Programming Techniques for Scientific Simulations I: A Detailed Textbook

Week 2: The Build Process and `make`

Based on lecture slides

October 27, 2025

Contents

1 Introduction: The Compiler

Welcome to our in-depth discussion on the C++ build process. Before we can write complex scientific simulations, we must first understand the fundamental tool that turns our human-readable ideas into a language the machine can execute: the **compiler**.

A compiler is a sophisticated software program that acts as a master translator. Its job is to take **source code**—the instructions you write in a high-level programming language like C++—and translate it into **machine code**, the low-level binary instructions that a computer's processor understands.

Think of it this way: you write a detailed recipe in English (C++), but the chef (the CPU) only speaks a specific binary dialect. The compiler is the translator who converts your English recipe into a precise, unambiguous set of instructions in the chef's native language.

In this course, we will primarily focus on two of the most powerful and widely-used C++ compilers:

- **GNU Compiler Collection (GCC):** We will use its C++ compiler, `g++`.
- **Clang:** A newer compiler, often used with the LLVM backend.

There are many others, but these are the industry standards on Linux and macOS. These tools are incredibly complex, and a good way to start learning about them is to read their manuals. On a Unix-like system, you can type the following into your terminal:

```
$ man c++
```

This command will display the "manual page" for the C++ compiler, revealing a vast number of options and features.

2 The Four-Stage Assembly Line of Compilation

When you compile a simple program by typing a single command, it feels like an instantaneous, single-step process:

```
$ c++ hello.cpp
```

Behind the scenes, the compiler is executing a complex, four-stage "assembly line" to produce your final program. This entire process can be visualized as a flow:

`hello.cpp` → **Preprocessor** → `hello.ii` → **Compiler** → `hello.s` → **Assembler** → `hello.o` → **Linker** → `a.out`

At the final stage, the Linker also pulls in any necessary **libraries**, which are pre-compiled bundles of code (like `libm.a` for math functions or `libgcc.a` for fundamental compiler support).

Let's break down these four distinct stages. You can observe them yourself by telling the compiler to be "verbose" and to "save temporary files" with these flags:

```
$ c++ --verbose -save-temps hello.cpp
```

This command will run the full process but will leave behind the intermediate files (`hello.ii` and `hello.s`) and the object file (`hello.o`), allowing you to inspect the output of each stage.

2.1 Stage 1: The Preprocessor

The first tool in the chain is the **preprocessor**. It is a relatively simple text-processing tool that manipulates your source code *before* the actual compiler sees it. It doesn't understand C++ syntax; it only obeys special commands called **preprocessor directives**, which are lines that begin with a hash symbol (#).

Real-world Analogy: Imagine the compiler is a master legal translator. Before the translator gets a document, an administrative assistant (the preprocessor) goes through it. The assistant follows simple instructions on sticky notes (`#` directives), like "Find every instance of 'XXX' and replace it with 'Hello'" (`#define`) or "Staple this other document here" (`#include`).

Let's look at the most common directives.

2.1.1 `#define` and `#undef`

The `#define` directive is used to create **macros**. In its simplest form, it's a direct text search-and-replace. By convention, macro names are written in `UPPERCASE` to distinguish them from regular variables.

```
// This is your C++ code (pre1.cpp)
#define XXX "Hello"
std::cout << XXX;
```

After the preprocessor runs, the code given to the compiler is:

```
// This is what the compiler sees
std::cout << "Hello";
```

Macros can also take arguments, making them look like functions:

```
// This is your C++ code (pre2.cpp)
#define SUM(A,B) A+B
std::cout << SUM(3,4);
```

This is transformed by the preprocessor into:

```
// This is what the compiler sees
std::cout << 3+4;
```

Warning - A Common Pitfall: This direct text substitution is blind and can be dangerous. Consider this code: `int x = SUM(2, 3) * 5;` The preprocessor

will turn this into: `int x = 2+3 * 5;` Due to operator precedence (multiplication before addition), `x` will be 17, not 25! A safer (though still not perfect) macro would be: `#define SUM(A,B) ((A)+(B))`

You can also define macros from the command line using the `-D` flag. This is extremely useful for turning features on or off.

```
$ c++ -DXXX=3 -DYYY
```

This is exactly equivalent to adding this to the top of your source file:

```
#define XXX 3
#define YYY
```

The counterpart to defining is `#undef`, which undefines a macro.

```
#define XXX 4
int x = XXX; // Becomes: int x = 4;
#undef XXX
int y = XXX; // ERROR: 'XXX' is not defined
```

You can also undefine from the command line with the `-U` flag.

