Clang compiler frontend and Clang tools

Understand internals of a top-rated C/C++ compiler frontend and create your own tools [1]

Ivan Murashko

Packt

BIRMINGHAM—MUMBAI

[1] This hasn't been finalized yet

Clang compiler frontend and Clang tools

Group Product Manager: TBD
Publishing Product Manager: Kunal Sawant
Content Development Editor: Rosal Colaco
Technical Editor: TBD
Copy Editor: Safis Editing
Project Coordinator: Manisha Singh
Proofreader: Safis Editing
Indexer: Subalakshmi Govindhan
Production Designer: Sinhayna Bais
Marketing Coordinator: TBD

First published: March 2023
Production reference: 1260522

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-80461-383-2

www.packt.com

Contributors

About the author

Ivan Murashko is a C++ software engineer with a PhD in physics from Peter the Great St. Petersburg Polytechnic University in Russia. His scientific interests include quantum optics, quantum information theory, and quantum computation. Ivan has more than 20 years of experience with C/C++ programming and has worked on projects involving cryptography, search engines, VoIP, and compilers. In addition to these areas, he also has an interest in functional languages such as Haskell and Scala, as well as category theory and logic. Since 2020, Ivan has been working with LLVM, focusing on the clang compiler frontend and code analysis and modification tools.

TBD[2]

About the reviewer

TBD has been developing software for nearly 20 years, of which Python has always been a major element. He's been a core developer of the Python language since 2010. Tal holds a B.Sc. in Math & Physics from Tel Aviv University. He likes hiking, computer games, philosophical sci-fi , and spending time with his family.

For the past eight years, Tal has been developing educational technology, first at Compedia where he built a group developing VR and AR education apps, and later at the startup FullProof of which he was a co-founder.

Tal currently works at Rhino Health, a startup working to the enable development and use of medical AI models with patient data from across the globe while preserving patient privacy.[3]

Share Your Thoughts

Once you've read Clang compiler frontend and Clang tools, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback.

[2] This is an example of dedication, please edit it.
[3] TBD

https://packt.link/r/1-804-61383-5

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

Replace NameOfTheProduct with your title name.

Replace the dummy QR Code image with your product QR code image.

Replace ISBN10P with the 10-digit ISBN-10P from EPIC.

This page should be added after the Front Matter.

Table Of Contents

Preface

The Clang is a C/C++ and Objective-C compiler that is an integral part of the LLVM (Low Level Virtual Machine) project. When we talk about Clang , we can refer to two different things. The first one is the compiler frontend, which is the part of the compiler responsible for parsing and performing semantic reasoning about the program. We also use the word Clang to refer to the compiler itself, which is also referred to as the compiler driver. The driver is responsible for invoking the compiler, which can be thought of as a manager that calls different parts of the compiler such as the compiler frontend and other parts necessary for successful compilation (middle-end, back-end, assembler, linker).

The book is mostly focused on the Clang compiler frontend, but it also includes some other relevant parts of LLVM that are critical for the frontend internals. The LLVM project evolves very fast, and some of its parts may be completely rewritten between different revisions. We will use a specific version of LLVM in the book - version 16.x, which was first released in March 2023 [Community, 2022c].

The Clang is compiler for C family of languages. Thus it supports such languages as C, Objective-C, C++ and Objective-C++. We are going mostly focus on C++ realisation. That assumes that we will refer C++ standard very often. Despite the fact that LLVM uses C++17 [for Standardization, 2017]for implementation it implements the latest version of standard and we will use C++20 version of standard [for Standardization, 2020] for references.

Who this book is for

This book is intended for experienced C++ software engineers who have no prior experience with compiler design, but who want to gain this knowledge and put it into practice. It may also be useful for engineers who want to learn about how Clang works, as well as its specific features such as performance improvements and modularity, which enables the creation of powerful custom compiler tools

What this book covers

The book is divided into two parts. The first one provides basic information about the LLVM project and how it can be installed. It also describes useful development tools and configurations used for exploring LLVM code later in the book. The internal Clang architecture is the next main topic in the first part of the book. Knowledge about the Clang internals and its place inside LLVM is essential for any development related to Clang . The Clang is also very good example of well designed software that can be used as a sample of good design pattern.

The final topic in the first part is compilation performance, particularly how it can be improved. We describe several Clang features that may significantly improve compilation speed, such as C++ modules, header maps, and others.

The Clang follows the primary paradigm of LLVM - everything is a library - which allows the creation of a variety of different tools. The second part of the book is about such tools. We discuss clang-tidy, a powerful framework for creating lint checks. We examine simple checks based on AST (abstract syntax tree) matching, as well as more powerful ones based on advanced techniques like CFG (control flow graph). The list of tools is not limited to code analysis, but also includes refactoring tools and IDE support.

Download the example code files

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Clang-Compiler-Frontend`. 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!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://static.packt-cdn.com/downloads/9781801071109_ColorImages.pdf`.

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: "Any attempt to run the code that has such issues will immediately cause the interpreter to fail, raising a `SyntaxError` exception."

A block of code is set as follows:

```
int main() {
```

```
  return 0;
}
```

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

```
$ python3 script.py
```

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

- »> for interactive Python shell
- $ for Bash shell (macOS and Linux)
- > for CMD or PowerShell (Windows)

Warnings or important notes appear like this.

Important note Warnings or important notes appear like this.

Tips and tricks appear like this.

Tips or 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 `www.packt.com/submit-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 `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `www.packt.com`.

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 required 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.

1

Environment setup

In this chapter we will discuss the basic steps to be done to setup the environment for future experiments with Clang . The setup is appropriate for Unix-based systems such as Linux and Mac OS (Darwin). In addition, the reader will get important information on how to download, configure, and build the LLVM source code. We will continue with a short session that shows you 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:

- Technical requirements
- Prerequisite
- Getting to known 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 debugger as the source investigation tool. LLVM has its own debugger - LLDB. We will build it as our first tool build from LLVM monorepo.

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 build configuration tool. It also can use wide range of different build tools such as Unix Makefiles, Ninja and so on. 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 as soon as it provides more faster build process and the most LLVM developers use it [Community, 2023d]

1.1.1 CMake as build 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 [Wikipedia contributors, 2022c], OpenCV [Wikipedia contributors, 2022b], Google Test [Wikipedia contributors, 2022a], and others.

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 commands 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 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 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 [Lattner and Adve, 2004].

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, GCC (GNU Compile Collection) was used as the C/C++ frontend for LLVM. But that had some disadvantages, 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 the Objective-C language 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 (see Chapter 5, clang-tidy linter framework and Chapter 8, IDE support and code navigation).

LLVM and Clang have a very clear architecture and are written in C++. That makes possible to investigate and use it by any C++ developer. As result we can see huge community created around LLVM and extremely fast grows 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. From other side the Clang usage is not limited by the personal computers but can also be used to compile code for mobile platforms such as iOS and different embedded systems.

1.2.2.1 Linux

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

1.2.2.2 Darwin

The Clang is considered as 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 LLVM-based. However, you may need to install additional command-line tools, such as CMake and Ninja, either as separate packages or through package systems like MacPorts or Homebrew. For example, you can get CMake using Homebrew as follows

```
$ brew install cmake
```

or for MacPorts:

```
$ sudo port install cmake
```

1.2.2.3 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 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` [Community", "2023"], it is a command-line compiler driver for Clang that is designed to be used as a drop-in replacement for the Microsoft Visual C++ (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 would 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 the book.

1.2.3 LLVM/clang project structure

The Clang source is a part of LLVM monorepo (monolithic repository). LLVM started to use monorepo in 2019 as a part of its transition to git [Community, 2019] (TBD - verify the statement). The decision was driven by number of factors such as better code reuse, improved efficiency and collaboration. Thus you can find all LLVM projects in one place. As seen in Preface, we will be using LLVM version 16.x in this book. The following command will allow you to download it:

```
$ git clone https://github.com/llvm/llvm-project.git -b release/16.x
$ cd llvm-project
```

The most important parts of the llvm-project that will be used in the book are shown in fig. 1.1. 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 project
- `clang` - the clang driver and frontend
- `clang-tools-extra` - there are different clang tools that will be covered at the second part of the book

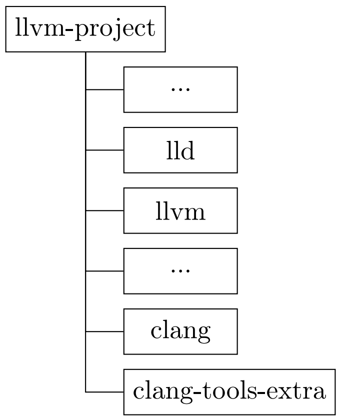


Figure 1.1: LLVM project tree: there are projects that are important for the book: lldb (LLDB debugger), llvm (some common libs are located here), clang (C/C++ compiler frontend and driver) and clang-tools-extra (different clang tools such as clang-tidy and clangd)

Most projects have the same structure shown in fig. 1.2.

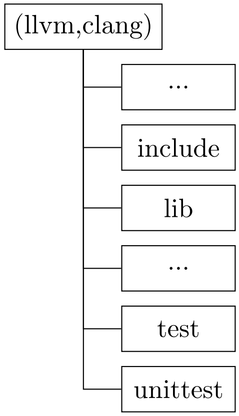


Figure 1.2: Typical LLVM project structure: the primary source code is located at include and lib folders. The test folder keeps LIT end-to-end tests and unittest folder keeps tests written for google test framework

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 4.3, LLVM test framework.

The most important projects for us are `clang` and `clang-tools-extra`. The `clang` folder contains the frontend and driver. [1] 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 driver.

Another important folder is `clang-tools-extra`. It contains some tools based on clang Abstract Syntax Tree (AST). There are:

- `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 hundreds of different checks.
- `clang-tools-extra/clang-format` - a code formatting tool.

[1] the compiler driver is used to run different stages of compilation (parsing, optimisation, link etc.). You can get more info about it at Section 2.2, Clang driver overview

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 with a concrete example of the build for LLDB .

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

```
$ 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 generate `Makefile` located at the build folder, thus you can use simple `make` command to start the build process.

We will use more advanced configuration commands in the book. The command is shown in fig. 1.3.

```
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 1.3: CMake: basic configuration used for the build at the book

The are several LLVM/cmake options specified:

- `-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 build unnecessary targets
- `-DLLVM_ENABLE_PROJECTS="lldb;clang;clang-tools-extra"` specifies LLVM projects that we care about
- `-DLLVM_USE_SPLIT_DWARF=ON` - spits debug information into separate files. This option saves disk space as well as memory consumption during the LLVM build.

We used `-DLLVM_USE_SPLIT_DWARF=ON` to save some space on the disc. For instance the clang build with the option enabled takes 20Gb[2] space but it takes 31Gb space with the option disabled. Note that the option requires compiler used for clang build to support it. You might also notice that we create the build for one specific architecture - `"X86"`. This option also saved some space for us because otherwise all supported architecture will be built and the required space will increase from 20GB to 27Gb.

You can save more space if you will use dynamic libraries instead of static ones. `-DBUILD_SHARED_LIBS=ON` will build each LLVM component as shared library. The used space will be 14Gb at the case and the overall config command will look 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 -DBUILD_SHARED_LIBS=ON ../llvm
```

For performance purposes on Linux you might want to use `gold` linker instead of default one. The `gold` linker is an alternative to the GNU Linker that was developed as part of the GNU Binary Utilities (binutils) package. It is designed to be faster and more efficient than the GNU Linker, especially when linking large projects. One way it achieves this is by using a more efficient algorithm for symbol resolution and a more compact file format for the resulting executable. It can be enabled with `-DLLVM_USE_LINKER=gold` option. The result configuration command will look like.

```
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_LINKER=gold -DLLVM_USE_SPLIT_DWARF=ON -DBUILD_SHARED_LIBS=ON
 ↪ ../llvm
```

[2] The configuration specified at fig. 1.3 was used for the build. The build was run as `ninja clang`

The debug build can be very slow and you may want to consider an alternative. A good compromise between debuggability and performance is the release build with debug information. To obtain this build, you can change the `CMAKE_BUILD_TYPE` flag to `RelWithDebInfo` in your overall configuration command. The command will then look like this:

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

The following table keeps the list of some popular options [Community, 2023].

Option	Description
`CMAKE_BUILD_TYPE`	Specifies build configuration. Possible values are `Release|Debug|RelWithDebInfo|MinSizeRel` . The `Release` and `RelWithDebInfo` are optimized for performance, `MinSizeRel` is optimized for size.
`CMAKE_INSTALL_PREFIX`	Installation prefix
`CMAKE_C,CXX_FLAGS`	Extra C/C++ flags be used for compilation
`CMAKE_C,CXX_COMPILER`	C/C++ compiler be used for compilation. You might want to specify a non-default compiler to use some options that are not available or not supported by the default compiler
`LLVM_ENABLE_PROJECTS`	The projects to be enabled. We will use `clang;clang-tools-extra`
`LLVM_USE_LINKER`	Specifies the linker be used. There are several options that include `gold` and `lld`

Table 1.1: Configuration options

1.3.2 Build

We need to call Ninja to build the desired project. The command for Clang build will look like

```
$ ninja clang
```

You can also run unit and end-to-end tests for the compiler with

```
$ ninja check-clang
```

The compiler binary can be found as `bin/clang` in the `build` folder.

You can also install the binaries into the folder specified with `-DCMAKE_INSTALL_PREFIX` option. It can be done as follows

```
$ ninja install clang
```

The folder `../install` (specified as the installation folder at fig. 1.3) will have the following structure

```
$ ls ../install
bin  include  lib  libexec  share
```

1.3.3 The LLVM debugger, its build and usage

The LLVM debugger, LLDB , has been created with a look at GDB (GNU debugger). Some of its commands repeat the counterparts from GDB . You may ask the question: "Why do we need a new debugger if we have a good one?" The answer can be found in the different architecture solutions used by GCC and LLVM. LLVM uses a modular architecture, and different parts of the compiler can be reused. For example, the Clang frontend can be reused in the debugger, resulting in actual support for modern C/C++ features. For example, the print command in `lldb` can specify any valid language constructions and you can use some modern C++ features with the `lldb` print command.

In contrast, GCC uses a monolithic architecture, and it's hard to separate the C/C++ frontend from other parts. Therefore, GDB has to implement language features separately, which may take some time before modern language features implemented in GCC become available in GDB .

You may find some info about LLDB build and typical usage scenario in the following example. We are going to create a separate folder for release build

```
$ cd llvm-project
$ mkdir release
$ cd release
```

We configure our project in Release mode and specify the `lldb` and `clang` projects only

```
cmake -G Ninja -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=../install
 ↪  -DLLVM_TARGETS_TO_BUILD="X86" -DLLVM_ENABLE_PROJECTS="lldb;clang" ../llvm
```

We are going to build both Clang and LLDB using no more than 4 concurrent processes

```
$ ninja clang lldb -j4
```

You can install the created executables with the following command

```
$ ninja install-clang install-lldb
```

The binary will be installed into the folder specified via `-DCMAKE_INSTALL_PREFIX` config command argument.

