, which points to the start of the search table, and return the parsed `eh_frame_hdr` object:

```
        --snip--
        auto search_table = cur.position();
        return { start, search_table, fde_count, table_enc, nullptr };
    }
}
```

We don't yet have an `sdb::call_frame_information` object to store as the parent for this object, so we pass `nullptr` for now; we'll fill in the parent later. With the parser complete, we can implement the `[]` operator for looking up entries in the table.

Searching the EH Frame Table

It's time to write a binary search algorithm. Unfortunately, we can't use the existing `std::binary_search` function in the C++ standard library, because we're operating on raw bytes that we must interpret as we go. Also, while we'll perform a more or less standard binary search, we don't require an exact match. In other words, rather than retrieving the FDE's initial address, we might look for an address that lies in that FDE's range of addresses. The algorithm we'll implement looks like this:

1. Start with `low = 0` and `high = fde_count - 1`.
2. Find the midpoint (`mid`) between `low` and `high`.
3. Decode the initial address of the table entry whose index is `mid`.
4. If that address is less than the address we're looking for, set `low` to `mid + 1`. If, instead, that address is greater than the address we're looking for, set `high` to `mid - 1`. Otherwise, set `high` to `mid` and stop looking.
5. Go back to step 2 so long as `low` doesn't exceed `high`.

At the end of this process, `high` will contain the index of the entry whose initial address is closest to, but doesn't exceed, the address we're looking for.

To jump to an arbitrary table entry, we must know each entry's byte size, held in the `table_enc` field of the `.eh_frame_hdr` section. Let's write a function in *sdb/src/dwarf.cpp* to get the byte size for a given encoding scheme:

```
namespace {
    std::size_t eh_frame_pointer_encoding_size(std::uint8_t encoding) {
        switch (encoding & 0x7) {
```

```
            case DW_EH_PE_absptr: return 8;
            case DW_EH_PE_udata2: return 2;
            case DW_EH_PE_udata4: return 4;
            case DW_EH_PE_udata8: return 8;
            default: sdb::error::send("Invalid pointer encoding");
            }
        }
    }
```

Recall from the description in "Common Information Entries" on page 430 that the least significant 3 bits of the encoding byte determine whether the value is encoded as an `LEB128` or as a 2-byte, 4-byte, or 8-byte integer. As such, we mask out the other bits and compare the result to the relevant `DW_EH_PE_*` macro. Note that we don't need to check the signed versions of those macros, because we mask out the sign bit.

With this helper function written, we can implement the binary search function. Let's begin by retrieving the start of the `.text` section, so we can use it for decoding pointers, and then calculating the size of each table row:

```
const std::byte*
sdb::call_frame_information::eh_hdr::operator[](file_addr address) const {
    auto elf = address.elf_file();
    auto text_section_start = *elf->get_section_start_address(".text");
    auto encoding_size = eh_frame_pointer_encoding_size(encoding);
    auto row_size = encoding_size * 2;
```

Each row of the table stores two values, so the row size is twice the size of the pointer encoding. Next is the main loop, which translates the algorithm I described in this section into code:

```
--snip--
std::size_t low = 0;
std::size_t high = count - 1;
while (low <= high) {
    std::size_t mid = (low + high) / 2;

    cursor cur({ search_table + mid * row_size, ❶
                 search_table + count * row_size });
    auto current_offset = elf->data_pointer_as_file_offset(cur.position());
    auto eh_hdr_offset = elf->data_pointer_as_file_offset(start);
    auto entry_address = parse_eh_frame_pointer(*elf, cur, encoding, current_offset.off(),
        text_section_start.addr(), eh_hdr_offset.off(), 0);

    if (entry_address < address.addr()) {
        low = mid + 1;
    }
    else if (entry_address > address.addr()) {
        if (mid == 0)
            sdb::error::send("Address not found in eh_hdr");
```

```
            high = mid - 1;
        }
        else {
            high = mid;
            break;
        }
    }
```

The trickiest bit of this algorithm is calculating the cursor position and then parsing the pointer value found there ❶. The cursor's start location is `search_table + mid * row_size` because `mid * row_size` translates the midpoint table index into a byte offset from the start of the search table. Adding this offset to the pointer to the start of the search table gives us a pointer to the byte corresponding to the table entry at index `mid`. Similarly, `count * row_size` gives us the byte offset of the end of the search table.

When we call `parse_eh_frame_pointer`, we pass the current parse offset as the program counter value and the offset of the start of the `.eh_frame_hdr` section as the data section base address. The current function being executed is irrelevant, so we pass `0` as the last argument.

Another tricky bit of this algorithm involves the `if...else` blocks toward the end. If we find an entry that is lower than the desired address, we set `low` to `mid + 1`. If the desired address is between the entries at `mid` and `mid + 1` at that point of the algorithm, however, we'll end up looking at the element one past the one we actually want. This is fine, because in the condition where we find an entry greater than the desired address, we set `high` to `mid - 1`, which gets us to the correct entry in the end.

After this code runs, the table index for the desired entry will be stored in `high`. All that's left is to parse the second pointer in that entry to retrieve the object file offset of the FDE we're looking for:

```
    --snip--
    cursor cur({ search_table + high * row_size + encoding_size,
                 search_table + count * row_size });
    auto current_offset = elf->data_pointer_as_file_offset(cur.position());
    auto eh_hdr_offset = elf->data_pointer_as_file_offset(start);
    auto fde_offset_int = parse_eh_frame_pointer(
        *elf, cur, encoding, current_offset.off(),
        text_section_start.addr(), eh_hdr_offset.off(), 0);
    sdb::file_offset fde_offset{ *elf, fde_offset_int };
    return elf->file_offset_as_data_pointer(fde_offset);
}
```

The cursor's start position is `search_table + high * row_size + encoding_size`, because `search_table + high * row_size` gives us a pointer to the start of the desired entry and adding `encoding_size` skips over the `initial_address` value, resulting in a pointer to the encoded FDE pointer. We parse this pointer, convert it to a data pointer from which we can read, and return it.

Adding the Parsers to the Debugger

At this point, we've created a functional lookup table for FDEs. Now we can hook our various parsers into the `sdb::dwarf` and `sdb::call_frame_information` types. Add an `eh_hdr` member to `sdb::call_frame_information` as well as a constructor that fills in all of the members in *sdb/include/libsdb/dwarf.hpp*:

```
namespace sdb {
    class call_frame_information {
    public:
        --snip--
        call_frame_information(const dwarf* dwarf, eh_hdr hdr)
            : dwarf_(dwarf), eh_hdr_(hdr) {
            eh_hdr_.parent = this;
        }

    private:
        --snip--
        eh_hdr eh_hdr_;
    };
}
```

The constructor fills in the members of `sdb::call_frame_information` and then sets the parent of `eh_hdr_` to itself. Add members to `sdb::dwarf` to store and expose the call frame information:

```
namespace sdb {
    class dwarf {
    public:
        --snip--
        const call_frame_information& cfi() const { return *cfi_; }

    private:
        --snip--
        std::unique_ptr<call_frame_information> cfi_;
    };
}
```

Recall that `sdb::call_frame_information` objects can't be copied or default-constructed, so we store a `std::unique_ptr` instead of storing the `call_frame_information` object directly.

In *sdb/src/dwarf.cpp*, we implement a `parse_call_frame_information` type that parses the `.eh_frame_hdr` section:

```
namespace {
    std::unique_ptr<sdb::call_frame_information>
    parse_call_frame_information(sdb::dwarf& dwarf) {
        auto eh_hdr = parse_eh_hdr(dwarf);
```

```
        return std::make_unique<sdb::call_frame_information>(
            &dwarf, eh_hdr);
    }
}
```

We parse the `.eh_frame_hdr` section and then wrap the data in an `sdb::call_frame_information` object.

Call the `parse_call_frame_information` function in the constructor for `sdb::dwarf` to initialize the CIE map:

```
sdb::dwarf::dwarf(const sdb::elf& parent) : elf_(&parent) {
    compile_units_ = parse_compile_units(*this, parent);
    cfi_ = parse_call_frame_information(*this);
}
```

All the parsing and data plumbing of the call frame information is now complete.

Summary

In this chapter, you learned how DWARF call frame information encodes the information that a debugger needs to unwind the program stack. You implemented parsers for CIEs and FDEs and hooked them in to `libsdb`.

In the next chapter, you'll implement a stack unwinder that uses the parsers you just implemented.

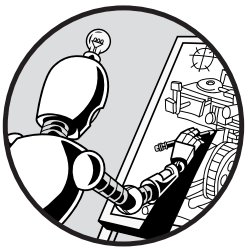
Check Your Knowledge

1. What are the two kinds of entry in DWARF call frame information?
2. Why are there two kinds of entry rather than just one?
3. What is the canonical frame address (CFA)?
4. What DWARF register number is used for the return address on x64?
5. What is augmentation data?
6. In which documents is the format of call frame information on Linux systems defined?

16

STACK UNWINDING

Our fingers brush the edges
of spirals of light
winding down
through the cracked tiles.



In the previous chapter, you learned about DWARF call frame information and implemented parsers for CIEs and FDEs. In this chapter, you'll use these parsers to write a stack unwinder. First, you'll write an unwinder for a single stack frame. Based on this unwinder, you'll write a full stack unwinder that can handle an arbitrary number of frames, including inlined frames. Finally, you'll expose the new functionality to command line users.

Executing Call Frame Information

Recall that DWARF call frame information essentially specifies a huge table. Each row of the table lists a program counter value. The rules in that row tell you how to unwind the current stack frame when the program counter has a value in the noninclusive range between that address and the program

counter value in the next table row (or the end of the range represented by the matching FDE, in the case of the last row in the table). Table rows encode two things: the CFA for the start of the current stack frame, and rules for restoring registers to the values that they would have if the current function immediately returned to its caller. For example, consider the following four instructions that form a function prologue:

```
1149:       endbr64
114d:       push   %rbp
114e:       mov    %rsp,%rbp
1151:       lea    0xeac(%rip),%rax
```

The important changes here are that the instruction at `0x114d` pushes `rbp` to the stack and the instruction at `0x114e` sets the base pointer for the current function to the current value of the stack pointer. When the program counter is at `0x114e`, the call frame information should indicate how to restore the value of `rbp` from the stack.

Here are the relevant CFI instructions, as printed by `dwarfdump` with the `--print-eh-frame` flag:

```
0x00001149: <off cfa=08(r7) > <off r16=-8(cfa) >
0x0000114e: <off cfa=16(r7) > <off r6=-16(cfa) > <off r16=-8(cfa) >
0x00001151: <off cfa=16(r6) > <off r6=-16(cfa) > <off r16=-8(cfa) >
```

The entry with the program counter value `0x1149` gives unwind information for when the program counter is between `0x1149` and `0x114e`, noninclusive. It states that the CFA is at an offset of 8 bytes from the register with DWARF ID 7 (`rsp`) and that the old value for the register with DWARF ID 16 (the return address, so `rip`) is stored at an offset of −8 bytes from the CFA. When the program counter reaches `0x114e`, there is a new set of unwind information. Since an 8-byte value was pushed to the stack, the CFA is now 16 bytes away from `rsp`, and there's a new rule giving the CFA offset for `rbp` (DWARF ID 6). When the program counter reaches `0x1151`, the CFA rule is changed to use `rbp` for CFA calculations rather than `rsp`, so that the function can happily push to and pop from the stack without requiring further changes to the CFA rule.

Storing the whole table for an entire program is infeasible, so it's instead encoded as a stream of binary instructions that build it up piece by piece, similarly to the line table program. We can find the series of instructions to rebuild the table entry for a given program counter value in the matching FDE and its linked CIE.

In this section, we'll write an interpreter for DWARF call frame information that can compute the unwind rules for a single stack frame.

Representing Rules

The rule for restoring a register can be one of the following:

`undefined` It's not possible to restore the register value.

`register(R)` The previous value of the register is stored in another register, with the DWARF register number `R`.

`same_value` The register hasn't been modified from its previous value. (This is a special case of `register(R)` where `R` is the same as the register for which the rule is defined.)

`offset(N)` The previous value of the register is saved at an offset of `N` from the current CFA.

`val_offset(N)` The previous value of the register is the current CFA plus `N`.

`expression(E)` The previous value of the register is located at the address produced by executing the DWARF expression `E`.

`val_expression(E)` The previous value of the register is the value produced by executing the DWARF expression `E`.

We'll handle all but the last two rules in this chapter. DWARF expressions are considerably more complex, so we'll wait until Chapter 19 to tackle them. In addition to register rules, we must handle the following rules for computing the CFA:

`register_and_offset(R,N)` The CFA is calculated by taking the address stored in the register with the DWARF register number `R` and adding the offset `N` to it.

`expression(E)` The CFA is calculated by executing the DWARF expression `E`.

Again, we'll ignore the DWARF expression option for now. In *sdb/src/dwarf.cpp*, let's create types to represent these rules:

```
namespace {
    struct undefined_rule {};
    struct same_rule {};
    struct offset_rule {
        std::int64_t offset;
    };
    struct val_offset_rule {
        std::int64_t offset;
    };
    struct register_rule {
        std::uint32_t reg;
    };
```

```
    struct cfa_register_rule {
        std::uint32_t reg;
        std::int64_t offset;
    };
}
```

Now we can think about the call frame information instructions we'll need to interpret to build the relevant row of the unwind table for a given machine instruction. DWARF 4 has 25 instructions that we need to handle. These instructions operate on an abstract machine similar to the one for the line table program and will do things like set the current register rule for a specific register or set the address of the instruction to which the current table row applies.

Unlike for the line table program, the DWARF standard doesn't lay out an explicit set of registers for the call frame information abstract machine, but you can piece one together by reading through the list of instructions. Fortunately, I've done that already, so you don't have to. We'll use the following set of fields for the abstract machine:

`location` The file address of the instruction to which a line of the table applies.

`cfa_rule` The rule for computing the CFA.

`register_rules` The set of rules for restoring registers.

`cie_register_rule` The set of rules for restoring registers that results from running only the instructions in the `initial_instructions` field of the linked CIE for this FDE. We need this to support an instruction that resets the `register_rules` field back to this state.

`register_rule_stack` A stack of register and CFA rules that can be pushed to and popped from. We need this because certain instructions can save and restore the `register_rules` field using stack operations.

Based on this information, we can define a type to represent our abstract machine, which we'll call `unwind_context`. Alongside fields for these items, we'll add a field to hold a parsing cursor. Add the type to *sdb/src/dwarf.cpp*:

```
#include <variant>

namespace {
    struct unwind_context {
        cursor cur{ {nullptr, nullptr} };
        sdb::file_addr location;
        cfa_register_rule cfa_rule;

        using rule = std::variant<
            undefined_rule, same_rule, offset_rule,
            val_offset_rule, register_rule>;
        using ruleset = std::unordered_map<std::uint32_t, rule>;
        ruleset cie_register_rules;
```

```
        ruleset register_rules;
        std::vector<std::pair<ruleset, cfa_register_rule>> rule_stack;
    };
}
```

We initialize the cursor with a null span; we'll fill it in manually later. We use a `std::variant` to store the potential register restore rules, and we define `ruleset` as a mapping from DWARF register numbers to register restore rules.

Returning Registers

With `unwind_context` defined, we can start to sketch out the main entry point to all the work we've done this chapter: the `sdb::call_frame_information::unwind` function. This function should take the `sdb::process` being debugged, the program counter value to unwind (as a file address), and the current set of registers. It should return a new set of registers that represents the machine state in the caller.

To support this behavior, we'll need to make some changes to our existing `sdb::registers` type. Currently, this type isn't default-constructible or copyable, which makes it quite unwieldy for our current use cases. It can't express undefined registers and doesn't support modifying register values without also writing those values into the inferior. It should also track the CFA for the stack frame and have a function for writing all register updates to the inferior to accommodate situations in which we assign some register set to another set and want to reflect those registers in the running process. Let's modify the type in *sdb/include/libsdb/registers.hpp*:

```
namespace sdb {
    class process;
    class registers {
    public:
        registers() = default;
        registers(const registers&) = default;
        registers& operator=(const registers&) = default;
        --snip--
        void write(const register_info& info, value val, bool commit=true);
        --snip--
        void write_by_id(register_id id, value val, bool commit=true) {
            write(register_info_by_id(id), val, commit);
        }
        bool is_undefined(register_id id) const;
        void undefine(register_id id);

        virt_addr cfa() const { return cfa_; }
        void set_cfa(virt_addr addr) { cfa_ = addr; }
        void flush();
```

```
    private:
        --snip--
        std::vector<std::size_t> undefined_;
        virt_addr cfa_;
    };
}
```

We mark the default constructor and copy operations as defaulted rather than deleted. We also add `commit` parameters to the two register-writing functions and default them to `true`. These functions control whether to also write to the registers in the inferior.

To avoid large refactors in the way we read and write registers, we implement undefined registers by keeping a list of offsets into the user area we should consider undefined. We track offsets rather than register IDs so that undefining a subregister also marks its superregister as undefined, and vice versa. Add a member to track this information and declare two functions, `is_undefined` and `undefined`, to push values to this list and query it. Also add a member that stores the CFA and members to set and get it. Finally, add a `flush` function that will write all of the current registers back to the process.

Implement `is_undefined` and `undefined` in *sdb/src/registers.cpp*:

```
void sdb::registers::undefine(register_id id) {
    std::size_t canonical_offset = register_info_by_id(id).offset >> 1;
    undefined_.push_back(canonical_offset);
}

bool sdb::registers::is_undefined(register_id id) const {
    std::size_t canonical_offset = register_info_by_id(id).offset >> 1;
    return std::find(begin(undefined_), end(undefined_), canonical_offset)
        != end(undefined_);
}
```

In both of these functions, we shift the register offset 1 bit to the right so that registers at a byte offset like `hl` are considered the same as their containing register. We then add this canonicalized offset to the undefined set, or look it up.

In *sdb/src/registers.cpp*, modify the `read` function to throw an exception if the requested register is undefined:

```
sdb::registers::value sdb::registers::read(const register_info& info) const {
    if (is_undefined(info.id))
        sdb::error::send("Register is undefined");
    --snip--
}
```

Also implement `flush`, which should write all GPRs, FPRs, and debug registers:

```
void sdb::registers::flush() {
    proc_->write_fprs(data_.i387);
```

```
        proc_->write_gprs(data_.regs);
        auto info = register_info_by_id(register_id::dr0);
        for (auto i = 0; i < 8; ++i) {
            if (i == 4 or i == 5) continue;
            auto reg_offset = info.offset + sizeof(std::uint64_t) * i;
            auto ptr = reinterpret_cast<std::byte*>(data_.u_debugreg + i);
            auto bytes = from_bytes<std::uint64_t>(ptr);
            proc_->write_user_area(reg_offset, bytes);
        }
    }
```

We call `write_gprs` and `write_fprs` to write the GPRs and FPRs. For the debug registers, we loop eight times (one for each debug register), writing the data of each debug register into the user area. Recall that `DR4` and `DR5` are obsolete aliases for `DR6` and `DR7`, so we don't write those two.

The last required modification is to implement conditional writing to the process registers. In the `write` function, guard the last `if...else` block with a check to the `commit` parameter:

```
void sdb::registers::write(const register_info& info, value val, bool commit) {
    --snip--

    if (commit) {
        if (info.type == register_type::fpr) {
            proc_->write_fprs(data_.i387);
        }
        else {
            auto aligned_offset = info.offset & ~0b111;
            proc_->write_user_area(aligned_offset,
                from_bytes<std::uint64_t>(bytes + aligned_offset));
        }
    }
}
```

In this way, we'll write the new register value to the process only if `commit` is `true`. Now we can start writing the stack unwinder.

Writing a Stack Unwinder

The stack unwinding implementation will find the FDE for the given program counter value, parse it, and execute its call frame information instructions until it produces the table row corresponding to the program counter. Then, it will execute the unwind rules for that row. Declare this `unwind` function in *sdb/include/libsdb/dwarf.hpp*:

```
#include <libsdb/registers.hpp>

namespace sdb {
    class process;
```

```
    class call_frame_information {
    public:
        --snip--
        registers unwind(
            const process& proc,
            file_addr pc,
            registers& regs) const;
        --snip--
    };
}
```

The function takes the `sdb::process` being debugged, the program counter value to unwind, and the current set of registers. We take the program counter value as an `sdb::file_addr` rather than just retrieving it from the register set so that we know which ELF file is associated with that program counter value. (Knowing this will be important when we implement shared library support in the next chapter.) We can't mark the register set as `const` because `unwind` will fill in the CFA information for the current register set. The function returns a new set of registers that represents the machine state for the caller.

We'll assume the existence of two functions. The first one, `execute_cfi_instruction`, takes the ELF file and the FDE, executes a single instruction at the given cursor, and updates the unwind context appropriately. The second function, `execute_unwind_rules`, executes the unwind rules computed by the instructions and returns the set of registers for the caller's stack frame. Implement `unwind` in *sdb/src/dwarf.cpp*:

```
#include <libsdb/process.hpp>

sdb::registers sdb::call_frame_information::unwind(
      const sdb::process& proc, file_addr pc, registers& regs) const {
    auto fde_start = eh_hdr_[pc];
    auto eh_frame_end = dwarf_->elf_file()->get_section_contents(".eh_frame").end();

    cursor cur({ fde_start, eh_frame_end }); ❶
    auto fde = parse_fde(*this, cur);
    if (pc < fde.initial_location
        or pc >= fde.initial_location + fde.address_range) {
        sdb::error::send("No unwind information at PC");
    }

    unwind_context ctx{}; ❷
    ctx.cur = cursor(fde.cie->instructions);

    while (!ctx.cur.finished()) {
        execute_cfi_instruction(*dwarf_->elf_file(), fde, ctx, pc);
    }
```

```
    ctx.cie_register_rules = ctx.register_rules; ❸
    ctx.cur = cursor(fde.instructions);
    ctx.location = fde.initial_location;

    while (!ctx.cur.finished() and ctx.location <= pc) {
        execute_cfi_instruction(*dwarf_->elf_file(), fde, ctx, pc);
    }

    return execute_unwind_rules(ctx, regs, proc); ❹
}
```

We begin by looking up the program counter value in the `.eh_frame_hdr` search table, which gives us a pointer to the start of the relevant FDE. We need an end location for the cursor we'll use to parse the FDE, so we retrieve a pointer to the end of the `.eh_frame` section. We then create the cursor, parse the FDE, and make sure this FDE contains the given program counter value ❶.

With the FDE located and parsed, we create an unwind context and initialize the cursor with the instructions stored in the linked CIE ❷. We then loop, continually executing the instructions until the cursor reaches the end of the span.

After the loop terminates, `ctx.register_rules` will store the current set of register rules, so we copy it into the `ctx.cie_register_rules` field to save the register rules for later ❸. We reset the cursor to point to the instructions for the FDE and set the current location to the value in the FDE's `initial_location` field.

We now execute the instructions stored in the FDE until either the cursor reaches the end of the span or the location of the current entry in the table exceeds the program counter value we're unwinding, indicating we've computed the correct row of the table. Recall that a table row applies to program counter values ranging from the one listed in the row up to, but not including, the one in the subsequent table row. After this loop, `ctx.register_rules` and `ctx.cfa_rule` will contain the correct set of rules for unwinding one stack frame from this program counter. We execute the unwind rules we just computed and return the result ❹.

The last two big pieces of work we need to do are to implement the `execute_cfi_instruction` and `execute_unwind_rules` functions. We'll start with `execute_cfi_instruction`.

Executing Instructions

The interpreter for call frame information instructions is essentially a big `switch` statement that updates the current unwind context based on the opcode and operands of each instruction.

We must implement 25 instructions. Rather than me explaining all of them at once, we'll tackle them in batches. Begin by writing the function signature and assigning some convenience variables:

```
namespace {
    void execute_cfi_instruction(
          const sdb::elf& elf,
          const sdb::call_frame_information::frame_description_entry& fde,
          unwind_context& ctx, sdb::file_addr pc) {
        auto& cie = *fde.cie;
        auto& cur = ctx.cur;

        auto text_section_start = *elf.get_section_start_address(".text");
        auto plt_start = elf.get_section_start_address(".got.plt")
            .value_or(sdb::file_addr{});
```

We grab references to the CIE and current cursor; we'll use these variables often, and the shorter names will clean up our code. We also grab the file addresses of the start of the `.text` and `.got.plt` sections, as we'll need these to parse encoded pointers.

Call frame information instructions require one or more bytes to encode. The most significant 2 bits of the first byte encode the primary opcode. The other 6 bits can store either an extended opcode or operands for the instruction.

We'll begin by implementing the three instructions that the primary opcode encodes, namely `DW_CFA_advance_loc`, `DW_CFA_offset`, and `DW_CFA_restore`:

```
--snip--
auto opcode = cur.u8();
auto primary_opcode = opcode & 0xc0;
auto extended_opcode = opcode & 0x3f;
if (primary_opcode) {
    switch (primary_opcode) {
    case DW_CFA_advance_loc:
        ctx.location += extended_opcode * cie.code_alignment_factor;
        break;
    case DW_CFA_offset: {
        auto offset = ❶
            static_cast<std::int64_t>(cur.uleb128()) * cie.data_alignment_factor;
        ctx.register_rules.emplace(extended_opcode, offset_rule{ offset });
        break;
    }
    case DW_CFA_restore:
        ctx.register_rules.emplace(
            extended_opcode, ctx.cie_register_rules.at(extended_opcode));
        break;
    }
}
```

We read the opcode byte from the cursor and retrieve the potential primary and extended opcodes by masking the appropriate values. If the primary opcode isn't 0, we need to execute one of the three instructions just mentioned.

The `DW_CFA_advance_loc` instruction advances the current location by the value encoded in the least significant 6 bits of the opcode, multiplied by the `code_alignment_factor`. The `DW_CFA_offset` instruction takes two operands: a DWARF register number encoded in the least significant 6 bits of the opcode and an additional `ULEB128` offset. Then, it sets the rule for the given register to `offset(N)`, where `N` is the given offset multiplied by `data_alignment_factor`. The `DW_CFA_restore` instruction takes one operand, a DWARF register number encoded in the least significant 6 bits of the opcode, and then resets the rule for the given register to its original value in the CIE's register rules. The CIE stores the values of `code_alignment_factor` and `data_alignment_factor`. Note that GCC's unwinder treats `DW_CFA_restore` in the same way as `DW_CFA_same_value`, but we're going to follow the spec and the behavior of `libunwind`.

Note that we use the `extended_opcode` variable to refer to operands encoded in the least significant 6 bits of the opcode. Pay particular attention to the cast of the parsed `ULEB128` ❶; we must do this because C++ integer promotion rules would otherwise cast the signed `data_alignment_factor` variable to an unsigned integer, potentially discarding the sign and making the offset positive when it shouldn't be.

The remaining instructions all use the extended opcode. The first batch we'll focus on includes the set of instructions that change the current code location:

```
--snip--
else if (extended_opcode) {
    switch (extended_opcode) {
    case DW_CFA_set_loc: {
        auto current_offset = elf.data_pointer_as_file_offset(cur.position());
        auto loc = parse_eh_frame_pointer(
            elf, cur, cie.fde_pointer_encoding,
            current_offset.off(), text_section_start.addr(),
            plt_start.addr(), fde.initial_location.addr());
        ctx.location = sdb::file_addr{ elf, loc };
        break;
    }
    case DW_CFA_advance_loc1:
        ctx.location += cur.u8() * cie.code_alignment_factor;
        break;
    case DW_CFA_advance_loc2:
        ctx.location += cur.u16() * cie.code_alignment_factor;
        break;
    case DW_CFA_advance_loc4:
        ctx.location += cur.u32() * cie.code_alignment_factor;
        break;
```

If the extended opcode isn't 0, we switch based on its value. The instruction `DW_CFA_set_loc` takes an encoded pointer and sets the current location to that pointer value. This will always increase the value of the location. Next, the `DW_CFA_advance_loc{1,2,4}` instruction takes a fixed-width integer of the specified byte size and advances the current location by this amount.

In the case of `DW_CFA_set_loc`, we use `parse_eh_frame_pointer` to parse the encoded pointer. As usual, we pass the current parser offset as well as the start of the `.text` and `.got.plt` sections, as the first two base addresses. Now, however, we have a function base address as well: the `initial_location` field of the FDE. The fixed-size advances are comparatively straightforward; we just parse the relevant integral type and advance the location.

Next are the instructions for defining the CFA rule:

```
--snip--
case DW_CFA_def_cfa:
    ctx.cfa_rule.reg = cur.uleb128();
    ctx.cfa_rule.offset = cur.uleb128();
    break;
case DW_CFA_def_cfa_sf:
    ctx.cfa_rule.reg = cur.uleb128();
    ctx.cfa_rule.offset = cur.sleb128() * cie.data_alignment_factor;
    break;
case DW_CFA_def_cfa_register:
    ctx.cfa_rule.reg = cur.uleb128();
    break;
case DW_CFA_def_cfa_offset:
    ctx.cfa_rule.offset = cur.uleb128();
    break;
case DW_CFA_def_cfa_offset_sf:
    ctx.cfa_rule.offset = cur.sleb128() * cie.data_alignment_factor;
    break;
case DW_CFA_def_cfa_expression:
    sdb::error::send("DWARF expressions not yet implemented");
```

We implement six instructions:

- The `DW_CFA_def_cfa` instruction takes a `ULEB128` representing a DWARF register number `R` and a `ULEB128` representing an offset `N`, and then defines the CFA rule as `register_and_offset(R,N)`.
- The `DW_CFA_def_cfa_sf` instruction takes a `ULEB128` representing a DWARF register number `R` and an `SLEB128` representing an offset `N`, then defines the CFA rule as `register_and_offset(R,N*data_alignment_factor)`.
- The `DW_CFA_def_cfa_register` instruction takes a `ULEB128` representing a DWARF register number and replaces the current register number of the CFA rule with the given one, keeping the old offset.

- The `DW_CFA_def_cfa_offset` instruction takes a `ULEB128` representing an offset and replaces the current offset of the CFA rule with the given one, keeping the old register.
- The `DW_CFA_def_cfa_offset` instruction takes an `SLEB128` representing an offset and replaces the current offset of the CFA rule with the given one multiplied by `data_alignment_factor`, keeping the old register.
- The `DW_CFA_def_cfa_expression` instruction takes a `DW_FORM_exprloc` operand that encodes a DWARF expression `E` and then sets the CFA rule to `expression(E)`.

We won't support DWARF expressions until Chapter 19, so we throw an exception if we have to execute one. Note that, unlike for the `DW_CFA_offset` instruction, we don't have to cast the offsets we parse here before multiplying them by the data alignment factor because they're already signed.

Now we implement the register rules. Once again, we'll watch out for those unsigned offsets we need to multiply with the signed data-alignment factor:

```
--snip--
case DW_CFA_undefined:
    ctx.register_rules.emplace(cur.uleb128(), undefined_rule{});
    break;
case DW_CFA_same_value:
    ctx.register_rules.emplace(cur.uleb128(), same_rule{});
    break;
case DW_CFA_offset_extended: {
    auto reg = cur.uleb128();
    auto offset = static_cast<std::int64_t>(
        cur.uleb128()) * cie.data_alignment_factor;
    ctx.register_rules.emplace(reg, offset_rule{ offset });
    break;
}
case DW_CFA_offset_extended_sf: {
    auto reg = cur.uleb128();
    auto offset = cur.sleb128() * cie.data_alignment_factor;
    ctx.register_rules.emplace(reg, offset_rule{ offset });
    break;
}
case DW_CFA_val_offset: {
    auto reg = cur.uleb128();
    auto offset = static_cast<std::int64_t>(
        cur.uleb128()) * cie.data_alignment_factor;
    ctx.register_rules.emplace(reg, val_offset_rule{ offset });
    break;
}
```

```
case DW_CFA_val_offset_sf: {
    auto reg = cur.uleb128();
    auto offset = cur.sleb128() * cie.data_alignment_factor;
    ctx.register_rules.emplace(reg, val_offset_rule{ offset });
    break;
}
case DW_CFA_register: {
    auto reg = cur.uleb128();
    ctx.register_rules.emplace(
        reg, register_rule{ static_cast<std::uint32_t>(cur.uleb128()) });
    break;
}
case DW_CFA_expression:
    sdb::error::send("DWARF expressions not yet implemented");
case DW_CFA_val_expression:
    sdb::error::send("DWARF expressions not yet implemented");
case DW_CFA_restore_extended: {
    auto reg = cur.uleb128();
    ctx.register_rules.emplace(reg, ctx.cie_register_rules.at(reg));
    break;
}
```

Each instruction takes a `ULEB128` representing the DWARF register whose rule it should change. First, the `DW_CFA_undefined` instruction sets the rule to `undefined`, while `DW_CFA_same_value` sets it to `same_value`.

The next four instructions take an additional argument representing an offset `N`. The `DW_CFA_offset_extended` instruction takes a `ULEB128` operand and sets the rule to `offset(N * data_alignment_factor)`. The `DW_CFA_offset_extended_sf` instruction takes an `SLEB128` operand and sets the rule to `offset(N * data_alignment_factor)`. The `DW_CFA_val_offset` instruction takes a `ULEB128` operand and sets the rule to `val_offset(N * data_alignment_factor)`. `DW_CFA_val_offset_sf` takes an `SLEB128` operand and sets the rule to `val_offset(N * data_alignment_factor)`.

Next, `DW_CFA_register` takes an additional `ULEB128` operand that represents a DWARF register number `R` and sets the rule for the first register operand to `register(R)`. `DW_CFA_expression` takes an additional `DW_FORM_block` operand that represents a DWARF expression `E` and sets the rule for the register to `expression(E)`. Prior to the execution of `E`, it will push the CFA onto the DWARF expression stack. (You'll learn what this means in Chapter 19.) `DW_CFA_val_expression` is similar to the previous instruction, except it sets the rule for the register to `val_expression(E)`. `DW_CFA_restore_extended` restores the register rule to the one in `cie_register_rules`.

Finally, we implement the two stack operations:

```
--snip--
case DW_CFA_remember_state:
    ctx.rule_stack.push_back({ ctx.register_rules, ctx.cfa_rule });
    break;
```

```
            case DW_CFA_restore_state:
                ctx.register_rules = ctx.rule_stack.back().first;
                ctx.cfa_rule = ctx.rule_stack.back().second;
                ctx.rule_stack.pop_back();
                break;
            }
        }
    }
}
```

The `DW_CFA_remember_state` instruction pushes the current register rules onto the rule stack without modifying the current ruleset, while `DW_CFA_restore_state` sets the current register rules to the ruleset on the top of the rule stack and then removes that ruleset from the stack.

We covered many instructions, but we're now done with the instruction interpreter. All that's left to make the `unwind` function work is to implement execution of register rules.

Executing Register Rules

To execute the register rules, we should make a copy of the old register values, loop over the set of rules, and modify the register values based on the relevant rule. The register rules have the type `std::variant<undefined_rule, same_rule, offset_rule, val_offset_rule, register_rule>`, so we must check a few options to figure out which type a rule stores and then operate on it. For the sake of simplicity, we'll opt to do this in an `if...then...else` chain. Implement `execute_unwind_rules` in *sdb/src/dwarf.cpp*:

```
namespace {
    sdb::registers execute_unwind_rules(
          unwind_context& ctx, sdb::registers& old_regs,
          const sdb::process& proc) {
        auto unwound_regs = old_regs;

        auto cfa_reg_info = sdb::register_info_by_dwarf(ctx.cfa_rule.reg);
        auto cfa = std::get<std::uint64_t>(
            old_regs.read(cfa_reg_info)) + ctx.cfa_rule.offset;
        old_regs.set_cfa(sdb::virt_addr{ cfa });
        unwound_regs.write_by_id(sdb::register_id::rsp, { cfa }, false);

        for (auto [reg, rule] : ctx.register_rules) {
            auto reg_info = sdb::register_info_by_dwarf(reg);

            if (auto undef = std::get_if<undefined_rule>(&rule)) {
                unwound_regs.undefine(reg_info.id);
            }
```

```
            else if (auto same = std::get_if<same_rule>(&rule)) {
                // Do nothing.
            }
            else if (auto reg = std::get_if<register_rule>(&rule)) {
                auto other_reg = sdb::register_info_by_dwarf(reg->reg);
                unwound_regs.write(reg_info, old_regs.read(other_reg), false);
            }
            else if (auto offset = std::get_if<offset_rule>(&rule)) {
                auto addr = sdb::virt_addr{ cfa + offset->offset };
                auto value = sdb::from_bytes<std::uint64_t>(
                    proc.read_memory(addr, 8).data());
                unwound_regs.write(reg_info, { value }, false);
            }
            else if (auto val_offset = std::get_if<val_offset_rule>(&rule)) {
                auto addr = cfa + val_offset->offset;
                unwound_regs.write(reg_info, { addr }, false);
            }
        }
        return unwound_regs;
    }
}
```

First, we make a copy of the old registers. We then calculate the CFA by reading the value of the register specified in the CFA rule and adding the supplied offset. We set the CFA for the old register set. The CFA for this frame marks both the beginning of the current stack frame and the end of the previous stack frame, which will have been the value of the stack pointer for the previous stack frame, so we restore `rsp` to the value of the CFA.

We then loop over all the register rules and apply them to the copy we made of the registers. For `undefined_rule`, we call `undefine` on the new register set with the supplied register ID. If the rule is `same_rule`, we don't have to do anything for that register: restoring it is a no-op. For `register_rule`, the old value is stored in the given register, so we read it and write the value into the register we're restoring (making sure to supply `false` for the `commit` argument so we don't write this value back into the process). For `offset_rule`, we read the memory at the current CFA plus the given offset and store that in the register. The `val_offset` value is similar, but we simply write the value of the CFA plus the offset rather than reading the memory there. Finally, we return the new register set.

Now that we can unwind a single stack frame, we can move on to unwinding the whole stack, including handling inlined frames.

Unwinding the Stack

We'll place the stack unwinding code in (you guessed it) `sdb::stack`. We'll need a new type to represent a stack frame. We must also create a place to store frames in `sdb::stack` and functions to track and manipulate the current set of frames.

Creating a Stack Frame Type

Let's start by defining a new `sdb::stack_frame` type. This type should wrap the set of registers for that frame, the address to report in backtraces for the frame, the DIE representing the function to which the frame belongs, whether the frame was inlined, and the source location to which it points. Add this type to *sdb/include/libsdb/stack.hpp*:

```
#include <libsdb/registers.hpp>

namespace sdb {
    struct stack_frame {
        registers regs;
        virt_addr backtrace_report_address;
        die func_die;
        bool inlined = false;
        source_location location;
    };
}
```

We need to hold a specific member for the backtrace report address because when we unwind the stack, the program counter value of caller frames will point to the instruction after the call instruction, where the callee will return after completion. We want to report the address of the call instruction for backtraces, so we'll calculate it as we unwind.

Tracking and Manipulating Frames

Now we can augment `sdb::stack` with the functionality needed to work with stack frames:

```
#include <libsdb/types.hpp>

namespace sdb {
    class stack {
    public:
        --snip--
        void simulate_inlined_step_in() {
            --inline_height_;
            current_frame_ = inline_height_;
        }

        void unwind();
        void up() { ++current_frame_; }
        void down() { --current_frame_; }

        span<const stack_frame> frames() const;
        bool has_frames() const { return !frames_.empty(); }
        const stack_frame& current_frame() const { return frames_[current_frame_]; }
```

```
        std::size_t current_frame_index() const {
            return current_frame_ - inline_height_;
        }

        const registers& regs() const;
        virt_addr get_pc() const;

    private:
        --snip--
        std::vector<stack_frame> frames_;
        std::size_t current_frame_ = 0;
    };
}
```

We add data members to track the current set of stack frames, as well as the frame that the debugger is currently examining. We then modify `simulate_inlined_step_in` to also update the current frame whenever a simulated step in occurs. We add an `unwind` function that will fill in `frames_`, which we'll call whenever the process stops. The `up` and `down` functions change the current frame being debugged.

The `frames` function will return the current set of frames, not including those that the compiler inlined (which we're pretending the process hasn't yet entered). The `has_frames`, `current_frame`, and `current_frame_index` functions are all simple retrieval functions, so we implement them inline. Finally, we add some convenience helpers to get the set of registers and the program counter for the current frame.

Let's implement these functions in *sdb/src/stack.cpp*. The `unwind` function is the most complex by far, so we'll tackle it last. We'll start with `sdb::stack::frames` in *sdb/src/stack.cpp*:

```
sdb::span<const sdb::stack_frame>
sdb::stack::frames() const {
    return { frames_.data() + inline_height_,
             frames_.size() - inline_height_ };
}
```

All frames below `inline_height_` in the `frames_` member are inlined frames that we're pretending the process hasn't yet entered. Therefore, we return a span that begins at an offset of `inline_height_` into the `frames_` vector and has a size equal to that vector minus the inline height.

Next, the `regs` function looks like this:

```
const sdb::registers& sdb::stack::regs() const {
    return frames_[current_frame_].regs;
}
```

We simply grab the current frame from the list and return a reference to its `regs` member.

The `get_pc` function just returns the program counter from the result of `regs`:

```
sdb::virt_addr sdb::stack::get_pc() const {
    return virt_addr{
        regs().read_by_id_as<std::uint64_t>(sdb::register_id::rip)
    };
}
```

Now we move on to arguably the most difficult function we've had to implement in this chapter: `sdb::stack::unwind`. Recall that we'll call this function every time the process stops and that it should recompute the current set of stack frames. Let's call it from `sdb::target::notify_stop`. Replace the current implementation of that function in *sdb/src/target.cpp* with this:

```
void sdb::target::notify_stop(const sdb::stop_reason& reason) {
    stack_.unwind();
}
```

The algorithm for the unwinder is quite involved, especially when it comes to dealing with inlined frames. As such, we'll work through two full examples before implementing the `unwind` function.

Understanding the Unwinding Algorithm

The best way to understand the unwinding algorithm is to walk through some examples that highlight the differences between inlined and non-inlined functions. Consider the following code, from a file called *force_inline.cpp*:

```
#include <cstdio>

__attribute__((always_inline))
void inlined() {
    puts("I am inlined");
}

void not_inlined() {
    inlined();
    puts("I am not inlined");
}

int main() {
    not_inlined();
}
```

The `always_inline` attribute inlines the function that follows it into all call sites. As a result, this code contains one inlined function and one non-inlined one. The generated assembly code for this program looks something like this:

```
not_inlined:
1163:       endbr64
```

```
1167:       push   %rbp
1168:       mov    %rsp,%rbp
116b:       lea    0xe92(%rip),%rax
1172:       mov    %rax,%rdi
1175:       call   puts
117a:       nop
117b:       lea    0xe8f(%rip),%rax
1182:       mov    %rax,%rdi
1185:       call   puts
118a:       nop
118b:       pop    %rbp
118c:       ret

main:
118d:       endbr64
1191:       push   %rbp
1192:       mov    %rsp,%rbp
1195:       call   not_inlined
119a:       mov    $0x0,%eax
119f:       pop    %rbp
11a0:       ret
```

In `not_inlined`, the instructions ranging from file address `0x116b` to `0x117a` represent the inlined body of the `inlined` function. The relevant DWARF information for this code looks like this:

```
DW_TAG_subprogram
  DW_AT_external              yes(1)
  DW_AT_name                  not_inlined
  DW_AT_decl_file             0x00000001 force_inline.cpp
  DW_AT_decl_line             0x00000008
  DW_AT_decl_column           0x00000006
  DW_AT_low_pc                0x00001163
  DW_AT_high_pc               0x0000118d
  DW_TAG_inlined_subroutine
    DW_AT_abstract_origin       <points to following DW_TAG_subprogram>
    DW_AT_low_pc                0x0000116b
    DW_AT_high_pc               0x0000117b
    DW_AT_call_file             0x00000001 force_inline.cpp
    DW_AT_call_line             0x00000009
    DW_AT_call_column           0x0000000c

DW_TAG_subprogram
  DW_AT_name                  inlined
  DW_AT_decl_file             0x00000001 /force_inline.cpp
  DW_AT_decl_line             0x00000004
  DW_AT_decl_column           0x00000006
  DW_AT_inline                DW_INL_inlined
```

Its line table looks like this:

```
Source lines (from CU-DIE at .debug_info offset 0x0000000b):

            NS new statement, BB new basic block, ET end of text sequence
            PE prologue end, EB epilogue begin
            IS=val ISA number, DI=val discriminator value
<pc>        [lno,col] NS BB ET PE EB IS= DI= uri: "filepath"
0x00001149  [   4,16] NS uri: "force_inline.cpp"
0x00001151  [   5, 9] NS
0x00001160  [   6, 1] NS
0x00001163  [   8,20] NS
0x0000116b  [   5, 9] NS
0x0000117a  [   6, 1] NS
0x0000117b  [  10, 9] NS
0x0000118a  [  11, 1] NS
0x0000118d  [  13,12] NS
0x00001195  [  14,16] NS
0x0000119a  [  15, 1] NS
0x000011a1  [  15, 1] NS ET
```

Now let's say that the process has stopped and that the program counter points to the load address of the instruction at file address `0x1185` (the second `call` instruction inside `non_inlined`). What stack frames should we report, and what should their contents be? Let's focus on just the contents of the `rip` register, the source location that instruction is part of, and whether the frame represents an inlined function call.

At the beginning of stack unwinding, `rip` will be `0x1185` (assuming the load address is the same as the file address, for simplicity). This program counter value falls between the `DW_AT_low_pc` and `DW_AT_high_pc` values for the `not_inlined` DIE:

```
DW_AT_low_pc                0x00001163
DW_AT_high_pc               0x0000118d
```

This value doesn't fall within the range of a `DW_TAG_inlined_subroutine` DIE, so we know that there are no inlined frames at this program counter value. To get the source location, we look up the line table entry corresponding to `0x1185`:

```
0x0000117b  [  10, 9] NS
```

This is the last entry whose address doesn't exceed the value we're looking for, so it's the correct entry. As such, the current stack frame looks like this:

Program counter `0x1185`

Source location `force_inline.cpp:10`

Inlined `false`

When we unwind a single frame, we should have the program counter value `0x119a`. This part is very important: when we unwind a stack frame, the program counter value of the unwound frame will point to the instruction after the function call, not to the function call instruction itself. This is because the unwound program counter value is the return address of the function down the stack.

However, users expect the debugger to report the functions above the deepest frame as stopped on function calls, so we need to fix the program counter values we receive when executing the call frame information instructions. We could examine the machine code to find the start of the call instruction, but an easier approach is to just subtract 1 from the unwound PC value and then look up that value in the line table. Carrying this out for the value `0x119a` gives us the entry `0x00001195 [ 14,16] NS`. Thus, we can generate the following stack frame for the `main` function:

Program counter `0x1195`

Source location `force_inline.cpp:14`

Inlined `false`

Further frames would give program counter values that lie outside of this ELF file, so we stop unwinding there.

The stack unwinding process is a bit more complicated when inlining is involved. Let's say that the process has stopped with the program counter pointing to the load address of the instruction at file address `0x1175` (the `call` instruction inside the inlined `inlined` function).

At the beginning of stack unwinding, `rip` will be `0x1175`. We can tell that there is an inlined frame here because `0x1175` falls within the range of a `DW_TAG_inlined_subroutine` DIE:

```
DW_TAG_inlined_subroutine
  DW_AT_abstract_origin       <points to following DW_TAG_subprogram>
  DW_AT_low_pc                0x0000116b
  DW_AT_high_pc               0x0000117b
  DW_AT_call_file             0x00000001 force_inline.cpp
  DW_AT_call_line             0x00000009
  DW_AT_call_column           0x0000000c
```

Because an inlined frame exists at this program counter value, we need to report two logical stack frames for this location. If `inlined` called a function that was itself inlined, we'd need to report three logical stack frames; the number of frames to report is theoretically unbounded.

Let's try to compute a stack frame for the `inlined` function. We already have the program counter, and we retrieve the following source location from the line table: `0x0000116b [ 5, 9] NS`. The frame looks like this:

Program counter `0x1175`

Source location `force_inline.cpp:5`

Inlined `true`

We now need to work out what the stack frame for the outer `non_inlined` function should look like. The program counter shouldn't be `0x1175`: it should point to the start of the inlined frame because this is the machine instruction that most closely corresponds to the logical function call. Likewise, the source location should point to the location of the call to `inlined` within `non_inlined`.

We retrieve the program counter value by looking at the `DS_AT_low_pc` value of the inlined subroutine DIE: `0x0000116b`. We don't retrieve the source location by looking up this value in the line table because the line table would report the source for the inlined function. Instead, we get this value from the `DW_AT_call_*` attributes on the inlined subroutine DIE:

```
DW_AT_call_file             0x00000001 force_inline.cpp
DW_AT_call_line             0x00000009
```

The frame looks like this:

Program counter `0x116b`

Source location `force_inline.cpp:9`

Inlined `false`

We'd then follow the same process as the example with no inlining for unwinding the final frame representing the `main` function, which gives us this frame:

Program counter `0x1195`

Source location `force_inline.cpp:14`

Inlined `false`

After unwinding to the `main` function, we're done.

Writing the Unwinding Function

Now that we've manually walked through the unwinding process, let's try implementing `unwind` piece by piece in *sdb/src/stack.cpp*. As we removed the call to `reset_inline_height` in `sdb::target::notify_stop`, that function should be the first thing we call inside `unwind`. We should also set the current frame to whatever inline height we calculated:

```
void sdb::stack::unwind() {
    reset_inline_height();
    current_frame_ = inline_height_;
```

Next, we'll declare some variables to store the current program counter as both a virtual address and a file address. We'll also store the process and current registers, then clear the old set of frames:

```
    --snip--
    auto virt_pc = target_->get_process().get_pc();
```

```
auto file_pc = target_->get_pc_file_address();
auto& proc = target_->get_process();
auto regs = proc.get_registers();

frames_.clear();
```

We'll ensure that the program counter points to a valid ELF file and then keep unwinding frames until we hit a frame with a program counter that lives outside of the ELF file we're operating on (indicating that this function belongs to some shared library or that we've hit the topmost frame):

```
--snip--
auto elf = file_pc.elf_file();
if (!elf) return;

while (virt_pc.addr() != 0 and elf == &target_->get_elf()) {
    // Create stack_frame objects and unwind another frame.
}
```

For every frame, we'll calculate the inline stack at the program counter for that frame. If we encounter inlined functions, we'll need to create multiple `sdb::stack_frame` objects for the physical frame and push them into `frames_`. If we find no inlined functions, we need to create and push only a single frame.

We'll write a `create_base_frame` function that creates the frame for the bottommost function in the inline stack (which may be the only frame in the inline stack) and write a second function called `create_inline_stack_frames` that handles the additional multiple frames we need to create for inline stacks. After updating `frames_`, we unwind another frame using the call frame information and update the program counter values that our loop uses:

```
--snip--
while (virt_pc.addr() != 0 and elf == &target_->get_elf()) {
    auto& dwarf = elf->get_dwarf();
    auto inline_stack = dwarf.inline_stack_at_address(file_pc);
    if (inline_stack.empty()) return;

    if (inline_stack.size() > 1) {
        create_base_frame(regs, inline_stack, file_pc, true);
        create_inline_stack_frames(regs, inline_stack, file_pc);
    }
    else {
        create_base_frame(regs, inline_stack, file_pc, false);
    }
    regs = dwarf.cfi().unwind(proc, file_pc, frames_.back().regs);
    virt_pc = virt_addr{
        regs.read_by_id_as<std::uint64_t>(register_id::rip) - 1
    };
    file_pc = virt_pc.to_file_addr(target_->get_elf());
```

```
        elf = file_pc.elf_file();
    }
}
```

We grab the DWARF information needed to calculate the inline stack. If we get no frame information, we return. We then call the `create_*` functions, which require the set of registers and the program counter on which we're currently operating. We also pass the inline stack so we don't have to recompute it. For inline stacks, we pass `true` as the last argument to `create_base_frame` to indicate that that frame should be marked as inlined. Conversely, we pass `false` if we find no inline stack, to mark it as not inlined.

Note that after we unwind a frame, we subtract 1 from the program counter value in the returned frame before looping. Recall from the worked examples that the return address points to the instruction after the function call in the caller, but we need to continue unwinding from the function call instruction itself. Subtracting 1 from the program counter puts it into the range of the previous instruction; our line table parser can then find a line table entry that gives the starting address for that instruction. Finally, we compute the new program counter as a file address and extract the ELF file for it (which may be `nullptr` in the case that the program counter points inside code for a shared library or code that is just-in-time compiled).

Declare those two `create_*` functions in *sdb/include/libsdb/stack.hpp*:

```
namespace sdb {
    class stack {
    public:
        --snip--

    private:
        void create_inline_stack_frames(
            const sdb::registers& regs,
            const std::vector<sdb::die> inline_stack,
            file_addr pc);

        void create_base_frame(
            const registers& regs,
            const std::vector<sdb::die> inline_stack,
            file_addr pc,
            bool inlined);

        --snip--
    };
}
```

Now we'll implement `create_base_frame`, as it's the simpler of the functions. It should add a new stack frame to `frames_`:

```
void sdb::stack::create_base_frame(
      const registers& regs,
```

```
      const std::vector<sdb::die> inline_stack,
      file_addr pc,
      bool inlined) {
    auto backtrace_pc = pc.to_virt_addr();
    auto line_entry = pc.elf_file()->get_dwarf().line_entry_at_address(pc);
    if (line_entry != line_table::iterator{})
        backtrace_pc = line_entry->address.to_virt_addr();

    frames_.push_back({ regs, backtrace_pc, inline_stack.back(), inlined });
    frames_.back().location = source_location{
        line_entry->file_entry, line_entry->line };
}
```

If this isn't the bottommost frame of the entire call stack, the program counter passed to this function points inside the call instruction that led to the function for this frame, so we first need to find the start of the call instruction. We do this by finding the line table entry that corresponds to that instruction and then calculating its start address as a virtual address. If this frame is the bottommost frame of the entire call stack, following this process does no harm and shouldn't modify the program counter value. (The approach creates additional overhead, but the code is simpler.)

We create a new frame with the given registers, the calculated call instruction address, and the DIE at the end of the inline stack. We also pass the `inlined` parameter we were given to mark the frame in the correct manner. Finally, we update the location of the frame to point to the relevant file and line.

The last piece of the puzzle is the `create_inline_stack_frames` function. This function should iterate backward over all frames other than the base frame, creating `sdb::stack_frame` objects as it goes:

```
void sdb::stack::create_inline_stack_frames(
      const registers& regs,
      const std::vector<sdb::die> inline_stack,
      file_addr pc) {
    for (auto it = inline_stack.rbegin() + 1; it != inline_stack.rend(); ++it) {
        auto inlined_pc = std::prev(it)->low_pc().to_virt_addr();
        frames_.push_back(stack_frame{ regs, inlined_pc, *it });
        frames_.back().inlined = std::next(it) != inline_stack.rend();
        frames_.back().location = std::prev(it)->location();
    }
}
```

We walk backward over the frames of the inline stack, save for the base one (which we've already handled), and create entries for each of them. As such, we use `rbegin` and `rend` rather than `begin` and `end` to loop backward, and we add 1 to the begin iterator to start at the second-to-last element.

