Clang LLVM frontend as a modern C++ source-code generation tool

Research and Development Project

Name	Student Number
Morten Haahr Kristensen	201807664
Mikkel Kirkegaard	201808851

Supervisor	Email
Lukas Esterle	lukas.esterle@ece.au.dk

Department of Electrical and Computer Engineering
Aarhus University

March 16, 2023

Abstract

Contents

1 Introduction

2 Background

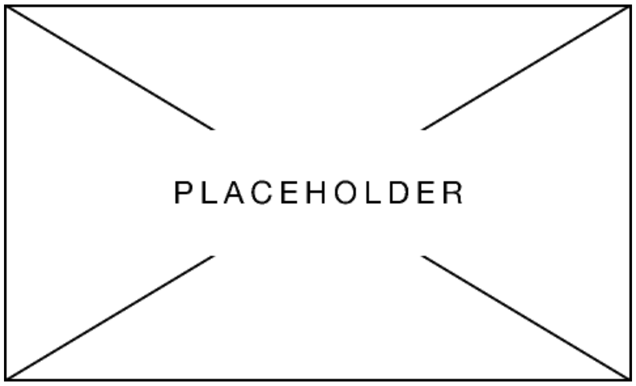

Figure 2.1: LLVM and Clang overview

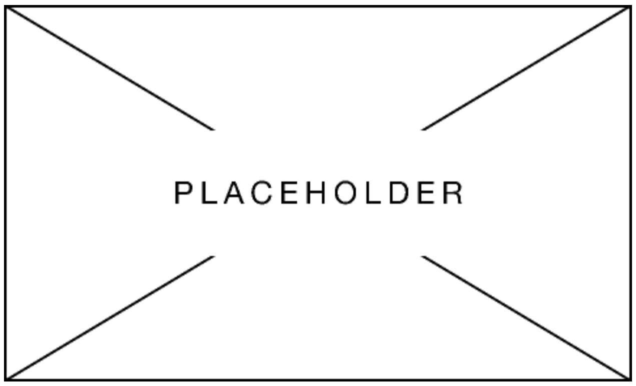

Figure 2.2: Clang and LibTooling overview

3 Project description

4 Methods

5 Requirements

5.1 Functional requirements

5.2 Non-functional requirements

6 Architecture

7 Development

7.1 Installing LLVM and Clang

One might think installing LLVM and Clang should be a straightforward process but in reality, it can be quite complex. Therefore, this section means to describe the process that was used during the R&D project. The section is heavily inspired by [1] but more specialized to account for the concrete project.

The process of compiling LLVM, Clang and LibTooling can be considered a two-step process where they initially are compiled using an arbitrary C++ compiler and then compiled using Clang itself.

Before compiling the projects, one must install the needed tools, which include ”CMake” and ”Ninja”. They can be installed using a package manager or by compiling it locally through `https://github.com/martine/ninja.git` and `git://cmake.org/stage/cmake.git`. Furthermore, one needs to have a working C++ compiler installed. The rest of this section assumes the compiler GCC is installed.

LLVM and Clang can then be compiled for the first time using GCC. Assuming the terminal is used, this can be done as seen in listing 7.1. First, the LLVM repository is cloned which also contains the Clang project. Line 2 - 4 goes inside the repository and sets up the build folder. Line 5 uses CMake to configure the project where Ninja is used as the generator[1], Clang and Clang Tools are enabled, tests are enabled and it should be built in release mode. The final line instructs Ninja to build the projects. Note that it is possible to skip building and running the tests but it is recommended to ensure that the build process succeeded.

```bash
1  git clone https://github.com/llvm/llvm-project.git
2  cd llvm-project
3  mkdir build
4  cd build
5  cmake -G Ninja ../llvm -DLLVM_ENABLE_PROJECTS="clang" -DLLVM_BUILD_TESTS=ON -DCMAKE_BUILD_TYPE=Release
6  ninja
```

Listing 7.1: Bash commands to initially compile LLVM and Clang.

The next steps consist of testing the targets to ensure that the compilation went successfully. This is done by running the tests as seen in the two first lines on listing 7.2. Finally, the initial version of Clang that is compiled with an arbitrary compiler is installed.

