Appendix A

The Preprocessor

The preprocessor is that part of the compiler that performs various text manipulations on your program prior to the actual translation of your source code into object code. You can give text manipulation commands to the preprocessor. These commands are called preprocessor directives, and although not technically part of the C++ language, they expand the scope of its programming environment.

The preprocessor is a holdover from C and is not as important to C++ as it is to C. Also, some preprocessor features have been rendered redundant by newer and better C++ language elements. However, since many programmers still use the preprocessor, and because it is still part of the C++ language environment, it is briefly discussed here.

The C++ preprocessor contains the following directives:

#define	#error	#include
#if	#else	#elif
#endif	#ifdef	#ifndef
#undef	#line	#pragma

As is apparent, all preprocessor directives begin with a # sign. Each will be examined here in turn.

#define

#define is used to define an identifier and a character sequence that will be substituted for the identifier each time it is encountered in the source file. The identifier is called a macro name, and the replacement process is called macro substitution. The general form of the directive is

#define macro-name character-sequence

Notice that there is no semicolon in this statement. There can be any number of spaces between the identifier and the start of the character sequence, but once the sequence begins, it is terminated only by a newline.

For example, if you wanted to use the word “UP” for the value 1 and the word “DOWN” for the value 0, you would declare these two #defines:

#define UP 1 #define DOWN 0

These statements will cause the compiler to substitute a 1 or a 0 each time the name UP or DOWN is encountered in your source file. For example, the following will print 102 on the screen:

It is important to understand that the macro substitution is simply the replacing of an identifier with its associated string. Therefore, if you want to define a standard message, you might write something like this:

#define GETFILE "Enter File Name" // ... cout << GETFILE;

C++ will substitute the string “Enter File Name” when the macro name GETFILE is encountered. To the compiler, the cout statement will actually appear to be

cout << "Enter File Name";

No text substitutions will occur if the macro name occurs within a quoted string. For example,

```
#define GETFILE "Enter File Name"
// ...
cout << "GETFILE is a macro name\n";
```

will not print

```
Enter File Name is a macro name
```

but rather will print

```
GETFILE is a macro name
```

If the string is longer than one line, you can continue it on the next by placing a backslash at the end of the line, as shown in this example:

#define LONG_STRING "this is a very long \ string that is used as an example"

It is common practice among C++ programmers to use capital letters for macro names. This convention helps anyone reading the program know at a glance that a macro substitution will take place. Also, it is best to put all #defines at the start of the file, or perhaps in a separate include file, rather than sprinkling them throughout the program.

One last point: C++ provides a better way of defining constants than by using #define. This is to use the const specifier. However, many C++ programmers migrated from C, where #define is commonly used for this purpose. Thus, you will likely see it frequently in C++ code, too.

Function-Like Macros

The #define directive has another feature: The macro name can have arguments. Each time the macro name is encountered, the arguments associated with it are replaced by the actual arguments found in the program. This creates a function-like macro. Here is an example:

```
// Use a function-like macro.

#include <iostream>
using namespace std;

#define MIN(a,b)  (((a)<(b)) ? a : b)

int main()
{
  int x, y;

  x = 10;
  y = 20;
  cout << "The minimum is " << MIN(x, y);

  return 0;
}
```

When this program is compiled, the expression defined by MIN(a,b) will be substituted, except that x and y will be used as the operands. That is, the cout statement will be substituted to look like this:

```
cout << "The minimum is: " << ((x)<(y)) ? x : y);
```

In essence, the function-like macro is a way to define a function that has its code expanded inline rather than called.

The apparently redundant parentheses surrounding the MIN macro are necessary to ensure proper evaluation of the substituted expression because of the relative precedence of the operators. In fact, the extra parentheses should be applied in virtually all function-like macros. Otherwise, there can be surprising results. For example, consider this short program, which uses a macro to determine whether a value is even or odd:

```
// This program will give the wrong answer.

#include <iostream>
using namespace std;

#define EVEN(a) a%2==0 ? 1 : 0
int main()
{
  if(EVEN(9+1)) cout << "is even";
  else cout << "is odd";

  return 0;
}
```

This program will not work correctly because of the way the macro substitution is made. When compiled, the EVEN(9+1) is expanded to

9+1%2==0 ? 1 : 0

As you should recall, the % (modulus) operator has higher precedence than the plus operator. This means that the % operation is first performed on the 1 and that the result is added to 9, which (of course) does not equal 0. This causes the EVEN macro to improperly report the value of 9+1 as odd. To fix the problem, there must be parentheses around a in the macro definition of EVEN, as is shown in this corrected version of the program:

```
// This program is now fixed.

#include <iostream>
using namespace std;

#define EVEN(a) (a)%2==0 ? 1 : 0

int main()
{
  if(EVEN(9+1)) cout << "is even";
  else cout << "is odd";

  return 0;
}
```

Now, the 9+1 is evaluated prior to the modulus operation and the resulting value is properly reported as even. In general, it is a good idea to surround macro parameters with parentheses to avoid unforeseen troubles like the one just described.

