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 disadvan-

tage 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

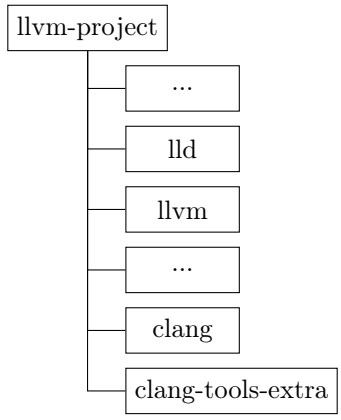

Figure 1.1: LLVM project tree: there are projects that are important for the book: lldb (LLDB debugger), llvm (some common libs are located here), clang (C/C++ compiler frontend and driver) and clang-tools-extra (different clang tools such as clang-tidy and clangd)

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

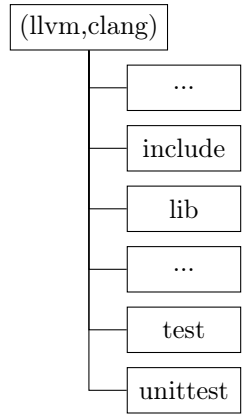

Figure 1.2: Typical LLVM project structure: the primary source code is located at include and lib folders. The test folder keeps LIT end-to-end tests and unittest folder keeps tests written for google test framework

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

```
1 int main() {
2   return 0;
3 }
```

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

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

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