```
1 $ llvm-project/install/bin/lldb main
2 (lldb) target create "./main"
3 ...
4 (lldb) b main
5 Breakpoint 1: where = main`main + 11 at main.cpp:2:3,...
6 (lldb) r
7 Process 1443051 launched: ...
8 Process 1443051 stopped
9 * thread #1, name = 'main', stop reason = breakpoint 1.1
10     frame #0: 0x000055555555513b main`main at main.cpp:2:3
11    1    int main() {
12 -> 2      return 0;
13    3    }
14 (lldb) q
```

Figure 1.4: LLDB session example

We will use the following simple C++ program for the example debugger session

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

The program can be compiled using the following command

`llvm-project/install/bin/clang main.cpp -o main -g -O0`

As you may notice we don't use optimization (`-O0` option) and store debug info at the binary (`-g` option).

Typical debug session for the created executable is shown in fig. 1.4. There are several actions done:

- Run the debug session with `llvm-project/install/bin/lldb src/main`, where `main` is the exacutable we want to debug, see fig. 1.4, line 1.
- We set breakpoint at the `main` function, see fig. 1.4, line 4.
- Run the session with `r` command, see fig. 1.4, line 6.

- We can see that the process is interrupted at the breakpoint, see fig. 1.4, line 8, 12.
- We finish the session with `q` command, see fig. 1.4, line 14.

We are going to use LLDB as one of our tools for the Clang internals investigation. We will use the same sequence of commands that is shown in fig. 1.4. You can also use another debugger, such as GDB , that has a similar set of commands as LLDB .

1.4 Test project: syntax check with clang tool

For our first test project we will create a simple clang tool that runs compiler and checks syntax for provided source file. We will create so called out-of-tree LLVM project i.e. the project that will use LLVM but will be located outside the main LLVM source tree.

There are several actions to be done to create the project:

- The required LLVM libraries and headers have to be built and installed
- We have to create a build configuration file for our test project
- The source code that uses LLVM has to be created

We will start with the first step and will install clang support libraries and headers. We will use the following configuration command for CMake:

```
cmake -G Ninja -DCMAKE_BUILD_TYPE=Debug -DCMAKE_INSTALL_PREFIX=../install
↪  -DLLVM_TARGETS_TO_BUILD="X86" -DLLVM_ENABLE_PROJECTS="clang"
↪  -DLLVM_USE_LINKER=gold -DLLVM_USE_SPLIT_DWARF=ON -DBUILD_SHARED_LIBS=ON
↪  ../llvm
```

Figure 1.5: LLVM CMake configuration for simple syntax check clang tool

As you may notice, we enabled only one project: `clang`, all other options are standard for our debug build. The command has to be run from a created `build` folder inside LLVM source tree, as it was suggested at Section 1.3.1, Configuration with CMake.

The required libraries and headers can be installed with the following command

```
$ ninja install
```

The installation will be done into `install` folder as it was specified at `CMAKE_INSTALL_PREFIX` option.

We have to create two files for our project:

- `CMakeLists.txt` the build configuration file
- `TestProject.cpp` the project source code

The configuration file will accept a path to the LLVM install folder via `LLVM_HOME` environment variable. The configuration file is the following:

```
1 cmake_minimum_required(VERSION 3.16)
2 project("syntax-check")
3
4 if ( NOT DEFINED ENV{LLVM_HOME})
5   message(FATAL_ERROR "$LLVM_HOME is not defined")
6 else()
7   message(STATUS "$LLVM_HOME found: $ENV{LLVM_HOME}")
8   set(LLVM_HOME $ENV{LLVM_HOME} CACHE PATH "Root of LLVM installation")
9   set(LLVM_LIB ${LLVM_HOME}/lib)
10  set(LLVM_DIR ${LLVM_LIB}/cmake/llvm)
11  find_package(LLVM REQUIRED CONFIG)
12  include_directories(${LLVM_INCLUDE_DIRS})
13  link_directories(${LLVM_LIBRARY_DIRS})
14  set(SOURCE_FILES SyntaxCheck.cpp)
15  add_executable(syntax-check ${SOURCE_FILES})
16  set_target_properties(syntax-check PROPERTIES COMPILE_FLAGS "-fno-rtti")
17  target_link_libraries(syntax-check
18    LLVMSupport
19    clangBasic
20    clangFrontend
21    clangSerialization
22    clangTooling
23  )
24 endif()
```

Figure 1.6: CMake file for simple syntax check clang tool

The most important parts of the file are

- Line 2: We specify the project name (syntax-check). That also be the name for our executable

- Lines 4-7: Test for `LLVM_HOME` environment variable
- Line 10: We set a path to LLVM CMake helpers
- Line 11: We load LLVM CMake package from the paths specified at line 10
- Line 14: We specify our source file that should be compiled
- Line 16: We setup an additional flag for compilation: `-fno-rtti`. The flag is required as soon as LLVM is built without RTTI. This is done in an effort to reduce code and executable size [Community, 2023b].
- Line 18-22 We specify the required libraries to be linked to our program

The source code for our tool is the following

```cpp
#include "clang/Frontend/FrontendActions.h" // clang::SyntaxOnlyAction
#include "clang/Tooling/CommonOptionsParser.h"
#include "clang/Tooling/Tooling.h"
#include "llvm/Support/CommandLine.h" // llvm::cl::extrahelp

namespace {
llvm::cl::OptionCategory TestCategory("Test project");
llvm::cl::extrahelp
    CommonHelp(clang::tooling::CommonOptionsParser::HelpMessage);
} // namespace

int main(int argc, const char **argv) {
  llvm::Expected<clang::tooling::CommonOptionsParser> OptionsParser =
      clang::tooling::CommonOptionsParser::create(argc, argv, TestCategory);
  if (!OptionsParser) {
    llvm::errs() << OptionsParser.takeError();
    return 1;
  }
  clang::tooling::ClangTool Tool(OptionsParser->getCompilations(),
                                 OptionsParser->getSourcePathList());
  return Tool.run(
      clang::tooling::newFrontendActionFactory<clang::SyntaxOnlyAction>()
          .get());
}
```

The most important part of the file are

- Line 7-9: The majority of compiler tools have the same set of command line arguments. LLVM command line library [Community, 2023c] provides some API to process compiler command options. We setup the library at line 7. We also setup additional help message at lines 8-9.
- Line 13-18: We parse command line arguments
- Line 19-23: We create and run our clang tool
- Line 22: We use `clang::SyntaxOnlyAction` frontend action that will run syntax and semantic check on the input file. You can get more info about frontend actions later at Section 2.3.1, Frontend action.

We have to specify a path to LLVM `install` folder to build our tool. As it was mentioned above, the path has to be specified via `LLVM_HOME` environment variable. Our configuration command (see fig. 1.5) specifies the path as `install` folder inside LLVM project source tree. Thus we can build our tool as follows

```
export LLVM_HOME=<...>/llvm-project/install
mkdir build
cd build
cmake -G Ninja ..
ninja
```

We can run the tool as follows

```
$ ./build/syntax-check --help
USAGE: syntax-check [options] <source0> [... <sourceN>]
...
```

The program will successively terminate if we run it on a valid C++ source file, but it will produce an error message if it's run on a broken C++ file:

```
$ ./syntax-check mainbroken.cpp
...
Running without flags.
mainbroken.cpp:2:11: error: expected ';' after return statement
  return 0
          ^
          ;
1 error generated.
mainbroken.cpp.
```

We can also run our tool under LLDB debugger.

