Preprocessor Directives

As you might recall from Chapter 2, "What Is a Program?," the C++ compiler routes your programs through a *preprocessor* before it compiles them. The preprocessor can be called a "pre-compiler" because it preprocesses and prepares your source code for compiling before your compiler receives it.

Because this *preprocess* is so important to C++, you should familiarize yourself with it before learning more specialized commands in the language. Regular C++ commands do not affect the preprocessor. You must supply special non-C++ commands, called *preprocessor directives*, to control the preprocessor. These directives enable you, for example, to modify your source code before the code reaches the compiler. To teach you about the C++ preprocessor, this chapter

- Defines preprocessor directives
- Introduces the `#include` preprocessor directive
- Introduces the `#define` preprocessor directive
- Provides examples of both

Almost every proper C++ program contains preprocessor directives. This chapter teaches you the two most common: `#include` and `#define`.

Understanding Preprocessor Directives

Preprocessor directives are commands that you supply to the preprocessor. All preprocessor directives begin with a pound sign (#). Never put a semicolon at the end of preprocessor directives, because they are preprocessor commands and not C++ commands. Preprocessor directives typically begin in the first column of your source program. They can begin in any column, of course, but you should try to be consistent with the standard practice and start them in the first column wherever they appear. Figure 6.1 illustrates a program that contains three preprocessor directives.

Preprocessor directives

```
// Filename: C6PRE.CPP
// C++ program that demonstrates preprocessor directives.

#include <iostream.h>
#define AGE 28
#define MESSAGE "Hello, world"

main()
{
   int i = 10, age;    // i is assigned a value at declaration
                       // age is still UNDEFINED

   age = 5;            // Defines the variable, age, as five.

   i = i * AGE;        // AGE is not the same as the variable, age.

   cout << i << " " << age << " " << AGE << "\n";  // 280 5 28
   cout << MESSAGE;    // Prints "Hello world".

   return 0;
}
```

Figure 6.1. Program containing three preprocessor directives.

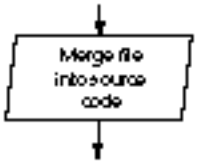
Preprocessor directives temporarily change your source code.

Preprocessor directives cause your C++ preprocessor to change your source code, but these changes last only as long as the compilation. When you look at your source code again, the preprocessor is finished with your file and its changes are no longer in the file. Your preprocessor does not in any way compile your program or change your actual C++ commands. This concept confuses some beginning C++ students, but just remember that your program has yet to be compiled when your preprocessor directives execute.

It has been said that a preprocessor is nothing more than a text-editor on your program. This analogy holds true throughout this chapter.

The #include Directive

The `#include` preprocessor directive merges a disk file into your source program. Remember that a preprocessor directive does nothing more than a word processing command does to your program; word processors also are capable of file merging. The format of the `#include` preprocessor directive follows:




```
#include <filename>
```

or

```
#include "filename"
```

In the `#include` directive, the `filename` must be an ASCII text file (as your source file must be) located somewhere on a disk. To better illustrate this rule, it might help to leave C++ for just a moment. The following example shows the contents of two files on disk. One is called OUTSIDE and the other is called INSIDE.

These are the contents of the OUTSIDE file:

```
Now is the time for all good men

#include <INSIDE>

to come to the aid of their country.
```

The INSIDE file contains the following:

```
A quick brown fox jumped
over the lazy dog.
```

Assume you can run the OUTSIDE file through the C++ preprocessor, which finds the `#include` directive and replaces it with the entire file called INSIDE. In other words, the C++ preprocessor directive merges the INSIDE file into the OUTSIDE file—at the `#include` location—and OUTSIDE expands to include the merged text. After the preprocessing ends, OUTSIDE looks like this:

```
Now is the time for all good men

A quick brown fox jumped
over the lazy dog.

to come to the aid of their country.
```

The INSIDE file remains on disk in its original form. Only the file containing the `#include` directive is changed. This change is only temporary; that is, OUTSIDE is expanded by the included file only for as long as it takes to compile the program.