For each frame, we add a new `sdb::stack_frame` object to `frames_`. First, we compute the address at which the frame below this one was inlined, which is

given by its low_pc value. We create a new sdb::stack_frame with the same set of registers as the base frame, the computed inline address, and the DIE that points to the relevant function.

All frames other than the first one on inline_stack should be marked as inlined, so we set the inlined member of the frame based on whether incrementing this iterator would get you to the beginning of the list (not to the end, because we're going backward). We also update the location for this frame based on the inline address.

If you've gotten this far, you've successfully implemented a full stack unwinder. This is one of the most complex parts of the entire book, so take some time to congratulate yourself. Next, we'll hook the unwinder into the user interface, which is considerably simpler.

Exposing Stack Unwinding to the User

We must augment the debugger's user interface in a few ways. First, we'll add up and down commands to change the current stack frame being examined. We'll also make register read commands operate on the current stack frame and add a backtrace command.

We won't support the writing of registers on the current stack frame, as doing so isn't possible in every situation (for example, for undefined registers) and in some cases, it could corrupt the program state (for example if the current value of a register_rule is also present in the callee).

Let's start with the up and down commands.

Adding up and down Commands

The up and down commands should call the up and down functions on the target's stack and then print the current code location. Let's factor the code printing functionality out of handle_stop so we can reuse it. Modify handle_stop in *sdb/tools/sdb.cpp* like this:

```
namespace {
    void handle_stop(sdb::target& target, sdb::stop_reason reason) {
        print_stop_reason(target, reason);
        if (reason.reason == sdb::process_state::stopped) {
            print_code_location(target);
        }
    }
}
```

We take the body out of the if statement and replace it with a call to print_code_location, which we'll implement next. While we move that old code to print_code_location, we can also simplify it, because the stack now tells us what code we should be printing:

```
namespace {
    void print_code_location(sdb::target& target) {
```

```
        if (target.get_stack().has_frames()) {
            auto& frame = target.get_stack().current_frame();
            print_source(frame.location.file->path, frame.location.line, 3);
        }
        else {
            print_disassembly(target.get_process(), target.get_process().get_pc(), 5);
        }
    }
}
```

If we have sufficient debug information to unwind the stack, we grab the current frame and print the corresponding code, along with three lines of context. Otherwise, we print the dissassembly around the program counter.

We'll add commands to the `handle_command` function to update the current frame and print the code location. In *sdb/tools/sdb.cpp*, create these branches above the handler for the `disassemble` command, so that `d` runs the `down` command rather than `disassemble`:

```
namespace {
    void handle_command(
          std::unique_ptr<sdb::target>& target,
          std::string_view line) {
        --snip--
        else if (is_prefix(command, "up")) {
            target->get_stack().up();
            print_code_location(*target);
        }
        else if (is_prefix(command, "down")) {
            target->get_stack().down();
            print_code_location(*target);
        }
        --snip--
    }
}
```

Also add help information for these commands to `print_help`:

```
namespace {
    void print_help(const std::vector<std::string>& args) {
        if (args.size() == 1) {
            std::cerr << R"(Available commands:
    --snip--
    down        - Select the stack frame below the current one
    --snip--
    up          - Select the stack frame above the current one
    --snip--
)";
        }
        --snip--
```

```
    }
}
```

Now we can move on to reading registers.

Reading Registers from Other Frames

Because users can now use the `up` and `down` commands to select the active stack frame, they'll expect any registers reported by the debugger to correspond to the selected stack frame rather than the deepest one. Let's modify the current register commands to do so.

As the register commands must look at the call stack, we need to pass the `sdb::target`, rather than the `sdb::process`, to `handle_register_command`. Modify the call inside `handle_command`:

```
namespace {
    void handle_command(
          std::unique_ptr<sdb::target>& target,
          std::string_view line) {
        --snip--
        else if (is_prefix(command, "register")) {
            handle_register_command(*target, args);
        }
        --snip--
    }
}
```

Update `handle_register_command` to take the target and then forward it to `handle_register_read`:

```
namespace {
    void handle_register_command(
        sdb::target& target,
        const std::vector<std::string>& args) {
        --snip--
        if (is_prefix(args[1], "read")) {
            handle_register_read(target, args);
        }
        else if (is_prefix(args[1], "write")) {
            handle_register_write(target.get_process(), args);
        }
        --snip--
    }
}
```

Update the first parameter to be `sdb::target& target` rather than the process and then pass `target` to `handle_register_read`. Also update the call to `handle_register_write` to grab the `process` object out of the target.

Next, modify `handle_register_read` so it retrieves the registers to print from the call stack rather than from the process. Also handle undefined registers:

```
namespace {
    void handle_register_read(
       ❶ sdb::target& target,
         const std::vector<std::string>& args) {
        auto format = [](auto t) {
            --snip--
        };

     ❷ auto& regs = target.get_stack().regs();
        auto print_register_value = [&](auto info) {
            if (regs.is_undefined(info.id)) {
                fmt::print("{}:\tundefined\n", info.name);
            }
            else {
                auto value = regs.read(info);
                fmt::print("{}:\t{}\n", info.name, std::visit(format, value));
            }
        };

        if (args.size() == 2 or
            (args.size() == 3 and args[2] == "all")) {
            for (auto& info : sdb::g_register_infos) {
                if (args.size() == 3 or info.type == sdb::register_type::gpr) {
                    print_register_value(info);
                }
            }
        }
        else if (args.size() == 3) {
            try {
                auto info = sdb::register_info_by_name(args[2]);
                print_register_value(info);
            }
            catch (sdb::error& err) {
                std::cerr << "No such register\n";
                return;
            }
        }
        else {
            print_help({ "help", "register" });
        }
    }
}
```

Update the first parameter to be a reference to the target ❶. Grab a reference to the registers for the current stack frame, and declare a lambda function that prints out the value of a register or an indication that the register is undefined ❷. Then, modify the blocks that carry out the register printing and make them call `print_register_value`.

Printing the Backtrace

Now we can move on to printing a backtrace. Add a `print_backtrace` function to *sdb/tools/sdb.cpp* that prints a summary of the call stack to the console:

```
namespace {
    void print_backtrace(const sdb::target& target) {
        auto& stack = target.get_stack();
        auto i = 0;
        for (auto& frame : stack.frames()) {
            auto pc = frame.backtrace_report_address;
            auto func_name = target.function_name_at_address(pc);

            std::string message = i == stack.current_frame_index() ? "*" : " ";
            message += fmt::format("[{}]: {:#x} {}", i++, pc.addr(), func_name);
            if (frame.inlined) {
                message += fmt::format(" [inlined] {}", *frame.func_die.name());
            }
            fmt::print("{}\n", message);
        }
    }
}
```

We retrieve the call stack from the target and initialize a loop counter for the stack frames. We then loop over each frame, printing out a summary of it. For each frame, we grab the address that the unwinder says to use for backtrace reports and retrieve the function name that corresponds to it. Next, we compute a message to print out. If the loop counter matches the current frame index, we begin the summary line with a star (*) to indicate that it is the active frame. We then append the loop counter, the program counter value, and the function name. If this frame is inlined, we add a marker to it and give the name of the inlined function. Finally, we print out the message.

Add a command to `handle_command` that calls `print_backtrace`:

```
namespace {
    void handle_command(
          std::unique_ptr<sdb::target>& target,
          std::string_view line) {
        --snip--
        else if (is_prefix(command, "backtrace")) {
            print_backtrace(*target);
```

```
        }
        --snip--
    }
}
```

We're done with the main backtracing implementation! We can now go back to our implementation of `sdb::target::step_out` and use the stack unwinder rather than the `rbp` register. Also, because we know how many stack frames there currently are, we can support stepping out of recursive functions by repeatedly running up to the return address until a stack frame is popped. In *sdb/src/target.cpp*, replace the definitions of the `frame_pointer`, `return_address_bytes`, and `return_address` variables with this:

```
    --snip--
    auto& regs = stack.frames()[stack.current_frame_index() + 1].regs;
    virt_addr return_address{
        regs.read_by_id_as<std::uint64_t>(register_id::rip)
    };

    sdb::stop_reason reason;
    for (auto frames = stack.frames().size();
        stack.frames().size() >= frames;) {
        reason = run_until_address(return_address);
        if (!reason.is_breakpoint()
            or process_->get_pc() != return_address) {
            return reason;
        }
    }
    return reason;
    --snip--
```

We grab the stack frame above this one and retrieve the program counter value for that frame, which will be the return address for the current frame. We then keep running up to that address until the number of stack frames is less than it was at the start of the stepping procedure. As usual, if the process halts for some other reason, we return that reason.

Testing

Now we can write a small test to validate the code. Launch *test/targets/step*, set a breakpoint on `scratch_ears`, and continue the process. Because there are two levels of inlining at the first instruction of `scratch_ears`, the process halts two frames above that function, in `find_happiness`. Step in twice to get inside `scratch_ears` and then generate a backtrace. It should look like this:

```
$ tools/sdb test/targets/step
Launched process with PID 3046
sdb> break set scratch_ears
sdb> c
```

```
Process 1716 stopped with signal TRAP at 0x555555555195,
step.cpp:5 (find_happiness) (breakpoint 1)
  12 }
  13
  14 void find_happiness() {
> 15     pet_cat();
  16     std::puts("Found happiness");
  17 }
  18
  19 int main() {
  20
sdb> step
Process 1716 stopped with signal TRAP at 0x555555555195,
step.cpp:5 (find_happiness) (single step)
   7
   8 __attribute__((always_inline))
   9 void pet_cat() {
> 10     scratch_ears();
  11     std::puts("Done petting cat");
  12 }
  13
  14 void find_happiness() {
  15
sdb> step
Process 1716 stopped with signal TRAP at 0x555555555195,
step.cpp:5 (find_happiness) (single step)
  2
  3 __attribute__((always_inline))
  4 void scratch_ears() {
> 5     std::puts("Scratching ears");
  6 }
  7
  8 __attribute__((always_inline))
  9 void pet_cat() {
  10
sdb> back
*[0]: 0x555555555195 find_happiness [inlined] scratch_ears
 [1]: 0x555555555195 find_happiness [inlined] pet_cat
 [2]: 0x555555555195 find_happiness
 [3]: 0x5555555551cf main
```

Note that the first three frames all have the same program counter value due to the inlining.

Let's turn this exercise into an automated test. Add a new test case to *sdb/test/tests.cpp*:

```
TEST_CASE("Stack unwinding", "[unwind]") {
    auto target = target::launch("targets/step");
    auto& proc = target->get_process();
```

```
    target->create_function_breakpoint("scratch_ears").enable();
    proc.resume();
    proc.wait_on_signal();
    target->step_in();
    target->step_in();

    std::vector<std::string_view> expected_names = {
        "scratch_ears",
        "pet_cat",
        "find_happiness",
        "main"
    };

    auto frames = target->get_stack().frames();
    for (auto i = 0; i < frames.size(); ++i) {
        REQUIRE(frames[i].func_die.name().value()
            == expected_names[i]);
    }
}
```

We follow the same steps as before: launch the test program, set a breakpoint on `scratch_ears`, resume, step in twice, and then check the backtrace. We ensure that the functions reported by the backtrace are the ones we expect. This test should pass.

This is a very small test, but if we completely break stack unwinding, we'll know about it.

Summary

In this chapter, you learned how ELF files express stack unwinding information. You wrote a parser for this information and created a function to unwind a frame of the call stack based on it. You then implemented a full stack unwinder based on this facility. Using this unwinder, you augmented the debugger with the ability to change the active stack frame being debugged, read register values from different frames, and generate a backtrace.

In the next chapter, we'll take a break from DWARF parsing to look at how processes load shared libraries and how we can trace this loading.

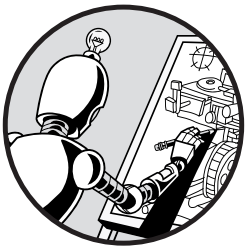
Check Your Knowledge

1. What register restore rules does DWARF call frame information support?
2. Why is the `undefined` register rule necessary?
3. What CFA rules can be expressed in DWARF call information?
4. How does inlining complicate stack unwinding?

17

SHARED LIBRARIES

We walk the corridors, searching the shelves and rearranging them, looking for lines of meaning amid leagues of cacophony and incoherence, reading the history of the past and our future, collecting our thoughts and collecting the thoughts of others, and every so often glimpsing mirrors, in which we may recognize creatures of the information.
–Jorge Luis Borges, "The Library of Babel"



Until now, we've restricted our discussion to code that lives inside the executable that holds the program's entry point. However, many programs rely on shared libraries, whose code resides in a completely different ELF file, with a different set of DWARF information.

To further complicate matters, programs don't necessarily load shared libraries when they start, as they may open the libraries at runtime using `dlopen`. On Linux, most plug-in systems work in this way; users of a program drop shared libraries into a known folder, and the program opens these and then calls initialization routines with known names inside of those shared libraries.

In this chapter, you'll learn how Linux systems load programs and add facilities to your debugger to trace the loading and unloading of shared libraries. Using this tracer, you'll add support for setting breakpoints and performing stack unwinding across shared library boundaries.

Because you're probably pretty tired of dealing with DWARF information at this point, you'll be happy to hear that we'll do zero DWARF parsing in the next two chapters.

Program Loading

Program loading is the process of taking an executable from the filesystem, loading it into memory, preparing it for execution, and jumping to its entry point. Recall from Chapter 11 that ELF files are split into segments. *Program headers* describe the segments of the program relevant to program loading. For example, here is a textual dump of the program headers for the *hello_sdb* program, as produced by `readelf`:

```
$ readelf -l test/targets/hello_sdb
--snip--
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000628 0x0000000000000628  R      0x1000
  LOAD           0x0000000000001000 0x0000000000001000 0x0000000000001000
                 0x0000000000000175 0x0000000000000175  R E    0x1000
  LOAD           0x0000000000002000 0x0000000000002000 0x0000000000002000
                 0x00000000000000f4 0x00000000000000f4  R      0x1000
❶ LOAD           0x0000000000002db8 0x0000000000003db8 0x0000000000003db8
                 0x0000000000000258 0x0000000000000260  RW     0x1000
  DYNAMIC        0x0000000000002dc8 0x0000000000003dc8 0x0000000000003dc8
                 0x00000000000001f0 0x00000000000001f0  RW     0x8
  NOTE           0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000030 0x0000000000000030  R      0x8
  NOTE           0x0000000000000368 0x0000000000000368 0x0000000000000368
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_PROPERTY   0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000030 0x0000000000000030  R      0x8
  GNU_EH_FRAME   0x0000000000002010 0x0000000000002010 0x0000000000002010
                 0x0000000000000034 0x0000000000000034  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000002db8 0x0000000000003db8 0x0000000000003db8
                 0x0000000000000248 0x0000000000000248  R      0x1
--snip--
```

Each pair of lines in this table describes a segment (you can pass the `-W` flag to `readelf` to show one line per entry). As you can see, each segment

specifies a type, the file offset at which it begins, a relevant file address, a relevant physical address (ignored on Linux because user-space programs aren't allowed to access physical memory directly), their byte size within the ELF file, their byte size when loaded in memory, permission flags, and segment alignment. The segment alignment is usually the size of a memory page (which is usually 4KiB on x64), and the offset and virtual address of a `LOAD` segment must differ from the alignment boundary by the same amount (in other words, they must be congruent modulo the alignment). This ensures that file pages can be `mmap`ped into memory.

The size in memory may differ from the size on disk in some cases. For example, segment 5 ❶ (segment numbers are zero-based) has the file size `0x258` but the memory size `0x260`. This occurs because the segment includes the program's uninitialized global data, a bunch of 0 bytes that the program fills in while it runs. Because there's no point in storing a bunch of 0s in the binary, this program header instead instructs the loader to allocate an extra 8 bytes for uninitialized globals.

A segment's type can be any of the following:

`LOAD` Specifies a segment to load into memory at the given file address.

`DYNAMIC` Specifies dynamic linking information.

`INTERP` Specifies the location and size of the path to the dynamic linker.

`NOTE` Specifies the location and size of auxiliary information about the binary. This segment is often used to provide information about core dumps or to communicate between the linker and C standard library.

`PHDR` Specifies where to load the program headers themselves so that other tools (like debuggers!) can read them.

`TLS` Provides information about thread-local storage.

`GNU_EH_FRAME` Locates the stack unwinding information (and points to the same memory as the `.eh_frame` section).

`GNU_PROPERTY` Specifies a special `NOTE` section that contains information specific to the dynamic linker. Points to the same memory as the `.note.gnu.property` section.

`GNU_RELRO` Some loaded segments require *relocation*, in which the dynamic linker resolves references to external symbols to their correct locations in memory. Sections of type `GNU_RELRO` mark areas of memory that should be made read-only after this relocation is complete.

`GNU_STACK` Indicates whether the stack should be executable.

You'll find most of the segment types defined in the SYSV ABI, but the `GNU_*` ones are GCC extensions.

The loading process differs significantly depending on whether the dynamic linker needs to be involved, which occurs if the executable is position-independent or depends on shared libraries (including the standard library). We'll first consider program loading in cases when the dynamic linker isn't involved and then consider the case where it is.

Static Executables

We call executables that don't require the dynamic linker *static executables*, and their program headers don't include an `INTERP` segment.

Because program loading is a privileged operation, the code for loading executables lives inside the Linux kernel. When a program requests that the kernel load a program through a call to one of the `exec` functions, the kernel first cleans up any information from the old process that it no longer needs. Examples include additional running threads, old memory maps, any file descriptors marked `O_CLOEXEC`, and signal handlers.

After cleaning up the old process, the kernel begins setting up for the new one. It allocates memory to hold the process's stack, then loops through all of the `LOAD` segments in the program headers and loads them at the correct positions.

The kernel maps the entirety of a special ELF file called the *virtual dynamic shared object (vDSO)* into the process's address space. The vDSO is a shared library that implements certain syscalls by reading directly from kernel space without having to perform a context switch into the kernel, making these syscalls run roughly four times as fast. On x64, the syscalls implemented in the vDSO are `clock_gettime`, `getcpu`, `gettime`, and `gettimeofday`.

Once it has set up the virtual address space, the kernel initializes the process's stack. At the top of the stack, it puts the auxiliary vector (which you learned about in Chapter 11), the environment variables for the process, and the command line arguments. Finally, the kernel jumps back to the program's entry point, in user space.

Let's consider how dynamic linking modifies this loading process.

Dynamic Executables

Dynamic executables list an `INTERP` segment in their program headers, and they change the loading process we just discussed in a few ways.

One way is address space layout randomization (ASLR), discussed in Chapter 7. Dynamic executables aren't necessarily position-independent, as they could be non-PIEs that depend on shared libraries, but for PIEs that don't explicitly disable ASLR, the kernel will allocate the stack at a random location and offset the file addresses specified in the program headers by a single random value.

The next big difference is that the entry point to which the kernel eventually jumps after setting up program execution isn't the entry point of the program we're trying to run, but the entry point of the dynamic linker. The dynamic linker runs in user space rather than in the kernel, which has a few benefits. For example, we can patch and improve the linker without having to reload the kernel, and there are fewer security concerns, as the linker can't perform arbitrary privileged operations. As such, the kernel loads the dynamic linker specified in the `INTERP` segment, sets up an address space in which it can run, provides it with the ELF file to be loaded, and then context-switches into the dynamic linker.

The dynamic linker has two main jobs: loading dependencies and carrying out relocations. The first job involves locating the dependencies on disk of the program it's loading and then mapping those dependencies into the virtual memory of the new process. The second job is adjusting any references that are sensitive to where parts of the program are loaded, which it does by pointing those references to the real virtual addresses of the relevant functions or variables. Such references may point elsewhere inside the main executable or to part of a shared library.

In the next two sections, I'll discuss dependency loading and relocation in more detail.

Loading Dependencies

The loading of dependencies involves a few communication channels. The dynamic linker needs information about the environment in which to run the new program, and it must know which program to load. The kernel provides these details when it loads the dynamic linker. The linker also needs information about the shared libraries on which the program depends, which it retrieves from the `.dynamic` section of the program's ELF file, pointed to by the `DYNAMIC` segment in the program headers. Finally, the dynamic linker must communicate information about the dynamic libraries it has loaded, including what their virtual addresses are, to other tools, such as our debugger. It achieves this with a *rendezvous structure* that the linker maintains in the address space of the running process.

The .dynamic Section

For the most part, the compiler and the static linker populate the `.dynamic` section when they generate the ELF file for the program being loaded. This section communicates information about dependencies and relocations to the dynamic linker. It is loaded into memory as part of the usual loading process, and the dynamic linker updates it to provide additional information to the running program and any tools that interact with it.

The `.dynamic` section consists of a list of entries. Each entry has a tag and a value, and the list ends with an entry whose tag and value are both 0. Together, the SYSV ABI, the LSB, and GNU extensions define dozens of entry types that this section can store. Many aren't relevant to us, so I won't discuss them all. Instead, I'll show you an example of the `.dynamic` section for the *hello_sdb* executable and mention some of its most interesting entries:

```
$ readelf -d test/targets/hello_sdb
Dynamic section at offset 0x2dc8 contains 27 entries:
  Tag                Type                 Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x000000000000000c (INIT)               0x1000
 0x000000000000000d (FINI)               0x1168
 0x0000000000000019 (INIT_ARRAY)         0x3db8
 0x000000000000001b (INIT_ARRAYSZ)       8 (bytes)
```

```
0x000000000000001a (FINI_ARRAY)         0x3dc0
0x000000000000001c (FINI_ARRAYSZ)       8 (bytes)
0x000000006ffffef5 (GNU_HASH)           0x3b0
0x0000000000000005 (STRTAB)             0x480
0x0000000000000006 (SYMTAB)             0x3d8
0x000000000000000a (STRSZ)              141 (bytes)
0x000000000000000b (SYMENT)             24 (bytes)
0x0000000000000015 (DEBUG)              0x0
0x0000000000000003 (PLTGOT)             0x3fb8
0x0000000000000002 (PLTRELSZ)           24 (bytes)
0x0000000000000014 (PLTREL)             RELA
0x0000000000000017 (JMPREL)             0x610
0x0000000000000007 (RELA)               0x550
0x0000000000000008 (RELASZ)             192 (bytes)
0x0000000000000009 (RELAENT)            24 (bytes)
0x000000000000001e (FLAGS)              BIND_NOW
0x000000006ffffffb (FLAGS_1)            Flags: NOW PIE
0x000000006ffffffe (VERNEED)            0x520
0x000000006fffffff (VERNEEDNUM)         1
0x000000006ffffff0 (VERSYM)             0x50e
0x000000006ffffff9 (RELACOUNT)          3
0x0000000000000000 (NULL)               0x0
```

As you can see, this section has 27 entries. You can learn more about them by looking up their names in section 11.3 of the generic part of the LSB core specification. The most interesting entries include the following:

`NEEDED` Names a shared library on which the executable depends. In this case, the program depends on *libc.so.6*, which is the C standard library. The dynamic linker will search for this library in several directories, including default system locations like */lib*, any directories named in this `.dynamic` section's `RPATH` entry, and directories specified in the `LD_LIBRARY_PATH` environment variable, among others.

`INIT` Specifies a function to call before `main`.

`FINI` Specifies a function to call after `main` exits.

`DEBUG` Specifies a pointer to the dynamic linker rendezvous structure. Note that this is `0` in the ELF file; the dynamic linker will relocate it after loading the `.dynamic` section into memory.

`PLTGOT` Specifies a pointer to the procedure linkage table, used to implement function calls across shared libraries.

`RELA` Specifies the relocations for the binary.

For our purposes, the most important of these entries is `DEBUG`. Let's look at the format of the structure to which it points at runtime.

The Rendezvous Structure

The rendezvous structure is an area of memory that the dynamic linker uses to communicate with debuggers and other tools that need to track the loading and unloading of shared libraries. Its structure is as follows:

```
struct r_debug {
    int r_version;
    struct link_map *r_map;
    ElfW(Addr) r_brk;
    enum {
        RT_CONSISTENT,
        RT_ADD,
        RT_DELETE,
    } r_state;
    ElfW(Addr) r_ldbase;
  };
```

The `r_version` field specifies the version number for the rendezvous structure. This will usually be `1`, but it will be `2` in the case that the library was opened with the `dlmopen` syscall rather than `dlopen` (note the added `m`). This syscall is not nearly as commonly used, so we're not going to worry about it. The `r_map` field is a pointer to a linked list of information about loaded dynamic libraries. I'll show you the `link_map` definition shortly.

The `r_brk` field is a pointer to the `_dl_debug_state` function inside of the dynamic linker. It's a completely empty function called before and after the linker loads or unloads a shared library. The idea is that debuggers can set a breakpoint on this function, and whenever it's hit, they can reread the `r_map` member of the structure to learn about any changes in mapped libraries.

The `r_state` field specifies the state of the loaded library list. If the member is `RT_ADD` or `RT_DELETE`, the linker is about to add or remove an entry from `r_map`, whereas if the member is `RT_CONSISTENT`, the operation is complete and the debugger can read the updated `r_map` member. Finally, the `r_ldbase` field is the load address of the dynamic linker itself.

The `link_map` type has more than a dozen fields, but we can consider most of these to be implementation details for the linker, and they may change without notice. We should rely on only five fields, shown here:

```
struct link_map {
    ElfW(Addr) l_addr;
    char *l_name;
    ElfW(Dyn) *l_ld;
    struct link_map *l_next;
    struct link_map *l_prev;
    --snip--
};
```

Each entry in the loaded library list describes information about a single shared library or the main executable. The `l_addr` entry is the virtual address at which the library is loaded, while `l_name` is the path to the loaded ELF file. In the case of the main executable, the path is empty, and for the vDSO, it's *linux-vdso.so.1*, which doesn't actually exist on disk. Otherwise, it is an absolute path to the ELF file on disk.

The `l_ld` field is the virtual address of the loaded `.dynamic` section for the object file; `l_next` is a pointer to the next item in the linked list, which terminates with a null pointer; and `l_prev` is a pointer to the previous item in the linked list. For the first entry in the list, this is a null pointer.

Note that we can't consider the rendezvous structure initialized until the dynamic linker has finished and then jumped to the real entry point of the executable it's loading.

At this point, we should have enough information to track shared library loading in our debugger. Before we implement this feature, let's discuss relocations, which will help you make sense of the dynamic linker's job and understand the assembly instructions that encode references to external symbols.

Relocations

The dynamic linker needs to resolve two important kinds of references when carrying out relocations: references to data and references to functions. Data references are much easier to deal with, so we'll consider these first.

Code in the main executable might reference a global variable inside a shared library, or vice versa. To see an example, save the following code in a file called *sdb/test/targets/marshmallow.cpp*:

```
#include <iostream>
int libmeow_client_cuteness = 100;
bool libmeow_client_is_cute();

int main() {
    std::cout << "Cuteness rating: " << libmeow_client_cuteness << '\n';
    std::cout << "Is cute: " << std::boolalpha << libmeow_client_is_cute() << '\n';
}
```

This program defines a `libmeow_client_cuteness` variable and a function called `libmeow_client_is_cute`, whose definition exists elsewhere (we'll write this next). Inside `main`, it prints out the cuteness rating and then prints out the result of calling `libmeow_client_is_cute` (`std::boolalpha` here will make it so that `true` or `false` is printed rather than `1` or `0`).

In a new *sdb/test/targets/libmeow.cpp* file, define the `libmeow_client_is_cute` function and use the `libmeow_client_cuteness` variable:

```
extern int libmeow_client_cuteness;

bool libmeow_client_is_cute() {
```

```
    return libmeow_client_cuteness > 50;
}
```

Now build a shared library from this file, create an executable from the *marshmallow.cpp* file, and link them together by adding this file to *sdb/test/targets/CMakeLists.txt*:

```
add_test_cpp_target(marshmallow)
add_library(meow SHARED "libmeow.cpp")
target_compile_options(meow PRIVATE -g -O0 -fPIC -gdwarf-4)
target_link_libraries(marshmallow PRIVATE meow)
```

The `SHARED` keyword specifies that we'd like to build a shared library rather than a static one. We pass the `-fPIC` argument to tell the compiler to output position-independent code.

The machine code that loads the value of `libmeow_client_cuteness` in both the *marshmallow* executable and *libmeow.so* must point to the same memory location for this program to run. My version of GCC has opted to use RIP-relative addressing for the load instruction inside the *marshmallow* executable; this loads the value from an address relative to the current program counter so that the address doesn't need to be relocated. You can see this by examining the disassembly (here, I've passed the `--no-show-raw-insn` flag in order to not show the raw bytes of the machine code):

```
$ objdump -D --no-show-raw-insn test/targets/marshmallow
--snip--
1252:       mov    0x2db8(%rip),%eax        # 4010 <libmeow_client_cuteness>
1258:       mov    %eax,%esi
--snip--
```

This loads the value from the address given by offsetting the virtual address of the program counter after the current instruction by `0x2db8`. As you can see in the output, the program counter after that instruction has the file address `0x1258`, so the value will be loaded from file address `0x4010`. If we dump the `.data` section of the executable, we can see that this address stores `100`, which is the value of `libmeow_client_cuteness`:

```
$ readelf -x .data test/targets/marshmallow
Hex dump of section '.data':
  0x00004000 00000000 00000000 08400000 00000000 .........@......
  0x00004010 64000000                             d...
```

The value at `0x4010` is `0x64`, which is 100 in decimal.

For the reference inside *libmeow.so*, however, the process is a bit more complicated. A simple option would be to relocate the load instruction for `libmeow_client_cuteness` inside *libmeow.so* to point to wherever that variable is located inside the *marshmallow* executable. This solution has a problem, however: the linker can't relocate the reference to `libmeow_client_cuteness` inside the code for the `libmeow_client_is_cute` function because the `.text` section is usually loaded as read-only, for security reasons. Even if it could

relocate the reference, it's possible that the program makes 10,000 references to `libmeow_client_cuteness`, in which case the linker would need to make 10,000 modifications to the code, which could significantly slow down the launching of the process.

The linker solves this problem with the global offset table.

Global Offset Table

The *global offset table (GOT)* enables updating references that reside in read-only sections. It also facilitates relocating a symbol by updating only one location, rather than every single reference to it.

The basic idea is to introduce a level of indirection to external symbol references. Instead of reading an address directly to retrieve a variable, a program can read from the global offset table to retrieve the address of the variable and then read the memory at that address to retrieve the variable itself. If the global offset table is mapped in memory as read/write and references in the `.text` section point to the global offset table, the dynamic linker can perform relocations on the GOT rather than on the `.text` section.

To see this design in the code generated for `libmeow_client_is_cude`, have a look at the disassembly:

```
$ objdump -D --no-show-raw-insn test/targets/libmeow.so
00000000000010f9 <_Z22libmeow_client_is_cutev>:
--snip--
    1101:       mov    0x2ee8(%rip),%rax    # 3ff0 <libmeow_client_cuteness>
    1108:       mov    (%rax),%eax
    110a:       cmp    $0x32,%eax
--snip--
```

The `cmp $0x32,%eax` instruction is equivalent to the `libmeow_client_cuteness > 50` statement in the code. Note, however, that there are two load instructions rather than one: first an address is retrieved from file address `0x3ff0`, and then the value of `libmeow_client_cuteness` is retrieved from the address that was just loaded. If we check the section headers of *libmeow.so*, we can see that file address `0x3ff0` lives in the global offset table, in the `.got` section:

```
$ readelf -S test/targets/libmeow.so
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
--snip--
  [17] .got              PROGBITS         0000000000003fd8  00002fd8
       0000000000000028  0000000000000008  WA       0     0     8
```

The `.got` section begins at file address `0x3fd8` and has a size of `0x28`, so it spans up to `0x4000`, and as such it contains `0x3ff0`.

We now know that the reference to `libmeow_client_cuteness` inside *libmeow.so* points to the global offset table, so next we need to understand how the entry

in the global offset table is updated to point at the definition of `libmeow_client_cuteness` inside the *marshmallow* executable. This is achieved with a relocation record.

Relocation Records

Relocation records live in sections named `.rel.<ID>` or `.rela.<ID>`, where `<ID>` specifies the element to which the relocations apply (usually a section name or `dyn`, for dynamic relocations).

Relocation records consist of a file address to which they apply, a relocation type, and additional information about how to apply the relocation that is specific to each relocation type. Some relocations also have an integer called an *addend*, which the dynamic linker adds to the computed value before committing the relocation to memory. In sections beginning with `.rela`, the addend is explicitly provided in the relocation record. In sections beginning with `.rel`, the addend is stored in memory that is to be updated. You can see the relocations for an ELF file with `readelf -r`. Here is the relocation record for *libmeow.so* that copies the data for the `libmeow_client_cuteness` variable from *libmeow.so*:

```
$ readelf -rW test/targets/libmeow.so
Relocation section '.rela.dyn' at offset 0x458 contains 8 entries:
Offset   Info             Type             Sym. Value       Sym. Name + Addend
--snip--
00003ff0 0000000400000006 R_X86_64_GLOB_DAT 0000000000000000 libmeow_client_cuteness + 0
```

This tells the dynamic linker to locate the symbol `libmeow_client_cuteness` and write the address of that symbol into file address `0x3ff0`, which is the global offset table that we looked at earlier. It's the linker's job to find a definition for `libmeow_client_cuteness` among the object files to which it has access. The type field shows the relocation type, which in this case is the x64 global data relocation type. The info field points to the symbol table entry for `libmeow_client_cuteness` (which is the symbol at index 4) and gives an enumerator value for the relocation type (`R_X86_64_GLOB_DAT` has the value `6`).

To visualize the entire relocation process I've just discussed, imagine that *marshmallow* is loaded at `0xab000000` (the load address of *libmeow.so* is irrelevant for this example). Figure 17-1 shows the references before relocation.

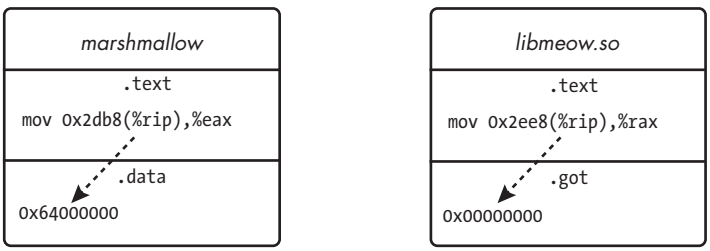


Figure 17-1: The references between object files before relocation

The reference to `libmeow_client_cuteness` inside *marshmallow* points to the value in the `.data` section for *marshmallow*. The reference inside *libmeow.so* points to an entry in `.got` that is currently not pointing anywhere (it is essentially a null pointer). Figure 17-2 shows the result of the relocation.

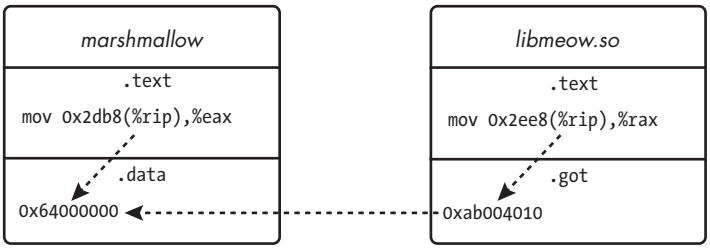


Figure 17-2: The references between object files after relocation

The dynamic linker relocated the reference to `libmeow_client_cuteness` inside *libmeow.so* by updating the global offset table entry to point to the definition inside the *marshmallow* executable.

Next, let's discuss how function calls across libraries work.

Procedure Linkage Table

You might assume that function calls work in the same way as variable references, meaning the program calls the function through a pointer in the global offset table. This is partly true, though again, there are a few problems.

First, there are usually way more references to functions across shared library boundaries than there are references to variables. Executing one relocation for every referenced shared library function on program startup can have a noticeable impact on program startup time.

Second, the instructions for direct calls and indirect calls (those made through a pointer) have different encodings and instruction lengths on x64, and some function calls use the `jmp` instruction rather than the `call` instruction. If the static linker gets object files from the compiler that make direct calls to unresolved symbols, it can't change those direct calls to indirect calls through the global offset table without disassembling the instruction on which they're performing the relocation and potentially introducing extra `nop` instructions. Relocations should be easy to apply and shouldn't require disassembling instructions for a single relocation type.

The *procedure linkage table (PLT)* solves problem one by deferring the resolution of the real function address until the function is called for the first time (a practice called *lazy binding*) and solves problem two by providing a fixed location to call, enabling the encoding of external calls as direct calls.

NOTE *Lazy binding can have security issues, since a malicious function definition may be bound at runtime. As such, toolchains often disable it by default, by setting the `BIND_NOW` flag in the ELF file's `.dynamic` section.*

Here is the PLT for the `libmeow_client_is_cute` function inside the *marshmallow* executable:

```
0000000000001040 <_Z22libmeow_client_is_cutev@plt>:
    1040:       jmp    *0x2f52(%rip)        # 3f98 <_Z22libmeow_client_is_cutev@Base>
    1046:       push   $0x1
    104b:       jmp    1020 <_init+0x20>
```

Note that I compiled this example with the `-fcf-protection=none` flag to disable certain security features that make the PLT more complicated than it needs to be.

You can see that the function consists of three instructions. The first is an indirect jump through the global offset table entry for `libmeow_client_is_cute`, which is initialized to point at the subsequent `push` instruction, at `0x1046`. This instruction pushes the value `1` onto the stack, which is an index into `.rela.plt`. The relocation at that index then informs the dynamic linker to overwrite `libmeow_client_is_cute`'s global offset table entry with the real address of the function. The entry looks like this:

```
Offset   Info             Type               Sym. Value       Sym. Name + Addend
--snip--
00003f98 0000000300000007 R_X86_64_JUMP_SLOT 0000000000000000 _Z22libmeow_client_is_cutev + 0
```

Through a bit of indirection, the next jump instruction calls the function `__dl_runtime_resolve` inside of the dynamic linker, which performs this relocation, and then jumps to the `libmeow_client_is_cute` function. On subsequent calls, the global offset table entry for `libmeow_client_is_cute` stores the real address of the function inside of *libmeow.so*. Subsequent calls to the procedure linkage table entry for `libmeow_client_is_cute` will hit that first `jmp` instruction and jump straight to the definition of `libmeow_client_is_cute`.

Tracing Shared Library Loading

With the theory out of the way, we can begin writing code to trace shared library loads. To do so, we'll follow this algorithm:

1. Set an internal breakpoint on the real entry point of the executable. When we hit the breakpoint, the dynamic linker should be initialized.
2. Walk through the loaded library list in the rendezvous structure, parsing the ELF files for every shared library noted there and adding them to a collection in `sdb::target`. We'll also dump the vDSO to disk so we can reference it in the same way as other shared libraries.
3. Set an internal breakpoint on the `_dl_debug_state` function, a pointer to which is stored in the `r_brk` member of the rendezvous structure.
4. Whenever we hit the `_dl_debug_state` function breakpoint and the `r_state` member of the rendezvous structure is `RT_CONSISTENT`, reread `r_map`, adding any new shared libraries and unloading any ones that were removed.

In a few places, the algorithm carries out certain actions when we hit a specific breakpoint. Let's add functionality to `sdb::breakpoint` to facilitate this.

Adding Breakpoint Hit Callbacks

Users of `libsdb` should be able to register a callback on `sdb::breakpoint` objects to call whenever the breakpoint is hit. This callback will take no arguments and return a `bool` telling the process whether it should immediately resume. Modify *sdb/include/libsdb/breakpoint.hpp* like so:

```
#include <functional>
namespace sdb {
    class breakpoint {
    public:
        --snip--
        void install_hit_handler(std::function<bool(void)> on_hit) {
            on_hit_ = std::move(on_hit);
        }

        bool notify_hit() const {
            if (on_hit_) return on_hit_();
            return false;
        }

    protected:
        --snip--
        std::function<bool(void)> on_hit_;
    };
}
```

We add an `on_hit_` member function that stores the hit handler. For this member's type, we use a specialization of `std::function`, which can hold an arbitrary function-like object, such as a pointer to a function, a function object, or a closure. We also add a function called `install_hit_handler` that sets this member and a `notify_hit` function that calls the hit handler, if there is one. By default, the process shouldn't immediately resume when the breakpoint is hit, so `notify_hit` returns `false` if there is no hit handler installed.

We should call the `notify_hit` function inside `sdb::process::wait_on_signal`. Modify the existing code in *sdb/src/process.cpp* like this:

```
sdb::stop_reason sdb::process::wait_on_signal() {
    --snip--

    if (is_attached_ and state_ == process_state::stopped) {
        --snip--
        if (reason.info == SIGTRAP) {
            if (reason.trap_reason == trap_type::software_break and
                breakpoint_sites_.contains_address(instr_begin) and
```

```
                breakpoint_sites_.get_by_address(instr_begin).is_enabled()) {
                set_pc(instr_begin);

                auto& bp = breakpoint_sites_.get_by_address(instr_begin);
                if (bp.parent_) {
                    bool should_restart = bp.parent_->notify_hit();
                    if (should_restart) {
                        resume();
                        return wait_on_signal();
                    }
                }
            }
            --snip--
        }
        --snip--
    }

    return reason;
}
```

If the process stopped due to a signal, the signal was a `SIGTRAP`, the trap reason was a software breakpoint, and an enabled breakpoint site exists above the program counter, then we try to notify this hit breakpoint. We find the breakpoint site using the address at which the process stopped. If that site has a parent breakpoint, we call `notify_hit` on it. If `notify_hit` returns `true`, we resume the process and call `wait_on_signal` again to wait until it next halts.

With this support in place, we can start to locate the dynamic linker rendezvous structure.

Locating the Rendezvous Structure

Recall that if a program links to dynamic libraries, the process will begin inside of the dynamic linker, which won't initialize the rendezvous structure until we get to the executable's real entry point. As such, in `sdb::target::launch`, we'll set an internal breakpoint on the ELF file's entry point after creating the target. When the breakpoint is hit, we know the rendezvous structure exists and can locate it in memory. We'll do this with a function called `sdb::target::resolve_dynamic_linker_rendezvous`, which we'll write shortly. Replace the `return tgt;` statement at the end of `sdb::target::launch` in *sdb/src/target.cpp* with this code to set the program entry break point:

```
std::unique_ptr<sdb::target>
sdb::target::launch(
    --snip--
    auto entry_point = virt_addr{ tgt->get_process().get_auxv()[AT_ENTRY] };
    auto& entry_bp = tgt->create_address_breakpoint(entry_point, false, true);
    entry_bp.install_hit_handler([target = tgt.get()] {
```

```
        target->resolve_dynamic_linker_rendezvous();
        return true;
    });
    entry_bp.enable();
    return tgt;
}
```

We retrieve the real entry point of the executable from the auxiliary vector. We set an internal breakpoint on it and then install a hit handler that calls `resolve_dynamic_linker_rendezvous`. The process should restart immediately after resolving the address of the rendezvous structure, so the hit handler returns `true`. After installing the hit handler, we enable the breakpoint and return the created target.

Note that the lambda function's capture specifier is `[target = tgt.get()]`. The `tgt` variable is a `std::unique_ptr`, so we can't capture it by value because `std::unique_ptr`s aren't copyable. We also shouldn't capture it by reference because it will go out of scope when the function ends and we'll be left with a dangling reference. Instead, we extract the pointer it wraps and capture that pointer in the closure.

We should also call `resolve_dynamic_linker_rendezvous` when attaching to an existing process. Modify `sdb::target::attach` to call this function just before returning:

```
std::unique_ptr<sdb::target>
sdb::target::attach(pid_t pid) {
    --snip--
    tgt->resolve_dynamic_linker_rendezvous();
    return tgt;
}
```

Now we can write the `resolve_dynamic_linker_rendezvous` function. Declare it in *sdb/include/libsdb/target.hpp*, along with a data member that will store the resolved address. Also declare a `reload_dynamic_libraries` function, which we'll call to populate a list of loaded libraries, and a `read_dynamic_linker_rendezvous` function, which will read the resolved dynamic linker rendezvous:

```
#include <link.h>
namespace sdb {
    class target {
        --snip--
        std::optional<r_debug> read_dynamic_linker_rendezvous() const;
    private:
        --snip--
        void resolve_dynamic_linker_rendezvous();
        void reload_dynamic_libraries();
        --snip--
        virt_addr dynamic_linker_rendezvous_address_;
    };
}
```

Implement the `resolve_dynamic_linker_rendezvous` function in *sdb/src/target.cpp*. It should locate the dynamic section of the inferior, read it, locate the dynamic linker rendezvous structure address in the `DT_DEBUG` entry, read the list of loaded dynamic libraries, and then set an internal breakpoint on the `_dl_debug_state` function. This breakpoint should reread the dynamic library list whenever that function is called:

```
void sdb::target::resolve_dynamic_linker_rendezvous() {
    if (dynamic_linker_rendezvous_address_.addr()) return;

    auto dynamic_section = main_elf_->get_section(".dynamic");
    auto dynamic_start = file_addr{ *main_elf_, dynamic_section.value()->sh_addr };
    auto dynamic_size = dynamic_section.value()->sh_size;
    auto dynamic_bytes = process_->read_memory(
        dynamic_start.to_virt_addr(), dynamic_size);

    std::vector<Elf64_Dyn> dynamic_entries(
        dynamic_size / sizeof(Elf64_Dyn));
    std::copy(dynamic_bytes.begin(), dynamic_bytes.end(),
        reinterpret_cast<std::byte*>(dynamic_entries.data()));

    for (auto entry : dynamic_entries) {
        if (entry.d_tag == DT_DEBUG) {
            dynamic_linker_rendezvous_address_ = sdb::virt_addr{ entry.d_un.d_ptr };
            reload_dynamic_libraries();

            auto debug_info = read_dynamic_linker_rendezvous();
            auto debug_state_addr = sdb::virt_addr{ debug_info->r_brk };
            auto& debug_state_bp = create_address_breakpoint(
                debug_state_addr, false, true);
            debug_state_bp.install_hit_handler([&] {
                reload_dynamic_libraries();
                return true;
            });
            debug_state_bp.enable();
        }
    }
}
```

This snippet uses `main_elf_` instead of `elf_`, as we're going to be renaming that member shortly. We immediately return if we've already resolved the address of the rendezvous structure. We then read the `.dynamic` section of the inferior by calculating its start position and size and read the data there using `sdb::process::read_memory`. So we don't have to manually interpret the bytes of the section, we copy the data into a vector of `Elf64_Dyn` objects, which look like this:

```
typedef struct {
    Elf64_Sxword d_tag;
```

```
        union {
            Elf64_Xword d_val;
            Elf64_Addr d_ptr;
    } d_un;
} Elf64_Dyn;
```

Each object stores the tag for the entry and then its value, either as a pointer or as an integer.

To copy the dynamic section into a vector, we first initialize the vector with the correct number of empty entries, which we calculate by dividing the size of the section by the size of an entry and then copying the bytes of the dynamic section into the vector.

After performing this copy, we loop over the entries of the dynamic section. The `DT_DEBUG` entry stores the rendezvous structure's address, so when we find that tag, we store the address in the `dynamic_linker_rendezvous_address_` member and call `reload_dynamic_libaries` to initialize a list of the loaded libraries.

Finally, we set the breakpoint on `_dl_debug_state`. Recall that the `r_brk` member of the rendezvous structure holds a pointer to this function. We read the rendezvous structure from the inferior and set a breakpoint on the address stored in its `r_brk` member. We install a hit handler that reloads the set of loaded dynamic libraries and continues the process and then enable the breakpoint.

Before we read the loaded library list, we need to create a way for the debugger to store multiple ELF files.

Handling Multiple ELF Files

Currently, `sdb::target` stores a pointer to exactly one `sdb::elf` object. To support shared libraries, however, we need the ability to store multiple `sdb::elf` objects. To enable this, we'll create a relatively simple `sdb::elf_collection` type. Define it at the bottom of *sdb/include/libsdb/elf.hpp*:

```
#include <string_view>
namespace sdb {
    class elf_collection {
    public:
        void push(std::unique_ptr<elf> elf) {
            elves_.push_back(std::move(elf));
        }

        template <class F>
        void for_each(F f);
        template <class F>
        void for_each(F f) const;

        const elf* get_elf_containing_address(virt_addr address) const;
        const elf* get_elf_by_path(std::filesystem::path path) const;
        const elf* get_elf_by_filename(std::string_view name) const;
```

```
    private:
        std::vector<std::unique_ptr<elf>> elves_;
    };
}
```

The type wraps a vector of unique pointers to `sdb::elf`, with a few additional features. We can push new ELF files into the collection, iterate over them with `for_each`, and retrieve specific ELF files by their full path, filename, or an address. Implement the `for_each` functions in the same header because they're templates:

```
namespace sdb {
    template <class F>
    void elf_collection::for_each(F f) {
        for (auto& elf : elves_) {
            f(*elf);
        }
    }
    template <class F>
    void elf_collection::for_each(F f) const {
        for (const auto& elf : elves_) {
            f(*elf);
        }
    }
}
```

Implement the other three functions in *sdb/src/elf.cpp*:

```
const sdb::elf* sdb::elf_collection::get_elf_containing_address(
      virt_addr address) const {
    for (auto& elf : elves_) {
        if (auto section = elf->get_section_containing_address(address); section) {
            return elf.get();
        }
    }
    return nullptr;
}

const sdb::elf* sdb::elf_collection::get_elf_by_path(
      std::filesystem::path path) const {
    for (auto& elf : elves_) {
        if (elf->path() == path) {
            return elf.get();
        }
    }
    return nullptr;
}
```

```
const sdb::elf* sdb::elf_collection::get_elf_by_filename(
      std::string_view name) const {
    for (auto& elf : elves_) {
        if (elf->path().filename() == name) {
            return elf.get();
        }
    }
    return nullptr;
}
```

These functions simply iterate over the ELF files and return the one that corresponds to the given data, or a null pointer if none exists.

To hook this code into `sdb::target`, replace the existing `elf_` member with two members: one to store the `sdb::elf_collection`, and one to point to the main ELF file for the executable, for convenience. Also replace the `get_elf` overloads with `get_elves` and `get_main_elf`. Finally, populate `elves_` and `main_elf_` in the constructor:

```
namespace sdb {
    class target {
    public:
        --snip--
        elf_collection& get_elves() { return elves_; }
        const elf_collection& get_elves() const { return elves_; }
        elf& get_main_elf() { return *main_elf_; }
        const elf& get_main_elf() const { return *main_elf_; }

    private:
        target(std::unique_ptr<process> proc, std::unique_ptr<elf> obj)
            : process_(std::move(proc))
            , stack_(this)
            , main_elf_(obj.get()) {
            elves_.push(std::move(obj));
        }
        --snip--
        elf_collection elves_;
        elf* main_elf_;
    };
}
```

This change has a few knock-on effects on the rest of the codebase because the assumption that only one ELF file exists has permeated through the implementation. To minimize the number of changes we need to make to existing functions, let's add an overload of `sdb::virt_addr::to_file_address` that takes an `sdb::elf_collection` and finds the ELF file that matches the stored virtual address before translating it to a file address. Declare it in *sdb/include/libsdb/types.hpp*:

```
namespace sdb {
    class elf_collection;
```

```
class virt_addr {
public:
    --snip--
    file_addr to_file_addr(const elf_collection& elves) const;
    --snip--
};
```

We forward-declare `sdb::elf_collection` rather than including the header to avoid cyclical dependencies. Implement this function in *sdb/src/types.cpp*:

```
sdb::file_addr sdb::virt_addr::to_file_addr(const elf_collection& elves) const {
    auto obj = elves.get_elf_containing_address(*this);
    if (!obj) return file_addr{};
    return to_file_addr(*obj);
}
```

The function calls `sdb::elf_collection::get_elf_containing_address` to locate the correct ELF file for which to perform the translation and then defers to the existing overload of `to_file_addr`, so long as it found an ELF file.

At several points in the codebase, we assume we need to check only a single `sdb::dwarf` object. One is in `sdb::line_breakpoint::resolve`, which looks only in the main ELF file's DWARF information for line entries. Let's add a function to `sdb::target` that looks for line entries in any ELF file. Declare it in *sdb/include/libsdb/target.hpp*:

```
namespace sdb {
    class target {
        --snip--
        std::vector<line_table::iterator> get_line_entries_by_line(
            std::filesystem::path path, std::size_t line) const;
        --snip--
    };
}
```

This function should call `sdb::dwarf::line_table::get_entry_by_address` for every ELF file. Implement it in *sdb/src/target.cpp*:

```
std::vector<sdb::line_table::iterator> sdb::target::get_line_entries_by_line(
      std::filesystem::path path, std::size_t line) const {
    std::vector<sdb::line_table::iterator> entries;
    elves_.for_each([&](auto& elf) {
        for (auto& cu : elf.get_dwarf().compile_units()) {
            auto new_entries = cu->lines().get_entries_by_line(path, line);
            entries.insert(entries.end(), new_entries.begin(), new_entries.end());
        }
    });
    return entries;
}
```

We create an empty vector to hold the entries we find. We then loop over every ELF file in `elves_`, call `get_entries_by_line` on each compile unit in that ELF file's DWARF information, and collect the results into that single vector. Finally, we return the entries we found.

In `sdb::line_breakpoint::resolve` in *sdb/src/breakpoint.cpp*, we get rid of the outer loop and call `sdb::target::get_line_entries_by_line` instead of calling `sdb::dwarf::line_table::get_entry_by_address`. The rest of the code can stay the same:

```
void sdb::line_breakpoint::resolve() {
    auto entries = target_->get_line_entries_by_line(file_, line_);
    for (auto entry : entries) {
        --snip--
    }
}
```

We must also update `sdb::stack::unwind`. We used to continue unwinding so long as the program counter was within the main ELF file; now, we should keep unwinding while we have any ELF file that corresponds to the program counter. Also, at the end of the loop, we should call `to_file_addr` with `sdb::elf_collection` for the target rather than just for the main ELF file. Make these changes in *sdb/src/stack.cpp*:

```
void sdb::stack::unwind() {
    --snip--
    while (virt_pc.addr() != 0 and elf) {
        --snip--
        file_pc = virt_pc.to_file_addr(target_->get_elves());
        --snip--
    }
}
```

We have a few more updates to make. For reasons similar to the previous change, the `sdb::target::get_pc_file_address` and `function_name_at_address` functions in *sdb/src/target.cpp* should pass the `sdb::elf_collection` to `to_file_addr`:

```
sdb::file_addr sdb::target::get_pc_file_address() const {
    return process_->get_pc().to_file_addr(elves_);
}

std::string_view sdb::target::function_name_at_address(virt_addr address) const {
    auto file_address = address.to_file_addr(elves_);
    auto obj = file_address.elf_file();
    if (!obj) return "";
    --snip--
}
```

We also add a check to `function_name_at_address` to return an empty string if it can't find the ELF file for the given address.

