1ST EDITION

Clang Compiler Frontend

Get to grips with the internals of a C/C++ compiler frontend and create your own tools

IVAN MURASHKO

Clang Compiler Frontend

Get to grips with the internals of a C/C++ compiler frontend and create your own tools

Ivan Murashko

‹packt›

Clang Compiler Frontend

Associate Group Product Manager: Kunal Sawant
Senior Editor: Rounak Kulkarni
Senior Content Development Editor: Rosal Colaco
Technical Editor: Jubit Pincy
Copy Editor: Safis Editing
Project Coordinator: Deeksha Thakkar
Indexer: Pratik Shirodkar
Production Designer: Vijay Kamble
Business Development Executive: Debadrita Chatterjee
Senior Developer Relations Marketing Executive: Shrinidhi Monaharan

First published: March 2024

Production reference: 1290224

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK

ISBN 978-1-83763-098-1

www.packtpub.com

Contributors

About the author

Ivan Murashko is a C++ software engineer. He earned his Ph.D. in Physics from Peter the Great St. Petersburg Polytechnic University and has over 20 years of C++ programming experience, mostly on Linux. Since 2020, he has worked with LLVM compilers and has been an LLVM committer since 2021. His areas of interest include the Clang compiler frontend, Clang Tools (such as Clang-Tidy and Clangd), and performance optimizations for compilers and compiler tools.

I want to thank my wife, Irina, who was patient and supported me throughout the writing of this book.

About the reviewer

Aditya Agrawal comes from the city of joy, Kolkata, West Bengal. He is currently working as a Software Engineer in the Systems Domain. He graduated with a master's degree in Computer Science from the reputed Indian Institute of Technology, Madras where he was introduced to the world of Compilers, Parallel Programming, and systems. Aditya has published a research paper that allows one to add point-to-point synchronizations for their OpenMP parallel programs (called UWPro). He has read a lot of books and tutorials on how to work with Compilers and the like. Aditya has experience working with RISCV during his tenure at MIPS Embedded Technologies as a full-time RISCV Developer. In his spare time, he loves to play video games and take part in various community events involving LLVM Social, RV Bangalore User Group, and so on.

Table of Contents

Preface

Low Level Virtual Machine (**LLVM**), is a collection of modular and reusable compiler and toolchain technologies used to develop compilers and compiler tools, such as linters and refactoring tools. LLVM is written in C++ and can be considered a good example of a well-structured project that uses interesting techniques aimed at making it reusable and efficient. The project can also be considered an excellent example of compiler architecture; diving into it will give you a sense of how compilers are organized and how they function. This should help to understand usage patterns and apply them accordingly.

One of the key components of LLVM is the C/C++ compiler known as Clang. This compiler is widely used across various companies and has been designated as the default compiler for certain development environments, notably for macOS development. Clang will be the primary focus of our investigation in this book, with particular attention to its frontend—the part that is closest to the C/C++ programming language. Specifically, the book will include examples demonstrating how the C++ standard is implemented within the compiler.

A pivotal aspect of LLVM's design is its modularity, which facilitates the creation of custom tools that exploit the compiler's comprehensive capabilities. A notable example covered in the book is the Clang-Tidy linter framework, designed to identify undesirable code patterns and recommend corrections. Although it includes several hundred checks, you may not find one specific to your project's needs. However, the book will provide you with the foundation necessary to develop such a check from the beginning.

LLVM is an actively evolving project with two major releases each year. At the time the book was written, the latest stable release was version 17. Meanwhile, a release candidate

for version 18 was introduced in January 2024, with its official release anticipated to coincide with the publication of the book. The book's content has been verified against the latest compiler version, 18, ensuring it provides insights based on the most current compiler implementation available.