A few real-life examples might help, because the OUTSIDE and INSIDE files are not C++ programs. You might want to include a file containing common code that you frequently use. Suppose you print your name and address quite often. You can type the following few lines of code in every program that prints your name and address:

```
cout << "Kelly Jane Peterson\n";
cout << "Apartment #217\n";
cout << "4323 East Skelly Drive\n";
cout << "New York, New York\n";
cout << "              10012\n";
```

Instead of having to retype the same five lines again and again, you type them once and save them in a file called MYADD.C. From then on, you only have to type the single line:

```
#include <myadd.c>
```

This not only saves typing, but it also maintains consistency and accuracy. (Sometimes this kind of repeated text is known as a *boilerplate*.)

You usually can use angled brackets, <>, or double quotation marks, "", around the included filename with the same results. The angled brackets tell the preprocessor to look for the *include* file in a default include directory, set up by your compiler. The double quotation marks tell the preprocessor first to look for the include file in the directory where the source code is stored, and then, to look for it in the system's include directory.

Most of the time, you do see angled brackets around the included filename. If you want to include sections of code in other programs, be sure to store that code in the system's include directory (if you use angled brackets).

The #include directive is most often used for system header files.

Even though `#include` works well for inserted source code, there are other ways to include common source code that are more efficient. You learn about one technique, called writing *external functions*, in Chapter 16, "Writing C++ Functions."

This source code `#include` example serves well to explain what the `#include` preprocessor directive does. Despite this fact, `#include` seldom is used to include source code text, but is more often used to include special system files called *header* files. These system files help C++ interpret the many built-in functions that you use. Your C++ compiler comes with its own header files. When you (or your system administrator) installed your C++ compiler, these header files were automatically stored on your hard drive in the system's include directory. Their filenames always end in *.h* to differentiate them from regular C++ source code.

The most common header file is named iostream.h. This file gives your C++ compiler needed information about the built-in `cout` and `cin` operators, as well as other useful built-in routines that perform input and output. The name "*iostream.h*" stands for *input/output stream header*.

At this point, you don't have to understand the iostream.h file. You only have to place this file before `main()` in every program you write. It is rare that a C++ program does not need the iostream.h file. Even when the file is not needed, including it does no harm. Your programs can work without iostream.h as long as they do not use

an input or output operator defined there. Nevertheless, your programs are more accurate and hidden errors come to the surface much faster if you include this file.

Throughout this book, whenever a new built-in function is described, the function's matching header file is included. Because almost every C++ program you write includes a `cout` to print to the screen, almost every program contains the following line:

Include the built-in C++ header file called iostream.h.

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

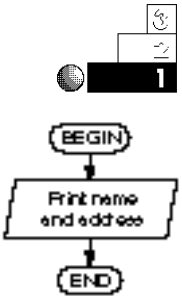
In the last chapter, you saw the `strcpy()` function. Its header file is called string.h. Therefore, if you write a program that contains `strcpy()`, include its matching header file at the same time you include `<iostream.h>`. These appear on separate lines, such as:

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

The order of your include files does not matter as long as you include the files before the functions that need them. Most C++ programmers include all their needed header files before `main()`.

These header files are simply text files. If you like, find a header file such as stdio.h on your hard drive and look at it. The file might seem complex at this point, but there is nothing "hidden" about it. Don't change the header file in any way while looking at it. If you do, you might have to reload your compiler to restore the file.

Examples



1. The following program is short. It includes the name-and-address printing routine described earlier. After printing the name and address, it ends.

```
// Filename: C6INC1.CPP
// Illustrates the #include preprocessor directives.
#include <iostream.h>
```

```
main()
{
#include “myadd.c”
return 0;
}
```

The double quotation marks are used because the file called MYADD.C is stored in the same directory as the source file. Remember that if you type this program into your computer (after typing and saving the MYADD.C file) and then compile your program, the MYADD.C file is included only as long as it takes to compile the program. Your compiler does not see this file. Your compiler acts as if you have typed the following:

```
// Filename: C6INCL1.CPP
// Illustrates the #include preprocessor directive.
#include <iostream.h>
main()
{
cout(“Kelly Jane Peterson\n”;
cout(“Apartment #217\n”;
cout(“4323 East Skelly Drive\n”;
cout(“New York, New York\n”;
cout(“              10012\n”;
return 0;
}
```

This explains what is meant by a preprocessor: The changes are made to your source code before it’s compiled. Your original source code is restored as soon as the compile is finished. When you look at your program again, it appears as originally typed, with the `#include` statement.

