Chapter 10: Pre-Processing

Notes

- There are no placement restrictions on pre-processor directives. They can occur literally anywhere in source code.

The "Other" Compiler

Thus far you've had to deal increasingly more with the alien '#include' in order to access helper functions. It's finally time to discover what these are and what other interesting things there are related to it. You've already noticed it doesn't follow any syntax I've described. Sure it seems to take a file parameter, but it's enclosed in pointy brackets (<>) rather than double quotes; what gives?

Those lines, also known in verbal discussions as "pound includes", are *pre-processor directives*, or a command interpreted by the C++ Pre-Processor. The C++ pre-processor is like a compiler, but it mucks with the source code itself before it is actually compiled. All pre-processor directives begin with a pound sign. This pound sign must be the first character that isn't white space on its line.

Pre-processor directives are practically a second language that lives in harmony with C++. The directives allow for source code manipulation before the source code is compiled. A model airplane comes with instructions and parts. The instructions would be the pre-processor commands and the parts would be the code. The end result is a model airplane that can be flown much like a program that can be executed.

Obviously the term *pre*-processor comes from the fact that these directives are interpreted prior to the code being compiled. So how do they affect the code? Well, the one you have used already doesn't affect code it simply groups it together.

#include

The term "including files" applies to using the 'include' pre-processor directive. This directive causes the entire contents of the file specified to be inserted or *included*. You can't use a function until it is declared, which is why you must *include* other files which contain the function prototype. If you forget to include the file that a function is prototyped in, the compiler will scream (figuratively speaking) in your face because it won't know what you're talking about. Consider:

```
int main()
{
```

```
    cout << "Hello World" << endl;
}
```

The compiler can parse this into separate parts, but it doesn't know what some of them mean because they haven't been declared. In this case the syntax is all correct, but the identifiers 'cout' and 'endl' are unknown, as well as the insertion operator (<<) as it applies to all these things.

Anything that isn't a C++ keyword is user-defined; that is, it is defined/declared in source code. With its built-in knowledge, a C++ compiler would be able to understand what "Hello World" is, but it doesn't know what 'cout' is. The declarations necessary to use 'cout' are hidden deep within the file 'iostream.h' which is why it must be included:

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

The file 'iostream.h' comes standard with all C++ software. It contains declarations necessary for using 'cout' and 'cin' among other things. When a source file is included it is also pre-processed. That is, if the file 'iostream.h' contains an include directive (#include) then it will be processed as well. Many times including one file, such as 'iostream.h', will cause many more to be included as well.

C++ files with an '.h' extension usually indicate that they are *header files*. A header file contains C++ source code that is typically only declarative in nature: like function prototypes but no function bodies. A header file is like the header on a page in a book. It doesn't contain any content, it simply declares things. Header files will rarely contain definitions.

Actual definitions for the things declared in standard header files such as 'iostream.h' are in code *libraries*. That is, source code that is already compiled into packages of sort. These pre-compiled packages need only to be attached to your program after it is also compiled. The header file is the link between the two. The compiler will link together all the declarations so that your program and these libraries can interact. If you were to build a new model airplane, you would most likely not build everything from scratch. You'd probably use a pre-made engine. The engine hooks to the rest of your plane through some kind of connector. Think of your custom plane as your C++ program and the engine as a library. The header file does not *cause* this to happen so much as it *allows* this to happen. The header file is the compilers eyes for seeing the functions in the library and for the programmer calling them in their program. With the model airplane you *know* there are connectors on the engine because you see them. A code library is just a block of pure machine logic, but with the header file the compiler knows what to look for. I'll talk more on code libraries, as if I haven't enough, in a later chapter when I explain how to break large programs into separate modules that are linked together.

Custom Headers

There are two syntaxes for pound includes. One is using greater than less than, “pointy”, brackets as you have done and the other is using double quotes. The former causes files to be included from default directories that the compiler knows about. These directories are specific to the C++ software, but all compilers allow these “default” directories to be modified. So when you run your C++ compiler from a console, it knows that the call to include <iostream.h> means to look in a specific directory (or directories) on the hard drive; not in the current directory.