The `std::target::find_functions` function in the same file needs to loop over all ELF files in the target, which we can do by wrapping the code, save for the first and last statements, in a lambda, making a couple of small changes, and passing it to `sdb::elf_collection::for_each`:

```
sdb::target::find_functions_result
sdb::target::find_functions(std::string name) const {
    find_functions_result result;

    elves_.for_each([&](auto& elf) {
        auto dwarf_found = ❶ elf.get_dwarf().find_functions(name);
        if (dwarf_found.empty()) {
            auto elf_found = ❷ elf.get_symbols_by_name(name);
            for (auto sym : elf_found) {
                result.elf_functions.push_back(std::pair{ ❸ &elf, sym });
            }
        }
        else {
            --snip--
        }
    });
    return result;
}
```

There are three changes needed in the lambda body to go from `elf_` to `elf` ❶ ❷ ❸.

We've made all of the changes necessary for supporting multiple ELF files. Now we can move on to reading the loaded library list.

Reading the Loaded Library List

When we implemented `resolve_dynamic_linker_rendezvous`, we left two functions unimplemented: `read_dynamic_linker_rendezvous` and `reload_dynamic_libraries`. The former should read the rendezvous structure from the inferior, if its address has been resolved, and return it as an `r_debug` object. The latter function should walk the loaded library list inside the `r_debug` object and populate `sdb::target::elves_` with any new ELF files.

Let's start by implementing `read_dynamic_linker_rendezvous` in *sdb/src/target.cpp*:

```
std::optional<r_debug>
sdb::target::read_dynamic_linker_rendezvous() const {
    if (dynamic_linker_rendezvous_address_.addr()) {
        return process_->read_memory_as<r_debug>(
            dynamic_linker_rendezvous_address_);
    }
    return std::nullopt;
}
```

If the rendezvous structure address has been resolved, we read the memory at that location as an `r_debug` object and return the result. Otherwise, we return an empty optional.

The `reload_dynamic_libraries` function is a fair bit more complex, so we'll tackle it piece by piece. For the sake of simplicity, I won't handle unloading shared libraries, which would require some more work to disable any breakpoints set on locations within the unloaded libraries. We'll begin by reading the rendezvous structure, returning if the reading fails, and grabbing the pointer to the first entry in the loaded library list:

```
void sdb::target::reload_dynamic_libraries() {
    auto debug = read_dynamic_linker_rendezvous();
    if (!debug) return;

    auto entry_ptr = debug->r_map;
```

We'll loop until we hit the end of the list, indicated by a null pointer. We can't just walk the list as though it were a regular C++ linked list because the pointers all point inside of the inferior process rather than inside of the debugger. So, we need to use the memory-reading functions in `sdb::process`. For each entry in the map, we read the `link_map` structure and point `entry_ptr` to the next entry:

```
    while (entry_ptr != nullptr) {
        auto entry_addr = virt_addr(
            reinterpret_cast<std::uint64_t>(entry_ptr));
        auto entry = process_->read_memory_as<link_map>(entry_addr);
        entry_ptr = entry.l_next;
```

Next, we read the path to the ELF file, encoded as a null-terminated string in the `l_name` field of the map entry. The maximum path size on Linux is 4,096 bytes, so we read that amount (recall that `sdb::process::read_memory` supports partial reads in case reading 4KiB from that address hits an inaccessible memory page). By convention, an empty string identifies the executable itself, so if the path we read is empty, we should continue to the next entry:

```
        auto name_addr = virt_addr(
            reinterpret_cast<std::uint64_t>(entry.l_name));
        auto name_bytes = process_->read_memory(name_addr, 4096);
        auto name = std::filesystem::path{
            reinterpret_cast<char*>(name_bytes.data()) };
        if (name.empty()) continue;
```

We should check whether we've already created an `sdb::elf` object for this shared library, which we can do by looking up the path in `sdb::target ::elves_`. It should be an absolute path to the ELF file for that shared library, except in the case of the vDSO. Recall that the vDSO doesn't reside on disk and is identified by the name *linux-vdso.so.1*. We'll dump the vDSO to disk

and store an absolute path to it in the corresponding `sdb::elf` object, so if the library name in the list entry identifies the vDSO, we should perform a lookup by filename rather than by path in the `sdb::elf_collection`:

```
        const elf* found = nullptr;
        const auto vdso_name = "linux-vdso.so.1";
        if (name == vdso_name) {
            found = elves_.get_elf_by_filename(name.c_str());
        }
        else {
            found = elves_.get_elf_by_path(name);
        }
```

If we didn't find a corresponding `sdb::elf` object, we should create one, notify it of where it was loaded in memory, and add it to `sdb::target::elves_`. Also, if this entry is for the vDSO, we should first dump the vDSO to disk, which we'll do with an aptly named but currently nonexistent, function called `dump_vdso`:

```
        if (!found) {
            if (name == vdso_name) {
                name = dump_vdso(*process_, virt_addr{ entry.l_addr });
            }
            auto new_elf = std::make_unique<elf>(name);
            new_elf->notify_loaded(virt_addr{ entry.l_addr });
            elves_.push(std::move(new_elf));
        }
    }
```

Finally, we should resolve all breakpoints in the target, as the user may have set breakpoints that require the DWARF information of one of the shared libraries to resolve:

```
    breakpoints_.for_each([&](auto& bp) {
        bp.resolve();
    });
}
```

The only thing left to do is implement `dump_vdso`, which we'll define in the same file:

```
#include <fstream>
namespace {
    std::filesystem::path dump_vdso(
          const sdb::process& proc, sdb::virt_addr address) {
        char tmp_dir[] = "/tmp/sdb-XXXXXX";
        mkdtemp(tmp_dir);
        auto vdso_dump_path = std::filesystem::path(tmp_dir) / "linux-vdso.so.1";
        std::ofstream vdso_dump(vdso_dump_path, std::ios::binary);
```

```
        auto vdso_header = proc.read_memory_as<Elf64_Ehdr>(address);
        auto vdso_size = vdso_header.e_shoff +
            vdso_header.e_shentsize * vdso_header.e_shnum;
        auto vdso_bytes = proc.read_memory(address, vdso_size);
        vdso_dump.write(
            reinterpret_cast<const char*>(vdso_bytes.data()), vdso_bytes.size());
        return vdso_dump_path;
    }
}
```

We must dump the vDSO into a file called exactly *linux-vdso.so.1* so that its filename matches the entry in the rendezvous structure. To do this, we create a new temporary directory using `mkdtemp`. This command takes a string whose last six characters are `XXXXXX`, fills in those placeholders with random characters to make a unique directory name, and then creates a directory at that path. We declare the variable as `char tmp_dir[]` rather than as `const char*` because we'll update the string with the correct path. We then construct a full path to the dump location by appending `linux-vdso.so.1` and construct a `std::ostream` with which we can write to the file.

Next, we grab the ELF header for the vDSO, found at the address given to `dump_vdso` as an argument, and calculate the size of the ELF file. Calculating the size of any arbitrary ELF file is tricky because it may not necessarily have section headers that give us an easy way to locate the end. Luckily, the vDSO does have section headers, so we can calculate its size by offsetting the start of the section headers by the number of section entries multiplied by the size of an entry. We then read the calculated number of bytes from the start of the vDSO, write them to the temporary file, and return its path.

Before we test this code, let's make *sdb* include information about the ELF file to which a function belongs when it prints out function names. Modify `sdb::target::function_name_at_address` in *sdb/src/target.cpp* like so:

```
std::string sdb::target::function_name_at_address(virt_addr address) const {
    auto file_address = address.to_file_addr(elves_);
    auto obj = file_address.elf_file();
    auto func = obj->get_dwarf().function_containing_address(file_address);
❶ auto elf_filename = obj->path().filename().string();
❷ std::string func_name = "";

    if (func and func->name()) {
    ❸ func_name = *func->name();
    }
    else if (auto elf_func = obj->get_symbol_containing_address(file_address);
        elf_func and ELF64_ST_TYPE(elf_func.value()->st_info) == STT_FUNC) {
    ❹ func_name = obj->get_string(elf_func.value()->st_name);
    }

❺ if (!func_name.empty()) {
        return elf_filename + "`" + func_name;
```

```
    }
    return "";
}
```

We add a variable to hold the ELF filename ❶ and a variable to hold the computed function name ❷. Instead of immediately returning inside the branches, we set `func_name` to the found name ❸ ❹. After trying to compute the function name if we found one, we prefix it with the ELF filename and a backtick ❺, so the result will look like `libsdb.so`function_name_at_address`. Let's test this feature on the *marshmallow* executable:

```
$ tools/sdb test/targets/marshmallow
Launched process with PID 68736
sdb> break set libmeow_client_is_cute
sdb> c
Cuteness rating: 100
Process 68736 stopped with signal TRAP at 0x7ffff7fb7101,
libmeow.cpp:4 (libmeow.so`libmeow_client_is_cute) (breakpoint 1)
  1 extern int libmeow_client_cuteness;
  2
  3 bool libmeow_client_is_cute() {
> 4     return libmeow_client_cuteness > 50;
  5 }
sdb> back
*[0]: 0x7ffff7fb7101 libmeow.so`libmeow_client_is_cute
 [1]: 0x5555555552a0 marshmallow`main
```

As expected, when the target program loads the *libmeow.so* library, the debugger resolves the `libmeow_client_is_cute` breakpoint. Note that the function name is qualified with the ELF file in which it's defined and that stack unwinding works across the libraries. Let's end this chapter by writing an automated test.

Testing

Add a new test case to *sdb/test/tests.cpp* that does the same thing as our manual test. It should launch *marshmallow*, set a breakpoint on `libmeow_client_is_cute`, and ensure that the breakpoint is hit and that the backtrace is correct:

```
TEST_CASE("Shared library tracing works", "[dynlib]") {
    auto dev_null = open("/dev/null", O_WRONLY);
    auto target = target::launch("targets/marshmallow", dev_null);
    auto& proc = target->get_process();

    target->create_function_breakpoint("libmeow_client_is_cute").enable();
    proc.resume();
    proc.wait_on_signal();

    REQUIRE(target->get_stack().frames().size() == 2);
```

```
    REQUIRE(
        target->get_stack().frames()[0].func_die.name().value() == "libmeow_client_is_cute");
    REQUIRE(target->get_stack().frames()[1].func_die.name().value() == "main");
    REQUIRE(target->get_pc_file_address().elf_file()->path().filename() == "libmeow.so");
    close(dev_null);
}
```

Because we changed the `function_name_at_address` output format, we need to fix the source-level stepping test. Prefix all the function names with `step`\`, like this:

```
TEST_CASE("Source-level stepping", "[target]") {
    --snip--
    REQUIRE(target->function_name_at_address(pc) == "step`main");
    --snip--
    REQUIRE(target->function_name_at_address(pc) == "step`main");
    --snip--
    REQUIRE(target->function_name_at_address(pc) == "step`find_happiness");
    --snip--
    REQUIRE(target->function_name_at_address(pc) == "step`find_happiness");
    --snip--
    REQUIRE(target->function_name_at_address(pc) == "step`main");
}
```

We're done! Run the tests, which should all pass.

Summary

In this chapter, you learned about how the Linux kernel loads programs when the dynamic linker is involved and when it isn't. You learned how the `.dynamic` section communicates information between the kernel and dynamic linker and how the dynamic linker rendezvous structure provides the debugger with information about the dynamic libraries loaded in the process. You also learned how relocations fix references between shared libraries, with help from the global offset table for variables and the procedure linkage table for function calls.

You implemented tracing for dynamic library loading by relying on the dynamic linker rendezvous structure and added hit handlers to `sdb::breakpoint`. You also augmented the textual output of the debugger with shared library information and tested your changes.

In the next chapter, you'll learn about multithreaded programs on Linux and add support for threads to the debugger.

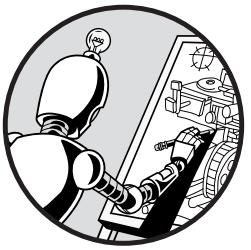
Check Your Knowledge

1. What do program headers describe?
2. Which segment type provides the path to the dynamic linker?
3. What is the purpose of the vDSO?
4. Where does the kernel begin execution of programs whose executables are PIEs or that depend on dynamic libraries?
5. What type of `.dynamic` section entry specifies a dynamic library dependency?
6. What is the purpose of the dynamic linker rendezvous structure?
7. How can the dynamic linker rendezvous structure be located?
8. To which function does the dynamic linker rendezvous structure maintain a pointer for debuggers to set a breakpoint on?
9. What is the purpose of relocations?
10. What is the purpose of the global offset table?
11. Why is the global offset table not used directly for function calls?
12. What is lazy binding?

18

MULTITHREADING

A knot of wool,
needle lost an aeon before,
the desire to be unwound.



Programs sometimes need to take multiple actions at the same time. For example, a GUI may need to perform a lot of heavy processing whenever someone clicks a button, and the user will have a terrible experience if the entire interface stops working while that processing occurs. Instead, the program can introduce multiple *threads* of execution to perform several tasks at once. In this chapter, you'll learn about how Linux systems handle threads and extend your debugger with the ability to trace multiple threads in a process.

Threads on Linux

On Linux, processes and threads are closely related. The kernel implements both as *tasks*, which represent a single unit of execution for the scheduler.

The key difference between a thread and a process is that multiple threads can share a single virtual address space, set of file descriptors, and set of signal handlers, whereas processes cannot.

Because threads and processes use the same machinery in the kernel, threads have their own PIDs that differ from the PID of the process that spawned them. We often refer to these as *thread IDs (TIDs)*. Threads that belong to the same process have the same *thread group ID (TGID)*, which is equal to the PID of the original process.

Different threads may have different states of execution: while one thread is running, another thread may exit, and yet another might stop due to a signal. Debuggers generally operate in one of two modes: all-stop mode, where the debugger must stop all threads before inspecting the program state, and non-stop mode, where it can stop and inspect individual threads while others continue running. LLDB always runs in all-stop mode. GDB runs in all-stop mode by default, but users can enable non-stop mode. We'll implement all-stop mode because it's generally easier to inspect program behavior while other threads aren't running and modifying the program state.

The pthreads Library

Linux creates threads with the `clone` syscall, but because this is a very low-level utility, application programmers generally use the `pthreads` library instead. As an example, create a new file at *sdb/test/targets/multi_threaded.cpp* with the following program, which spawns multiple threads with `pthreads`:

```
#include <pthread.h>
#include <vector>
#include <iostream>
#include <unistd.h>

void* say_hi(void*) {
    std::cout << "Thread " << gettid() << " reporting in\n";
    return nullptr;
}

int main() {
    std::vector<pthread_t> threads(10);

    for (auto& thread : threads) {
        pthread_create(&thread, nullptr, say_hi, nullptr);
    }

    for (auto& thread : threads) {
        pthread_join(thread, nullptr);
    }
}
```

The `say_hi` function prints the TID of the thread in which it's running. The `main` function creates a vector of 10 thread handles. It then loops through them, calling `pthread_create` on each one in turn. The first argument is a pointer to the thread handle, the second contains optional attributes for the new thread (such as the amount of memory to allocate for its stack), the third is a pointer to an initial function that the thread will call when it starts, and the fourth is an argument to pass to the initial function. As soon as the program calls `pthread_create`, it creates a new thread and executes the `say_hi` function. After creating all the threads, it waits until they all complete with the `pthread_join` function. This function takes the thread to wait on and an optional out parameter at which to store the return value of the thread.

Add this code to *sdb/test/targets/CMakeLists.txt*:

```
--snip--
add_test_cpp_target(multi_threaded)
target_link_libraries(multi_threaded pthread)
```

Build and run the program, and you should see something like (but likely not the same as) this:

```
$ test/targets/multi_threaded
Thread Thread 11774 reporting in
Thread 11776 reporting in
11775 reporting in
Thread 11779 reporting in
Thread 11780 reporting in
Thread 11777 reporting in
Thread 11778 reporting in
Thread 11781 reporting in
Thread 11782 reporting in
Thread 11783 reporting in
```

Notice a problem: the text is out of order and, in some places, interrupted. This is due to race conditions, which we'll look at next.

Race Conditions

A *race condition* occurs when the behavior of the program relies on the timing of two or more threads carrying out an action. The threads involved are said to *race*, and the output of such a program is *nondeterministic*; we can't predict exactly what its behavior will be because we don't know which thread will finish first. In the *multi_threaded* program, the order of the lines of output depends on the order of the threads' execution. Also, the output of some threads may interrupt the output of other threads because threads (and processes) don't necessarily execute in their entirety before exiting; they may be *preempted* by the operating system, which suspends their execution and resumes some other thread or process.

Programmers usually avoid race conditions by introducing explicit synchronization into the program to make the order of operations deterministic or to guard threads from simultaneously performing actions that should be mutually exclusive.

The debugger we've been writing over the course of this book is single-threaded, and it will remain so, but we still need to be wary of race conditions, because the order in which the debugger intercepts the signals sent to the inferior process may be nondeterministic. We'll look at some examples of this behavior as we implement support for multithreading in the debugger.

ptrace and the procfs

To provide the user with an accurate picture of the threads running in a given process, we need to retrieve the current list of threads when we attach to a process and then receive notifications about the creation of new threads. We can do the former thanks to the Linux procfs and the latter thanks to `ptrace`.

The */proc/<pid>/task* directory has a subdirectory for every thread whose name is the TID of the thread. For example, if a process with the PID `42` has spawned two threads, the *task* directory might look like this:

```
$ ls /proc/42/task
42 43 44
```

The thread with the TID `42` is the main thread for the process, and the other two are the threads that the process has spawned. As such, when we attach to a process, we can read the list of subdirectories in the *task* directory to find the current set of threads.

The `ptrace` syscall has an option called `PTRACE_O_TRACECLONE` that we can pass to a `PTRACE_SETOPTIONS` request. When the option is enabled, the kernel will send a `SIGTRAP` to any thread that spans a new thread and send a `SIGSTOP` to the new thread. We can figure out that a `SIGTRAP` signifies the creation of a thread by checking the status returned by `waitpid`. If `status>>8 == (SIGTRAP | (PTRACE_EVENT_EXEC<<8))` is `true`, the `SIGTRAP` was due to a new thread being created.

We can also use `ptrace` to inspect and manipulate threads other than the main one by passing their TIDs instead of the PID of the process. For example, calling `ptrace` with the TID of a new thread and the `PTRACE_GETREGS` request would retrieve the registers for the new thread. Requests like `PTRACE_CONT` and `PTRACE_SINGLESTEP` operate on a single thread, so sending a `PTRACE_CONT` to the PID of the running process will continue only the main thread, not any additional threads.

Tracing Threads

With the background out the way, we can implement thread tracing in the debugger.

Trapping New Threads

We'll start by enabling the PTRACE_O_TRACECLONE request when the debugger attaches to the process. We can do this in the existing set_ptrace_options function in *sdb/src/process.cpp*:

```
namespace {
    void set_ptrace_options(pid_t pid) {
        if (ptrace(PTRACE_SETOPTIONS, pid, nullptr,
                   PTRACE_O_TRACESYSGOOD | PTRACE_O_TRACECLONE) < 0) {
            sdb::error::send_errno(
                "Failed to set TRACESYSGOOD and TRACECLONE options");
        }
    }
}
```

Options for ptrace are bit flags, so we bitwise OR the PTRACE_O_TRACECLONE flag with the existing PTRACE_O_TRACESYSGOOD flag to get the final value to set.

Identifying Thread Creation

We need to identify stops that come from thread creation events, so let's extend sdb::stop_reason to do that. We'll add a clone option to the sdb::trap_type enum and then add a member sdb::stop_reason to track the TID of the thread that reported the stop. Make these modifications in *sdb/include/libsdb/process.hpp*:

```
namespace sdb {
    enum class trap_type {
        single_step, software_break, hardware_break,
        syscall, clone, unknown
    };

    struct stop_reason {
        --snip--
        stop_reason(pid_t tid, int wait_status);

        stop_reason(pid_t tid, process_state reason, std::uint8_t info,
            std::optional<trap_type> trap_reason = std::nullopt,
            std::optional<syscall_information> syscall_info = std::nullopt)
            : reason(reason)
            , info(info)
            , trap_reason(trap_reason)
            , syscall_info(syscall_info)
            , tid(tid)
        {}
        --snip--

        pid_t tid;
```

```
    };
}
```

We add the `clone` enumerator to `trap_type`, a `tid` member to `stop_reason`, and parameters for the TID to both of the constructors, ensuring we initialize the `tid` member in the second constructor.

Modify the implementation of the first constructor in *sdb/src/process.cpp* to store the TID and set the trap reason if this stop occurs due to a clone event:

```
sdb::stop_reason::stop_reason(pid_t tid, int wait_status)
    : tid(tid) {
    if ((wait_status >> 8) == (SIGTRAP | (PTRACE_EVENT_CLONE << 8))) {
        trap_reason = trap_type::clone;
    }
    --snip--
}
```

We initialize the `tid` member with the given argument. At the start of the constructor body, we check if the stop was due to a clone and, if so, store `trap_type::clone` as the trap reason.

Representing Thread States

We need a data structure to represent the state of a thread. It should hold the thread's TID, its registers, its most recent stop reason, and its current state. Let's add it to *sdb/include/libsdb/process.hpp*, then add members to `sdb::process` that track the current set of threads and identify the thread the debugger is currently interacting with. Because the `thread_state` type is responsible for tracking the registers for a given thread, we'll also remove the existing `registers_` member from `sdb::process`:

```
namespace sdb {
    struct thread_state {
        pid_t tid;
        registers regs;
        stop_reason reason;
        process_state state = process_state::stopped;
    };

    class process {
        --snip--
        void set_current_thread(pid_t tid) { current_thread_ = tid; }
        pid_t current_thread() const { return current_thread_; }

        std::unordered_map<pid_t, thread_state>&
        thread_states() { return threads_; }

        const std::unordered_map<pid_t, thread_state>&
        thread_states() const { return threads_; }
```

```
    private:
        --snip--
        std::unordered_map<pid_t, thread_state> threads_;
        pid_t current_thread_ = 0;
    };
}
```

With the data types in place, we need to populate them. Add a function named `populate_existing_threads` to `sdb::process` and call it from the constructor. Also initialize `current_thread` to be the given PID and remove the initialization of the old `registers_` member:

```
namespace sdb {
    class process {
        --snip--
    private:
        --snip--
        process(pid_t pid, bool terminate_on_end, bool is_attached)
            : pid_(pid)
            , terminate_on_end_(terminate_on_end)
            , is_attached_(is_attached)
            , current_thread_(pid) {
            populate_existing_threads();
        }

        void populate_existing_threads();
    };
}
```

Implement `populate_existing_threads` in *sdb/src/process.cpp*. This function should read the contents of */proc/<pid>/task* and create `sdb::thread_state` objects for each of the listed TIDs:

```
void sdb::process::populate_existing_threads() {
    auto path = "/proc/" + std::to_string(pid_) + "/task";
    for (auto& entry : std::filesystem::directory_iterator(path)) {
        auto tid = std::stoi(entry.path().filename().string());
        threads_.emplace(tid, thread_state{ tid, registers(*this, tid) });
    }
}
```

We compute the path to the correct *task* directory, iterate over all of the subdirectories, convert their names to integers that represent the TIDs, and create new entries in the `threads_` map for each one. The `sdb::registers` type doesn't currently store the TID to which the registers belong, so let's add a member for this information and initialize it in *sdb/include/libsdb/registers.hpp*:

```
namespace sdb {
    class registers {
```

```
        --snip--
    private:
        --snip--
        registers(process& proc, pid_t tid) : proc_(&proc), tid_(tid) {}
        --snip--
        pid_t tid_;
    };
}
```

Threads have operating system–level state information, including their registers and TIDs, but they also have symbol-level state information, such as their call stacks, which should live in `sdb::target` rather than in `sdb::process`. Let's add a type called `sdb::thread` and a member to hold a collection of them to `sdb::target`. We should also add a function that `sdb::process` can call to notify the target that a new thread has been created or terminated. In a similar way to how we removed `registers_` from `sdb::process`, we'll also remove the `stack_` member from `sdb::target`. Make these modifications in *sdb/include/libsdb/target.hpp*:

```
namespace sdb {
    struct thread {
        thread(thread_state* state, stack frames)
            : state(state), frames(std::move(frames)) {}
        thread_state* state;
        stack frames;
    };

    class target {
    public:
        --snip--
        std::unordered_map<pid_t, thread>& threads() {
            return threads_;
        }
        const std::unordered_map<pid_t, thread>& threads() const {
            return threads_;
        }

        void notify_thread_lifecycle_event(const sdb::stop_reason& reason);

    private:
        --snip--
        std::unordered_map<pid_t, thread> threads_;
    };
}
```

The `thread` type points to a `thread_state` object that lives in `sdb::process` and stores the stack frames for that thread. We replace the `stack_` member with a `threads_` member that stores the stack frames and `threads` functions

to retrieve this map. Finally, we add a `notify_thread_lifecycle_event` function that the process can call whenever a new thread appears or one exits.

Initialize the threads for the target in the constructor:

```
target(std::unique_ptr<process> proc, std::unique_ptr<elf> obj)
    : process_(std::move(proc))
    , main_elf_(obj.get()) {
    elves_.push(std::move(obj));
    auto pid = process_->pid();
    for (auto& [tid, state] : process_->thread_states()) {
        threads_.emplace(tid, thread(&state, stack{this, tid}));
    }
}
```

We remove the initialization of the old `stack_` member. Inside the constructor body, we loop over all the thread states held in `process_` and create corresponding `thread` objects. The `sdb::stack` constructor doesn't currently take a TID, but we'll change that next.

Add a member to `sdb::stack` to hold the TID to which the stack belongs:

```
namespace sdb {
    class stack {
    public:
        stack(target* tgt, pid_t tid) : target_(tgt), tid_(tid) {}
        pid_t tid() const { return tid_; }

    private:
        --snip--
        pid_t tid_ = 0;
    };
}
```

We also add a parameter to its constructor that initializes the member and a member function to retrieve it.

Supporting Multithreaded Processes

Unfortunately, we now have a major refactor ahead of us. Many of the functions in `sdb::process` and `sdb::target`, like `resume` and `step_in`, assume that a process has only one thread. We have a few options for dealing with this problem:

1. Move all of the functions from `sdb::process` to `sdb::thread_state` and from `sdb::target` to `sdb::thread`.
2. Add a TID parameter to all of these functions that indicates the thread on which to operate.
3. Add an optional TID parameter and, if the caller doesn't pass a TID, default to the active thread's TID.

Option 1 is the best decision from an API design standpoint, but it requires a lot of boring, manual work, and this isn't a book about API design. Option 2 will break much of the code in the command line driver and our tests and make using `libsdb` quite frustrating. Option 3 is a decent midpoint; our existing client code will continue to work, the API should remain user-friendly, and we won't have to make too many modifications to the library code.

We'll start with `sdb::process`. Add optional TID arguments to the `resume`, `get_registers`, `write_user_area`, `write_fprs`, `write_gprs`, `get_pc`, `set_pc`, `get_registers`, `step_instruction`, and `get_hardware_stoppoint` functions in *sdb/include/libsdb/process.hpp*:

```
namespace sdb {
    class process {
        void resume(std::optional<pid_t> otid = std::nullopt);

        registers& get_registers(
              std::optional<pid_t> otid = std::nullopt);
        const registers& get_registers(
              std::optional<pid_t> otid = std::nullopt) const;

        void write_user_area(
            std::size_t offset, std::uint64_t data,
            std::optional<pid_t> otid = std::nullopt);
        void write_fprs(
            const user_fpregs_struct& fprs,
            std::optional<pid_t> otid = std::nullopt);
        void write_gprs(
            const user_regs_struct& fprs,
            std::optional<pid_t> otid = std::nullopt);

        virt_addr get_pc(
              std::optional<pid_t> otid = std::nullopt) const;

        void set_pc(
              virt_addr address,
              std::optional<pid_t> otid = std::nullopt);

        sdb::stop_reason step_instruction(
            std::optional<pid_t> otid = std::nullopt);

        std::variant<breakpoint_site::id_type, watchpoint::id_type>
        get_current_hardware_stoppoint(
            std::optional<pid_t> otid = std::nullopt) const;

        --snip--
```

```
    private:
        void read_all_registers(pid_t tid);
        bool should_resume_from_syscall(const stop_reason& reason);
        --snip--
    };
}
```

We also change the `maybe_resume_from_syscall` function into a function that returns whether the thread should resume, called `should_resume_from_syscall`. For the functions currently declared inline, we'll move their definitions into the implementation file, as the header is getting rather cluttered. So, we turn them into declarations. We also add a nonoptional TID parameter to the private `read_all_registers` function.

Unfortunately, C++ doesn't allow us to set a default argument as the value of a data member, so we use a little trick: we take a `std::optional<pid_t>` as an argument and set `std::nullopt` as the default argument. Then, we'll call `otid.value_or(current_thread_)` to get either the argument that the caller passed or the current thread, in case the caller passed no argument.

One last change we need to make to the header is in `wait_on_signal`. This function should optionally take a TID to wait on, but by default, it will wait on all threads. The `waitpid` function uses `-1` as an indicator for "wait on all threads," so we'll use same approach here:

```
namespace sdb {
    class process {
    public:
        --snip--
        void wait_on_signal(pid_t to_await = -1);
        --snip--
    };
}
```

Let's make the necessary changes to *sdb/src/process.cpp*, starting with the definitions of the functions that used to be defined inline in the header:

```
sdb::virt_addr sdb::process::get_pc(std::optional<pid_t> otid) const {
    return virt_addr{
        get_registers(otid).read_by_id_as<std::uint64_t>(register_id::rip)
    };
}

void sdb::process::set_pc(
      virt_addr address, std::optional<pid_t> otid) {
    get_registers(otid).write_by_id(register_id::rip, address.addr());
}

sdb::registers& sdb::process::get_registers(
      std::optional<pid_t> otid) {
```

```
    auto tid = otid.value_or(current_thread_);
    return threads_.at(tid).regs;
}

const sdb::registers& sdb::process::get_registers(
      std::optional<pid_t> otid) const {
    return const_cast<process*>(this)->get_registers(otid);
}
```

In `get_pc`, we pass the `otid` argument through to `get_registers`. The rest of the function remains unchanged. In the `set_pc` function, we make the same modification, passing the argument through to `get_registers`. The non-`const` overload for `get_registers` first computes the TID of the thread that it should return the registers for and then retrieves that register set from the `threads_` member. The `const` overload uses the `const_cast` trick we've used several times before to defer the implementation to the non-`const` overload.

Now we need to fix the rest of the functions whose declarations we changed. The `write_user_area`, `write_fprs`, and `write_gprs` functions need to compute the correct TID and pass that to `ptrace`:

```
void sdb::process::write_user_area(
      std::size_t offset, std::uint64_t data, std::optional<pid_t> otid) {
    auto tid = otid.value_or(current_thread_);
    if (ptrace(PTRACE_POKEUSER, tid, offset, data) < 0) {
        error::send_errno("Could not write to user area");
    }
}

void sdb::process::write_fprs(
      const user_fpregs_struct& fprs, std::optional<pid_t> otid) {
    auto tid = otid.value_or(current_thread_);
    if (ptrace(PTRACE_SETFPREGS, tid, nullptr, &fprs) < 0) {
        error::send_errno("Could not write floating point registers");
    }
}

void sdb::process::write_gprs(const user_regs_struct& gprs, std::optional<pid_t> otid) {
    auto tid = otid.value_or(current_thread_);
    if (ptrace(PTRACE_SETREGS, tid, nullptr, &gprs) < 0) {
        error::send_errno("Could not write general purpose registers");
    }
}
```

The private `read_all_registers` function needs to pass the TID to each of its three `ptrace` calls and each of its three calls to `get_registers`:

```
void sdb::process::read_all_registers(pid_t tid) {
    if (ptrace(PTRACE_GETREGS, tid, nullptr, &get_registers(tid).data_.regs) < 0) {
        error::send_errno("Could not read GPR registers");
```

```
    }
    if (ptrace(PTRACE_GETFPREGS, tid, nullptr, &get_registers(tid).data_.i387) < 0) {
        error::send_errno("Could not read FPR registers");
    }
    for (int i = 0; i < 8; ++i) {
        --snip--
        std::int64_t data = ptrace(PTRACE_PEEKUSER, tid, info.offset, nullptr);
        --snip--
        get_registers(tid).data_.u_debugreg[i] = data;
    }
}
```

The `resume` function has to compute the TID and pass it to the calls to `get_pc`, `waitpid`, and `ptrace`. Additionally, at the end of the function, it should set the state of the thread on which it operated and of the entire process:

```
void sdb::process::resume(std::optional<pid_t> otid) {
    auto tid = otid.value_or(current_thread_);
    auto pc = get_pc(tid);
    if (--snip--) {
        --snip--
        if (ptrace(PTRACE_SINGLESTEP, tid, nullptr, nullptr) < 0) {
            error::send_errno("Failed to single step");
        }
        --snip--
        if (waitpid(tid, &wait_status, 0) < 0) {
            error::send_errno("waitpid failed");
        }
        --snip--
    }

    --snip--
    if (ptrace(request, tid, nullptr, nullptr) < 0) {
        error::send_errno("Could not resume");
    }
    threads_.at(tid).state = process_state::running;
    state_ = process_state::running;
}
```

The `step_instruction` function needs to compute the TID and pass it to `get_pc`, `ptrace`, and `wait_on_signal`:

```
sdb::stop_reason sdb::process::step_instruction(std::optional<pid_t> otid) {
    auto tid = otid.value_or(current_thread_);
    --snip--
    auto pc = get_pc(tid);
    --snip--
    if (ptrace(PTRACE_SINGLESTEP, tid, nullptr, nullptr) < 0) {
        error::send_errno("Could not single step");
```

```
    }
    auto reason = wait_on_signal(tid);
    --snip--
}
```

The `get_current_hardware_stoppoint` function can just pass the optional TID through to its call to `get_registers`:

```
std::variant<sdb::breakpoint_site::id_type, sdb::watchpoint::id_type>
sdb::process::get_current_hardware_stoppoint(std::optional<pid_t> otid) const {
    auto& regs = get_registers(otid);
    --snip--
}
```

We must update a few other function calls, even though we didn't change their signatures. The `augment_stop_reason` function should extract the TID from the given `sdb::stop_reason`. It has one call to `ptrace` and one call to `get_registers` that need updating:

```
void sdb::process::augment_stop_reason(sdb::stop_reason& reason) {
    auto tid = reason.tid;
    --snip--
    if (ptrace(PTRACE_GETSIGINFO, tid, nullptr, &info) < 0) {
        error::send_errno("Failed to get signal info");
    }

    if (reason.info == (SIGTRAP | 0x80)) {
        --snip--
        auto& regs = get_registers(tid);
        --snip--
    }
    --snip--
}
```

We also need to modify the `maybe_resume_from_syscall` function to just return whether to resume:

```
bool sdb::process::should_resume_from_syscall(
      const stop_reason& reason) {
    --snip--
        if (found == end(to_catch)) {
            return true;
        }
    --snip--
    return false;
}
```

We update the function signature and change the code inside the `if` statement to return `true` rather than restarting the thread, and then we return `false` rather than the stop reason. This approach also eliminates the

problem I mentioned in Chapter 10 of the stack potentially getting too large when many syscalls are resumed from, as the eventual recursive call we'll make to `wait_on_signal` is easier for the compiler to optimize.

The last two functions in `sdb::process` that we will need to update are `set_hardware_stoppoint` and `clear_hardware_stoppoint`. The debug registers for each thread are separate, so we need to set and clear hardware stop points on every thread. We'll just compute the necessary changes for the active thread and then copy the same data into the other threads' registers:

```
int sdb::process::set_hardware_stoppoint(
    virt_addr address, stoppoint_mode mode, std::size_t size) {
    --snip--
    for (auto& [tid, _] : threads_) {
        if (tid == current_thread_) continue;
        auto& other_regs = get_registers(tid);
        other_regs.write_by_id(
            static_cast<register_id>(id), address.addr());
        other_regs.write_by_id(register_id::dr7, masked);
    }

    return free_space;
}
```

Before returning, we loop over all threads that aren't the active one and write the computed values into their registers. Next, we'll do the same for `clear_hardware_stoppoint`:

```
void sdb::process::clear_hardware_stoppoint(int index) {
    --snip--
    for (auto& [tid, _] : threads_) {
        if (tid == current_thread_) continue;
        auto& other_regs = get_registers(tid);
        other_regs.write_by_id(static_cast<register_id>(id), 0);
        other_regs.write_by_id(register_id::dr7, masked);
    }
}
```

Again, we loop over all threads other than the active one and write the same values into those threads' registers as we just wrote into the active one's.

The `sdb::register::write` function currently writes only to the active thread, so let's change that. Modify *sdb/src/registers.cpp* like this so that the calls to `write_fprs` and `write_user_area` also pass the TID for the thread to which the registers correspond:

```
void sdb::registers::write(
      const register_info& info, value val, bool commit) {
    --snip--
    if (commit) {
        if (info.type == register_type::fpr) {
```

```
            proc_->write_fprs(data_.i387, tid_);
        }
        else {
            --snip--
            proc_->write_user_area(aligned_offset,
                from_bytes<std::uint64_t>(bytes + aligned_offset), tid_);
        }
    }
}
```

Similarly, `sdb::register::flush` should pass through the correct TID in its calls to `write_fprs`, `write_gprs`, and `write_user_area`:

```
void sdb::registers::flush() {
    proc_->write_fprs(data_.i387, tid_);
    proc_->write_gprs(data_.regs, tid_);
    --snip--
    for (auto i = 0; i < 8; ++i) {
        --snip--
        proc_->write_user_area(reg_offset, bytes, tid_);
    }
}
```

With all those changes out of the way, we can now move on to updating `wait_on_signal`.

Multithreaded Signal Handling

We need to completely rewrite `wait_on_signal`, which handles process state changes, because handling signals when multiple threads are running is much more complicated than for just one thread. We'll move a bunch of the existing code into a `handle_signal` function, but we'll need to modify even that.

Before we write any code, let's think carefully about a few cases we should accommodate when `waitpid` returns inside of `wait_on_signal`:

We shouldn't report the signal or stop other threads.
We may intercept a signal that we shouldn't report to the user, such as those for thread creation or shared library load events. In those cases, we should restart the signaled thread and wait for another signal.

We shouldn't report the signal, and other threads have stopped.
If the previously discussed case occurs, but other threads have also stopped (potentially for reasons that we should report), we can once again restart the first signaled thread and wait for another signal. As soon as we call `waitpid` again, we'll get a signal from one of the other stopped threads and can decide what to do with it.

We should report the signal, and other threads are still running.
Recall that the debugger should operate in all-stop mode, meaning it must stop all threads before it can inspect or manipulate the program state. As such, if we intercept a signal that we should report to a user, we must first stop all threads. We can do this by sending a `SIGSTOP` to all threads other than the one that was signaled. We should use `waitpid` to wait for all of the other threads to stop after we send the `SIGSTOP`s before returning control to the user.

We should report the signal, and other threads have stopped.
This is the most complex scenario. Multiple threads may hit a breakpoint at the same time, or some threads may exit while others stop at a breakpoint. One thread may even exit after the debugger intercepts a `SIGTRAP` from another thread, but before the debugger sends out `SIGSTOP`s. We can't even detect whether a thread has terminated before sending it a `SIGSTOP` because it might exit between the detection and the signal being sent. Even worse, if we send a `SIGSTOP` to a thread that is already stopped due to a signal, this `SIGSTOP` will be queued up and immediately sent to the thread when it resumes, so the debugger will report a `SIGSTOP` it shouldn't be reporting.

To handle this situation, when we move to stop all running threads, we'll send `SIGSTOP`s to all threads other than the one originally signaled. We'll then use `waitpid` to wait for them all to stop. We'll inspect the status returned by `waitpid` to see the real reason each thread stopped. If the thread exited, we'll remove it from the list of tracked threads. If it stopped due to a `SIGSTOP`, we'll assume it's the `SIGSTOP` we sent. If it stopped due to some other signal (like a `SIGTRAP`), we'll record that the signal has a pending `SIGSTOP`. Then, the next time that thread gets a signal, if it's a `SIGSTOP`, we'll just restart the thread and carry on as if nothing happened.

In the case that some thread stopped due to a signal we shouldn't report, we'll report it anyway, as we're operating in all-stop mode and shouldn't restart that thread.

The main thread has exited.
If the main thread (the one whose TID matches the PID of the process) exits, we'll consider the whole process to have exited. This isn't necessarily the case, as processes can continue so long as any of their threads are still running, but it will simplify our implementation.

A non-main thread has exited.
If a thread other than the main thread exits, we'll print out a message saying that the thread exited, but we won't stop the process or consider the whole process terminated.

The user requests a single step for a thread with a pending `SIGSTOP`.
This is a tricky case: the current implementation of `step_instruction` issues a `PTRACE_SINGLESTEP` request and then calls `wait_on_signal`. But if there's a pending `SIGSTOP`, the thread will immediately stop, and then `wait_on_signal` will restart it. We could track whether a thread should be single-stepping and call `step_instruction` instead of `resume` inside of

wait_on_signal if a thread is stepping and needs restarting due to a signal we shouldn't report. We'll take the simpler approach, however, and check for a pending SIGSTOP on the thread before we single-step it. If one exists, we'll resume the thread and call waitpid to consume the signal before sending the PTRACE_SINGLESTEP request. In some cases, this approach will do the wrong thing. For example, if the thread single-steps into a syscall that shouldn't be reported, it will be sent a PTRACE_CONT instead of a PTRACE_SINGLESTEP. However, these are quite rare cases, so we'll pick the simpler implementation.

Updating the Signal Handling Function

Let's start implementing wait_on_signal. This function has a fair bit of nuance to it, so we'll write it piece by piece. We'll start with the call to waitpid:

```
sdb::stop_reason sdb::process::wait_on_signal(pid_t to_await) {
    int wait_status;
    int options = __WALL;
    pid_t tid;
    if ((tid = waitpid(to_await, &wait_status, options)) < 0) {
        error::send_errno("waitpid failed");
    }
```

We've made a few changes here. First, we now pass __WALL instead of 0 as the options argument. This argument tells waitpid to wait on both processes and threads. We pass to_await as the TID to await on, which defaults to -1, indicating that the function should wait until any child, not just the one with the given TID, has changed state. We also store the return value of waitpid, which is the TID of the thread whose state has changed.

Next, we'll handle the signal, which will involve potentially augmenting the stop reason with additional information about the stop, fixing the program counter for software breakpoints, reading registers, and unwinding the stack (all of the actions we used to perform inside of wait_on_signal).

When we inspect the signal, we may decide that we should resume the stopped thread (for example, if a stop occurs on an internal breakpoint or a syscall that shouldn't be caught). As such, we'll make a handle_signal function that returns a std::optional<sdb::stop_reason>. We'll use this function both for the original signal that wait_on_signal intercepts and for signals that we intercept when attempting to stop all running threads. It will take the initial stop reason given by waitpid and a bool indicating whether this is the original signal or one for a thread we're attempting to forcefully stop, and then it will return either an augmented stop reason or an empty optional, indicating that we should restart the thread.

We'll call this function inside of wait_on_signal:

```
    --snip--
    stop_reason reason(tid, wait_status);
    auto final_reason = handle_signal(reason, true);
```

```
    if (!final_reason) {
        resume(tid);
        return wait_on_signal(to_await);
    }
```

We compute the initial stop reason with the status returned by `waitpid`. We then handle the signal, passing `true` as the second argument to indicate that this is the original stop. If `handle_signal` returns an empty optional, we resume the stopped thread and re-call `wait_on_signal`.

If `handle_signal` returned an augmented stop reason, we should update the relevant `sdb::thread_state` object:

```
    --snip--
    reason = *final_reason;
    auto& thread = threads_.at(tid);
    thread.reason = reason;
    thread.state = reason.reason;
```

If the thread exited or terminated early, we should report this fact to both the target and the command line. We'll factor this information into a `report_thread_lifecycle_event` function. If the main thread ended, we should report that the process itself has finished. Otherwise, we should wait for another signal:

```
    --snip--
    if (reason.reason == process_state::exited or
        reason.reason == process_state::terminated) {
        report_thread_lifecycle_event(reason);
        if (tid == pid_) {
            state_ = reason.reason;
            return reason;
        }
        else {
            return wait_on_signal(-1);
        }
    }
```

At this point, we have a signal representing a stop we should report back to the user, so we'll stop all running threads, clean up any that exited, set the current state and active thread of the process, and then return:

```
    stop_running_threads();
    reason = cleanup_exited_threads(tid).value_or(reason);

    state_ = reason.reason;
    current_thread_ = tid;
    return reason;
}
```

We pass the TID of the thread whose signal we originally intercepted to `cleanup_exited_threads` so that that function can skip the original thread for cleanup, as we've already handled it. This function may find that the main thread exited, so it will return a `std::optional<sdb::stop_reason>`, which will store the reason to report to the user in the case that the main thread exited.

Let's add declarations for all of the functions we just pretended exist to *sdb/include/libsdb/process.hpp*. We'll add another member to `sdb::thread_state` that tracks whether that thread has a pending `SIGSTOP`. We'll also add functions to `sdb::process` for resuming every thread and swallowing pending `SIGSTOP`s. Finally, we'll declare two private functions called `send_continue` and `step_over_breakpoint`, which we'll use to implement `resume_all_threads`:

```
namespace sdb {
    struct thread_state {
        --snip--
        bool pending_sigstop = false;
    };

    class process {
    public:
        --snip--

        void stop_running_threads();
        void resume_all_threads();

        std::optional<sdb::stop_reason> cleanup_exited_threads(
            pid_t main_stop_tid);
        void report_thread_lifecycle_event(const stop_reason& reason);

        std::optional<stop_reason> handle_signal(
            stop_reason reason, bool is_main_stop);
        --snip--

    private:
        --snip--
        void swallow_pending_sigstop(pid_t tid);
        void send_continue(pid_t tid);
        void step_over_breakpoint(pid_t tid);
        --snip--
    };
}
```

Next, we'll implement `stop_running_threads` and `resume_all_threads`.

Stopping and Resuming Threads

The obvious way to implement the `resume_all_threads` function is to just iterate over all the threads the process is tracking and call `resume` with their

IDs. However, this is wrong for a very subtle reason: if one thread is running while another thread attempts to step over a breakpoint, that first thread may zoom past the breakpoint while it's disabled. To address this, we'll split up the existing `resume` function into to two halves: `step_over_breakpoint` and `send_continue`. Implement them in *sdb/src/process.cpp* and replace the existing implementation of `resume`:

```
void sdb::process::resume(std::optional<pid_t> otid) {
    auto tid = otid.value_or(current_thread_);
    step_over_breakpoint(tid);
    send_continue(tid);
}

void sdb::process::step_over_breakpoint(pid_t tid) {
    auto pc = get_pc(tid);
    if (breakpoint_sites_.enabled_stoppoint_at_address(pc)) {
        --snip--
    }
}

void sdb::process::send_continue(pid_t tid) {
    auto request =
    --snip--
    state_ = process_state::running;
}
```

The `resume` function keeps its first line, which retrieves the TID to operate on, then it calls the two new functions. The rest of the old code from `resume` is split between those functions: the second line and subsequent `if` statement go into the `step_over_breakpoint` function, and the rest goes in `send_continue`.

With this split complete, we can implement `resume_all_threads`:

```
void sdb::process::resume_all_threads() {
    for (auto& [tid, _] : threads_) {
        step_over_breakpoint(tid);
    }
    for (auto& [tid, _] : threads_) {
        send_continue(tid);
    }
}
```

Do not be tempted to merge those two `for` loops; if you do, you'll have the same problem that the original `resume` function had.

We can now resume threads, so let's think about how to stop them. To stop a thread, we should send it a `SIGSTOP`. We would usually achieve this with the `kill` syscall, but `kill` sends a signal to the entire process rather than to a single thread, so the wrong thread might handle it. Instead, we can use

tgkill (thread group kill), which takes the TGID of the thread group, the TID of the thread to signal, and the signal to send.

Once we've sent the stop signal, we should wait until the thread has stopped, handle the signal, and record the stop reason. We should also record whether there is now a pending SIGSTOP, in case we intercepted a different signal (indicating that the thread had already stopped, as we intercepted the original signal). Implement the function in *sdb/src/process.cpp*:

```
void sdb::process::stop_running_threads() {
    for (auto& [tid, thread] : threads_) {
        if (thread.state == process_state::running) {
            if (!thread.pending_sigstop) {
                tgkill(pid_, tid, SIGSTOP);
            }

            int wait_status;
            waitpid(tid, &wait_status, 0);

            stop_reason thread_reason(tid, wait_status);
            if (thread_reason.reason == process_state::stopped) {
                if (thread_reason.info != SIGSTOP) {
                    thread.pending_sigstop = true;
                }
                else if (thread.pending_sigstop) {
                    thread.pending_sigstop = false;
                }
            }

            thread_reason = handle_signal(
                thread_reason, false).value_or(thread_reason);
            threads_.at(tid).reason = thread_reason;
            threads_.at(tid).state = thread_reason.reason;
        }
    }
}
```

We loop over all threads. If the thread state says that the thread is running (which could be inaccurate if the thread has stopped in the meantime), and it doesn't already have a pending SIGSTOP, we send it a SIGSTOP. We wait on that specific thread, passing its TID to waitpid, and initialize an sdb::stop_reason object with the resulting status.

If the process was stopped due to a signal, then there are two possibilities we care about. If that signal wasn't a SIGSTOP, then there must now be a pending SIGSTOP on this thread, so we record that. Conversely, if the signal was a SIGSTOP and there's one recorded as pending, then we assume that this is the SIGSTOP we were waiting for and reset pending_sigstop.

Regardless of what the signal was, we call handle_signal, passing false as the second argument, as this signal isn't the original one that caused the

stop. The `handle_signal` function should never mandate a thread restart for threads other than the one that caused the original stop, so we can safely get the value out of the returned `std::optional` without checking it. Finally, we update the reason that the thread stopped, along with its state.

Cleaning Up Exited Threads

Next, let's clean up any threads that exited during the stopping and resuming process. We'll loop over all threads other than the one that caused the main stop. If they exited, we'll report this fact to the target and the command line and then remove them from the `threads_` member:

```
std::optional<sdb::stop_reason>
sdb::process::cleanup_exited_threads(pid_t main_stop_tid) {
    std::vector<pid_t> to_remove;
    std::optional<stop_reason> to_report;
    for (auto& [tid, thread] : threads_) {
        if (tid != main_stop_tid and
            (thread.state == process_state::exited or
             thread.state == process_state::terminated)) {
            report_thread_lifecycle_event(thread.reason);
            to_remove.push_back(tid);
            if (tid == pid_) {
                to_report = thread.reason;
            }
        }
    }

    for (auto tid : to_remove) {
        threads_.erase(tid);
    }
    return to_report;
}
```

Removing elements from containers while you're looping over them can be a recipe for disaster, as it invalidates any iterators that point into the containers. There are ways of dealing with this problem by calling the versions of `erase` that take iterators, but I find it clearer to iterate once, recording the elements that need removing, and then remove them all in a separate pass. As such, we initialize a `to_remove` variable that will hold the elements to remove. We also initialize an optional stop reason to report in the case that we find that the main thread has exited. We loop over all threads. If the current thread wasn't the one that triggered the stop and has exited or terminated, we report that event and push its TID into `to_remove`. If the main thread has exited, we record its stop reason. We then erase all of the entries in `threads_` that we recorded as needing to be removed. Finally, we return the stop reason that the debugger should report, if any.

Reporting Lifecycle Events

If a thread is created or exits, we should notify both the target and the command line. The target will use this information to update its own data structures, and the command line will print a message such as "Thread 42 exited." However, we shouldn't print to the command line from `libsdb`; that should be the job of the command line driver. As such, we'll add a callback to `sdb::process` that the command line can install to receive notifications when a thread starts or exits.

Add a new data member to `sdb::process` in *sdb/include/libsdb/process.hpp* that stores the callback, and a member function to install one:

```
#include <functional>
namespace sdb {
    class process {
    public:
        --snip--
        void install_thread_lifecycle_callback(
              std::function<void(const stop_reason&)> callback) {
            thread_lifecycle_callback_ = std::move(callback);
        }

    private:
        --snip--
        std::function<void(const stop_reason&)> thread_lifecycle_callback_;
    };
}
```

We store any function-like object that takes a stop reason and returns nothing. The `install_thread_lifecycle_callback` function sets this callback. We'll call this function in *sdb/tools/sdb.cpp* when we get around to making the changes necessary to support multithreading in the driver.

In the meantime, we implement `report_thread_lifecycle_event` in *sdb/src/process.cpp* to call both this callback and the `sdb::target::notify_thread_lifecycle_event` function we declared earlier but haven't implemented yet:

```
void sdb::process::report_thread_lifecycle_event(
      const sdb::stop_reason& reason) {
    if (thread_lifecycle_callback_) {
        thread_lifecycle_callback_(reason);
    }
    if (target_) {
        target_->notify_thread_lifecycle_event(reason);
    }
}
```

The last change we need to make to `sdb::process` is to implement `handle_signal`.

Handling the Signals

The `handle_signal` function will record any new threads the program spawns, handle pending `SIGSTOP`s, compute stop reasons, and handle stop points. Much of the implementation of `handle_signal` comes from the original implementation of `wait_on_signal`, but it's different enough to merit going through it in its entirety, piece by piece. We begin by extracting the thread's TID from the given stop reason, as we'll use it several times in this function, and then check for thread creation signals:

```
std::optional<sdb::stop_reason> sdb::process::handle_signal(
      stop_reason reason, bool is_main_stop) {
    auto tid = reason.tid;

    if (reason.trap_reason and
        *reason.trap_reason == trap_type::clone and
        is_main_stop) {
        return std::nullopt;
    }
```

If this stop is due to a clone event (because a thread was created) and this signal caused the original stop, we immediately return and restart the thread. We'll handle thread creation when we intercept the `SIGSTOP` that is sent to new threads.

If the debugger is attached to this process and the thread stopped due to a signal, we may need to take any of several potential actions. First, we'll handle the case in which we intercepted a signal sent to a new thread:

```
    --snip--
    if (is_attached_ and reason.reason == process_state::stopped) {
        if (!threads_.count(tid)) {
            threads_.emplace(tid, thread_state{ tid, registers(*this, tid) });
            report_thread_lifecycle_event(reason);
            if (is_main_stop) {
                return std::nullopt;
            }
        }
```

If we intercept a signal for a thread that the process isn't tracking yet, then a new thread must have been created, so we create a new `sdb::thread_state` object, add this to `threads_`, and then report the creation to the target and command line. If this signal caused the original stop, we don't want to report the signal, so we return and restart the thread.

Next, we handle the case in which we were trying to stop all threads and sent a `SIGSTOP` to a thread that was already stopped:

```
        --snip--
        if (threads_.at(tid).pending_sigstop and reason.info == SIGSTOP) {
            threads_.at(tid).pending_sigstop = false;
```

```
        return std::nullopt;
    }

    read_all_registers(tid);
    augment_stop_reason(reason);
```

If the signal we intercepted was a `SIGSTOP` and the thread to which it was sent has a pending `SIGSTOP`, we reset the `pending_sigstop` field, return, and restart the thread. Otherwise, we're most likely reporting a stop for this signal, so we read all the registers for the signaled thread and augment the stop reason with additional information about the signal.

