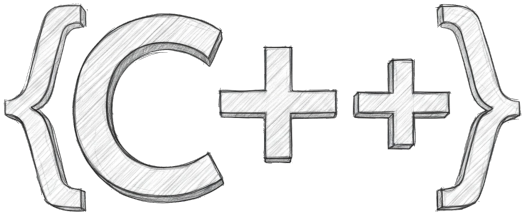
# The Build Process

## Introduction to Modern C++

Ryan Baker




## Contents

**1 The Build Process** **2**
1.1 The Preprocessor . . . 2
1.1.1 Text Replacement `#define` . . . 2
1.1.2 File Inclusion `#include` . . . 6
1.1.3 Header `.h` Files . . . 9
1.1.4 Conditional Compilation `#if`, `#ifdef` . . . 10
1.1.5 Translation Units . . . 11
1.2 The Compiler . . . 13
1.2.1 The Compilation Process . . . 13
1.2.2 Compiler Output . . . 17
1.2.3 Compiler Flags . . . 19
1.3 The Linker . . . 21

# 1 The Build Process

Recall that C++ is a compiled language. This means its source code must be translated into machine instructions before execution. This translation occurs through a sequence of steps collectively known as the **build process**, which consists of three major stages: **preprocessing**, **compilation**, and **linking**.

$$\texttt{.cpp} \rightarrow \boxed{\text{Preprocessor}} \rightarrow \boxed{\text{Compiler}} \rightarrow \boxed{\text{Linker}} \rightarrow \texttt{exe}$$

Each stage transforms the code in a distinct way, producing intermediate outputs that can be examined to gain insights into the compilation process. In this section, we will explore how these stages work, what they produce, and how we can analyze them to better understand our code.

## 1.1 The Preprocessor

The first stage of the build process is preprocessing. The preprocessor's job is to edit the source code text according to a set of **preprocessor directives**. Preprocessor directives begin with the `#` symbol and are processed line-by-line before compilation begins. A comprehensive list of preprocessor directives and their definitions can be found in on page 12.

### 1.1.1 Text Replacement `#define`

The `#define` directive is used to define macros. A **macro** is a symbol that the preprocessor replaces with a value, expression, or code snippet. `#define` instructs the preprocessor to replace every occurrence of a defined symbol with its corresponding replacement string. This, for example, allows you to define preprocessor constants:

```cpp
#include <iostream>

#define PI 3.14159

int main()
{
    std::cout << PI << std::endl; // prints 3.14159
}
```

In the code above, the preprocessor replaces every instance of `PI` with `3.14159`. However, it's crucial to remember that the preprocessor does not understand C++ semantics–it simply performs text substitution. This lack of type checking or validation can lead to unexpected behavior if macros are not used carefully.

Additionally, `#define` can be used to define macros without a replacement string. In this case, the macro is simply a placeholder for a condition or symbol. This is commonly used for conditional compilation (§ 1.1.4) to enable or disable certain parts of the code depending on whether the macro is defined:

```
#define DEBUG
```

The `#define` directive can also be used to create function-like macros that accept arguments:

```
#include <iostream>

#define SQUARE(x) x * x
#define MAX(a, b) a > b ? a : b

int main()
{
    std::cout << SQUARE(5) << std::endl;
    std::cout << MAX(2, 3) << std::endl;
}
```

In the example above, the preprocessor replaces `SQUARE(5)` with `5 * 5` and `MAX(2, 3)` with `2 > 3 ? 2 : 3`. However, the potential for unexpected behavior increases significantly with function-like macros. This is because the preprocessor simply performs text substitution, without understanding operator precedence or modification sequencing. Consider the following example:

*square.cpp*

```
#include <iostream>

#define SQUARE(x) x * x
```

```
int main()
{
    std::cout << SQUARE(3 + 3) << std::endl;
}
```

One would expect this code to yield $6 \times 6 = 36$, however, when we run it:

Terminal

```
$ clang++ -std=c++23 square.cpp
$ ./a.out
15
```

We get 15 instead. This happens because the preprocessor expanded `SQUARE(3 + 3)` into `3 + 3 * 3 + 3`, which evaluates to 15 due to the order of operations. To prevent this type of error, we can wrap every argument in parentheses `()`:

```
#define SQUARE(x) ((x) * (x))
```

By doing so, we ensure that the expression inside the macro is evaluated before applying the multiplication. With this modification, `SQUARE(3 + 3)` will correctly evaluate to 36. However, we're not entirely safe yet. Consider the following example:

*square2.cpp*

```
#include <iostream>

#define SQUARE(x) ((x) * (x))

int main()
{
    int i { 5 };
    std::cout << SQUARE(++i) << std::endl;
}
```

Again, one might expect this to return $6 \times 6 = 36$ but when we run it:

Terminal

```
$ clang++ -std=c++23 square2.cpp
$ ./a.out
42
```

We get 42 instead. This happens because when the preprocessor expands `SQUARE(++i)`, it turns it into `((++i)*(++i))`. This results in `i` being incremented twice–first to 6, then to 7–leading to the multiplication of $6 \times 7$ (or $7 \times 6$). Since the macro is evaluated twice with the side-effect of modifying the variable, the result is not as expected.

> Unfortunately, there is no simple way to combat this problem, other than forgoing function-like macros altogether. For this reason, it is highly recommended to leave work up to the compiler whenever possible. As a rule of thumb: **the compiler is smart, but the preprocessor is stupid**. The compiler can optimize your code, catch potential bugs, and provide helpful warnings, but the preprocessor simply performs text substitution without any understanding of the code's logic or semantics. Hence, given the choice between a function-like macro and a function, you should almost always prefer a function–especially with modern C++ features such as lambdas and compile-time functions, which offer better type safety, readability, and performance.

Now that we've mercilessly condemned preprocessor macros, let's talk about how to rid them from our codebase entirely. The `#undef` directive can be used to undefine a previously defined macro:

```
1 #define PI 3.14159
2 double area1 { PI * 1 * 1 };
3
4 #undef PI
5 double area2 { PI * 2 * 2 }; // error! PI is undefined
```

The reason for doing this is quite simple: we want to prevent naming conflicts. By undefining macros that are no longer needed, we avoid the risk of accidental redefinitions or clashes with other parts of the code.

### 1.1.2 File Inclusion `#include`

The `#include` directive is used to include the contents of one file in another. It can be thought of as the "copy-and-paste" directive, where the preprocessor searches for the specified file and pastes its contents into the source code. There are two variations of `#include`: angle brackets `<>` and double quotes `""`:

```
#include <iostream> // brackets
#include "myfile.h" // quotes
```

The difference lies in the search path for the file. With angle brackets, the preprocessor searches the **include path**, which is a directory (or list of directories) containing various standard librares and headers. With double quotes, it first looks in the current directory and, if not found, checks the include path.

> While you could technically use quotes for all includes, it's best practice to reserve angle brackets for standard library headers (like `iostream`) and use double quotes for project-specific or local files.

The `#include` directive essentially copies and pastes the contents of the included file in place of the directive:

*x.cpp*

```
int x { 500 };
```

*main.cpp*

```
#include <iostream>

#include "x.cpp" // copy and paste x.cpp's contents here

int main()
{
    std::cout << x << std::endl;
}
```

The two files above are functionally equivalent to:

*main.cpp*

```cpp
#include <iostream>

int x { 500 };

int main()
{
    std::cout << x << std::endl;
}
```

To further demonstrate the simple and literal nature of `#include`, consider this more ridiculous, unholy, and outright blasphemous example:

*int.cpp*

```cpp
int
```

*mainfunc.cpp*

```cpp
main()
```

*openbrace.cpp*

```cpp
{
```

*message.cpp*

```cpp
std::cout << "Hello, World!" << std::endl;
```

*closebrace.cpp*

```cpp
}
```

*helloworld.cpp*

```cpp
#include <iostream>

#include "int.cpp"
#include "mainfunc.cpp"
#include "openbrace.cpp"
#include "message.cpp"
#include "closebrace.cpp"
```

**Terminal**

```
$ clang++ -std=c++23 helloworld.cpp && ./a.out
Hello, World!
```

After including the other five files into `helloworld.cpp`, we are left with:

```cpp
#include <iostream>

int
main()
{
std::cout << "Hello, World!" << std::endl;
}
```

When compiled and run, this produces the expected output. This example highlights how `#include` simply pastes the contents of the files, completely disregarding logic or structure.

It should go without saying that your projects should never resemble the example above. Any developer writing production code in this manner can, should, and will be fired–if not prematurely conscripted. A proper approach to modular file structure is covered in § 1.1.3 Header `.h` Files.

### 1.1.3 Header `.h` Files

Header files are commonly used in C++ to organize code, especially in larger projects. Their primary role is to **declare the interface** of a module or library, allowing for separation between the declaration of functionality and its implementation. However, there are no strict rules in C++ about what should or shouldn't go into a header file–it's mostly up to the developer or the project's conventions. Here's an example of a simple header file, `mylib.h`:

*mylib.h*

```
1 void greet(); // declare the function greet()
```

In this example, `mylib.h` declares the `greet()` function. The **definition** (or **implementation**) of this function would typically be placed in the corresponding source file, `mylib.cpp`:

*mylib.cpp*

```
1 // include mylib.h to inherit the declaration of greet()
2 #include "mylib.h"
3
4 void greet()
5 {
6     std::cout << "Hello from MyLib!" << std::endl;
7 }
```

By separating declarations (in header files, `.h`) and definitions (in source files, `.cpp`), we improve code readability, reusability, and modularity. This structure allows different parts of the program to focus on their specific responsibilities, making the codebase easier to maintain and understand.

A useful convention is to name header files after the module or class they describe. Additionally, some C++ developers prefer using the `.hpp` extension instead of `.h` to indicate that the header contains C++-specific code rather than C code. However, this is purely a matter of convention and personal or project-specific preference.

### 1.1.4 Conditional Compilation `#if`, `#ifdef`

The preprocessor provides several mechanisms for conditionally including or excluding code. One of the most common directives is `#if`, which evaluates a condition and includes the subsequent block only if the condition is true:

```
#define DEBUG 1

#if DEBUG
    // debugging logic...
#endif
```

You can also use `#else` and `#elif` (else-if) to create branching logic within your preprocessor directives:

```
#define CONTROL 2
#if (CONTROL == 1)
    // ...
#elif (CONTROL == 2)
    // ...
#elif (CONTROL == 3)
    // ...
#else
    // ...
#endif
```

Note that any preprocessor conditional block must be terminated with the `#endif` directive. The `#ifdef` and `#ifndef` directives provide further control by allowing you to include a block if a macro is defined (`#ifdef`) or not defined (`#ifndef`):

```
#define DEBUG

#ifdef DEBUG
    // debugging logic...
#endif
```

A very common use case for conditional compilation is **include guards**. An include guard prevents the same file from being included multiple times within a translation unit, which could cause redefinition errors. For example, an include guard for `mylib.h` might look like this:

*mylib.h*

```
#ifndef MYLIB_H_INCLUDED
#define MYLIB_H_INCLUDED

void greet();

#endif // MYLIB_H_INCLUDED
```

When `#include "mylib.h"` is encountered for the first time, the preprocessor macro `MYLIB_H_INCLUDED` has not been defined, so the code within `mylib.h` is included. If the file is included again, the symbol `MYLIB_H_INCLUDED` has already been defined, so the contents of the file are skipped, thus avoiding redefinition errors.