The use of macro substitutions in place of real functions has one major benefit: Because macro substitution code is expanded inline, no overhead of a function call is incurred, so the speed of your program increases. However, this increased speed might be paid for with an increase in the size of the program, due to duplicated code.

Although still commonly seen in C++ code, the use of function-like macros has been rendered completely redundant by the inline specifier, which accomplishes the same goal better and more safely. (Remember, inline causes a function to be expanded inline rather than called.) Also, inline functions do not require the extra parentheses needed by most function-like macros. However, function-like macros will almost certainly continue to be a part of C++ programs for some time to come, because many former C programmers continue to use them out of habit.

#error

When the #error directive is encountered, it forces the compiler to stop compilation. This directive is used primarily for debugging. The general form of the directive is

#error error-message

Notice that the error-message is not between double quotes. When the compiler encounters this directive, it displays the error message and other information and terminates compilation. Your implementation determines what information will actually be displayed. (You might want to experiment with your compiler to see what is displayed.)

#include

The #include preprocessor directive instructs the compiler to include either a standard header or another source file into the file that contains the #include directive. The name of a standard header should be enclosed between angle brackets, as shown in the programs throughout this book. For example,

#include <fstream>

includes the standard header for file I/O.

When including another source file, its name can be enclosed between double quotes or angle brackets. For example, the following two directives both instruct C++ to read and compile a file called sample.h:

#include <sample.h>
#include "sample.h"

When including a file, whether the filename is enclosed by quotes or angle brackets determines how the search for the specified file is conducted. If the filename is enclosed between angle brackets, the compiler searches for it in one or more implementation-defined directories. If the filename is enclosed between quotes, then the compiler searches for it in some other implementation-defined directory, which is typically the current working directory. If the file is not found in this directory, the search is restarted as if the filename had been enclosed between angle brackets. Since the search path is implementation defined, you will need to check your compiler's user manual for details.

Conditional Compilation Directives

There are several directives that allow you to selectively compile portions of your program's source code. This process, called conditional compilation, is widely used by commercial software houses that provide and maintain many customized versions of one program.

#if, #else, #elif, and #endif

The general idea behind the #if directive is that if the constant expression following the #if is true, then the code between it and an #endif will be compiled; otherwise, the code will be skipped over. #endif is used to mark the end of an #if block. The general form of #if is

#if constant-expression statement sequence

#endif

If the constant expression is true, the block of code will be compiled; otherwise, it will be skipped. For example:

```
// A simple #if example.

#include <iostream>
using namespace std;

#define MAX 100

int main()
{
#if MAX>10
  cout << "Extra memory required.\n";
#endif

  // ...
  return 0;
}
```

This program will display the message on the screen because, as defined in the program, MAX is greater than 10. This example illustrates an important point. The expression that follows the #if is evaluated at compile time. Therefore, it must contain only identifiers that have been previously defined and constants. No variables can be used.

The #else directive works in much the same way as the else statement that forms part of the C++ language: it establishes an alternative if the #if directive fails. The previous example can be expanded to include the #else directive, as shown here:

```
// A simple #if/#else example.

#include <iostream>
using namespace std;

#define MAX 6

int main()
{
#if MAX>10
  cout << "Extra memory required.\n");
#else
  cout << "Current memory OK.\n";
#endif

  // ...

  return 0;
}
```

In this program, MAX is defined to be less than 10, so the #if portion of the code is not compiled, but the #else alternative is. Therefore, the message Current memory OK. is displayed.

Notice that the #else is used to mark both the end of the #if block and the beginning of the #else block. This is necessary because there can only be one #endif associated with any #if.

The #elif means "else if" and is used to establish an if-else-if ladder for multiple compilation options. The #elif is followed by a constant expression. If the expression is true, then that block of code is compiled, and no other #elif expressions are tested or compiled. Otherwise, the next in the series is checked. The general form is

```
#if expression
  statement sequence
#elif expression 1
  statement sequence
#elif expression 2
  statement sequence
#elif expression 3
  statement sequence
// ...
#elif expression N
  statement sequence
#endif
```

For example, this fragment uses the value of COMPILED_BY to define who compiled the program:

```
#define JOHN 0
#define BOB 1
#define TOM 2

#define COMPILED_BY JOHN

#if COMPILED_BY == JOHN
  char who[] = "John";
#elif COMPILED_BY == BOB
  char who[] = "Bob";
#else
  char who[] = "Tom";
#endif
```