Who this book is for

The book is written for C++ engineers who don't have prior knowledge of compilers but wish to gain this knowledge and apply it to their daily activities. It provides an overview of the Clang compiler frontend, an essential yet often underestimated part of LLVM. This section of the compiler, along with a collection of powerful tools, enables programmers to enhance code quality and the overall development process. For example, Clang-Tidy offers more than 500 different lint checks that detect anti-patterns in code (such as use after move) and help maintain code style and standards. Another notable tool is Clang-Format, which allows specifying various formatting rules suitable for your project. These tools can also be considered an integral part of the development process. For instance, the language server (Clangd) is a critical service providing navigation and refactoring support for your IDE.

Understanding compiler internals might be crucial for anyone wanting to create and use such tools. The book provides the necessary foundation to begin this journey, covering basic LLVM architecture and offering a detailed description of Clang internals. It includes examples from LLVM source code and custom tools that extend the basic functionality provided by the compiler. Additionally, the book addresses compilation databases and various performance optimizations that can enhance the build speed of your projects. This knowledge should help C++ developers correctly apply the compiler to their work activities.

What this book covers

Chapter 1, Environment Setup, describes the basic steps required to set up the environment for future experiments with Clang, suitable for Unix-based systems such as Linux and

Darwin (macOS). In addition, readers will learn how to download, configure, and build LLVM source code. We will also create a simple Clang Tool to verify the syntax of the provided source code.

Chapter 2, Clang Architecture, examines the internal architecture of the Clang compiler. Starting with the basic concept of a compiler, we will explore how it is implemented in Clang. We will look at various parts of the compiler, including the driver, preprocessor (lexer), and parser. We will also examine examples that show how the C++ standard is implemented in Clang.

Chapter 3, Clang AST, talks about **Clang Abstract Syntax Tree (AST)**, which is the basic data structure produced by the parser. We will explore how the AST is organized in Clang and how it can be traversed. We will also delve into AST Matchers — a powerful tool provided by Clang for locating specific AST nodes.

Chapter 4, Basic Libraries and Tools, explores basic LLVM libraries and tools, including the LLVM **Abstract Data Type (ADT)** library, used across all LLVM code. We will investigate TableGen, a **Domain-Specific Language (DSL)** used to generate C++ code in various parts of LLVM. Additionally, we will explore **LLVM Integrated Tester (LIT)** tool used for creating powerful end-to-end tests. Using the knowledge gained, we will create a simple Clang plugin to estimate source code complexity.

Chapter 5, Clang-Tidy Linter Framework, covers Clang-Tidy, a linter framework based on Clang AST, and creates a simple Clang-Tidy check. We will also discuss how compilation errors affect the AST and the results provided by different Clang Tools, such as Clang-Tidy.

Chapter 6, Advanced Code Analysis, goes further and considers another advanced data structure used for code analysis: **Control Flow Graphs (CFG)**. We will investigate typical cases for its application and create a simple Clang-Tidy check that utilizes this data structure.

Chapter 7, Refactoring Tools, Clang provides advanced tools for code modification and refactoring. We will explore different ways to create a custom refactoring tool, including one based on the Clang-Tidy linter framework. We will also explore Clang-Format, an extremely fast utility for automatic code formatting.

Chapter 8, IDE Support and Clangd, presents Clangd - a Language Server used in various IDEs, such as **Visual Studio Code (VS Code)**, to provide intelligent support, including navigation and code modification. Clangd exemplifies the utility of the powerful modular architecture of LLVM. It utilizes various Clang tools, such as Clang-Tidy and Clang-Format, to enhance the development experience in VS Code. Compiler performance is crucial for this tool, and we will explore several techniques Clangd employs to improve its performance, thereby offering the best experience to developers.

Appendix 1: Compilation Database, describes the Compilation Database—a method for providing complex compilation commands to different Clang Tools. This functionality is crucial for integrating Clang Tools such as Clangd and Clang-Tidy into real C/C++ projects.

Appendix 2: Build Speed Optimizations, covers several compiler performance optimizations that can be used to enhance compiler performance. We will cover Clang precompiled headers and Clang modules, which represent a serialized AST that can be loaded much faster than building it from scratch.