### 1.1.5 Translation Units

The output of the preprocessing stage is known as a **translation unit**. A translation unit is the preprocessed result of a single source file, with all preprocessor directives recursively expanded. This means that a translation unit includes any files that have been included via `#include`. As a result, the preprocessor's output is still valid C++ code. We can inspect the preprocessor's output by using the `-E` flag with `Clang` or `g++`:

*define.cpp*

```
#include <iostream>

#define X 350

int main() {
    std::cout << X << std::endl;
}
```

Terminal

```
$ clang++ -std=c++23 -E define.cpp > translation_unit.cpp
$ clang++ -std=c++23 translation_unit.cpp && ./a.out
350
```

In this example, the `-E` flag tells `Clang` to stop after the preprocessing step, at which point we redirect the output into `translation_unit.cpp`. We then compile `translation_unit.cpp` and run it like any other C++ file. The result is the same as if we had compiled `define.cpp`.

| Directive | Description | Section |
|---|---|---|
| `#define` | Defines a macro, which can be an object-like macro or a function-like macro. | § 1.1.1 |
| `#undef` | Undefines a previously defined macro. | § 1.1.1 |
| `#include` | Includes the contents of a specified file. | § 1.1.2 |
| `#if` | Begins a conditional block; code within is compiled only if the condition is true. | § 1.1.4 |
| `#elif` | Specifies an alternate condition within a `#if` block. | § 1.1.4 |
| `#else` | Specifies the code to be compiled if none of the preceeding `#if` or `#elif` checks are true. | § 1.1.4 |
| `#ifdef` | Checks if a macro is defined; code within is compiled only if the macro is defined. | § 1.1.4 |
| `#ifndef` | Checks if a macro is not defined; code within is compiled only if the macro is not defined. | § 1.1.4 |
| `#endif` | Ends a conditional compilation block. | § 1.1.4 |
| `#error` | Generates a compilation error with a message. | |
| `#warning` | Generates a warning with a message. | |
| `#line` | Changes the compiler's idea of the current line. | |
| `#pragma` | Provides additional, implementation-defined instructions to the compiler. | |

Table 1: Overview of Preprocessor Directives and Their Functions

## 1.2 The Compiler

Compilation is the second stage of the build process. During this phase, the compiler takes the preprocessed output (translation unit) and converts it into an **object file**. This section offers a brief overview of the compilation process, provides tools for inspecting the compiler's output, and includes a directory of common compiler flags on page 19.

### 1.2.1 The Compilation Process

What follows is a brief overview of the compilation process. This is by no means an exhaustive resource for learning about compilers. Compiler design is both an art and a science in and of itself, and this section is intended only to provide a high-level understanding to support our exploration of C++. Nonetheless, the six major steps in the compilation process are as follows:

1. **Lexical Analysis: Tokenization**

   The first step in the compilation process is breaking the source code into **tokens**–the smallest meaningful units. Tokens can include keywords (`int`, `for`, `return`), identifiers (`x`, `foo`, `main`), operators (`+`, `=`, `&&`), literals (`5`, `"Hello"`), or symbols (`{` , `;`, `]`). For example, the code:

   ```
   1 int y = x + 5;
   ```

   might be tokenized as:

   ```
   1 [int] [y] [=] [x] [+] [5] [;]
   ```

   At this stage, the compiler detects tokenization errors, such as the inclusion of an extraneous symbol:

   ```
   1 // error, extraneous symbol: '@'
   2 int @ main()
   3 {
   4 }
   ```

2. **Syntax Analysis: Parsing**

   After tokenization, the next step in the compilation process is **parsing**, where the compiler checks if the source code follows the proper grammar or syntax rules of the language. For example, this code is valid:

```
int x { 42 };
```

   But this code is invalid:

```
42 {} x  ;int
```

   Although both snippets have the same tokens, the second example violates the syntactic structure of C++. The parser catches errors like these by analyzing the arrangement of tokens. Its output is a **parse tree**, which reflects the syntactic structure of the program. This step ensures the code follows the rules for valid expressions and statements.