```
1 $  llvm-project/install/bin/lldb ./syntax-check -- main.cpp
2 ...
3 (lldb) b clang::ParseAST
4 ...
5 (lldb) r
6 ...
7 Running without flags.
8 Process 608249 stopped
9 * thread #1, name = 'syntax-check', stop reason = breakpoint 1.1
10     frame #0: ... clang::ParseAST(...) at ParseAST.cpp:116:3
11    113
12    114         void clang::ParseAST(...) {
13    115           // Collect global stats on Decls/Stmts ...
14 -> 116           if (PrintStats) {
15    117             Decl::EnableStatistics();
16    118             Stmt::EnableStatistics();
17    119           }
18 (lldb) c
19 Process 608249 resuming
20 Process 608249 exited with status = 0 (0x00000000)
21 (lldb)
```

Figure 1.7: LLDB session for clang tool test project

We run `syntax-check` as the primary binary and set `main.cpp` source file as an argument for the tool (section 1.4, line 1). We also set a breakpoint at `clang::ParseAST` function (section 1.4, line 3). The function is the primary entry point for source code parsing. We run program at line 5 and continue the execution after breakpoint at line 18.

We will use the same debugging techniques later in the book when we will investigate clang source code.

1.5 Summary

In the chapter, we covered the history of the LLVM project, obtained the source code for LLVM, and explored its internal structure. We learned about the tools used to build LLVM, such as CMake and Ninja. We studied the various configuration options for building LLVM and how they can be used to optimize resources, including disk space. We built Clang and LLDB in release mode and used the resulting tools to compile a basic program and run it under the debugger. We also created a simple clang tool and run it under LLDB debugger.

The next chapter will introduce you to compiler design architecture and how it appears in the context of Clang . We will primarily focus on the Clang frontend, but we will also cover the important concept of the Clang driver - the backbone that manages all stages of the compilation process, from parsing to linking.

1.6 Further reading

- Getting Started with the LLVM System, https://llvm.org/docs/GettingStarted.html
- Building LLVM with CMake, https://llvm.org/docs/CMake.html
- Clang Compiler User's Manual, https://clang.llvm.org/docs/UsersManual.html

2

Clang architecture

In this chapter, we will examine the internal architecture of Clang and its relationship with other LLVM components. We will begin with an overview of the overall compiler architecture, with a specific focus on the clang driver. As the backbone of the compiler, the driver runs all compilation phases and controls their execution. Finally, we will concentrate on the frontend portion of the Clang compiler, which includes lexical and semantic analysis, and produces an Abstract Syntax Tree (AST) as its primary output. The AST forms the foundation for most clang tools, and we will examine it more closely in the next chapters.

The following topics will be covered in this chapter:

- Compiler overview
- Clang driver overview, including an explanation of the compilation phases and their execution
- Clang frontend overview, which covers frontend actions, preprocessor, parser, and sema

2.1 Compilers overview

Despite the fact that compilers are used to translate programs from one form to another, they can also be considered as large software systems that use various algorithms and data structures. The knowledge obtained from studying compilers is not specific to compilers

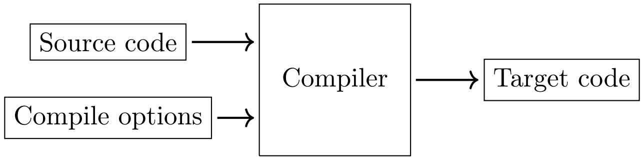


Figure 2.1: Compiler takes source code and compile options and transform them into a code on the target platform

and can be applied to other software as well. On the other hand, compilers are also a subject of active scientific research, and there are many unexplored areas and topics to investigate.

You can find some basic information about the internal structure of a compiler here. We will keep it as basic as possible so the information is applicable to any compiler, not just Clang. We will briefly cover all phases of compilation, which will help to understand Clang's position in the overall compiler architecture.

2.1.1 Compiler workflow

The primary function of a compiler is to convert a program written in a specific programming language (such as C/C++ or FORTRAN) into a format that can be executed on a target platform. This process involves the use of a compiler, which takes the source file and any compilation flags, and produces a build artifact, such as an executable or object file, as shown in fig. 2.1. The term "target platform" can have a broad meaning. It can refer to machine code that is executed on the same host, as is typically the case. But it can also refer to cross-compilation, where the compiler generates code for a different computer architecture than the host. For example, code for a mobile application or embedded application running on ARM can be generated using an Intel machine as the host. Additionally, the target platform is not limited to machine code only. For example, some early C++ compilers (such as "cc") would produce pure C code as output. This was done because, at the time, C was the most widely used and well-established programming language, and the C compiler was the most reliable way to generate machine code. This approach allowed early C++ programs to be run on a wide range of platforms since most systems already had a C compiler available. The produced C code could then be compiled into binary by another compiler.

We are going to focus on compilers that produce binary code, and a typical compiler workflow for such a compiler is shown in fig. 2.2. The stages of compilation can be described as follows:

- Frontend: The Frontend does lexical analysis and parsing, which includes both syntax analysis and semantic analysis. The syntax analysis assumes that your program is

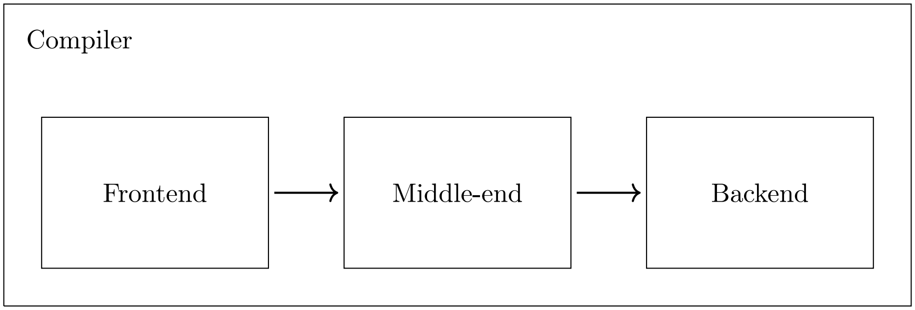


Figure 2.2: Typical compiler workflow: source program is passed via different stages: frontend, middle-end and backend

well-organized according to the language grammar rules. The semantic analysis performs checks on the program's meaning and rejects invalid programs, such as those that use wrong types.

- Middle-end: The Middle-end performs various optimizations on the intermediate representation (IR) code (LLVM-IR for clang).

- Backend: The Backend of a compiler takes the optimized or transformed IR and generates machine code or assembly code that can be executed by the target platform.

The source program is transformed into different forms as it passes through the various stages. For example, the Frontend produces IR code, which is then optimized by the Middle-end and finally converted into native code by the Backend (see fig. 2.3).

2.1.2 Frontend

Our primary focus will be on the frontend, so we will examine its components. The frontend also transforms the source code into various forms before it produces the IR. The first component of the frontend is the Lexer (see fig. 2.4). It converts the source code into a set of tokens, which are used to create a special data structure called the abstract syntax tree (AST). The final component, code generator (Codegen), traverses the AST and generates the IR from it.

We will use a simple C/C++ program that calculates the maximum of two numbers to demonstrate the workings of the Frontend. The code for the program is as follows:

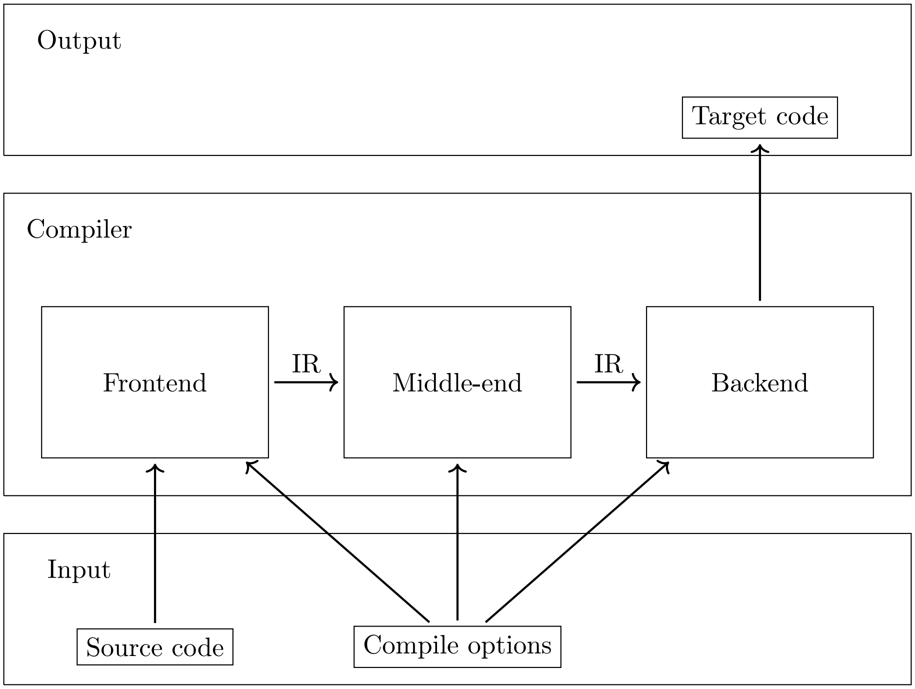


Figure 2.3: Source code transformation by compiler: Input data consists of Source code and Compile options. The source code is transformed by Frontend into IR (Intermediate representation). Middle-end does different optimizations on IR and passes the final (optimazed) result to Backend. Backend generates the Target code. Frontend, Middle-end and Backend use Compile options as setting for the code transformations

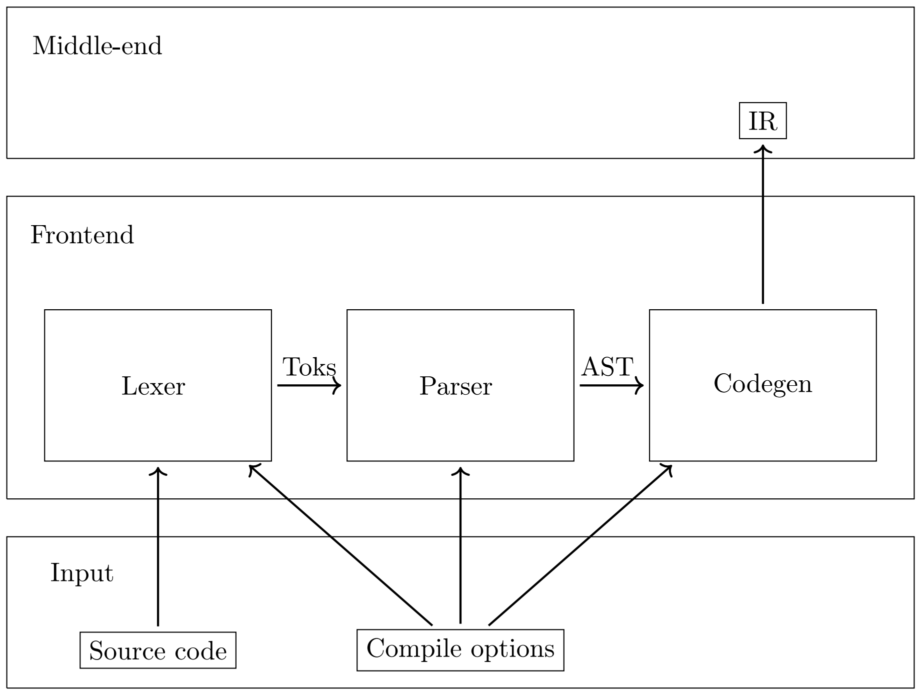


Figure 2.4: Compiler frontend: source code is transformed into a set of tokens (Toks) by Lexer . Parser takes the tokens and creates Abstract syntax tree (AST). Codegen generates IR from AST

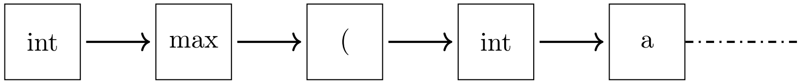


Figure 2.6: Lexer : the program source is converted into a stream of tokens

```
int max(int a, int b) {
  if (a > b)
    return a;
  return b;
}
```

Figure 2.5: Test program for compiler frontend investigations

2.1.2.1 Lexer

The Frontend process starts with the Lexer , which converts the input source into a stream of tokens. In our example program (see fig. 2.5), the first token is the keyword `int` , which represents the integer type. This is followed by the identifier `max` for the function name. The next token is the left parenthesis `(` , and so on (see fig. 2.6).

2.1.2.2 Parser

The Parser is the next component following the Lexer . The primary output produced by the Parser is called as abstract syntax tree (AST). This tree represents the abstract syntactic structure of the source code written in a programming language. The Parser generates the AST by taking the stream of tokens produced by the Lexer as input and organizing them into a tree-like structure. Each node in the tree represents a construct in the source code, such as a statement or expression, and the edges between nodes represent the relationships between these constructs.

The AST for our example program is shown in fig. 2.7. As you can see, our function (`max`) has two parameters (`a` and `b`) and a body. The body is marked as a compound statement in fig. 2.7. The compound statement consists of other statements, such as `return` and `if` . The variables `a` and `b` are used in the bodies of these statements. You may also be interested in the real AST generated by Clang for the compound statement, the result of which is shown in fig. 2.8.

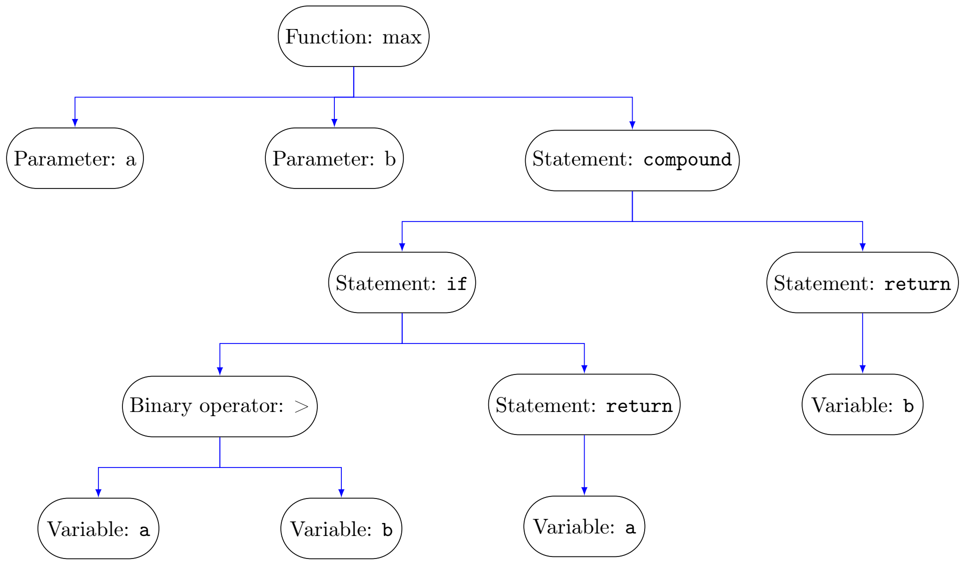


Figure 2.7: The AST for our example program that calculates maximum of 2 numbers

The Parser performs two activities:

1. Syntax analysis: the Parser constructs the AST by analyzing the syntax of the program.
2. Semantic analysis: the Parser analyzes the program semantically.

One of the primary goals of analysis is error detection, and the Parser produces an error message if it fails in syntax or semantic analysis. We can get a sense of this by considering what types of errors are detected by syntax analysis and which ones are detected by semantic analysis.

Syntax analysis assumes that the program should be correct in terms of the grammar specified for the language. For example, the following program is invalid in terms of syntax because a semicolon is missing at the last return statement:

```
int max(int a, int b) {
  if (a > b)
    return a;
  return b // missing ;
}
```

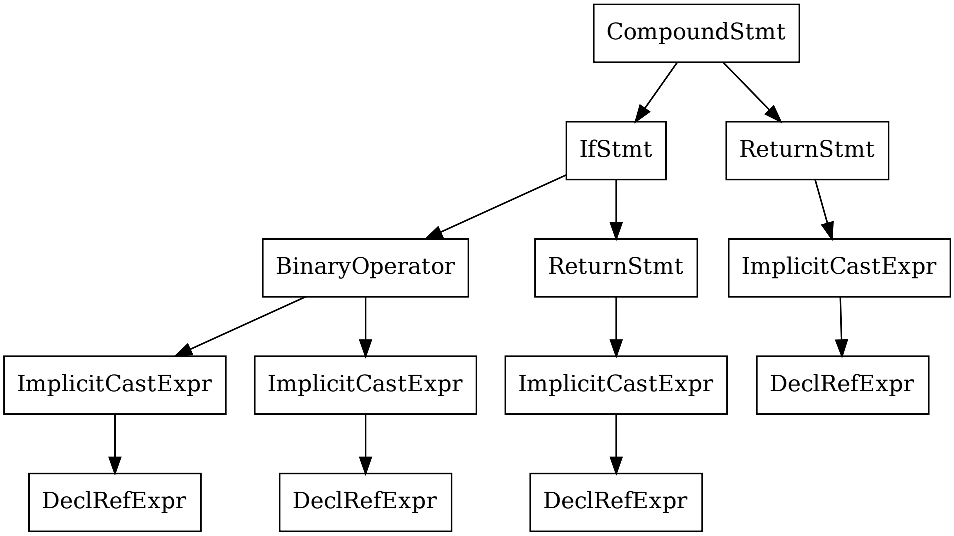


Figure 2.8: The AST for compound statement generated by Clang . The tree generated by *clang -cc1 -ast-view <...>* command

Clang produces the following output for the program:

```
max_invalid_syntax.cpp:4:11: error: expected ';' after return statement
  return b // missing ;
          ^
          ;
```

On the other hand, a program can be syntactically correct but have no sense. The Parser should detect a semantic error in such cases. For instance, the following program has a semantic error related to the wrongly used type for the return value:

```cpp
int max(int a, int b) {
  if (a > b)
    return a;
  return &b; // invalid return type
}
```

Clang produces the following output for the program:

```
max_invalid_sema.cpp:4:10: error: cannot initialize return object of type \
'int' with an rvalue of type 'int *'
  return &b; // invalid return type
         ^~
```

AST is mainly constructed as a result of syntax analysis, but for certain languages, such as C++, semantic analysis is also crucial for constructing the AST, particularly for C++ template instantiation.

During syntax analysis, the compiler verifies that the template declaration adheres to the language's grammar and syntax rules, including the proper use of keywords such as "template" and "typename", as well as the formation of the template parameters and body.

Semantic analysis, on the other hand, involves the compiler performing template instantiation, which generates the AST for specific instances of the template. It's worth noting that the semantic analysis of templates can be quite complex, as the compiler must perform tasks such as type checking, name resolution, and more for each template instantiation. Additionally, the instantiation process can be recursive and lead to a significant amount of code duplication, known as code bloat. To combat this, C++ compilers employ techniques such as template instantiation caching to minimize the amount of redundant code generated.

2.1.2.3 Codegen

The Codegen [1] or code generator, which is the final component of the compiler's frontend, has the primary goal of generating the Intermediate Representation (IR). For this purpose, the compiler traverses the AST generated by the parser and converts it into another source code that is called the Intermediate Representation or IR. The IR is a language-independent representation, allowing the same middle-end component to be used for different frontends (FORTRAN vs C++).

The use of Intermediate Representations (IRs) in compilers is a concept that has been around for several decades. The idea of using an intermediate representation to represent the source code of a program during compilation has evolved over time, and the exact date when IR was first introduced in compilers is not clear.

However, it is known that the first compilers in the 1950s and 1960s did not use IRs and instead translated source code directly into machine code. By the 1960s and 1970s, researchers had begun experimenting with using IRs in compilers to improve the efficiency and flexibility of the compilation process.

One of the first widely used IRs was the three-address code, which was used in the mid-1960s in IBM/360's FORTRAN compiler. Other early examples of IRs include the register transfer language (RTL) and the static single assignment (SSA) form, which were introduced in the 1970s and 1980s respectively.

Today, the use of IRs in compilers is a standard practice, and many compilers use multiple IRs throughout the compilation process. This allows for more powerful optimization and code generation techniques to be applied.

2.2 Clang driver overview

When discussing compilers, we typically refer to a command-line utility that initiates and manages the compilation process. For example, to use the GNU Compiler Collection, one must call `gcc` to start the compilation process. Similarly, to compile a C++ program using Clang, one must call `clang` as the compiler. The program that controls the compilation process is known as the driver. The driver coordinates different stages of compilation and connects them together. In the book, we will be focusing on LLVM and using clang as the driver for the compilation process.

It may be confusing for readers that the same word, "clang," is used to refer to both the compiler front-end and the compilation driver. In contrast, with GCC, the driver and C++ compiler are separate executables [2]. However, "clang" is a single executable that functions as both the driver and the compiler front-end. To use Clang as the compiler front-end only,

[1] It's worth mentioning that we also have another Codegen component as a part of Backend that generate the target code.

[2] The C/C++ compiler in GCC is a separate executable called "cc"

the special option `-cc1` must be passed to it.

2.2.1 Example program

We will use the simple "Hello world!" example program for our experiments with clang driver. The main source file is called as `hello.cpp`. The file implements a trivial C++ program that prints "Hello world!" to the standard output:

