Clang compiler frontend

Ivan Murashko

November 4, 2022

Contents

Chapter 1

Introduction

The `clang` is C/C++ and ObjC compiler. It's an integral part of LLVM project. When we say about clang we can say about 2 different things. The first one is the compiler frontend i.e. the part of compiler that is responsible for parsing and semantic reasoning about the program. We also use the word `clang` when we say about the compiler itself. It's also referred as compiler driver. The driver is responsible for compiler invocation i.e. it can be considered as a manager that calls different parts of the compiler such as the compiler frontend as well as other parts that are required 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 book is separated into 2 parts. The first one provides basic info about LLVM project and how it can be installed. It also describes useful development tools and configurations used for LLVM code exploration later in the book. The internal `clang` architecture is the next primary topic for the first part of the book. The knowledge about `clang` internals and its place inside LLVM is essential for any development related to `clang` . The final topic for the first part is compilation performance and especially how it can be improved. You can find a description for several `clang` features that might significantly improve the compilation speed. There are C++ modules, header maps and others.

The `clang` follows primary paradigm of LLVM - everything is a library, that allows to create a bunch of different tools. The second part of the book is about such tools. We consider clang-tidy - powerful framework to create lint checks. We consider simple checks based on AST (abstract syntax tree) matching as well as more powerful ones based on advanced techniques such as CFG (control flow graph). The list of tools is not limited by the code

analysis but also include refactoring tools as well as IDE support.

The book also includes a lot of different examples that can be found at [4].

Part I

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.

Chapter 2

Environment setup

The chapter describes basic steps to be done to set up the environment to be used for future experiments with a clang. The setup is appropriate for Unix-based systems such as Linux and Darwin (MacOS). In addition, the reader will get important info on how to download, configure and build LLVM source code.

2.1 Prerequisite

Our primary goal will be clang frontend investigation and that will assume some prerequisites that has to be installed.

There are

1. OS requirement (Linux, Darwin)
2. build tools (cmake, ninja)
3. debugger lldb

2.2 LLVM history and project structure

There is a short history of LLVM project and brief overview into its organization

2.2.1 Getting the source code

The clang source is a part of LLVM. You can get it with the following command

```
git clone https://github.com/llvm/llvm-project.git
cd llvm-project
```

2.3 Source code compilation

We are going to compile our source code in debug mode to be suitable for future investigations with debugger.

2.3.1 Configuration with cmake

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

```
mkdir build
cd build
```

Run configure script

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

The are several options specified:

- `-DLLVM_TARGETS_TO_BUILD="X86"` specifies exact targets to be build. It will avoid build unnecessary targets
- `LLVM_ENABLE_PROJECTS="clang;clang-tools-extra"` specifies LLVM projects that we care about
- `LLVM_USE_LINKER=gold` - uses gold linker
- `LLVM_USE_SPLIT_DWARF=ON` - spits debug information into separate files. This option saves disk space as well as memory consumption during the LLVM build. The option require compiler used for clang build to support it

For debugging purposes you might want to change the `-DCMAKE_BUILD_TYPE` into `Debug`. Thus you overall config command will look like

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

2.3.2 Build

The build is trivial

```
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` at the build folder.

Chapter 3

Basic libraries and tools

LLVM has been written in C++ language and currently (July 2022) uses the c++17 version of the C++ standard [3]. 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.

3.1 Basic libs

TBD

3.2 TableGen

TBD

3.3 LIT: LLVM test framework

TBD

Chapter 4

Architecture

You can find some information about the clang's internal architecture and its relationship with other LLVM components. We will start with the clang driver - the backbone for the compiler as soon as it runs all compilation phases and controls their execution. Next, we will focus on the first part of the frontend that includes lexical and semantic analysis and produces AST (abstract syntax tree) as the primary output. The AST is the foundation for most clang tools, and we will look at it more deeply.

4.1 Clang driver overview

When we spoke about `clang` we should separate 2 things:

- driver
- compiler frontend

Both of them are called `clang` but perform different operations. The driver does the following:

- it invokes and controls different stages of compilation process (see fig. 4.1)
- it also produces the final result of the compilation (executable, object file, etc.)
- the driver is responsible for check environment and setup correct compilation flags, for instance be sure that `/usr/include` is in the search paths for Linux OS.

The stages are standard for ordinary compiler and nothing special is there:

Figure 4.1: Clang driver

- Frontend: it does lexical analysis and parsing.
- Middle-end: it does different optimization on the intermediate representation (LLVM-IR) code
- Backend: Native code generation
- Assembler: Running assembler
- Linker: Running linker

The driver is invoked by the following command

```
clang main.cpp -o main -lstdc++
```

The driver also adds a lot of additional arguments, for instance search paths for system includes that could be platform specific. You can use `-###` clang option to print actual command line used by the driver

```
$clang -### main.cpp -o main -lstdc++
clang version 12.0.1 (Fedora 12.0.1-1.fc34)
Target: x86_64-unknown-linux-gnu
Thread model: posix
InstalledDir: /usr/bin
 "/usr/bin/clang-12" "-cc1" "-triple"
  ↪  "x86_64-unknown-linux-gnu" "-emit-obj" "-mrelax-all" ...
```

4.2 Clang frontend overview

One may see that the clang compiler toolchain corresponds the pattern wildly described at different compiler books [5]. Despite the fact, the frontend part is quite different from a typical compiler frontend described at the books. The primary reason for this is C++ language and it's complexity. Some features (macros) can change the source code itself another (typedef) can effect on token kind. As result the relations between different frontend components can be represented as it's shown on fig 4.2

Figure 4.2: Clang frontend components

You can bypass the driver call and run the clang frontend with `-cc1` option. We are going to use it for our future experiments with one simple program:

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

The first part is the **Lexer**. It's primary goal is to convert the input program into a stream of tokens. The token stream can be printed with `-dump-tokens` options as follows