Now we'll handle breakpoints, watchpoints, and syscall traps. The code here is mostly the same as its original implementation in `wait_on_signal`, so we won't spend too long on it:

```
    --snip--
    if (reason.info == SIGTRAP) {
        auto instr_begin = get_pc(tid) - 1;
        if (reason.trap_reason == trap_type::software_break and
            breakpoint_sites_.contains_address(instr_begin) and
            breakpoint_sites_.get_by_address(instr_begin).is_enabled()) {
            set_pc(instr_begin, tid);

            auto& bp = breakpoint_sites_.get_by_address(instr_begin);
            if (bp.parent_) {
                bool should_restart = bp.parent_->notify_hit();
                if (should_restart and is_main_stop) {
                    return std::nullopt;
                }
            }
        }
        else if (reason.trap_reason == trap_type::hardware_break) {
            auto id = get_current_hardware_stoppoint(tid);
            if (id.index() == 1) {
                watchpoints_.get_by_id(std::get<1>(id)).update_data();
            }
        }
        else if (reason.trap_reason == trap_type::syscall and
                 is_main_stop and
                 should_resume_from_syscall(reason)) {
            return std::nullopt;
        }
    }
```

If the signal occurred due to a software breakpoint enabled at the position just above the program counter, we set the program counter back by one and restart the thread if the breakpoint's hit handler tells us to and this signal caused the original stop. If the signal occurred due to a watchpoint, we

update its data. If the signal occurred due to a syscall, this signal caused the original stop, and the syscall isn't being tracked, we resume the thread.

Finally, we notify the target of the stop so it can unwind the thread's stack, then return the augmented stop reason:

```
        if (target_) target_->notify_stop(reason);
    }
    return reason;
}
```

Lastly, we must swallow any pending `SIGSTOP`s if a thread is single-stepped. We need to do this when `sdb::process::step_instruction` is called and when single-stepping over breakpoints in `sdb::process::resume`:

```
void sdb::process::resume(std::optional<pid_t> otid) {
    --snip--
    if (breakpoint_sites_.enabled_stoppoint_at_address(pc)) {
        --snip--
        swallow_pending_sigstop(tid);
        if (ptrace(PTRACE_SINGLESTEP, tid, nullptr, nullptr) < 0) {
            --snip--
        }
        --snip--
    }
    --snip--
}

sdb::stop_reason sdb::process::step_instruction(
      std::optional<pid_t> otid) {
    --snip--
    swallow_pending_sigstop(tid);
    if (ptrace(PTRACE_SINGLESTEP, tid, nullptr, nullptr) < 0) {
        --snip--
    }
}
```

In both cases, we call `swallow_pending_sigstop` just before issuing the `PTRACE_SINGLESTEP` request. The `swallow_pending_sigstop` implementation looks like this:

```
void sdb::process::swallow_pending_sigstop(pid_t tid) {
    if (threads_.at(tid).pending_sigstop) {
        ptrace(PTRACE_CONT, tid, nullptr, nullptr);
        waitpid(tid, nullptr, 0);
        threads_.at(tid).pending_sigstop = false;
    }
}
```

If the thread has a pending SIGSTOP, we continue it and call waitpid to consume the signal. We then record that the thread no longer has a pending SIGSTOP.

This completes all of the changes needed for sdb::process. Now we must make a similar but smaller set of changes to sdb::target.

Tracing Threads in the Target

As in sdb::process, we must update several functions in sdb::target to take an optional TID as an argument, which will default to the current thread's TID. For sdb::target, these functions include step_in, step_out, step_over, line_entry_at_pc, run_until_address, get_pc_file_address, and the two overloads of get_stack. Update them in *sdb/include/libsdb/target.hpp*:

```
namespace sdb {
    class target {
    public:
        --snip--
        stack& get_stack(std::optional<pid_t> otid = std::nullopt) {
            auto tid = otid.value_or(process_->current_thread());
            return threads_.at(tid).frames;
        }
        const stack& get_stack(std::optional<pid_t> otid = std::nullopt) const {
            return const_cast<target*>(this)->get_stack(otid);
        }

        sdb::stop_reason step_in(std::optional<pid_t> otid = std::nullopt);
        sdb::stop_reason step_out(std::optional<pid_t> otid = std::nullopt);
        sdb::stop_reason step_over(std::optional<pid_t> otid = std::nullopt);

        sdb::line_table::iterator line_entry_at_pc(
            std::optional<pid_t> otid = std::nullopt) const;
        sdb::stop_reason run_until_address(
            virt_addr address, std::optional<pid_t> otid = std::nullopt);

        file_addr get_pc_file_address(std::optional<pid_t> otid = std::nullopt) const;
        --snip--
    };
}
```

Most of these changes occur entirely in the parameter list; the only functional changes are in the implementations of the two get_stack overloads. The non-const version grabs the stack from the thread with the given TID (or the current thread's stack, if the caller passed no TID), and the const version defers to the non-const version.

Let's update the definitions of get_pc_file_address and line_entry_at_pc in *sdb/src/target.cpp*:

```
sdb::file_addr sdb::target::get_pc_file_address(
      std::optional<pid_t> otid) const {
    return process_->get_pc(otid).to_file_addr(elves_);
}

sdb::line_table::iterator
sdb::target::line_entry_at_pc(std::optional<pid_t> otid) const {
    auto pc = get_pc_file_address(otid);
    --snip--
}
```

In the former, we pass the argument through to `get_pc`. In the latter, we pass the argument through to `get_pc_file_address`. Next, we'll update `run_until_address`:

```
sdb::stop_reason sdb::target::run_until_address(
      virt_addr address, std::optional<pid_t> otid) {
    auto tid = otid.value_or(process_->current_thread());
    --snip--
    process_->resume(tid);
    auto reason = process_->wait_on_signal(tid);
    if (reason.is_breakpoint()
        and process_->get_pc(tid) == address) {
        --snip--
    }
    --snip--
    threads_.at(tid).state->reason = reason;
    return reason;
}
```

We pass the TID through the calls to `resume`, `wait_on_signal`, and `get_pc`, and we store the computed reason in the relevant `thread_state` object.

The `step_*` functions need a fair number of changes, but they're fairly mechanical. Let's start with `step_out`:

```
sdb::stop_reason sdb::target::step_out(std::optional<pid_t> otid) {
    auto tid = otid.value_or(process_->current_thread());
    auto& stack = get_stack(tid);
    --snip--
    if (has_inline_frames and at_inline_frame) {
        --snip--
        return run_until_address(return_address, tid);
    }
    --snip--
    for (auto frames = stack_.frames().size();
         stack_.frames().size() >= frames;) {
        reason = run_until_address(return_address, tid);
        --snip--
```

```
    }
    return reason;
}
```

At the start of the function, we calculate the correct TID. We update the call to `get_stack` and both calls to `run_until_address`. Let's move on to `step_in`:

```
sdb::stop_reason sdb::target::step_in(std::optional<pid_t> otid) {
    auto tid = otid.value_or(process_->current_thread());
    auto& stack = get_stack(tid);
    auto& thread = threads_.at(tid);
    if (stack.inline_height() > 0) {
        --snip--
        stop_reason reason(tid, process_state::stopped, SIGTRAP, trap_type::single_step);
        thread.state->reason = reason;
        return reason;
    }

    auto orig_line = line_entry_at_pc(tid);
    do {
        auto reason = process_->step_instruction(tid);
        if (!reason.is_step()) {
            thread.state->reason = reason;
            return reason;
        }
    } while ((line_entry_at_pc(tid) == orig_line
        or line_entry_at_pc(tid)->end_sequence)
        and line_entry_at_pc(tid) != line_table::iterator{});

    auto pc = get_pc_file_address(tid);
    --snip--
    if (pc.elf_file() != nullptr) {
        --snip--
        if (func and func->low_pc() == pc) {
            auto line = line_entry_at_pc(tid);
            --snip--
            if (line != line_table::iterator{}) {
                --snip--
                return run_until_address(line->address.to_virt_addr(), tid);
            }
        }
    }

    stop_reason reason(
        tid, process_state::stopped, SIGTRAP, trap_type::single_step);
    thread.state->reason = reason;
    return reason;
}
```

At the start of the function, we calculate the correct TID and grab references to the stack and thread to operate on. Before all of the existing `return reason;` statements, we store the reason in the thread's state. We also pass the TID to both constructions of `sdb::stop_reason`, the four calls to `line_entry_at_pc`, the call to `get_pc_file_address`, the call to `step_instruction`, and the call to `run_until_address`.

We're almost done; let's finish by updating `step_over`:

```
sdb::stop_reason sdb::target::step_over(std::optional<pid_t> otid) {
    auto tid = otid.value_or(process_->current_thread());
    auto& thread = threads_.at(tid);
    auto& stack = get_stack(tid);
    auto orig_line = line_entry_at_pc(tid);
    --snip--
    do {
        --snip--
        if (has_inline_frames and at_start_of_inline_frame) {
            --snip--
            reason = run_until_address(return_address, tid);
            if (!reason.is_step()
                or process_->get_pc(tid) != return_address) {
                thread.state->reason = reason;
                return reason;
            }
        }
        else if (auto instructions = disas.disassemble(2, process_->get_pc(tid));
            instructions[0].text.rfind("call") == 0) {
            reason = run_until_address(instructions[1].address, tid);
            if (!reason.is_step()
                or process_->get_pc(tid) != instructions[1].address) {
                thread.state->reason = reason;
                return reason;
            }
        }
        else {
            reason = process_->step_instruction(tid);
            if (!reason.is_step()) {
                thread.state->reason = reason;
                return reason;
            }
        }
    } while (line_entry_at_pc(tid) == orig_line or
        line_entry_at_pc(tid)->end_sequence and
        line_entry_at_pc(tid) != line_table::iterator{});
    thread.state->reason = reason;
    return reason;
}
```

Again, we calculate the correct TID at the start of the thread and grab references to the stack and thread. We also store the computed stop reason in the thread state before all `return` statements. We then update the calls to `line_entry_at_pc`, both calls to `run_until_address`, the three calls to `get_pc`, the call to `run_until_address`, the call to `step_instruction`, and the two calls to `line_entry_at_pc`.

Lastly, we need to update `notify_stop`:

```
void sdb::target::notify_stop(const sdb::stop_reason& reason) {
    threads_.at(reason.tid).frames.unwind();
}
```

We unwind the stack for the thread to which the signal corresponds.

Now for something new! Implement `notify_thread_lifecycle_event` to update the `threads_` structure:

```
void sdb::target::notify_thread_lifecycle_event(
      const stop_reason& reason) {
    auto tid = reason.tid;
    if (reason.reason == process_state::stopped) {
        auto& state = process_->thread_states()[tid];
        threads_.emplace(
            tid, thread{ &state, stack{this, tid} });
    }
    else {
        threads_.erase(tid);
    }
}
```

If the stop reason is a signal, this is a new thread creation event, so we create a new `sdb::thread` object. Otherwise, this is a thread exit event, so we erase the object with the given TID.

Before we can expose threads on the command line, we must also make a few changes to `sdb::stack`. In *sdb/src/stack.cpp*, update the `inline_stack_at_pc` function to forward the TID to which the stack corresponds through the call to `get_pc_file_address`:

```
std::vector<sdb::die>
sdb::stack::inline_stack_at_pc() const {
    auto pc = target_->get_pc_file_address(tid_);
    --snip--
}
```

Also update the `unwind` function to pass the TID through the calls to `get_pc`, `get_pc_file_address`, and `get_registers`:

```
void sdb::stack::unwind() {
    --snip--
    auto virt_pc = target_->get_process().get_pc(tid_);
    auto file_pc = target_->get_pc_file_address(tid_);
```

```
    auto& proc = target_->get_process();
    auto regs = proc.get_registers(tid_);
    --snip--
}
```

We're done! Now we can expose this new functionality on the command line.

Exposing Threads to the User

Most of the code currently in *sdb/tools/sdb.cpp* should work as is because we made passing TIDs to most of the functions in `sdb::process` and `sdb::target` optional. Nevertheless, we must still make a few updates and add some extra functionality.

First, we'll write a `thread_lifecycle_callback` function and install it on the `sdb::process` to print a message whenever a thread is created or destroyed:

```
namespace {
    void thread_lifecycle_callback(const sdb::stop_reason& reason) {
        std::string_view action;
        switch (reason.reason) {
        case sdb::process_state::exited: action = "exited"; break;
        case sdb::process_state::terminated: action = "terminated"; break;
        case sdb::process_state::stopped: action = "created"; break;
        }
        fmt::print("Thread {} {}\n", reason.tid, action);
    }
}
```

We create an empty string view that we'll fill in with the event type. We switch over the stop reason and populate the `action` variable with a string to print out. Finally, we print a message like "Thread 42 exited," depending on the calculated action and the TID of the thread to which the event corresponds. Install this callback in the `main` function, before the call to `main_loop`:

```
int main(int argc, const char* argv[]) {
    --snip--
    try {
        --snip--
        target->get_process().install_thread_lifecycle_callback(
            thread_lifecycle_callback);
        main_loop(target);
    }
    --snip--
}
```

We'll also add a new `thread` command to the driver. This command will have two subcommands: `list`, which will list the current threads and their

stop reasons, and `select <TID>`, which will set the active thread. The output for the `list` subcommand should look like this:

```
 Thread 12124: stopped with signal TRAP at 0x5555555552fa,
 multi_threaded.cpp:8 (multi_threaded`say_hi)  (breakpoint 1)
*Thread 12123: stopped with signal TRAP at 0x5555555552fa,
 multi_threaded.cpp:8 (multi_threaded`say_hi)  (breakpoint 1)
 Thread 12122: stopped with signal TRAP at 0x7ffff7c65a2d
```

You can see the threads, information about their stop reasons, and the currently selected thread prefixed with an asterisk.

Add a branch for this command in `handle_command`. While you're in that function, also update the `continue` command to call `resume_all_threads` rather than just `resume`:

```
namespace {
    void handle_command(
          std::unique_ptr<sdb::target>& target, std::string_view line) {
        --snip--
        if (is_prefix(command, "continue")) {
            process->resume_all_threads();
            auto reason = process->wait_on_signal();
            handle_stop(*target, reason);
        }
        --snip--
        else if (is_prefix(command, "thread")) {
            handle_thread_command(*target, args);
        }
        --snip--
    }
}
```

The `thread` command hands execution off to a `handle_thread_command` function, which we'll implement now:

```
namespace {
    void handle_thread_command(
          sdb::target& target, const std::vector<std::string>& args) {
        if (args.size() < 2) {
            print_help({ "help", "thread" });
            return;
        }

        if (is_prefix(args[1], "list")) {
            for (auto& [tid, thread] : target.threads()) {
                auto prefix = tid == target.get_process().current_thread() ? "*" : " ";
                fmt::print(
                    "{}Thread {}: {}\n", prefix, tid,
                    get_signal_stop_reason(target, thread.state->reason));
```

```
            }
        }

        else if (is_prefix(args[1], "select")) {
            if (args.size() != 3) {
                print_help({ "help", "thread" });
                return;
            }
            auto tid = sdb::to_integral<pid_t>(args[2]);
            if (!tid) {
                std::cerr << "Invalid thread id\n";
                return;
            }
            target.get_process().set_current_thread(*tid);
        }
    }
}
```

If the user passes an incorrect number of arguments, we print out a help message and return. Otherwise, if the subcommand is `list`, we loop over all threads. We compute the prefix for the line, which is an asterisk if this thread is the currently selected one and a blank space otherwise. We then print out the information for that thread. If the subcommand is `select`, we parse the TID as an integer and set the current thread in the target's process. If the user passes the wrong number of arguments, we again print a help message, and if parsing the TID fails, we print an error message.

Add the usual help messages to the `print_help` function:

```
namespace {
    void print_help(const std::vector<std::string>& args) {
        if (args.size() == 1) {
            std::cerr << R"(Available commands:
    --snip--
    thread      - Commands for operating on threads
    --snip--
)";
        }
        --snip--
        else if (is_prefix(args[1], "thread")) {
            std::cerr << R"(Available commands:
    list
    select <thread ID>
)";
        }
        --snip--
    }
}
```

We need to update three existing functions. The first is `get_sigtrap_info`, whose calls to `get_pc` and `get_current_hardware_stoppoint` need updating. The second is `get_signal_stop_reason`, whose calls to `get_pc` and `line_entry_at_pc` need updating:

```
namespace {
    std::string get_sigtrap_info(
          const sdb::process& process, sdb::stop_reason reason) {
        if (reason.trap_reason == sdb::trap_type::software_break) {
            auto& site = process.breakpoint_sites().get_by_address(
                process.get_pc(reason.tid));
            --snip--
        }
        if (reason.trap_reason == sdb::trap_type::hardware_break) {
            auto id = process.get_current_hardware_stoppoint(reason.tid);
            -snip--
        }
    }

    std::string get_signal_stop_reason(
          const sdb::target& target, sdb::stop_reason reason) {
        --snip--
        auto pc = process.get_pc(reason.tid);
        --snip--
        auto line = target.line_entry_at_pc(reason.tid);
        --snip--
    }
}
```

The final, more functional change is in `print_stop_reason`, which should refer to the thread that stopped if the stop occurred due to a signal, or to the whole process in the case of an exit or a termination:

```
namespace {
    void print_stop_reason(
          const sdb::target& target, sdb::stop_reason reason) {
        switch (reason.reason) {
        case sdb::process_state::exited:
            fmt::print("Process {} exited with status {}\n",
                target.get_process().pid(),
                static_cast<int>(reason.info));
            return;
        case sdb::process_state::terminated:
            fmt::print("Process {} terminated with signal {}\n",
                target.get_process().pid(),
                sigabbrev_np(reason.info));
            return;
        case sdb::process_state::stopped:
```

```
                fmt::print("Thread {} {}\n",
                    reason.tid, get_signal_stop_reason(target, reason));
                return;
            }
        }
    }
```

If an exit event occurred, we report that the process exited with the given status. If a termination occurred, we report that the process terminated with the signal name of the given signal. For regular signals, we report that the thread stopped and call `get_signal_stop_reason` to get a detailed message to print.

Now that we've completed the implementation, let's do some testing.

Testing

We'll test the new functionality with the *multi_threaded* program we wrote at the start of the chapter. Recall that it spawns 10 threads, each of which calls the `say_hi` function. Let's set a breakpoint on `say_hi`, step one of the threads that hits the breakpoint, step another one, and make sure that everything seems consistent.

Here is what one possible outcome might look like. Your output may look different, as the number of threads that hit the breakpoint depends on the scheduler:

```
$ tools/sdb test/targets/multi_threaded
Launched process with PID 16478
sdb> break set say_hi
sdb> c
Thread 16479 created
Thread 16480 created
Thread 16481 created
--snip--
sdb> thread list
 Thread 16481: stopped with signal TRAP at 0x5555555552fa,
 multi_threaded.cpp:7 (multi_threaded`say_hi) (breakpoint 1)
*Thread 16480: stopped with signal TRAP at 0x5555555552fa,
 multi_threaded.cpp:7 (multi_threaded`say_hi) (breakpoint 1)
 Thread 16479: stopped with signal TRAP at 0x5555555552fa,
 multi_threaded.cpp:7 (multi_threaded`say_hi) (breakpoint 1)
 Thread 16478: stopped with signal TRAP at 0x7ffff7c65a2d
sdb> next
--snip--
sdb> thread list
 Thread 16481: stopped with signal TRAP at 0x5555555552fa,
 multi_threaded.cpp:7 (multi_threaded`say_hi) (breakpoint 1)
*Thread 16480: stopped with signal TRAP at 0x555555555316,
 multi_threaded.cpp:7 (multi_threaded`say_hi) (single step)
```

```
 Thread 16479: stopped with signal TRAP at 0x5555555552fa,
 multi_threaded.cpp:7 (multi_threaded`say_hi) (breakpoint 1)
 Thread 16478: stopped with signal TRAP at 0x7ffff7c65a2d
sdb> thread select 16481
sdb> next
--snip--
sdb> thread list
*Thread 16481: stopped with signal TRAP at 0x555555555316,
 multi_threaded.cpp:7 (multi_threaded`say_hi) (single step)
 Thread 16480: stopped with signal TRAP at 0x555555555316,
 multi_threaded.cpp:7 (multi_threaded`say_hi) (single step)
 Thread 16479: stopped with signal TRAP at 0x5555555552fa,
 multi_threaded.cpp:7 (multi_threaded`say_hi) (breakpoint 1)
 Thread 16478: stopped with signal TRAP at 0x7ffff7c65a2d
```

We launch the program, set the breakpoint, and continue. The process spawns three threads, each of which hits the breakpoint. If we step one of the threads, its program counter changes and none of the other threads change state, which is what we're expecting. If we then select a different thread and step that, that thread is the only thread that changes state, just as we expect.

Let's write an automated test. We'll set a breakpoint on the `say_hi` function and keep resuming the process until all threads hit the breakpoint. We'll then resume one more time, and the process should exit. Add this to *sdb/test/tests.cpp*:

```
#include <set>
TEST_CASE("Multi-threading works", "[threads]") {
    auto dev_null = open("/dev/null", O_WRONLY);
    auto target = target::launch("targets/multi_threaded", dev_null);
    auto& proc = target->get_process();

    target->create_function_breakpoint("say_hi").enable();

    std::set<pid_t> tids;

    stop_reason reason;
    do {
        proc.resume_all_threads();
        reason = proc.wait_on_signal();
        for (auto& [tid, thread] : proc.thread_states()) {
            if (thread.reason.reason == sdb::process_state::stopped and
                tid != proc.pid()) {
                tids.insert(tid);
            }
        }
    } while (tids.size() < 10);
```

```
    REQUIRE(tids.size() == 10);

    proc.resume_all_threads();
    reason = proc.wait_on_signal();
    REQUIRE(reason.reason == sdb::process_state::exited);
    close(dev_null);
}
```

We launch the program and set a breakpoint on `say_hi`. We can't just resume the program 10 times, because multiple threads may hit the breakpoint on a single resumption. Instead, we create a `std::set` of TIDs. Every time we resume the process, if a non-main thread stopped, we insert the TID into the set. Ideally, we'd first check if it stopped due to a `SIGTRAP`, but in the case that a thread gets sent a `SIGSTOP` when it's about to execute a line with a breakpoint on it, our debugger misses that breakpoint. We keep going until all 10 threads hit the breakpoint, make sure we receive the expected number of TIDs, and then resume the process one more time and ensure that it exits.

Summary

In this chapter, you learned how Linux systems handle threads and extended your debugger with the ability to trace multiple threads in a process. You explored race conditions and addressed some of the ones that can affect the debugger. You also learned how `ptrace` and `waitpid` operate for multithreaded programs and used them to control exactly which threads the debugger manipulates. You then refactored the existing types in the debugger to support multiple threads and added tracing capabilities.

In the next chapter, you'll resume handling DWARF information and implement DWARF expression support, used for some stack unwinding scenarios. This debugger feature will also enable the tasks of the final two main chapters: manipulating variables and calling functions inside the inferior.

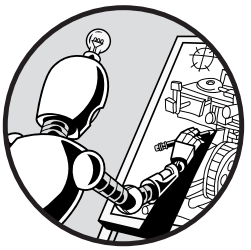
Check Your Knowledge

1. What is the difference between a thread and a process on Linux systems?
2. Which syscall creates a thread?
3. Which `ptrace` option halts the process when it spawns a new thread?
4. Which higher-level library is usually used to manage threads (as it saves the programmer from having to interact with syscalls)?
5. What is a race condition?
6. Which procfs directory lists all threads that are part of a process?

19

DWARF EXPRESSIONS

A figure
of speech
guiding you
to the forest.



The DWARF parser we have written so far can locate functions, but it can't yet locate variables. While finding functions is fairly easy, as they're loaded from the binary to a single point in memory for the entirety of the program, variables are more mercurial; they may live in registers for part of a function, move to the stack so the registers can serve other purposes, and then move back to registers. Furthermore, as you learned in Chapter 16, locating the stack frame for the current function isn't trivial, which further complicates locating variables.

DWARF *expressions* provide a way for the compiler to encode arbitrarily complex schemes for locating variables at runtime. They are encoded in a custom binary scheme that maintains a stack of values, and each instruction

manipulates the stack in some way: by pushing values to it, popping values from it, adding the two values at the top of the stack, and so on.

The language is sufficiently powerful that you could perform any algorithmic calculation with it. We call languages with this property *Turing-complete*, after early computer scientist Alan Turing. You probably didn't expect to write an interpreter for a Turing-complete programming language so you could locate a variable, but I promise it won't be too difficult. In this chapter, you'll implement an interpreter for DWARF expressions and add basic support for reading global variables to your debugger.

A Taxonomy of DWARF Expressions

Before we dive into the details, let's take a look at a few examples of DWARF expressions:

```
DW_OP_reg3
DW_OP_fbreg -50
DW_OP_bregx 54 32 DW_OP_deref
DW_OP_reg3 DW_OP_piece 4 DW_OP_reg10 DW_OP_piece 2
```

Each line represents a single expression. The first indicates that the value lives in the register with the DWARF ID 3, and the second indicates that the value lives at a byte offset of −50 from the *frame base*, another term for the canonical frame address (CFA) you learned about in Chapter 16. The third expression indicates that the address at which the value is stored is at a 32-byte offset from the contents of register 54. The final expression says that register 3 stores the first 4 bytes of the object, and register 10 stores the final 2 bytes.

These DWARF expressions live in DIE attributes that encode locations. For variables, this is the `DW_AT_location` attribute in the `DW_TAG_variable` or `DW_TAG_formal_parameter` DIE that represents that variable.

In this section, I'll detail the taxonomy of DWARF expressions, which is shown in Figure 19-1.

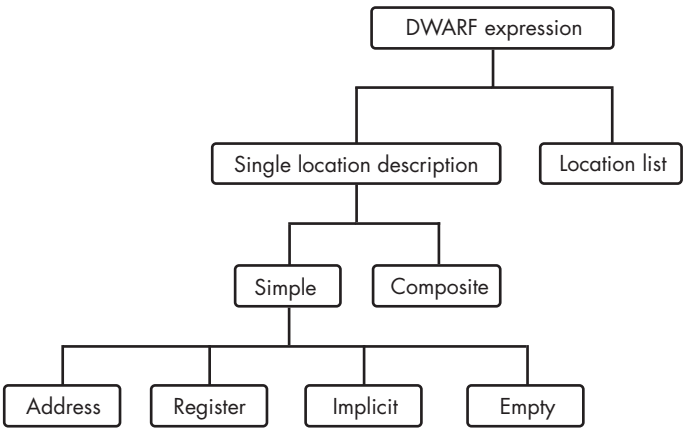


Figure 19-1: The taxonomy of DWARF expressions

A DWARF expression encodes either a single location description or a location list. A *single location description* locates a variable that uses a single address for its entire lifetime, whereas a *location list* locates a variable that has different locations depending on the current value of the program counter. Location lists consist of several single location descriptions, alongside the ranges of program counter values for which they are valid.

Single Location Descriptions

There are two types of single location descriptions: simple location descriptions and composite location descriptions. A *simple location description* locates a variable stored in one contiguous block (for example, a single register or uninterrupted area of memory), whereas a *composite location description* locates a variable that splits its storage between multiple areas (for example, a structure whose two members live in separate registers). Composite location descriptions consist of two or more simple location descriptions, each accompanied by the size of the data stored at that location.

An expression sometimes explicitly states its type; otherwise, we can deduce it from context. For now, it suffices to know that DWARF encodes single location descriptions with the form `DW_FORM_exprloc`, a `ULEB128` that holds the byte size of the expression followed by the data for the expression, and encodes location lists with the form `DW_FORM_sec_offset`, an offset into the `.debug_loc` section that stores the data for the location list.

DWARF uses a series of expression instructions that operate on a stack of values to encode single location descriptions, and a single location's type depends on the kinds of instructions used in the expression. If an expression uses either the `DW_OP_piece` or `DW_OP_bit_piece` instruction, it represents a composite location description; otherwise, it represents a simple location description.

Simple Location Descriptions

Simple location descriptions themselves have multiple classes:

Register Consists of a single instruction specifying the register that stores the variable in question. Registers whose DWARF IDs are between 0 and 31 inclusive are encoded with the opcodes between `DW_OP_reg0` and `DW_OP_reg31`. That is, the opcode `DW_OP_reg0` states that the variable is stored in the register with DWARF ID 0, the opcode `DW_OP_reg15` states that it is stored in the register with DWARF ID 15, and so on. Registers whose DWARF IDs lie outside of that range are encoded with the opcode `DW_OP_regx`, which has a single `ULEB128` operand giving the DWARF ID of the register in which the variable is stored. However, if the expression is in a frame information context (either in the `.eh_frame` section or in a `DW_AT_frame_base` attribute), then these opcodes are instead used to compute an address location description by pushing the value of the relevant register to the stack.

Implicit Indicates that the variable has no storage; the expression computes the value of the variable. Implicit location descriptions can encode locations in one of two ways. The simplest is with the `DW_OP_implicit_value` opcode, which has one `ULEB128` argument giving the length of the data for the variable, followed by a block of that size that encodes the data itself. The more complicated scheme uses the stack manipulation opcodes involved in representing address location descriptions, then ends the DWARF expression with a `DW_OP_stack_value` opcode, which indicates that the value at the top of the stack is the actual value of the variable rather than its address.

Address Indicates that the variable lives at the address computed by the expression. Non-empty DWARF expressions that don't use the previously mentioned opcodes all encode address location descriptions. After the execution of all of the instructions in the DWARF expression, the value at the top of the stack gives the address of the variable.

Empty An empty location description. If a DWARF expression has a length of zero, it represents an empty location description, meaning the address and contents of the variable being described are unknown. This usually occurs when the compiler optimizes out a variable.

Composite Location Descriptions

DWARF encodes composite location descriptions as a series of simple location descriptions, each terminated by a single `DW_OP_piece` or `DW_OP_bit_piece` opcode. The `DW_OP_piece` opcode has one `ULEB128` operand that gives the byte size of the data stored at the location given by the simple location description that precedes it. The `DW_OP_bit_piece` opcode has two operands: one `ULEB128` that gives the bit size (not byte size) of the data, and one `ULEB128` that gives the bit offset from the location given by the preceding simple location description at which the data is stored. You'll primarily find the `DW_OP_bit_piece` opcode used for types with bitfield members.

Executing Simple Location Descriptions

Let's write the code to parse and execute simple location descriptions. We'll create an `sdb::dwarf_expression` type that represents these descriptions and can evaluate them.

Defining Result Types

Begin by defining types that represent the various potential results of a simple location description evaluation. Add this code to *sdb/include/libsdb/dwarf.hpp*, above the definition of `sdb::attr`:

```
namespace sdb {
    class dwarf;
    class dwarf_expression {
```

```
    public:
        struct address_result {
            virt_addr address;
        };
        struct register_result {
            std::uint64_t reg_num;
        };
        struct data_result {
            span<const std::byte> data;
        };
        struct literal_result {
            std::uint64_t value;
        };
        struct empty_result {};
        using simple_location = std::variant<
            address_result, register_result,
            data_result, literal_result, empty_result>;

        struct pieces_result {
            struct piece {
                simple_location location;
                std::uint64_t bit_size;
                std::uint64_t offset = 0;
            };
            std::vector<piece> pieces;
        };
        using result = std::variant<simple_location, pieces_result>;
    };
}
```

We create `address_result`, `register_result`, and `empty_result` types, which correspond to address, register, and empty location descriptions. The `data_result` and `literal_result` types correspond to the two different ways of encoding implicit location descriptions: `data_result` is for the `DW_OP_implicit_result` opcode, which can have variably sized data, and `literal_result` is for the `DW_OP_stack_value` opcode, whose result is always 64 bits in size on x64. We define a `simple_location` type alias to a `std::variant` of all the possible simple location description types.

We then create a `pieces_result` type, which stores a vector of piece descriptions, each of which stores a single location, the bit size of the data, and the bit offset from the `location` member at which the data is really stored. We can use a single type to represent the results of both `DW_OP_piece` and `DW_OP_bit_piece` opcodes, as `DW_OP_piece` *`<byte size>`* is equivalent to `DW_OP_bit_piece` (*`<byte size>`*`*8) 0`.

Evaluating Expressions

Next, let's add data members, a constructor, and an `eval` member function, which will evaluate the expression and return the result:

```
namespace sdb {
    class dwarf;
    class dwarf_expression {
    public:
        --snip--
        dwarf_expression(const dwarf& parent,
                         span<const std::byte> expr_data,
                         bool in_frame_info)
            : parent_(&parent)
            , expr_data_(expr_data)
            , in_frame_info_(in_frame_info) {}

        result eval(
            const sdb::process& proc,
            const registers& regs,
            bool push_cfa = false) const;
    private:
        const dwarf* parent_;
        span<const std::byte> expr_data_;
        bool in_frame_info_;
    };
}
```

We store a pointer to the parent `sdb::dwarf` object so we can query it for the additional information needed to evaluate the expression, as well as the expression's data. Some instructions need to be interpreted differently depending on whether the expression resides in the `.eh_frame` section or a `DW_AT_frame_base` attribute, so we also store an `in_frame_info_` member that tracks this. The `eval` function will need the ability to read memory and registers, so it takes those values as parameters. It also takes an optional argument that indicates whether the CFA should be pushed to the stack before the expression is evaluated. This is used for running the `expression` and `val_expression` register restore rules when unwinding the stack.

Let's begin by implementing `eval` in *sdb/src/dwarf.cpp*. It's a long function, so we'll do it piece by piece, starting with some setup:

```
#include <functional>

sdb::dwarf_expression::result
sdb::dwarf_expression::eval(
      const sdb::process& proc, const registers& regs, bool push_cfa) const {
    cursor cur({ expr_data_.begin(), expr_data_.end() });
```

```
std::vector<std::uint64_t> stack;
if (push_cfa) stack.push_back(regs.cfa().addr());

std::optional<simple_location> most_recent_location;
std::vector<pieces_result::piece> pieces;

bool result_is_address = true;
```

We create a cursor to parse the data we need. We initialize an empty stack of values and push the CFA to it if required. We also initialize a variable for holding the most recently computed location and a vector to hold any pieces that we collect. By default, we assume that the result of the DWARF expression is an address, but a `DW_OP_stack_value` opcode may override this value, so we make a Boolean variable to track it.

Performing Stack Operations

Next, we'll define some helpers for some of the stack operations. Several stack operations perform an arithmetic or logical operation on the top two elements of the stack. We'll create a couple of lambda functions that can make this code shorter:

```
--snip--
auto binop = [&](auto op) {
    auto rhs = stack.back();
    stack.pop_back();
    auto lhs = stack.back();
    stack.pop_back();
    stack.push_back(op(lhs, rhs));
};

auto relop = [&](auto op) {
    auto rhs = static_cast<std::int64_t>(stack.back());
    stack.pop_back();
    auto lhs = static_cast<std::int64_t>(stack.back());
    stack.pop_back();
    stack.push_back(op(lhs, rhs) ? 1 : 0);
};
```

Both lambdas take a function that represents an arithmetic or relational operator. The arithmetic version pops the top two values from the stack and pushes back the result of calling the given operator with those values. The relational version pops the top two values from the stack, interprets them as signed integers, and then pushes either `1` or `0` to the stack, depending on whether the given operator returned `true` or `false`.

Some of the operations will require the current program counter value and the DIE for the function that is currently being executed, so we grab them:

```
--snip--
auto virt_pc = virt_addr{
    regs.read_by_id_as<std::uint64_t>(register_id::rip)
};
auto pc = virt_pc.to_file_addr(*parent_->elf_file());
auto func = parent_->function_containing_address(pc);
```

We read the current program counter value, convert it to a file address, and then look up the function containing that address.

Finding the Current Simple Location Description

As the last bit of setup before we start executing instructions, we'll write a lambda that figures out which simple location description the current state of the expression interpreter represents:

```
--snip--
auto get_current_location = [&]() {
    simple_location loc;
    if (stack.empty()) {
        loc = most_recent_location.value_or(empty_result{});
        most_recent_location.reset();
    }
    else if (result_is_address) {
        loc = address_result{ virt_addr{stack.back()} };
        stack.pop_back();
    }
    else {
        loc = literal_result{ stack.back() };
        stack.pop_back();
        result_is_address = true;
    }
    return loc;
};
```

If the stack is currently empty, the current location is either the most recently computed location (a variable or implicit data location) or an empty location. After checking the most recent location, we reset it. If the result is an address, the current location is an address location, and we'll find its value on the top of the stack. Otherwise, the current location is an implicit literal location, and we reset `result_is_address` back to the default.

Handling Opcode Ranges

Now we can start executing instructions. We'll loop until the cursor hits the end of its data, executing each instruction as we go. We'll start with the opcodes that cover ranges of values. These include the DW_OP_reg0 through DW_OP_reg31 opcodes you've seen already; the DW_OP_lit0 through DW_OP_lit31 opcodes, which push the specified integer onto the stack; and the DW_OP_breg0 through DW_OP_breg31 opcodes, which push the value stored in the given register, offset by a single SLEB128 operand, to the stack:

```
--snip--
while (!cur.finished()) {
    auto opcode = cur.u8();

    if (opcode >= DW_OP_lit0 and opcode <= DW_OP_lit31) {
        stack.push_back(opcode - DW_OP_lit0);
    }
    else if (opcode >= DW_OP_breg0 and opcode <= DW_OP_breg31) {
        auto reg = opcode - DW_OP_breg0;
        auto reg_val = regs.read(sdb::register_info_by_dwarf(reg));
        auto offset = cur.sleb128();
        stack.push_back(std::get<std::uint64_t>(reg_val) + offset);
    }
    else if (opcode >= DW_OP_reg0 and opcode <= DW_OP_reg31) {
        auto reg = opcode - DW_OP_reg0;
        if (in_frame_info_) {
            auto reg_val = regs.read(sdb::register_info_by_dwarf(reg));
            stack.push_back(std::get<std::uint64_t>(reg_val));
        }
        else {
            most_recent_location = register_result{
                static_cast<std::uint64_t>(reg)
            };
        }
    }
```

We parse the opcode, which is an unsigned 8-bit integer. If the opcode is one of the DW_OP_lit*<number>* opcodes, we push *<number>* to the stack. If it's one of the DW_OP_breg_*<DWARF register ID>* opcodes, we read the register with the given register ID as a 64-bit integer, offset the value by the SLEB128 operand, and push the result to the stack. If it's one of the DW_OP_reg_*<DWARF* register ID> opcodes, we need to vary our behavior depending on whether this expression is in a frame information context. If it is, we read the value of the register and push the result to the stack; this is the equivalent of DW_OP _breg*N* with an argument of 0 but takes less space to encode. If the expression is not in a frame information context, we record that the most recent location names the given register ID. The static_cast is there to silence compiler warnings.

Handling Individual Opcodes

Let's move on to opcodes we need to handle individually. We'll start with `DW_OP_addr`, which pushes its single 64-bit file address operand to the stack, and the `DW_OP_const<specifier>` opcodes, which push their single integer operand to the stack. The specifier is an optional byte size (`1`, `2`, `4`, or `8`) followed by either a `u` indicating an unsigned integer or an `s` indicating a signed integer. If the opcode includes no byte size, the integer is an `LEB128`:

```
--snip--
switch (opcode) {
case DW_OP_addr: {
    auto addr = file_addr{
        *parent_->elf_file(), cur.u64()
    };
    stack.push_back(addr.to_virt_addr().addr());
    break;
}
case DW_OP_const1u:
    stack.push_back(cur.u8());
    break;
case DW_OP_const1s:
    stack.push_back(cur.s8());
    break;
case DW_OP_const2u:
    stack.push_back(cur.u16());
    break;
case DW_OP_const2s:
    stack.push_back(cur.s16());
    break;
case DW_OP_const4u:
    stack.push_back(cur.u32());
    break;
case DW_OP_const4s:
    stack.push_back(cur.s32());
    break;
case DW_OP_const8u:
    stack.push_back(cur.u64());
    break;
case DW_OP_const8s:
    stack.push_back(cur.s64());
    break;
case DW_OP_constu:
    stack.push_back(cur.uleb128());
    break;
case DW_OP_consts:
    stack.push_back(cur.sleb128());
    break;
```

For the DW_OP_addr opcode, we parse the address and convert it to a real virtual address before pushing it to the stack. For the other opcodes, we simply parse the type given by the specifier and push it to the stack.

Next are the DW_OP_bregx and DW_OP_fbreg opcodes. The former is essentially the same as the DW_OP_breg_*<number>* opcodes for which we've already added support, except it specifies the register using a ULEB128 operand rather than encoding the register in the opcode. The DW_OP_fbreg opcode pushes to the stack the result of evaluating the DWARF expression held in the current function's DW_AT_frame_base attribute, offset by a single SLEB128 operand. The DW_AT_frame_base usually points to the function's CFA:

```
        --snip--
        case DW_OP_bregx: {
            auto reg_val = regs.read(
                sdb::register_info_by_dwarf(cur.uleb128()));
            stack.push_back(
                std::get<std::uint64_t>(reg_val) + cur.sleb128());
            break;
        }
        case DW_OP_fbreg: {
            auto offset = cur.sleb128();
            auto fb_loc = func.value()[DW_AT_frame_base]
                .as_evaluated_location(proc, regs, /*in_frame_info=*/true);
            auto fb_addr = read_frame_base_result(fb_loc, regs);
            stack.push_back(fb_addr.addr() + offset);
            break;
        }
```

For DW_OP_bregx, we read the register given by the ULEB128 operand, offset its value by the next SLEB128 operand, and push the result to the stack.

The DW_OP_fbreg opcode is a little more complicated, as we need to evaluate a nested DWARF expression. We parse the SLEB128 offset and store it. We then evaluate the frame base DWARF expression by retrieving the current function's DW_AT_frame_base attribute and calling an as_evaluated_location function on it, which we'll write later. This will evaluate the DWARF expression in that attribute and return the result as an sdb::dwarf_expression::result object. Recall that expressions in DW_AT_frame_base have to be interpreted slightly differently, so we pass true as the argument for in_frame_info.

Next, we need to retrieve the frame base from this DWARF expression result. We'll write a function named read_frame_base_result to do this work so that the code stays clean. For now, we call this function with the computed location and the current register set. Finally, we offset that address by the integer we parsed at the start and push the result to the stack.

Let's implement read_frame_base_result in *sdb/src/dwarf.cpp*:

```
namespace {
    sdb::virt_addr read_frame_base_result(
          const sdb::dwarf_expression::result& loc,
```

```
          const sdb::registers& regs) {
        auto simple_loc = std::get_if<sdb::dwarf_expression::simple_location>(&loc);
        if (!simple_loc) sdb::error::send("Unsupported frame base location");
        if (auto addr_res = std::get_if<sdb::dwarf_expression::address_result>(simple_loc)) {
            return addr_res->address;
        }
        sdb::error::send("Unsupported frame base location");
    }
}
```

The result should only ever be a simple address location; it doesn't really make sense to have the address split between multiple registers, and register locations should be impossible because the relevant DWARF expression instructions are handled as addresses in frame information contexts. We extract the simple address result and throw an exception if there's a problem.

Performing Stack Operations

The next few opcodes implement several classic stack operations, including duplicating the value at the top of the stack, pushing or popping a value to or from the stack, and more:

```
        --snip--
        case DW_OP_dup:
            stack.push_back(stack.back());
            break;
        case DW_OP_drop:
            stack.pop_back();
            break;
        case DW_OP_pick:
            stack.push_back(stack.rbegin()[cur.u8()]);
            break;
        case DW_OP_over:
            stack.push_back(stack.rbegin()[1]);
            break;
        case DW_OP_swap:
            std::swap(stack.rbegin()[0], stack.rbegin()[1]);
            break;
        case DW_OP_rot:
            std::rotate(
                stack.rbegin(), stack.rbegin() + 1, stack.rbegin() + 3);
            break;
```

The `DW_OP_dup` opcode duplicates the value on the top of the stack, while `DW_OP_drop` pops the value on the top of the stack and discards it.

The `DW_OP_pick` opcode pushes to the stack the value stored at `stack.rbegin()[N]`, where *N* is an unsigned 8-bit operand to the instruction. The `std::vector::rbegin` function retrieves a reverse iterator, so `stack.rbegin()[N]`

retrieves the element *N* places from the back of the vector, which represents the top of the stack.

The DW_OP_over opcode duplicates the value underneath the value on the top of the stack, DW_OP_swap swaps the two values at the top of the stack, and DW_OP_rot rotates the top three values of the stack, such that stack[0] becomes stack[2], stack[1] becomes stack[0], and stack[2] becomes stack[1]. We perform this by calling std::rotate with reverse iterators.

Executing Dereferencing Instructions

Next, we handle the dereferencing instructions. Continue the implementation like so:

```
--snip--
case DW_OP_deref: {
    auto addr = virt_addr{ stack.back() };
    stack.back() = proc.read_memory_as<std::uint64_t>(addr);
    break;
}
case DW_OP_deref_size: {
    auto addr = virt_addr{ stack.back() };
    auto size_to_read = cur.u8();
    auto mem = proc.read_memory(addr, size_to_read);
    std::uint64_t res = 0;
    std::copy(mem.data(), mem.data() + mem.size(),
        reinterpret_cast<std::byte*>(&res));
    stack.back() = res;
    break;
}
case DW_OP_xderef:
    sdb::error::send("DW_OP_xderef not supported");
case DW_OP_xderef_size:
    sdb::error::send("DW_OP_xderef_size not supported");
```

The DW_OP_deref opcode pushes the value stored at the address on the top of the stack to the stack. DW_OP_deref_size retrieves the value stored at the address on the top of the stack, reading the number of bytes specified by a single unsigned 8-bit operand. Then, it zero-extends the value to 64 bits and pushes the result to the stack.

The DW_OP_xderef opcode is an extended dereference that also takes an address space in which to operate. This is for machines that have multiple address spaces, like GPUs, and is unused on x64, so we won't support it. DW_OP_xderef_size is an extended sized dereference that also takes an address space to operate in. Similarly, we won't support this.

Addresses on the stack are virtual addresses. For DW_OP_deref, we pop the top address from the stack and read 64 bits from that address. For DW_OP_deref_size, we read the number of bytes specified by the argument and then

copy the bytes into a 64-bit integer filled with 0 bits. We throw exceptions for the two unsupported opcodes.

Pushing Specific Entities to the Stack

Certain instructions push the addresses of specific entities to the stack:

```
--snip--
case DW_OP_push_object_address:
    sdb::error::send("Unsupported opcode DW_OP_push_object_address");
case DW_OP_form_tls_address:
    sdb::error::send("Unsupported opcode DW_OP_form_tls_address");
case DW_OP_call_frame_cfa:
    stack.push_back(regs.cfa().addr());
    break;
```

We won't implement two of these entity-pushing opcodes. The opcode `DW_OP_push_object_address` supports certain Fortran dynamic types, which we don't care about. The opcode `DW_OP_form_tls_address` translates the address on the top of the stack to an address within the thread-local storage block for the object file to which the address belongs, and implementing it would require a significant amount of extra code. We throw exceptions for these two opcodes.

We do implement `DW_OP_call_frame_cfa`, which pushes the CFA for the current stack frame to the top of the DWARF expression stack. For this opcode, we retrieve the CFA address and push it to the stack.

Performing Arithmetic, Bitwise, and Relational Operations

Most of the arithmetic and bitwise operations follow the same pattern: they pop the top two values off the stack and push the result of executing a bitwise operator on them. The value on the top of the stack becomes the right-side operand, and the value below the one on the top of the stack becomes the left-side operand.

I won't describe these opcodes individually, as they're pretty straightforward. The only one that breaks the pattern slightly is the division operator, `DW_OP_div`, which interprets the operands as signed integers:

```
--snip--
case DW_OP_minus:
    binop(std::minus{});
    break;
case DW_OP_mod:
    binop(std::modulus{});
    break;
case DW_OP_mul:
    binop(std::multiplies{});
    break;
```

```
        case DW_OP_and:
            binop(std::bit_and{});
            break;
        case DW_OP_or:
            binop(std::bit_or{});
            break;
        case DW_OP_plus:
            binop(std::plus{});
            break;
        case DW_OP_shl:
            binop([](auto lhs, auto rhs) { return lhs << rhs; });
            break;
❶       case DW_OP_shr:
            binop([](auto lhs, auto rhs) { return lhs >> rhs; });
            break;
❷       case DW_OP_shra:
            binop([](auto lhs, auto rhs) {
                return static_cast<std::int64_t>(lhs) >> rhs; });
            break;
        case DW_OP_xor:
            binop(std::bit_xor{});
            break;
        case DW_OP_div: {
            auto rhs = static_cast<std::int64_t>(stack.back());
            stack.pop_back();
            auto lhs = static_cast<std::int64_t>(stack.back());
            stack.pop_back();
            stack.push_back(static_cast<std::uint64_t>(lhs / rhs));
            break;
        }
```

We implement the majority of these opcodes by passing one of the function objects from the <*functional*> header to the `binop` lambda we defined earlier. There are no function objects in the standard library for shifting operations, so we pass closures for those.

The difference between `DW_OP_shr` (logical shift) ❶ and `DW_OP_shra` (arithmetic shift) ❷ is that logical shifting replaces shifted bits with 0s, whereas arithmetic shifting replaces them with 1s if the shifted integer is negative and 0s if it is positive. We implement the former by shifting the unsigned integer at the top of the stack and the latter by first interpreting it as a signed integer. Performing a right shift on a signed integer in C++ is implementation-defined behavior, and GCC on x64 defines it to perform an arithmetic shift (as do most implementations). Finally, we implement division by interpreting the integers as signed before dividing them and pushing back the result.

A few arithmetic and bitwise operations don't follow the pattern we just described. These include `DW_OP_abs`, which interprets the value on the top of the stack as a signed integer and replaces it with the absolute value (removing the sign if there is one); `DW_OP_neg`, which interprets the value on

the top of the stack as a signed integer and replaces it with its negation; `DW_OP_plus_uconst`, which takes a single `ULEB128` argument and adds it to the value on the top of the stack; and `DW_OP_not`, which performs a bitwise negation on the value on the top of the stack.

Continue the implementation by handling these opcodes:

```
--snip--
case DW_OP_abs: {
    auto sval = static_cast<std::int64_t>(stack.back());
    sval = std::abs(sval);
    stack.back() = static_cast<std::uint64_t>(sval);
    break;
}
case DW_OP_neg: {
    auto neg = -static_cast<std::int64_t>(stack.back());
    stack.back() = static_cast<std::uint64_t>(neg);
    break;
}
case DW_OP_plus_uconst:
    stack.back() += cur.uleb128();
    break;
case DW_OP_not:
    stack.back() = ~stack.back();
    break;
```

Next are the six relational operators, corresponding to the `<`, `<=`, `>`, `>=`, `==`, and `!=` operators:

```
--snip--
case DW_OP_le:
    relop(std::less_equal{});
    break;
case DW_OP_ge:
    relop(std::greater_equal{});
    break;
case DW_OP_eq:
    relop(std::equal_to{});
    break;
case DW_OP_lt:
    relop(std::less{});
    break;
case DW_OP_gt:
    relop(std::greater{});
    break;
case DW_OP_ne:
    relop(std::not_equal_to{});
    break;
```

We implement these opcodes with the `relop` lambda we defined earlier.

Executing Control Flow Instructions

Next, we execute a handful of control flow instructions. `DW_OP_skip` represents an unconditional branch; it moves the cursor forward or backward by the amount given by a signed 2-byte operand. On the other hand, `DW_OP_bra` represents a conditional branch; it moves the cursor forward or backward by the amount given by a signed 2-byte operand if the value at the top of the stack isn't `0`. Implement these branch instructions as follows:

```
--snip--
case DW_OP_skip:
    cur += cur.s16();
    break;
case DW_OP_bra:
    if (stack.back() != 0) {
        cur += cur.s16();
    }
    stack.pop_back();
    break;
case DW_OP_call2:
    sdb::error::send("Unsupported opcode DW_OP_call2");
case DW_OP_call4:
    sdb::error::send("Unsupported opcode DW_OP_call4");
case DW_OP_call_ref:
    sdb::error::send("Unsupported opcode DW_OP_call_ref");
```

Note that we don't implement the `DW_OP_call[2/4/_ref]` opcodes, which represent function calls. These instructions hand off execution to the DWARF expression in the `DW_AT_location` attribute of the DIE identified by the operand, which can be 2 bytes, 4 bytes, or an offset identifying the DIE whose location it should execute.

The specification for `DW_OP_call_ref` is actually impossible to implement as written; even GDB doesn't implement it, so we can ignore it. The other two versions are very seldom used and add significant implementation overhead, so we'll ignore them, too. LLDB doesn't implement any of these operations either, so we're in good company.

Executing Non-Address Location Types

Certain instructions can indicate variable location types other than addresses. The `DW_OP_regx` instruction takes a `ULEB128` operand that encodes a DWARF register number. Like the `DW_OP_regN` instructions that you already implemented, its function depends on whether it is used in a frame information context. If it is, this instruction pushes the value of the specified register to the stack (in which case the variable location is an address location). If this DWARF expression is not in a frame information context, the `DW_OP_regx` instruction indicates that the variable lives in the specified register.

There are two other non-address location types to handle: `DW_OP_implicit_value` indicates that the variable has no location but has a known value encoded by a `ULEB128` operand giving the length of the data followed by the data itself, and `DW_OP_stack_value` indicates that the variable has no location but has a known value stored on the top of the stack.

Continue the implementation by handling these opcodes:

```
--snip--
case DW_OP_regx:
    if (in_frame_info_) {
        auto reg_val = regs.read(
            sdb::register_info_by_dwarf(cur.uleb128()));
        stack.push_back(
            std::get<std::uint64_t>(reg_val));
    }
    else {
        most_recent_location = register_result{
            cur.uleb128() };
    }
    break;

case DW_OP_implicit_value: {
    auto length = cur.uleb128();
    most_recent_location = data_result{
        span<const std::byte>{cur.position(), length} };
    break;
}
case DW_OP_stack_value:
    result_is_address = false;
    break;
```

For `DW_OP_regx`, we store a `register_result` with the parsed register number as the most recent location. For `DW_OP_implicit_value`, we instead store a `data_result` with the parsed block. For `DW_OP_stack_value`, we just record that the result isn't an address; the `get_current_location` function we wrote earlier handles the rest.

Also handle the `DW_OP_nop` instruction, which does nothing:

```
--snip--
case DW_OP_nop:
    break;
```

This opcode serves the purpose of aligning addresses to particular boundaries for platforms that require it.

Handling Composite Locations

Finally, we'll execute the two instructions for defining composite locations. `DW_OP_piece` indicates that the most recent location stores one piece of the

data for this variable, and the operand gives the piece's byte size. `DW_OP_bit_piece` indicates that the most recent location stores one piece of the data for this variable, but its location or size isn't aligned to a byte. A `ULEB128` operand gives the bit size of the piece, and a second `ULEB128` operand gives the bit offset from the start of this piece's location at which the data is actually stored.