```cpp
#include <iostream>

int main() {
  std::cout << "Hello world!" << std::endl;
  return 0;
}
```

You can compile the source with

```
$ clang hello.cpp -o /tmp/hello -lstdc++
```

Figure 2.9: Compilation for *hello.cpp* : We use *clang* executable as the compiler. We also use standard C++ library i.e. we link the executable with *-lstdc++* . The result is stored at */tmp/hello*

As you may see, we specified `-lstc++` library option because we used <`iostream`> header from the standard C++ library. We also specified the output for executable (`/tmp/hello`) with `-o` option.

2.2.2 Compilation phases

We used 2 inputs for our example program. The first one is our source code, the second one is a shared library for standard C++ library. The clang driver should combine the inputs together, pass them via different phases of compilation process and finally provide the executable file on the target platform.

Clang uses the same process as shown in fig. 2.2. You can ask Clang to show the phases using `-ccc-print-phases` additional argument

```
$ clang hello.cpp -o /tmp/hello -lstdc++ -ccc-print-phases
```

The output for the command is the following

```
              +- 0: input, "hello.cpp", c++
         +- 1: preprocessor, {0}, c++-cpp-output
      +- 2: compiler, {1}, ir
   +- 3: backend, {2}, assembler
+- 4: assembler, {3}, object
|- 5: input, "1%dM", object
6: linker, {4, 5}, image
```

We can visualize the output as shown in fig. 2.10. As we can see in fig. 2.10, the driver receives an input file `hello.cpp` , which is a C++ file. The file is processed by the preprocessor, and we obtain the preprocessor output (marked as `c++-cpp-output`). The result is compiled into IR form by the compiler, and then the backend converts it into assembly form. This form is later transformed into an object file. The final object file is combined with another object (`libstdc++`) to produce the final binary (`image`).

2.2.3 Tools execution

The phases are combined into several tool executions. The Clang driver invokes different programs to produce the final executable. Specifically, for our example, it calls the `clang` compiler and the `ld` linker. Both programs require additional arguments that are set up by the driver.

For instance, our example program (`hello.cpp`) includes the following header:

```
#include <iostream>
...
```

We have not specified any additional arguments (such as search paths, for example `-I`) when we invoked the compilation. However, different architectures and operating systems might have different paths for locating headers.

On Fedora 37, the header is located in the `/usr/include/c++/12/iostream` folder. We can examine a detailed description of the process executed by the driver and the arguments used with the `-###` option:

```
$ clang hello.cpp -o /tmp/hello -lstdc++ -###
```

The output for this command is quite extensive, and certain parts have been omitted here. Please refer to fig. 2.11.

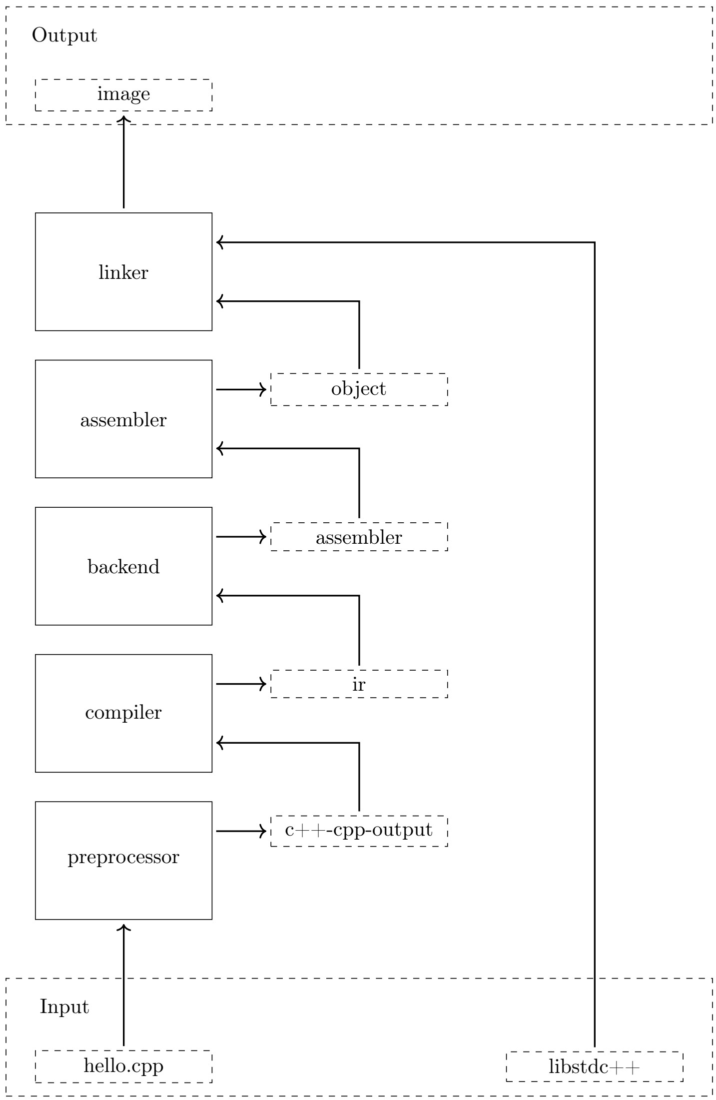


Figure 2.10: Clang driver phases

```
1 clang version 16.0.0 (https://github.com/llvm/llvm-project.git ...)
2  "<...>/llvm-project/build/bin/clang-16"
3     "-cc1" ... \
4     "-internal-isystem" \
5     "/usr/include/c++/12" ... \
6     "-internal-isystem" \
7     "/usr/include/c++/12/x86_64-redhat-linux" ... \
8     "-internal-isystem" ... \
9     "<...>/llvm-project/build/lib/clang/16/include" ... \
10    "-internal-externc-isystem" \
11    "/usr/include" ... \
12    "-o" "/tmp/hello-XXX.o" "-x" "c++" "hello.cpp"
13 ".../bin/ld" ... \
14    "-o" "/tmp/hello" ... \
15    "/tmp/hello-XXX.o" \
16    "-lstdc++" ...
```

Figure 2.11: Clang driver, tools execution, the host system is Fedora 37.

As we can see in fig. 2.11, the driver initiates two processes: `clang-16` with the `-cc1` flag (see lines 2-12) and the linker `ld` (see lines 13-16). The clang compiler implicitly receives several search paths, as seen in lines 5, 7, 9 and 11. These paths are necessary for the inclusion of the iostream header in the test program. The output of the first executable (`/tmp/hello-XXX.o`) serves as input for the second one (see lines 12 and 15). The arguments `-lstdc++` and `-o /tmp/hello` are set for the linker, while the first argument (hello.cpp) is provided for the compiler invocation (first executable).

The process can be visualized as shown in fig. 2.12, where we can see that two executables are executed as part of the compilation process. The first one is `bin/clang-16` with a special flag (`-cc1`). The second one is the linker: `bin/ld` .

2.2.4 Combine all together

We can summarize the knowledge we have acquired so far using fig. 2.13. The figure illustrates two different processes started by the clang driver. The first one is `clang -cc1` (compiler), and the second one is `ld` (linker). The compiler process is the same executable as the clang driver (`clang`), but it is run with a special argument: `-cc1` . The compiler produces an object file that is then processed by the linker (`ld`) to generate the final bi-

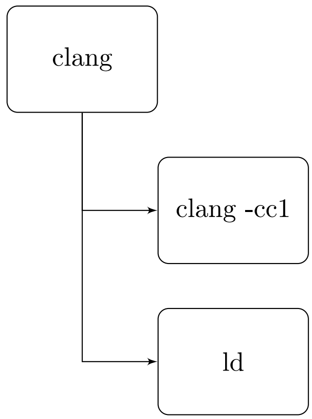


Figure 2.12: Clang driver, tools execution: clang driver runs 2 executables: The clang executable with -cc1 flag and linker - ld executable

nary.

In fig. 2.13, we can observe similar components of the compiler mentioned earlier (see Section 2.1, Compilers overview). However, the main difference is that the preprocessor (part of the lexer) is shown separately, while the frontend and middle-end are combined into the compiler. Additionally, the figure depicts an assembler that is executed by the driver to generate the object code. It is important to note that the assembler can be integrated, as shown in fig. 2.13, or it may require a separate process to be executed.

> Here is an example of specifying an external assembler using the `-c` (compile only) and `-o` (output file) options, along with the appropriate flags for your platform:
>
> ```
> $<...>/llvm-project/build/bin/clang -c hello.cpp -o /tmp/hello.o
> as -o /tmp/hello.o /tmp/hello.s
> ```

2.2.5 Debugging clang

We're going to step through a debugging session for our compilation process, illustrated in fig. 2.9. Our chosen point of interest, or breakpoint, is the `clang::ParseAST` function. In a typical debug session, which resembles the one outlined in fig. 1.4, you would feed command-line arguments following the `-` symbol. The command should look like this:

```
$lldb <...>/llvm-project/build/bin/clang -- hello.cpp -o /tmp/hello -lstdc++
```

In this case, `<...>` represents the directory path used to clone the LLVM project.

Unfortunately, this approach doesn't work with the Clang compiler:

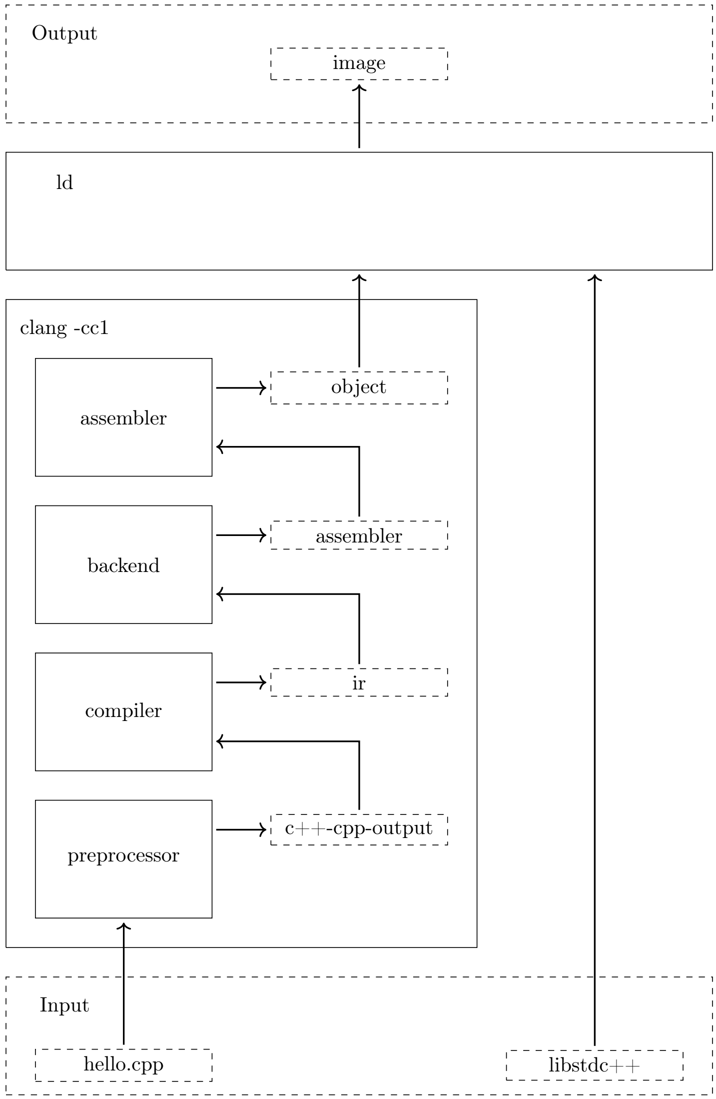


Figure 2.13: Clang driver: The driver got input file *`hello.cpp`* that is a C++ file. It starts 2 processes: *`clang`* and *`ld`*. The first one does real compilation and starts the integrated assembler. The last one is the linker (*`ld`*) that produces the final binary (*`image`*) from the result got from compiler and external library (*`libstdc++`*)