To get the most out of this book

You will need to have an understanding of C++, especially C++17, which is used for LLVM and throughout the examples in the book. The provided examples are assumed to be run on a Unix-like operating system, with Linux and Darwin (Mac OS) being considered the operating system requirements for the book. We will use Git to clone the LLVM source tree and start working on it. Some tools also need to be installed, such as CMake and Ninja, which will be actively used to build the examples and the LLVM source code.

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Clang-Compiler-Frontend-Packt`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, and user input. Here is an example: "The first two parameters specify the declaration `(clang::Decl)` and the statement for the declaration `(clang::Stmt)`."

A block of code is set as follows:

```
int main() {
  return 0;
}
```

Any command-line input or output is written as follows:

```
$ ninja clang
```

We use `<...>` as a placeholder for the folder where the LLVM source code was cloned.

Some code examples will be representing input of shells. You can recognize them by specific prompt characters:

- **`(lldb)`** for interactive LLDB shell
- $ for Bash shell (macOS and Linux)
- `>` for interactive shell provided by different Clang Tools, such as Clang-Query

Important note

Warnings or important notes appear like this.

Tip

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `https://www.packtpub.com/support/errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `https://partnerships.packt.com/contributors/`.

Share your thoughts

Once you've read *Clang Compiler Frontend*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:

`https://download.packt.com/free-ebook/9781837630981`

2. Submit your proof of purchase.
3. That's it! We'll send your free PDF and other benefits to your email directly.

Part 1

Clang Setup and Architecture

You can find some info about LLVM internal architecture and how Clang fits into it. There is also description how to install and build Clang and Clang-Tools, description for basic LLVM libraries and tools used across LLVM project and essential for Clang development. You can find description for some Clang features and their internal implementation.

This part has the following chapters:

- *Chapter 1, Basic Libraries and Tools*
- *Chapter 2, Clang Architecture*
- *Chapter 3, Clang AST*
- *Chapter 4, Basic Libraries and Tools*

1

Environment Setup

In this chapter, we will discuss the basic steps of setting up the environment for future experiments with Clang . The setup is appropriate for Unix-based systems such as Linux and Mac OS (Darwin). In addition, you will get important information on how to download, configure, and build the LLVM source code. We will continue with a short session that explains how to build and use the **LLVM debugger (LLDB**), which will be used as the primary tool for code investigation throughout the book. Finally, we will finish with a simple Clang tool that can check C/C++ files for compilation errors. We will use LLDB for a simple debug session for the created tool and clang internal. We will cover the following topics:

- Prerequisites
- Getting to know LLVM
- Source code compilation
- How to create a custom Clang tool

1.1 Technical requirements

Downloading and building LLVM code is very easy and does not require any paid tools. You will require the following:

- Unix-based OS (Linux, Darwin)
- Command line git
- Build tools: CMake and Ninja

We will use the debugger as the source investigation tool. LLVM has its own debugger, LLDB. We will build it as our first tool from LLVM monorepo: `https://github.com/llvm/llvm-project.git`.

Any build process consists of two steps. The first one is the project configuration and the last one is the build itself. LLVM uses CMake as a project configuration tool. It also can use a wide range of build tools, such as Unix Makefiles, and Ninja. It can also generate project files for popular IDEs such as Visual Studio and XCode. We are going to use Ninja as the build tool because it speeds up the build process, and most LLVM developers use it. You can find additional information about the tools here: `https://llvm.org/docs/GettingStarted.html`.

The source code for this chapter is located in the `chapter1` folder of the book's GitHub repository: `https://github.com/PacktPublishing/Clang-Compiler-Frontend-Packt/tree/main/chapter1`

1.1.1 CMake as project configuration tool

CMake is an open source, cross-platform build system generator. It has been used as the primary build system for LLVM since version 3.3, which was released in 2013.