[1]What is a generator

```
ninja check
ninja clang-test
sudo ninja install
```

Listing 7.2: Bash commands to test the LLVM and Clang projects and then finally install them.

Clang should now be compiled using Clang to avoid name mangling issues [2], i.e., ensure that the symbolic names the linker assigns to library functions do not overlap. This time the project "cmake-tools-extra" should also be included to build LibTooling and the complementary example projects. The option `-DCMAKE_BUILD_TYPE=RelWithDebInfo` is also a possibility if one wishes to include debug symbols in the libraries. This however comes with a performance trade-off, as Clang itself will also be compiled with debug symbols, but can be useful during development. The group has yet to find a way to compile LibTooling with debug symbols but Clang without it.

```
cmake -G Ninja ../llvm -DLLVM_ENABLE_PROJECTS="clang;clang-tools-extra" -DCMAKE_BUILD_TYPE=Release
↪  -DCMAKE_CXX_COMPILER=clang++
ninja
```

Listing 7.3: Bash commands to compile LLVM, LibTooling and Clang with Clang as compiler.

Finally, the steps from listing 7.2 should be repeated to verify the new compilation and install the tools.

7.2 Build environment

The documentation for writing applications using LibTooling such as [3, 4] mainly concerns writing tools as part of the LLVM project repository. While this is good for contributing to the project, it is not ideal for version control and developing stand-alone projects. It was necessary to create a build environment that allowed for out-of-tree builds that utilize LibTooling. A similar attempt was made in [5] but the project has been abandoned since 2020 and LLVM has since moved from a distributed repository architecture to a monolithic repository architecture, so most of [5] was obsolete. The following section is dedicated to describing the important decisions made related to the build environment.

7.2.1 Build settings

Initially, some general settings for the project are configured which can be seen in listing 7.4. Line 1 forces Clang as the compiler which is highly recommended as LibTooling was compiled with Clang. Choosing another compiler may result in parts of the project being compiled with another standard library implementation, e.g., libstd++ that is the default for GCC. This may cause incompatibility between the application binary interfaces (ABIs) which is considered undefined behaviour, essentially leaving the entire program behaviour unspecified [6]. This concept is also known as ABI breakage. Line 2 makes CMake generate the file "compile_commands.json" during configuration and contains information related to which compilation command should be invoked on which source file. The file is used by LibTooling as it is needed in order to generate the AST across several compilation units in order to determine their relations [7]. It should be enabled in all the CMake projects that should allow for Clang AST analysis. Line 3 defines the C++

standard version, which is set to C++17 since LibTooling was compiled with this. Line 4 defines the output directory of the executable to be in `<build_folder>/bin` which has importance concerning how LibTooling searches for include directories at run-time as described in section 7.2.2. Finally, line 5 disables Run-Time Type Information (RTTI). RTTI allows the program to identify the type of an object at runtime, among others, by enabling the usage of the functions `dynamic_cast` and `typeid`. Since LLVM leaves it to the user whether RTTI should be used or not, it will attempt to use an implementation with RTTI if it is not explicitly disabled resulting in a nasty linker error when the implementation has not been compiled.

```
1 set(CMAKE_CXX_COMPILER clang++)
2 set(CMAKE_EXPORT_COMPILE_COMMANDS 1)
3 set(CMAKE_CXX_STANDARD 17)
4 set(CMAKE_RUNTIME_OUTPUT_DIRECTORY "${CMAKE_BINARY_DIR}/bin")
5 add_compile_options(-fno-rtti)
```

Listing 7.4: General settings for the CMake build environment.