On the other hand, including a file using double quotes is supposed to include files from the current directory, i.e. where your source file is. This method is typically used for *custom headers*, or ones that you have written specific to your program. Really everything is a custom header because it has been written by someone and it declares things not built into the C++ language. But the term can also mean more specific things. The file ‘iostream.h’ is common and used among many programs while the file ‘bigfootfinder.h’ is pretty-much specific to only one.

Creating a custom header is as simple as creating a new source file. Rather than saving it with a ‘.cpp’ extension, you’ll save it with ‘.h’. It really only makes sense to have a custom header if you have lots of functions and other things to declare. You can also use it for comments about your program as well as listing out all your other includes. Consider the following source:

<insert source with multiple includes and function declarations>

The above source could be split into two files: a header file and a source file. A *source file* is generally any text file that contains non-declaratory C++ code (i.e. actual meat). The source file would contain all the real logic and the header file would contain the standard includes and the function prototypes. The source file would contain a single include at the top for the custom header:

<show source and header files>

Custom headers are useful for large programs, but certainly not small ones like the ones I use for my examples. It is also certainly possible to put actual C++ code into header files because all the directive actually does is paste in the contents of the file. In small programs there are really no draw-backs, but using this technique to build a program from separate modules (as will be seen later) it is extremely hazardous.

Author’s Preference: Don’t put *any* non-declaratory C++ code into your headers. You’ll find it a pain it the ass later on so you might as well save yourself some of that pain.

Macros

Include directives are far from the only commands available in the pre-processor stock-pile. There are also directives for defining, comparing, and destroying *macros*. A macro

in the C++ pre-processor is basically what a variable is *in* C++. It contains a value. The difference is that a macro contains source to be inserted into the source code. All macros are inherently strings, because they are a sequence of characters meant to be inserted into the source code text. A macro is like an include file in that it is replaced with the source code it represents, like an include directive is replaced by the contents of the file being included.

Macros are created using the *define* directive:

```
#define <name> <contents>
```

The pointy brackets are to separate the two distinct values, they aren't actually part of the syntax. A macro name follows the same rules as any identifier even though it is part of the pre-processor and not the compiler. The contents of a macro can be literally *anything* that you can type in. Contents begin immediately after the name ends and end when the line ends:

```
#define HELLOWORLD "Hello World"
```

Author's Preference: Typically macros are distinguished from other identifiers in code by making them all upper-case.

The macro above would contain the contents "Hello World" *exactly*. That is, if you put 'HELLOWORLD' anywhere in your code it would be replaced by its contents:

```
cout << HELLOWORLD << endl;
```

The name of a macro outside of a pre-processor command (a line that doesn't begin with a pound sign (#)) is replaced with the contents of the macro. The above code would have been pre-processed to actually read:

```
cout << "Hello World" << endl;
```

A macro can be made to span multiple lines by putting a backslash at the end of each line you want to continue. This in effect causes the multiple lines to be interpreted as one single one:

```
#define HELLOWORLD "Hello \
World"
```

Remember that pre-processing does not follow the normal rules of C++, which is why macros can span multiple lines without breaking a sweat when normal string literals cannot. The above interprets 'HELLOWORLD' to still be "Hello World" as a single line. Macros are always a single line, but since the C++ language is not fickle about line breaks, it's not a big deal.

Comments are still ignored by macro definitions, but compilers may treat them in different ways.

Author's Preference: Best not to upset mean compilers by try to mix single-line comments into multi-line macros. Insert a comment *above* multi-line macros rather than to the right. For single-line macros I still sometimes put single-line comments (starting with '//') to the right of the definition.