2.1.2 `#ifdef`, `#if`, and Conditional Compilation

The preprocessor can be used to **conditionally compile** code. This means you can include or exclude blocks of code from the final program based on whether a macro is defined.

The simplest form is `#ifdef` (if defined):

```
// This is your C++ code (pre3.cpp)
#ifdef DEBUG
    std::cout << "Debug: x = " << x << std::endl;
#else
    // Do nothing in release mode
#endif
```

Real-world Analogy: This is like a "confidential" stamp. The preprocessor assistant is told, "If you see the 'DEBUG' stamp on the folder (`-DDEBUG`), then include this extra debug paragraph. Otherwise, just skip it."

You can see this in action by running the preprocessor (with the `-E` flag to stop after this stage) in two different ways:

```
# 1. Normal mode: The #ifdef block is false
$ c++ -E pre3.cpp

# 2. Debug mode: The #ifdef block is true
$ c++ -E -DDEBUG pre3.cpp
```

In the second case, the debug-printing code will be present in the `.ii` file. In the first, it will be gone, as if it never existed.

For more complex logic, you can use `#if`, `#elif` (else if), and logical operators. This is often used to handle differences between compilers or operating systems:

```
#if !defined(_GNUC_)
    std::cout << "A non-GNU compiler";
#elif __GNUC__ <= 2 && __GNUC_MINOR__ < 95
    std::cout << "gcc before 2.95";
#elif __GNUC__ == 2
    std::cout << "gcc after 2.95";
#elif __GNUC__ >= 3
    std::cout << "gcc version 3 or higher";
#endif
```

2.1.3 `#error`

This directive tells the preprocessor to stop compilation and print an error message. This is a powerful way to enforce requirements.

```
#if !defined(__GNUC__)
    #error This program requires the GNU compilers
#endif
```

If you try to compile this code with a non-GNU compiler, the build will fail immediately with your custom message.

2.1.4 `#include`

This is the most common directive. It tells the preprocessor to find the specified file and literally paste its entire contents into the current file at that exact location.

There are two forms, which tell the preprocessor *where* to look:

- `#include <iostream>`: The **angle brackets** `< >` tell the preprocessor to search for the file in the standard "system" include directories.
- `#include "myfile.h"`: The **double quotes** `" "` tell the preprocessor to search for the file *first* in the current directory (the same directory as the file containing the `#include`), and if it's not found there, to *then* search in the system directories.

If you organize your own header files into a subdirectory (e.g., an `include/` folder), you must tell the compiler to add that folder to its search path using the `-I` flag:

```
$ c++ -E -Iinclude my_program.cpp
```

This command adds the `include` directory to the list of places to search for headers.

2.1.5 Inspecting Preprocessor Output

To see the result of all this text substitution, you can run *only* the preprocessor step by using the `-E` flag:

```
$ c++ -E hello.cpp
```

The output will be the full, "expanded" C++ code that will be passed to the compiler. It's often thousands of lines long, as all the system headers (like `<iostream>`) are expanded.

2.2 Stage 2: The Compiler (Source to Assembly)

After the preprocessor generates the intermediate `.ii` file (which is just a massive C++ source file), it's passed to the *real* **compiler**. This stage is the heart of the translation. It parses the C++ syntax, checks for errors, and translates the high-level C++ code into low-level, but still human-readable, **assembly code** (a `.s` file).

You can stop the process after this stage using the `-S` flag:

```
$ c++ -S -O0 functioncall.cpp
```

This creates `functioncall.s`. The `-O0` flag is important: it means "no optimization." The resulting assembly code will be a very literal, step-by-step translation of your C++ code.