```
$ clang -cc1 -dump-tokens src/simple/main.cpp
```

The output of the command is below

```
int 'int'        [StartOfLine]  Loc=<src/simple/main.cpp:1:1>
identifier 'main'        [LeadingSpace]
↪  Loc=<src/simple/main.cpp:1:5>
l_paren '('             Loc=<src/simple/main.cpp:1:9>
r_paren ')'             Loc=<src/simple/main.cpp:1:10>
l_brace '{'      [LeadingSpace] Loc=<src/simple/main.cpp:1:12>
return 'return'  [StartOfLine] [LeadingSpace]
↪  Loc=<src/simple/main.cpp:2:3>
numeric_constant '0'     [LeadingSpace]
↪  Loc=<src/simple/main.cpp:2:10>
semi ';'                Loc=<src/simple/main.cpp:2:11>
r_brace '}'      [StartOfLine]  Loc=<src/simple/main.cpp:3:1>
eof ''          Loc=<src/simple/main.cpp:3:2>
```

Another part is the **Parser**. It produces AST that can be shown with the following command

```
$ clang -cc1 -ast-dump src/simple/main.cpp
```

The output of the command is below

```
TranslationUnitDecl 0x5581925de3b8 <<invalid sloc>> <invalid
↪  sloc>
|-TypedefDecl 0x5581925dec20 <<invalid sloc>> <invalid sloc>
↪  implicit __int128_t '__int128'
| `-BuiltinType 0x5581925de980 '__int128'
|-TypedefDecl 0x5581925dec90 <<invalid sloc>> <invalid sloc>
↪  implicit __uint128_t 'unsigned __int128'
| `-BuiltinType 0x5581925de9a0 'unsigned __int128'
|-TypedefDecl 0x5581925df008 <<invalid sloc>> <invalid sloc>
↪  implicit __NSConstantString '__NSConstantString_tag'
| `-RecordType 0x5581925ded80 '__NSConstantString_tag'
|   `-CXXRecord 0x5581925dece8 '__NSConstantString_tag'
|-TypedefDecl 0x5581925df0a0 <<invalid sloc>> <invalid sloc>
↪  implicit __builtin_ms_va_list 'char *'
| `-PointerType 0x5581925df060 'char *'
|   `-BuiltinType 0x5581925de460 'char'
|-TypedefDecl 0x5581926237d8 <<invalid sloc>> <invalid sloc>
↪  implicit __builtin_va_list '__va_list_tag[1]'
| `-ConstantArrayType 0x558192623780 '__va_list_tag[1]' 1
|   `-RecordType 0x5581925df190 '__va_list_tag'
|     `-CXXRecord 0x5581925df0f8 '__va_list_tag'
`-FunctionDecl 0x558192623880 <src/simple/main.cpp:1:1,
↪  line:3:1> line:1:5 main 'int ()'
  `-CompoundStmt 0x5581926239c0 <col:12, line:3:1>
    `-ReturnStmt 0x5581926239b0 <line:2:3, col:10>
      `-IntegerLiteral 0x558192623990 <col:10> 'int' 0
```

4.3 Clang AST

The AST is the skeleton for clang frontend. It's also the primary instrument for linters and other clang tools. The AST keeps result of syntax and semantic analysis and represent a tree with leafs for different objects, such as function declaration for loop body etc. The clang provides advanced tools for search (match) different nodes. The tools are implemented in the form of DSL (domain specific language). There is important to understand how its implemented to be able using it.

TBD

Chapter 5

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.

5.1 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.

5.1.1 User guide

Generate you pch file is simple [2]. 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.

5.2 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]

5.2.1 User guide

.

The C++20 standard [1] introduced 2 concepts related to modules. The first one is ordinary modules described at section 10 of [1]. 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 first one (primary one described at [1]) will be call **explicit modules**. We will start with the implicit modules first.

5.2.2 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.

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

Explicit modules

TBD

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.

5.2.3 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`

5.3 Header-Map files

TBD

Part II

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).

Chapter 6

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++/ObjC 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.

6.1 clang-tidy overview and usage examples

TBD

6.2 clang-tidy internal design

TBD

6.3 Custom clang-tidy check

TBD

Chapter 7

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.

7.1 Usage cases

TBD

7.2 CFG and life time analysis

TBD

7.3 Custom CFG check

TBD

Chapter 8

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.

8.1 Code modification and clang-format

TBD

8.2 Custom code modification tool

TBD

Chapter 9

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.

9.1 VSCode and LSP

TBD

9.2 clangd internals

TBD

9.3 Custom extension for LSP

TBD

Index

Bibliography

[1] *C++20*. 2021.

[2] Clang compiler user's manual, 2022.

[3] [llvm] update c++ standard to 17, 2022.

[4] I. Murashko. Source code examples for clang compiler fronmtend, 2022.

[5] L. Torczon and K. Cooper. *Engineering A Compiler*. Elsevier Inc., 2nd edition, 2012.