```
1 $ lldb <...>/llvm-project/build/bin/clang -- src/part1/ch2_arch/hello.cpp -o
  ↪ /tmp/hello.o -lstdc++
2 ...
3 (lldb) b clang::ParseAST
4 ...
5 (lldb) r
6 ...
7 2 locations added to breakpoint 1
8 ...
9 Process 247135 stopped and restarted: thread 1 received signal: SIGCHLD
10 Process 247135 stopped and restarted: thread 1 received signal: SIGCHLD
11 Process 247135 exited with status = 0 (0x00000000)
12 (lldb)
```

As we can see from line 7, the breakpoint was set but the process finished successfully (line 11) without any interruptions. In other words, our breakpoint didn't trigger in this instance.

Understanding the internals of clang-driver can help us identify the problem at hand. As mentioned earlier, the clang executable acts as a driver in this context, running two separate processes (refer to fig. 2.12). Therefore, if we wish to debug the compiler, we need to run it using the `-cc1` option.

> It's worth mentioning a certain optimization implemented in clang in 2019 [Ganea", "2019"]. When using the `-c` option, the clang driver doesn't spawn a new process for the compiler:
>
> ```
> $<...>/llvm-project/build/bin/clang -c hello.cpp -o /tmp/hello.o -###
> clang version 16.0.0 ...
> InstalledDir: <...>/llvm-project/build/bin
>  (in-process)
>  "<...>/llvm-project/build/bin/clang-16" "-cc1" ... "hello.cpp"
>  ...
> ```
>
> As shown above, the clang driver does not spawn a new process and instead calls the "cc1" tool within the same process. This feature not only improves the compiler's performance but can also be leveraged for clang debugging.

Upon using `-cc1` option and excluding the `-lstdc++` option (which is specific to the second process, the ld linker), the debugger will generate the following output:

```
1 $ lldb <...>/llvm-project-16/build/bin/clang -- -cc1 hello.cpp -o /tmp/hello.o
2 ...
3 (lldb) b clang::ParseAST
4 ...
5 (lldb) r
6 ...
7 2 locations added to breakpoint 1
8 Process 249890 stopped
9 * thread #1, name = 'clang', stop reason = breakpoint 1.1
10     frame #0: 0x00007fffe803eae0 ... at ParseAST.cpp:116:3
11    113
12    114  void clang::ParseAST(...) {
13    115    // Collect global stats on Decls/Stmts ...
14 -> 116    if (PrintStats) {
15    117      Decl::EnableStatistics();
16    118      Stmt::EnableStatistics();
17    119    }
18 (lldb) c
19 Process 249890 resuming
20 hello.cpp:1:10: fatal error: 'iostream' file not found
21 #include <iostream>
22          ^~~~~~~~~~
23 1 error generated.
24 Process 249890 exited with status = 1 (0x00000001)
25 (lldb)
```

Thus we can see that we were able to successfully set the breakpoint, but the process ended with an error (see lines 20-24). This error arose because we omitted certain search paths, which are typically appended implicitly by the clang driver, necessary to find all the includes required for successful compilation.

We can successfully execute the process if we explicitly include all necessary arguments in the compiler invocation. Here's how to do that:

```
$ lldb <...>/llvm-project/build/bin/clang -- -cc1 -internal-isystem /usr/include/c++/12 -internal-isystem /usr/include/c++/12/x86_64-redhat-linux -internal-isystem <...>/llvm-project/build/lib/clang/16/include -internal-externc-isystem /usr/include hello.cpp -o /tmp/hello.o
...
(lldb) b clang::ParseAST
...
(lldb) r
...
2 locations added to breakpoint 1
Process 251736 stopped
* thread #1, name = 'clang', stop reason = breakpoint 1.1
    frame #0: 0x00007fffe803eae0 ... at ParseAST.cpp:116:3
   113
   114  void clang::ParseAST(...) {
   115    // Collect global stats on Decls/Stmts ...
-> 116    if (PrintStats) {
   117      Decl::EnableStatistics();
   118      Stmt::EnableStatistics();
   119    }
(lldb) c
Process 251736 resuming
Process 251736 exited with status = 0 (0x00000000)
(lldb)
```

In conclusion, we have successfully demonstrated the debugging of a clang compiler invocation. The techniques presented can be effectively employed for exploring the internals of a compiler and addressing compiler-related bugs.

2.3 Clang frontend overview

It's evident that the clang compiler toolchain conforms to the pattern widely described in various compiler books [Cooper and Torczon, 2012]. However, the clang's frontend part diverges significantly from a typical compiler frontend. The primary reason for this distinction is the complexity of the C++ language. Some features, like macros, can modify the source code itself, while others, like typedef, can influence the kind of token. Clang can also generate output in a variety of formats. For instance, the following command generates an aesthetically pleasing HTML view of the program shown in fig. 2.5:

```
$ clang -cc1 -emit-html max.cpp
```

Take note that we pass the argument to emit the HTML form of the source program to the clang frontend, specified with the `-cc1` option. Alternatively, you can pass an option to the frontend via the `-Xclang` option, which requires an additional argument representing the option itself. For example:

```
$ clang -fsyntax-only -Xclang -emit-html max.cpp
```

You may notice that in the command above, we utilized the `-fsyntax-only` option, instructing Clang to only execute the preprocessor, parser, and semantic analysis stages.

Accordingly, we can instruct the Clang frontend to perform different actions and produce varying types of output based on the provided compilation options. The base class for these actions is termed `FrontendAction` .

2.3.1 Frontend action

The Clang frontend is capable of executing only one frontend action at a time. A frontend action is a specific task or process that the frontend performs based on the provided compiler option. Below is a list of some possible frontend actions (the table only includes a subset of the available frontend actions):

FrontendAction	Compiler option	Description
EmitObjAction	`-emit-obj` (default)	Compile to an object file
EmitBCAction	`-emit-llvm-bc`	Compile to LLVM bytecode
EmitLLVMAction	`-emit-llvm`	Compile to LLVM readable form
ASTPrintAction	`-ast-print`	Build ASTs and then pretty-print them.
HTMLPrintAction	`-emit-html`	Prints the program source in HTML form
DumpTokensAction	`-dump-tokens`	Prints preprocessor tokens

Table 2.1: Frontend actions

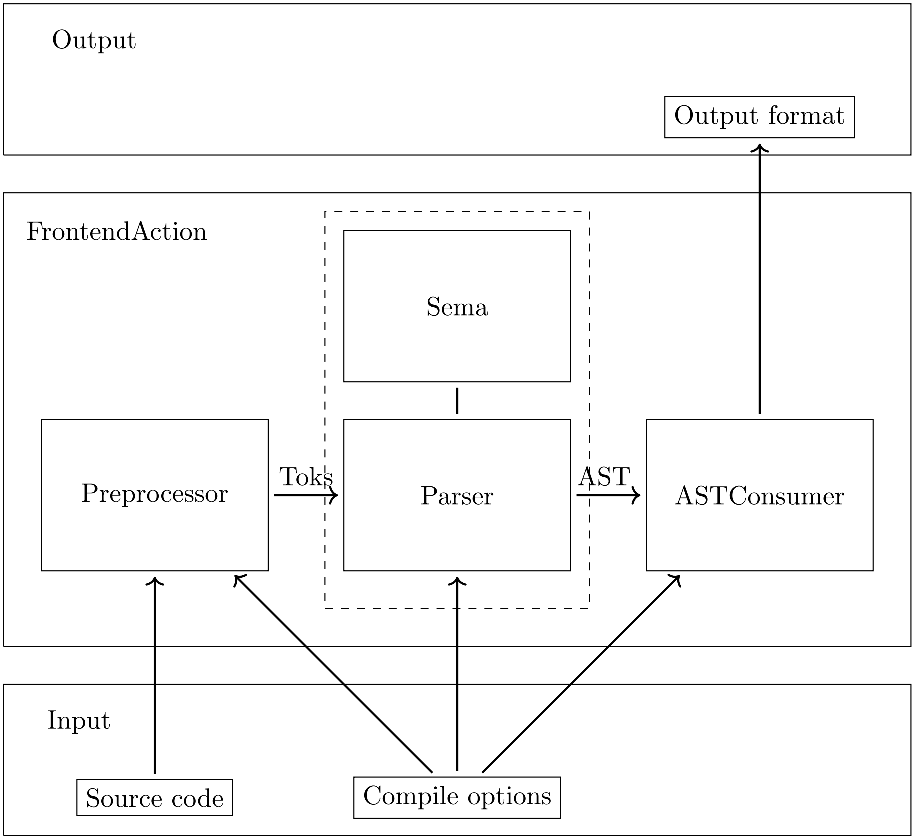


Figure 2.14: Clang frontend components

The diagram shown in fig. 2.14 illustrates the basic frontend architecture, which is similar to the architecture shown in fig. 2.4. However, there are notable differences specific to Clang.

One significant change is the naming of the lexer. In Clang, the lexer is referred to as the Preprocessor. This naming convention reflects the fact that the lexer implementation is encapsulated within the `Preprocessor` class. This alteration was inspired by the unique aspects of the C/C++ language, which includes special types of tokens (macros) that require specialized preprocessing.

Another noteworthy deviation is found in the parser component. While conventional compilers typically perform both syntax and semantic analysis within the parser, Clang distributes these tasks across different components. The `Parser` component focuses solely on syntax analysis, while the `Sema` component handles semantic analysis.

Furthermore, Clang offers the ability to produce output in different forms or formats. For example, the `CodeGenAction` class serves as the base class for various code generation actions, such as `EmitObjAction` or `EmitLLVMAction`.

We will use the code for `max` function from fig. 2.5 for our future exploration of the Clang frontend's internals:

```
int max(int a, int b) {
  if (a > b)
    return a;
  return b;
}
```

By utilizing the `-cc1` option, we can directly invoke the Clang frontend, bypassing the driver. This approach allows us to examine and analyze the inner workings of the Clang frontend in greater detail.

2.3.2 Preprocessor

The first part is the Lexer that is called as Preprocessor in Clang. Its primary goal is to convert the input program into a stream of tokens. You can print the token stream using the `-dump-tokens` options as follows

```
$ clang -cc1 -dump-tokens max.cpp
```

The output of the command is as shown below:

```
int 'int'        [StartOfLine]  Loc=<max.cpp:1:1>
identifier 'max'         [LeadingSpace] Loc=<max.cpp:1:5>
l_paren '('             Loc=<max.cpp:1:8>
int 'int'               Loc=<max.cpp:1:9>
identifier 'a'   [LeadingSpace] Loc=<max.cpp:1:13>
comma ','               Loc=<max.cpp:1:14>
int 'int'        [LeadingSpace] Loc=<max.cpp:1:16>
identifier 'b'   [LeadingSpace] Loc=<max.cpp:1:20>
r_paren ')'             Loc=<max.cpp:1:21>
l_brace '{'      [LeadingSpace] Loc=<max.cpp:1:23>
if 'if'  [StartOfLine] [LeadingSpace]   Loc=<max.cpp:2:3>
l_paren '('      [LeadingSpace] Loc=<max.cpp:2:6>
identifier 'a'          Loc=<max.cpp:2:7>
greater '>'      [LeadingSpace] Loc=<max.cpp:2:9>
identifier 'b'   [LeadingSpace] Loc=<max.cpp:2:11>
```

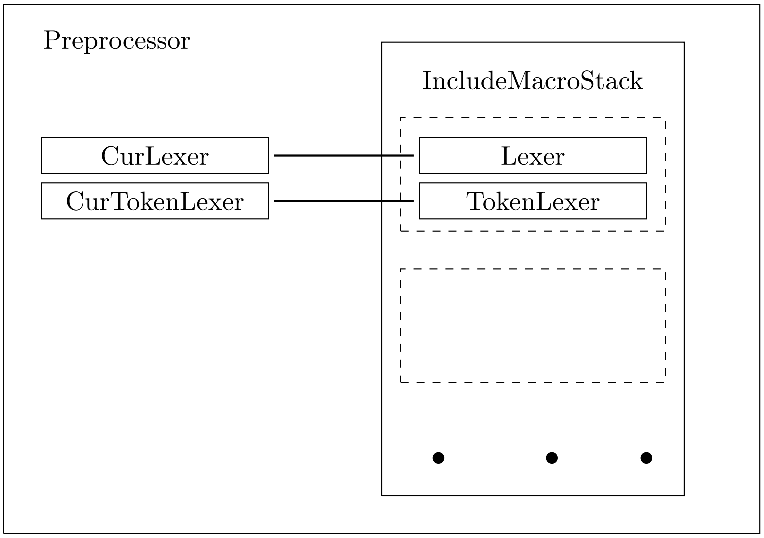


Figure 2.15: Preprocessor (clang lexer) class internals

```
r_paren ')'             Loc=<max.cpp:2:12>
return 'return'  [StartOfLine] [LeadingSpace]   Loc=<max.cpp:3:5>
identifier 'a'   [LeadingSpace] Loc=<max.cpp:3:12>
semi ';'                Loc=<max.cpp:3:13>
return 'return'  [StartOfLine] [LeadingSpace]   Loc=<max.cpp:4:3>
identifier 'b'   [LeadingSpace] Loc=<max.cpp:4:10>
semi ';'                Loc=<max.cpp:4:11>
r_brace '}'      [StartOfLine]  Loc=<max.cpp:5:1>
eof ''          Loc=<max.cpp:5:2>
```

As we can see, there are different types of tokens, such as language keywords (e.g., `int`, `return`), identifiers (e.g., `max`, `a`, `b`, etc.), and special symbols (e.g., semicolon, comma, etc.). The tokens for our small program are called normal tokens, which are returned by the lexer.

In addition to normal tokens, Clang has an additional type of token called annotation tokens. The primary difference is that these tokens also store additional semantic information. For instance, a sequence of normal tokens can be replaced by the parser with a single annotation token that contains information about the type or C++ scope. The primary reason for using such tokens is performance, as it allows for the prevention of reparsing when the parser needs to backtrack.

C/C++ language has some specifics that influence the internal implementation of the `Preprocessor` class. The first one is about macros. The `Preprocessor` class has two different helper classes to retrieve tokens:

- The `Lexer` class is used to convert a text buffer into a stream of tokens.
- The `TokenLexer` class is used to retrieve tokens from macro expansions.

It should be noted that only one of these helpers can be active at a time.

Another specific aspect of C/C++ is the *#include* directive[3]. In this case, we need to maintain a stack of includes, where each include can have its own `TokenLexer` or `Lexer`, depending on whether there is a macro expansion within it. As a result, the `Preprocessor` class keeps a stack of lexers (`IncludeMacroStack` class) for each *#include* directive, as shown in fig. 2.15.

2.3.3 Parser and Sema

The Parser and Sema are crucial components of the Clang compiler frontend. They handle the syntax and semantic analysis of the source code, producing an AST as output. This tree can be visualized for our test program using the command:

```
$ clang -cc1 -ast-dump max.cpp
```

The output of this command is shown below