Before LLVM began using CMake, it used autoconf, a tool that generates a configure script that can be used to build and install software on a wide range of Unix-like systems. However, autoconf has several limitations, such as being difficult to use and maintain and having poor support for cross-platform builds. CMake was chosen as an alternative to

autoconf because it addresses these limitations and is easier to use and maintain.

In addition to being used as the build system for LLVM, CMake is also used for many other software projects, including Qt, OpenCV, and Google Test.

1.1.2 Ninja as build tool

Ninja is a small build system with a focus on speed. It is designed to be used in conjunction with a build generator, such as CMake, which generates a build file that describes the build rules for a project.

One of the main advantages of Ninja is its speed. It is able to execute builds much faster than other build systems, such as Unix Makefiles, by only rebuilding the minimum set of files necessary to complete the build. This is because it keeps track of the dependencies between build targets and only rebuilds targets that are out of date.

Additionally, Ninja is simple and easy to use. It has a small and straightforward command-line interface, and the build files it uses are simple text files that are easy to read and understand.

Overall, Ninja is a good choice for build systems when speed is a concern, and when a simple and easy-to-use tool is desired.

One of the most useful Ninja option is `-j` . This option allows you to specify the number of threads to be run in parallel. You may want to specify the number depending on the hardware you are using.

Our next goal is to download the LLVM code and investigate the project structure. We also need to set up the necessary utilities for the build process and establish the environment for our future experiments with LLVM code. This will ensure that we have the tools and dependencies in place to proceed with our work efficiently.

1.2 Getting to know LLVM

Let's begin by covering some foundational information about LLVM, including the project history as well as its structure.

1.2.1 Short LLVM history

The Clang compiler is a part of the LLVM project. The project was started in 2000 by Chris Lattner and Vikram Adve as their project at the University of Illinois at Urbana–Champaign [26].

LLVM was originally designed to be a next-generation code generation infrastructure that could be used to build optimizing compilers for many programming languages. However, it has since evolved into a full-featured platform that can be used to build a wide variety of tools, including debuggers, profilers, and static analysis tools.

LLVM has been widely adopted in the software industry and is used by many companies and organizations to build a variety of tools and applications. It is also used in academic research and teaching and has inspired the development of similar projects in other fields.

The project received an additional boost when Apple hired Chris Lattner in 2005 and formed a team to work on LLVM. LLVM became an integral part of the development tools created by Apple (XCode).

Initially, **GNU Compile Collection (GCC)** was used as the C/C++ frontend for LLVM. But that had some problems. One of them was related to GNU **General Public License (GPL**) that prevented the frontend usage at some proprietary projects. Another disadvantage was the limited support for Objective-C in GCC at the time, which was important for Apple. The Clang project was started by Chris Lattner in 2006 to address the issues.

Clang was originally designed as a unified parser for the C family of languages, including C, Objective-C, C++, and Objective-C++. This unification was intended to simplify maintenance by using a single frontend implementation for multiple languages, rather than maintaining multiple implementations for each language. The project became successful very quickly. One of the primary reasons for the success of Clang and LLVM was their modularity. Everything in LLVM is a library, including Clang . It opened the opportunity to create a lot of amazing tools based on Clang and LLVM, such as clang-tidy and clangd, which will be covered later in the book (*Chapter 5, Clang-Tidy Linter Framework* and *Chapter 8, IDE Support and Clangd*).

LLVM and Clang have a very clear architecture and are written in C++. That makes it possible to be investigated and used by any C++ developer. We can see the huge community created around LLVM and the extremely fast growth of its usage.

1.2.2 OS support

We are planning to focus on OS for personal computers here, such as Linux, Darwin, and Windows. On the other hand, Clang is not limited by personal computers but can also be used to compile code for mobile platforms such as iOS and different embedded systems.