Now, try this:

```
$ c++ -S -O3 functioncall.cpp
```

The `-O3` flag tells the compiler to use its highest level of optimization. If you compare the two `.s` files, the `-O3` version will look radically different. The compiler will have rearranged, simplified, and perhaps even deleted parts of your code to make it run as fast as possible.

One common optimization is **inlining**. If you have a small function, the compiler might decide that it's faster to just copy-paste the function's assembly code directly into the spot where it's called, rather than performing the "overhead" of a formal function call (jumping to a new part of memory and back).

Assembly code can be hard to read, especially C++ "mangled" symbols (which look like `_Z4sqrtd`). You can use a tool like `c++filt` to "demangle" them, or use the fantastic online "Compiler Explorer" at https://godbolt.org/ to see this translation in real-time.

2.3 Stage 3: The Assembler (Assembly to Machine Code)

The **assembler** performs the final, mechanical translation. It takes the human-readable assembly file (`.s`) and converts it into a **relocatable object file** (`.o`). This file contains pure machine code—the binary 1s and 0s that the processor directly executes.

This `.o` file is a *binary file*, not a text (ASCII) file. If you try to open it in a text editor, you will see gibberish.

It is called “relocatable” because it’s not a complete program. It’s a single, compiled *component*. It may contain references to functions or variables defined in other files (like `std::cout` or a `square` function you wrote). It doesn’t know the final memory addresses for these things. It just has “placeholders” or “IOUs” that the linker will resolve later.

2.4 Stage 4: The Linker (Objects to Executable)

The **linker** is the final stage of the assembly line. Its job is to be the master project manager. It takes all the object files (`.o`) you’ve compiled, plus any **libraries** (like `libm.a` for math functions) that you’ve requested, and links them all together into one.

Real-world Analogy: If the compiler/assembler builds the individual "chapters" (`.o` files) of a book, the linker is the "bookbinder." It gathers all the chapters, finds the "Table of Contents" and "Index" (libraries), and then resolves all the cross-references. For example, when Chapter 1 (`main.o`) says "see function in Chapter 5 (`square.o`)", the linker finds the exact page (memory address) of that function and "links" the two, creating the final, complete, and executable book (`a.out`).

The final output is a single, **executable object file**. On Linux/macOS, this is named `a.out` by default; on Windows, it would be an `.exe` file. This is the program you can actually run.

2.5 Problem 2.1: Static & Dynamic Arrays

Here is a practical exercise to apply these concepts.

Task: Write a program which first reads in the number of values, n. Then, read in n values from standard input.

1. Normalize the loaded sequence so that the sum of all values is 1.
2. Print out the normalized sequence in *reverse order*.

Implement this in two ways:

1. Set a maximum number of input values, `max`, and allocate a **static array** of length `max`.
2. Do the same using **dynamic arrays** so that the input size is no longer limited. A good option to achieve this is to use `std::vector`.

Experimentation: Try compiling your program with different compiler options, such as `-Wall`, `-Wextra`, `-std=c++XX` (e.g., `-std=c++17`), and `-pedantic`. Check the compiler’s manual (`man c++`) to see what these flags do.

Hint: If you use `std::cin` to read, you can "pipe" input to your program from a file or another command. If your program is named `main`, you can test it with a file `input.txt` (containing "3 1 2 3") like this:

```
$ ./main < input.txt
```

Or you can generate the input on-the-fly:

```
# Uses a pipe
$ echo "3 1 2 3" | ./main

# Uses process substitution
$ ./main <<< "3 1 2 3"
```

3 Modular Programming: Segmenting Code

As programs grow, putting all your code into one giant file (e.g., `main_original.cpp`) becomes completely unmanageable. Any small change requires recompiling the entire massive file, and it's impossible to find anything.

The solution is **code refactoring**—the process of restructuring existing source code without changing its external behavior. We will segment our program by splitting the code into several files.

- `main_original.cpp` (The "Before" state)

```cpp
#include <iostream>
double square(double x) {
    return x*x;
}
int main() {
    std::cout << square(5.) << std::endl;
    return 0;
}
```

We will refactor this into three separate files. This requires understanding one of the most important concepts in C++: the difference between a **declaration** and a **definition**.

