2.9 — Introduction to the preprocessor

ALEX AUGUST 30, 2021

Translation and the preprocessor

When you compile your code, you might expect that the compiler compiles the code exactly as you've written it. This actually isn't the case.

Prior to compilation, the code file goes through a phase known astranslation. Many things happen in the translation phase to get your code ready to be compiled (if you're curious, you can find a list of translation phases here). A code file with translations applied to it is called a translation unit.

The most noteworthy of the translation phases involves the preprocessor. The preprocessor is best thought of as a separate program that manipulates the text in each code file.

When the preprocessor runs, it scans through the code file (from top to bottom), looking for preprocessor directives. Preprocessor directives (often just called *directives*) are instructions that start with a*#* symbol and end with a newline (NOT a semicolon). These directives tell the preprocessor to perform specific particular text manipulation tasks. Note that the preprocessor does not understand C++ syntax -- instead, the directives have their own syntax (which in some cases resembles C++ syntax, and in other cases, not so much).

The output of the preprocessor goes through several more translation phases, and then is compiled. Note that the preprocessor does not modify the original code files in any way -- rather, all text changes made by the preprocessor happen temporarily in-memory each time the code file is compiled.

In this lesson, we'll discuss what some of the most common preprocessor directives do.

As an aside…

`Using directives` (introduced in lesson2.8 -- Naming collisions and an introduction to namespaces) are not preprocessor directives (and thus are not processed by the preprocessor). So while the term `directive` *usually* means a `preprocessor directive`, this is not always the case.

Includes

You've already seen the *#include* directive in action (generally to #include <iostream>). When you *#include* a file, the preprocessor replaces the #include directive with the contents of the included file. The included contents are then preprocessed (along with the rest of the file), and then compiled.

Consider the following program:

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, world!";
    return 0;
}
```

When the preprocessor runs on this program, the preprocessor will replace `#include <iostream>` with the preprocessed contents of the file named “iostream”.

Since *#include* is almost exclusively used to include header files, we’ll discuss *#include* in more detail in the next lesson (when we discuss header files in more detail).

Macro defines

The *#define* directive can be used to create a macro. In C++, a macro is a rule that defines how input text is converted into replacement output text.

There are two basic types of macros: *object-like macros*, and *function-like macros*.

Function-like macros act like functions, and serve a similar purpose. We will not discuss them here, because their use is generally considered dangerous, and almost anything they can do can be done by a normal function.

Object-like macros can be defined in one of two ways:

```
#define identifier
#define identifier substitution_text
```

The top definition has no substitution text, whereas the bottom one does. Because these are preprocessor directives (not statements), note that neither form ends with a semicolon.

Object-like macros with substitution text

When the preprocessor encounters this directive, any further occurrence of the identifier is replaced by *substitution_text*. The identifier is traditionally typed in all capital letters, using underscores to represent spaces.

Consider the following program:

```cpp
#include <iostream>

#define MY_NAME "Alex"

int main()
{
    std::cout << "My name is: " << MY_NAME;

    return 0;
}
```

The preprocessor converts the above into the following:

```
// The contents of iostream are inserted
here

int main()
{
    std::cout << "My name is: " << "Alex";

    return 0;
}
```

Which, when run, prints the output `My name is: Alex`.

Object-like macros were used as a cheaper alternative to constant variables. Those times are long gone as compilers got smarter and the language grew. Object-like macros should only be seen in legacy code anymore.

We recommend avoiding these kinds of macros altogether, as there are better ways to do this kind of thing. We discuss this more in lesson 4.14 -- Const, constexpr, and symbolic constants.

Object-like macros without substitution text

Object-like macros can also be defined without substitution text.

For example:

```
#define
USE_YEN
```

Macros of this form work like you might expect: any further occurrence of the identifier is removed and replaced by nothing!

This might seem pretty useless, and it *is useless* for doing text substitution. However, that's not what this form of the directive is generally used for. We'll discuss the uses of this form in just a moment.

Unlike object-like macros with substitution text, macros of this form are generally considered acceptable to use.

Conditional compilation

The *conditional compilation* preprocessor directives allow you to specify under what conditions something will or won't compile. There are quite a few different conditional compilation directives, but we'll only cover the three that are used by far the most here: *#ifdef*, *#ifndef*, and *#endif*.

The *#ifdef* preprocessor directive allows the preprocessor to check whether an identifier has been previously *#define*d. If so, the code between the *#ifdef* and matching *#endif* is compiled. If not, the code is ignored.

Consider the following program:

```
#include <iostream>

#define PRINT_JOE

int main()
{
#ifdef PRINT_JOE
    std::cout << "Joe\n"; // if PRINT_JOE is defined, compile this code
#endif

#ifdef PRINT_BOB
    std::cout << "Bob\n"; // if PRINT_BOB is defined, compile this code
#endif

    return 0;
}
```

Because PRINT_JOE has been #defined, the line `cout << "Joe\n"` will be compiled. Because PRINT_BOB has not been #defined, the line `cout << "Bob\n"` will be ignored.

***#ifndef* is the opposite of *#ifdef*, in that it allows you to check whether an identifier has *NOT* been *#define*d yet.**

```
#include <iostream>

int main()
{
#ifndef PRINT_BOB
    std::cout << "Bob\n";
#endif

    return 0;
}
```

This program prints “Bob”, because PRINT_BOB was never *#define*d.

In place of `#ifdef PRINT_BOB` and `#ifndef PRINT_BOB`, you’ll also see `#if defined(PRINT_BOB)` and `#if !defined(PRINT_BOB)`. These do the same, but use a slightly more C++-style syntax.

#if 0

One more common use of conditional compilation involves using *#if 0* to exclude a block of code from being compiled (as if it were inside a comment block):

```cpp
#include <iostream>

int main()
{
    std::cout << "Joe\n";

#if 0 // Don't compile anything starting here
    std::cout << "Bob\n";
    std::cout << "Steve\n";
#endif // until this point

    return 0;
}
```

The above code only prints “Joe”, because “Bob” and “Steve” were inside an*#if 0*block that the preprocessor will exclude from compilation.

This also provides a convenient way to “comment out” code that contains multi-line comments (which can’t be commented out using another multi-line comment due to multi-line comments being non-nestable):

```
#include <iostream>

int main()
{
    std::cout << "Joe\n";

#if 0 // Don't compile anything starting here
    std::cout << "Bob\n";
    /* Some
     * multi-line
     * comment here
     */
    std::cout << "Steve\n";
#endif // until this point

    return 0;
}
```

Object-like macros don’t affect other preprocessor directives

Now you might be wondering:

```
#define PRINT_JOE

#ifdef PRINT_JOE
// ...
```

Since we defined *PRINT_JOE* to be nothing, how come the preprocessor didn’t replace *PRINT_JOE* in ***#ifdef PRINT_JOE*** with nothing?

Macros only cause text substitution for normal code. Other preprocessor commands are ignored. Consequently, the *PRINT_JOE* in ***#ifdef PRINT_JOE*** is left alone.

For example:

```
#define FOO 9 // Here's a macro substitution

#ifdef FOO // This FOO does not get replaced because it’s part of another preprocessor directive
    std::cout << FOO; // This FOO gets replaced with 9 because it's part of the normal code
#endif
```

In actuality, the output of the preprocessor contains no directives at all -- they are all resolved/stripped out before compilation, because the compiler wouldn’t know what to do with them.

The scope of defines

Directives are resolved before compilation, from top to bottom on a file-by-file basis.

Consider the following program:

```
#include <iostream>

void foo()
{
#define MY_NAME "Alex"
}

int main()
{
 std::cout << "My name is: " << MY_NAME;

 return 0;
}
```

Even though it looks like ***#define MY_NAME “Alex”*** is defined inside function *foo*, the preprocessor won’t notice, as it doesn’t understand C++ concepts like functions. Therefore, this program behaves identically to one where ***#define MY_NAME “Alex”*** was defined either before or immediately after function *foo*. For general readability, you’ll generally want to #define identifiers outside of functions.

Once the preprocessor has finished, all defined identifiers from that file are discarded. This means that directives are only valid from the point of definition to the end of the file in which they are defined. Directives defined in one code file do not have impact on other code files in the same project.

Consider the following example:

function.cpp:

```cpp
#include <iostream>

void doSomething()
{
#ifdef PRINT
    std::cout << "Printing!";
#endif
#ifndef PRINT
    std::cout << "Not printing!";
#endif
}
```

main.cpp:

```cpp
void doSomething(); // forward declaration for function doSomething()

#define PRINT

int main()
{
    doSomething();

    return 0;
}
```

The above program will print:

```
Not printing!
```

Even though PRINT was defined in *main.cpp*, that doesn't have any impact on any of the code in *function.cpp* (PRINT is only #defined from the point of definition to the end of main.cpp). This will be of consequence when we discuss header guards in a future lesson.

Next lesson
2.10 Header files

Back to table of contents

Previous lesson
2.8 Naming collisions and an introduction to namespaces

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

DP N N FOUT

Newest

X

X