Linux

The GCC is the default set of dev tools on Linux, especially `gcc` (for C programs) and `g++` (for C++ programs) being the default compilers. Clang can also be used to compile source code on Linux. Moreover, it mimics to `gcc` and supports most of its options. LLVM support might be limited for some GNU tools, however; for instance, GNU Emacs does not support LLDB as a debugger. But despite this, Linux is the most suitable OS for LLVM development and investigation, thus we will mainly use this OS (Fedora 39) for future examples.

Darwin (macOS)

Clang is considered the main build tool for Darwin. The entire build infrastructure is based on LLVM, and Clang is the default C/C++ compiler. The developer tools, such as the debugger (LLDB), also come from LLVM. You can get the primary developer utilities from XCode, which are based on LLVM. However, you may need to install additional command-line tools, such as CMake and Ninja, either as separate packages or through package systems such as MacPorts or Homebrew.

For example, you can get CMake using Homebrew as follows:

```
$ brew install cmake
```

or for MacPorts:

```
$ sudo port install cmake
```

Windows

On Windows, Clang can be used as a command-line compiler or as part of a larger development environment such as Visual Studio. Clang on Windows includes support for the **Microsoft Visual C++ (MSVC)** ABI, so you can use Clang to compile programs that use the **Microsoft C runtime library (CRT)** and the C++ **Standard Template Library (STL)**. Clang also supports many of the same language features as GCC, so it can be used as a drop-in replacement for GCC on Windows in many cases.

It's worth mentioning `clang-cl` [9]. It is a command-line compiler driver for Clang that is designed to be used as a drop-in replacement for the MSVC compiler, `cl.exe` . It was introduced as part of the Clang compiler, and is created to be used with the LLVM toolchain.

Like `cl.exe` , `clang-cl` is designed to be used as part of the build process for Windows programs, and it supports many of the same command-line options as the MSVC compiler. It can be used to compile C, C++, and Objective-C code on Windows, and it can also be used to link object files and libraries to create executable programs or **dynamic link libraries (DLLs)**.

The development process for Windows is different from that of Unix-like systems, which require additional specifics that might make the book material quite complicated. To avoid this complexity, our primary goal is to focus on Unix-based systems, such as Linux and Darwin, and we will omit Windows-specific examples in this book.

1.2.3 LLVM/Clang project structure

The Clang source is a part of the LLVM **monolithic repository (monorepo)**. LLVM started to use the monorepo in 2019 as a part of its transition to Git [4]. The decision was

driven by several factors, such as better code reuse, improved efficiency, and collaboration. Thus you can find all the LLVM projects in one place. As mentioned in the Preface, we will be using LLVM version 18.x in this book. The following command will allow you to download it:

```
$ git clone https://github.com/llvm/llvm-project.git -b release/18.x
$ cd llvm-project
```

Figure 1.1: Getting the LLVM code base

Important note

The release 18 is the latest version of LLVM, expected to be released in March 2024. This book is based on the version from January 23, 2024, when the release branch was created.

The most important parts of the **llvm-project** that will be used in the book are shown in Figure 1.2.

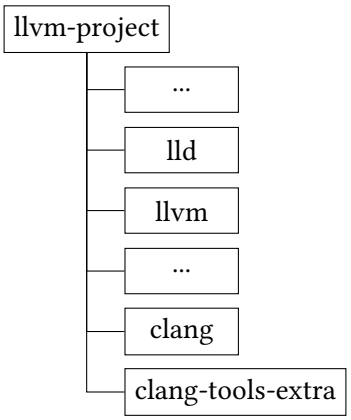

Figure 1.2: LLVM project tree

There are:

- `lld` : The LLVM linker tool. You may want to use it as a replacement for standard linker tools, such as GNU `ld`

- `llvm` : Common libraries for LLVM projects
- `clang` : The clang driver and frontend
- `clang-tools-extra` : These are different clang tools that will be covered in the second part of the book

Most projects have the structure shown in Figure 1.3.