3.1 Declarations vs. Definitions

- **Definition:** This is the actual implementation of the function—the code in curly braces (`{...}`) that does the work. A function can only be **defined once** in an entire program. (This is the "One Definition Rule," or ODR).
- **Declaration (or Prototype):** This is just the function's "signature" or "contract." It tells the compiler what the function is named, what it returns, and what arguments it takes, but it has no body (it ends with a semicolon). A function can be **declared many times**.

Real-world Analogy: A **declaration** is a "menu item" at a restaurant. It tells you the name of the dish, what's in it, and what it costs. You can have many menus (many declarations). The **definition** is the "recipe in the kitchen" that actually explains *how* to make the dish. There can only be one official recipe (one definition).

Before you can “order” (call) a function, the compiler must have at least seen its “menu item” (declaration).

3.2 Using Header Files

The easiest way to manage declarations and share them between files is to use **header files** (conventionally ending in `.h` or `.hpp`).

Here is our refactored program:

File: `square.hpp` (The Header File / The "Contract")

```
// This is the DECLARATION
double square(double);
```

File: `square.cpp` (The Implementation / The "Kitchen")

```
#include "square.hpp" // Good practice to include its own header

// This is the DEFINITION
double square(double x) {
    return x*x;
}
```

File: `main.cpp` (The Client / The "Customer")

```
#include <iostream>
#include "square.hpp" // Includes the DECLARATION

int main() {
    // This call is legal because we included square.hpp
    std::cout << square(5.) << std::endl;
    return 0;
}
```

3.3 Compiling and Linking Separately

Now that our code is split, we can build it in pieces. We use the `-c` flag, which tells the compiler: "Compile this file, but do not link it. Just create the `.o` object file."

1. **Compile `square.cpp`:**

   ```
   $ c++ -c square.cpp
   ```

   This command reads `square.cpp`, finds the definition for `square`, and creates `square.o`.

2. **Compile `main.cpp`:**

   ```
   $ c++ -c main.cpp
   ```

This command reads `main.cpp`. It sees the *declaration* of `square` (from `square.hpp`) and the *call* to `square`. It trusts that the definition will be provided later and creates `main.o`.

3. **Link the object files:**

   ```
   $ c++ main.o square.o
   ```

   This is the final "bookbinding" step. The linker takes `main.o` and `square.o`. It sees that `main.o` needs the *definition* of `square` and finds it in `square.o`. It "links" them together into the final executable, `a.out`.

4. **(Better) Link and name the output:**

   ```
   $ c++ main.o square.o -o square
   ```

   The `-o` flag specifies the **output** file name, creating an executable named `square`.

3.4 Include Guards: Preventing Duplication

There's a critical problem with `#include`. What if a file accidentally includes the same header twice?

Consider this "diamond of death" scenario:

- `grandfather.h`: contains `struct foo { ... };`
- `father.h`: contains `#include "grandfather.h"`
- `child.cpp`: contains `#include "grandfather.h"` and `#include "father.h"`

When the preprocessor runs on `child.cpp`, it will first include `grandfather.h`. Then, it will include `father.h`, which *also* includes `grandfather.h`. The result is that the compiler sees the definition of `struct foo` twice, which is a "redeclaration error," and the build fails.

The solution is an **include guard**. This is a standard preprocessor trick:

File: `grandfather.h`

```
// 1. If GRANDFATHER_H is NOT defined:
#ifndef GRANDFATHER_H
// 2. Then define it:
#define GRANDFATHER_H

// 3. ...and process the entire file...
struct foo {
    int member;
};

// 4. End the conditional block
#endif /* GRANDFATHER_H */
```

How it works: The first time the file is included, `GRANDFATHER_H` is not defined. The preprocessor enters the block, defines `GRANDFATHER_H`, and processes the file. The *second* time it's included (in the same compilation), `GRANDFATHER_H` *is* defined, so the preprocessor skips the entire block from `#ifndef` to `#endif`.