```
TranslationUnitDecl 0xc4b578 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0xc4bde0 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| `-BuiltinType 0xc4bb40 '__int128'
...
`-FunctionDecl 0xc91580 <max.cpp:1:1, line:5:1> line:1:5 max 'int (int, int)'
  |-ParmVarDecl 0xc91428 <col:9, col:13> col:13 used a 'int'
  |-ParmVarDecl 0xc914a8 <col:16, col:20> col:20 used b 'int'
  `-CompoundStmt 0xc917b8 <col:23, line:5:1>
    |-IfStmt 0xc91750 <line:2:3, line:3:12>
    | |-BinaryOperator 0xc916e8 <line:2:7, col:11> 'bool' '>'
    | | |-ImplicitCastExpr 0xc916b8 <col:7> 'int' <LValueToRValue>
    | | | `-DeclRefExpr 0xc91678 <col:7> 'int' lvalue ParmVar 0xc91428 'a' 'int'
    | | `-ImplicitCastExpr 0xc916d0 <col:11> 'int' <LValueToRValue>
    | |   `-DeclRefExpr 0xc91698 <col:11> 'int' lvalue ParmVar 0xc914a8 'b' 'int'
    | `-ReturnStmt 0xc91740 <line:3:5, col:12>
    |   `-ImplicitCastExpr 0xc91728 <col:12> 'int' <LValueToRValue>
    |     `-DeclRefExpr 0xc91708 <col:12> 'int' lvalue ParmVar 0xc91428 'a' 'int'
    `-ReturnStmt 0xc917a8 <line:4:3, col:10>
      `-ImplicitCastExpr 0xc91790 <col:10> 'int' <LValueToRValue>
        `-DeclRefExpr 0xc91770 <col:10> 'int' lvalue ParmVar 0xc914a8 'b' 'int'
```

Clang utilizes a hand-written recursive-descent parser [Community, 2023a]. This parser can be considered simple, and this simplicity was one key reason for its selection. Additionally,

[3] which is also applicable to the import directive

the complex rules specified for the C/C++ languages necessitated an ad-hoc parser with easily adaptable rules.

Let's explore how this works with our example. Parsing begins with a top-level declaration known as a `TranslationUnitDecl`, representing a single translation unit. The C++ standard defines a translation unit as follows [for Standardization, 2020, lex.separate]:

> A source file together with all the headers (16.5.1.2) and source files included (15.3) via the preprocessing directive #include, less any source lines skipped by any of the conditional inclusion (15.2) preprocessing directives, is called a translation unit.

The parser first recognizes that the initial tokens from the source code correspond to a function definition, as defined in the C++ standard [for Standardization, 2020, dcl.fct.def.general]:

```
function-definition :
        ... declarator ... function-body
        ...
```

The corresponding code is below

```
int max(...) {
  ...
}
```

The function definition necessitates a declarator and function body. We'll start with the declarator, defined in the C++ standard as [for Standardization, 2020, dcl.decl.general]:

```
declarator:
        ...
        ... parameters-and-qualifiers ...
...
parameters-and-qualifiers:
        ( parameter-declaration-clause ) ...
...
parameter-declaration-clause:
        parameter-declaration-list ...
parameter-declaration-list:
        parameter-declaration
        parameter-declaration-list , parameter-declaration
```

In other words, the declarator specifies a list of parameter declarations within brackets. The corresponding piece of code from the source is as follows:

```cpp
... (int a, int b)
  ...
```

The function definition, as stated above, also requires a function body. The C++ standard specifies the function body as follows: [for Standardization, 2020, dcl.fct.def.general]

```
function-body:
        ... compound-statement
        ...
```

Thus the function body consists of a compound statement, which is defined as follows in the C++ standard [for Standardization, 2020, stmt.block]

```
compound-statement:
       { statement-seq ... }
statement-seq:
       statement
       statement-seq statement
```

Therefore, it describes a sequence of statements enclosed within `{...}` brackets.

Our program has two types of statements: the conditional (if) statement and the return statement. These are represented in the C++ grammar definition as follows [for Standardization, 2020, stmt.pre]:

```
statement:
        ...
        selection-statement
        ...
        jump-statement
        ...
```

In this context, the selection statement corresponds to the `if` condition in our program, while the jump statement corresponds to the `return` operator.

Let's examine the jump statement in more detail [for Standardization, 2020, stmt.jump.general]:

```
jump-statement:
        ...
        return expr-or-braced-init-list;
        ...
```

where expr-or-braced-init-list is defined as [for Standardization, 2020, dcl.init.general]:

```
expr-or-braced-init-list:
        expression
        ...
```

In this context, the `return` keyword is followed by an expression and a semicolon. In our case, there's an implicit cast expression that automatically converts the variable into the required type (int).

It can be enlightening to examine the parser's operation through the LLDB debugger:

```
1 $ lldb <...>/llvm-project/build/bin/clang -- -cc1 max.cpp
2 ...
3 (lldb) b clang::Parser::ParseReturnStatement
4 (lldb) r
5 ...
6 (lldb) c
7 ...
8 * thread #1, name = 'clang', stop reason = breakpoint 1.1
9     frame #0: ... clang::Parser::ParseReturnStatement(...) at
   ↪  ParseStmt.cpp:2358:3
10    2355 ///         'co_return' expression[opt] ';'
11    2356 ///         'co_return' braced-init-list ';'
12    2357 StmtResult Parser::ParseReturnStatement() {
13 -> 2358   assert((Tok.is(tok::kw_return) || Tok.is(tok::kw_co_return)) &&
14    2359          "Not a return stmt!");
15    2360   bool IsCoreturn = Tok.is(tok::kw_co_return);
16    2361   SourceLocation ReturnLoc = ConsumeToken();  // eat the 'return'.
17 (lldb) bt
18   * frame #0: ... clang::Parser::ParseReturnStatement( ...
19     ...
20     frame #2: ... clang::Parser::ParseStatementOrDeclaration( ...
21     frame #3: ... clang::Parser::ParseCompoundStatementBody( ...
22     frame #4: ... clang::Parser::ParseFunctionStatementBody( ...
23     frame #5: ... clang::Parser::ParseFunctionDefinition( ...
24 ...
```

Figure 2.16: Second return statement parsing at max.cpp example program

As you can see in fig. 2.16, line 3, we've set a breakpoint for the parsing of return statements[4]. Our program has two return statements. We bypass the first call (line 6) and halt at the second method invocation (line 13). The backtrace (from the 'bt' command at line

[4] Specifically, at the `clang::Parser::ParseReturnStatement` method

17) displays the call stack for the parsing process. This stack mirrors the parsing blocks we described earlier, adhering to the C++ grammar detailed in [for Standardization, 2020, lex.separate].

The parsing results in the generation of AST. We can also inspect the process of AST creation using the debugger. To do this, we need to set a corresponding breakpoint at the `clang::ReturnStmt::Create` method:

```
$ lldb <...>/llvm-project/build/bin/clang -- -cc1 max.cpp
...
(lldb) b clang::ReturnStmt::Create
(lldb) r
...
(lldb) c
...
* thread #1, name = 'clang', stop reason = breakpoint 1.1
    frame #0: ... clang::ReturnStmt::Create(...) at Stmt.cpp:1204:8
   1201
   1202 ReturnStmt *ReturnStmt::Create( ... ) {
-> 1204   bool HasNRVOCandidate = NRVOCandidate != nullptr;
   1205   ...
   1206   ...
   1207   return new (Mem) ReturnStmt(RL, E, NRVOCandidate);
(lldb) bt
* thread #1, name = 'clang', stop reason = breakpoint 1.1
  * frame #0: ... clang::ReturnStmt::Create( ...
    frame #1: ... clang::Sema::BuildReturnStmt( ...
    frame #2: ... clang::Sema::ActOnReturnStmt( ...
    frame #3: ... clang::Parser::ParseReturnStatement( ...
    frame #4: ... clang::Parser::ParseStatementOrDeclarationAfterAttributes( ...
    ...
```

As can be seen, the AST node for the return statement is created by the Sema component.

The beginning of the return statement parser can be located in frame 4:

```
(lldb) f 4
```

```
frame #4: ... clang::Parser::ParseStatementOrDeclarationAfterAttributes( ...
   325      break;
   326    case tok::kw_return:              // C99 6.8.6.4: return-statement
-> 327      Res = ParseReturnStatement();
   328      SemiError = "return";
   329      break;
   330    case tok::kw_co_return:            // C++ Coroutines: ...
(lldb)
```

As we can observe, there is a reference to the C99 standard [International Organization for Standardization (ISO), 1999] for the corresponding statement. The standard [International Organization for Standardization (ISO), 1999] provides a detailed description of the statement and the process for handling it.

The code assumes that the current token is of type `tok::kw_return`, and in this case, the parser invokes the relevant `clang::Parser::ParseReturnStatement` method.

While the process of AST node creation can vary across different C++ constructs, it generally follows the pattern displayed in fig. 2.17.

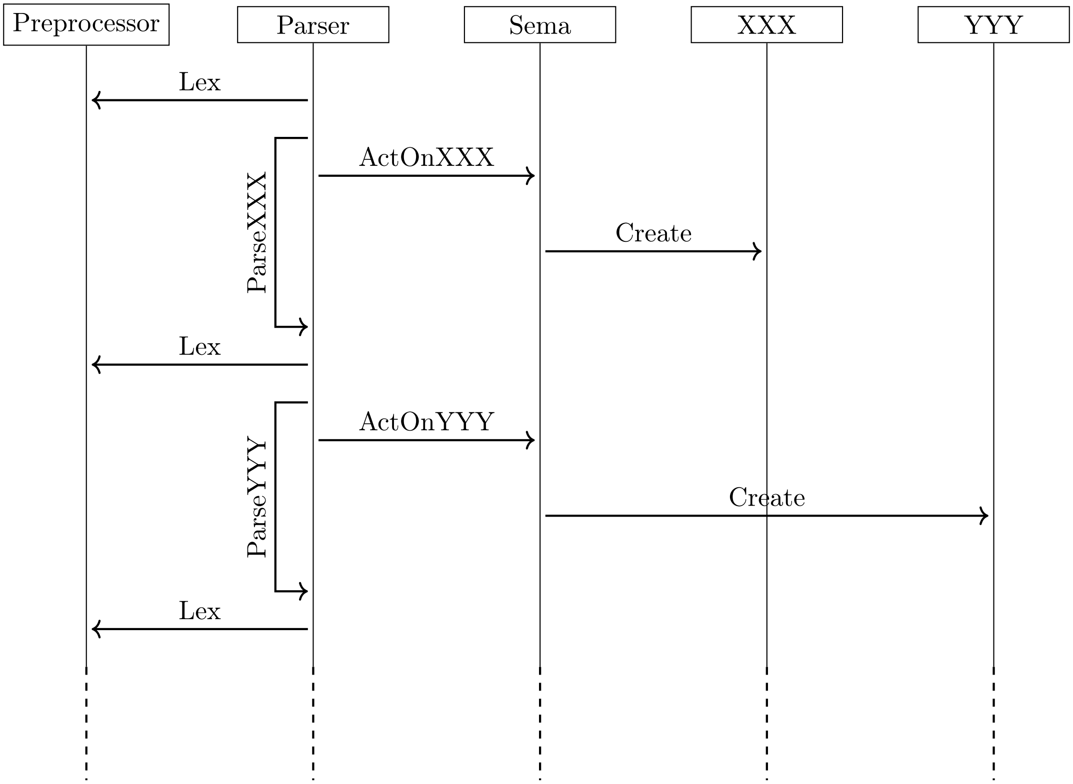


Figure 2.17: C++ parsing in Clang frontend

As can be seen, the `Parser` invokes the `Preprocessor::Lex` method to retrieve a token from the lexer. It then calls a method corresponding to the token, for example, `Parser:ParseXXX` for the token `XXX`. This method then calls `Sema::ActOnXXX`, which creates the corresponding object using `XXX::Create`. The process is then repeated with a new token.

With this, we have now fully explored how the typical compiler frontend flow is implemented in Clang. We can see how the lexer component (the preprocessor) works in tandem with the parser (which comprises the parser and sema components) to produce the primary data structure for future code generation: the Abstract Syntax Tree (AST). The AST is not only essential for code generation but also for code analysis and modification. Clang provides easy access to the AST, thereby enabling the development of a diverse range of compiler tools.

2.4 Summary

In this chapter, we have acquired a basic understanding of compiler architecture and delved into the various stages of the compilation process, with a focus on the Clang driver. We have explored the internals of the Clang frontend, studying the preprocessor that transforms a program into a set of tokens, and the parser, which interacts with a component called 'Sema'. Together, these elements perform syntax and semantic analysis.

The upcoming chapter will center on the Clang Abstract Syntax Tree (AST)—the primary data structure employed in various Clang tools. We will discuss its construction and the methods for traversing it.

2.5 Further reading

- Working Draft, Standard for Programming Language C++: https://eel.is/c++draft/
- "Clang" CFE Internals Manual: https://clang.llvm.org/docs/InternalsManual.html
- Keith Cooper and Linda Torczon: Engineering A Compiler, 2012 [Cooper and Torczon, 2012]

3

Clang AST

The AST is the skeleton for the Clang frontend. It is also the primary instrument for linters and other Clang tools. The AST keeps the results of syntax and semantic analysis and represents a tree [1] with leaf nodes for different objects, such as function declarations and loop bodies. Clang provides advanced tools for searching (matching) different nodes, which are implemented in the form of a DSL (domain-specific language). It is important to understand how it is implemented to be able to use it.

We will start with the basic data structures and class hierarchy that Clang uses to create the AST. We will discuss the different methods used for AST traversal and some helper classes for node matching during the traversal.

3.1 AST

Typical tree structure implemented in C++ has all nodes derived from a base class. Clang uses a different approach. It splits different C++ constructions into separate groups with basic classes for each of them.

- Statements. `clang::Stmt` is the basic class for all statements. That includes ordinary statements such as `clang::IfStmt` as well as expressions and other C++ constructions

[1] The Clang AST is not a real tree since backward edges are possible, making the graph a more suitable term to describe Clang's AST. We will cover some specific cases later (TBD link).

- Declarations. `clang::Decl` is the basic class for declarations. This includes a variable, typedef, function, struct, etc. There is also a separate basic class for declarations with context i.e. declarations that might contain another declarations: `clang::DeclContext`. Translation units (`clang::TranslationUnitDecl` class) and namespaces (`clang::NamespaceDecl` class) are typical examples for declarations with context.

- Types. `clang::Type` is basic class for types representations.

Lets look at all the groups in details.

3.1.1 Statements

`Stmt` is the basic class for all statements. The statements can be combined into 2 sets (see fig. 3.1). The first one contains statements with values and an opposite group is for statements without values.

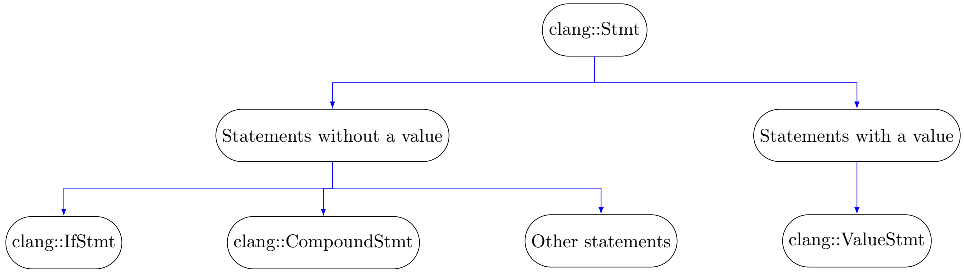


Figure 3.1: Clang AST: Statements

The group of statements without a value consist of different C++ constructions such as if-statement (`clang::IfStmt` class) or compound statement (`CompoundStmt` class). The majority of all statements falls into the group.

The group of statement with a value consists of one base class `clang::ValueStmt` that has several children such as `clang::LabelStmt` (for labels representation) or `clang::ExprStmt` (for expressions representation), see fig. 3.2.

3.1.2 Declarations

Declarations can also be combined into 2 primary groups: Declarations with context and without. Declarations with context can be considered as placeholders for other declarations. For example C++ namespace as well as translation unit or function declaration might contain other declarations. Declaration of a friend entity (`clang::DeclFriend`) can be considered as an example of a declaration without context.

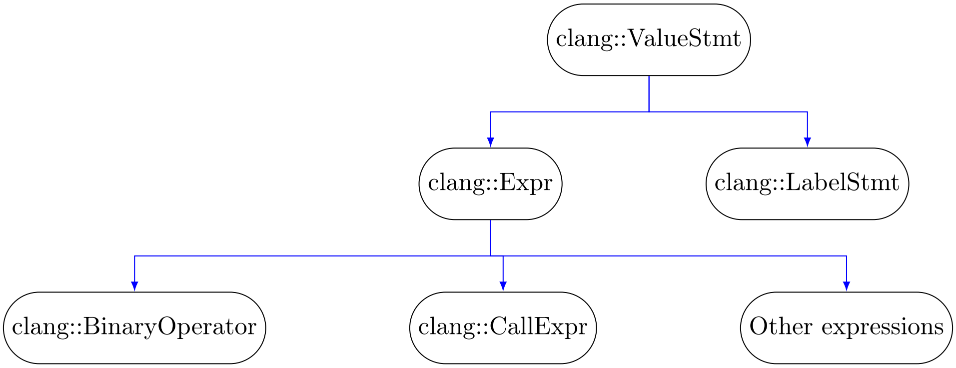


Figure 3.2: Clang AST: Statements with a value

It has to be noted that classes that are inherited from `DeclContext` have also `clang::Decl` as their top parent.

Some declarations can be redeclared as in the following example