Let's execute these instructions:

```
        --snip--
        case DW_OP_piece: {
            auto byte_size = cur.uleb128();
            simple_location loc = get_current_location();
            pieces.push_back(pieces_result::piece{ loc, byte_size*8 });
            break;
        }
        case DW_OP_bit_piece: {
            auto bit_size = cur.uleb128();
            auto offset = cur.uleb128();
            simple_location loc = get_current_location();
            pieces.push_back(pieces_result::piece{ loc, bit_size, offset });
            break;
        }
        }
    }
```

For `DW_OP_piece`, we parse the byte size, get the current location, and record the piece. The `piece` type stores the size of the piece in bits rather than bytes, so we multiply the byte size by eight before storing it. For `DW_OP_bit_piece`, we parse the bit size and offset, get the current location, and record the new piece.

After handling all the potential opcodes, we need to return the result. If there were any piece descriptions, we should return a `pieces_result` that stores the descriptions. Otherwise, we should just return the single result computed by `get_current_location`:

```
    --snip--
    if (!pieces.empty()) {
        return pieces_result{ pieces };
    }

    return get_current_location();
}
```

We've finished implementing single location descriptions. Let's move on to location lists.

Executing Location Lists

DWARF encodes location lists in a manner very similar to range lists, which you implemented as part of Chapter 12. They consist of a sequence of entries.

Each entry is either a location entry, an end-of-list entry, or a base address selection entry.

Location entries consist of two 64-bit unsigned integers that encode the start and end range for the entry, followed by a 16-bit unsigned integer giving the length of the following expression and the data for a single location description. These entries express the start and end range values as offsets from the current base address and denote the noninclusive range of program counter values to which they apply.

Like in range lists, *end-of-list* entries encode the end of the list and consist of two 64-bit integers with the value 0. Each *base address selection* entry consists of a 64-bit integer whose bits are all set to 1 and a 64-bit integer representing the base address for subsequent location entries. The `DW_AT_low_pc` attribute of the DIE to which this location list belongs gives the initial base address.

Let's define a type for representing location lists to *sdb/include/libsdb/dwarf.hpp*, underneath the definition of `sdb::dwarf_expression`:

```
namespace sdb {
    class location_list {
    public:
        location_list(
            const dwarf& parent, const compile_unit& cu,
            span<const std::byte> expr_data, bool in_frame_info)
            : parent_(&parent)
            , cu_(&cu)
            , expr_data_(expr_data)
            , in_frame_info_(in_frame_info) {}

        dwarf_expression::result eval(
            const sdb::process& proc, const registers& regs) const;
    private:
        const dwarf* parent_;
        const compile_unit* cu_;
        span<const std::byte> expr_data_;
        bool in_frame_info_;
    };
}
```

This interface is very similar to that of `sdb::dwarf_expression`. The difference is that `sdb::location_list` also requires a reference to the compile unit for the list so we can read its `DW_AT_low_pc` attribute to calculate the initial base address.

Implement the `eval` function in *sdb/src/dwarf.cpp*. It should find the single location description that corresponds to the program counter value in the given register set and evaluate that expression:

```
sdb::dwarf_expression::result
sdb::location_list::eval(
```

```
  const sdb::process& proc, const registers& regs) const {
    auto virt_pc = virt_addr{
        regs.read_by_id_as<std::uint64_t>(register_id::rip)
    };
    auto pc = virt_pc.to_file_addr(*parent_->elf_file());
    auto func = parent_->function_containing_address(pc);

    cursor cur({ expr_data_.begin(), expr_data_.end() }); ❶
    constexpr auto base_address_flag = ~static_cast<std::uint64_t>(0);
    auto base_address = cu_->root()[DW_AT_low_pc].as_address().addr();

    auto first = cur.u64();
    auto second = cur.u64();
    while (!(first == 0 and second == 0)) { ❷
        if (first == base_address_flag) { ❸
            base_address = second;
        }

        else { ❹
            auto length = cur.u16();
            if (pc.addr() >= base_address + first and ❺
                pc.addr() < base_address + second) {
                dwarf_expression expr(
                    *parent_, { cur.position(), cur.position() + length }, in_frame_info_);
                return expr.eval(proc, regs);
            }
            else { ❻
                cur += length;
            }
        }
        first = cur.u64();
        second = cur.u64();
    }

    return dwarf_expression::empty_result{};
}
```

First, we compute the program counter value as a file address and retrieve the DIE corresponding to the currently executing function. We initialize a cursor for the location list's data ❶, create a variable that stores the base address flag (a 64-bit integer with all bits set to 1), and retrieve the initial base address from the root compile unit DIE. Next, we read the list's first entry and loop until we read an end-of-list entry (one where both elements are 0s) ❷.

If the first element of the entry is the base address flag ❸, the entry is a base address selector, so we update the base address to the value stored in the second element. Otherwise, this entry is a location entry ❹. Such an

entry begins with a 2-byte unsigned integer that encodes the entry's data length, so we parse this.

If the range for the entry contains the current program counter value ❺, we've found the entry we're looking for, so we construct an `sdb::dwarf_expression` out of the entry's data and then evaluate it and return the result. We pass along `in_frame_info_` so that DWARF expression implements the correct behavior for the register instructions. If this entry's range doesn't contain the current program counter value ❻, we skip past the data and read the next entry. Finally, if we got to the end of the list without finding an entry that matches the current program counter, we return an empty result.

We've now supported both single location descriptions and location lists. Let's add some helper functions to `sdb::attr` to expose these through DIE attributes.

Exposing Attributes to the User

Users of `libsdb` should be able to call `as_expression` on a DIE attribute to retrieve its contents as an `sdb::dwarf_expression` and call `as_location_list` to retrieve the attribute's contents as a location list. For convenience, they should also be able to call `as_evaluated_location`, which should indicate whether the given attribute represents a single location description or a location list, and then evaluate the contents. Add declarations for these functions to `sdb::attr` in *sdb/include/libsdb/dwarf.hpp*:

```
namespace sdb {
    class attr {
    public:
        --snip--
        dwarf_expression as_expression(bool in_frame_info) const;
        location_list as_location_list(bool in_frame_info) const;
        dwarf_expression::result as_evaluated_location(
            const sdb::process& proc,
            const registers& regs,
            bool in_frame_info) const;

        --snip--
    };
}
```

The first two functions require only an argument to state whether the expression is in a frame information context because `sdb::attr` already tracks all the other necessary data. The final function additionally requires a process and register set to pass on to the expression evaluator.

Implement these functions in *sdb/dwarf.cpp*, starting with the first two:

```
sdb::dwarf_expression sdb::attr::as_expression(bool in_frame_info) const {
    cursor cur({ location_, cu_->data().end() });
```

```
    auto length = cur.uleb128();
    span<const std::byte> data{ cur.position(), length };
    return dwarf_expression{ *cu_->dwarf_info(), data, in_frame_info };
}

sdb::location_list sdb::attr::as_location_list(bool in_frame_info) const {
    auto section = cu_->dwarf_info()->elf_file()->get_section_contents(
        ".debug_loc");

    cursor cur({ location_, cu_->data().end() });
    auto offset = cur.u32();

    span<const std::byte> data(section.begin() + offset, section.end());
    return location_list{ *cu_->dwarf_info(), *cu_, data, in_frame_info };
}
```

The `as_expression` function creates a cursor, parses the length of the single location expression, creates a span that covers the expression's data, and returns an `sdb::dwarf_expression` that represents the single location description.

DWARF encodes location lists as an offset into the `.debug_loc` section, so `as_location_list` locates that section, parses the 4-byte offset stored in the attribute, computes a span for the data, and returns an `sdb::location_list` that represents the location list. Note that we won't know the length of the location list before evaluating it, so we just use the end of the `.debug_loc` section as the marker for the end of the data. This is fine because the location list's end-of-list entry tells us when to stop parsing entries.

Now that we've defined these two functions, we can implement the function `as_evaluated_location`:

```
sdb::dwarf_expression::result
sdb::attr::as_evaluated_location(
      const sdb::process& proc,
      const registers& regs,
      bool in_frame_info) const {
    if (form_ == DW_FORM_exprloc) {
        auto expr = as_expression(in_frame_info);
        return expr.eval(proc, regs);
    }
    else if (form_ == DW_FORM_sec_offset) {
        auto loc_list = as_location_list(in_frame_info);
        return loc_list.eval(proc, regs);
    }
    else {
        error::send("Invalid location type");
    }
}
```

DWARF encodes single location descriptions with the form `DW_FORM_exprloc`, so if the attribute has this form, we call `as_expression` and evaluate the result. Location lists instead have the form `DW_FORM_sec_offset`; if we encounter that, we call `as_location_list` and then evaluate the result. If the attribute's form is something else, an error has occurred, so we throw an exception.

Before we use our new DWARF expression support to read some variables, let's add support to the stack unwinder for DWARF expressions.

Augmenting the Stack Unwinder

When we first implemented stack unwinding in Chapter 16, we skipped over DWARF expressions. Recall that two register restoration rules involve DWARF expressions: `val_expression(E)` and `expression(E)`. In the first case, `val_expression(E)`, executing the DWARF expression `E` gives the previous value of the register. In the second case, `expression(E)`, executing the DWARF expression `E` gives that previous value's address.

One CFA rule, also named `expression(E)`, involves DWARF expressions as well. The rule calculates the CFA by executing the DWARF expression `E`. Lastly, DWARF expressions can appear in the following three DWARF call frame information instructions:

`DW_CFA_expression` Takes a `ULEB128` operand representing the DWARF register number for which the rule is being defined and an additional `DW_FORM_block` operand that represents a DWARF expression `E` and sets the rule for the register to `expression(E)`. Prior to the execution of `E`, this expression pushes the CFA to the DWARF expression stack.

`DW_CFA_val_expression` The same as `DW_CFA_expression`, but defines the register's rule to be `val_expression(E)` rather than `expression(E)`.

`DW_CFA_def_cfa_expression` Takes a `DW_FORM_exprloc` operand that encodes a DWARF expression `E` and sets the CFA rule to `expression(E)`.

Let's add support for these rule types and instructions. In *sdb/src/dwarf.cpp*, add new register rule types for `expression(E)` and `val_expression(E)`, and add a new CFA rule type for `expression(E)`:

```
namespace {
    struct register_rule {
        --snip--
    };
    struct expr_rule {
        sdb::dwarf_expression expr;
    };
    struct val_expr_rule {
        sdb::dwarf_expression expr;
    };
    struct cfa_register_rule {
        --snip--
```

```
    };
    struct cfa_expr_rule {
        sdb::dwarf_expression expr;
    };
}
```

Replace the current `cfa_rule` member of the `unwind_context` type with a `std::variant` type covering the two possible rule types, add the two new register restore rule types to the `rule` type alias, and modify `rule_stack` to use the new type for the CFA rule:

```
namespace {
    struct unwind_context {
        --snip--
        using cfa_rule_type = std::variant<cfa_register_rule, cfa_expr_rule>;
        cfa_rule_type cfa_rule;
        using rule = std::variant<
            undefined_rule, same_rule, offset_rule,
            val_offset_rule, register_rule,
            expr_rule, val_expr_rule>;
        --snip--
        std::vector<std::pair<ruleset, cfa_rule_type>> rule_stack;
    };
}
```

We also need to update our existing handling of the CFA register rule instructions in `execute_cfi_instruction`, as the rule is now a `std::variant` rather than a `cfa_register_rule`:

```
--snip--
case DW_CFA_def_cfa:
    ctx.cfa_rule = cfa_register_rule{
     ❶ static_cast<std::uint32_t>(cur.uleb128()),
        static_cast<std::uint32_t>(cur.uleb128())
    };
    break;
case DW_CFA_def_cfa_sf:
    ctx.cfa_rule = cfa_register_rule{
     ❷ static_cast<std::uint32_t>(cur.uleb128()),
        cur.sleb128() * cie.data_alignment_factor
    };
    break;
case DW_CFA_def_cfa_register:
    std::get<cfa_register_rule>(ctx.cfa_rule).reg = cur.uleb128();
    break;
case DW_CFA_def_cfa_offset:
    std::get<cfa_register_rule>(ctx.cfa_rule).offset = cur.uleb128();
    break;
```

```
case DW_CFA_def_cfa_offset_sf:
    std::get<cfa_register_rule>(ctx.cfa_rule).offset =
        cur.sleb128() * cie.data_alignment_factor;
    break;
--snip--
```

For the first two instructions, we replace the existing `cfa_rule` member with a new `cfa_register_rule` that has the same content as before. For the other three instructions, we assume that the current CFA rule is a register rule, so we use `std::get` to retrieve the `cfa_register_rule` object and update the relevant member to the same value we did previously. All the `static_casts` in this snippet are to silence compiler warnings.

For those of you worried about C++'s evaluation order when we parse two values from the cursor in a single expression ❶ ❷, have no fear, because the order is defined to be left to right when using list initialization (which is the initialization form used here with the curly brackets).

Now we can add support for `DW_CFA_def_cfa_expression`. Replace the existing code (which just throws an exception) with this:

```
--snip--
case DW_CFA_def_cfa_expression: {
    auto length = cur.uleb128();
    auto expr = sdb::dwarf_expression{
        elf, { cur.position(), cur.position() + length }, true };
    ctx.cfa_rule = cfa_expr_rule{ expr };
    break;
}
--snip--
```

We read the `ULEB128` data length, create an `sdb::dwarf_expression` with the expression's data, and store a `cfa_expr_rule` with this expression as the current CFA rule. We pass `true` as the final argument because this expression is in a frame information context.

Also replace the current handling of the `DW_CFA_expression` and `DW_CFA_val_expression` instructions, like so:

```
--snip--
case DW_CFA_expression: {
    auto reg = cur.uleb128();
    auto length = cur.uleb128();
    auto expr = sdb::dwarf_expression{
        elf, { cur.position(), cur.position() + length }, true };
    ctx.register_rules.emplace(reg, expr_rule{ expr });
    break;
}
case DW_CFA_val_expression: {
    auto reg = cur.uleb128();
    auto length = cur.uleb128();
    auto expr = sdb::dwarf_expression{
        elf, { cur.position(), cur.position() + length }, true };
```

```
            ctx.register_rules.emplace(reg, val_expr_rule{ expr });
            break;
        }
        --snip--
```

These implementations are very similar to the one we used for the `DW_CFA_def_cfa_expression` instruction. However, we also parse the DWARF register number operand and store it in the created register rule.

We need to handle both possible CFA rules in `execute_unwind_rules`. This should resolve the real address of the CFA, update the old register set's CFA, and continue on as before:

```
namespace {
    sdb::registers execute_unwind_rules(
          unwind_context& ctx, sdb::registers& old_regs,
          const sdb::process& proc) {
        auto unwound_regs = old_regs;

        auto dwexp_addr_result = [&](const auto& res) {
            auto& loc = std::get<sdb::dwarf_expression::simple_location>(res);
            auto& addr_res = std::get<sdb::dwarf_expression::address_result>(loc);
            return sdb::virt_addr{ addr_res.address.addr() };
        };

        std::uint64_t cfa;
        if (auto reg_rule = std::get_if<cfa_register_rule>(&ctx.cfa_rule)) {
            auto reg_info = sdb::register_info_by_dwarf(reg_rule->reg);
            cfa = std::get<std::uint64_t>(old_regs.read(reg_info)) +
                reg_rule->offset;
        }
        else if (auto expr = std::get_if<cfa_expr_rule>(&ctx.cfa_rule)) {
            auto res = expr->expr.eval(proc, old_regs);
            cfa = dwexp_addr_result(res).addr();
        }
        old_regs.set_cfa(sdb::virt_addr{ cfa });
        --snip--
    }
}
```

All DWARF expression results in call frame information instructions are single address results, so we introduce a lambda function to easily extract the address that the expression computes. If the CFA rule is a register rule, we follow the same procedure as before: we read the rule's register, add the rule's offset to the result, and store this as the CFA. If the CFA rule is a DWARF expression rule, we evaluate it, extract the address from the result, and store that as the CFA. After computing the CFA, we store it in the old register set, as we used to.

In the same function, we also need to handle the two new register restore rules. Add code to handle these in new branches underneath the existing one for handling `val_offset` rules:

```
else if (auto val_offset = std::get_if<val_offset_rule>(&rule)) {
    --snip--
}
else if (auto expr = std::get_if<expr_rule>(&rule)) {
    auto res = expr->expr.eval(proc, old_regs, true);
    auto addr = dwexp_addr_result(res);
    auto value = proc.read_memory_as<std::uint64_t>(addr);
    unwound_regs.write(reg_info, { value }, false);
}
else if (auto val_expr = std::get_if<val_expr_rule>(&rule)) {
    auto res = val_expr->expr.eval(proc, old_regs, true);
    auto addr = dwexp_addr_result(res);
    unwound_regs.write(reg_info, { addr.addr() }, false);
}
```

In each case, we evaluate the DWARF expression and extract the computed address. We pass `true` as the final argument to push the CFA to the stack prior to evaluation. For `expression` rules, we read the memory at that address and store it as the unwound register value. For `val_expression` rules, we store the computed address as the unwound register value.

At this point, the implementation of DWARF expressions is complete!If you've never implemented an interpreter for a Turing-complete language before, you can finally say that you have.

Reading Global Variables

Displaying the contents of variables to the user with the correct type information is a more complex topic, and we'll mostly leave it until the next chapter. In this section, however, we'll add some very basic support for reading global integer variables so we can test the functionality we just added.

Reading DWARF Expression Results

Let's start by adding a function to `sdb::target` that takes a DWARF expression result and returns the bytes that make up the variable stored there. Declare this function in *sdb/include/libsdb/target.hpp*:

```
namespace sdb {
    class target {
    public:
        --snip--
        std::vector<std::byte> read_location_data(
            const dwarf_expression::result& loc, std::size_t size,
            std::optional<pid_t> otid = std::nullopt) const;
        --snip--
```

```
    };
}
```

The function takes the DWARF expression result that locates the data, a number of bytes to read, and which thread to operate on (because we may have to read from the registers of a specific thread).

Implement it in *sdb/src/target.cpp*. The implementation is rather complex, especially for composite locations, so we'll go through it bit by bit. We'll start by following the usual process of calculating which thread to operate on by checking the `otid` argument, which may have the default value of `std::nullopt`:

```
std::vector<std::byte> sdb::target::read_location_data(
      const dwarf_expression::result& loc, std::size_t size,
      std::optional<pid_t> otid) const {
    auto tid = otid.value_or(process_->current_thread());
```

Next, we'll begin handling simple locations, starting with register locations. We want to copy the contents of the register into a vector of bytes, but `std::register::read` returns a `std::variant` whose size and type depend on which register is being read. We can use `std::visit` to extract the bytes from this variant:

```
--snip--
if (auto simple_loc = std::get_if<sdb::dwarf_expression::simple_location>(&loc)) {
    if (auto reg_loc = std::get_if<sdb::dwarf_expression::register_result>(simple_loc)) {
        auto reg_info = register_info_by_dwarf(reg_loc->reg_num);
        auto reg_value = threads_.at(tid).frames.current_frame().regs.read(reg_info);
        auto get_bytes = [](auto value) {
            std::vector<std::byte> bytes(sizeof(value));
            auto begin = reinterpret_cast<const std::byte*>(&value);
            std::copy(begin, begin + sizeof(value), bytes.data());
            return bytes;
        };
        return std::visit(get_bytes, reg_value);
    }
```

We try to extract a simple location from the given result. If the operation succeeds, we try to extract a register result, and if that succeeds, we perform the copy. We retrieve the register info for the DWARF register number stored in the result object, then read the value of the relevant register in the requested thread. We next define a generic `get_bytes` lambda function that returns the contents of its argument as a vector of bytes. We pass it to `std::visit`, along with the register value. Because we'll call the `get_bytes` lambda with the contents of the variant, the `value` parameter will be of type `std::uint64_t` if the variant stores a `std::uint64_t`, of type `sdb::byte64` if it stores an `sdb::byte64`, and so on.

Next, let's handle address results, which should read the requested number of bytes from the memory at the address stored in the result:

```
    --snip--
    else if (
        auto addr_res = std::get_if<sdb::dwarf_expression::address_result>(simple_loc)) {
        return process_->read_memory(addr_res->address, size);
    }
```

Data results are also pretty straightforward. We construct the return value (a `std::vector<std::byte>`) with the iterators for the beginning and end of the data span, which copies the data from the span into the vector:

```
    --snip--
    else if (auto data_res = std::get_if<sdb::dwarf_expression::data_result>(simple_loc)) {
        return { data_res->data.begin(), data_res->data.end() };
    }
```

Literal results are similar. We reinterpret a pointer to the stored value as a pointer to bytes and pass this value to the `std::vector` constructor to copy the bytes into the return value:

```
    --snip--
    else if (auto literal_res =
          std::get_if<sdb::dwarf_expression::literal_result>(simple_loc)) {
        auto begin = reinterpret_cast<const std::byte*>(&literal_res->value);
        return { begin, begin + size };
    }
}
```

Finally, we need to handle composite results. This is significantly trickier than the previous cases because pieces aren't necessarily aligned to byte boundaries. We'll start by implementing the simple case, in which the data is aligned on a byte boundary:

```
--snip--
else if (auto pieces_res = std::get_if<sdb::dwarf_expression::pieces_result>(&loc)) {
    std::vector<std::byte> data(size);
    std::size_t offset = 0;
    for (auto& piece : pieces_res->pieces) {
        auto byte_size = (piece.bit_size + 7) / 8;
        auto piece_data = read_location_data(piece.location, byte_size, otid);
        if (offset % 8 == 0 and piece.offset == 0 and piece.bit_size % 8 == 0) {
            std::copy(piece_data.begin(), piece_data.end(), data.begin() + offset / 8);
            offset += piece.bit_size;
        }
```

We try to extract the `pieces_result` object. If that succeeds, we initialize the return vector and create a variable to store the current bit offset at which new data should be written into this vector. We iterate over all of the pieces

in the result, reading the data for the current piece's location by recursively calling `read_location_data`.

We pass the number of bytes this function should read, which we calculate by adding seven to the piece's bit size and dividing it by eight. We must do this because if the bit size is not exactly divisible by eight, integer division will cut off the remainder, but we want to round up toward the nearest byte.

If the current bit offset and the piece's bit size are aligned to a byte, and the piece's bit offset is 0, we're in the simple case and can copy the bytes into the `data` vector.

Now we'll tackle the more complicated case, where we need to handle copying data that isn't aligned to a byte. The algorithm needs to work for arbitrarily sized data copied from a potentially non-byte-aligned position to a different potentially non-byte-aligned position. We'll pretend that we have a function called `memcpy_bits` that performs this work, with this signature:

```
void memcpy_bits(std::uint8_t* dest, std::uint32_t dest_bit,
    const std::uint8_t* src, std::uint32_t src_bit,
    std::uint32_t n_bits);
```

Let's use this function to perform the data copy:

```
    --snip--
            else {
                auto dest = reinterpret_cast<std::uint8_t*>(data.data());
                auto src = reinterpret_cast<const std::uint8_t*>(piece_data.data());
                memcpy_bits(dest, 0, src, piece.offset, piece.bit_size);
            }
        }

        return data;
    }
    sdb::error::send("Invalid location type");
}
```

We cast the destination and source pointers to the correct types. We call `memcpy_bits` with these pointers, supplying 0 as the bit offset into the destination data into which we should copy, `piece.offset` as the bit offset into the source data at which we should start the copy, and `piece.bit_size` as the number of bits to copy. The error send at the end is mostly to silence compiler warnings but may come in handy if some other location type is added in another DWARF version.

Let's implement `memcpy_bits` in *sdb/include/bit.hpp*:

```
namespace sdb {
    inline void memcpy_bits(std::uint8_t* dest, std::uint32_t dest_bit,
          const std::uint8_t* src, std::uint32_t src_bit,
          std::uint32_t n_bits) {
        for (; n_bits; --n_bits, ++src_bit, ++dest_bit) {
            std::uint8_t dest_mask = 1 << (dest_bit % 8);
            dest[dest_bit / 8] &= ~dest_mask;
```

```
            auto src_mask = 1 << (src_bit % 8);
            auto corresponding_src_bit_set = src[src_bit / 8] & src_mask;
            if (corresponding_src_bit_set) {
                dest[dest_bit / 8] |= dest_mask;
            }
        }
    }
}
```

We copy the data one bit at a time. For each bit, we first clear that bit in the destination data. We then check if the corresponding bit in the source data is set and, if so, set the relevant bit in the destination data. There are more efficient ways in which we could implement this function, but this way is much simpler, and we don't expect copying DWARF expression piece results to be a very common operation.

Let's move on to reading global variables.

Indexing Global Variables

Currently, `sdb::dwarf` enables us to find functions with DIEs defined in that DWARF file. We index these functions so we don't need to traverse the entire DIE tree whenever we look for a function. Let's extend this index to also include global variables.

Add a global variable index to `sdb::dwarf` in *sdb/include/libsdb/dwarf.hpp*. Also add an `in_function` parameter to `index_die` to track whether the DIE currently being indexed is nested inside of a function and therefore not a global variable:

```
namespace sdb {
    class dwarf {
        --snip--
    private:
        --snip--
        void index_die(const die& current, bool in_function = false) const;
        --snip--
        mutable std::unordered_multimap<std::string, index_entry>
        global_variable_index_;
    };
}
```

We give the `in_function` parameter a default argument on `false` to minimize updates needed to existing calls. Update the implementation of `index_die` in *sdb/src/dwarf.cpp* to also add entries to the global variable index:

```
void sdb::dwarf::index_die(const die& current, bool in_function) const {
    --snip--
    if (has_range and is_function) {
        --snip--
    }
```

```
    auto has_location = current.contains(DW_AT_location);
    auto is_variable = current.abbrev_entry()->tag == DW_TAG_variable;
    if (has_location and is_variable and !in_function) {
        if (auto name = current.name()) {
            index_entry entry{ current.cu(), current.position() };
            global_variable_index_.emplace(*name, entry);
        }
    }

    if (is_function) in_function = true;
    for (auto child : current.children()) {
        index_die(child, in_function);
    }
}
```

If the DIE has both a `DW_AT_location` attribute and the `DW_TAG_variable` tag and isn't nested inside of a function DIE, this DIE belongs to a global variable we can locate. In that case, we attempt to get its name. If we succeed, we create a new index entry for that global variable and add it to the index. After handling variables, we update `in_function` if the current DIE belongs to a function. We then index all child DIEs, passing the new value of `in_function` so that any variable DIEs nested inside of function DIEs won't be added to the index.

Let's add a `find_global_variable` function to `sdb::dwarf` for querying this index. Declare this function in *sdb/include/libsdb/dwarf.hpp*:

```
namespace sdb {
    class dwarf {
    public:
        --snip--
        std::optional<die> find_global_variable(std::string name) const;
        --snip--
    };
}
```

Implement it in *sdb/src/dwarf.cpp*. For now, it will merely look at the global variable index:

```
std::optional<sdb::die> sdb::dwarf::find_global_variable(std::string name) const {
    index();
    auto it = global_variable_index_.find(name);
    if (it != global_variable_index_.end()) {
        cursor cur({ it->second.pos, it->second.cu->data().end() });
        return parse_die(*it->second.cu, cur);
    }
    return std::nullopt;
}
```

First, we call `index` to ensure that the index is ready (recall that the `index` function makes sure this check happens only once). We then search the global variable index for a variable with that name. If we don't find one, we return `std::nullopt`. If we do, we parse the DIE that starts at the location held in the index and return it. Note that, for simplicity, we don't handle namespaces, which mostly involves string processing and isn't super interesting. Feel free to implement this feature yourself, however.

Adding a Variable Lookup Interface

Let's add a very simple interface for variable reading to the command line driver. Add a new `variable` command to `handle_command`, in *sdb/tools/sdb.cpp*:

```
namespace {
    void handle_command(
          std::unique_ptr<sdb::target>& target,
          std::string_view line) {
        --snip--
        else if (is_prefix(command, "variable")) {
            handle_variable_command(*target, args);
        }
        --snip--
    }
}
```

For now, implement a basic version of `handle_variable_command`, which you'll extend in the next chapter. We'll support a single subcommand, `read <variable name>`, which reads the value of the variable with the given name as a 64-bit integer and prints it out:

```
namespace {
    void handle_variable_command(
          sdb::target& target, const std::vector<std::string>& args) {
        if (args.size() < 3) {
            print_help({ "help", "variable" });
            return;
        }

        if (is_prefix(args[1], "read")) {
            auto die = target.get_main_elf().get_dwarf().find_global_variable(args[2]);
            auto loc = die.value()[DW_AT_location].as_evaluated_location(
                target.get_process(), target.get_stack().current_frame().regs, false);
            auto value = target.read_location_data(loc, 8);
            std::uint64_t res = 0;
            std::copy(value.begin(), value.end(), reinterpret_cast<std::byte*>(&res));
            std::cout << "Value: " << res << '\n';
        }
```

```
    }
}
```

If the user doesn't supply enough arguments, we print a help message. Otherwise, we try to read the variable with the given name. We locate the DIE for it, evaluate its location, read the value, copy the result into a 64-bit integer, and print the value. This code is quite brittle, as we're assuming that the variable names a 64-bit integer, but it should do for some initial testing.

Extend the help information with this new command:

```
namespace {
    void print_help(const std::vector<std::string>& args) {
        if (args.size() == 1) {
            std::cerr << R"(Available commands:
    --snip--
    variable    - Commands for operating on variables
    --snip--
)";
        }
        else if (is_prefix(args[1], "variable")) {
            std::cerr << R"(Available commands:
    read <variable>
)";
        }
        --snip--
    }
}
```

Now we can move on to testing.

Testing

To test the new variable-reading feature, create a new test target at *sdb/test/targets/global_variable.cpp* with these contents:

```
#include <cstdint>

std::uint64_t g_int = 0;

int main() {
    g_int = 1;
    g_int = 42;
}
```

We make a single global variable called `g_int`, which we initialize to `0` and assign twice in the body of the `main` function. Add this target to *sdb/test/targets/CMakeLists.txt*:

```
--snip--
add_test_cpp_target(global_variable)
--snip--
```

Build the test and manually ensure that reading variables works. Set a breakpoint on `main`, continue the program, and then step through it, printing the value of `g_int` at each step. It should print `0`, then `1`, and then `42`:

```
$ tools/sdb test/targets/global_variable
Launched process with PID 1280
sdb> break set main
sdb> c
Thread 1280 stopped with signal TRAP at 0x555555555131,
global_variable.cpp:5 (global_variable`main)  (breakpoint 1)
  2 std::uint64_t g_int = 0;
  3
  4 int main() {
> 5     g_int = 1;
  6     g_int = 42;
  7 }
sdb> var read g_int
Value: 0
sdb> next
Thread 1280 stopped with signal TRAP at 0x55555555513b,
global_variable.cpp:6 (global_variable`main)  (single step)
   3
   4 int main() {
   5     g_int = 1;
>  6     g_int = 42;
   7 }
sdb> var read g_int
Value: 1
sdb> next
Thread 1280 stopped with signal TRAP at 0x555555555145,
global_variable.cpp:7 (global_variable`main)  (single step)
   4 int main() {
   5     g_int = 1;
   6     g_int = 42;
>  7 }
sdb> var read g_int
Value: 42
```

Let's translate this procedure into an automated test:

```
TEST_CASE("Can read global integer variable", "[variable]") {
    auto target = target::launch("targets/global_variable");
    auto& proc = target->get_process();

    target->create_function_breakpoint("main").enable();
    proc.resume();
    proc.wait_on_signal();

    auto var_die = target->get_main_elf().get_dwarf().find_global_variable("g_int");
    auto var_loc = var_die.value()[DW_AT_location]
        .as_evaluated_location(proc, proc.get_registers(), false);
    auto res = target->read_location_data(var_loc, 8);
    auto val = from_bytes<std::uint64_t>(res.data());

    REQUIRE(val == 0);

    target->step_over();
    res = target->read_location_data(var_loc, 8);
    val = from_bytes<std::uint64_t>(res.data());

    REQUIRE(val == 1);

    target->step_over();
    res = target->read_location_data(var_loc, 8);
    val = from_bytes<std::uint64_t>(res.data());

    REQUIRE(val == 42);
}
```

We launch *global_variable*, set a breakpoint on `main`, and resume the process. When the process hits the breakpoint, we find the DIE for the `g_int` variable, read its location attribute, get the data there, and convert it to a `std::uint64_t`. We require that the value be `0`, then we step, reread the variable, require that it be `1`, step again, reread, and require that it be `42`. We don't need to reevaluate the location DWARF expression because it's a global variable, so its position won't depend on the program counter or registers.

Finally, let's also add a test for a more complex DWARF expression:

```
TEST_CASE("DWARF expressions work", "[dwarf]") {
    std::vector<std::uint8_t> piece_data = {
        DW_OP_reg16, DW_OP_piece, 4, DW_OP_piece, 8, DW_OP_const4u,
        0xff, 0xff, 0xff, 0xff, DW_OP_bit_piece, 5, 12
    };

    auto target = target::launch("targets/step");
    auto& proc = target->get_process();
```

```
    sdb::span<const std::byte> data {
        reinterpret_cast<std::byte*>(piece_data.data()), piece_data.size() };
    auto expr = sdb::dwarf_expression(target->get_main_elf().get_dwarf(), data, false);
    auto res = expr.eval(proc, proc.get_registers());

    auto& pieces = std::get<sdb::dwarf_expression::pieces_result>(res).pieces;
    REQUIRE(pieces.size() == 3);
    REQUIRE(pieces[0].bit_size == 4 * 8);
    REQUIRE(pieces[1].bit_size == 8 * 8);
    REQUIRE(pieces[2].bit_size == 5);
    REQUIRE(std::get<dwarf_expression::register_result>(pieces[0].location).reg_num == 16);
    REQUIRE(std::get_if<dwarf_expression::empty_result>(&pieces[1].location) != nullptr);
    REQUIRE(std::get<dwarf_expression::address_result>(pieces[2].location)
        .address.addr() == 0xffffffff);
    REQUIRE(pieces[0].offset == 0);
    REQUIRE(pieces[1].offset == 0);
    REQUIRE(pieces[2].offset == 12);
}
```

We create a DWARF expression manually that consists of three pieces. The first piece is 4 bytes long and lives in register 16. The second is 8 bytes long and has no location. The third is 12 bits long and starts at a 5-bit offset inside of the memory at address `0xffffffff`. We create a dummy target, as our parser requires one, then make an `sdb::dwarf_expression` out of the data we created and evaluate it. We issue a series of requirements on the result: that there be three pieces with the sizes, contents, and offsets we specified. These tests should pass.

Summary

In this chapter, you implemented an interpreter for DWARF expressions and added basic support for reading global variables to your debugger. You learned about the taxonomy of DWARF expressions, how the stack machine is encoded, and how call frame information uses DWARF expressions. You also added support for DWARF expressions to your existing stack unwinder.

In the next chapter, you'll add type information parsing to your DWARF parser and use the parser to read, write, and visualize more complex variables.

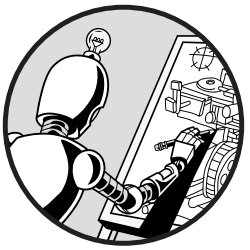
Check Your Knowledge

1. What different types of location descriptions can DWARF expressions encode?
2. With which other DWARF entity do location lists share an almost identical encoding?
3. Which DWARF expression instruction allows copying data from an arbitrary location of the stack?
4. Which DWARF expression instruction is not implementable as it is specified?

20

VARIABLES AND TYPES

Change comes
whether we will it
or not.



One of the most common operations programmers carry out when debugging their programs is printing out the values of variables, which can reveal how the data in a program mutates over the course of its execution. Now that the debugger supports DWARF expressions, you can implement variable printing in this chapter.

Currently, the debugger can print only global variables that are 64-bit unsigned integers. You'll extend this support to both global and local variables of almost any type by parsing the type information stored in DWARF. You'll support the visualization of complex types and add support for visualizing subobjects and array elements, like `cats[1].name`.

I won't walk through the steps required to implement writing values to variables. The process is very similar to the one for reading variables, except reversed, but implementing it would add many more pages of code. I also won't walk through reading from thread-local storage, but I'll outline roughly how thread-local storage works in Chapter 22.

Type DIEs

DWARF information expresses types as DIEs, and different DIE tags apply to different categories of types:

`DW_TAG_base_type` Used for types built into the language, like `int`, `float`, and `char`

`DW_TAG_unspecified_type` Defined very vaguely in the DWARF specification but used in C++ for types like `auto`, `decltype(auto)`, and `decltype(nullptr)` that do not name concrete types

`DW_TAG_const_type` A `const`-qualified type

`DW_TAG_pointer_type` A pointer type

`DW_TAG_ptr_to_member_type` A C++ pointer-to-member type, explained in more detail later in this section

`DW_TAG_reference_type` An lvalue reference type (`T&`)

`DW_TAG_rvalue_reference_type` An rvalue reference type (`T&&`)

`DW_TAG_volatile_type` A `volatile`-qualified type

`DW_TAG_typedef` A `typedef` or type alias (for example, `using T = int;`)

`DW_TAG_array_type` A potentially multidimensional array type

`DW_TAG_class_type` A class defined with the `class` keyword

`DW_TAG_structure_type` A class defined with the `struct` keyword

`DW_TAG_union_type` A `union` type

DWARF uses several other tags for specific languages, but these aren't relevant to C++.

Type DIEs typically refer to one another through their `DW_AT_type` attributes. For example, DWARF information could represent the C++ declaration `const int* const i;` with the following set of DIEs:

```
<0x0000002e>    DW_TAG_variable
                    DW_AT_name                  i
                    DW_AT_type                  <0x00000042>
<0x00000042>    DW_TAG_const_type
                    DW_AT_type                  <0x00000047>
<0x00000047>    DW_TAG_pointer_type
                    DW_AT_byte_size             0x00000008
                    DW_AT_type                  <0x0000004d>
<0x0000004d>    DW_TAG_const_type
                    DW_AT_type                  <0x00000053>
<0x00000053>    DW_TAG_base_type
                    DW_AT_byte_size             0x00000004
                    DW_AT_encoding              DW_ATE_signed
                    DW_AT_name                  int
```

The DIE for the `i` variable points to the DIE at `0x42`, which is the first `DW_TAG_const_type` DIE. This in turn references the DIE for the pointer type, which references the second `DW_TAG_const_type` DIE, which finally references the `int` base type DIE.

Base Types

Base types have a name, a size (encoded as either a `DW_AT_byte_size` or a `DW_AT_bit_size` attribute), and an encoding. These DIES use several types of encodings:

`DW_ATE_boolean` A Boolean type

`DW_ATE_float` A floating-point type (including `double` and `long float`)

`DW_ATE_signed` A signed integer type

`DW_ATE_signed_char` A signed character type; on x64, includes both `char` and `signed char`, although they're distinct types

`DW_ATE_unsigned` An unsigned integer type

`DW_ATE_unsigned_char` An unsigned character type

`DW_ATE_UTF` A Unicode character type; in C++17, used for the `char16_t` and `char32_t` types

For a concrete example, take a look at the DIE describing the `int` type in the previously shown DWARF information. As with DIE tags, DWARF has several other base type encodings that aren't used in C++.

Array Types

Array DIEs have a `DW_AT_type` attribute that gives the type of the array's elements and child DIEs of type `DW_TAG_subrange_type` that specify the bounds of each dimension of the array. The first child specifies the bounds of the first dimension, the second child specifies those for the second dimension, and so on. These children have `DW_AT_upper_bound` attributes that give the highest valid index of the dimension (in other words, one less than the size of the dimension). For example, the declaration `int my_ints[3][5];` would result in a DIE subtree that looks like this:

```
DW_TAG_array_type
    DW_AT_type      (0x0000004b "int")

    DW_TAG_subrange_type
        DW_AT_type    (0x00000044 "long unsigned int")
        DW_AT_upper_bound     (0x02)

    DW_TAG_subrange_type
        DW_AT_type    (0x00000044 "long unsigned int")
        DW_AT_upper_bound     (0x04)
```

The element type is `int`, so the array type's `DW_AT_type` attribute points to the DIE for the `int` base type. The DIE has two children that give the upper bounds of the two dimensions of the array. Array indices in C++ are of type `std::size_t`, which for x64 is `long unsigned int`, so the subrange DIEs have `DW_AT_type` attributes that point to the DIE for the `long unsigned int` type.

Class and Union Types

DWARF describes members of class and union types with `DW_TAG_member` child DIEs. These DIEs usually have a `DW_AT_name` attribute that holds the member's name, a `DW_AT_type` attribute that references the type DIE for the member, and a `DW_AT_data_member_location` attribute that gives the byte offset into the enclosing object at which the member lives. For example, consider the following type definition:

```
struct cat {
    const char* name;
    int age;
};
```

The type DIE for this type looks like this:

```
DW_TAG_structure_type
    DW_AT_name      ("cat")
    DW_AT_byte_size (0x10)

    DW_TAG_member
        DW_AT_name    ("name")
        DW_AT_type    (0x00000072 "const char *")
        DW_AT_data_member_location    (0x00)
    DW_TAG_member
        DW_AT_name    ("age")
        DW_AT_type    (0x00000043 "int")
        DW_AT_data_member_location    (0x08)
```

The `name` member is the first member of the object and as such lives at a byte offset of 0. The `name` member has a size of 8 bytes; therefore, the `age` member starts at an offset of 8. The `DW_AT_data_member_location` attribute takes into account any padding within the object, so it's always more accurate to reference this attribute rather than simply adding up the sizes of previous members.

Bitfields are a bit more complicated due to some historical issues. A *bitfield* is a C++ feature that allows integral types to take up a number of bits not aligned on a byte boundary. They're typically used to pack data very tightly to improve cache performance, or to interface with hardware that requires packing data together. Consider the following modifications to the previously shown type:

```
#include <cstdint>

struct cat {
    const char* name;
    std::uint8_t age : 5;
    std::uint8_t color : 3;
};
```

Here, the `age` and `color` members take up 5 and 3 bits, respectively. There are two ways in which DWARF information could encode this fact, one of which is deprecated in DWARF 4, even though compilers still sometimes output it.

The deprecated way to store this information is with the `DW_AT_bit_offset` attribute. Here is a stripped-down example:

```
DW_TAG_structure_type
  DW_AT_name                  cat
  DW_AT_byte_size             0x00000010
  DW_TAG_member
    DW_AT_name                  name
    DW_AT_data_member_location  0
  DW_TAG_member
    DW_AT_name                  age
    DW_AT_byte_size             0x00000001
    DW_AT_bit_size              0x00000005
    DW_AT_bit_offset            0x00000003
    DW_AT_data_member_location  8
  DW_TAG_member
    DW_AT_name                  color
    DW_AT_byte_size             0x00000001
    DW_AT_bit_size              0x00000003
    DW_AT_bit_offset            0x00000000
    DW_AT_data_member_location  8
```

The `DW_AT_bit_offset` attribute stores the distance in bits between the most significant bit of the aligned storage (whose size is given by the `DW_AT_byte_size` attribute) for the member and the most significant bit of the real data. This is awkward for little-endian machines because the attribute expresses the distance from the end of the storage to the end of the data, and you'll need to perform additional calculations to find the start of the data.

The preferred method of describing bitfields is with the `DW_AT_data_bit_offset` member:

```
DW_TAG_structure_type
  DW_AT_name                  cat
  DW_AT_byte_size             0x00000010
  DW_TAG_member
    DW_AT_name                  name
```

```
    DW_AT_data_member_location  0
  DW_TAG_member
    DW_AT_name                  age
    DW_AT_bit_size              0x00000005
    DW_AT_data_bit_offset       64
  DW_TAG_member
    DW_AT_name                  color
    DW_AT_bit_size              0x00000003
    DW_AT_data_bit_offset       69
```

The `DW_AT_data_bit_offset` attribute stores the distance in bits from the start of the enclosing type to the start of the data. We'll need to handle both of these methods of encoding bitfields.

Member Pointer Types

An infrequently used feature of C++, *member pointers* let you refer indirectly to the member data or member functions of a type. To demonstrate how they work (and so we have a program to use for testing later), create a file at *sdb/test/targets/member_pointer.cpp* with the following contents:

```
#include <iostream>

struct cat {
    const char* name;
    void meow() const { std::cout << "meow\n"; }
};
int main() {
    const char* (cat::*data_ptr) = &cat::name;
    void (cat::*func_ptr)() const = &cat::meow;

    cat marshmallow{ "Marshmallow" };

    auto name = marshmallow.*data_ptr;
    (marshmallow.*func_ptr)();
}
```

We define a `cat` type that has a `name` data member and a `meow` member function. At the start of `main`, we define a member pointer called `data_ptr` that can point to any data member of `cat` whose type is `const char*` and initialize it to point to `cat::name`. (Yes, the syntax is very strange.) We similarly define a member function pointer called `func_ptr` that can point to any member function of `cat` so long as it takes no arguments and returns `void`. We initialize the pointer to `cat::meow`.

Importantly, member pointers don't point to members of a specific instance of the class; they must be applied to a class instance in order to extract the instance's relevant member. To demonstrate this, we create a `cat` called Marshmallow, retrieve its name using the `data_ptr` member pointer, then call `meow` using the `func_ptr` member function pointer.

The utility of these member pointers may not be immediately obvious, but they can be very handy when dealing with higher-order functions. For example, C++20 ranges enable you to retrieve a range representing the names of all the cats in a `cats` range by writing `std::views::transform(cats, &cat::name)`.

Add *member_pointer* to *sdb/test/targets/CMakeLists.txt*:

```
--snip--
add_test_cpp_target(member_pointer)
--snip--
```

Now we can design a type to store information about the target program's types.

Storing Type Information

Let's create an `sdb::type` class that we'll interact with to visualize type data and retrieve information about types in the target program. Put this code in a new file called *sdb/include/libsdb/type.hpp*:

```
#ifndef SDB_TYPE_HPP
#define SDB_TYPE_HPP

#include <string_view>
#include <optional>
#include <libsdb/dwarf.hpp>

namespace sdb {
    class type {
    public:
        type(die die)
            : die_(std::move(die)) {}

        die get_die() const { return die_; }
        std::size_t byte_size() const;
        bool is_char_type() const;

    private:
        std::size_t compute_byte_size() const;

        die die_;
        mutable std::optional<std::size_t> byte_size_;
    };
}

#endif
```

We write a constructor that takes the DIE for the type and stores it in a `die_` member. We also write a function to retrieve this DIE and declare a function for retrieving the byte size of the type. Computing this value may

require a fair bit of work, so we factor the real calculation into a `compute_byte_size` private member function, which we'll call the first time we call `byte_size`, to populate the `byte_size_` member. We declare this member as `mutable` because it's a cache variable that `const` member functions can modify. Finally, we declare a helper function for figuring out whether a type is a character-like type, `is_char_type`, that will make our lives easier later on.

We'll mostly retrieve types from `DW_AT_type` attributes, so let's add an `as_type` member function to `sdb::attr`. Declare it in *sdb/include/libsdb/dwarf.hpp*:

```
namespace sdb {
    class type;
    class attr {
    public:
        --snip--
        type as_type() const;
        --snip--
    };
}
```

We forward-declare `sdb::type` and then add an `as_type` member function that returns an instance of this type. Implement this function in *sdb/src/dwarf.cpp*:

```
#include <libsdb/type.hpp>

sdb::type sdb::attr::as_type() const {
    return sdb::type{ as_reference() };
}
```

DWARF encodes attributes of type `DW_AT_type` as references to the relevant type DIE, so we call `as_reference` to retrieve this DIE and then pass that to the constructor for `sdb::type`.

Now we can start implementing the `sdb::type` class's member functions, starting with `byte_size`. Create a new file at *sdb/src/type.cpp* with these contents:

```
#include <libsdb/type.hpp>

std::size_t sdb::type::byte_size() const {
    if (!byte_size_.has_value())
        byte_size_ = compute_byte_size();
    return *byte_size_;
}
```

We check whether we've already computed the byte size for this type. If we haven't, we compute it and cache the result. Finally, we return the cached result. Add this new file to *sdb/src/CMakeLists.txt*:

```
add_library(libsdb ... type.cpp)
```

Next, let's implement `compute_byte_size`, back in *sdb/src/type.cpp*:

```
std::size_t sdb::type::compute_byte_size() const {
    auto tag = die_.abbrev_entry()->tag;

    if (tag == DW_TAG_pointer_type) { ❶
        return 8;
    }
    if (tag == DW_TAG_ptr_to_member_type) { ❷
        auto member_type = die_[DW_AT_type].as_type();
        if (member_type.get_die().abbrev_entry()->tag == DW_TAG_subroutine_type) {
            return 16;
        }
        return 8;
    }
    if (tag == DW_TAG_array_type) { ❸
        auto value_size = die_[DW_AT_type].as_type().byte_size();
        for (auto& child : die_.children()) {
            if (child.abbrev_entry()->tag == DW_TAG_subrange_type) {
                value_size *= child[DW_AT_upper_bound].as_int() + 1;
            }
        }
        return value_size;
    }
    if (die_.contains(DW_AT_byte_size)) { ❹
        return die_[DW_AT_byte_size].as_int();
    }
    if (die_.contains(DW_AT_type)) { ❺
        return die_[DW_AT_type].as_type().byte_size();
    }

    return 0;
}
```

First, we extract the tag from the type's abbreviation entry. If the type is a pointer type ❶, we return `8`.

If the type is a pointer to a member ❷, the size depends on what type it points to. On Linux systems, pointers to member functions are actually twice as large as pointers to member data because they store an additional 8 bytes used to support multiple inheritance. You'll find this fact specified in the Itanium C++ ABI, which was originally designed for the discontinued Intel Itanium processors but is used for essentially all C++ implementations on Linux. As such, we check the type to which the member pointer points and return `16` if that type is a subroutine (function) type or `8` otherwise.

If the type is an array type ❸, the computation is more involved. Recall that array DIEs have children of type `DW_TAG_subrange_type` that specify the bounds of each dimension of the array as well as a `DW_AT_type` attribute that specifies the element type of the array. As such, we compute the byte size of an array type by first computing the byte size of the type DIE's `DW_AT_type`

attribute and then multiplying this size by the size of each dimension (which is 1 plus the value of the DW_AT_upper_bound attribute).

If the DIE is neither a pointer nor an array, we examine its other attributes. If the DIE has a DW_AT_byte_size attribute ❹, we use that. If it instead has a DW_AT_type attribute, we defer to the size of the type pointed to by that attribute ❺. Otherwise, we don't know the size of the type, so we return 0.

Before we implement is_char_type, we'll create a useful function template helper to use in a few places in this chapter and the next: strip<*Tags*>, for stripping layers of qualifiers, pointers, or reference types off of a type. For instance, we can use strip<DW_TAG_const_type, DW_TAG_volatile_type> for stripping away layers of const and volatile. This approach uses a couple of handy C++ features for minimizing code duplication: variadic templates and fold expressions. Define the helper inside sdb::type, in *sdb/include/libsdb/type.hpp*:

```
namespace sdb {
    class type {
    public:
        --snip--
     ❶ template<int... Tags>
        type strip() const {
            auto ret = *this;
            auto tag = ret.get_die().abbrev_entry()->tag;
         ❷ while (((tag == Tags) or ...)) {
                ret = ret.get_die()[DW_AT_type].as_type().get_die();
                tag = ret.get_die().abbrev_entry()->tag;
            }
            return ret;
        }
        --snip--
    };
}
```

This template parameter list ❶ indicates that the strip function template takes any number of integer template arguments. It stores the arguments in Tags, which we call a *parameter pack*. In the definition, we initialize ret to be the current type and retrieve its DIE's tag. We then loop, stripping away one DIE at a time by replacing ret with the DIE referred to by the DW_AT_type attribute.

The code inside the while condition ❷ is a *fold expression*, which runs some expression across every element in a parameter pack and collects the result using a binary operator. Note that the extra set of parentheses is required, as that's part of the syntax for fold expressions. In this case, we check each tag in the Tags parameter pack against the tag of the current DIE and combine the results with the or operator. The outcome will indicate whether any tag in the given set of template arguments matches the tag we're currently examining.

Using this function template, we can define the additional helper functions `strip_cv_typedef` for stripping just qualifiers and typedefs from type, `strip_cvref_typedef` for also stripping reference types, and `strip_all` for additionally stripping references and pointer types:

```
namespace sdb {
    class type {
    public:
        --snip--
        type strip_cv_typedef() const {
            return strip<DW_TAG_const_type,
                         DW_TAG_volatile_type,
                         DW_TAG_typedef>();
        }
        type strip_cvref_typedef() const {
            return strip<DW_TAG_const_type,
                         DW_TAG_volatile_type,
                         DW_TAG_typedef,
                         DW_TAG_reference_type,
                         DW_TAG_rvalue_reference_type>();
        }
        type strip_all() const {
            return strip<DW_TAG_const_type,
                         DW_TAG_volatile_type,
                         DW_TAG_typedef,
                         DW_TAG_reference_type,
                         DW_TAG_rvalue_reference_type,
                         DW_TAG_pointer_type>();
        }
        --snip--
    };
}
```

Next, let's implement `is_char_type` in *sdb/src/type.cpp*:

```
bool sdb::type::is_char_type() const {
    auto stripped = strip_cv_typedef().get_die();
    if (!stripped.contains(DW_AT_encoding)) return false;
    auto encoding = stripped[DW_AT_encoding].as_int();
    return stripped.abbrev_entry()->tag == DW_TAG_base_type and
        encoding == DW_ATE_signed_char or
        encoding == DW_ATE_unsigned_char;
}
```

The `is_char_type` function first strips the type of qualifiers and ensures that the type has a `DW_AT_encoding` attribute. If it's a base type and the encoding is either `DW_ATE_signed_char` or `DW_ATE_unsigned_char`, we return `true`.

This code completes the implementation of `sdb::type`.

Visualizing Typed Data

To visualize variables, we'll need to operate on values in addition to their types. For this purpose, we'll implement a type called `sdb::typed_data` that combines an `sdb::type` with a sequence of bytes and supports the operations we'll need. Define it in *sdb/include/libsdb/type.hpp*:

```
#include <vector>

namespace sdb {
    class process;
    class typed_data{
    public:
        typed_data(std::vector<std::byte> data, type value_type,
                   std::optional<virt_addr> address = std::nullopt)
            : data_(std::move(data)), type_(value_type), address_(address) {}

        const std::vector<std::byte>& data() const { return data_; }
        const std::byte* data_ptr() const { return data_.data(); }
        const type& value_type() const { return type_; }
        std::optional<virt_addr> address() const { return address_; }

        typed_data fixup_bitfield(
            const sdb::process& proc, const sdb::die& member_die) const;
        std::string visualize(const sdb::process& proc, int depth = 0) const;

    private:
        std::vector<std::byte> data_;
        type type_;
        std::optional<virt_addr> address_;
    };
}
```

We declare a `visualize` member function that returns a string representing the contents of an object of this type. The function takes a reference to an `sdb::process` (because it may need to read memory), a span representing the data for the object, and a visualization depth for visualizing nested objects such as classes. We declare members to hold the data and its type, and an optional member for storing the address of the data. The address will be useful for implementing expression evaluation, but not all data will have a virtual memory address, such as data that lives in a register.