A non-standard, but very common, alternative is `#pragma once`, which you can place at the top of the header file. It's simpler but may not be supported by all compilers.

```
#pragma once

struct foo {
    int member;
};
```

3.5 Assertions: `<cassert>`

The `<cassert>` header provides the `assert` macro, a powerful tool for sanity-checking your code. An **assertion** is a statement of a condition that you, the programmer, "assert" must be true at that point in the code.

```
#include <cassert> // Note: <cassert>, not <assert.h> in C++

double my_sqrt(double x) {
    // We assert this precondition must be true
    assert(x >= 0);
    // ... code to calculate square root ...
}
```

If the expression inside `assert()` evaluates to `false` at runtime, the program will immediately **abort** and print an error message stating which assertion failed, in which file, and on which line. This helps you find bugs instantly, rather than letting your program continue with "bad" data.

Assertions are safety checks for development. For a final, high-performance "production" build, you might want to turn them all off. The `<cassert>` header is designed for this. It internally looks something like this:

```
#ifdef NDEBUG
    #define assert(e) ((void)0) // Replaced with... nothing!
#else
    #define assert(e) /* ... complex implementation ... */
#endif
```

This means you can compile your code in two modes:

```
# 1. Debug mode: Assertions are ON
$ c++ assert.cpp

# 2. Release mode: Assertions are OFF
$ c++ -DNDEBUG assert.cpp
```

The `-DNDEBUG` flag (which stands for "No Debug") causes all `assert` statements to be preprocessed into nothing, incurring zero performance cost.

4 Libraries: Reusable Code Collections

A **library** is a collection of useful, pre-compiled functions (and data). Instead of us all writing our own `printf` or `sin` (sine) functions, we use them from standard libraries. Libraries come in two main "flavors".

4.1 Static vs. Shared Libraries

- **Static Libraries (`.a` or `.lib`):**

  These are also known as "archives" (the `.a` extension).

  Analogy: A static library is like a "toolbox." When you build your program (the "project"), the linker (the "builder") goes to the toolbox, takes *copies* of only the tools (functions) you need, and puts them directly into your project box (the executable).

  Result: Your final executable is larger, but it's completely **self-contained**. It doesn't need the library "toolbox" to exist on the user's computer.

- **Shared Libraries (`.so` or `.dll`):**

  These are "shared objects" (`.so`) or "dynamic-link libraries" (`.dll` on Windows).

  Analogy: A shared library is like a "central, public tool-shed" that everyone in the city uses. When you build your program, the linker does not copy the tools. Instead, it puts a *note* in your project box that says, "When you need a hammer, go to the tool-shed at 123 Main St. and use theirs."

  Result: Your executable is much smaller. However, it **depends** on that shared library file existing on the user's system when it runs. The benefit is that many programs can all "share" that one library in memory, saving resources.

4.2 Creating Your Own Libraries

Let's turn our `square` function into a library.

4.2.1 Making a Static Library (`libsquare.a`)

1. **Compile to an object file:**

   ```
   $ c++ -c square.cpp
   ```

   This gives us `square.o`.

2. **Archive the object file(s):**

   ```
   $ ar -crs libsquare.a square.o
   ```

The `ar` (archiver) tool packs `square.o` into the library file. The flags `-crs` mean "create" the archive, "run" quickly, and "save" the index.

By convention, static libraries are named `libsomething.a`. We'll see why in a moment.

4.2.2 Making a Shared Library (`libsquare.so`)

1. **Compile to a Position Independent Code (PIC) object file:**

   ```
   $ c++ -fPIC -c square.cpp
   ```

   The `-fPIC` flag is crucial. It generates "position-independent code," which means the code can be loaded into *any* memory address, which is essential for a shared library (since the operating system decides where to load it at runtime).

2. **Link into a shared object:**

   ```
   $ c++ -shared -fPIC -o libsquare.so square.o
   ```

   The `-shared` flag tells the compiler to create a shared library instead of an executable.

By convention, shared libraries are named `libsomething.so`.

4.3 Using Your Libraries

Now, let's use our new library to build `main.cpp`. Assume we've put `square.hpp` in a folder named `include/` and our library `libsquare.a` (or `.so`) in a folder named `lib/`.

