C++

Muchang Bahng

Winter 2024

Contents

1 Translation

Translating C++ code to a binary consists of multiple steps:

1. Preprocessing the code.
2. Compiling each file independently.
3. Linking all the files.

Conventionally, all of these are called *compiling*, but it really isn't.

1.1 Preprocessing

When preprocessing, we do some boring stuff like removing comments. However, the main job is to take care of **preprocessing directives**, which are expressions with the `#` symbol. The most obvious is the `#include` directives, which **replaces the include directive with the contents of the included file**. That is, `#include` is really just a way to substitute code.

1. including with angle brackets, e.g. `#include <iostream>`, means that the compiler is looking for this file in the standard library files.
2. including with double quotes, e.g. `#include "tensor.h"`, means that the compiler is looking for this file locally in your project directory. It means you've written it.

Other directives is the `#define` directive.

1. You can define it to substitute text. It is conventionally in all upper-case.

```
#define NAME "Muchang"  // all instances of NAME will be replaced with "Muchang"
```

2. Or you can define it without substitution text, where further occurrences of `NAME` will be replaced by nothing.

```
#define NAME
```

The second isn't used for substitution, but rather for **conditional compilation**, which can be useful. You just wrap C++ statements around as such.

```
#ifdef NAME
...
#endif
```

```
#ifndef NAME
...
#endif
```

To see the output after preprocessing, use the `-E` flag.

```
g++ main.cpp -E
```

1.2 Compilation

We only compile files one at a time and independently. When the compiler compiles a file, it goes through each line sequentially. Therefore, we must ensure that all functions/variables/classes are *declared* first before they are called. *Forward declaration* makes this a lot easier.

There is a difference between a declaration and a definition.

Definition 1.1 (ODR)

Remember the ODR (One Definition Rule):

1. Within a file, each function, variable, type, or template in a given scope can only have one definition. Definitions occurring in different scopes (e.g. local variables defined inside different functions, or functions defined inside different namespaces) do not violate this rule.
2. Within a program, each function or variable in a given scope can only have one definition.[a]

To be honest, ODR 2 really implies ODR 1, since once the directives are preprocessed or the object files are linked, we are really left with one executable file.

Example 1.1 (ODR 1 Violation)

The following shows that in the same file, there are multiple variables defined in the function scope of `main`, and there are two definitions of `foo` in the global scope.

```cpp
int main() {
  int x;
  int x;

  return 0;
}
.
```

```cpp
int foo() { return 5; }
int foo() { return 5; }

int main() {
  std::cout << foo();
  return 0;
}
```

Example 1.2 (ODR 2 Violation)

Say that `main.cpp` has the `main()` method that calls on `int add(int x, int y)`, which is forward declared. However, say that we define add in two places.

```cpp
// foo.cpp
int add(int x, int y) {
  return x + y;
}
```

```cpp
// bar.cpp
int add(int x, int y) {
  return x + y;
}
```

Then, if we run `g++ main.cpp foo.cpp bar.cpp`, the linker will complain that there is a function redefinition.

1.3 Linking

Remember, declaration is not the same thing as definition. When we do the linking, we go through all the source files in our project and match all the declarations with our definitions. The source files must all be written in the compile command.

```
g++ main.cpp add.cpp
g++ add.cpp main.cpp
```

This should not be order dependent. The source files can be

[a] This rule exists because programs can have more than one file. For example, if you have two definitions of `int add(int, int)` in two different files, the linker does not know which one to connect the declaration to.

```cpp
// main.cpp
int add(int x, int y); // declaration

int main() {
  int z = add(2, 3);
  return 0;
}
```

```cpp
// add.cpp
// definition
int add(int x, int y) {
  return x + y;
}
.
.
```

1.4 Header Files

To be honest, we can just include forward declarations everywhere, but this does not scale well to large projects. If we had a set of declarations that we wanted to use over a bunch of files, we can package them nicely using a **header file**.