Usage of macros varies greatly. Because they contain actual source code, they can be used for a great deal of things. Macros existed in C++'s predecessor C and that is what NULL was. Somewhere in the mounds of standard headers, the following directive existed:

```
#define NULL 0
```

And we've interchanged the meaning of zero and NULL ever since (even when people tell us not too ☺). In C++ NULL is typically declared as a constant global void pointer to a zero address:

```
const void *NULL = 0;
```

It amounts, however, to the same thing. Macros could have just as easily been used for this:

```
#define NULL ((void*)0)
```

And they just might be in some compilers. Macros are used less in C++ than in C, because of such things as constants. In olden days people told us to define PI with a macro so we'd never have to write 3.14 -- blah blah -- ever again, but these days we're told to use a global constant. Follow your heart; they amount to practically the same thing. ☺

Macros can also be defined outside of your source files by the compiler and some of them are implicitly defined. That is, they always exist and are used for whatever purpose you want. These "standard" macros are:

<list and explain standard macros>

Defining custom macros with your compiler is compiler-specific. Most compilers have a flag or something you can set for this purpose. Consult your C++ software's documentation for more details.

Destroying Macros

Unlike variables you can actually destroy macros in specific places. Say for instance you only use this macro in a certain place in your source file … for simplifying the code or

whatnot. You could define it at the top and then destroy it at the bottom. To destroy a macro, use the *undefine* directive '#undef':

```
#undef <name>
```

For example:

```
#define HELLOWORLD "Hello World"
cout << HELLOWORLD << endl;
#undef HELLOWORLD
```

A macro exists as soon as the compiler happens across the define directive and will continue to exist until the pre-processing is finished or an undefine directive is reached. All macros are, for all intensive purposes, equivalent to global variables.

Flow Control

It is possible in the pre-processor language to create rudimentary flow control constructs. This is done through the pre-processor command 'if'. True conditions cause the code associated within the 'if' command to be present, otherwise it is removed completely. Remember that pre-processing affects the source code *itself* (not the logic), before it is actually compiled. For example, if you want certain code present only if the macro INDEVELOPMENT is set to one (1), you could do the following:

```
#if (INDEVELOPMENT == 1)
// code to insert
#endif
```

The syntax of the 'if' pre-processor command is much like that of the 'if' statement in C++ code. One of the differences is that its boundaries are marked by '`#if`' and '`#endif`' rather than curly braces. Also, the condition may not span multiple lines by default. You must end a line with a backslash in order to continue it on the next. Without the backslash, the pre-processor only sees the single line.

```
#if (INDEVELOPMENT == 1 && \
NOTINDEVELOPMENT == 0)
// code to insert
#endif
```

Numeric and string literals are acceptable in the conditions. Unlike within the language itself, the C++ pre-processer can handle the comparison of strings as if they were primitive data types:

```
#if (PROGNAME == "hello.cpp")
// code to insert
#endif
```

This is because all macros, even numbered ones, are always actually seen as strings inside these comparisons; so they are treated as simply "a piece of data" rather than a piece of data of a specific type like 'int' or 'char'. But just because macros are actually strings of raw data does not mean you can't test numeric conditions.

Typical numeric conditional operators like greater-than (>) and less-than (<) are still acceptable, but they will not work correctly on data that cannot be translated into a number:

```
#define ZING 2

#if (ZING > 1)
// code to insert
#endif
```

An 'else' command is also present if you want a proper 'if/else' without resorting to two (2) separate if's:

```
#if (ZING > 1)
// code to insert
#else
// code to insert
#endif
```

However, it is not feasible by the pre-processors of some C++ software to do nested if's. Because of this, you cannot use the standard 'if else if' technique that you would with the C++ language. Instead, you can use the 'elif' command which is syntactically equivalent to an 'if' command, but must come after a previous 'if' (and before the 'endif'):

```
#if (ZING > 1)
// code to insert
#elif (ZING < 1)
// code to insert
#else
// code to insert
#endif
```

Blarg.

Existence Checking

If a macro does not exist, a condition using it will not work properly. The value of a non-existent macro is implementation dependent. You can, however, detect if a macro is defined or not. This is done in one of two ways: using '#ifdef' or the 'defined' operator.