We also define member functions for retrieving the members and a convenience function for retrieving a `const std::byte*` to the start of the data, which will simplify some of the visualization code. We initialize the member data in the constructor. We also declare a `fixup_bitfield` function, which we'll use to preprocess bitfield members before visualizing them.

Let's begin writing the type visualizer. We'll factor out most of this function into functions for dealing with different kinds of types:

```
std::string sdb::typed_data::visualize(
      const sdb::process& proc, int depth) const {
    auto die = type_.get_die();
    switch (die.abbrev_entry()->tag) {
    case DW_TAG_base_type:
        return visualize_base_type(*this);
    case DW_TAG_pointer_type:
        return visualize_pointer_type(proc, *this);
    case DW_TAG_ptr_to_member_type:
        return visualize_member_pointer_type(*this);
    case DW_TAG_array_type:
        return visualize_array_type(proc, *this);
    case DW_TAG_class_type:
    case DW_TAG_structure_type:
    case DW_TAG_union_type:
        return visualize_class_type(proc, *this, depth);
    case DW_TAG_enumeration_type:
    case DW_TAG_typedef:
    case DW_TAG_const_type:
    case DW_TAG_volatile_type:
        return sdb::typed_data{ data_, die[DW_AT_type].as_type() }
            .visualize(proc);
    default: sdb::error::send("Unsupported type");
    }
}
```

We switch on the tag of the type DIE and dispatch to a relevant handler function. We handle the class, structure, and union tags with a single visualizer, then handle the enumeration, typedef, const, and volatile tags by retrieving their `DW_AT_type` attribute and visualizing the underlying type.

Before we implement the specific visualizers, we'll perform a bunch of string formatting, so let's link `libsdb` against `fmtlib`, a library introduced in Chapter 6 that can help with this task. Edit the `target_link_libraries` command in *sdb/src/CMakeLists.txt* like so:

```
target_link_libraries(libsdb PRIVATE Zydis::Zydis fmt::fmt)
```

Now we can dive into the specific visualizers, starting with the easy ones and leaving the more difficult for the end.

Member Pointers

We'll start with the visualizer for member pointers. Add this code to *sdb/src/type.cpp*:

```
#include <fmt/format.h>

namespace {
```

```
    std::string visualize_member_pointer_type(const sdb::typed_data& data) {
        return fmt::format("0x{:x}",
            sdb::from_bytes<std::uintptr_t>(data.data_ptr()));
    }
}
```

For both member data pointers and member function pointers, we read the first 64 bits of the data and visualize them as a pointer. We ignore the extra data that member function pointers store for supporting multiple inheritance.

Pointers

Next, let's write the slightly more complex handler for regular pointer types. The additional complexity is that we should print out a string if the pointer is to a character type:

```
#include <libsdb/process.hpp>

namespace {
    std::string visualize_pointer_type(
          const sdb::process& proc, const sdb::typed_data& data) {
        auto ptr = sdb::from_bytes<std::uint64_t>(data.data_ptr());
        if (ptr == 0) return "0x0";
        if (data.value_type().get_die()[DW_AT_type].as_type().is_char_type()) {
            return fmt::format("\"{}\"", proc.read_string(sdb::virt_addr{ ptr }));
        }
        return fmt::format("0x{:x}", ptr);
    }
}
```

First, we interpret the data as a 64-bit integer. If the pointer value is 0, we return `0x0`. We then check whether the type to which this pointer points is a character type. If so, we read the string from the process using a function, `sdb::process::read_string`, that we'll implement next, and return it surrounded in quotes. If the pointer doesn't point to a character type, we just return a hexadecimal representation of the pointer.

The `visualize_pointer_type` function depends on `sdb::process::read_string`. Declare this function in *sdb/include/libsdb/process.hpp*:

```
namespace sdb {
    class process {
    public:
        --snip--
        std::string read_string(virt_addr address) const;
        --snip--
    };
}
```

The function takes a virtual address to read from and returns the null-terminated string that resides at that address. Implement it in *sdb/src/process.cpp*:

```
std::string sdb::process::read_string(virt_addr address) const {
    std::string ret;
    while (true) {
        auto data = read_memory(address, 1024);
        for (auto c : data) {
            if (c == std::byte{ 0 }) return ret;
            ret.push_back(static_cast<char>(c));
        }
    }
}
```

We read 1KiB of data at a time: an amount not too wasteful for smaller strings but not so small that reading large strings requires thousands of syscalls. Each time we read a new chunk of data, we loop over the characters, pushing them into the string until we reach a null byte. We don't push the null byte into the returned string because `std::string` handles the null terminator for us.

Classes

Next, let's handle class types. We want to produce a nested visualization of the object's members. For example, consider the following type and variable declarations:

```
struct cat {
    const char* name;
    int age;
};

struct person {
    const char* name;
    int age;
    cat pet;
};

cat marshmallow { "Marshmallow", 4 };
person sy { "Sy", 33, marshmallow };
```

We create types to represent cats, with names and ages, and people, with names, ages, and pets. We then create a cat called Marshmallow who is 4 years old and a person called Sy who is 33 years old and has Marshmallow as a pet.

The visualization for sy should look like this:

```
{
    name: "Sy"
    age: 33
    pet: {
        name: "Marshmallow"
        age: 4
    }
}
```

Nested class members are indented and enclosed in curly brackets. Implement the `visualize_class_type` function in *sdb/src/type.cpp*:

```
namespace {
    std::string visualize_class_type(
          const sdb::process& proc, const sdb::typed_data& data, int depth) {
       std::string ret = "{\n";
       for (auto& child : data.value_type().get_die().children()) {
           if (child.abbrev_entry()->tag == DW_TAG_member and
               child.contains(DW_AT_data_member_location) or
               child.contains(DW_AT_data_bit_offset)) {
               auto indent = std::string(depth + 1, '\t');
               auto byte_offset = child.contains(DW_AT_data_member_location) ?
                   child[DW_AT_data_member_location].as_int() :
                   child[DW_AT_data_bit_offset].as_int() / 8;
               auto pos = data.data_ptr() + byte_offset;
               auto subtype = child[DW_AT_type].as_type();
               std::vector<std::byte> member_data{
                   pos, pos + subtype.byte_size() };
               auto data = sdb::typed_data{ member_data, subtype }
                   .fixup_bitfield(proc, child);
               auto member_str = data.visualize(proc, depth + 1);
               auto name = child.name().value_or("<unnamed>");
               ret += fmt::format("{}{}: {}\n", indent, name, member_str);
           }
       }
       auto indent = std::string(depth, '\t');
       ret += indent + "}";
       return ret;
   }
}
```

We start the visualization with an open brace on a new line. We then loop over all children of the type's DIE that are members of the class (indicated by the `DW_TAG_member` tag) and aren't declared `static` (indicated by the presence of a `DW_AT_data_member_location` or `DW_AT_data_bit_offset` attribute).

We compute an indentation string for this member, which is a number of tab characters (`\t`) equal to the current nesting depth plus one. We

also compute the byte offset of the data for this member, found in either the `DW_AT_data_member_location` attribute or the `DW_AT_data_bit_offset` attribute (which we divide by eight to retrieve the offset in bytes). To compute the start of the member's data, we add this offset to the start of the enclosing object's data. We also retrieve the member's type from its `DW_AT_type` attribute.

Using the data start location and member type information, we compute a vector that contains the data for the member. If this member is a bitfield, we may need to shift the data and mask it into the correct position before visualization, which we do with the `fixup_bitfield` function we'll write shortly. We then visualize that data with the `visualize` function. We pass `depth + 1` as the depth parameter to increase the nesting level. Members don't necessarily have names (for example, anonymous union members), so we try to extract the name of the member and use `<unnamed>` as a default if the member doesn't have a name. We then append the new member information to the current visualization data. Finally, we add an indented close brace and return the computed visualization.

In this function, you could also choose to visualize base classes. To do so, you would loop over the non-static members of the type pointed to by this class's children of type `DW_TAG_inheritance`.

Let's move on to the `fixup_bitfield` function. To support its implementation, we'll add a `get_bitfield_information` member function to `sdb::die`. This member function will handle extracting the information we'll need to fix the data from the DIE. Declare it in *sdb/include/libsdb/dwarf.hpp*:

```
namespace sdb {
    class die {
    public:
        --snip--
        struct bitfield_information {
            std::uint64_t bit_size;
            std::uint64_t storage_byte_size;
            std::uint8_t bit_offset;
        };
        std::optional<bitfield_information> get_bitfield_information(
            std::uint64_t class_byte_size) const;
        --snip--
    };
}
```

We define a `bitfield_information` type to store the bit size of the data, the byte size of the enclosing storage, and the bit offset into that storage at which the data we need lives. The `get_bitfield_information` function takes the byte size of the class to which this field belongs (which we'll use as the storage byte size in some circumstances) and returns the bitfield information for this DIE, so long as it exists. Define the function in *sdb/src/dwarf.cpp*:

```
std::optional<sdb::die::bitfield_information> sdb::die::get_bitfield_information(
      std::uint64_t class_byte_size) const {
```

```
    if (!contains(DW_AT_bit_offset) and !contains(DW_AT_data_bit_offset)) {
        return std::nullopt;
    }
    auto bit_size = (*this)[DW_AT_bit_size].as_int();
    auto storage_byte_size = contains(DW_AT_byte_size) ?
        (*this)[DW_AT_byte_size].as_int() :
        class_byte_size;
    auto storage_bit_size = storage_byte_size * 8;
    std::uint8_t bit_offset = 0;
    if (contains(DW_AT_bit_offset)) {
        auto offset_field = (*this)[DW_AT_bit_offset].as_int();
        bit_offset = storage_bit_size - offset_field - bit_size;
    }
    if (contains(DW_AT_data_bit_offset)) {
        bit_offset = (*this)[DW_AT_data_bit_offset].as_int() % 8;
    }
    return bitfield_information{ bit_size, storage_byte_size, bit_offset };
}
```

If the DIE doesn't contain a `DW_AT_bit_offset` or `DW_AT_data_bit_offset` field, it doesn't represent a bitfield, so we return `std::nullopt`. Bitfields always have a `DW_AT_bit_size` field, so we retrieve the bit size of the data from there. The DIE may have a `DW_AT_byte_size` member giving the size of the aligned storage, but it also might not, in which case we use the size of the enclosing class.

Recall that the bit offset is encoded with either a `DW_AT_bit_offset` or a `DW_AT_data_bit_offset` attribute. In the case of the former, the value is the distance between the end of the aligned storage and the end of the data. We calculate the offset to the start of the data by subtracting the sum of the bit size and the distance to the end of the data from the storage's bit size.

If the offset is instead encoded with a `DW_AT_data_bit_offset` attribute, which gives the offset from the start of the enclosing object to the start of the data, the computation is simpler. Because we calculated the start byte of the data inside of `visualize_class_type` before calling this function and passed a data span starting there to `visualize_member`, we need only take the `DW_AT_data_bit_offset` modulo 8 to get the bit offset from the start of the `data` parameter. Finally, we return the data we computed.

With this support written, we can implement `fixup_bitfield`. This function should extract the data for the bitfield and return an `sdb::typed_data` object that stores it as if it were a regular integral type. We can use the `memcpy_bits` function that we wrote in the last chapter for this:

```
sdb::typed_data sdb::typed_data::fixup_bitfield(
      const sdb::process& proc,
      const sdb::die& member_die) const {
    auto stripped = type_.strip_cv_typedef();
    auto bitfield_info = member_die.get_bitfield_information(stripped.byte_size());
    if (bitfield_info) {
```

```
        auto [bit_size, storage_byte_size, bit_offset] = *bitfield_info;

        std::vector<std::byte> fixed_data;
        fixed_data.resize(storage_byte_size);

        auto dest = reinterpret_cast<std::uint8_t*>(fixed_data.data());
        auto src = reinterpret_cast<const std::uint8_t*>(data_.data());
        memcpy_bits(dest, 0, src, bit_offset, bit_size);

        return { fixed_data, type_ };
    }
    return *this;
}
```

We strip any qualifiers and typedefs from the member's type and then attempt to extract its bitfield information. If we get bitfield information, we perform the fixup by creating a vector to hold the extracted data and copying only the bits for the bitfield member into the start of that vector; then we return the fixed data. If the member isn't a bitfield, we just return a copy of the existing data and type.

Arrays

Let's move on to visualizing array types. We'll implement most of this work in a recursive `visualize_subrange` function, which we'll call from `visualize_array_type`:

```
namespace {
    std::string visualize_array_type(
          const sdb::process& proc, const sdb::typed_data& data) {
        std::vector<std::size_t> dimensions;
        for (auto& child : data.value_type().get_die().children()) {
            if (child.abbrev_entry()->tag == DW_TAG_subrange_type) {
                dimensions.push_back(child[DW_AT_upper_bound].as_int() + 1);
            }
        }
        std::reverse(dimensions.begin(), dimensions.end());
        auto value_type = data.value_type().get_die()[DW_AT_type].as_type();
        return visualize_subrange(proc, value_type, data.data(), dimensions);
    }
}
```

First, we collect a vector of the sizes of each dimension. Recall that the size of a dimension represented by a `DW_TAG_subrange_type` DIE is one higher than the DIE's `DW_AT_upper_bound` attribute value. We want to recursively process the dimensions of the array in the order in which they're defined. We'll do this in `visualize_subrange` by treating the back of the `dimensions` vector as the current dimension to process and popping that dimension from the back

of the vector before recursing. As such, we reverse the `dimensions` vector before passing it to `visualize_subrange` so that the first dimension is at the back of the vector.

Let's implement `visualize_subrange` now. This recursive function should visualize the dimension currently at the back of the `dimensions` vector. If the vector is empty, it should visualize an element of the array:

```
#include <numeric>

namespace {
    std::string visualize_subrange(
          const sdb::process& proc, const sdb::type& value_type,
          sdb::span<const std::byte> data, std::vector<std::size_t> dimensions) {
        if (dimensions.empty()) { ❶
            std::vector<std::byte> data_vec { data.begin(), data.end() };
            return sdb::typed_data{ std::move(data_vec), value_type }.visualize(proc);
        }

        std::string ret = "[";
        auto size = dimensions.back();
        dimensions.pop_back();
        auto sub_size = std::accumulate(
            dimensions.begin(), dimensions.end(),
            value_type.byte_size(), std::multiplies<>());
        for (std::size_t i = 0; i < size; ++i) { ❷
            sdb::span<const std::byte> subdata{ data.begin() + i * sub_size, data.end() };
            ret += visualize_subrange(proc, value_type, subdata, dimensions);

            if (i != size - 1) { ❸
                ret += ", ";
            }
        }
        return ret + "]";
    }
}
```

First, we handle the base case for when `dimensions` is empty ❶. In this case, we visualize the array element that is stored in the `data` parameter. Otherwise, we handle the recursive case, which visualizes one dimension of the array. We begin the visualization with a left square bracket. We then remove the back element of the `dimensions` vector, storing the current value aside. We loop over each element of the dimension, visualizing the element ❷.

If the array being visualized is multidimensional, the elements of this dimension could themselves be arrays, so we need to calculate the byte size of the element by multiplying together all of the remaining dimension sizes and then multiplying that by `value_size`. We use `std::accumulate` for this. We then compute a span that covers the data for the element. This span will

start at an offset from the start of the dimension's data given by multiplying the current element index by the size of an element.

We use `data.end()` to mark the end of the data; we'll likely be passing a larger span than necessary, but the visualizer will take only the data it needs. We then recursively call `visualize_subrange` with the `subdata` span and append its result to the current visualization data. If this isn't the last element ❸, we append a comma and space to the visualization data. Finally, outside of the loop, we append a right square bracket character and return the result.

Base Types

Let's move on to the final visualizer: the one for base types. Because we already handled bitfields when we implemented `visualize_member`, this function is reasonably straightforward. One restriction is that we won't implement support for the `DW_ATE_UTF` encoding, as this would add significant complexity and text encoding isn't the focus of this book:

```
namespace {
    std::string visualize_base_type(const sdb::typed_data& data) {
        auto& type = data.value_type();
        auto die = type.get_die();
        auto ptr = data.data_ptr();

        switch (die[DW_AT_encoding].as_int()) {
        case DW_ATE_boolean:
            return sdb::from_bytes<bool>(ptr) ? "true" : "false";
        case DW_ATE_float:
            if (die.name() == "float")
                return fmt::format("{}", sdb::from_bytes<float>(ptr));
            if (die.name() == "double")
                return fmt::format("{}", sdb::from_bytes<double>(ptr));
            if (die.name() == "long double")
                return fmt::format("{}", sdb::from_bytes<long double>(ptr));
            sdb::error::send("Unsupported floating point type");
        case DW_ATE_signed:
            switch (type.byte_size()) {
            case 1: return fmt::format("{}", sdb::from_bytes<std::int8_t>(ptr));
            case 2: return fmt::format("{}", sdb::from_bytes<std::int16_t>(ptr));
            case 4: return fmt::format("{}", sdb::from_bytes<std::int32_t>(ptr));
            case 8: return fmt::format("{}", sdb::from_bytes<std::int64_t>(ptr));
            default: sdb::error::send("Unsupported signed integer size");
            }
        case DW_ATE_unsigned:
            switch (type.byte_size()) {
            case 1: return fmt::format("{}", sdb::from_bytes<std::uint8_t>(ptr));
            case 2: return fmt::format("{}", sdb::from_bytes<std::uint16_t>(ptr));
            case 4: return fmt::format("{}", sdb::from_bytes<std::uint32_t>(ptr));
            case 8: return fmt::format("{}", sdb::from_bytes<std::uint64_t>(ptr));
```

```
            default: sdb::error::send("Unsupported unsigned integer size");
            }
        case DW_ATE_signed_char:
            return fmt::format("{}", sdb::from_bytes<signed char>(ptr));
        case DW_ATE_unsigned_char:
            return fmt::format("{}", sdb::from_bytes<unsigned char>(ptr));
        case DW_ATE_UTF:
            sdb::error::send("DW_ATE_UTF is not implemented");
        default:
            sdb::error::send("Unsupported encoding");
        }
    }
}
```

We begin by retrieving the type for the data, the type's DIE, and a pointer to the start of the object's data. We then switch over the type's encoding. For Boolean types, we print `true` if the value is `true` and `false` otherwise. For floating-point types, we check the name of the type and visualize the relevant type. For signed and unsigned integers, we switch over the byte size of the type and visualize the relevant type. For signed and unsigned `char` types, we visualize the matching type. Finally, we throw an exception for UTF code point types.

Finding Local Variables

Let's add the ability to find local variables to `libsdb`. DWARF expresses the debug information for local variables with DIEs of type `DW_TAG_variable` and `DW_TAG_formal_parameter`. (The second type applies to function parameters.) Additional scopes inside of functions (for example, a scope delimited by the braces of an `if` statement) use DIEs of type `DW_TAG_lexical_block`. To see these DIES in action, create a file at *sdb/test/targets/blocks.cpp* with the following contents:

```
#include <iostream>

int main(int argc, const char** argv) {
    int i = 1;
    std::cout << i;
    {
        int i = 2;
        std::cout << i;
        {
            int i = 3;
            std::cout << i;
        }
    }
}
```

This code declares two nested lexical scopes that redefine the variable `i`. Add it to *test/targets/CMakeLists.txt*, as we'll use this file for testing later:

```
--snip--
add_test_cpp_target(blocks)
```

Here is an example of abbreviated DWARF information for the code we just wrote:

```
DW_TAG_subprogram
  DW_AT_name                  main
  DW_AT_low_pc                0x00001129
  DW_AT_high_pc               <offset-from-lowpc> 43 <highpc: 0x00001154>
  DW_TAG_formal_parameter
    DW_AT_name                  argc
  DW_TAG_formal_parameter
    DW_AT_name                  argv
  DW_TAG_variable
    DW_AT_name                  i
  DW_TAG_lexical_block
    DW_AT_low_pc                0x00001146
    DW_AT_high_pc               <offset-from-lowpc> 7 <highpc: 0x0000114d>
    DW_TAG_variable
      DW_AT_name                  i
    DW_TAG_lexical_block
      DW_AT_low_pc                0x0000114d
      DW_AT_high_pc               <offset-from-lowpc> 7 <highpc: 0x00001154>
      DW_TAG_variable
        DW_AT_name                  i
```

The `DW_TAG_subprogram` DIE for the `main` function contains DIEs that represent parameters and local variables. Local variables declared in deeper scopes are children of the `DW_TAG_lexical_block` DIE that represents the scope in which they are declared.

As such, to locate a local variable with a given name, we can find the deepest scope that corresponds to the current program counter value and then walk outward to the main `DW_TAG_subprogram` DIE, checking all variables declared in those scopes until we find one that matches.

Let's add a function to `sdb::dwarf` called `scopes_at_address` that will return a vector of all the `DW_TAG_lexical_block` DIEs to which the given program counter value belongs, followed by the `DW_TAG_subprogram` DIE to which these scopes belong. We'll also add a member function called `find_local_variable` for finding local variables. Because the variables found will differ depending on the current execution location, this function should take a program counter value at which to perform the lookup. Make these changes in *sdb/include/libsdb/dwarf.hpp*:

```
namespace sdb {
    class dwarf {
```

```
    public:
        --snip--
        std::optional<die> find_local_variable(std::string name, file_addr pc) const;
        std::vector<die> scopes_at_address(file_addr address) const;
        --snip--
    };
}
```

Implement `scopes_at_address` in *sdb/src/dwarf.cpp*. We'll have it call a recursive `scopes_at_address_in_die` function, which we'll write next:

```
std::vector<sdb::die> sdb::dwarf::scopes_at_address(file_addr address) const {
    auto func = function_containing_address(address);
    if (!func) return {};

    std::vector<sdb::die> scopes;
    scopes_at_address_in_die(*func, address, scopes);
    scopes.push_back(*func);
    return scopes;
}
```

First, we try to locate the function that contains the given address. If we don't find one, we return an empty vector. We then define a vector to hold any scopes we find and call `scopes_at_address_in_die`, which will fill this vector with DIEs of type `DW_TAG_lexical_block`. Finally, we push the DIE representing the function to the scopes and return them.

Let's implement `scopes_at_address_in_die`:

```
namespace {
    void scopes_at_address_in_die(
          const sdb::die& die, sdb::file_addr address,
          std::vector<sdb::die>& scopes) {
        for (auto& c : die.children()) {
            if (c.contains_address(address)) {
                scopes_at_address_in_die(c, address, scopes);
                scopes.push_back(c);
            }
        }
    }
}
```

We loop over the children of the given DIE. If the child DIE contains the given address, we recursively call `scopes_at_address_in_die` and then push the child to the back of the vector. This way, the front of the vector will be the most deeply nested scope containing the given address.

Let's implement the `find_local_variable` member function:

```
std::optional<sdb::die> sdb::dwarf::find_local_variable(
      std::string name, file_addr pc) const {
    auto scopes = scopes_at_address(pc);
```

```
    for (auto& scope : scopes) {
        for (auto& child : scope.children()) {
            auto tag = child.abbrev_entry()->tag;
            if ((tag == DW_TAG_variable or
                tag == DW_TAG_formal_parameter) and
                child.name() == name) {
                return child;
            }
        }
    }
    return std::nullopt;
}
```

We first compute the list of scopes to which the program counter belongs. We then loop through the child DIEs of each scope, and if we find one that declares a variable matching the given name, we return it. If we don't find a matching local variable, we return an empty optional.

Before we expose all of this new functionality to users on the command line, we need one last piece of support in `libsdb`: the ability to resolve member variable names and array elements.

Resolving Indirect Names

Users should be able to read variables like `marshmallow.age`, `cat_pointer->name`, or `my_cats[0].friend_cat->cuteness`. To support this feature, we'll add a member function to `sdb::target` that resolves indirect names like these, as well as a function called `find_variable` that finds a local variable in the current scope, if one exists, and otherwise tries to find a global variable with the given name. Declare these functions in *sdb/include/libsdb/target.hpp*:

```
namespace sdb {
    class typed_data;
    class target {
    public:
        --snip--
        typed_data resolve_indirect_name(std::string name, file_addr pc) const;
        std::optional<die> find_variable(std::string name, file_addr pc) const;
        --snip--
    };
}
```

The `resolve_indirect_name` function takes the full name of the data to read and the program counter value at which to resolve the name, and it returns the relevant type and value. In Chapter 21, we'll extend this function to handle member function names as well. Implement `find_variable` in *sdb/src/target.cpp*:

```
std::optional<sdb::die> sdb::target::find_variable(
      std::string name, sdb::file_addr pc) const {
```

```
    auto& dwarf = pc.elf_file()->get_dwarf();
    auto local = dwarf.find_local_variable(name, pc);
    if (local) return local;

    std::optional<die> global = std::nullopt;
    elves_.for_each([&](auto& elf) {
        auto& dwarf = elf.get_dwarf();
        auto found = dwarf.find_global_variable(name);
        if (found) {
            global = *found;
        }
    });
    return global;
}
```

We look for local variables in the DWARF for the ELF file that contains the current program counter value. If we find a local variable with the given name there, we return it. Otherwise, we look for global variables. These could be in any of the ELF files we've loaded, so we loop over all of them, searching for global variables with the given name. If we find one, we return it. Otherwise, we return `std::nullopt`.

Next, we'll implement `resolve_indirect_name` in the same file. This function should support `.`, `->`, and `[]` operators for accessing members and array elements (we won't support the `*` or `&` operators for dereferencing and taking addresses because sticking to just postfix operators simplifies the parsing significantly; users can always use `[0]` in place of the `*` operator, and we'll add a `variable location` command that they can use instead of the `&` operator):

```
#include <libsdb/type.hpp>
#include <libsdb/parse.hpp>

sdb::typed_data sdb::target::resolve_indirect_name(
      std::string name, sdb::file_addr pc) const {
    auto op_pos = name.find_first_of(".-[");

    auto var_name = name.substr(0, op_pos);
    auto& dwarf = pc.elf_file()->get_dwarf();

    auto data = get_initial_variable_data(*this, var_name, pc);

    while (op_pos != std::string::npos) {
     ❶ if (name[op_pos] == '-') {
            if (name[op_pos + 1] != '>') {
                sdb::error::send("Invalid operator");
            }
            data = data.deref_pointer(get_process());
            op_pos++;
```

```
        }
    ❷ if (name[op_pos] == '.' or name[op_pos] == '>') {
            auto member_name_start = op_pos + 1;
            op_pos = name.find_first_of(".-[", member_name_start);
            auto member_name = name.substr(
                member_name_start, op_pos - member_name_start);
            data = data.read_member(get_process(), member_name);
            name = name.substr(member_name_start);
        }
    ❸ else if (name[op_pos] == '[') {
            auto int_end = name.find(']', op_pos);
            auto index_str = name.substr(op_pos + 1, int_end - op_pos - 1);
            auto index = to_integral<std::size_t>(index_str);
            if (!index) {
                sdb::error::send("Invalid index");
            }
            data = data.index(get_process(), *index);
            name = name.substr(int_end + 1);
        }
        op_pos = name.find_first_of(".-[");
    }

    return data;
}
```

We begin by finding the first `.`, `-`, or `[` character in the string (or if there is none, `std::string::npos`), which marks the end of the name of the initial variable to read. Based on this index, we compute the name of this initial variable by taking a substring from the start of the string to the operator position. For example, in the name `marshmallow.age`, we'd retrieve `marshmallow`. Next, we get the DWARF file corresponding to the given program counter value and retrieve the initial variable name with a `get_initial_variable_data` function that we'll write soon.

After this initialization process, we loop, handling the different components of the name. If the operator we found starts with `-`, this must be a `->` operator ❶. We throw an exception if not and otherwise dereference the pointer value using an `sdb::typed_data::deref_pointer` function that we'll write shortly. The process for reading the named member of the dereferenced value is the same process we'll follow for the `.` operator, so we'll do that in a common branch. For now, we just increment `op_pos` so that the character directly to the right of `name[op_pos]` is the one that begins the name of the member variable.

If the operator is `.` or `->` ❷, we now need to read the named member variable. The start of this name is at the index directly after `op_pos`, so we create a `member_name_start` variable to hold it. We then reset `op_pos` to be the index of the next `.`, `-`, or `[` character in the string by providing `member_name_start` as the starting index to `find_first_of`. The member name is the substring between `member_name_start` and the index we just computed. For example, in

marshmallow.friend_cat->age, the name will be friend_cat on the first iteration and age on the second. We read the member using an sdb::typed_data::read_member function, which we'll write shortly, and then set name to be the substring starting at the member name we just read, to prepare for the next iteration of the loop.

The last operator we support is the [] operator ❸. If the first character we found was a left square bracket, we find the corresponding closing bracket. Between these two characters is the index to be read. We parse this index to an integer and throw an exception if the parsing fails. If the conversion was successful, we index the previous value using an sdb::typed_data::index function, which we'll write soon, and set name to begin after the closing square bracket character.

Before starting the next iteration of the loop, we set op_pos to be the start of the next operator (or std::string::npos if there is no subsequent operator). Finally, we return the data of the most recent value we read.

Let's implement get_initial_variable_data. This will locate the data for the first part of the indirect name. For example, it will return the data for marshmallow in the name marshmallow.friend_cat->age:

```
namespace {
    sdb::typed_data get_initial_variable_data(
          const sdb::target& target, std::string name, sdb::file_addr pc) {
        auto var = target.find_variable(name, pc);
        if (!var) {
            sdb::error::send("Variable not found");
        }
        auto var_type = var.value()[DW_AT_type].as_type();

        auto loc = var.value()[DW_AT_location].as_evaluated_location(
            target.get_process(), target.get_stack().current_frame().regs, false);
        auto data_vec = target.read_location_data(loc, var_type.byte_size());

        std::optional<sdb::virt_addr> address;
        if (auto single_loc = std::get_if<sdb::dwarf_expression::simple_location>(&loc)) {
            if (auto addr_res =
                  std::get_if<sdb::dwarf_expression::address_result>(single_loc)) {
                address = addr_res->address;
            }
        }
        return { std::move(data_vec), var_type, address };
    }
}
```

We find the DIE for the computed initial variable and retrieve its type. We then evaluate its location, retrieve its value, and retrieve its address, so long as lives at a single address. Finally, we return the data we just computed.

We call three functions in `resolve_indirect_name` that we need to add to `sdb::typed_data`. These are `deref_pointer`, `read_member`, and `index`. Declare them in *sdb/include/libsdb/type.hpp*:

```
namespace sdb {
    class typed_data {
    public:
        --snip--
        typed_data deref_pointer(
            const sdb::process& proc) const;
        typed_data read_member(
            const sdb::process& proc,
            std::string_view member_name) const;
        typed_data index(
            const sdb::process& proc,
            std::size_t index) const;
        --snip--
    };
}
```

Implement the functions in *sdb/src/type.cpp*, starting with `deref_pointer`. It should interpret the given data as a pointer and read the memory at the address given by the pointer:

```
sdb::typed_data sdb::typed_data::deref_pointer(
      const sdb::process& proc) const {
    auto stripped_type_die = type_.strip_cv_typedef().get_die();
    auto tag = stripped_type_die.abbrev_entry()->tag;
    if (tag != DW_TAG_pointer_type) {
        sdb::error::send("Not a pointer type");
    }
    sdb::virt_addr address{ sdb::from_bytes<std::uint64_t>(data_.data()) };
    auto value_type = stripped_type_die[DW_AT_type].as_type();
    auto data_vec = proc.read_memory(
        address, value_type.byte_size());
    return { std::move(data_vec), value_type, address };
}
```

The pointer type may be qualified with `const` or `volatile` qualifiers or have levels of typedefs, so we strip these before inspecting the type's tag. If it's not a pointer, we throw an exception. Otherwise, we convert the data to a virtual address, retrieve the type that the pointer points to, and read the memory at the relevant address. We read the number of bytes given by the value type's DIE and then return the data we read alongside the data's type and its address.

Let's move on to `read_member`. This function should find the DIE corresponding to the member with the given name and read the data at its location:

```
sdb::typed_data sdb::typed_data::read_member(
      const sdb::process& proc, std::string_view member_name) const {
    auto die = type_.get_die();
    auto children = die.children();
    auto it = std::find_if(children.begin(), children.end(),
        [&](auto& child) { return child.name().value_or("") == member_name; });
    if (it == children.end()) {
        sdb::error::send("No such member");
    }
    auto var = *it;
    auto value_type = var[DW_AT_type].as_type();

    auto byte_offset = var.contains(DW_AT_data_member_location) ?
        var[DW_AT_data_member_location].as_int() :
        var[DW_AT_data_bit_offset].as_int() / 8;
    auto data_start = data_.begin() + byte_offset;
    std::vector<std::byte> member_data{ data_start, data_start + value_type.byte_size() };

    auto data = address_ ?
        typed_data{ std::move(member_data), value_type, *address_ + byte_offset } :
        typed_data{ std::move(member_data), value_type };
    return data.fixup_bitfield(proc, var);
}
```

We retrieve the DIE for the type, and that DIE's children. We attempt to find a child DIE that has a name that matches the given member. If we don't find one, we throw an exception. Otherwise, this DIE represents a member of the type, so we extract this member's type and compute its byte offset.

The DIE could express the byte offset as a `DW_AT_data_member_location` attribute or use a `DW_AT_data_bit_offset` attribute to give the bit offset, which we divide by eight to retrieve the byte offset. We compute the start of the data by adding the byte offset to the beginning of the data range for the enclosing object.

Based on this start position and the member's byte size, we copy the member's data into a vector. We then create an `sdb::typed_data` object that represents this member. If the enclosing object has address information, we store updated address information in the resulting object; otherwise, we store no address information. Finally, we fix the data if the member is a bitfield.

The last member function we need to implement is `index`, which should support both pointer types and array types:

```
sdb::typed_data sdb::typed_data::index(
      const sdb::process& proc, std::size_t index) const {
    auto parent_type = type_.strip_cv_typedef().get_die();
    auto tag = parent_type.abbrev_entry()->tag;
    if (tag != DW_TAG_array_type and tag != DW_TAG_pointer_type) {
        sdb::error::send("Not an array or pointer type");
    }
    auto value_type = parent_type[DW_AT_type].as_type();
    auto element_size = value_type.byte_size();
    auto offset = index * element_size;
    if (tag == DW_TAG_pointer_type) {
        sdb::virt_addr address{ sdb::from_bytes<std::uint64_t>(data_.data()) };
        address += offset;
        auto data_vec = proc.read_memory(
            address, element_size);
        return { std::move(data_vec), value_type, address };
    }
    else {
        std::vector<std::byte> data_vec{
            data_.begin() + offset,
            data_.begin() + offset + element_size };
        if (address_) {
            return { std::move(data_vec), value_type, *address_ + offset };
        }
        return { std::move(data_vec), value_type };
    }
}
```

As in `deref_pointer`, we first strip any qualifiers and typedefs from the type. If the type isn't a pointer or an array type, we throw an exception. Next, we calculate the size of the values that are pointed to or stored in the array. We compute the offset of the desired value by multiplying this size by the given index. If the type is a pointer, we interpret the given data as a virtual address, add the computed offset to it, read the memory at that address, and return it. Otherwise, the type must be an array, so we copy the relevant bytes from the given data into a new vector. If this data has an address, we offset it and store it in the return value; otherwise, we just return the computed data and type.

We've finished adding support for variable visualization to `libsdb`. Now we can add support for it in the command line driver.

Exposing Variables to the User

The command line driver currently has a `variable read` command that supports only global integer variables. We'll extend it to support visualizing

variables of any type that are either local or global. We'll also add a `variable locals` command that prints all local variables in the current scope and a `variable location` command that prints the location of the given variable. We'll implement these in separate helper functions:

```
namespace {
    void handle_variable_command(
          sdb::target& target, const std::vector<std::string>& args) {
        if (args.size() < 2) {
            print_help({ "help", "variable" });
            return;
        }

        if (is_prefix(args[1], "locals")) {
            handle_variable_locals_command(target);
            return;
        }

        if (args.size() < 3) {
            print_help({ "help", "variable" });
            return;
        }

        if (is_prefix(args[1], "read")) {
            handle_variable_read_command(target, args);
        }
        else if (is_prefix(args[1], "location")) {
            handle_variable_location_command(target, args);
        }
    }
}
```

We check the number of arguments passed and print an error message if the user passes the wrong number. Depending on the subcommand, we dispatch to the relevant `handle_variable_*_command` function.

Let's implement `handle_variable_locals_command` first:

```
#include <libsdb/type.hpp>
#include <unordered_set>

namespace {
    void handle_variable_locals_command(sdb::target& target) {
        auto pc = target.get_pc_file_address();
        auto scopes = pc.elf_file()->get_dwarf().scopes_at_address(pc);
        std::unordered_set<std::string> seen;
        for (auto& scope : scopes) {
            for (auto& var : scope.children()) {
                std::string name(var.name().value_or(""));
```

```
                auto tag = var.abbrev_entry()->tag;
                if (tag == DW_TAG_variable or tag == DW_TAG_formal_parameter and
                    !name.empty() and !seen.count(name)) {
                    auto loc = var[DW_AT_location].as_evaluated_location(
                        target.get_process(), target.get_stack().current_frame().regs, false);
                    auto type = var[DW_AT_type].as_type();
                    auto value = target.read_location_data(loc, type.byte_size());
                    auto str = sdb::typed_data{ std::move(value), type }
                        .visualize(target.get_process());
                    fmt::print("{}: {}\n", name, str);
                    seen.insert(name);
                }
            }
        }
    }
}
```

We retrieve the current program counter and the list of scopes that contain that address. If any variables in nested scopes shadow variables in outer scopes, we should print out only the most nested variable. For this reason, we keep a `std::unordered_set` of the names of variables we've already seen. We then walk over all of the scopes that contain the program counter value, from the deepest nested to the outermost function DIE.

For each child DIE in that scope, we retrieve the DIE's name and tag. If the DIE represents a variable that has a name and that we haven't already seen, we compute its location, read the data at that location, retrieve a visualization of the data, and print out that visualization. Finally, we insert the name of the variable into the `seen` set so that we don't print out the values of any shadowed variables.

Next, let's write `handle_variable_read_command`:

```
namespace {
    void handle_variable_read_command(
          sdb::target& target, const std::vector<std::string>& args) {
        auto name = args[2];
        auto pc = target.get_pc_file_address();
        auto data = target.resolve_indirect_name(name, pc);
        auto str = data.visualize(target.get_process());
        fmt::print("Value: {}\n", str);
    }
}
```

We retrieve the name of the variable to read, get the current program counter value for the current thread, resolve the indirect name, retrieve a visualization of the resulting value, and then print out this visualization.

To finish the implementation, let's write `handle_variable_location_command`:

```
namespace {
    void handle_variable_location_command(
```

```
  sdb::target& target, const std::vector<std::string>& args) {
        auto name = args[2];
        auto pc = target.get_pc_file_address();
        auto var = target.find_variable(name, pc);
        if (!var) {
            std::cerr << "Variable not found\n";
            return;
        }

        auto loc = var.value()[DW_AT_location].as_evaluated_location(
            target.get_process(), target.get_stack().current_frame().regs, false);

        auto print_simple_location = [](auto* loc) {
            if (auto reg_loc = std::get_if<sdb::dwarf_expression::register_result>(loc)) {
                auto name = sdb::register_info_by_dwarf(reg_loc->reg_num).name;
                fmt::print("Register: {}\n", name);
            }
            else if (auto addr_res = std::get_if<sdb::dwarf_expression::address_result>(loc)) {
                fmt::print("Address: {:#x}\n", addr_res->address.addr());
            }
            else {
                fmt::print("None");
            }
        };

        if (auto simple_loc = std::get_if<sdb::dwarf_expression::simple_location>(&loc)) {
            print_simple_location(simple_loc);
        }
        else if (auto pieces_res = std::get_if<sdb::dwarf_expression::pieces_result>(&loc)) {
            for (auto& piece : pieces_res->pieces) {
                fmt::print("Piece: offset = {}, bit size = {}, location = ",
                    piece.offset, piece.bit_size);
                print_simple_location(&piece.location);
            }
        }
    }
}
```

We grab the name of the variable to search for and the current program counter then attempt to find the variable. If we don't find one, we print an error message and return. If we find one, we evaluate its location. We define a lambda function that prints a single location. If the variable resides in a register or at a memory address, we print information for the relevant register or address; otherwise, we print `None`. If the variable we found has a single location, we use the lambda we wrote to print information for it. If the variable instead consists of several pieces, we print information about each piece, alongside the usual information printed by the `print_single_location` lambda.

We're now done with the variable and type support. Before finishing the chapter, let's write a few tests.

Testing

First, we'll test global variables, arrays, pointers, and bitfields. Modify the existing *sdb/test/targets/global_variable.cpp* file like this:

```
#include <cstdint>

std::uint64_t g_int = 0;

int main() {
    g_int = 1;
    g_int = 42;
}

struct cat {
    const char* name;
    int age : 5;
    int color : 3;
};

struct person {
    const char* name;
    int age;
    cat* pets;
    int num_pets;
};

cat marshmallow { "Marshmallow", 4, 1 };
cat lexical_cat { "Lexical Cat", 8, 2 };
cat milkshake { "Milkshake", 4, 3 };
cat cats[] = { marshmallow, lexical_cat, milkshake };
person sy { "Sy", 33, cats, 3 };
```

After the `main` function, we've added several new types and global variables. The `cat` type has a C string member and two bitfield members. The `person` type has a string member, two `int` members, and a pointer member. We then define several global variables using these types, including an array.

You should now be able to build and launch this program, set a breakpoint on the `main` function, hit the breakpoint, and examine the values of the variables. Here is an example:

```
$ tools/sdb test/targets/global_variable
Launched process with PID 2759
sdb> break set main
sdb> c
```

```
Thread 3596 stopped with signal TRAP at 0x555555555131, global_variable.cpp:5
(global_variable`main)  (breakpoint 1)
  2 std::uint64_t g_int = 0;
  3
  4 int main() {
> 5     g_int = 1;
  6     g_int = 42;
  7 }
  8
  9 struct cat {
  10
sdb> var read sy.pets[0].name
Value: "Marshmallow"
sdb> var read cats
Value: [{
        name: "Marshmallow"
        age: 4
        color: 1
}, {
        name: "Lexical Cat"
        age: 8
        color: 2
}, {
        name: "Milkshake"
        age: 4
        color: 3
}]
```

Note that the variable values display as expected. Feel free to play around, printing the values of other variables to ensure that they work.

Let's write a short automated test for this program. Add a test case to *sdb/test/tests.cpp*:

```
#include <libsdb/type.hpp>

TEST_CASE("Global variables", "[variable]") {
    auto target = target::launch("targets/global_variable");
    auto& proc = target->get_process();

    target->create_function_breakpoint("main").enable();
    proc.resume();
    proc.wait_on_signal();

    auto name = target->resolve_indirect_name(
        "sy.pets[0].name", target->get_pc_file_address());
    auto name_vis = name.visualize(target->get_process());
    REQUIRE(name_vis == "\"Marshmallow\"");
```

```
        auto cats = target->resolve_indirect_name(
            "cats[1].age", target->get_pc_file_address());
        auto cats_vis = cats.visualize(target->get_process());
        REQUIRE(cats_vis == "8");
    }
```

We launch the *global_variable* program, set a breakpoint on the `main` function, and continue until it's hit. We then visualize `sy.pets[0].name` and `cats[1].age` and ensure that the values are as expected. This test should pass.

Let's also add tests for the *blocks* and *member_pointer* programs we implemented earlier, starting with the former:

```
TEST_CASE("Local variables", "[variable]") {
    auto dev_null = open("/dev/null", O_WRONLY);
    auto target = target::launch("targets/blocks", dev_null);
    auto& proc = target->get_process();

    target->create_function_breakpoint("main").enable();
    proc.resume();
    proc.wait_on_signal();
    target->step_over();

    auto var_data = target->resolve_indirect_name("i", target->get_pc_file_address());
    REQUIRE(from_bytes<std::uint32_t>(var_data.data_ptr()) == 1);

    target->step_over();
    target->step_over();

    var_data = target->resolve_indirect_name("i", target->get_pc_file_address());
    REQUIRE(from_bytes<std::uint32_t>(var_data.data_ptr()) == 2);

    target->step_over();
    target->step_over();

    var_data = target->resolve_indirect_name("i", target->get_pc_file_address());
    REQUIRE(from_bytes<std::uint32_t>(var_data.data_ptr()) == 3);
    close(dev_null);
}
```

We launch the *blocks* program, set a breakpoint on `main`, hit it, and step over one line to reach the first `std::cout` statement. At this point, the value of `i` should be `1`. We step over two more times to reach the second output statement and ensure that `i` is now `2`. Finally, we step over two more times to reach the final output statement and ensure the value is `3`. At each of these steps, the `resolve_indirect_name` function will find one of the three different DIEs that correspond to the three `i` variables, picking the correct one based on the current program counter value. This test should pass.

All that is left is to write a short test for the *member_pointer* program:

```
TEST_CASE("Member pointers", "[variable]") {
    auto target = target::launch("targets/member_pointer");
    auto& proc = target->get_process();
    target->create_line_breakpoint("member_pointer.cpp", 10).enable();
    proc.resume();
    proc.wait_on_signal();

    auto data_ptr = target->resolve_indirect_name(
        "data_ptr", target->get_pc_file_address());
    auto data_vis = data_ptr.visualize(proc);
    REQUIRE(data_vis == "0x0");

    auto func_ptr = target->resolve_indirect_name(
        "func_ptr", target->get_pc_file_address());
    auto func_vis = func_ptr.visualize(proc);
    REQUIRE(func_vis != "0x0");
}
```

We launch the *member_pointer* program, set a breakpoint on line 10 of *member_pointer.cpp* (the definition of the `marshmallow` variable), and hit it. We then visualize the `data_ptr` and `func_ptr` variables. The `data_ptr` variable should be `0x0`, because it should point to the first member of the `cat` type, which is at offset 0. The `func_ptr` variable should point to the `cat::meow` function. We don't have an easy way to look up the address of this function yet (we'll handle this task in the next chapter), so we just ensure that we don't receive a null pointer value.

That concludes the testing for the chapter. As always, feel free to test more rigorously if you like.

Summary

In this chapter, you extended the existing support for visualizing variables from 64-bit global integers to almost any kind of variable, global or local. In the next chapter, you'll implement a minimal expression evaluator that will allow users to call functions inside of the running program from the debugger.

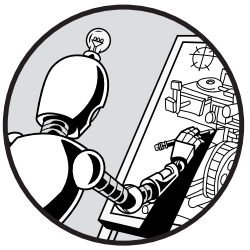
Check Your Knowledge

1. What is the difference between `DW_AT_bit_offset` and `DW_AT_data_bit_offset`?
2. How are the bounds of array types encoded?
3. Why are member function pointers larger in size than regular function pointers?

21

EXPRESSION EVALUATION

Calling out, willing
a reply
from the other side
of the world
with hands clasped, softly.



In this final coding chapter of the book, you'll implement one of the most powerful and complex features of debuggers: expression evaluation. *Expression evaluation* allows users to enter an expression in the programming language in which the debugged program was written and have it run in the process's current execution context.

For example, the user could enter `expr marshmallow.have_birthday()` to call the `have_birthday` member function on the `marshmallow` variable in the current scope, potentially modifying its state. If the variable had an `age` member, for example, this function would probably increase its value. In this chapter, you'll add support to your debugger for evaluating single function call expressions inside of the running process.

Supporting Expression Evaluation

Expression evaluation presents us with a problem: what kinds of expressions should we support? Production debuggers like GDB and LLDB try to support essentially every kind of C and C++ expression that we could write. Unfortunately, to do so, they have to implement a just-in-time C++ compiler. This isn't a book about compilers, so we'll limit ourselves to a very small subset of C++ expressions.

The most interesting part of expression evaluation is how the debugger can execute a function inside of the running process and extract the result, so we'll focus on this task. Here are some examples of expressions we'll support:

- Passing arguments to single functions, as in `feed(marshmallow)`
- Calling member functions, as in `marshmallow.have_birthday()`
- Passing string arguments, integers, floats, and Booleans, as in `get_cat("Marshmallow")`

We won't support implicit conversions of arguments. In other words, the argument types should match the parameter types exactly. We also won't support default arguments. We will, however, support calling overloaded functions. Here are other examples of expressions we won't support:

- Assignment, such as `my_cat = marshmallow`
- Arithmetic or Boolean expressions, such as `42 + 1 == 43`
- Nested function calls, such as `feed(get_cat("Marshmallow"))`
- Chained function calls, such as `get_cat("Marshmallow").have_birthday()`

By restricting the set of expressions the debugger can support, we can make it much more feasible to add expression evaluation support. Here are the main steps our expression evaluator needs to perform:

1. Parse the function name and arguments.
2. Locate the function to be called (potentially performing overload resolution).
3. Set up the stack and registers with the program arguments, as specified in the SYSV and Itanium ABIs.
4. Push the return address onto the stack (we'll discuss exactly where the function call should return to shortly).
5. Set a breakpoint on the return address.
6. Set the program counter to the start of the function that we want to call.
7. Continue the process until the return address breakpoint is hit.
8. Read the result of the function from the stack or registers, as specified in the SYSV and Itanium ABIs.

To parse the function name and arguments, we'll extend the function `sdb::target::resolve_indirect_name` that we wrote in the last chapter to also support function names. We'll locate the function to call and perform overload resolution using DWARF information.

The locations of arguments and return values specified in the ABI documents are part of the *calling conventions*, which are sets of rules that compilers must follow so that the code for function calls and the code for the callee can work together. These rules allow us to, say, compile a shared library with one compiler on one computer, compile an executable with a different compiler on a different computer, and have function calls across them work properly. The SYSV ABI specifies most of these rules, and you'll find some C++-specific details, like the rules for passing references, in the Itanium ABI.

Finding the return address requires a bit of thought, as we need to set a breakpoint on it to be notified when the function call terminates. You may consider using the current location of the program counter, but in the general case, this is problematic if the user calls the currently executing function by evaluating an expression, which might hit the internal breakpoint. We generally can't use the stack because it's often not marked as executable in the memory maps, so this would result in a segmentation violation. Another option is to call `mmap` inside the running process to allocate some memory that is definitely marked as executable and not used for anything else, but this punts the problem forward, as we need a return address to set a breakpoint on in order to call `mmap`.

Instead, we'll rely on the fact that, by the time the user is trying to evaluate an expression, the debugger should have already hit the internal entry point breakpoint we use for scanning shared libraries, and won't enter the entry point again. As such, we can use the address of the entry point as the return address and reuse the entry point breakpoint we've already set to be notified when the function call ends.

Extending Name Lookup

Currently, the debugger can look up the names of variables only, but we can modify the `sdb::target::resolve_indirect_name` function to support functions, too. First, we'll change the return type so it can return the data for a variable, the DIEs for matching functions, or both, in the case of member functions:

```
#include <libsdb/type.hpp>

namespace sdb {
    class target {
    public:
        --snip--
        struct resolve_indirect_name_result {
            std::optional<typed_data> variable;
            std::vector<die> funcs;
```

```
        };
        resolve_indirect_name_result resolve_indirect_name(
            std::string name, sdb::file_addr pc) const;
        --snip--
    };
}
```

We introduce a new type to store all of the data that needs returning and update the return type of `resolve_indirect_name` to use it. Next, we'll make the following changes to the implementation in *sdb/src/target.cpp*:

```
sdb::target::resolve_indirect_name_result
sdb::target::resolve_indirect_name(std::string name, sdb::file_addr pc) const {
    auto op_pos = name.find_first_of(".-[("); ❶

    if (name[op_pos] == '(') { ❷
        auto func_name = name.substr(0, op_pos);
        auto funcs = find_functions(func_name);
        return { std::nullopt, std::move(funcs.dwarf_functions) };
    }
    --snip--
    while (op_pos != std::string::npos) {
        --snip--
        if (name[op_pos] == '.' or name[op_pos] == '>') {
            auto member_name_start = op_pos + 1;
            op_pos = name.find_first_of(".-[(,", member_name_start); ❸
            auto member_name = name.substr(member_name_start, op_pos - member_name_start);
            if (name[op_pos] == '(') { ❹
                std::vector<die> funcs;
                auto stripped_value_type = data.value_type().strip_cvref_typedef();
                for (auto& child : stripped_value_type.get_die().children()) {
                    if (child.abbrev_entry()->tag == DW_TAG_subprogram and
                        child.contains(DW_AT_object_pointer) and
                        child.name() == member_name) {
                        funcs.push_back(child);
                    }
                }
                if (funcs.empty()) {
                    sdb::error::send("No such member function");
                }
                return { std::move(data), std::move(funcs) };
            }
            --snip--
        }
        else if (name[op_pos] == '[') {
            --snip--
        }
        op_pos = name.find_first_of(".-[(,"); ❺
```

```
    }

    return { std::move(data), {} }; ❻
}
```

We update the return type to match the declaration and add a `(` option to the calls to `find_first_of` ❶ ❸ ❺ and a `,` option to the last two. Then, we add a branch after the `op_pos` declaration that handles the case where the given name belongs to a non-member function ❷. This branch uses the existing `sdb::target::find_functions` function to locate any relevant DIEs and then returns them, using `std::nullopt` as the found variable, since there isn't one.

We also add a branch after the declaration of `member_name` that handles the case where the given name belongs to a member function ❹. This branch loops through the children of the DIE for the variable of which the function is a member and collects any DIEs that correspond to non-static member functions with the correct name. We locate non-static member functions using the `DW_AT_object_pointer` attribute, which identifies the parameter DIE corresponding to the `this` pointer. If we find no functions, we throw an exception. Otherwise, we return both the variable of which the function is a member and the list of functions we found with the correct name.

Finally, we update the final `return` statement for the case where the given name belongs to a variable and not to any function ❻.

Let's also add support for passing the results of previously evaluated expressions as arguments to other functions. For example, we should support uses like these:

```
sdb> expr get_cat("Marshmallow")
$0: {
        name: "Marshmallow"
        age: 4
}
sdb> expr get_age($0)
$1: 4
sdb> expr $0.give_command("get off the table")
Marshmallow, get off the table
```

We'll number the subsequent results of evaluated expressions and allow the user to pass these results, both as regular function arguments and as the object for a member function invocation. Later in this chapter, we'll add a `get_expression_result` function to `sdb::target` that will return an `sdb::typed_data` object representing the result of an expression with the given index. For now, we'll assume it exists and call it from the existing `get_initial_variable_data`. Update this function in *sdb/src/target.cpp*:

```
namespace {
    sdb::typed_data get_initial_variable_data(
          const sdb::target& target, std::string name, sdb::file_addr pc) {
        if (name[0] == '$') {
```

```
            auto index = sdb::to_integral<std::size_t>(name.substr(1));
            if (!index) {
                sdb::error::send("Invalid expression result index");
            }
            return target.get_expression_result(*index);
        }

        --snip--
    }
}
```

We introduce a new `if` block for when the name starts with a `$` character. If the name starts with a `$`, we parse the integer following this character, throw an exception if this parsing fails, and then return the expression result with the given index.