#ifs and #elifs can be nested. In this case, the #endif, #else, or #elif associate with the nearest #if or #elif. For example, the following is perfectly valid:

```
#if COMPILED_BY == BOB
   #if DEBUG == FULL
      int port = 198;
   #elif DEBUG == PARTIAL
      int port = 200;
   #endif
#else
   cout << "Bob must compile for debug output.\n";
#endif
```

#ifdef and #ifndef

Another method of conditional compilation uses the directives #ifdef and #ifndef, which mean "if defined" and "if not defined," respectively, and refer to macro names. The general form of #ifdef is

```
#ifdef macro-name
      statement sequence
#endif
```

If macro-name has been previously defined in a #define statement, the statement sequence between the #ifdef and #endif will be compiled. The general form of #ifndef is

```
#ifndef macro-name
      statement sequence
#endif
```

If macro-name is currently undefined by a #define statement, then the block of code is compiled. Both the #ifdef and #ifndef can use an #else or #elif statement. Also, you can nest #ifdefs and #ifndefs in the same way as #ifs.

#undef

The #undef directive is used to remove a previously defined definition of a macro name. The general form is

```
#undef macro-name
```

Consider this example:

```
#define TIMEOUT 100
#define WAIT 0
// ...
#undef TIMEOUT
#undef WAIT
```

Here, both TIMEOUT and WAIT are defined until the #undef statements are encountered. The principal use of #undef is to allow macro names to be localized to only those sections of code that need them.

Using defined

In addition to #ifdef, there is a second way to determine if a macro name is defined. You can use the #if directive in conjunction with the defined compile-time operator. For example, to determine if the macro MYFILE is defined, you can use either of these two preprocessing commands:

```
#if defined MYFILE
or
#ifdef MYFILE
```

You can also precede defined with the ! to reverse the condition. For example, the following fragment is compiled only if DEBUG is not defined:

```
#if !defined DEBUG
cout << "Final version!\n";
#endif
```

#line

The #line directive is used to change the contents of _ _LINE_ _ and _ _FILE_ _, which are predefined macro names. _ _LINE_ _ contains the line number of the line currently being compiled, and _ _FILE_ _ contains the name of the file being compiled. The basic form of the #line command is

```
#line number "filename"
```

Here number is any positive integer, and the optional filename is any valid file identifier. The line number becomes the number of the current source line, and the filename becomes the name of the source file. #line is primarily used for debugging purposes and for special applications.

#pragma

The #pragma directive is an implementation-defined directive that allows various instructions, defined by the compiler's creator, to be given to the compiler. The general form of the #pragma directive is

```
#pragma name
```

Here, name is the name of the #pragma you want. If the name is unrecognized by the compiler, then the #pragma directive is simply ignored and no error results.

To see what pragmas your compiler supports, check its documentation. You might find some that are valuable to your programming efforts. Typical #pragmas include those that determine what compiler warning messages are issued, how code is generated, and what library is linked.

The # and ## Preprocessor Operators

C++ supports two preprocessor operators: # and ##. These operators are used in conjunction with #define. The # operator causes the argument it precedes to become a quoted string. For example, consider this program:

```
#include <iostream>
using namespace std;

#define mkstr(s)  # s

int main()
{
  cout << mkstr(I like C++);

  return 0;
}
```

The C++ preprocessor turns the line

cout << mkstr(I like C++);

into

cout << "I like C++";

The ## operator is used to concatenate two tokens. Here is an example:

```
#include <iostream>
using namespace std;

#define concat(a, b)  a ## b

int main()
{
  int xy = 10;

  cout << concat(x, y);

  return 0;
}
```

The preprocessor transforms

cout << concat(x, y);

into

cout << xy;

If these operators seem strange to you, keep in mind that they are not needed or used in most programs. They exist primarily to allow some special cases to be handled by the preprocessor.

Predefined Macro Names

C++ specifies six built-in predefined macro names. They are

__LINE__
__FILE__
__DATE__
__TIME__
__STDC__
__cplusplus

The __LINE__ and __FILE__ macros were described in the discussion of #line. Briefly, they contain the current line number and filename of the program when it is being compiled.

The __DATE__ macro contains a string of the form month/day/year that is the date of the translation of the source file into object code.

The __TIME__ macro contains the time at which the program was compiled. The time is represented in a string having the form hour:minute:second.

The meaning of __STDC__ is implementation-defined. Generally, if __STDC__ is defined, then the compiler will accept only standard C/C++ code that does not contain any nonstandard extensions.

A compiler conforming to ANSI/ISO Standard C++ will define __cplusplus as a value containing at least six digits. Nonconforming compilers will use a value with five or fewer digits.