The first way is a custom flow-control command called 'ifdef' or "if defined". The one argument to this should be the name of the macro you are checking the existence of. The

rest of the block is the same as before: possibly containing an ‘else’ command and ending with ‘endif’:

```
#ifdef SOMEMACRO
// code to insert
#else
// code to insert
#endif
```

An alternative to this is the ‘defined’ operator which you give a macro name and use in conjunction with the normal ‘if’ command:

```
#if (defined(SOMEMACRO))
#else
#endif
```

I don’t know of any advantages of one over the other. However, I *have* found that it is perfectly acceptable to nest normal ‘if’ commands *within* and ‘ifdef’ block:

```
#ifdef SOMEMACRO
    #if (DEBUG == 1)
        // code to insert
    #endif
#endif
```

The opposite of ‘ifdef’ is ‘ifndef’ which is equivalent to ‘`if !defined(`*`macro`*`)`’, it will insert the code following it only if the macro specified is *not* defined:

```
#ifndef SOMEMACRO
   // code to insert
#endif
```

Blarg.

Parenthesis Unnecessary

Although I’ve used them extensively in my examples, parentheses are largely unnecessary with pre-processor commands. Unlike the C++ language ‘if’s themselves, a pre-processor ‘if’ does not require them:

```
#if DEBUG == 1
    // code to insert
#endif
```

Even the ‘defined’ function does not require them:

```
#if defined DEBUG
    // code to insert
#endif
```

Author's Preference: I still use parenthesis with pre-processor commands because it makes them easier to read.

Raising Errors

The 'error' pre-processor command can be used to raise errors. The syntax is:

```
#error <reason message>
```

When this command is reached, the compiler will fail the compilation and print out the message you specified.

Author's Suggestion: I've used this in the past to make sure certain header files, which define specific macros, have been included.

Concise Types

Using macros, you can create coding short-cuts. Many of these are usually short-cuts to typing in variable *types*. When you type in 'unsigned int' over and over you start developing a hunger to shorten it to 'uint' or something similar. This could be done like so:

```
#define uint unsigned int
```

Now, having the above macro defined you could type in:

```
uint x;
```

And the pre-processor would expand it to the following before it was compiled:

```
unsigned int x;
```

There is another way to do the same thing within the language of C++ rather than the pre-processor. This is through the keyword 'typedef' which is an abbreviation for "type definition". Abbreviate type names can be created with this keyword that are still friendly with debugging and source-code spidering software. Simply put, if you want to define a short-cut type name use 'typedef' rather than '#define'.

The syntax of this keyword looks exactly like a variable declaration except for being preceded by 'typedef'. Rather than declaring a variable, however, you are declaring a type of the same name. So, to declare[1] the 'uint' type with 'typdef' as we had done above with macros:

[1] A type is incidentally both declared *and defined* when it is created with 'typedef'.

```
typedef unsigned int uint;
```

For all practical purposes, this method is simply the *reverse* of macros where you specify the name (our abbreviated type) followed by the value (the actual type itself). Once you have declared the type, you can use it as if it *was* a type:

```
uint x;
uint *px;
```

Yes, you can even use a short-cut to create a pointer of the type abbreviated. Even more complicated, you can declare a 'typedef' to *be* a pointer type and have a pointer to that (more on this in "pointer pointers" later on in the book). Sometimes 'typedef' is used to create "reference types" which are simply a pointer to the abbreviated type:

```
typedef unsigned int *uint_ptr;
```

Anytime we create a variable with the above 'uint_ptr', we create a pointer to 'unsigned int' variables:

```
uint x;
uint_ptr px = &x;
```

Blarg.

Pragma Directive

The 'pragma' directive may as well have been named 'enigma' instead. It means whatever the compiler software wants it to mean. A 'pragma' directive can have whatever effect it wants to on the source, resulting program, and even the compiler itself.