3. **Semantic Analysis**

   Next, the compiler checks the code for logical correctness by analyzing the parse tree. This step ensures that the code makes sense in terms of variable types, function usage, and scope. For example, this code passes the parsing phase but fails at semantic analysis:

```
int x { "forty-two" };
```

   The code is syntactically valid, but semantically incorrect because an integer variable is being assigned a string value.

4. **Intermediate Code Generation**

   At this stage, the compiler has confirmed the correctness of the code. It now translates the code into an intermediate representation (IR), which is independent of the target machine. This step makes the code easier to optimize and facilitates porting the program to different hardware architectures.

5. **Code Optimization**

   The compiler then attempts to make the final executable smaller, faster, or otherwise more efficient without altering its intended functionality. Modern compilers are highly sophisticated in the optimizations they can perform. Some common optimization strategies include:

   - **Constant Folding**: Replacing constant expressions at compile-time:

   ```
   1 int weeks { 365 / 7 };
   ```

   This code can be optimized into:

   ```
   1 int weeks { 52 };
   ```

   Constant folding eliminates the need for a run-time calculation.

   - **Loop Hoisting**: Removing repeated computations from a loop body:

   ```
   1 for (int i = 0; i < 1000; ++i)
   2 {
   3     int k { calculation() };
   4     // ...
   5 }
   ```

   This code can be optimized into:

   ```
   1 int k { calculation() };
   2 for (int i = 0; i < 1000; ++i)
   3 {
   4     // ...
   5 }
   ```

   Since the value of `k` does not change with each loop iteration, the compiler moves the calculation outside the loop to avoid redundant computation, while ensuring that `k` maintains its intended scope.

- **Dead Code Elimination**: Removing code that cannot execute:

```
void foo()
{
    // ...
    return;
    // this code is 'dead' ...
}
```

  In this example, the compiler can eliminate the unreachable code after the `return` statement, as it will never be executed.

- **Common Expression Removal**: Eliding redundant computations:

```
int a { x * y }, b { x * y * z };
```

  The above code can be optimized into:

```
int a { x * y }, b { a * z };
```

  By calculating `a` once and reusing it in the second expression, the compiler avoids recomputing `x * y` twice.

> In general, you should avoid sacrificing code readability for micro-performance improvements. The compiler is much better at performing optimizations than you are. For example, I've seen code like this:

```
int log10(int n) { // find floor(log10(n))
    int log { -1 };
    for (int i = 1; i <= n; ++log) {
        i = (i << 3) + (i << 1); // i *= 10
    }
    return log;
}
```

In this case, the programmer tried to optimize the expression `i * 10` by replacing it with `(i << 3)+(i << 1)`. The first issue here is that there's no guarantee the bitshift and addition will actually be faster than multiplication–it might even be slower depending on the processor. The second issue is that any competent compiler will likely perform this optimization automatically, and without hurting everyone's eyes in the process.

I'm not suggesting that you shouldn't optimize your code, but rather that you should avoid micro-optimizations that hurt readability. The compiler will likely handle those for you. On the other hand, algorithmic optimizations are always worth your attention because compilers generally can't identify those on their own.

6. **Code Generation**

   In this final phase of compilation, the compiler translates the optimized intermediate representation (IR) into machine code that is specific to the target processor. This machine code is what the CPU can directly execute.