7.2.2 Run-time include directories

When executing binaries created with LibTooling, a big part of the process consists of analyzing the source code that is to be analyzed, which is done following the pipeline as shown in fig. 2.1. Most projects written in C++ make use of the C++ standard library that implements many commonly used functionalities in C++. Naturally, these definitions would also have to be specified to the tool in order to be analyzed, however, it would be tedious for the user to specify manually on every tool invocation. For practical reasons, LibTooling provides a mechanism for the automatic discovery of header files that should be included when parsing source files. It finds the headers by using the relative path that the binary is being executed from by searching for a parent directory with the pattern `lib/clang/<std_version>/include`, where `<std_version>` indicates which version of the standard library Clang was compiled with (in this project it was 17). This hard-coded approach is quite simple but limited, as it forces the users to only run the tool in a directory where the headers can be found in the relative directory `<current_dir>/../lib/clang/17/include`. If the user attempts to run it somewhere else, and the analyzed files make use of standard library features, they will get an error while parsing the files (e.g. that the header `<stddef.h>` was not found). This issue makes it more difficult to write truly independent tools as they still need some reference to the Clang headers, which would essentially mean moving the executable to the directory where Clang was compiled.

One existing solution is to provide the location of the headers as an argument to the binary when executed. This is possible since tools written with LibTooling invoke the parser of Clang, from where it is possible to forward the include directory as an argument to the compiler e.g. by specifying `-- -I"/usr/local/lib/clang/17"`. However, this was found to be impractical since the location of the include path may vary depending on the system and forgetting to write the path results in errors that can be very difficult to decipher.

Instead, it was decided to create a build environment where the user must provide the location of the Clang headers when configuring CMake or an appropriate error message is generated. Through CMake, the necessary headers are then copied to the build directory.

The solution is by no means perfect, as the user is still forced to execute the binary from the build directory. In many situations, this is sufficient, as most IDEs follow this behaviour as default and it allows the projects to be built out-of-tree. If the user wishes to run the binary from outside the build directory, they still have the option of specifying the location through the `-- -I"<clang_include>"` option. The solution can be found in the functions.cmake file.

In the future, it may be desirable to explore a solution using the LLVM command line library to search some commonly used directories for the Clang headers.

7.3 Transformer

Clang provides an interface into its C++ AST called LibTooling. LibTooling is aimed at developers who want to build standalone tools and services that run clang tools.[8]

Tools that use LibTooling run what is called 'FrontendActions' over the specified code. It is through these frontend actions the tool can interact with the source code. The LibTooling tools work by parsing the command line options the tool is invoked with through the llvm command line parser. These options can be customised to allow the tools to work in a user-defined manner. After the command options are parsed, the tool needs an ASTMatcher. The ASTMatcher is the DSL formula for traversing the Clang C++ AST. There are many different matchers which allow for extensive and custom matching of the AST.[9, 3] If the matcher finds a valid C++ AST a call to a user-defined *Consumer* callback is made with the matched AST. This allows the user to further process the AST and to perform the task the tool is meant to solve. It is also possible to write custom matchers if the built-in ones are not enough to solve the task.

LibTooling is used in many different types of tools such as static analysis tools, code refactoring, language standard migration tools and much more.[10]

Because the Clang AST is so comprehensive, the Clang development team has provided a construct called a `Transformer`. A Transformer is a way to couple a *rule* together with a consumer callback. A rule is a combination of an AST matcher, a *change* and some metadata. A rule could be specified as follows: "The name 'MkX' is not allowed in our code base, so find all functions with the name 'MkX' and change it to 'MakeX"'.This could be translated into LibTooling rule:

```
auto RenameFunctionWithInvalidName = makeRule(
    functionDecl(hasName("MkX")).bind("fun"),
    changeTo(clang::transformer::name("fun"), cat("MakeX")),
    cat("The name ``MkX`` is not allowed for functions; the function has been renamed")
);
```

Listing 7.5: Example of a LibTooling Rule that renames a method 'MkX' to 'MakeX' and provides a reason for the renaming.

Where `functionDecl(hasName("MkX")).bind("fun")` is the ASTMatcher that matches all function declarations with the name "MkX" and binds it to the name "fun". The line `changeTo(clang::transformer::name("fun"), cat("MakeX"))` is specifying the change to be made, which is to change the name of match bound to "fun" to "MakeX". The last line in listing 7.5 is the metadata that is associated with this rule. When the rule matches the wanted AST expression it creates a `AtomicChange` according to the specified change.