1. **Compile `main.cpp`:**

   ```
   $ c++ -c -Iinclude main.cpp
   ```

   This creates `main.o`. The `-Iinclude` flag tells the compiler to "look in the `include` directory for headers" (so it can find `square.hpp`).

2. **Link `main.o` against our library:**

   ```
   $ c++ -o square main.o -Llib -lsquare
   ```

   This is the key step. We use two new flags to tell the linker:

   - `-Llib`: "Look in the `lib` directory for libraries."
   - `-lsquare`: "Find and link the library named `square`." The linker automatically adds the `lib` prefix and `.a` or `.so` suffix, searching for `libsquare.a` or `libsquare.so`.

Note: The order of libraries matters! If library `libA` uses functions from `libB`, you must link them in the order: `-lA -lB`.

4.4 Documenting Your Library

When you create a library, you are creating a "contract" with its users. Good documentation is essential and should include:

- **Synopsis:** The function declarations (i.e., the header file contents).
- **Semantics:** A clear explanation of what the function does.
- **Preconditions:** What must be true *before* a user calls the function? (e.g., "pointer must not be null," "value must be positive").
- **Postconditions:** What do you guarantee will be true *after* the function returns (assuming the preconditions were met)?
- **Dependencies:** What other libraries or components does this library need?
- **Exception guarantees:** How does the function behave if an error occurs? (e.g., "no-throw," "basic," or "strong" guarantee).

For example, our `square` function:

- **Synopsis:** `double square(double x);`
- **Semantics:** Calculates the square of `x`.
- **Preconditions:** `std::abs(x) <= std::sqrt(std::numeric_limits<double>::max())` (i.e., the square will not overflow a `double`).
- **Postconditions:** The return value is `x*x`.
- **Dependencies:** None.
- **Exception guarantees:** No-throw. Will not throw an exception.

4.5 Problem 2.2: Simpson Integration Library

This is a more advanced exercise to tie all these concepts together.

Task:

1. Take the Simpson integration algorithm (from a previous exercise) and wrap it in a function. This function should take a **function pointer** to the integrand, the integration interval, and the number of bins. Use `assert` to check the validity of input parameters (e.g., number of bins must be positive and even).
2. **Refactor** your Simpson integration function into a header and source file (`integrate.h`, `integrate.cpp`).

3. **Document** your Simpson integration function in the header. What are its preconditions and postconditions?
4. **Write a `Makefile`** (which we will learn about next) that compiles the function for you. Make sure it only recompiles the files that have changed.
5. **Compile a static library**, `libintegrate.a`, that contains your Simpson integration function. Rewrite your `Makefile` to link your main program against this library.

Hint: You will need to learn about the syntax for "pointers to functions."

5 Build Automation with `make`

As you saw in the last exercise, the build process for even a simple program can become tedious:

```
$ c++ -c a.cpp
$ c++ -c b.cpp
$ c++ -c c.cpp
...
$ c++ -Iinclude -o my_program main.o a.o b.o c.o -Llib -lA -lB
```

This is tedious and error-prone. What happens if you change a header file, `a.h`? You have to remember to recompile `a.cpp` and `main.cpp` (if it includes `a.h`), and then re-link. It's too much to track by hand.

The solution is a **build automation tool**. We will focus on one of the oldest and most common: **Make**.

5.1 What is `make`?

Make is a build automation tool created by Stuart Feldman at Bell Labs in the 1970s. Its primary job is to build **targets** (like executables, libraries, or even PDF reports) by following **rules** and managing **dependencies**.

Its most important feature is that **it keeps track of updates!**

Real-world Analogy: `make` is a "smart chef." You give it a "recipe book" (a file named `Makefile`) and tell it, "I want to `make` the 'square' executable."