Because we altered the interface for `resolve_indirect_name`, we need to update its use in `handle_variable_command` in *sdb/tools/sdb.cpp*:

```
namespace {
    void handle_variable_read_command(
          sdb::target& target, const std::vector<std::string>& args) {
        --snip--
        auto str = data.variable->visualize(target.get_process());
        --snip--
    }
}
```

We change `data->visualize` to `data.variable->visualize`.

Parsing Arguments

We can now move on to parsing arguments into `sdb::typed_data` instances. We'll support the following types of arguments: strings, enclosed in `"` characters; characters, enclosed in `'` characters; Booleans (`true` or `false`); integers; floating-point numbers; variables (which can potentially be indirect, like `sy.pet`); and results of previously evaluated expressions, for which we added support in the previous section.

While turning the argument data into a vector of bytes is fairly straightforward, producing an `sdb::type` that represents the data's type is more complex, as `sdb::type` uses DIEs as the main source of information. We could search for a DIE that has the correct type; for example, if the user passes an integer, we could look for the DIE representing the `int` type and store this DIE as the data's type. But because we're not implementing implicit conversions like integral promotions, users would be able to pass these integers only to functions that take exactly the type `int`, not `long`, `unsigned int`, and so on. The story would be the same for `float` and `double`. As such, we'll extend `sdb::type` to represent some built-in types and handle those in a special way in the rest of the code.

Representing Built-in Types

Extend the definition of `sdb::type` in *sdb/include/libsdb/type.hpp* to represent built-in types:

```
#include <variant>

namespace sdb {
    enum class builtin_type {
        string, character, integer, boolean, floating_point
    };
    class process;
    class type {
    public:
        type(die die)
            : info_(std::move(die))
        --snip--
        type (builtin_type type)
            : info_(type) {}

        die get_die() const {
            if (!std::holds_alternative<die>(info_)) {
                sdb::error::send("Type is not from DWARF info");
            }
            return std::get<die>(info_);
        }

        builtin_type get_builtin_type() const {
            if (!std::holds_alternative<builtin_type>(info_)) {
                sdb::error::send("Type is not a builtin type");
            }
            return std::get<builtin_type>(info_);
        }

        bool is_from_dwarf() const { return std::holds_alternative<die>(info_); }

    private:
        --snip--
        std::variant<die, builtin_type> info_;
    };
}
```

Add a `builtin_type` enum to represent the various literal types we'll support. Remove the `die_` field and introduce an `info_` field that is a `std::variant` of an `sdb::die` or a `builtin_type`. Update the existing constructor to fill in the `info_` field and then add a new constructor that takes a `builtin_type` and stores it in the variant member. The existing constructor for DIEs will continue to work as is, so we don't need to modify it. Change the implementation of `get_die` to throw an exception if the variant doesn't currently hold an

sdb::die, and return it otherwise. Also add an equivalent function for retrieving the built-in type. Finally, add a member function that returns whether this type was created from a DIE.

Let's add support for built-in types to sdb::type::compute_byte_size in *sdb/src/type.cpp*. Make the following changes to the top of the function:

```
std::size_t sdb::type::compute_byte_size() const {
    if (!is_from_dwarf()) {
        switch (get_builtin_type()) {
        case builtin_type::boolean: return 1;
        case builtin_type::character: return 1;
        case builtin_type::integer: return 8;
        case builtin_type::floating_point: return 8;
        case builtin_type::string: return 8;
        }
    }

    auto& die_ = std::get<sdb::die>(info_);
    --snip--
}
```

If the type doesn't come from the DWARF information, it's a built-in type. We return 1 for Booleans and characters and 8 for integer types (meaning we default to long and double for integer and floating-point types, respectively). If the type does come from the DWARF information, we grab a reference to the DIE. I named it die_ with an underscore at the end so I wouldn't have to change the rest of the function's code, but you can give it a name without an underscore and perform the relevant search and replace if you like.

Let's also write to_byte_vec and to_byte_span functions, which will come in handy throughout this chapter for converting an object into a vector or span of its byte representation. Implement the functions in *sdb/include/bit.hpp*:

```
namespace sdb {
    template <class From>
    sdb::span<const std::byte> to_byte_span(const From& from) {
        return { as_bytes(from), sizeof(From) };
    }

    template <class From>
    std::vector<std::byte> to_byte_vec(const From& from) {
        std::vector<std::byte> ret(sizeof(From));
        std::memcpy(ret.data(), as_bytes(from), sizeof(From));
        return ret;
    }
}
```

The to_byte_span function returns a span of bytes from the start of the object to the end of the object. The to_byte_vec function copies the byte representation into a vector.

Parsing a Single Argument

With this support written, we can implement a `parse_argument` function that turns a string argument into an `sdb::typed_data` object. Implement it in *sdb/src/target.cpp*. It's rather long, so we'll implement it piece by piece:

```
namespace {
    sdb::typed_data parse_argument(
        sdb::target& target, pid_t tid, std::string_view arg) {
        if (arg.empty()) {
            sdb::error::send("Empty argument");
        }
        if (arg.size() > 2 and arg[0] == '"' and arg[arg.size() - 1] == '"') {
            auto ptr = target.inferior_malloc(arg.size() - 1);
            std::string arg_str{ arg.substr(1, arg.size() - 2) };
            auto data_ptr = reinterpret_cast<const std::byte*>(arg_str.data());
            sdb::span<const std::byte> data = {
                data_ptr, arg_str.size() + 1 };
            target.get_process().write_memory(ptr, data);
            return { sdb::to_byte_vec(ptr), sdb::builtin_type::string };
        }
```

If the argument is empty, we throw an exception. Otherwise, if the argument is enclosed in " characters, it's a string argument, and we need to copy this string into the memory of the running process. Before we perform this copy, we need to allocate enough memory in the running process to store the data. To do so, we'll write an `sdb::target::inferior_malloc` function later in this chapter that will call `malloc` inside of the running process and return the result. For now, we assume this function exists and call it with the size of the string. This is the size of the given argument minus one, because we don't include either quote character, but we do include the null terminator. After performing the allocation, we copy the data into the allocated memory and return an `sdb::typed_data` object with the resulting pointer value.

We then handle the other types:

```
        else if (arg == "true" or arg == "false") {
            auto value = arg == "true";
            return { sdb::to_byte_vec(value), sdb::builtin_type::boolean };
        }
        else if (arg[0] == '\'') {
            if (arg.size() != 3 or arg[2] != '\'') {
                sdb::error::send("Invalid character literal");
            }
            return { sdb::to_byte_vec(arg[1]), sdb::builtin_type::character };
        }
        else if (arg[0] == '-' or std::isdigit(arg[0])) {
            if (arg.find(".") != std::string::npos) {
                auto value = sdb::to_float<double>(arg);
                if (!value) {
```

```
                sdb::error::send("Invalid floating point literal");
            }
            return { sdb::to_byte_vec(*value), sdb::builtin_type::floating_point };
        }
        else {
            auto value = sdb::to_integral<std::int64_t>(arg);
            if (!value) {
                sdb::error::send("Invalid integer literal");
            }
            return { sdb::to_byte_vec(*value), sdb::builtin_type::integer };
        }
    }
    else {
        auto pc = target.get_pc_file_address(tid);
        auto res = target.resolve_indirect_name(std::string(arg), pc);
        if (!res.funcs.empty()) {
            sdb::error::send("Nested function calls not supported");
        }
        return *res.variable;
    }
  }
}
```

The other cases are simpler, as we don't have to allocate anything. If the argument is `true` or `false`, we return the value as a `builtin_type::boolean`. For character literals, we extract the middle character from between the quotes and return it with the `builtin_type::character` type, throwing an exception if the literal is invalid. If the argument begins with a - character or a digit, it's either a floating-point or an integer literal. If it contains a period, we attempt to parse it as a floating-point number. Otherwise, we interpret it as a signed integer. Finally, if it's none of these, it should name a variable, so we call `resolve_indirect_name` to find the variable, throw an exception if we get any function DIEs, and then return the found variable.

Collecting All Arguments

Now that we can parse a single argument, we can write a function to parse all arguments. This parsing should produce a `std::vector<sdb::typed_data>` holding all of the arguments to pass to the function. It also needs to deal with the implicit object argument (the `this` pointer) in cases where the user is trying to call a member function. According to the Itanium ABI, the implicit object argument gets passed by adding a pointer to the object to the front of the function's parameter list. Implement `collect_arguments` in *sdb/src/target.cpp*:

```
namespace {
    std::vector<sdb::typed_data> collect_arguments(
          sdb::target& target, pid_t tid, std::string_view arg_string,
```

```
      const std::vector<sdb::die>& funcs,
      std::optional<sdb::typed_data> object) {
    std::vector<sdb::typed_data> args;
    auto& proc = target.get_process();

    if (object) {
        std::vector<std::byte> data;
        if (object->address()) { ❶
            data = sdb::to_byte_vec(*object->address());
        }
        else { ❷
            auto& regs = proc.get_registers(tid);
            auto rsp = regs.read_by_id_as<std::uint64_t>(sdb::register_id::rsp);
            rsp -= object->value_type().byte_size();
            proc.write_memory(sdb::virt_addr{ rsp }, object->data());
            regs.write_by_id(sdb::register_id::rsp, rsp, true);
            data = sdb::to_byte_vec(rsp);
        }
        auto obj_ptr_die = funcs[0][DW_AT_object_pointer].as_reference(); ❸
        auto this_type = obj_ptr_die[DW_AT_type].as_type();
        args.push_back({ std::move(data), this_type });
    }

    auto args_start = 1; ❹
    auto args_end = arg_string.find(')');

    while (args_start < args_end) {
        auto comma_pos = arg_string.find(',', args_start);
        if (comma_pos == std::string::npos) {
            comma_pos = args_end;
        }
        auto arg_expr = arg_string.substr(args_start, comma_pos - args_start);
        args.push_back(parse_argument(target, tid, arg_expr));
        args_start = comma_pos + 1;
    }
    return args;
  }
}
```

We initialize the return data and grab a reference to the current process. If there is an implicit object argument to pass to a member function, we handle it. If we have an address for this variable ❶, our lives are easy; we store that pointer as the data for the argument. Otherwise, we spill the object to the stack ❷. To do this, we read the current stack pointer value and subtract the size of the object. (We subtract rather than add because the stack grows downward toward zero on x64.) We write the object's data into the new stack pointer position, update the stack pointer in the thread's registers, and store the new value of the stack pointer as the argument data.

With the data for the argument computed, we retrieve the DIE referenced by the function's `DW_AT_object_pointer` attribute ❸. This DIE should have the tag `DW_TAG_formal_parameter` with a `DW_AT_type` attribute that references the type of the `this` pointer. We can then create a new `typed_data` object to represent the `this` pointer and push it into the empty `args` vector.

After dealing with the implicit object argument, we handle all the other arguments. We expect `arg_string` to be a string containing all of the function arguments, including the opening and closing parentheses. So, the string index of the start of the first argument is 1, and the end position is the location of the closing parenthesis. We store these ❹ and then loop while there is still data to process. For each argument, we find the comma marking the end of the argument. If we can't locate one, we use the closing parenthesis index as the marker for the end of the argument instead. We extract the substring containing the next argument, pass that to the `parse_argument` function we wrote previously, and push the result into the `args` vector. Before the next loop iteration, we update `args_start` to be the first character after the comma. Finally, we return the computed arguments.

Overload Resolution

Now that we can find the set of functions the program might call and parse their arguments, we can move on to overload resolution. *Overload resolution* is the process of checking the arguments passed to a function against the parameters of many potential overloads and selecting the overload whose parameters match the arguments. Resolving overloads in exactly the same way as a C++ compiler is horrendously complicated; we'll implement a heavily simplified version that, while inaccurate and incomplete, will do the job in many cases.

To support the overload resolution, let's add a `parameter_types` function to `sdb::die` to make it easier to retrieve a list of a function DIE's parameter types. Declare it in *sdb/include/libsdb/dwarf.hpp*:

```
namespace sdb {
    class die {
    public:
        --snip--
        std::vector<type> parameter_types() const;
    };
}
```

Define the function in *sdb/src/dwarf.cpp*:

```
std::vector<sdb::type> sdb::die::parameter_types() const {
    std::vector<type> ret;
    if (!abbrev_->tag == DW_TAG_subprogram) return ret;
    for (auto& c : children()) {
        if (c.abbrev_entry()->tag == DW_TAG_formal_parameter) {
            ret.push_back(c[DW_AT_type].as_type());
```

```
        }
    }
    return ret;
}
```

If the DIE is not a subprogram, we return an empty vector. Otherwise, we loop over all of the children of the DIE and push back the types of any `DW_TAG_formal_parameter` DIEs we find. Finally, we return the computed vector.

Now we can implement `resolve_overload` in *sdb/src/target.cpp* to perform the heavily simplified overload resolution:

```
namespace {
    sdb::die resolve_overload(
          const std::vector<sdb::die>& funcs,
          const std::vector<sdb::typed_data>& args) {
        std::optional<sdb::die> matching_func;
        for (auto& func : funcs) {
            bool matches = true;
            auto arg_it = args.begin();
            auto params = func.parameter_types();

            if (args.size() == params.size()) {
                for (auto param_it = params.begin();
                    arg_it != args.end();
                    ++param_it, ++arg_it) {
                    if (*param_it != arg_it->value_type()) {
                        matches = false;
                        break;
                    }
                }
            }
            else {
                matches = false;
            }

         ❶ if (matches) {
                if (matching_func) sdb::error::send("Ambiguous function call");
                matching_func = func;
            }
        }
        if (!matching_func) sdb::error::send("No matching function");
        return *matching_func;
    }
}
```

We loop over the given candidates, looking for matches. If the number of arguments matches the number of parameters and each parameter type is equal to the value type of the corresponding argument, we consider it a

match. If we find a match ❶, we store it as the matched function. If we've already found a matching function, we instead consider the function call to be ambiguous and throw an exception. After examining all candidates, we throw an exception if we found no matching function. Otherwise, we return the match.

Our overload resolution implementation relies on equality checking for `sdb::type`. Let's implement that now. In *sdb/include/libsdb/type.hpp*, add a declaration for `operator==` and implement `operator!=` in terms of it:

```
namespace sdb {
    class type {
    public:
        --snip--
        bool operator==(const type& rhs) const;
        bool operator!=(const type& rhs) const {
            return !(*this == rhs);
        }
        --snip--
    }
}
```

The `operator!=` implementation negates the result of `operator==`.

Implement `operator==` in *sdb/src/type.cpp*. Because we're following a simplified approach, the result will be inaccurate in many cases. For instance, we won't take into account `const`ness, so we won't be able to call functions that are overloaded on `const`ness. We'll implement this function in pieces. First, we'll handle the case where both types are built in:

```
bool sdb::type::operator==(const type& rhs) const {
    if (!is_from_dwarf() and !rhs.is_from_dwarf()) {
        return get_builtin_type() == rhs.get_builtin_type();
    }
```

In this case, we compare the built-in types of each type for equality. Next, we'll handle the case where one type is built in and the other is not. We want to support both `a_builtin_type == a_dwarf_type` and `a_dwarf_type == a_builtin_type`, but we don't want to repeat a bunch of code for checking each side, so we'll first try to grab pointers to each type:

```
    const sdb::type* from_dwarf = nullptr;
    const sdb::type* builtin = nullptr;
    if (!is_from_dwarf()) {
        from_dwarf = &rhs;
        builtin = this;
    }
    else if (!rhs.is_from_dwarf()) {
        from_dwarf = this;
        builtin = &rhs;
    }
```

We assign the `from_dwarf` and `builtin` pointers to the left-hand side and right-hand side arguments, depending on which is which.

Next, we handle the type comparisons. For base DWARF types, we check the encoding against the built-in type. For pointer types, the only built-in type we support is a string, so we ensure that the DWARF pointer type points to a character type and that the built-in type is a string:

```
if (from_dwarf and builtin) {
    auto die = from_dwarf->strip_cvref_typedef().get_die();
    auto tag = die.abbrev_entry()->tag;
    if (tag == DW_TAG_base_type) {
        switch (die[DW_AT_encoding].as_int()) {
        case DW_ATE_boolean:
            return builtin->get_builtin_type() == builtin_type::boolean;
        case DW_ATE_float:
            return builtin->get_builtin_type() == builtin_type::floating_point;
        case DW_ATE_signed:
        case DW_ATE_unsigned:
            return builtin->get_builtin_type() == builtin_type::integer;
        case DW_ATE_signed_char:
        case DW_ATE_unsigned_char:
            return builtin->get_builtin_type() == builtin_type::character;
        default:
            return false;
        }
    }
    if (tag == DW_TAG_pointer_type) {
        return die[DW_AT_type].as_type().is_char_type() and
            builtin->get_builtin_type() == builtin_type::string;
    }
    return false;
}
```

Finally, we handle the case where both types come from DWARF. In this case, we'll remove qualifiers, references, typedefs, and pointers from both types and compare their names. If they're equal, we'll return `true`. Otherwise, we'll return `false`:

```
    auto lhs_stripped = strip_all();
    auto rhs_stripped = rhs.strip_all();

    auto lhs_name = lhs_stripped.get_die().name();
    auto rhs_name = rhs_stripped.get_die().name();
    if (lhs_name and rhs_name and *lhs_name == *rhs_name)
        return true;

    return false;
}
```

This will do for our needs. You can always augment this implementation yourself if you want to support more expression evaluation scenarios.

Calling Conventions

The functions we call expect any passed arguments to be in the positions specified by the calling conventions of the SYSV and Itanium ABIs. These rules are fairly complex because we need to be able to pass objects of any type, and, for performance reasons, we should use registers rather than RAM when possible. Before we can write code to evaluate expressions, we'll review these calling conventions.

Classifying Parameters

The SYSV ABI defines several parameter classes that indicate where to locate different arguments:

`INTEGER` Types passed and returned in general-purpose registers (GPRs)

`SSE` Types passed and returned in the lower 8 bytes of the `xmm` registers

`SSEUP` Types passed and returned in the upper bytes of the `xmm` registers

`X87` Types returned in the lower 8 bytes of the `st` registers

`X87UP` Types returned in the upper 2 bytes of the `st` registers

`COMPLEX_X87` Types returned in two `st` registers

`NO_CLASS` Used as a default class for padding, empty structures, and unions

`MEMORY` Types passed and returned on the stack

To locate both arguments and return values, we must sort types according to these classes. We classify all arguments in groups of 8 bytes, which the SYSV ABI imaginatively calls *eightbytes*. Here are the rules for classifying the basic types (all types other than class or array types):

- We assign pointers, as well as integer types that fit in an eightbyte, to the `INTEGER` class.
- We assign floating-point types that fit in an eightbyte, as well as `__m64` (a type used with some MMX funtions), to the `SSE` class.
- We split `__m128` (a type used with some XMM functions) and floating-point types that fit in 2 eightbytes into two halves, and assign the least significant half the `SSE` class and the most significant half the `SSEUP` class.
- We split `__m256` (a type used with some AVX functions) into 4 eightbytes and assign the least significant one to the `SSE` class and the others to the `SSEUP` class.

- We split `long double` into 2 eightbytes, assign the least significant 64 bits to the `X87` class, and assign the most significant 16 bits (plus 6 bytes of padding) to the `X87UP` class.
- We treat `__int128` as if it were composed of two 64-bit integers, with one exception: we must align arguments of type `__int128` that are stored in memory to 16 bytes rather than 8 bytes.
- We treat `std::complex<T>` in C++ or `complex T` in C99 as if it were composed of two objects of type `T`.
- We assign `complex long double` to the `COMPLEX_X87` class.

Some of these types, like `complex long double`, occur very rarely, and even GDB doesn't support them. For simplicity and parity with our existing register support, we won't support `__m64`, `__m128`, `__m256`, `__int128`, or `complex` types. We'll support everything else.

The classification of class and union types is significantly more complex and consists of multiple stages. We can't pass certain kinds of C++ types in registers because copying or destroying them requires additional steps. We refer to such types with the rather lengthy term *non-trivial for the purposes of calls (NTFPOC)*. As defined in the Itanium ABI, a type is NTFPOC if it has a non-trivial copy constructor, move constructor, or destructor, or if all of its copy and move constructors have been deleted.

A type has a non-trivial copy or move constructor or destructor in a few cases: if it's defined by the user rather than defined implicitly or defaulted on first declaration, if the type has any virtual members or virtual base classes, or if any members or base classes have a non-trivial copy or move constructor or destructor. Types that are non-trivial for the purposes of calls are passed by *invisible reference*: they are replaced in the argument list with a pointer to the object, which is given the class `INTEGER`.

NTFPOC is very close to the C++ concept of trivial copyability, but it isn't quite the same: trivial copyability also has constraints on assignment operators. We won't support passing NTFPOC types by value because this requires manually calling the copy constructor for each NTFPOC argument to a function and potentially manually calling the destructor as well. This is certainly possible to implement with the support that we'll write in this chapter, but it complicates the implementation even further, and this chapter is long and complex enough as it is.

If the type is larger than 8 eightbytes or contains fields that are not aligned to their expected boundaries, it is given the `MEMORY` class. Each eightbyte of the class is classified separately. Each field that belongs to a given eightbyte is classified and merged into a single class for that eightbyte with the following rules, which are run top to bottom (that is, as soon as one of these rules matches, that is the result of the merge):

- If both classes are the same, the result is this class.
- If one of the classes is `NO_CLASS`, the result is the other class.
- If one of the classes is `MEMORY`, the result is `MEMORY`.

- If one of the classes is INTEGER, the result is INTEGER.
- If one of the classes is X87, X87UP, or COMPLEX_X87, the result is MEMORY.
- Otherwise, the result is SSE.

After all of the eightbytes of the type are classified, there is a post-merger cleanup step that follows these rules, again run top to bottom:

- If one of the classes is MEMORY, we set all classes to MEMORY.
- If an X87UP class is not preceded by an X87 class, we set all classes to MEMORY.
- If the type is larger than 2 eightbytes and the first class is not SSE or any other class is not SSEUP, we set all classes to MEMORY. This is to support passing types with large vector members like __m256 in registers. Note that since we're not supporting passing vector types, we can consider all class types larger than 2 eightbytes as being passed in memory.
- If SSEUP is not preceded by SSE or SSEUP, it is changed to SSE.

Passing Arguments

To pass arguments to a function, we first classify all arguments from left to right using the algorithm you just read. Depending on the classes of each argument, that argument may be placed in memory or copied into registers. Each eightbyte of each argument is considered separately. The allocation algorithm is as follows:

- If the class is MEMORY, the eightbyte is allocated to the stack. Furthermore, if any eightbyte of an argument should be allocated to a register but no more are available, the entire argument is allocated to the stack.
- If the class is INTEGER, the eightbyte is allocated to the next available register of rdi, rsi, rdx, rcx, r8, and r9.
- If the class is SSE, the eightbyte is allocated to the next available register of the xmm registers, starting with xmm0 and proceeding through xmm7.
- If the class is SSEUP, the eightbyte is allocated to the next available eightbyte in the previously allocated xmm register.
- If the class is X87, X87UP, or X87_COMPLEX, the eightbyte is allocated to the stack; X87 objects are allocated to registers only when they are returned from a function.

Arguments that are allocated to the stack are pushed in reverse order: right to left. That is, the leftmost argument ends up at the top of the stack after all arguments are pushed. This is to simplify the implementation of varargs functions. If the function called is a varargs function, the al register must store the number of vector (SSE) registers allocated.

Returning Values

To determine the location of a function's return value, we classify the return type using the algorithm described earlier and consider each eightbyte of the type separately. Then, we execute the following algorithm:

- If the class is `MEMORY`, the caller must allocate storage for the return value and pass the address of this storage as if it were the first argument to the function (in the `rdi` register). This is usually done by allocating space on the caller's stack. On return, the `rax` register will store the address that was passed.
- If the class is `INTEGER`, the eightbyte is allocated to the next available register of `rax` and `rdx`.
- If the class is `SSE`, the eightbyte is allocated to the next available register of `xmm0` and `xmm1`.
- If the class is `SSEUP`, the eightbyte is allocated to the next available eightbyte in the previously allocated `xmm` register.
- If the class is `X87`, the eightbyte is allocated to the `st0` register.
- If the class is `X87UP`, the eightbyte is allocated to the higher-order bits of the `st0` register.
- If the class is `COMPLEX_X87`, the eightbyte representing the real part of the number is allocated to `st0` and the eightbyte representing the imaginary part of the number is allocated to `st1`.

Calling Functions in the Debugger

With the theory out of the way, we can start implementing the necessary support for calling functions in the debugger. We'll begin by creating a function that handles the low-level details of calling a function in the inferior process.

Performing Low-Level Inferior Calls

The first function we'll write won't deal with arguments or return values; it will merely set up the return address, jump to some address that marks the start of a function, and restore the old set of registers after the function completes. We'll call this function `sdb::process::inferior_call`. Declare it in *sdb/include/libsdb/process.hpp*:

```
namespace sdb {
    class process {
    public:
        --snip--
        sdb::registers inferior_call(
            sdb::virt_addr func_addr, sdb::virt_addr return_addr,
            const sdb::registers& regs_to_restore,
```

```
            std::optional<pid_t> otid = std::nullopt);
        --snip--
    };
}
```

The `inferior_call` function takes the address of a function to call, a return address (on which the caller should have set a breakpoint), a set of registers to restore when the function call completes, and an optional thread to run the function in. It returns the state of the registers after the function call completes, before they're restored to their old values.

Implement `inferior_call` in *sdb/src/process.cpp*:

```
sdb::registers sdb::process::inferior_call(
      sdb::virt_addr func_addr, sdb::virt_addr return_addr,
      const sdb::registers& regs_to_restore, std::optional<pid_t> otid) {
    auto tid = otid.value_or(current_thread_);
    auto& regs = get_registers(tid);

    regs.write_by_id(sdb::register_id::rip, func_addr.addr(), true);

    auto rsp = regs.read_by_id_as<std::uint64_t>(sdb::register_id::rsp);

    rsp -= 8;
    write_memory(
        sdb::virt_addr{ rsp }, sdb::to_byte_span(return_addr.addr()));
    regs.write_by_id(sdb::register_id::rsp, rsp, true);

    resume(tid);
    auto reason = wait_on_signal(tid);
    if (reason.reason != sdb::process_state::stopped) {
        sdb::error::send("Function call failed");
    }

    auto new_regs = regs;
    regs = regs_to_restore;
    regs.flush();
    if (target_) target_->notify_stop(reason);

    return new_regs;
}
```

We begin with the usual pattern: getting the thread ID to operate on, depending on what the user passed as an argument. For convenience, we also grab a reference to the registers for that thread. We set the program counter to point to the function we want to call. The return address should go right at the top of the stack, so we allocate 8 bytes on the stack, copy the return address into the allocated space, and update the stack pointer in the

inferior. This function assumes that the return address already has a breakpoint set on it, as handled by the caller, so we resume the thread, wait until it stops, and throw an exception if it exited. Finally, we restore the registers to their original values and notify the target that a stop occurred so that it can update its data structures. Note that `regs` is a reference, so assigning `regs_to_restore` to it performs the update to the registers for the current thread. Finally, we return the register state from after the call.

Allocating Memory for Arguments

Armed with `inferior_call`, we can now implement `sdb::inferior_malloc`, which we used when parsing string arguments. It takes a number of bytes to allocate and returns a virtual address that marks the beginning of the allocated region. We could also define an `inferior_free` function, but the areas we'll allocate should live for the lifetime of the program, so we won't bother freeing them. Declare `inferior_malloc` in *sdb/include/libsdb/target.hpp*:

```
namespace sdb {
    class target {
    public:
        --snip--
        virt_addr inferior_malloc(std::size_t size);
        --snip--
    };
}
```

Define `inferior_malloc` in *sdb/src/target.cpp*:

```
sdb::virt_addr sdb::target::inferior_malloc(std::size_t size) {
    auto saved_regs = process_->get_registers();

    auto malloc_funcs = find_functions("malloc").elf_functions;
    auto malloc_func = std::find_if(
        malloc_funcs.begin(), malloc_funcs.end(), [](auto& sym) {
            return sym.second->st_value != 0;
        });
    if (malloc_func == malloc_funcs.end()) {
        error::send("malloc not found");
    }

    file_addr malloc_addr{
        *malloc_func->first, malloc_func->second->st_value };
    auto call_addr = malloc_addr.to_virt_addr();

    auto entry_point = virt_addr{ process_->get_auxv()[AT_ENTRY] };
    breakpoints_.get_by_address(entry_point).install_hit_handler([&] {
        return false;
    });
```

```
    process_->get_registers().write_by_id(register_id::rdi, size, true);

    auto new_regs = process_->inferior_call(
        call_addr, entry_point, saved_regs);
    auto result = new_regs.read_by_id_as<std::uint64_t>(register_id::rax);

    return virt_addr{ result };
}
```

First, we save the current state of the registers so that `inferior_call` can restore them. We then find the definition of `malloc`, which lives inside the `libc` implementation. We'll assume the user is not using a debug build of `libc`, so `sdb::target::find_functions` will find it in the ELF symbol table rather than in the DWARF information. You can check both if you'd like to support debug builds of `libc`.

There are likely several symbols that match `malloc`, some of which will be undefined, so we look for the first that has a symbol value, which will correspond to the file address of `malloc` inside the `libc` shared library. If we don't find `malloc`, we throw an error; otherwise, we create a `file_addr` from the located symbol and convert it to a virtual address.

Recall from the start of the chapter that we'll use the entry point as the return address for calls. We locate the entry point breakpoint and replace its hit handler with one that returns `false` and therefore doesn't cause the process to immediately resume when the breakpoint is hit. We set up the argument for `malloc` by writing the number of bytes to allocate into `rdi`, following the calling conventions. We then call `inferior_call` and save the registers that resulted from the call to `malloc`. Recall that pointer return values usually get stored in the `rax` register. As such, we read `rax` from the returned registers and return it as a virtual address.

Classifying Parameters

Now that we've supported calling functions at a low level, we can think about the next level up: setting up arguments. To do this, we need to implement the algorithms for classifying arguments you learned about earlier in this chapter.

We'll create an enum for representing the argument classes, extend `sdb::type` with functions to get the parameter classes for the type, check whether it has unaligned fields, compute its expected alignment, and check whether the argument is NTFPOC. Make these changes in *sdb/include/libsdb/type.hpp*:

```
namespace sdb {
    enum class parameter_class {
        integer, sse, sseup, x87, x87up, complex_x87,
        memory, no_class
    };
```

```
    class type {
    public:
        --snip--
        std::size_t alignment() const;
        bool has_unaligned_fields() const;
        bool is_non_trivial_for_calls() const;
        std::array<parameter_class, 2> get_parameter_classes() const;
        --snip--
    };
}
```

The `sdb::parameter_class` enum corresponds to the set of classes you've already learned about. Because we won't support passing vector types like `__m256`, we only ever need to consider 2 eightbytes for any given type, so `get_parameter_classes` returns an array of two parameter classes, one for each eightbyte.

Getting Parameter Classes

Next, let's get the parameter class for the type. Implement `get_parameter_classes` in *sdb/src/type.cpp*:

```
std::array<sdb::parameter_class, 2> sdb::type::get_parameter_classes() const {
    std::array<parameter_class, 2> classes = { ❶
        parameter_class::no_class, parameter_class::no_class };

    if (!is_from_dwarf()) { ❷
        switch (get_builtin_type()) {
        case builtin_type::boolean: classes[0] = parameter_class::integer; break;
        case builtin_type::character: classes[0] = parameter_class::integer; break;
        case builtin_type::integer: classes[0] = parameter_class::integer; break;
        case builtin_type::floating_point: classes[0] = parameter_class::sse; break;
        case builtin_type::string: classes[0] = parameter_class::integer; break;
        }
        return classes;
    }

    auto stripped = strip_cv_typedef();
    auto die = stripped.get_die();
    auto tag = die.abbrev_entry()->tag;
    if (tag == DW_TAG_base_type and stripped.byte_size() <= 8) { ❸
        switch (die[DW_AT_encoding].as_int()) {
        case DW_ATE_boolean: classes[0] = parameter_class::integer; break;
        case DW_ATE_float: classes[0] = parameter_class::sse; break;
        case DW_ATE_signed: classes[0] = parameter_class::integer; break;
        case DW_ATE_signed_char: classes[0] = parameter_class::integer; break;
        case DW_ATE_unsigned: classes[0] = parameter_class::integer; break;
        case DW_ATE_unsigned_char: classes[0] = parameter_class::integer; break;
```

```
        default: sdb::error::send("Unimplemented base type encoding");
        }
    }
    else if (tag == DW_TAG_pointer_type or ❹
        tag == DW_TAG_reference_type or
        tag == DW_TAG_rvalue_reference_type) {
        classes[0] = parameter_class::integer;
    }
    else if (tag == DW_TAG_base_type and ❺
        die[DW_AT_encoding].as_int() == DW_ATE_float and
        stripped.byte_size() == 16) {
        classes[0] = parameter_class::x87;
        classes[1] = parameter_class::x87up;
    }
    else if (tag == DW_TAG_class_type or ❻
        tag == DW_TAG_structure_type or
        tag == DW_TAG_union_type or
        tag == DW_TAG_array_type) {
        classes = classify_class_type(*this);
    }
    return classes;
}
```

We initialize the classes to `no_class` ❶. We first handle built-in types, which include literal arguments that the user passes on the command line ❷. We give all of these types the class `INTEGER`, apart from floating-point numbers, which we give the class `SSE`. We give string arguments the class `INTEGER` because we'll pass a pointer to the start of the string, and we assign pointers the class `INTEGER`.

We then handle arguments that come from the program itself (that is, named variables), starting with base types that fit in a single eightbyte ❸. We assign these the classes specified by the classification algorithm.

Next, we handle pointer and reference types ❹. The Itanium ABI says that we should pass C++ references (both lvalue references and rvalue references) as pointers to the referenced object, so we assign them the class `INTEGER`, the same as pointers. We also handle base types that are larger than a single eightbyte ❺. Because of the simplifications we're making in our implementation, these include only `long double`.

Finally, we handle class and array types by calling a `classify_class_type` function which we'll write next, and then returning ❻. Note that in C and C++, array function parameters are actually pointer parameters. That is, the two following function declarations are equivalent:

```
void takes_pointer(int p[]);
void takes_pointer(int* p);
```

The DWARF information reflects this; it will have a formal parameter DIE pointing to a `DW_TAG_pointer_type` DIE in both cases. We still need to

handle array types in cases where the caller passes a class type with an array member, however. In those situations, we should treat the array as if it were a class with a member for each element of the array. We can pass it in registers if it's small enough and we can pass the stored types in registers.

Classifying Class and Array Types

To classify classes and arrays, let's implement `classify_class_type` next:

```
namespace {
    std::array<sdb::parameter_class, 2> classify_class_type(
          const sdb::type& type) {
        if (type.is_non_trivial_for_calls()) { ❶
            sdb::error::send("NTFPOC types are not supported");
        }

        if (type.byte_size() > 16 or ❷
            type.has_unaligned_fields()) {
            return {
                sdb::parameter_class::memory,
                sdb::parameter_class::memory
            };
        }

        std::array<sdb::parameter_class, 2> classes = { ❸
            sdb::parameter_class::no_class,
            sdb::parameter_class::no_class
        };

        if (type.get_die().abbrev_entry()->tag == DW_TAG_array_type) { ❹
            auto value_type = type.get_die()[DW_AT_type].as_type();
            classes = value_type.get_parameter_classes();
            if (type.byte_size() > 8 and classes[1] == sdb::parameter_class::no_class) {
                classes[1] = classes[0];
            }
        }
        else { ❺
            for (auto child : type.get_die().children()) {
                if (child.abbrev_entry()->tag == DW_TAG_member and
                    child.contains(DW_AT_data_member_location) or
                    child.contains(DW_AT_data_bit_offset)) {
                    classify_class_field(type, child, classes, 0);
                }
            }
        }

        if (classes[0] == sdb::parameter_class::memory or ❻
            classes[1] == sdb::parameter_class::memory) {
            classes[0] = classes[1] = sdb::parameter_class::memory;
```

```
        }
        else if (classes[1] == sdb::parameter_class::x87up and
            classes[0] != sdb::parameter_class::x87) {
            classes[0] = classes[1] = sdb::parameter_class::memory;
        }

        return classes;
    }
}
```

If the type is NTFPOC, we throw an exception, as we won't support passing such types by value ❶. If the type is larger than 2 eightbytes or contains unaligned fields, we pass it in memory ❷.

Next, we handle the case where the type can be passed in registers, depending on what its members are. We initialize the classes to `NO_CLASS` ❸. If the type is an array ❹, we classify its value type and use this as the class for the array. If the array is larger than an eightbyte but its elements fit in a single eightbyte, we copy the parameter class of the first eightbyte into the second so they're handled in the same way. If the type is a class or union type ❺, we loop over all non-static data members and classify them using a `classify_class_field` function we'll write shortly. The final argument to this function is the field's bit offset, which we'll use to support the passing of nested types in registers.

Now comes the post-merger cleanup pass ❻, which we'll only partly implement, as we're not supporting several types. If either eightbyte is passed in memory or `X87UP` isn't preceded by `X87`, we pass the whole type in memory. Finally, we return the computed classes.

Classifying Class Fields

Before we classify class fields, we'll add a couple of helpers to `sdb::type` for recognizing class and reference types. Declare them in *sdb/include/libsdb/type.hpp*:

```
namespace sdb {
    class type {
    public:
        --snip--
        bool is_class_type() const;
        bool is_reference_type() const;
        --snip--
    };
}
```

Define them in *sdb/src/type.cpp*:

```
bool sdb::type::is_class_type() const {
    if (!is_from_dwarf()) return false;
    auto stripped = strip_cv_typedef().get_die();
    auto tag = stripped.abbrev_entry()->tag;
```

```
        return tag == DW_TAG_class_type or
            tag == DW_TAG_structure_type or
            tag == DW_TAG_union_type;
    }

    bool sdb::type::is_reference_type() const {
        if (!is_from_dwarf()) return false;
        auto stripped = strip_cv_typedef().get_die();
        auto tag = stripped.abbrev_entry()->tag;
        return tag == DW_TAG_reference_type or
            tag == DW_TAG_rvalue_reference_type;
    }
```

These helpers strip qualifiers from the type and check whether the tag is one of the expected ones. Now we can implement `classify_class_field`, which should recursively classify the field of a type and merge together the classes for each eightbyte:

```
namespace {
    void classify_class_field(
          const sdb::type& type,
          const sdb::die& field,
          std::array<sdb::parameter_class, 2>& classes,
          int bit_offset) {
        auto bitfield_info = field.get_bitfield_information(type.byte_size());
        auto field_type = field[DW_AT_type].as_type();

        auto bit_size = bitfield_info ?
            bitfield_info->bit_size :
            field_type.byte_size() * 8;
        auto current_bit_offset = bitfield_info ?
            bitfield_info->bit_offset + bit_offset :
            field[DW_AT_data_member_location].as_int() * 8 + bit_offset;
        auto eightbyte_index = current_bit_offset / 64;

        if (field_type.is_class_type()) { ❶
            for (auto child : field_type.get_die().children()) {
                if (child.abbrev_entry()->tag == DW_TAG_member and
                    child.contains(DW_AT_data_member_location) or
                    child.contains(DW_AT_data_bit_offset)) {
                    classify_class_field(type, child, classes, current_bit_offset);
                }
            }
        }
        else { ❷
            auto field_classes = field_type.get_parameter_classes();
            classes[eightbyte_index] = merge_parameter_classes( ❸
                classes[eightbyte_index], field_classes[0]);
```

```
            if (eightbyte_index == 0) { ❹
                classes[1] = merge_parameter_classes(classes[1], field_classes[1]);
            }
        }
    }
}
```

Because the field could be a bitfield, we attempt to get bitfield information for it. We also extract the type of the field for convenience.

The field's bit size might be stored in the bitfield info; if not, we can calculate it by multiplying the field's byte size by eight. Similarly, the current bit offset into the argument we're classifying is the result of adding the `bit_offset` argument to either the bitfield's offset or the byte offset of the field multiplied by eight. The index of the eightbyte to which this field belongs (either 0 or 1) is then the result of dividing the current bit offset by the size of an eightbyte (64 bits).

If the field is itself a class type ❶, we classify its fields recursively. We loop over the non-static data members of the class, calling `classify_class_field` with the members and the current bit offset of the enclosing class inside of the argument being passed.

If the field isn't a class type ❷, we classify it, merging its parameter classes with the ones we've currently computed using a `merge_parameter_classes` function we'll write next. We always merge the field's first parameter class with the current parameter class in the eightbyte to which the field belongs ❸. The field may span 2 eightbytes, however, so we also merge the second parameter classes if this field belongs to the first eightbyte ❹.

Before we move on to implementing the other member functions we've already declared, let's implement `merge_parameter_classes`. This function performs the merge algorithm described in "Classifying Parameters" on page 644:

```
namespace {
    sdb::parameter_class merge_parameter_classes(
          sdb::parameter_class lhs, sdb::parameter_class rhs) {
        using namespace sdb;
        if (lhs == rhs) return lhs;

        if (lhs == parameter_class::no_class) return rhs;
        if (rhs == parameter_class::no_class) return lhs;

        if (lhs == parameter_class::memory or
            rhs == parameter_class::memory) {
            return parameter_class::memory;
        }

        if (lhs == parameter_class::integer or
            rhs == parameter_class::integer) {
            return parameter_class::integer;
```

```
                }

                if (lhs == parameter_class::x87 or
                    rhs == parameter_class::x87 or
                    lhs == parameter_class::x87up or
                    rhs == parameter_class::x87up or
                    lhs == parameter_class::complex_x87 or
                    rhs == parameter_class::complex_x87) {
                    return parameter_class::memory;
                }

                return parameter_class::sse;
            }
        }
```

This function applies a pretty direct translation of the textual algorithm we discussed earlier.

Checking Call Triviality

We can now move on to the `is_non_trivial_for_calls` function we declared earlier. This function returns whether the type is NTFPOC:

```
bool sdb::type::is_non_trivial_for_calls() const {
    auto stripped = strip_cv_typedef().get_die();
    auto tag = stripped.abbrev_entry()->tag;
    if (tag == DW_TAG_class_type or
        tag == DW_TAG_structure_type or
        tag == DW_TAG_union_type) {
        for (auto& child : stripped.children()) {
            if (child.abbrev_entry()->tag == DW_TAG_member and ❶
                child.contains(DW_AT_data_member_location) or
                child.contains(DW_AT_data_bit_offset)) {
                if (child[DW_AT_type].as_type().is_non_trivial_for_calls()) {
                    return true;
                }
            }
            if (child.abbrev_entry()->tag == DW_TAG_inheritance) { ❷
                if (child[DW_AT_type].as_type().is_non_trivial_for_calls()) {
                    return true;
                }
            }
            if (child.contains(DW_AT_virtuality) and ❸
                child[DW_AT_virtuality].as_int() != DW_VIRTUALITY_none) {
                return true;
            }
            if (child.abbrev_entry()->tag == DW_TAG_subprogram) { ❹
                if (is_copy_or_move_constructor(*this, child)) {
```

```
                if (!child.contains(DW_AT_defaulted) or
                    !child[DW_AT_defaulted].as_int() != DW_DEFAULTED_in_class) {
                    return true;
                }
            }
            else if (is_destructor(child)) {
                if (!child.contains(DW_AT_defaulted) or
                    !child[DW_AT_defaulted].as_int() != DW_DEFAULTED_in_class) {
                    return true;
                }
            }
        }
    }
    if (tag == DW_TAG_array_type) { ❺
        return stripped[DW_AT_type].as_type().is_non_trivial_for_calls();
    }
    return false;
}
```

For class and union types, we loop through all child DIEs. If any non-static data members ❶ or base classes ❷ are NTFPOC, we return `true`. If the type has any virtual data members or virtual base classes, represented by a `DW_AT_virtuality` member with a value of either `DW_VIRTUALITY_virtual` or `DW_VIRTUALITY_pure_virtual`, we return `true` ❸. If the type has a copy/move constructor or destructor child that isn't defaulted, we also return `true` ❹. Note that while `DW_AT_defaulted` is part of the DWARF 5 specification, GCC still omits it in DWARF 4 mode. We'll write the `is_copy_or_move_constructor` and `is_destructor` functions next.

For array types ❺, we return whether the value type of the array is NTFPOC.

Identifiying Constructors and Destructors

Let's implement `is_copy_or_move_constructor` and `is_destructor`, which identify functions that are either copy or move constructors or destructors. We'll start with the latter, as it's much simpler:

```
namespace {
    bool is_destructor(const sdb::die& func) {
        auto name = func.name();
        return name and
            name.value().size() > 1 and
            name.value()[0] == '~';
    }
}
```

We return whether the function has a name that begins with a tilde, which only destructors do.

Next, we'll consider copy and move constructors. There's no quick and easy way to identify these. We'll consider a function to be a copy or move constructor if it has the same name as the type and two parameters: a pointer to the type and an lvalue or rvalue reference to the type. The implementation is as follows:

```
namespace {
    bool is_copy_or_move_constructor(
          const sdb::type& class_type, const sdb::die& func) {
        auto class_name = class_type.get_die().name();
        if (class_name != func.name()) return false;

        int i = 0;
        for (auto child : func.children()) {
            if (child.abbrev_entry()->tag == DW_TAG_formal_parameter) {
                if (i == 0) {
                    auto type = child[DW_AT_type].as_type();
                    if (type.get_die().abbrev_entry()->tag != DW_TAG_pointer_type)
                        return false;
                    if (type.get_die()[DW_AT_type].as_type().strip_cv_typedef() != class_type)
                        return false;
                }
                else if (i == 1) {
                    auto type = child[DW_AT_type].as_type();
                    auto tag = type.get_die().abbrev_entry()->tag;
                    if (tag != DW_TAG_reference_type and
                        tag != DW_TAG_rvalue_reference_type)
                        return false;
                    auto ref = type.get_die()[DW_AT_type].as_type().strip_cv_typedef();
                    if (ref != class_type)
                        return false;
                }
                else {
                    return false;
                }
            }
            ++i;
        }
        return i == 2;
    }
}
```

We first ensure that the class and function have the same name. We then loop over the children of the function DIE, which gives us access to its parameters. If the first parameter isn't a pointer to the class type, we return `false`. If the second parameter isn't a reference to the class type, we return `false`. If there are more than two parameters, we also return `false`. Otherwise, we return `true`.

Checking for Unaligned Fields

The final functions we need to implement are `alignment` and `has_unaligned_fields`, which check whether a type's fields are properly aligned.

The SYSV ABI specifies a type's expected alignment. Class types should be aligned to the same boundary as their most strictly aligned member (that is, the member with the largest alignment boundary expectation). Arrays should be aligned to the same boundary as their element type. Other types should be aligned to the same boundary as their byte size. The `alignas` specifier can strengthen the alignment requirements of a type, but for simplicity, we'll ignore that possibility. Let's implement `alignment` to reflect this:

```
std::size_t sdb::type::alignment() const {
    if (!is_from_dwarf()) {
        return byte_size();
    }
    if (is_class_type()) {
        std::size_t max_alignment = 0;
        for (auto child : get_die().children()) {
            if (child.abbrev_entry()->tag == DW_TAG_member and
                child.contains(DW_AT_data_member_location) or
                child.contains(DW_AT_data_bit_offset)) {
                auto member_type = child[DW_AT_type].as_type();
                if (member_type.alignment() > max_alignment) {
                    max_alignment = member_type.alignment();
                }
            }
        }
        return max_alignment;
    }
    if (get_die().abbrev_entry()->tag == DW_TAG_array_type) {
        return get_die()[DW_AT_type].as_type().alignment();
    }
    return byte_size();
}
```

If the type is a built in one, we return the type's byte size. If it's a class type, we loop through the non-static members of the type, find the one that has the largest expected alignment, and return that alignment. If the type is an array, we return the alignment of its element type. Otherwise, we return the size of the type in bytes.

The `has_unaligned_fields` function should check whether any members of the class don't have their expected alignment. This could occur if, for instance, a user defines a class with `#pragma pack` to avoid padding between fields. Here is the implementation:

```
bool sdb::type::has_unaligned_fields() const {
    if (!is_from_dwarf()) {
        return false;
```

```
    }
    if (is_class_type()) {
        for (auto child : get_die().children()) {
            if (child.abbrev_entry()->tag == DW_TAG_member and
                child.contains(DW_AT_data_member_location)) {
                auto member_type = child[DW_AT_type].as_type();
                if (child[DW_AT_data_member_location].as_int() %
                    member_type.alignment() != 0) {
                    return true;
                }
                if (member_type.has_unaligned_fields()) {
                    return true;
                }
            }
        }
    }
    return false;
}
```

If the type is built-in, we return `false`. If it's a class type, we loop through the non-static members. If the member's byte offset isn't aligned to the expected boundary or if the member itself has unaligned fields, we return `true`. This completes our support for classifying types.

Implementing Argument Passing

Now we can implement the algorithm for passing a function's arguments. We'll do so in a function called `setup_arguments`, in *sdb/src/target.cpp*. The function is rather long and complex, so we'll write it piece by piece.

The function takes the target in which the function is running, a DIE representing the function we're trying to call, the arguments to call it with, the current set of registers in the machine, and an optional virtual address that points to the location where the return value should be stored. We'll start by defining the sequence of registers used for integer and SSE arguments, according to the calling conventions:

```
namespace {
    void setup_arguments(
          sdb::target& target, sdb::die func,
          std::vector<sdb::typed_data> args,
          sdb::registers& regs,
          std::optional<sdb::virt_addr> return_slot) {
        std::array<sdb::register_id, 6> int_regs = {
            sdb::register_id::rdi,
            sdb::register_id::rsi,
            sdb::register_id::rdx,
            sdb::register_id::rcx,
            sdb::register_id::r8,
```

```
        sdb::register_id::r9
    };

    std::array<sdb::register_id, 8> sse_regs = {
        sdb::register_id::xmm0,
        sdb::register_id::xmm1,
        sdb::register_id::xmm2,
        sdb::register_id::xmm3,
        sdb::register_id::xmm4,
        sdb::register_id::xmm5,
        sdb::register_id::xmm6,
        sdb::register_id::xmm7
    };
```

Next, we'll initialize some variables we'll need for executing the allocation algorithm: the index of the next free integer and SSE register, a list of arguments we've allocated to the stack, and the current stack pointer value. We'll also define a lambda function that rounds up a given size to the size of an eightbyte:

```
    auto current_int_reg = 0;
    auto current_sse_reg = 0;
    struct stack_arg {
        sdb::typed_data data;
        std::size_t size;
    };
    auto stack_args = std::vector<stack_arg>{};
    auto rsp = regs.read_by_id_as<std::uint64_t>(sdb::register_id::rsp);

    auto round_up_to_eightbyte = [](std::size_t size) {
        return (size + 7) & ~7;
    };
```

We track both the typed data and a size for stack arguments because their size gets rounded up to the nearest eightbyte, so the size allocated on the stack isn't necessarily the same as the size of the argument data.

Recall that if the function has a return value with the parameter class `MEMORY`, the caller allocates memory to store the return value and passes a pointer to this memory as the first argument to the function. We handle this fact by incrementing `current_int_reg` to record that the first integer register is no longer free and then writing the address of the return slot into that register. We'll ensure that `return_slot` always has a valid address when the return type should be stored in memory when we call `setup_arguments`, so we don't need to check whether `return_slot` is empty here:

```
    if (func.contains(DW_AT_type)) {
        auto ret_type = func[DW_AT_type].as_type();
        auto ret_class = ret_type.get_parameter_classes()[0];
        if (ret_class == sdb::parameter_class::memory) {
```

```
            current_int_reg++;
            regs.write_by_id(int_regs[0], return_slot->addr(), true);
        }
    }
```

With the return value handled, we can move on to the arguments. Before allocating arguments, we'll do a pass over them to replace arguments that we should pass by reference with a pointer to their data. If the argument we're passing lives in memory already, we pass a pointer to where it lives. Otherwise, we copy it onto the stack and pass a pointer to the copy:

```
    auto params = func.parameter_types();
    for (auto i = 0; i < params.size(); ++i) {
        auto& param = params[i];
        auto param_classes = param.get_parameter_classes();

        if (param.is_reference_type()) {
            if (args[i].address()) {
                args[i] = sdb::typed_data{
                    sdb::to_byte_vec(*args[i].address()),
                    sdb::builtin_type::integer };
            }
            else {
                rsp -= args[i].value_type().byte_size();
             ❶ rsp &= ~(args[i].value_type().alignment() - 1);
                target.get_process().write_memory(
                    sdb::virt_addr{ rsp }, args[i].data());
                args[i] = sdb::typed_data{
                    sdb::to_byte_vec(rsp),
                    sdb::builtin_type::integer };
            }
        }
    }
```

Before copying the value onto the stack, we align the stack pointer to the alignment requirement of the type we're passing ❶.

Now we can start allocating the arguments. We'll loop through the arguments, allocating them to either the stack or their relevant registers:

```
    for (auto i = 0; i < params.size(); ++i) {
        auto& arg = args[i];
        auto& param = params[i];
        auto param_classes = params[i].get_parameter_classes();
        auto param_size = param.byte_size();

        auto required_int_regs = std::count(
            param_classes.begin(), param_classes.end(),
            sdb::parameter_class::integer);
        auto required_sse_regs = std::count(
```

```
            param_classes.begin(), param_classes.end(),
            sdb::parameter_class::sse);

    ❶ if (current_int_reg + required_int_regs > int_regs.size() or
            current_sse_reg + required_sse_regs > sse_regs.size() or
            (required_int_regs == 0 and required_sse_regs == 0)) {
            auto size = round_up_to_eightbyte(param_size);
            stack_args.push_back({ args[i], size });
        }
    ❷ else {
            for (auto i = 0; i < param_size; i += 8) {
                sdb::register_id reg;
                switch (param_classes[i / 8]) {
                case sdb::parameter_class::integer:
                    reg = int_regs[current_int_reg++];
                    break;
                case sdb::parameter_class::sse:
                    reg = sse_regs[current_sse_reg++];
                    break;
                case sdb::parameter_class::no_class:
                    break;
                default:
                    sdb::error::send("Unsupported parameter class");
                }

                sdb::byte64 data;
                std::copy(
                    arg.data().begin() + i,
                    arg.data().begin() + i + 8,
                    data.begin());
                regs.write_by_id(reg, data, true);
            }
        }
    }
```

We start by making a few convenience variables and then compute how many integer and SSE registers the argument requires. We must allocate the argument on the stack if it requires no registers or if it requires registers but too few registers are available ❶. In this case, we push the argument into the `stack_args` vector, along with the size of the parameter type rounded up to the nearest eightbyte. If we can pass the argument in registers ❷, we loop over the eightbytes in the argument, allocate them a register of the correct type, and then copy the argument's data into the allocated register.