### 1.2.2 Compiler Output

The compiler produces an **object file** as its output, typically with a `.o` or `.obj` extension on Unix-like systems. This object file contains machine code that the processor can understand, but it is not yet a complete executable program. To produce an object file, we can use the `-c` flag with `Clang` and `g++`, instructing the compiler to compile the source file without linking:

*main.cpp*

```
1 int x { 401 };
2
3 int multiply(int a, int b) { return a * b; }
4
5 int main() {
6     multiply(x, 10);
7 }
```

Terminal

```
$ clang++ -std=c++23 -c main.cpp
```

This command produces `main.o`, the object file corresponding to `main.cpp`. Since object files are not human-readable, we need special tools to inspect their contents. For example, the `objdump` command (or similar tools) allows us to analyze an object file:

Terminal

```
$ objdump -tC main.o
code/main.o:    file format mach-o arm64

SYMBOL TABLE:
0000000000000000 l     F __TEXT,__text ltmp0
0000000000000044 l     O __DATA,__data ltmp1
0000000000000048 l     O __LD,__compact_unwind ltmp2
0000000000000000 g     F __TEXT,__text multiply(int, int)
0000000000000020 g     F __TEXT,__text _main
0000000000000044 g     O __DATA,__data _x
```

Here, the `-t` flag tells `objdump` to display the **symbol table**, which contains information about symbols (functions, variables, etc.) in the object file. The `-C` flag ensures that C++ symbols are demangled, revealing their original names instead of the compiler-mangled versions. From the output, we can recognize elements from our original C++ code:

- The function `multiply(int, int)`.
- The function `main()`.
- The variable `x`.

Understanding the symbol table gives insight into how the compiler translates C++ code into machine instructions before linking it into an executable. Instead of an object file, we can also generate **assembly code** using the `-S` flag with `Clang` and `g++`:

Terminal

```
$ clang++ -std=c++23 -S main.cpp
```

This command produces `main.s`, an assembly file corresponding to `main.cpp`. This file provides a closer look at the low-level instructions generated by the compiler, helping us understand how C++ constructs translate into assembly.

### 1.2.3 Compiler Flags

This section provides a directory of commonly used C++ compiler flags for `Clang` and `g++`, along with their descriptions. To view the full list of available flags for your compiler, you can use `clang++ -help` or `g++ --help`. These commands will display a comprehensive list of supported compiler options, including optimizations, warnings, debugging features, and more.

| `Clang` | `g++` | **Description** |
|---|---|---|
| `-o` | `-o` | Specify the output file. |
| `-E` | `-E` | Stop after preprocessing, do not compile. Output is a translation unit. |
| `-S` | `-S` | Stop after compilation, do not assemble. Output is an assembly file. |
| `-c` | `-c` | Stop after assembling, do not link. Output is an object file. |
| **Warning Flags** | | |
| `-Wall` | `-Wall` | Enables most common warnings that help catch potential issues. |
| `-Werror` | `-Werror` | Treats warnings as errors. |
| `-Wextra` | `-Wextra` | Enables additional warnings not covered by `-Wall`. |
| `-pedantic` | `-pedantic` | Enforces strict C++ compliance, warning about language extensions. |
| `-w` | `-w` | Suppress all warnings. |

| Clang | g++ | Description |
|---|---|---|
| **Optimization Flags** | | |
| `-O0` | `-O0` | No optimization, default setting for debugging. |
| `-O1` | `-O1` | Enables basic optimizations that don't significantly increase compile time. |
| `-O2` | `-O2` | More aggresive optimization. |
| `-O3` | `-O3` | Maximum optimizations, potentially increasing code size and compile time. |
| `-Os` | `-Os` | Optimizes for a reduced binary size. |
| `-ffast-math` | `-ffast-math` | Allow aggressive, potentially lossy floating-point optimizations. |
| **Additional Flags** | | |
| `-std=c++`*X* | `-std=c++`*X* | Use the specified C++ standard (e.g., `-std=c++23`). |
| `-v` | `-v` | Display verbose compilation details, including toolchain paths and options. |
| `-x` *lang* | `-x` *lang* | Explicitly specify the language for the following input files. |
| `-g` | `-g` | Include source-level debugging information in the binary. |
| `-I` *dir* | `-I` *dir* | Add a directory to the include path for header file lookup. |
| `-L` *dir* | `-L` *dir* | Add a directory to the search list for library linking. |
| `-l` *lib* | `-l` *lib* | Link with the specified library (e.g., `-lm` for the math library). |
| `-D` *macro* | `-D` *macro* | Define a macro, equivalent to `#define macro`. |
| `-U` *macro* | `-U` *macro* | Undefine a macro. |

