2.1. Manual compilation

Pieter P

In this section, we'll manually invoke the compiler to build a very simple "Hello, World"-style program that consists of multiple source files. The goal is to illustrate how building software works under the hood, and motivate the use of a build system or build system generator to automate this tedious task.

2.1.1. Example project

Consider a small project that consists of two libraries and a main program.

- The first library (A) defines the `say_hello()` function. It uses the `{fmt}` library to print a greeting to the console.
- The second library (B) uses the `say_hello()` function from library A to implement the `greet_many()` function that greets multiple people.
- Finally, the main program creates an array with names of people to greet, and calls library B's `greet_many()` function.

The project layout could look something like this:

```
├── liba
│   ├── a.cpp
│   └── a.hpp
├── libb
│   ├── b.cpp
│   └── b.hpp
└── main.cpp
```

Source code

The source code listed below is not too important in and of itself, what matters are the dependencies between the different source files.

Header A (liba/a.hpp)

```cpp
#pragma once

#include <string_view>

/// Print a greeting message.
void say_hello(std::string_view name);
```

Implementation A (liba/a.cpp)

```cpp
#include <fmt/core.h>
#include <a.hpp>

void say_hello(std::string_view name) {
    fmt::print("Hello, {}!\n", name);
}
```

Header B (libb/b.hpp)

```cpp
#pragma once

#include <span>
#include <string_view>

/// Print a greeting for each of the names in the array.
void greet_many(std::span<const std::string_view> names);
```

Implementation B (libb/b.cpp)

```cpp
#include <a.hpp>
#include <b.hpp>

void greet_many(std::span<const std::string_view> names) {
    for (auto name : names)
        say_hello(name);
}
```

Main program (main.cpp)

```cpp
#include <b.hpp>
#include <vector>

int main() {
    std::vector<std::string_view> people{"John", "Paul", "George", "Ringo"};
    greet_many(people);
}
```

2.1.2. Header files and implementation files

Most C++ source files fall into one of two categories: header files and implementation files. Implementation files (usually with extension `.cpp`, `.cc` or `.cxx`) are files that contain function or variable definitions, and are compiled

into executable code. Header files (usually with extension `.hpp`, `.h` or `.hxx`) contain declarations, function prototypes and class definitions. They are not compiled in isolation, but are intended to be included in implementation files or in other header files.
Broadly speaking, the API of a library (i.e. the declarations that are needed to *use* the library) is declared in header files. The actual executable code (i.e. the definitions of the functions provided by the library) exists across multiple implementation files.

2.1.3. The compilation process

The compiler performs different intermediate steps as part of the build process:

1. Pre-processing: including header files by expanding `#include` directives, performing macro substitution, handling `#if` directives, etc.
2. Compilation: parsing and interpreting the pre-processed source code, and composing the corresponding intermediate representation. Along the way, syntactical and semantic errors are reported.
3. Optimization: iteratively rewriting the intermediate representation to improve performance and/or memory usage, without changing the observable behavior of the code.
4. Code generation and assembly: converting the intermediate representation into executable machine code.
5. Linking: combining the machine code for different files and libraries into a single binary.

Pre-processing

The pre-processor's duty is to handle all pre-processor directives, like `#include`, `#if/#elif/#else/#endif`, `#define`, and it performs simple text-based macro expansion. It also strips the comments from the source code.

It is important to note that the preprocessor actually pastes the contents of header files into the source files that include them. This is done using relatively simple text substitution of the `#include` directive, with some bookkeeping to be able to recover the original file names and line numbers. An example of a pre-processed version of the file `b.cpp` from above is listed below. This is the kind of code that is actually handed to the compiler.

Pre-processed source code of implementation B (libb/b.i)

```
# 1 "libb/b.cpp"
# 1 "liba/a.hpp" 1

# 1 "/usr/include/c++/11/string_view" 1 3
/* Thousands of lines of standard library code omitted */
# 4 "liba/a.hpp" 2
                                                        // All comments were removed by the preprocessor
# 6 "liba/a.hpp"
void say_hello(std::string_view name);                  // This is line 6 of a.hpp
# 2 "libb/b.cpp" 2
# 1 "libb/b.hpp" 1

# 1 "/usr/include/c++/11/span" 1 3
/* More standard library code omitted */
# 4 "libb/b.hpp" 2

# 7 "libb/b.hpp"
void greet_many(std::span<const std::string_view> names);         // This is line 7 of b.hpp
# 3 "libb/b.cpp" 2

void greet_many(std::span<const std::string_view> names) {        // This is line 4 of b.cpp
    for (auto name : names)
        say_hello(name);
}
```

The directives starting with a `#` refer back to the original location of the source code. You can see how the headers `a.hpp` and `b.hpp` have been inlined and combined with the other contents of `b.cpp`. This inlining is done recursively: all `#include` directives are expanded, including standard library headers (which have been omitted for clarity). The preprocessor also removes all comments from the code, the comments in the snippet above were added manually to point out the different parts.