1. The chef looks at the rule for 'square'. It says: "To make `square`, you need `main.o` and `square.o`." (These are the **prerequisites** or **dependencies**).
2. It then checks: "Is `main.o` up-to-date?" It does this by comparing file modification times: "Is `main.cpp` (its prerequisite) *newer* than `main.o`?"
3. If `main.cpp` is newer, the chef knows `main.o` is "stale" and must be "re-cooked." It runs the **command** to re-compile it (e.g., `c++ -c main.cpp`).
4. It does the same check for `square.o`.
5. Once all prerequisites are up-to-date, it runs the final **command** to build the 'square' target (the linking step).

If you run `make` again without changing any files, the chef sees that all targets are newer than their prerequisites and does nothing.

We will use **GNU Make**, the standard on Linux and macOS. Be aware:

- Make is its own "little language," and its syntax is cryptic.
- It has no real debugger.
- It requires a good understanding of the command line shell.

5.2 Basic `Makefile` Syntax

`make` looks for a build file named `Makefile`, `makefile`, or `GNUmakefile` in your directory. This file contains the "recipes," which are called **rules**.

The basic syntax for a rule is:

```
target: prerequisites
[TAB]commands
```

- **target:** The file we want to build (e.g., `square.o`).
- **prerequisites:** The files needed to build the target (e.g., `square.cpp`).
- **commands:** The shell commands to execute to build the target.

CRITICAL WARNING: The command line *must* start with a literal **Tab** character, not spaces. This is the most common and frustrating error for new `make` users.

Comments in a `Makefile` start with `#`.

Here is a minimal example:

```
# hello.mk
# A simple example
hello:
    echo Hello students
```

This file defines one target, `hello`, which has no prerequisites and one command. To run it, we use the `-f` flag to specify the filename:

```
$ make -f hello.mk
echo Hello students
Hello students
```

You can also do a "dry run" with `-n` to see what commands `make` *would* run, without actually running them. This is very useful for debugging.

```
$ make -f hello.mk -n
echo Hello students
```

5.3 A C++ `Makefile` Example

Let's write a `Makefile` for our `square` program. We want to automate these manual commands:

```
$ c++ -c square.cpp
$ c++ -c main.cpp
$ c++ -o square main.o square.o
```

Here is our first attempt at a `Makefile` (`simple_try-01.mk`):

```
# simple_try-01.mk
square.o: square.cpp square.hpp
    c++ -c square.cpp
```

```
main.o: main.cpp square.hpp
    c++ -c main.cpp

square: main.o square.o
    c++ -o square main.o square.o
```

Note that we added `square.hpp` as a prerequisite for both object files. This is correct! If the header file changes, we *want* both `.cpp` files to be recompiled.

Now, let's try to run it:

```
$ make -f simple_try-01.mk
c++ -c square.cpp
```

It only built `square.o`! Why? **By default, `make` only builds the *first* target in the file**.

We can fix this by explicitly telling `make` which target to build:

```
$ make -f simple_try-01.mk square
```

This works, but it's not ideal.

5.4 Phony Targets: `all` and `clean`

A better solution is to add a new *first target* that depends on the final program. By convention, this target is called `all`.

```
# simple.mk
.PHONY: all
all: square

square.o: square.cpp square.hpp
    c++ -c square.cpp
...
```

The `all` target is now first, so it's the default. Its prerequisite is `square`. So, `make` says, "To build `all`, I must first build `square`." It then finds the `square` rule and proceeds as we want.

The `.PHONY: all` line is important. It tells `make` that `all` is a "phony" target; it's not a real file. This prevents `make` from getting confused if you ever create a file named `all`.

Another common phony target is `clean`, used to remove all generated files.

```
.PHONY: clean
clean:
    rm -f *.o square
```

We make it `.PHONY` because we *always* want to run this command, even if a file named `clean` happens to exist.

6 Improving the `Makefile` (The DRY Principle)

Our `Makefile` works, but it's not good. It's "WET" (Write Every Time). We want it to be "DRY" (Don't Repeat Yourself).

Our `Makefile` has several problems:

- **Repetition:** We write `square.cpp` twice, `main.o` twice, etc.
- **Hardcoded Compiler:** What if we want to use `clang++` instead of `c++`? We'd have to edit 3 lines.
- **Hardcoded Flags:** What if we need to add compiler flags like `-Wall`?
- **No Cleanup:** We haven't added a `clean` target.