```
extern int a;
int a = 1;
```

Such declarations have an additional parent that is implemented via `clang::Redeclarable<...>` template.

3.1.3 Types

C++ is a statically typed language, which means that the types of variables must be declared at compile-time. The types allow compiler to make a reasonable conclusion about the program meaning i.e. types are important part of semantic analysis. `clang::Type` is the basic class for types in Clang.

Types in C/C++ might have qualifiers that are called as CV-qualifiers at standard [for Standardization, 2020, basic.type.qualifier]. CV here is for 2 keywords `const` and `volatile` that can be used as the qualifier for a type. [2] Clang has a special class to support a type with qualifier: `clang::QualType` that is a pair of a pointer to `clang::Type` and a bit mask with information about the type qualifier. The class has a method to retrieve a pointer to the `clang::Type` and check different qualifiers. The code below shows how we can check a

[2] C99 standard has an additional type qualifier `restrict` that is also supported by clang [International Organization for Standardization (ISO), 1999, 6.7.3]

type for a const qualifier [3]

```
bool checkConst(QualType QT) {
  // Assigning to a const object has undefined behavior.
  if (QT.isConstQualified()) {
    Info.FFDiag(E, diag::note_constexpr_modify_const_type) << QT;
    return false;
  }
  return true;
}
```

It's worth mentioning that `clang::QualType` has `operator->()` and `operator*()` implemented, i.e. it can be considered as a smart pointer for underlying `clang::Type` class.

In addition to the qualifiers type can have additional information that represents different memory address models, for instance there can be a pointer to an object or reference. `clang::Type` has the following helper methods to check different address models:

- `clang::Type::isPointerType()` for pointer type check
- `clang::Type::isReferenceType()` for reference type check

Types in C/C++ can also use aliases that are introduced by using `typedef` or `using` keywords. The following code defines `foo` and `bar` as aliases for `int` type

```
using foo = int;
typedef int bar;
```

Original types, `int` at our case, are called as canonical. You can test if the type is canonical or not using `clang::QualType::isCanonical()` method. `clang::QualType` also provides a method to retrieve the canonical type from an alias: `clang::QualType::getCanonicalType()`.

3.2 AST traversal. Test tool

Compiler need to traverse AST as soon as its generated to produce IR code. Thus having a suitable data structure for tree traversal is essential for AST design, i.e. AST has to be designed to provide an easiest way for the tree traversal. Typical way is to have common base

[3] The code is from LLVM release 16.0, `clang/lib/AST/ExprConstant.cpp`, line 3855

class for AST nodes and the class has to provide a method to retrieve the node's children. As we know Clang's AST nodes do not have a common ancestor. How the tree traversal can be organized at the situation?

To answer on the question we will create a simple clang tool that uses recursive AST visitor for traversal. The tool will emit a message when it finds a reference to a declaration (i.e. variable usage) in the analysed source code. We will use the same CMake file as it was created for our first clang tool (see section 1.4), the only one additional library is added for the new tool: `clangAST`. The result `CMakeLists.txt` is the following:

```
project("astdumper")

if ( NOT DEFINED ENV{LLVM_HOME})
  message(FATAL_ERROR "$LLVM_HOME is not defined")
else()
  message(STATUS "$LLVM_HOME found: $ENV{LLVM_HOME}")
  set(LLVM_HOME $ENV{LLVM_HOME} CACHE PATH "Root of LLVM installation")
  set(LLVM_LIB ${LLVM_HOME}/lib)
  set(LLVM_DIR ${LLVM_LIB}/cmake/llvm)
  find_package(LLVM REQUIRED CONFIG)
  include_directories(${LLVM_INCLUDE_DIRS})
  link_directories(${LLVM_LIBRARY_DIRS})
  set(SOURCE_FILE AstDumper.cpp)
  add_executable(astdumper ${SOURCE_FILE})
  set_target_properties(astdumper PROPERTIES COMPILE_FLAGS "-fno-rtti")
  target_link_libraries(astdumper
    LLVMSupport
    clangAST
    clangBasic
    clangFrontend
    clangSerialization
    clangTooling
  )
```

The `main` function for our tool looks like

```
#include "clang/Tooling/CommonOptionsParser.h"
```

```
2 #include "clang/Tooling/Tooling.h"
3 #include "llvm/Support/CommandLine.h" // llvm::cl::extrahelp
4
5 #include "FrontendAction.hpp"
6
7 namespace {
8 llvm::cl::OptionCategory TestCategory("Test project");
9 llvm::cl::extrahelp
10     CommonHelp(clang::tooling::CommonOptionsParser::HelpMessage);
11 } // namespace
12
13 int main(int argc, const char **argv) {
14   llvm::Expected<clang::tooling::CommonOptionsParser> OptionsParser =
15       clang::tooling::CommonOptionsParser::create(argc, argv, TestCategory);
16   if (!OptionsParser) {
17     llvm::errs() << OptionsParser.takeError();
18     return 1;
19   }
20   clang::tooling::ClangTool Tool(OptionsParser->getCompilations(),
21                                  OptionsParser->getSourcePathList());
22   return Tool.run(clang::tooling::newFrontendActionFactory<
23                       clangbook::astdumper::FrontendAction>()
24                       .get());
25 }
```

As you may see (line 5,23), we use a custom frontend action for our project: `clangbook::astdumper::Front`
The code for the class is

```
#include "Consumer.hpp"
#include "clang/Frontend/FrontendActions.h"

namespace clangbook {
namespace astdumper {
class FrontendAction : public clang::ASTFrontendAction {
public:
  virtual std::unique_ptr<clang::ASTConsumer>
```

```
9   CreateASTConsumer(clang::CompilerInstance &CI,
10                    clang::StringRef File) override {
11    return std::make_unique<Consumer>();
12  }
13 };
14 } // namespace astdumper
15 } // namespace clangbook
```

As you may see we redefined `clang::ASTFrontendAction::CreateASTConsumer` where we create our custom `clangbook::astdumper::Consumer`(line 9).

The code for our custom AST consumer is the following:

```
#include "Visitor.hpp"
#include "clang/Frontend/ASTConsumers.h"

namespace clangbook {
namespace astdumper {
class Consumer : public clang::ASTConsumer {
public:
  Consumer() : V(std::make_unique<Visitor>()) {}

  virtual void HandleTranslationUnit(clang::ASTContext &Context) override {
    V->TraverseDecl(Context.getTranslationUnitDecl());
  }

private:
  std::unique_ptr<Visitor> V;
};
} // namespace astdumper
} // namespace clangbook
```

As you may see we create an example visitor and invoke it at `clang::ASTConsumer::HandleTranslationUn`
The code for the visior is the most interesting for us:

```
#include "clang/AST/RecursiveASTVisitor.h"
```

```
2
3 namespace clangbook {
4 namespace astdumper {
5 class Visitor : public clang::RecursiveASTVisitor<Visitor> {
6 public:
7   bool VisitDeclRefExpr(const clang::DeclRefExpr *DRE) {
8     llvm::errs() << "Found reference to a declaration: '"
9                  << DRE->getFoundDecl()->getName() << "'\n";
10    return true;
11  }
12 };
13 } // namespace astdumper
14 } // namespace clangbook
```

The code will print a message with the found object name when it finds a reference to a declaration, see lines 8-9.

We can build our program to use the same sequence of commands as for our test project, see Section 1.4, Test project: syntax check with clang tool

```
export LLVM_HOME=<...>/llvm-project/install
mkdir build
cd build
cmake -G Ninja -DCMAKE_BUILD_TYPE=Debug ..
ninja
```

As you may notice we used `-DCMAKE_BUILD_TYPE=Debug` option for CMake. That is because we want to investigate the result program under debugger.

The program we used for investigations previously (see fig. 2.5) will be used for AST traversal investigations as well:

```
1 int max(int a, int b) {
2   if (a > b)
3     return a;
4   return b;
5 }
```

The program has 4 references to declarations: variables 'a' and 'b' at line 2, variable 'a' at line 3 and variable 'b' at line 4.

We can run our program as follows:

```
$ ./astdumper max.cpp
...
Found reference to a declaration: 'a'
Found reference to a declaration: 'b'
Found reference to a declaration: 'a'
Found reference to a declaration: 'b'
```

Our test program uses `clang::RecursiveASTVisitor` and our next task will be to get a better understanding how it works. We will use LLDB debugger as our primary tool for the investigations.

3.3 Recursive AST Visitor

As it was mentioned earlier we are going to use debugger for the recursive visitor investigations. The debugger session can be run as follows

```
$ lldb ./astdumper -- max.cpp
```

We will look into the traversal that finds a function declaration. We can setup breakpoint into `clangbook::astdumper::Visitor::VisitDeclRefExpr` and run the debugging session:

```
(lldb) b clangbook::astdumper::Visitor::VisitDeclRefExpr
(lldb) r
...
* thread #1, name = 'astdumper', stop reason = breakpoint 1.1
    frame #0: ... ::Visitor::VisitDeclRefExpr(...) at Visitor.hpp:8:15
   5    class Visitor : public clang::RecursiveASTVisitor<Visitor> {
   6    public:
   7      bool VisitDeclRefExpr(const clang::DeclRefExpr *DRE) {
-> 8        llvm::errs() << "Found reference to a declaration: '"
   9                     << DRE->getFoundDecl()->getName() << "'\n";
   10       return true;
   11     }
(lldb)
```

We will start our investigation with 11th frame:

```
(lldb) f 11
frame #11: ... ::HandleTranslationUnit(...) at Consumer.hpp:11:20
   8      Consumer() : V(std::make_unique<Visitor>()) {}
   9
   10     virtual void HandleTranslationUnit(clang::ASTContext &Context) override {
-> 11       V->TraverseDecl(Context.getTranslationUnitDecl());
```

```
   12     }
   13
   14   private:
```

This is our code that starts the traversal procedure. We can use "up" and "down" command to navigate the stack. The next frame will be frame #10: be

```
(lldb) down
frame #10: ...::TraverseDecl(...) at DeclNodes.inc:645:1
   642  #ifndef TRANSLATIONUNIT
   643  #  define TRANSLATIONUNIT(Type, Base) DECL(Type, Base)
   644  #endif
-> 645  TRANSLATIONUNIT(TranslationUnit, Decl)
   646  #undef TRANSLATIONUNIT
   647
   648  LAST_DECL_RANGE(Decl, AccessSpec, TranslationUnit)
```

It reflects the fact that we start with translation unit traversal. Frame #8 gives us the code where we do the real traversal