Compilation, optimization and code generation

Modern compilers are among the most sophisticated pieces of software out there, and a detailed description falls beyond the scope of this document. TODO: add references

TODO: explain object files

2.1.4. Building the example project

To build the main executable for the example project above, some of the steps are combined, and the two main steps in the build process are:

① Pre-process and compile all implementation files into object files (covers pre-processing, compiling, optimizing, assembling).
② Link all object files and external libraries into an executable.

Using GCC, this can be done as follows:

Build commands

```
mkdir -p build
g++ -std=c++20 -c liba/a.cpp -I liba -o build/a.o           # ┐
g++ -std=c++20 -c libb/b.cpp -I liba -I libb -o build/b.o   # ├ ❶
g++ -std=c++20 -c main.cpp -I libb -o build/main.o          # ┘
g++ build/{a,b,main}.o -lfmt -o build/main                  # — ❷
./build/main
```

① The `-c` option causes GCC to preprocess and compile the given file, writing the resulting object file to the output file specified using the `-o` flag. When preprocessing `a.cpp`, GCC needs to locate the header file `a.hpp`, so the directory containing this file is added to the include path using the `-I` flag. Similarly, the preprocessing of `b.cpp` requires both `liba` and `libb` to be added to the search path, and `main.cpp` needs `libb`. We assume that the `{fmt}` library is installed in `/usr/include` or some other location that is already in GCC's default search path. If this were not the case, we would add another `-I` or `-isystem` flag to inform the preprocessor of its location.
② The invocation of GCC without the `-c` option links the three object files together into an executable. Since `a.cpp` makes use of the `{fmt}` library, we need to link to this library as well, using the `-l` flag. Here we again assume that the `libfmt.so` library is in a standard location, otherwise we would need to use the `-L` flag to add its location to the library search path.

The dependencies between the different files involved in the build process are visualized in Figure 1.

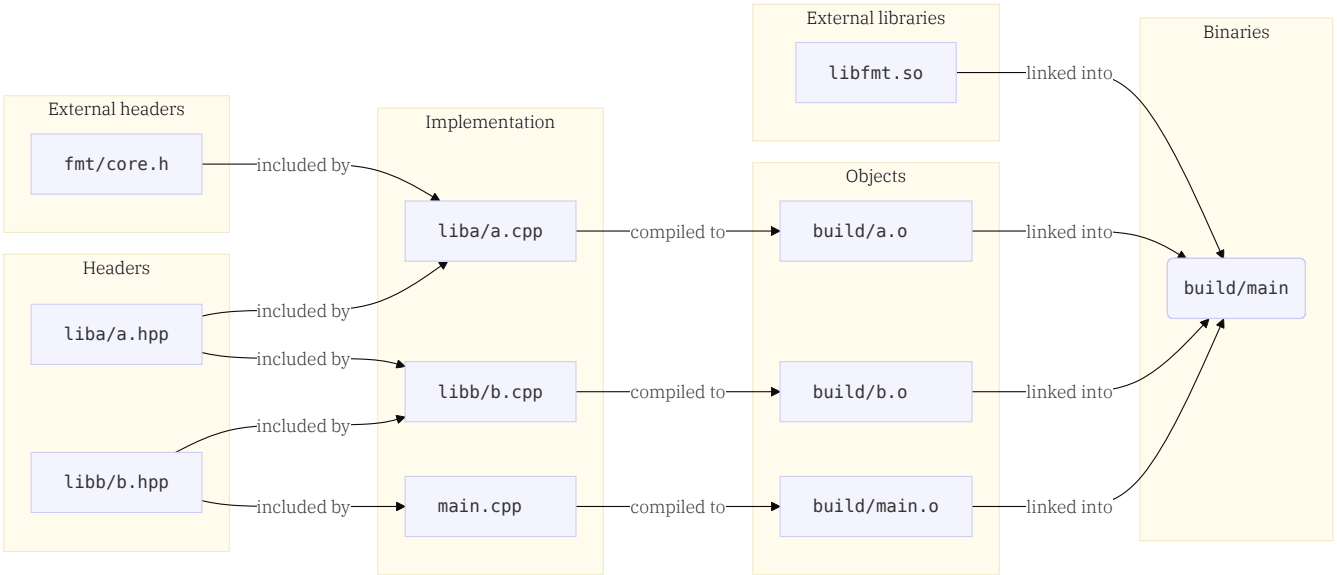

Figure 1: Dependencies between files used during the build process.