6.1 Automatic Variables

`make` has special "automatic variables" that are set for each rule. They are powerful shortcuts.

- `$@`: The file name of the target of the rule. ("The dish I'm making.")
- `$<`: The name of the *first* prerequisite. ("The first ingredient.")
- `$^`: The names of *all* prerequisites. ("All the ingredients.")

Let's update our `Makefile` to use them:

```
# simple.mk
.PHONY: all
all: square

square.o: square.cpp square.hpp
    c++ -c $<   # $< is square.cpp

main.o: main.cpp square.hpp
    c++ -c $<   # $< is main.cpp

square: main.o square.o
    c++ -o $@ $^   # $@ is square, $^ is main.o square.o
```

This is much better! The commands are now generic.

6.2 Predefined and Custom Variables

`make` also has many predefined variables for common tools. We should use them.

- `CXX`: The C++ compiler (default `g++`).
- `CXXFLAGS`: Extra flags for the C++ compiler.

- `RM`: The remove command (default `rm -f`).
- `LDFLAGS`: Extra flags to give to the linker (e.g., `-Llib`).
- `LDLIBS`: Library flags for the linker (e.g., `-lsquare`).

We can set these variables ourselves at the top of the `Makefile`, and refer to them using `$VAR` or `$(VAR)`.

Let's create our final, robust `Makefile`:

```
# Set our variables
CXX = c++
CXXFLAGS = -std=c++11
CXXFLAGS += -Wall -Wextra -Wpedantic
CXXFLAGS += -O3

.PHONY: all
all: square

square.o: square.cpp square.hpp
    ${CXX} ${CXXFLAGS} -c $<

main.o: main.cpp square.hpp
    ${CXX} ${CXXFLAGS} -c $<

square: main.o square.o
    ${CXX} ${CXXFLAGS} -o $@ $^ ${LDFLAGS} ${LDLIBS}

.PHONY: clean
clean:
    ${RM} *.o square
```

Notice we use `+=` to *append* to the `CXXFLAGS` variable. This `Makefile` is clean, easy to read, and easy to modify. If we want to change the compiler, we only edit one line.

6.3 Customizing Builds with `include`

What if your collaborators need different compilers or flags (e.g., one uses `g++-9` and another uses `clang++`)? If you all edit the `Makefile`, you'll have conflicts in version control.

A better way is to use `include`.

```
# Try to include a local config file
-include config.mk

# Set defaults if not provided
CXX ?= c++
```

```
CXXFLAGS ?= -std=c++11 -O3 -Wall

.PHONY: all
all: square
...
```

This is a more advanced but common pattern.

- `-include config.mk`: The dash – tells `make` "it's okay if this file doesn't exist."
- `CXX ?= c++`: The `?=` operator means "set this variable *only if* it hasn't already been set."

This allows each user to create their own `config.mk` file with their personal settings (e.g., `CXX = clang++`) and keep it out of version control. If `config.mk` exists, it sets the variables. If not, the `?=` defaults are used.

You should provide a `config.mk.example` file in version control for new users to copy.

7 Advanced `make` and Further Reading

This is just the beginning. More advanced `make` topics include:

- **Pattern Rules:** Write a single rule to build *all* `.o` files from `.cpp` files.
- **Automatic Dependency Generation:** A way to have `make` automatically detect which headers a `.cpp` file includes, so you don't have to list them by hand.

Finally, `make` is not just for building software! It's a general-purpose dependency-management tool. You can use it to "build" a scientific paper.

Analogy for Reproducibility: Imagine a `Makefile` for a paper.

- The `paper.pdf` (target) depends on `results.tex` and `plot.png`.
- `plot.png` (target) depends on `data.csv` and `plot.py`.
- `data.csv` (target) depends on `simulation.cpp`.

If you change your simulation code (`simulation.cpp`), typing `make` will automatically re-run the simulation, re-generate the data, re-draw the plot, and re-compile the PDF. This traces your entire workflow and is **excellent for reproducibility**.

For more information, see the official GNU Make documentation, or use the `man make` and `info make` commands.