If we have a bunch of functions and classes written in `foo.cpp`, then it is conventional to write a `foo.h` that contains all the declarations of these expressions. Then, whenever we need to write a new file `bar.cpp` that uses functions from `foo.cpp`, we can just `#include "foo.h"`, which replaces this directive (by the preprocessor) with all the forward declarations in `foo.h`. Boom easy.

```cpp
// add.cpp
int add(int x, int y) {
  return x + y;
}
```

```cpp
// add.h
int add(int x, int y);
.
.
```

Therefore when we call `add` in `main.cpp`, we can just `#include "add.h"` to put in the declarations, making everything good. Conventionally, it is best practice for a source file to also include its paired header (e.g. `add.cpp` should also contain `#include "add.h"` at the top). This allows the compiler to discover inconsistencies between the two files, and this extra cost is negligible.[1]

Example 1.3 (Definitions inside Header Files)

You should not add definitions (only declarations) to header files since if they are included in multiple header files, then we would have different definitions of the same function, leading to ODR 2 violation. Take a look at the following.

```cpp
// square.h
int getSquareSides() {
    return 4;
}
```

```cpp
// wave.h
#include "square.h"
```

With the following.

```cpp
#include "square.h"
#include "wave.h"
int main() {
  return 0;
}
```

This won't compile since

1. by including `square.h`, we have defined `getSquareSides()` in the global scope of `main.cpp`.
2. by including `wave.h`, we have included `square.h` which then substitutes this line with the definition of `getSquareSides()` again.

This is an ODR 1 violation.

[1] https://www.learncpp.com/cpp-tutorial/cpp-faq/#pairedheader

The simple fix to the above is to just remove the `#include "wave.h"`, but what if we needed some other function from `wave.h`? Resolving this issue is not trivial if say, half of the functions in `square.h` is needed in `wave.h` and the other half is needed in `main.cpp`. We must include both of them in `main.cpp`, but then we have an inevitable redefinition. Without separating `square.h` into separate files, solving this is impossible.

Even if we didn't have definitions in header files in the first place (which is bad practice in general), repeated declarations, which are still fine, are also not really ideal either. Furthermore, custom types are typically defined in header files, so redefining them leads to an ODR violation.

Definition 1.2 (Header Guards)

Fortunately, we have **header guards**, which are conditional compilation directives that tell the compiler to include a header file at most once to the main file. You can do this in two ways.

1. Just put this to the top of the header file. The compiler will take care of redeclaration/redefinitions for you. This isn't always fail-safe.

```
#pragma once
```

2. More manually, we can use a conditional compilation directive. Put this on the top of the header.

```
#ifndef HEADERFILE_H
#define HEADERFILE_H

...Header Contents...

#endif
```

   In the beginning, `HEADERFILE_H` is not defined, so we include all of this. In a second inclusion though, `HEADERFILE_H` is defined, so the preprocessor removes this.

Note that header guards limit the number of times a header can be included in a single given file, but the header may still be repeated across separate project files. This is what we want.

2 Types

Some types have the `_t` suffix, which just represents type. Some types have this and others don't. In C++, there is no exact size for each fundamental type (except for `char`, which is always 1 byte). There is however a lower bound, so you should always use the lower bound and for maximum portability, never assume that a type can store more bytes.

1. `sizeof(short) = 4`
2. `sizeof(int) = 4`
3. `sizeof(long) = 4`
4. `sizeof(long long) = 8`
5. `sizeof(float) = 4`
6. `sizeof(double) = 8`
7. `sizeof(long double) = 8`

Where does the `sizeof` operator come from?

We can also convert some types to different types. If the types are relatively similar, then the C++ implementation may do an **implicit typecast**. If not, then we do an **explicit typecast** in the following ways.

1. `static_cast<T>(foo)`
2. `(T)foo`
3. `dynamic_cast<T>(foo)`