This is then where the `Transformer` comes in. The transformer takes the resulting `AtomicChange` and the provided metadata and calls a *Consumer* callback with the findings. In this callback, the developers of the tool are then able to provide diagnostic messages and make source code changes if that is what the tool is providing.[4]

In the official documentation from Clang transformer shown in listing 7.5 is shown and explained. However, there is no official documentation on how to invoke the transformer and make it write the changes into the existing source code.

In the public repository for this R&D project, there is an example folder. This example folder contains multiple examples which show how to create Clang tools that write the results from a `Transformer` onto disk.[11] In the examples the `RefactoringTool` helper class is used to facilitate the source-code changes. This helper class is not documented in Clang's official documentation, but it contains helper methods that make source code changes easier for the developers.

7.4 Example of a C-style to std::array converter

This section will show an example of a tool that can find all constant-sized arrays and convert them to the C++11 `std::array` type. This transformation is recommended in the C++ core guidelines.[12]

When creating a tool there are multiple steps. The steps are:

1. Matching the AST nodes
2. Extracting information from the nodes
3. Creating the replacement source code

The github repo for this R&D project contains an example project with all the source code described in this section under "examples/c_style_conversion_tool".

7.4.1 Matching the AST

The first step in the process is to find the AST nodes of interest. In this example the declarations of arrays with a constant size. In order to specify the matcher needed it is essential to have some valid C++ code that contains the source code that needs to be changed. For this example tool, the constant array types are specified in the "examples/c_style_conversion_tool/input.cpp" file on github.

In order to get an idea of the type of AST nodes that are interesting to match, the AST for the entire file is "dumped" through clang by running `clang -cc1 -ast-dump input.cpp`. This command will show all of the AST for the file. The output of the command can be seen on

From the output it can be seen that a constant-sized array declaration consists of a DeclStmt that is a VarDecl

7.4.2 Getting information out of the nodes

7.4.2.1 Stencils

7.4.3 Creating the replacements

7.4.3.1 Range selection

7.4.4 Combining it all

8 Testing

9 Related work

10 Conclusion

10.1 Future work

References

[1] *Tutorial for Building Tools Using LibTooling and LibASTMatchers — Clang 17.0.0git Documentation.* URL: https://clang.llvm.org/docs/LibASTMatchersTutorial.html (visited on 02/02/2023).

[2] IBM. *IBM Documentation.* Apr. 14, 2021. URL: https://ibm.com/docs/en/i/7.3?topic=linkage-name-mangling-c-only (visited on 02/28/2023).

[3] *Matching the Clang AST — Clang 17.0.0git Documentation.* URL: https://clang.llvm.org/docs/LibASTMatchers.html (visited on 02/08/2023).

[4] *Clang Transformer Tutorial — Clang 17.0.0git Documentation.* URL: https://clang.llvm.org/docs/ClangTransformerTutorial.html (visited on 02/08/2023).

[5] Firat Kasmis. *Clang Out-of-Tree Build.* Jan. 28, 2023. URL: https://github.com/firolino/clang-tool (visited on 02/28/2023).

[6] cppreference. *Undefined Behavior - Cppreference.Com.* URL: https://en.cppreference.com/w/cpp/language/ub (visited on 02/28/2023).

[7] *JSON Compilation Database Format Specification — Clang 17.0.0git Documentation.* URL: https://clang.llvm.org/docs/JSONCompilationDatabase.html (visited on 02/16/2023).

[8] *LibTooling — Clang 17.0.0git Documentation.* URL: https://clang.llvm.org/docs/LibTooling.html (visited on 02/23/2023).

[9] *AST Matcher Reference.* URL: https://clang.llvm.org/docs/LibASTMatchersReference.html (visited on 02/07/2023).

[10] *External Clang Examples — Clang 17.0.0git Documentation.* URL: https://clang.llvm.org/docs/ExternalClangExamples.html (visited on 02/23/2023).

[11] Morten Haahr Kristensen. *Mortenhaahr/RD.* Feb. 23, 2023. URL: https://github.com/mortenhaahr/RD (visited on 02/23/2023).

[12] *C++ Core Guidelines.* URL: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Res-stack (visited on 03/03/2023).