2.1.5. Limitations of manual compilation

Manually invoking the compiler like this (with or without a shell script) has some serious downsides and challenges, including:

1. Repetitiveness and boilerplate: Building a C++ project involves many similar calls to the compiler. Although the number of commands to write can be somewhat reduced using wildcards or Bash scripting, we need a more structural solution that is easy to maintain when files are added or when the layout of the project changes.
2. Propagation of compiler flags: Some compiler options (such as preprocessor macros, C++ standard versions and include paths) need to be propagated from a library to its dependents. For example, library B uses library A, so the location of `a.hpp` needs to be added to the search paths when compiling `b.cpp`.
3. Propagation of linker flags: Even though `main.cpp` has no knowledge of the implementation of libraries A and B, it still needs to link to its direct dependency `b.o` and the transitive dependencies `a.o` and `libfmt.so`. The tree of transitive dependencies can become quite unwieldy, and they all need to come together for the final linking step.
4. Incremental compilation: Compiling a larger project in its entirety can often take several minutes. During development, this can become a serious impediment. We therefore only want to recompile the files that actually changed. Since the preprocessor pastes the contents of header files into the source files that include them (using simple text substitution of the `#include` directive), a change in a header file can cascade and require many source files to be recompiled. For example, if `b.hpp` is modified, both `b.cpp` and `main.cpp` have to be recompiled. For larger projects, the *included-by* graph can grow very complex, as demonstrated by Figure 2, and determining these dependencies manually is not an option.
5. Portability: The commands above only work for GCC. Users of your project might want to compile it using Visual Studio on Windows, or Xcode on macOS, or perhaps even cross-compile it for an embedded system. This would require maintaining multiple scripts or build descriptions, with the risk of them getting out of sync.
6. Lack of integration with IDEs and other tools: IDEs like Visual Studio, VS Code, CLion, Xcode, as well as language servers like clangd, or other tools such clang-tidy and Cppcheck won't be able to easily interact with your project if it is described using custom commands or shell scripts.
7. Implicit assumptions about dependencies: In the example above, we assumed that the `{fmt}` library was installed system-wide, with the necessary files in GCC's default search paths. However, this is not always possible, a user may have dependencies installed in their home folder, in a virtual environment, in a subdirectory of the project folder, or somewhere else entirely. Our build script should not make any assumptions or hard-code any paths. We need a standardized way to locate third-party libraries and other dependencies that works across systems.
8. Ease of use: By using a custom script or commands to build your project, you make it harder for users to get started with your software (especially when things don't work as expected), and it also increases the friction for potential contributors that want to help improve your software or build system. There is great value in making the installation of your software as easy as `cmake -Bbuild -S. && cmake --build build -j && cmake --install build`, which is the standard procedure for a majority of (newer) C and C++ projects.

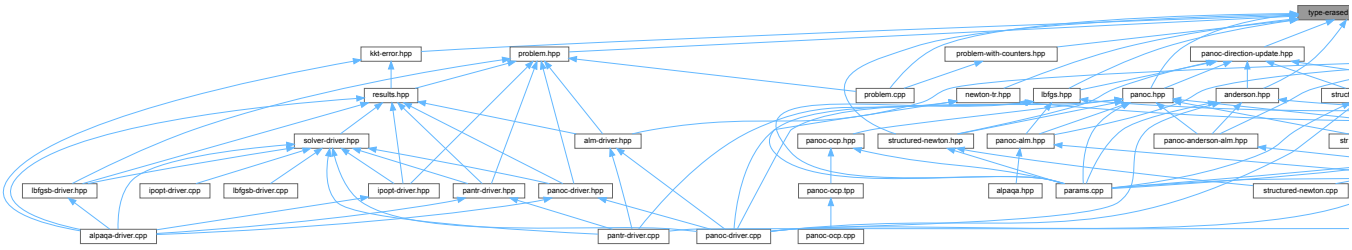

Figure 2: The "included-by" graph of a single header file in a real-world C++ project.

In the next chapter, we'll look at build systems and build system generators that provide solutions to the issues listed above.