1.10 — A first look at the preprocessor

BY ALEX ON JUNE 3RD, 2007 | LAST MODIFIED BY ALEX ON OCTOBER 8TH, 2017

The preprocessor is perhaps best thought of as a separate program that runs just before the compiler when you compile your program. When the preprocessor runs, it simply scans through each code file from top to bottom, looking for directives. **Directives** are specific instructions that start with a # symbol and end with a newline (NOT a semicolon). There are several different types of directives, which we will cover below.

The preprocessor is not smart -- it does not understand C++ syntax; rather, it simply manipulates text before the compiler runs. The output of the preprocessor is then sent to the compiler. 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.

Includes

You've already seen the #include directive in action. When you #include a file, the preprocessor copies the contents of the included file into the including file at the point of the #include directive. This is useful when you have information that needs to be included in multiple places (as forward declarations often are).

The #include command has two forms:

`#include <filename>` tells the preprocessor to look for the file in a special place defined by the operating system where header files for the C++ runtime library are held. You'll generally use this form when you're including headers that come with the compiler (e.g. that are part of the C++ standard library).

`#include "filename"` tells the preprocessor to look for the file in the directory containing the source file doing the #include. If it doesn't find the header file there, it will check any other include paths that you've specified as part of your compiler/IDE settings. That failing, it will act identically to the angled brackets case. You'll generally use this form for including your own header files.

Macro defines

The #define directive can be used to create a macro. A **macro** is a rule that defines how an input sequence (e.g. an identifier) is converted into a replacement output sequence (e.g. some 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, because their use is generally considered dangerous, and almost anything they can do can be done by an (inline) 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 declarations (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 'identifier' is replaced by 'substitution_text'. The identifier is traditionally typed in all capital letters, using underscores to represent spaces.

Consider the following snippet:

```
#define MY_FAVORITE_NUMBER 9

std::cout << "My favorite number is: " << MY_FAVORITE_NUMBER << std::endl;
```

The preprocessor converts this into the following:

```
std::cout << "My favorite number is: " << 9 << std::endl;
```

Which, when run, prints the output `My favorite number is: 9`.

We discuss this case (and why you shouldn't use it) in more detail in section **2.9 -- 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 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. The only conditional compilation directives we are going to cover in this section are #ifdef, #ifndef, and #endif.

The #ifdef preprocessor directive allow the preprocessor to check whether a value has been previously #defined. If so, the code between the #ifdef and corresponding #endif is compiled. If not, the code is ignored.

Consider the following snippet of code:

```
#define PRINT_JOE

#ifdef PRINT_JOE
std::cout << "Joe" << std::endl;
#endif

#ifdef PRINT_BOB
std::cout << "Bob" << std::endl;
#endif
```

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

#ifndef is the opposite of #ifdef, in that it allows you to check whether a name has NOT been defined yet.

```
#ifndef PRINT_BOB
std::cout << "Bob" << std::endl;
#endif
```

This program prints "Bob", because PRINT_BOB was never #defined.

Conditional compilation is used quite a bit in the form of header guards. We'll take a look at those in the next lesson.

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
```

The scope of defines

Directives are resolved before compilation, from top to bottom on a file-by-file basis. Once the preprocessor has finished, all directives 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:

```
#include <iostream>

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

main.cpp:

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

int main()
{
#define PRINT

    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 anything in function.cpp. This will be of consequence in the next lesson on header guards.

Finally, note that directives defined in a header file can be #included into multiple code files. This provides a way for directives to be defined in one place and used in multiple code files. We'll see an example of this in the next lesson as well.


1.10a -- Header guards


Index


1.9 -- Header files