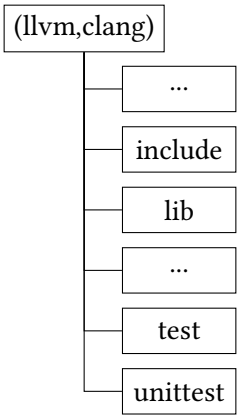

Figure 1.3: Typical LLVM project structure

LLVM projects, such as `clang` or `llvm` , typically contain two primary folders: `include` and `lib` . The `include` folder contains the project interfaces (header files), while the `lib` folder contains the implementation. Each LLVM project has a variety of different tests, which can be divided into two primary groups: unit tests located in the `unittests` folder and implemented using the Google Test framework, and end-to-end tests implemented using the **LLVM Integrated Tester (LIT)** framework. You can get more info about LLVM/Clang testing in *Section 4.5.2, LLVM test framework*.

The most important projects for us are `clang` and `clang-tools-extra` . The `clang` folder contains the frontend and driver.

Important note

The compiler driver is used to run different stages of compilation (parsing, optimization, link, and so on.). You can get more info about it at *Section 2.3, Clang driver overview.*

For instance, the lexer implementation is located in the `clang/lib/Lex` folder. You can also see the `clang/test` folder, which contains end-to-end tests, and the `clang/unittest` folder, which contains unit tests for the frontend and the driver.

Another important folder is `clang-tools-extra` . It contains some tools based on different Clang libraries. They are as follows:

- `clang-tools-extra/clangd` : A language server that provides navigation info for IDEs such as VSCode
- `clang-tools-extra/clang-tidy` : A powerful lint framework with several hundred different checks
- `clang-tools-extra/clang-format` : A code formatting tool

After obtaining the source code and setting up build tools, we are ready to compile the LLVM source code.

1.3 Source code compilation

We are compiling our source code in debug mode to make it suitable for future investigations with a debugger. We are using LLDB as the debugger. We will start with an overview of the build process and finish building the LLDB as a concrete example.

1.3.1 Configuration with CMake

Create a build folder where the compiler and related tools will be built:

```
$ mkdir build
$ cd build
```

The minimal configuration command looks like this:

```
$ cmake -DCMAKE_BUILD_TYPE=Debug ../llvm
```

The command requires the build type to be specified (e.g. `Debug` in our case) as well as the primary argument that points to a folder with the build configuration file. The configuration file is stored as `CMakeLists.txt` and is located in the `llvm` folder, which explains the `../llvm` argument usage. The command generates `Makefile` located in the build folder, thus you can use the simple `make` command to start the build process.

We will use more advanced configuration commands in the book. One of the commands looks like this:

```
cmake -G Ninja -DCMAKE_BUILD_TYPE=Debug -DCMAKE_INSTALL_PREFIX=../install
 ↪ -DLLVM_TARGETS_TO_BUILD="X86"
 ↪ -DLLVM_ENABLE_PROJECTS="lldb;clang;clang-tools-extra"
 ↪ -DLLVM_USE_SPLIT_DWARF=ON ../llvm
```

Figure 1.4: Basic CMake configuration

The are several LLVM/cmake options specified:

- `-G Ninja` specifies Ninja as the build generator, otherwise it will use make (which is slow).
- `-DCMAKE_BUILD_TYPE=Debug` sets the build mode. The build with debug info will be created. There is a primary build configuration for Clang internals investigations.
- `-DCMAKE_INSTALL_PREFIX=../install` specifies the installation folder.
- `-DLLVM_TARGETS_TO_BUILD="X86"` sets exact targets to be build. It will avoid building unnecessary targets.
- `-DLLVM_ENABLE_PROJECTS="lldb;clang;clang-tools-extra"` specifies the LLVM projects we want to build.
- `-DLLVM_USE_SPLIT_DWARF=ON` splits debug information into separate files. This option saves disk space as well as memory consumption during the LLVM build.