Next, we'll handle the arguments that we should pass on the stack. The SYSV ABI requires that the address at the end of the stack arguments be aligned to a 16-byte boundary. Also recall that stack arguments get pushed in reverse order, from right to left. As such, we'll subtract the number of

bytes of storage we require from the stack pointer, align it to a 16-byte boundary, and then copy the arguments such that the leftmost argument is at the top of the stack:

```
        for (auto& [_,size] : stack_args) {
            rsp -= size;
        }
    ❶ rsp &= ~0xf;

        auto start_pos = rsp;
        for (auto& [arg,size] : stack_args) {
            target.get_process().write_memory(
                sdb::virt_addr{ start_pos }, arg.data());
            start_pos += size;
        }
```

Bitwise ANDing the stack pointer with the complement of `0xf` aligns it to a 16-byte boundary ❶. Finally, we write the number of SSE registers we used to `rax`, as expected by varargs functions:

```
        regs.write_by_id(sdb::register_id::rax, current_sse_reg, true);
        regs.write_by_id(sdb::register_id::rsp, rsp, true);
    }
}
```

We also update the stack pointer.

Reading Return Values

Once we've called a function that has a return value, we may need to extract the return value from registers. Let's write a function called `read_return_value` that carries out this task.

Recall that we want the user to be able to reference return values with numbered variables, like `$0` and `$1`, after they've evaluated an expression. This means the memory for return values must stick around for the rest of the debugging session. Recall also that when the return value gets classified with the `MEMORY` class, the caller allocates memory to store the return value.

To support all of this, our expression evaluator will dynamically allocate storage for the return value, regardless of whether the return value has the `MEMORY` class. If the return value does have the `MEMORY` class, the evaluator will pass a pointer to the allocated storage it should use as the return storage. Otherwise, the return value is stored in registers, and it's the job of `read_return_value` to copy the data into the allocated storage so users can access it later.

The `read_return_value` function will take the target being executed, the DIE of the function we called, a virtual address to the storage allocated for

the return value, and the state of the registers after the function call. Implement the function in *sdb/src/target.cpp*:

```
namespace {
    sdb::typed_data read_return_value(
          sdb::target& target, sdb::die func,
          sdb::virt_addr return_slot, sdb::registers& regs) {
        auto ret_type = func[DW_AT_type].as_type();
        auto ret_classes = ret_type.get_parameter_classes();

        bool used_int = false;
        bool used_sse = false;

        if (ret_classes[0] == sdb::parameter_class::memory) {
            auto value = target.get_process().read_memory(
                return_slot, ret_type.byte_size());
            return { sdb::typed_data{
                std::move(value), func[DW_AT_type].as_type(), return_slot } };
        }

        if (ret_classes[0] == sdb::parameter_class::x87) {
            auto data = regs.read_by_id_as<long double>(sdb::register_id::st0);
            auto value = sdb::to_byte_vec(data);
            target.get_process().write_memory(return_slot, value);
            return { sdb::typed_data{
                std::move(value), func[DW_AT_type].as_type(), return_slot } };
        }

        std::vector<std::byte> value;
        for (auto ret_class : ret_classes) {
            if (ret_class == sdb::parameter_class::integer) {
                auto reg = used_int ? sdb::register_id::rdx : sdb::register_id::rax;
                used_int = true;
                auto data = regs.read_by_id_as<std::uint64_t>(reg);
                auto new_value = sdb::to_byte_vec(data);
                value.insert(value.end(), new_value.begin(), new_value.end());
            }
            else if (ret_class == sdb::parameter_class::sse) {
                auto reg = used_sse ? sdb::register_id::xmm1 : sdb::register_id::xmm0;
                used_sse = true;
                auto data = regs.read_by_id_as<sdb::byte128>(reg);
                value = { data.begin(), data.end() };
                target.get_process().write_memory(return_slot, value);
            }
            else if (ret_class != sdb::parameter_class::no_class) {
                sdb::error::send("Unsupported return type");
            }
        }
```

```
        target.get_process().write_memory(return_slot, value);
        return { sdb::typed_data{
            std::move(value), func[DW_AT_type].as_type(), return_slot } };
    }
}
```

We start by grabbing the return type and the classes for the return type, for convenience. We initialize Booleans to track whether we've used an integer or SSE register yet so we can return structures stored across two integer or SSE registers.

If the first class is `MEMORY`, the return value is already stored in the return slot, so we read the value and return it. If the first class is `X87`, the value must be a `long double` stored in the `st0` register. We read this register, write the result to the return slot, and return.

Otherwise, we'll find the return value in the integer or SSE registers, or potentially in both. We initialize an empty vector of bytes to store the return value. We then loop over the classes. If the current class is `INTEGER`, the register is `rdx` if we've already read an integer return value and `rax` otherwise. We set `used_int` to `true` to make sure we use the correct register if the function returns another integer. We then read the data from the relevant register and insert its value into the overall return value vector.

We perform the same process for the `SSE` class, except we read from either `xmm1` or `xmm0`, and we read an `sdb::byte128` out of the register instead of an integer. If the class is something else (and not `NO_CLASS`), we throw an exception. Finally, we write the value we read into the return slot and return all of the data we computed.

Indexing Member Functions

We're getting close to being able to call functions from the debugger. The next missing piece is to tie together the low-level machinery we've created to the DWARF information, as well as to our argument allocator. However, our ability to do so depends on one operation that is difficult to carry out with our existing DWARF support: finding the definition of a member function based on its declaration. Let's add better support for this.

Add a `get_member_function_definition` function to `sdb::dwarf`, along with a new index variable that we'll use to map declarations to their definitions. Make these changes in *sdb/include/libsdb/dwarf.hpp*:

```
namespace sdb {
    class dwarf {
    public:
        --snip--
        std::optional<die> get_member_function_definition(
            const sdb::die& declaration) const;
        --snip--
```

```
    private:
        --snip--
        mutable std::unordered_map<const std::byte*, index_entry>
        member_function_index_;
        --snip--
    };
}
```

The `get_member_function_definition` function returns a DIE representing the definition that corresponds to the given declaration, if it finds one. The `member_function_index_` variable maps from the byte position of a member function declaration DIE to an index entry representing the definition's DIE.

Member function definitions point to their declarations in a few ways. The simplest cases have a single DIE representing the declaration and a single DIE representing the definition that references the declaration in its `DW_AT_specification` attribute. In some cases, however, a compiler will spread the information for a definition across two DIEs to support representing inlining information, as shown in Figure 21-1.

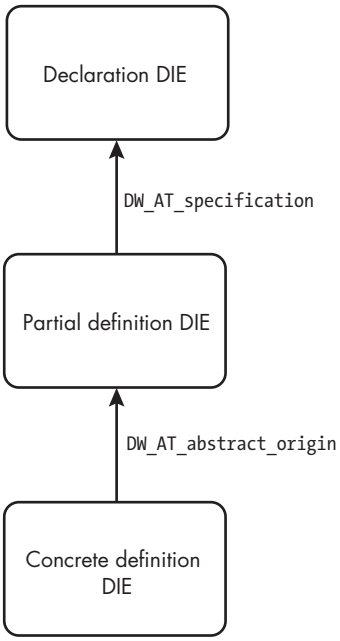


Figure 21-1: A chain of member function DIEs

In these cases, the DWARF information stores some of the details about the definition in a DIE that points to the declaration with a `DW_AT_specification` attribute, and it stores the rest of the information in a DIE that uses a `DW_AT_abstract_origin` attribute to point to the first piece of information.

Fill in the member function index by modifying the `sdb::dwarf::index_die` function in *sdb/src/dwarf.cpp* to add index entries for any function it finds with a `DW_AT_specification` or `DW_AT_abstract_origin` attribute:

```
void sdb::dwarf::index_die(const die& current, bool in_function) const {
    --snip--
    if (has_range and is_function) {
        --snip--
    }

    if (is_function) {
        if (current.contains(DW_AT_specification)) {
            index_entry entry{ current.cu(), current.position() };
            member_function_index_.insert(std::make_pair(
                current[DW_AT_specification].as_reference().position(), entry));
        }
        else if (current.contains(DW_AT_abstract_origin)) {
            index_entry entry{ current.cu(), current.position() };
            member_function_index_.insert(std::make_pair(
                current[DW_AT_abstract_origin].as_reference().position(), entry));
        }
    }
    --snip--
}
```

If the current DIE represents a function, we create an index entry so long as it has a `DW_AT_specification` or `DW_AT_abstract_origin` attribute. This entry maps the byte position of the DIE pointed to by that attribute to an index entry that corresponds to the definition DIE.

Implement `get_member_function_definition` in the same file:

```
std::optional<sdb::die>
sdb::dwarf::get_member_function_definition(
      const sdb::die& declaration) const {
    index();
    auto it = member_function_index_.find(declaration.position());
    if (it != member_function_index_.end()) {
        cursor cur({ it->second.pos, it->second.cu->data().end() });
        auto die = parse_die(*it->second.cu, cur);
        if (die.contains(DW_AT_low_pc) or die.contains(DW_AT_ranges)) {
            return die;
        }
        return get_member_function_definition(die);
    }
    return std::nullopt;
}
```

First, we call `index` to ensure that the DWARF information has been indexed. We then look for the byte position of the declaration DIE in the

member function index and parse a DIE at the resulting location if we find one. If that DIE has address information, we've found the final definition. If not, we could be looking at a chain of definitions, as in the case described earlier, with the `DW_AT_abstract_origin` attribute. In this situation, we recursively call `get_member_function_definition` to find the concrete definition. If we don't find a definition with address information, we return `std::nullopt`.

Performing High-Level Inferior Calls

Now we can tie all of this functionality together in a higher-level inferior call function, which we'll name `inferior_call_from_dwarf`. It should take the target being debugged, the DIE for the function it should call, the arguments to call it with, the return address to jump back to, and the thread on which to perform the call. After performing the function call, it should return the function's return value if there is one.

We'll implement this function piece by piece. Add it to *sdb/src/target.cpp*:

```
namespace {
    std::optional<sdb::typed_data> inferior_call_from_dwarf(
          sdb::target& target, sdb::die func,
          const std::vector<sdb::typed_data>& args,
          sdb::virt_addr return_addr, pid_t tid) {
        auto& regs = target.get_process().get_registers(tid);
        auto saved_regs = regs;

        sdb::virt_addr call_addr;
        if (func.contains(DW_AT_low_pc) or func.contains(DW_AT_ranges)) {
            call_addr = func.low_pc().to_virt_addr();
        }
        else {
            auto def = func.cu()->dwarf_info()->get_member_function_definition(func);
            if (!def) {
                sdb::error::send("No function definition found");
            }
            call_addr = def->low_pc().to_virt_addr();
        }

        std::optional<sdb::virt_addr> return_slot;
        if (func.contains(DW_AT_type)) {
            auto ret_type = func[DW_AT_type].as_type();
            return_slot = target.inferior_malloc(ret_type.byte_size());
        }

        setup_arguments(target, func, args, regs, return_slot);
        auto new_regs = target.get_process().inferior_call(
            call_addr, return_addr, saved_regs, tid);
```

```
        if (func.contains(DW_AT_type)) {
            return read_return_value(
                target, func, *return_slot, new_regs);
        }
        return std::nullopt;
    }
}
```

We save the current set of registers and then find the address of the function we should call. If the DIE we're given has address range information, we use that. Otherwise, it probably represents a member function declaration, so we look up the definition for the member function and use that. If that fails, we throw an exception.

If the function has a return value, we allocate space for it. We want the user to be able to access these return values later through numbered variables like `$0`, so we dynamically allocate the storage for the return value rather than allocating it on the stack, where it will be reclaimed after the current function exits.

Finally, we set up the arguments for the function call and perform the call. If the function has a return value, we read it and return the result; otherwise, we return `std::nullopt`.

Evaluating Expressions

We've implemented quite a lot of support code and are now ready to tie it all together. The main entry point to our expression evaluation engine will be `sdb::target::evaluate_expression`. We'll also add a data member for tracking the history of return values from evaluated expressions and a member function for retrieving a result given an index. Add these to *sdb/include/libsdb/target.hpp*:

```
namespace sdb {
    class target {
    public:
        --snip--
        struct evaluate_expression_result {
            typed_data return_value;
            std::uint64_t id;
        };
        std::optional<evaluate_expression_result> evaluate_expression(
            std::string_view expr,
            std::optional<pid_t> otid = std::nullopt);

        const typed_data& get_expression_result(std::size_t i) const;

    private:
        --snip--
```

```
        mutable std::vector<typed_data> expression_results_;
    }
}
```

The `evaluate_expression` function takes an expression to evaluate and, optionally, a thread on which to evaluate it. If the function called has a return value, `evaluate_expression` returns this value along with the integer index assigned to the result, which users will be able to access through variables like `$1`. The `get_expression_result` function takes this identifier and returns a reference to the relevant result. We mark `expression_results_` as `mutable` because the `get_expression_result` function may need to reread the data in one of the result slots in cases when that variable is passed by reference to a function as part of another expression evaluation.

Implement the two member functions in *sdb/src/target.cpp*, starting with `evaluate_expression`:

```
std::optional<sdb::target::evaluate_expression_result>
sdb::target::evaluate_expression(
      std::string_view expr, std::optional<pid_t> otid) {
    auto tid = otid.value_or(process_->current_thread());
    auto pc = get_pc_file_address(tid);

    auto paren_pos = expr.find('(');
    if (paren_pos == std::string::npos) {
        sdb::error::send("Invalid expression");
    }

    std::string name{ expr.substr(0, paren_pos + 1) };
    auto [variable, funcs] = resolve_indirect_name(name, pc);
    if (funcs.empty()) {
        sdb::error::send("Invalid expression");
    }

    auto entry_point = virt_addr{ process_->get_auxv()[AT_ENTRY] };
    breakpoints_.get_by_address(entry_point).install_hit_handler([&] {
        return false;
    });

    auto arg_string = expr.substr(paren_pos);
    auto args = collect_arguments(
        *this, tid, arg_string, funcs, variable);
    auto func = resolve_overload(funcs, args);
    auto ret = inferior_call_from_dwarf(
        *this, func, args, entry_point, tid);
    if (ret) {
        expression_results_.push_back(*ret);
        return evaluate_expression_result{
            std::move(*ret), expression_results_.size() - 1
```

```
        };
    }
    return std::nullopt;
}
```

First, we compute the correct thread on which to run and retrieve the program counter as a file address. We locate the first open parenthesis in the instruction, throwing an exception if we don't find one. The name of the function to call (which may be indirect, in the case of a member function) is the substring of the expression, starting from the beginning of the expression and going up to the parenthesis position. Note that we add one to the parenthesis position when passing it to `substr` because it expects a size rather than the end index. We resolve the name of the function and, if we didn't find one, throw an exception.

As we did in `inferior_malloc`, we replace the hit handler of the entry point breakpoint so we can use the entry point as the return address for the function call.

We find the function's arguments in the substring of the expression starting from the first open parenthesis and going up to the end of the expression. We pass this substring to `collect_arguments` to retrieve values and types for all the arguments. Based on these details, we resolve the overload for the function we should call. We then call the function and store any return value, which we push to the back of `expression_results_` and return along with its result index. Otherwise, we return `std::nullopt`. Next, let's implement `get_expression_result`:

```
const sdb::typed_data& sdb::target::get_expression_result(
      std::size_t i) const {
    auto& res = expression_results_[i];
    auto new_data = process_->read_memory(
        *res.address(), res.value_type().byte_size());
    res = typed_data{
        std::move(new_data), res.value_type(), res.address() };
    return res;
}
```

We grab a reference to the requested expression result. The value in memory may have changed since the expression was executed because a user may pass the result as a reference to a function call that modifies it. As such, before we return the data, we reread the value from memory.

Exposing Expression Evaluation to the User

All that's left to do is expose this new functionality to users and test it out. In *sdb/tools/sdb.cpp*, add a branch to `handle_command` that supports a new `expression <expression>` command:

```
namespace {
    void handle_command(std::unique_ptr<sdb::target>& target,
```

```
            std::string_view line) {
        --snip--
        else if (is_prefix(command, "expression")) {
            auto expr = line.substr(line.find(' ') + 1);
            auto ret = target->evaluate_expression(expr);
            if (ret) {
                auto str = ret->return_value.visualize(target->get_process());
                fmt::print("${}: {}\n", ret->id, str);
            }
        }
        --snip--
    }
}
```

The expression to execute is the substring of the typed line, starting after the space character following the `expression` command name. We evaluate this expression and, if it has a return value, visualize it.

Testing the Evaluator

This was the most code-heavy chapter in the whole book, but you're about to reap the benefits of all your work. Because this is the final programming chapter, I won't walk through adding automated tests, as you'll have no more opportunities to break the code unless you expand the project yourself. Instead, I'll walk through several manual testing scenarios. However, there are a few modifications that we need to make to the existing tests to make them build.

In *sdb/test/tests.cpp*, modify all the code that uses `resolve_indirect_name` to grab the `variable` member before doing anything else:

```
TEST_CASE("Global variables", "[variable]") {
    --snip--
    auto name_vis = name.variable->visualize(proc);
    --snip--
    auto cats_vis = cats.variable->visualize(proc);
    --snip--
}

TEST_CASE("Local variables", "[variable]") {
    --snip--
    REQUIRE(from_bytes<std::uint32_t>(var_data.variable->data_ptr()) == 1);
    --snip--
    REQUIRE(from_bytes<std::uint32_t>(var_data.variable->data_ptr()) == 2);
    --snip--
    REQUIRE(from_bytes<std::uint32_t>(var_data.variable->data_ptr()) == 3);
}
```

```
TEST_CASE("Member pointers", "[variable]") {
    --snip--
    auto data_vis = data_ptr.variable->visualize(proc);
    --snip--
    auto func_vis = func_ptr.variable->visualize(proc);
    --snip--
}
```

All tests should now build and pass.

Create a new file at *sdb/test/targets/expr.cpp*. We'll add a handful of types and functions to this file. First, define a `cat` type with some member functions and then create a few global `cat` instances and a `get_cat` function to find a cat by name:

```
#include <iostream>
#include <cstring>
#include <cstdint>
#include <stdexcept>

struct cat {
    const char* name;
    int age;
    void give_command(const char* command);
    void increase_age();
};
void cat::give_command(const char* command) {
    std::cout << name << ", " << command << std::endl;
}
void cat::increase_age() {
    age++;
}

cat marshmallow{ "Marshmallow", 4 };
cat milkshake{ "Milkshake", 4 };
cat lexa{ "Lexa", 8 };

cat get_cat(const char* name) {
    for (auto c : { marshmallow, milkshake, lexa }) {
        if (strcmp(c.name, name) == 0) {
            return c;
        }
    }
    throw std::runtime_error("No cat with that name");
}
```

The `give_command` member function outputs a message, while the `increase_age` function just increments the `age` member. We define these out of line to

force the compiler to generate definitions for them even if they're not used. The `get_cat` function compares the given name to the names for my cats and returns the correct one.

Next, define several overloaded functions that take arguments of different types, print them out, and return them:

```
int print_type(int i) {
    std::cout << "int " << i << '\n';
    return i;
}

double print_type(double d) {
    std::cout << "double " << d << '\n';
    return d;
}

const char* print_type(const char* s) {
    std::cout << "string " << s << '\n';
    return s;
}

char print_type(char c) {
    std::cout << "char " << c << '\n';
    return c;
}
```

We use `int`, `double`, `const char*`, and `char`, as we support all of these built-in types. Now define some classes of different sizes and additional overloads of `print_type` for them:

```
struct small {
    int i, j;
};

struct two_eightbyte {
    std::uint64_t i, j;
};

struct big {
    std::uint64_t i, j, k;
};

small print_type(small s) {
    std::cout << "small " << s.i << ' ' << s.j << '\n';
    return s;
}

two_eightbyte print_type(two_eightbyte t) {
```

```
    std::cout << "two_eightbyte " << t.i << ' ' << t.j << '\n';
    return t;
}

big print_type(big b) {
    std::cout << "big " << b.i << ' ' << b.j << ' ' << b.k << '\n';
    return b;
}

small s = { 1, 2 };
two_eightbyte t = { 3, 4 };
big b = { 5, 6, 7 };

int main() {

}
```

We should pass the `small` type in a single register, split the `two_eightbyte` type across two registers, and pass the `big` type in memory. We also define an empty `main` function.

Build and launch this program. Hit a breakpoint inside of the `main` function and then try evaluating expressions that use these different functions. Here's a conversation I had with my debugger that demonstrates how the expression evaluation should look:

```
$ tools/sdb test/targets/expr
Launched process with PID 13122
sdb> break set main
sdb> c
Thread 13122 stopped with signal TRAP at 0x555555555708, expr.cpp:84
(expr`main)  (breakpoint 1)
  81
  82 int main() {
  83
> 84 }
sdb> expr get_cat("Marshmallow")
$0: {
        name: "Marshmallow"
        age: 4
}
sdb> expr $0.give_command("have a nap")
Marshmallow, have a nap
sdb> expr $0.increase_age()
sdb> var read $0
Value: {
        name: "Marshmallow"
        age: 5
}
```

```
sdb> expr print_type(42)
int 42
$1: 42
sdb> expr print_type(42.42)
double 42.42
$2: 42.42
sdb> expr print_type('e')
char e
$3: 101
sdb> expr print_type(s)
small 1 2
$4: {
        i: 1
        j: 2
}
sdb> expr print_type(t)
two_eightbyte 3 4
$5: {
        i: 3
        j: 4
}
sdb> expr print_type(b)
bigg 5 6 7
$6: {
        i: 5
        j: 6
        k: 7
}
```

Calling `get_cat` returns the correct cat. I can then call member functions on the result using the `$0` syntax. Calling `give_command` prints out the expected message, proving that passing string literals works, and calling `increase_age` correctly increments the age, which we check by rereading the `$0` result variable. All of the `print_type` calls work, printing and returning the expected values, proving that we can both pass and return those types.

Summary

In this chapter, you learned about the calling conventions for the SYSV and Itanium ABIs. You then implemented an expression evaluator that targets these calling conventions and allows the user to execute simple function calls inside the running process.

You now have a fairly complete x64 native debugger. In the next chapter, you'll learn about some advanced topics that you can research and implement if you're looking for more features to add to the project.

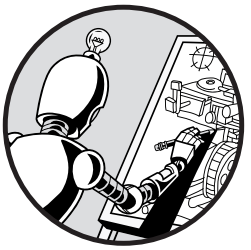
Check Your Knowledge

1. In which two documents are the calling conventions for C++ programs on Linux specified?
2. In which register is the first integer argument to a function passed?
3. What special calling convention rule comes into effect for varargs functions?
4. When are types with the parameter classification `X87` placed in registers during a function call sequence?
5. What does the term *non-trivial for the purposes of calls* refer to?
6. In what order are arguments pushed to the stack before a function call?
7. What is the stack alignment restriction for function calls?

22

ADVANCED TOPICS

For all sweeping flights
end on the ground
and breathe air anew.



In this chapter, I'll cover some advanced topics you may want to research yourself and add to your debugger. They're features that I didn't deem core to the idea of the book and that may require considerable amounts of additional code to implement. We'll cover uses of a client-server architecture for debuggers, "time-travel" debugging, exception catchpoints, non-stop mode, and follow-fork mode.

Remote Debugging

Remote debugging allows you to debug processes that are running on a different machine than the debugger itself. This is very useful for working with embedded systems as well as for debugging programs running on production machines or different operating systems. It also creates a useful logical division between high-level debugging operations and low-level interactions

with the operating system and hardware, similar to the one we created with the `sdb::target` and `sdb::process` types. In fact, even when operating locally (that is, when the debugger and debugee are on the same system), some debuggers work the same way they would in a remote debugging scenario.

To debug remote processes, debuggers launch a small server called a *debug stub* on the target machine and communicate with the stub via a text-based remote protocol called the *GDB remote serial protocol (GDB RSP)*. Figure 22-1 shows an example of a remote debugging setup.

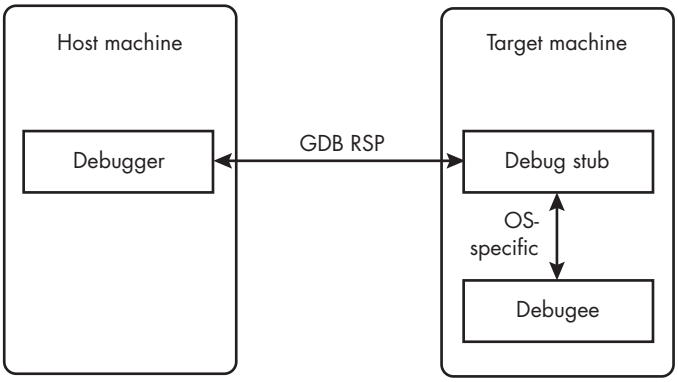


Figure 22-1: A remote debugging architecture

The host machine runs the debugger process, which communicates with the debug stub on the target machine via the GDB RSP. The debug stub communicates with the debugee process using techniques specific to the target machine's operating system, such as `ptrace` on Linux. The host machine and the target machine could be the same.

The GDB RSP is an ASCII-encoded plaintext protocol described in the GDB documentation. It transfers data in packets. Here is an example of a GDB RSP packet for setting a breakpoint:

```
$Z0,400570,1#43
```

The `$` character marks the beginning of the packet. The command type follows; in this case, it's `Z0`, which instructs the debug stub to set a software breakpoint. Commas separate commands and operands. This packet has two operands: the address at which to set the breakpoint (`0x400570`) and the size of the breakpoint (1 byte). Every packet ends with a `#` character and a checksum to ensure there is no data corruption.

The protocol is highly extensible, allowing custom packet types for platform- or language-specific actions and queries.

Development Tool Communication

Many development tools need to interface with debuggers programmatically. For example, if you're using Visual Studio Code to edit your program, you may want to debug your code from within the editor and have

it interface with GDB. There are several protocols that facilitate this, the two most popular being the GDB Machine Interface (GDB/MI) and the Debug Adapter Protocol (DAP).

Both of these protocols are text-based. The key design point that differentiates them from GDB RSP is the level of abstraction: GDB RSP commands are low-level operations that abstract the operating system layer of the debugger, whereas GDB/MI and DAP support high-level operations that drive a debugger programmatically rather than using a command line. For example, GDB RSP does not have commands for setting breakpoints on lines of source code, but GDB/MI and DAP do.

It's more common than you might think to have GDB RSP, GDB/MI, and DAP all working together in a single debugging scenario. For example, if you're using Visual Studio Code, the official C++ extension, and LLDB to debug your application, the editor communicates with the C++ extension over DAP, the extension communicates with LLDB over GDB/MI, and LLDB works in client-server mode even for local debugging, so it communicates with the debug stub over GDB RSP.

GDB Machine Interface

GDB/MI is an ASCII-encoded protocol that largely mirrors the command line interface but is intended for machine interaction. For example, here are the commands and responses for setting a temporary breakpoint on line 5 of *overloaded.cpp* using the command line interface:

```
(gdb) tbreak -source overloaded.cpp -line 5
Temporary breakpoint 1 at 0x2418: file /home/tartanllama/.vs/sdb/test/
targets/overloaded.cpp, line 5.
```

Temporary breakpoints are hit only once and then deleted. Here is the equivalent with the machine interface:

```
-break-insert -t --source overloaded.cpp --line 5
^done,bkpt={number="1",type="breakpoint",disp="del",enabled="y",addr=
"0x0000000000002418",func="print_type(int)",file="/home/tartanllama/.vs/sdb/
test/targets/overloaded.cpp",fullname="/home/tartanllama/.vs/sdb/
test/targets/overloaded.cpp", line="5",thread-groups=["i1"],times="0",
original-location="-source overloaded.cpp -line 5"}
```

GDB/MI commands tend to use flags (like `-t`) to specify special cases of commands rather than separate command names (like `tbreak`). The machine interface also gives back a lot more context so that the client doesn't need to send additional queries in order to retrieve information about the newly created breakpoint.

There isn't an exact one-to-one correspondence between command line and GDB/MI commands, although most important commands are available in both. For example, the command line has the command `whatis` for printing out formatted information about symbols, but since that's largely for human-readable output, GDB/MI doesn't have a `-what-is` command; clients

should instead use the `-symbol-info-XXX` commands. Similarly, GDB/MI has several commands for querying the level of support that the MI driver has for various GDB/MI functionalities, which the command line driver does not. However, the MI driver does accept regular command line–formatted commands in addition to GDB/MI commands, so clients can always fall back to that format.

You can find the documentation for GDB/MI alongside GDB's regular documentation.

Debug Adapter Protocol

DAP is a JSON-based protocol for communicating between IDEs/editors and debuggers. While GDB and GDB/MI are mostly intended for use with systems programming languages like C, C++, and Rust, DAP is designed to be language-agnostic and is intended to support any programming paradigm.

All DAP messages are prefixed with their content length and have their format described by the official specification at *https://microsoft.github.io/debug-adapter-protocol/*. Here is an example of a request to send the `next` command to the debugger:

```
{
  "command": "setBreakpoints",
  "arguments": {
    "source": {
      "name": "overloaded.cpp",
      "path": "sdb/src/test/targets/overloaded.cpp"
    },
    "breakpoints": [
      {
        "line": 5
      }
    ],
    "sourceModified": false
  },
  "type": "request",
  "seq": 3
}
```

The `seq` field is used to identify and order requests and responses. Here's the response from the debugger:

```
{
  "seq": 7,
  "type": "response",
  "request_seq": 3,
  "success": true,
  "command": "setBreakpoints",
  "body": {
```

```
    "breakpoints": [
      {
        "id": 1,
        "verified": false,
        "message": "The breakpoint is pending and will be resolved when debugging starts.",
        "line": 5
      }
    ]
  }
}
```

Note that the `request_seq` field matches the `seq` field of the `setBreakpoints` request.

As you can see, this format is much more verbose than GDB/MI, but it benefits from the ubiquitous support of JSON and the flexibility the format offers.

What You Should Use

If you're planning to add IDE support to your debugger, your best option is to implement a GDB/MI interface, as most common C++ development environments already know how to speak that protocol. You may find that you need to add more functionality to your debugger in order to implement enough of the protocol for existing IDEs to be able to use it for common debugger tasks.

Thread-Local Storage

Thread-local storage (TLS) provides support for variables whose lifetime and visibility is bound to a single thread. In C++, you declare them with the `thread_local` keyword. For example, consider the following code:

```
#include <iostream>
#include <thread>

int count_calls() {
    thread_local int times_called = 0;
    times_called++;
    return times_called;
}

int main() {
    auto job = []{
        std::cout << count_calls() << '\n'
                  << count_calls() << '\n'
                  << count_calls() << '\n';
    };
```

```
    std::thread t1(job);
    std::thread t2(job);
    std::thread t3(job);

    t1.join();
    t2.join();
    t3.join();
}
```

We define a `count_calls` function, which has a thread-local variable called `times_called`. We initialize it once per thread, in a manner similar to how we initialize function-scope variables declared `static` exactly once, the first time control flow reaches their definition. We then increment it on each call and return the new value.

In the `main` function, we define a lambda function that calls `count_calls` three times and outputs the results onto a new line each time. We create three threads that will run this job and wait until they've all finished execution. Because `times_called` is declared `thread_local`, each thread gets its own copy of the variable. As such, you'll see the sequence `1, 2, 3` printed out three times, potentially interleaved.

ELF files store thread-local variables in special sections marked with the `SHF_TLS` flag in the `sh_flags` field of their section headers. They store uninitialized variables in the `.tbss` section and initialized variables in the `.tdata` section. These are analogous to the `.bss` and `.data` sections. Similarly, the program headers for the segments that refer to thread-local data have the type `PT_TLS`.

Symbols that refer to thread-local variables have the `STT_TLS` symbol type in the ELF file. Their symbol value isn't a file address, but an offset from the start of the TLS block for the thread to which they belong. The `.tbss` section is always loaded directly after the `.tdata` section, so the debugger doesn't need to know which section a thread-local variable belongs to, only its offset from the TLS block start address.

When the program is running, the `fs` register stores the address of the TLS block for the currently executing thread. The C++ runtime handles this storage, so you don't need to do anything special to keep track of it. This register gives the debugger the last piece of information it needs to be able to read thread-local variables. The debugger reads the `fs` register, adds the TLS block offset of the variable, and reads the result.

DWARF information handles TLS with the DWARF expression instruction `DW_OP_form_tls_address`, which pops the value on the top of the stack and adds it to the address of the TLS block for this thread.

Time-Travel Debugging

Tools that support *time-travel debugging* enable you to rewind your program, making it easier to find bugs in tricky cases where you're not sure how to reproduce an error state, such as when debugging multithreaded programs with potential race conditions.

Different tools take different approaches to implementing this feature, each with its own trade-offs. WinDbg runs an emulator inside the debugger that captures all of the program behavior, making it very accurate and easy to use even with complex, multithreaded applications, albeit with a significant runtime overhead. UndoDB instruments the code and creates lightweight snapshots to capture only the nondeterministic parts (such as syscalls, atomic memory accesses, and I/O.) Mozilla's `rr` tool uses hardware performance counters to build an understanding of the program execution in a way that has an incredibly low runtime cost but forces the program to execute on a single thread.

If you want to look into time-travel debugging in more detail, these projects have all released videos with details about the implementation, and `rr` is open source, so you can read its code.

Exception Catchpoints

We implemented support for syscall catchpoints in Chapter 10, but there are several other types of catchpoints you could implement. For example, it can be useful to halt the program whenever a C++ exception is thrown or caught and then print information about it.

The Itanium C++ ABI specifies the rules for C++ exception handling on Linux. When an exception is thrown, the C++ runtime calls the `__cxa_throw` function, which has the following signature:

```
void __cxa_throw (
    void *thrown_exception, std::type_info *tinfo, void (*dest) (void *) );
```

The `thrown_exception` parameter points to the data of the exception being thrown, the `tinfo` parameter points to the `std::type_info` object that describes the exception's dynamic type, and the `dest` parameter points to the destructor that the code should call to destroy the thrown exception object once it has been handled.

By setting a breakpoint on `__cxa_throw`, the debugger can receive notifications about thrown exceptions. Based on the arguments to the function, it can retrieve the data for the exception and interpret it according to the dynamic type information to visualize it to the user.

Similarly, when exceptions are caught, the runtime calls the `__cxa_begin_catch` function, which has the following signature:

```
void *__cxa_begin_catch ( void *exceptionObject );
```

This function takes a pointer to the exception being caught. To retrieve type information for this exception, a debugger could track it from the matching `__cxa_throw` function.

Non-Stop Mode

Recall from Chapter 18 that most debuggers operate in all-stop mode, meaning they stop all threads whenever any one thread stops. Some operating

systems, notably Windows, operate in all-stop mode by default. Linux, however, operates in non-stop mode, where some threads can continue running while others are stopped. We forced it to run in all-stop mode by manually stopping any running threads when there is a stop.

GDB allows users to set non-stop mode with the command `set non-stop on`. You could support something similar in your debugger. Because the operating system does most of the work for us, most of the changes required to support non-stop mode involve how the debugger deals with threads. The obvious change is to not stop other threads when one thread stops, but you'd also need to change the `continue` command to operate only on the current thread.

Furthermore, you'd need to make sure you don't try to carry out operations that require the thread to be stopped, such as reading registers, on threads that are running. Much of the code for the debugger is written with the assumption that all threads are stopped whenever the user is interacting with the program, so you'll have to make many modifications if you decide to implement non-stop mode.

Follow-Fork Mode

When a program calls the `fork` system call to spawn a new process, the debugger stays attached to the original process. Sometimes, a user will want to instead debug the newly spawned process. Some debuggers, like GDB, support this in *follow-fork mode*. For example, a user can execute the command `set follow-fork-mode child` to debug the child process rather than the parent process.

Fortunately, `ptrace` has support for tracing calls to `fork`. If you set the `PTRACE_O_TRACEFORK` option in a `PTRACE_SETOPTIONS` request, when a process calls `fork`, the new process will be stopped with a `SIGSTOP` and `ptrace` will start tracing it.

If the new process calls `exec`, it will likely start executing an entirely different program, so you'll have to replace the current set of loaded ELF and DWARF files with the new ones. You can trace calls to `exec` by setting the `PTRACE_O_TRACEEXEC` option in `ptrace`.

Summary

In this chapter, you got a very brief taste of some additional features you could add to your debugger. You learned how debuggers use a client-server architecture to support both local and remote debugging and explored some of the techniques that debuggers use to travel through time. You also got a feel for how to trace C++ exceptions, implement non-stop debugging, and follow forked processes.

If you've made it this far, you now know more about debuggers than the vast majority of people in the industry. I hope you enjoyed the journey and are able to put some of these ideas and techniques into practice in your everyday work.

APPENDIX: CHECK YOUR KNOWLEDGE ANSWERS

Chapter 2: Compilation and Computer Architecture

1. ELF
2. DWARF
3. User space
4. System calls (syscalls)
5. `SIGINT`
6. Program counter or instruction pointer
7. Opcode and operand
8. Assembler
9. `0x11`, `0xba`, `0x5e`, `0xba`

Chapter 3: Attaching to a Process

1. `fork` and `exec` syscalls
2. `ptrace`
3. `waitpid`

Chapter 4: Pipes, procfs, and Automated Testing

1. Inter-process
2. 64KiB
3. Set additional flags

4. The procfs
5. The *stat* file

Chapter 5: Registers

1. GPR, x87, MMX, SSE, SSE2, AVX, AVX-512, and debug
2. `rip`
3. With the `e` prefix
4. MMX and x87
5. The SYSV ABI
6. Marshmallow, Milkshake, and Lexa
7. `long double`
8. Any registers other than GPR and debug registers
9. IEEE 754

Chapter 6: Testing Registers with x64 Assembly

1. Quadword
2. `.` and `:`
3. `.text`
4. `rax`
5. x87
6. `.asciz`

Chapter 7: Software Breakpoints

1. We set hardware breakpoints with debug registers, whereas we set software breakpoints by modifying the code that the inferior process is running. Software breakpoints are not limited, but hardware breakpoints can be used in certain security contexts where software breakpoints cannot.
2. `PTRACE_PEEKDATA` and `PTRACE_POKEDATA`
3. `int3`
4. By looking for "pie executable" or "shared object" in the output of the *file* utility
5. By calling the `personality` function with `ADDR_NO_RANDOMIZE`
6. The *maps* file

Chapter 8: Memory and Disassembly

1. They operate one word at a time.
2. To turn machine code into human-readable assembly
3. 15

Chapter 9: Hardware Breakpoints and Watchpoints

1. 4
2. `DR7`
3. Keep track of the value at the address, set a read/write stop point, and swallow any traps at which the value at the address has changed.
4. When the inferior process modifies its own code or attempts to detect software breakpoints

Chapter 10: Signals and Syscalls

1. `signal`
2. `SIGINT`
3. Functions able to be safely interrupted and resumed without issue
4. `setpgid`
5. `DR6`
6. `PTRACE_SYSCALL`

Chapter 11: Object Files

1. Executable, shared library, static library, core dump
2. ELF sections represent information relevant for linking; segments represent information relevant for loading and execution.
3. `mmap`
4. A list of null-terminated strings, where a string is represented by a byte offset into the string table
5. An array of identifier/value pairs that the operating system kernel uses to provide information about a process to user space

Chapter 12: Debug Information

1. DWARF information entry
2. No
3. `.debug_info`
4. To factor out common structure between DIEs
5. To encode variable-length integers in a space-efficient manner
6. Regular, base address selectors, and end-of-list indicators

Chapter 13: Line Tables

1. For space efficiency
2. Standard, extended, and special

3. To advance the current line and address and emit a matrix row, all in a single opcode
4. The program counter value to which the entry applies; the source file, line, and column that correspond to that program counter value; whether the entry represents a new statement, starts a basic block, ends a sequence, ends a function prologue, or begins a function epilogue; and a discriminator value

Chapter 14: Source-Level Breakpoints and Stepping

1. A compiler optimization where the body of the function is pasted into the call site
2. `DW_TAG_inlined_subroutine`
3. `rbp`, which the compiler may have chosen to use for other purposes

Chapter 15: Call Frame Information

1. Frame description entries (FDEs) and common information entries (CIEs)
2. So that the common parts can be reused
3. A specific point of the stack frame that we use as the base address for other computations
4. 16
5. It provides platform- and language-specific information and alternate data encoding schemes.
6. The SYSV ABI and LSB

Chapter 16: Stack Unwinding

1. `undefined`, `register(R)`, `same_value`, `offset(N)`, `val_offset(N)`, `expression(E)`, and `val_expression(E)`
2. Because a register value may not be restorable
3. `register_and_offset(R,N)` and `expression(E)`
4. Multiple logical frames may exist where there's only one physical frame.

Chapter 17: Shared Libraries

1. The segments of the ELF file
2. `INTERP`
3. To speed up certain syscalls by executing them in user space
4. Inside the dynamic linker
5. `NEEDED`
6. For the dynamic linker to communicate with tools that need to track the loading and unloading of shared libraries

7. By reading the `DEBUG` field of the `.dynamic` section
8. `_dl_debug_state`
9. To adjust references that are sensitive to where parts of the program are loaded, such that they point to the real virtual address of the relevant symbol
10. To enable updating references that reside in read-only sections and facilitate relocating symbols by updating only one location
11. Because there would be too many relocations to execute and instruction encodings may need to be changed
12. The act of delaying the resolution of a function's address until the function is called for the first time

Chapter 18: Multithreading

1. Threads can share a single virtual address space, set of file descriptors, and signal handlers, whereas processes cannot.
2. `clone`
3. `PTRACE_O_TRACECLONE`
4. `pthreads`
5. When the behavior of a program relies on the timing of two or more threads carrying out an action
6. The *task* directory

Chapter 19: DWARF Expressions

1. Address, register, implicit, empty, composite, and location lists
2. Range lists
3. `DW_OP_pick`
4. `DW_OP_call_ref`

Chapter 20: Variables and Types

1. The `DW_AT_bit_offset` attribute stores the distance in bits between the most significant bit of the aligned storage for the member and the most significant bit of the real data, whereas the `DW_AT_data_bit_offset` attribute stores the distance in bits from the start of the enclosing type to the start of the data.
2. With `DW_TAG_subrange_type` DIEs
3. To support multiple inheritance

Chapter 21: Expression Evaluation

1. The SYSV and Itanium ABIs
2. `rdi`

3. The number of vector registers used must be passed in the `al` register.
4. Only on returning from a function
5. Types that must be passed on the stack by invisible reference
6. In reverse lexical order
7. The address at the end of the stack arguments must be aligned to a 16-byte boundary.

GLOSSARY

address space layout randomization (ASLR) The randomization of the load position of a binary, carried out by the operating system as a security measure.

alignment An address is aligned to a particular boundary if the address is exactly divisible by the boundary size.

all-stop mode A debugger model for multithreaded programs where all threads are stopped if one thread is stopped.

application binary interface (ABI) A document that defines how pieces of machine code on a platform should communicate with one another. Includes calling conventions for functions and the layout of data.

assembler A tool for converting code written in assembly language into binary machine code.

assembly language A textual representation of machine code designed to be read and written by humans.

async-signal-safe A function that can be called within a signal handler. These functions either are reentrant or cannot be interrupted by another signal handler.

backtrace A visual representation of the current call stack.

big-endian A data representation for integers that stores the most significant byte at the smallest memory address and the least significant byte at the largest.

breakpoint Halts program execution when a particular piece of code is executed.

callee-saved registers Registers that a function must save before modifying and restore before returning. Defined by the ABI.

caller-saved registers Registers that a function can safely modify without having to save and restore. Defined by the ABI.

calling conventions The rules specified in an ABI for how arguments and return values are communicated between functions on a specific platform.

canonical frame address An address within a stack frame that is used as the base address for other frame address computations.

cat A lovable ball of fluff that lives in your house and makes a mess.

compiler A program that translates source code written in one programming language into machine code or another programming language.

compile unit A source file along with all of the header files it includes.

debug information Data that enables a debugger to relate machine code back to the source code that was used to produce it.

debug register A special-purpose hardware register used to create hardware breakpoints and watchpoints.

disassembler A tool that decodes binary machine instructions and converts them to human-readable assembly code.

DWARF information The debug information format used on Linux systems.

dyadic rational A floating-point number whose denominator is a power of two, thus making it directly expressible in binary.

dynamic linker A user-space program that is responsible for loading dynamic dependencies of programs.

dynamic linker rendezvous structure An area of memory that the dynamic linker uses to communicate with the kernel and debugger.

eightbyte A group of 8 bytes that is considered for the System V ABI type classification algorithms for calling conventions.

ELF file The object file format used on Linux systems.

endianness The order in which the bytes of an integer are stored on a platform.

exponent The location of the point in a floating-point number.

expression evaluation Running a source code expression inside the inferior process.

frame pointer A pointer to the beginning of the stack frame for the function that is currently executing.

function epilogue An area of code run at the end of a function's execution that tears down the call stack.

function inlining A compiler optimization where the body of a function is copied and pasted into the body of another function, eliminating the function call entirely.

function prologue An area of code at the beginning of a function that sets up the call stack.

global offset table (GOT) A table that stores the real offsets of symbols that are dynamically loaded.

hardware breakpoint A breakpoint set using architecture-specific debug registers.

hardware interrupt A signal that is sent to the operating system from the CPU on a variety of conditions, such as floating-point exceptions or memory access violations.

hardware registers Very small and fast areas of memory that are given specific names, such as the `rip` register that holds the current program counter value.

immediate operand An assembly language operand that encodes a constant value, such as the integer 42.

inferior A process that is being managed by a debugger.

instruction encoding The scheme by which machine code instructions are expressed in binary.

interrupt handler A function that is called when a specific hardware interrupt triggers.

interrupt vector table A data structure that allocates interrupt handlers for different types of hardware interrupts.

kernel The core part of an operating system that handles the most privileged operations.

kernel space Code that runs here is allowed to execute privileged operations not available to code that operates in user space.

lambda An anonymous function.

lazy binding The act of deferring the resolution of a function's address until the function is called for the first time.

little-endian A data representation for integers where the least significant byte is stored at the smallest address and the most significant byte at the largest.

load bias The difference between the address at which an object file is loaded in memory and the address assumed as the load address in the object file itself.

machine code/native code Binary-encoded instructions that run directly on a CPU.

mantissa The part of a floating-point number that represents its digits.

memory paging The splitting of memory into pages (usually 4KiB in size), which have their own memory protections and can be stored in external devices, being read into RAM when required.

mnemonic A human-readable name for a machine instruction.

move-only types C++ types that can't be copied but can be moved with move semantics.

move semantics A C++ feature that allows transferring resources between objects without making copies.

name mangling The process of turning the name of a program entity into a form that can be linked against. Commonly used for supporting function overloading and namespaces.

nondeterministic behavior Behavior that can't be predicted, potentially caused by a race condition.

non-stop mode A debugger model for multithreaded programs where some threads can continue executing while other threads are stopped for inspection.

object file A file that contains machine code and metadata that the operating system requires to execute that machine code.

opcode The part of a machine code instruction that indicates what kind of action to perform.

operand The part of a machine code instruction that indicates the data on which to perform the instruction.

position-independent executable (PIE) An executable that can be loaded at any virtual memory address.

POSIX A set of standards for compatibility between Unix-like operating systems.

preemption The process of an operating system halting the execution of a process so it can schedule the execution of another.

procedure linkage table (PLT) A table of function stubs that are relocated to jump to dynamically loaded functions.

process identifier (PID) A unique identifier for a running process, supplied by the operating system.

program counter/instruction pointer A hardware register that stores the memory address of the machine instruction that is currently being executed.

program loading The process by which a program is loaded from the filesystem into memory in order to be executed.

program stack A data structure that stores variables and information about the functions that a process is executing.

pure virtual function A C++ virtual function that must be overridden by a child class.

race condition A situation where the behavior of a program relies on the timing of two or more threads carrying out an action.

random access memory (RAM) An area of memory that is faster and smaller than external storage, but slower and larger than hardware registers.

reentrant Describes a function that can be safely executed concurrently without issue.

relocation The process of the static or dynamic linker fixing up memory references inside programs to point to their real virtual addresses.

resource acquisition is initialization (RAII) A C++ idiom in which the constructor for a class acquires the resources that an object needs and the destructor releases them.

section hashing Computing the hash of a binary section in order to detect changes to it at runtime.

section load bias The difference between the file offset and file address of an object file section.

signal A form of inter-process communication for notifying processes of events that occur in the system.

signal handler A function that is called when a process receives a particular signal.

sign extension The process of widening an integer to a larger byte size without changing its sign.

smart pointer A C++ type that acts like a pointer but manages the memory to which it points automatically, such as `std::unique_ptr`.

software breakpoint A breakpoint set by modifying the instructions of the running code to send a signal when a given machine instruction is executed.

stack frame An area of memory dedicated to a particular function invocation, in which it can store local variables, callee-saved register contents, and other information it needs to track.

stack pointer A hardware register that stores the address of the top of the program stack.

stack unwinding The process of re-creating the current program stack from a program counter value and set of registers.

stop point A breakpoint or watchpoint.

system call (syscall) A way for user-space programs to request privileged operations from the operating system kernel.

Unix Refers to a family of operating systems that are descended from the original Unix operating system. Includes operating systems based on the Linux kernel.

user area An area of memory that Linux systems dedicate to registers and metadata for running processes.

user space Code that runs here is not allowed to perform privileged operations that are reserved for the kernel, except through authorized channels such as system calls.

virtual dynamic shared object (vDSO) A shared library loaded into every user-space process's address space on Linux systems in order to make certain syscalls faster.

virtual memory A scheme for preserving memory safety across processes, abstracting the available storage of a machine, and presenting a linear view of memory that can be allocated to noncontiguous areas of physical memory, potentially on different devices.

watchpoint Similar to a breakpoint, but can be triggered when an address is read from or written to in addition to executed.

weak symbol A symbol in an object file that may be overridden by a stronger symbol.

zero extension The process of extending the byte width of an integer by filling the newly allocated space with zeros.

INDEX

D

E

F

G

H

I

K

L

M

R

S

T

U

V

W

X

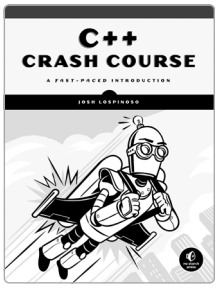
Y

Z

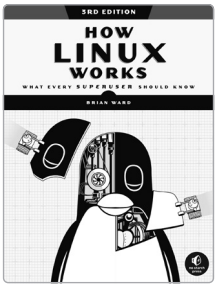
RESOURCES

Visit *https://nostarch.com/building-a-debugger* for errata and more information.

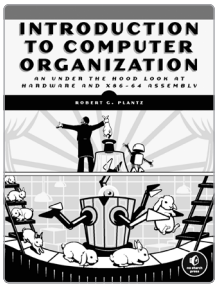
More no-nonsense books from **NO STARCH PRESS**



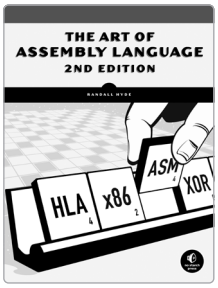
C++ CRASH COURSE
A Fast-Paced Introduction
BY JOSH LOSPINOSO
792 PP., $59.99
ISBN 978-1-59327-888-5



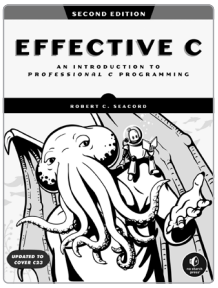
HOW LINUX WORKS, 3RD EDITION
What Every Superuser Should Know
BY BRIAN WARD
464 PP., $49.99
ISBN 978-1-7185-0040-2



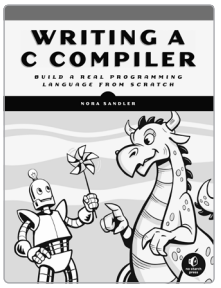
INTRODUCTION TO COMPUTER ORGANIZATION
An Under the Hood Look at Hardware and x86-64 Assembly
BY ROBERT G. PLANTZ
512 PP., $59.99
ISBN 978-1-7185-0009-9



THE ART OF ASSEMBLY LANGUAGE, 2ND EDITION
BY RANDALL HYDE
760 PP., $59.95
ISBN 978-1-59327-207-4



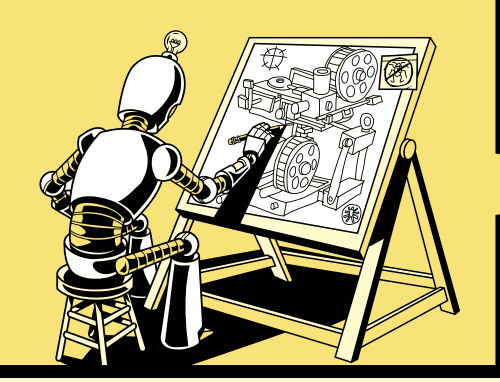
EFFECTIVE C, 2ND EDITION
An Introduction to Professional C Programming
BY ROBERT C. SEACORD
312 PP., $59.99
ISBN 978-1-7185-0412-7



WRITING A C COMPILER
Build a Real Programming Language from Scratch
BY NORA SANDLER
792 PP., $69.99
ISBN 978-1-7185-0042-6

PHONE:
800.420.7240 OR
415.863.9900

EMAIL:
SALES@NOSTARCH.COM
WEB:
WWW.NOSTARCH.COM



FROM BREAKPOINTS TO BACKTRACES

Discover what really happens when you set a breakpoint or step through code. *Building a Debugger* exposes the intricate inner workings of the systems you use every day. You'll learn the theory behind modern debugging tools and ultimately write a fully functional native debugger for x64 Linux systems.

As you build your debugger, you'll explore the Linux kernel, familiarize yourself with compiler tooling, learn just enough x64 assembly to be dangerous, and even write interpreters for debug information bytecode. You'll also enhance your debugger with features like shared library tracing, multithreading support, and register manipulation.

Your finished debugger will have the power to:

- Leverage the operating system, hardware, and platform conventions to inspect the state of a process
- Navigate the ELF executable format to understand a program's structure on the filesystem and in memory
- Interpret DWARF debugging information to map machine code back to source code
- Manipulate program execution with watchpoints, breakpoints, and step operations
- Locate and visualize variables in a running program
- Call functions inside of the debugged process on demand
- Unwind a program's call stack to generate backtraces

Whether you're a curious programmer, a reverse engineer, or a tool developer, *Building a Debugger* will give you valuable insight into how computers really function and make you a more effective programmer.

ABOUT THE AUTHOR

Sy Brand is Microsoft's C++ Developer Advocate with over a decade of experience in developer tooling, including debuggers, profilers, compilers, and language runtimes. They have contributed to the C++, DWARF, and HSA standards and hold a computer science degree from the University of St. Andrews, specializing in compiler implementation. Beyond tech, Sy is a poet, filmmaker, musician, activist, and parent to three cats and one human.

no starch press®

THE FINEST IN GEEK ENTERTAINMENT™

nostarch.com