Table 2: Common Compiler Flags for `Clang` and `g++`

## 1.3 The Linker

Linking is the final stage of the build process. The linker is responsible for combining multiple object files into a single executable program. It resolves **external references**, which are references that reach across translation units (i.e., files). To illustrate how the linker works, lets examine a project comprising three files:

*foo.h*

```
#ifndef FOO_H_INCLUDED
#define FOO_H_INCLUDED

void foo();

#endif // FOO_H_INCLUDED
```

*foo.cpp*

```
#include <iostream>
#include "foo.h"

void foo()
{
    std::cout << "fooing..." << std::endl;
}
```

*proj.cpp*

```
#include "foo.h"

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

In the example above, `foo.h` contains the declaration of `foo()`. This header file is meant to be included in source files to make the `foo()` function available for use. `foo.cpp` contains the implementation of `foo()`. Finally, `proj.cpp` contains the `main()` function, which makes a call to `foo()`. With this in mind, one might attempt to build this project with the following command:

Terminal

```
$ clang++ -std=c++23 proj.cpp
```

However, running this command will produce a linker error:

Terminal

```
$ clang++ -std=c++23 proj.cpp
Undefined symbols for architecture x86_64:
  "foo()", referenced from:
      _main in proj-94090f.o
ld: symbol(s) not found for architecture x86_64
```

This error can be immediately identified as a linker error due to the "`ld:`" in the output. The error message states that the symbol `foo()` is undefined. This occurs because `foo()` is defined in `foo.cpp`, but the linker is unable to find it when compiling only `proj.cpp`. The compiler cannot resolve references to `foo()` because the definition is in a different translation unit (i.e., `foo.cpp`).

To resolve this issue, we need to explicitly tell the linker where to find the definition of `foo()`. For illustrative purposes, we will first build the project manually, step by step, and then show how to compile and link it in a single command. First, note that since this is a linker error, we can still compile `proj.cpp` just fine with the `-c` flag:

Terminal

```
$ clang++ -std=c++23 -c proj.cpp
```

This command produces the object file `proj.o`. Similarly, this command:

Terminal

```
$ clang++ -std=c++23 -c foo.cpp
```

produces `foo.o`. We can then pass these object files to `Clang` to create the final executable. Heres how the complete process looks:

Terminal

```
$ clang++ -std=c++23 -c proj.cpp
$ clang++ -std=c++23 -c foo.cpp
$ clang++ proj.o foo.o
$ ./a.out
fooing...
```

When the compiler compiles `proj.cpp`, it encounters an external reference to `foo()`. At this point, the compiler trusts that the definition of `foo()` will be available to the linker. We can confirm this by inspecting the symbol table of `proj.o` using `objdump`:

Terminal

```
$ objdump -tC proj.o
SYMBOL TABLE:
0000000000000000 l     F __TEXT,__text ltmp0
0000000000000018 l     O __LD,__compact_unwind ltmp1
0000000000000000 g     F __TEXT,__text _main
0000000000000000         *UND* foo()
```

Notice the `UND` (undefined) next to `foo()`. This indicates that `foo()` is an undefined symbol in `proj.o`, meaning the linker will need to resolve this symbol when the object files are linked together. To build the project in one fell swoop, we can simply pass all the translation unit source files to `Clang` at once:

Terminal

```
$ clang++ -std=c++23 proj.cpp foo.cpp
$ ./a.out
fooing...
```