```
(lldb) f 8
frame #8: ...::TraverseDeclContextHelper(...) at RecursiveASTVisitor.h:1489:7
   1486
   1487   for (auto *Child : DC->decls()) {
   1488     if (!canIgnoreChildDeclWhileTraversingDeclContext(Child))
-> 1489       TRY_TO(TraverseDecl(Child));
   1490   }
   1491
   1492   return true;
```

The `clang::TranslationUnitDecl` is inherited from `clang::DeclContext` that is a holder for another declarations. The class provides `clang::DeclContext::decls` method that can be used for a loop over all declarations stored at the specific `clang::DeclContext` instance. Our translation unit keeps several declarations and one of them is our function. We can see it if we print an AST node for the `Child` variable that was selected for execution at our breakpoint [4]

```
(lldb) p Child->dump()
FunctionDecl ... max.cpp:1:1, line:5:1> line:1:5 max 'int (int, int)'
|-ParmVarDecl ... used a 'int'
|-ParmVarDecl ... used b 'int'
`-CompoundStmt ... <col:23, line:5:1>
...
```

It's worth mentioning that there is a common pattern for AST recursive visitor. It uses ad-hoc methods of AST nodes for the tree traversal. For instance it uses loop over declarations

4 It should be an AST node that corresponds to a declaration reference because we stopped at the top frame on the method that processes function declarations

stored at translation unit for finding a declaration reference.

3.4 AST matchers

TBD

3.5 Explore clang AST with clang-query

TBD

3.6 Errors processing

Clang tries to collect as many errors as possible while compiling a program. TBD

3.7 Summary

TBD

3.8 Further reading

- How to write RecursiveASTVisitor: https://clang.llvm.org/docs/RAVFrontendAction.html

4

Basic libraries and tools

LLVM has been written in C++ language and currently (since July 2022) uses the c++17 version of the C++ standard [Community, 2022b]. On the other side, it has a lot of internal implementations for fundamental containers with the primary goal of performance. Therefore, being familiar with the extensions is crucial if you want to work with LLVM and clang. In addition, LLVM also introduced additional development tools such as TableGen - DSL (domain-specific language) for structural data processing and LIT (LLVM test framework). You can find some info about the tools here.

We are planning to use a simple example project to demonstrate the tools. The example project will be Clang plugin that can estimate complexity of the source code being compiled and print a warning if the number of functions/methods exceeds the limit specified as a parameter.

4.1 Basic libs

TBD

4.2 TableGen

TBD

4.3 LLVM test framework

TBD

4.4 Summary

TBD

4.5 Further reading

- LLVM Coding Standards: https://llvm.org/docs/CodingStandards.html [Community, 2023b]

Part 2: Clang tools

You can find some info about different clang tools here. We will start with linters that are based on clang-tidy, continue with some advanced code analysis techniques (CFG and live time analysis). The next chapter will be about different refactoring tools such as clang-format. The last chapter will be about IDE support. We are going to investigate how VSCode can be extended with language server provided by LLVM (clangd).

5

clang-tidy linter framework

There is an introduction to clang-tidy - the clang-based linter framework that uses AST to find anti-patterns in the C/C++/Objective-C code. First, we will start with a clang-tidy description, what kind of checks it has and how we can use them. Later we will investigate the clang-tidy architecture and how we can create our custom lint check.

5.1 clang-tidy overview and usage examples

TBD

5.2 clang-tidy internal design

TBD

5.3 Custom clang-tidy check

TBD

6

Advanced code analysis

he clang-tidy checks from the previous chapter can be considered based on an advanced matching provided by AST. However, it might not be enough when you want to detect some complex problems, such as lifetime issues. We will introduce advanced code analysis tools based on CFG (control flow graph). The clang static analyser is an excellent example of such devices, but we also have some CFG integration into clang-tidy. The chapter will start with typical usage examples and continue with implementation details. We will finish with our custom check that uses advanced techniques.

6.1 Usage cases

TBD

6.2 CFG and life time analysis

TBD

6.3 Custom CFG check

TBD

7

Refactoring tools

The chapter is about code refactoring tools. For example, suppose you want to modify a set of files according to a new code style or want to detect and fix a specific problem in your project. The code modification tools can help here. We will start with a typical usage scenario and finish with our custom code modification tool.

7.1 Code modification and clang-format

TBD

7.2 Custom code modification tool

Idea: we are going to replace public vars in a class with custom get/set methods.

TBD

8

IDE support and code navigation

The chapter is about Language Server Protocol (LSP) and how you can use it to extend your IDE. As the primary IDE, we will use VSCode. LLVM has its implementation of LSP as clangd. We will start with a typical usage scenario, continue with implementation details and finish with our custom extension for LSP that we must implement on both client (VSCode extension) and server (clangd) sides.

8.1 VSCode and LSP

TBD

8.2 clangd internals

TBD

8.3 Custom extension for LSP

TBD

Part 3:
Clang frontend features

You can find some info about some clang features here. The features have the primary goal the compiler performance. Some of them (clang modules) are part of C++ standard but some (HeaderMaps) are clang specific feature.

9

Features

You can find some info about different clang features in the chapter. Some of them are used with clang tools such as precompiled headers that allow fast editing at different IDEs. Others are specific clang features that are not well known but can provide some benefits, for instance, improving overall compilation performance. We will start with a typical usage scenario for each component and finish with its implementation details.

9.1 Precompiled headers

Precompiled headers or pch is a clang feature that was designed with the goal to improve clang frontend performance. The basic idea was to create AST for a header file and reuse the AST for some purposes.

9.1.1 User guide

Generate you pch file is simple [Community, 2022a]. Suppose you have a header file with name header.h:

```
#pragma once

```

```
void foo() {
}
```

then you can generate a pch for it with

```
clang -x c++-header header.h -o header.pch
```

the option `-x c++-header` was used there. The option says that the header file has to be treated as a c++ header file. The output file is `header.pch` .

The precompiled headers generation is not enough and you may want to start using them. Typical C++ source file that uses the header may look like

```
// test pchs

#include "header.h"

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

As you may see, the header is included as follows

```
  ...
#include "header.h"
  ...
```

By default clang will not use a pch at the case and you have to specify it explicitly with

```
clang -include-pch header.pch main.cpp -o main -lstdc++
```

We can check the command with debugger and it will give us

```
$ lldb ~/local/llvm-project/build/bin/clang -- -cc1 -include-pch header.pch main.cpp -fsyntax-only
...
(lldb) b clang::ASTReader::ReadAST
...
(lldb) r
...
   4231   llvm::SaveAndRestore<SourceLocation>
-> 4232     SetCurImportLocRAII(CurrentImportLoc, ImportLoc);
   4233   llvm::SaveAndRestore<Optional<ModuleKind>> SetCurModuleKindRAII(
   4234       CurrentDeserializingModuleKind, Type);
   4235
(lldb) p FileName
(llvm::StringRef) $0 = (Data = "header.pch", Length = 10)
```

Note that only the first `-include-pch` option will be processed, all others will be ignored. It reflects the fact that there can be only one precompiled header for a translation unit.

9.2 Modules

Modules can be considered as a next step in evolution of precompiled headers. They also represent an parsed AST in binary form but form a DAG (tree) i.e. one module can include more than one another module [1]

9.2.1 User guide

.

The C++20 standard [for Standardization, 2020] introduced 2 concepts related to modules. The first one is ordinary modules described at section 10 of [for Standardization, 2020]. Another one is so call header unit that is mostly described at section 15.5. The header units can be considered as an intermediate step between ordinary headers and modules and allow to use `import` directive to import ordinary headers. The second approach was the main approach for modules implemented in clang and we will call it as implicit modules. The

[1]Compare that with precompiled header where only one precompiled header can be introduced for each compilation unit

first one (primary one described at [for Standardization, 2020]) will be call explicit modules. We will start with the implicit modules first.

9.2.2 Implicit modules

The key point for implicit clang modules is `modulemap` file. It describes relation between different modules and interface provided by the modules. The default name for the file is `module.modulemap` . Typical content is the following

```
module header1 {
  header "header1.h"
  export *
}
```

The header paths in the modulemap file has to be ether absolute or relative on the module map file location. Thus compiler should have a chance to find them to compile.

There are 2 options to process the configuration file: explicit or implicit. The first one (explicit) assumes that you pass it via `-fmodule-map-file=<path to modulemap file>` . The second one (default) will search for modulemap files implicitly and apply them. You can turn off the behaviour with `-fno-implicit-module-maps` command line argument.

9.2.2.1 Explicit modules

TBD

9.2.2.2 Some problems related to modules

The code that uses modules can introduce some non trivial behaviour of your program. Consider the project that consists of two headers.

`header1.h` :

```
#pragma once

int h1 = 1;
```

`header2.h` :

```
#pragma once

int h2 = 2;
```

The `header1.h` is included into the `main.cpp`

```
#include <iostream>

#include "header1.h"

int main() {
  std::cout << "Header1 value: " << h1 << std::endl;
  std::cout << "Header2 value: " << h2 << std::endl;
}
```

The code will not compile

```
clang++ -std=c++20 -fconstexpr-depth=1271242 main.cpp -o main
↪ -lstdc++
main.cpp:7:37: error: use of undeclared identifier 'h2'
  std::cout << "Header2 value: " << h2 << std::endl;
                                    ^
1 error generated.
```

but if you use the following `module.modulemap` file then it will compile with modules

```
module h1 {
  header "header1.h"
  export *
  module h2 {
    header "header2.h"
    export *
  }
}
```

The example shows how the visibility scope can be leaked when modules are used in the project.

9.2.3 Modules internals

Modules are processed inside `clang::Preprocessor::HandleIncludeDirective`. There is a `clang::Preprocessor::HandleHeaderIncludeOrImport` method.

The module is loaded by `clang::CompilerInstance::loadModuleFile`. The method caalls `clang::CompilerInstance::findOrCompileModuleAndReadAST`

9.3 Header-Map files

TBD

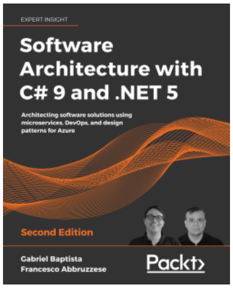
Other Books You Might Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

Modern Computer Architecture and Organization

Jim Ledin
ISBN: 978-1-83898-439-7

- Get to grips with transistor technology and digital circuit principles
- Discover the functional elements of computer processors
- Understand pipelining and superscalar execution
- Work with floating-point data formats
- Understand the purpose and operation of the supervisor mode
- Implement a complete RISC-V processor in a low-cost FPGA
- Explore the techniques used in virtual machine implementation
- Write a quantum computing program and run it on a quantum computer



Software Architecture with C# 9 and .NET 5 - Second Edition

Gabriel Baptista, Francesco Abbruzzese
ISBN: 978-1-80056-604-0

- Use different techniques to overcome real-world architectural challenges and solve design consideration issues
- Apply architectural approaches such as layered architecture, service-oriented architecture (SOA), and microservices

- Leverage tools such as containers, Docker, Kubernetes, and Blazor to manage microservices effectively
- Get up to speed with Azure tools and features for delivering global solutions
- Program and maintain Azure Functions using C# 9 and its latest features
- Understand when it is best to use test-driven development (TDD) as an approach for software development
- Write automated functional test cases
- Get the best of DevOps principles to enable CI/CD environments

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished NameOfTheProduct, we'd love to hear your thoughts! Scan the QR code below to go straight to the Amazon review page for this book and share your feedback.

https://packt.link/r/<ISBN10P>

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

Replace NameOfTheProduct with your title name.

Replace the dummy QR Code image with your product QR code image.

Replace ISBN10P with the 10-digit ISBN-10P from EPIC.

This page should be added after the Front Matter.

Index

Bibliography

LLVM Community. Moving llvm projects to github, 2019. URL https://llvm.org/docs/Proposals/GitHubMove.html.

LLVM Community. Clang compiler user's manual, 2022a. URL https://clang.llvm.org/docs/UsersManual.html.

LLVM Community. [llvm] update c++ standard to 17, 2022b. URL https://reviews.llvm.org/D130689.

LLVM Community. Llvm releases, 2022c. URL https://releases.llvm.org/.

LLVM Community. Building llvm with cmake, 2023. URL https://llvm.org/docs/CMake.html.

"LLVM Community". "msvc compatibility", "2023". URL "https://clang.llvm.org/docs/MSVCCompatibility.html".

LLVM Community. Clang features, 2023a. URL https://clang.llvm.org/features.html.

LLVM Community. Llvm coding standards, 2023b. URL https://llvm.org/docs/CodingStandards.html.

LLVM Community. Commandline 2.0 library manual, 2023c. URL https://llvm.org/docs/CommandLine.html.

LLVM Community. Getting started with the llvm system, 2023d. URL https://llvm.org/docs/GettingStarted.html.

Keith Cooper and Linda Torczon. Engineering A Compiler. Elsevier Inc., 2nd edition, 2012. ISBN 978-0-12-088478-0.

International Organization for Standardization. International standard iso/iec 14882:2017(e) – programming languages – c++, 2017. URL https://www.iso.org/standard/69466.html.

International Organization for Standardization. International standard iso/iec 14882:2020(e) – programming languages – c++, 2020. URL https://www.iso.org/standard/73560.html.

"Alexandre Ganea". [clang][driver] re-use the calling process instead of creating a new process for the cc1 invocation", "2019". URL "https://reviews.llvm.org/D69825".

International Organization for Standardization (ISO). ISO/IEC 9899:1999 - Programming languages - C, 1999. URL https://www.iso.org/standard/23482.html.

Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Mar 2004.

Wikipedia contributors. Google test — Wikipedia, the free encyclopedia, 2022a. URL https://en.wikipedia.org/w/index.php?title=Google_Test&oldid=1123707063. [Online; accessed 23-December-2022].

Wikipedia contributors. Opencv — Wikipedia, the free encyclopedia, 2022b. URL https://en.wikipedia.org/w/index.php?title=OpenCV&oldid=1113479408. [Online; accessed 23-December-2022].

Wikipedia contributors. Qt (software) — Wikipedia, the free encyclopedia, 2022c. URL https://en.wikipedia.org/w/index.php?title=Qt_(software)&oldid=1125020271. [Online; accessed 23-December-2022].