2. The following program copies a message into a character array and prints it to the screen. Because the `cout` and `strcpy()` built-in functions are used, both of their header files are included.

```
// Filename: C6INCL3.CPP
// Uses two header files.

#include <iostream.h>
#include <string.h>

main()
{
   char message[20];
   strcpy(message, "This is fun!");
   cout << message;
   return 0;
}
```

The #define Directive

The #define preprocessor directive is used in C++ programming, although not nearly as frequently as it is in C. Due to the const keyword (in C++) that enables you to define variables as constants, #define is not used as much in C++. Nevertheless, #define is useful for compatibility to C programs you are converting to C++. The #define directive might seem strange at first, but it is similar to a search-and-replace command on a word processor. The format of #define follows:

```
#define ARGUMENT1 argument2
```

The #define directive replaces every occurrence of a first argument with a second argument.

where ARGUMENT1 is a single word containing no spaces. Use the same naming rules for the #define statement's first argument as for variables (see Chapter 4, "Variables and Literals"). For the first argument, it is traditional to use uppercase letters—one of the only uses of uppercase in the entire C++ language. At least one space separates ARGUMENT1 from argument2. The argument2 can be any character, word, or phrase; it also can contain spaces or anything else you can type on the keyboard. Because #define is a preprocessor directive and not a C++ command, do not put a semicolon at the end of its expression.

The #define preprocessor directive replaces the occurrence of ARGUMENT1 everywhere in your program with the contents of

argument2. In most cases, the #define directive should go before main() (along with any #include directives). Look at the following #define directive:

Define the AGELIMIT *literal to* 21.

```
#define AGELIMIT 21
```

If your program includes one or more occurrences of the term AGELIMIT, the preprocessor replaces every one of them with the number 21. The compiler then reacts as if you actually had typed 21 rather than AGELIMIT, because the preprocessor changes all occurrences of AGELIMIT to 21 before your compiler reads the source code. But, again, the change is only temporary. After your program is compiled, you see it as you originally typed it, with #define and AGELIMIT still intact.

AGELIMIT is not a variable, because variables are declared and assigned values only at the time when your program is compiled and run. The preprocessor changes your source file before the time it is compiled.

The #define directive creates defined literals.

You might wonder why you would ever have to go to this much trouble. If you want 21 everywhere AGELIMIT occurs, you could type 21 to begin with! But the advantage of using #define rather than literals is that if the age limit ever changes (perhaps to 18), you have to change only one line in the program, not every single occurrence of the literal 21.

Because #define enables you easily to define and change literals, the replaced arguments of the #define directive are sometimes called *defined literals.* (C programmers say that #define "defines constants," but C++ programmers rarely use the word "constant" unless they are discussing the use of const.) You can define any type of literal, including string literals. The following program contains a defined string literal that replaces a string in two places.

```
// Filename: C6DEF1.CPP
// Defines a string literal and uses it twice.

#include <iostream.h>
#define MYNAME "Phil Ward"

main()
```

```
{
   char name[]=MYNAME;
   cout << "My name is " << name << "\n";    // Prints the array.
   cout << "My name is " << MYNAME << "\n"; // Prints the
                                            // defined literal.
   return 0;
}
```

The first argument of `#define` is in uppercase to distinguish it from variable names in the program. Variables are usually typed in lowercase. Although your preprocessor and compiler will not confuse the two, other users who look at your program can more quickly scan through and tell which items are defined literals and which are not. They will know when they see an uppercase word (if you follow the recommended standard for this first `#define` argument) to look at the top of the program for its actual defined value.

The fact that defined literals are not variables is even more clear in the following program. This program prints five values. Try to guess what those five values are before you look at the answer following the program.

```
// Filename: C6DEF2.CPP
// Illustrates that #define literals are not variables.

#include <iostream.h>

#define X1 b+c
#define X2 X1 + X1
#define X3 X2 * c + X1 - d
#define X4 2 * X1 + 3 * X2 + 4 * X3

main()
{
   int b = 2;     // Declares and initializes four variables.
   int c = 3;
   int d = 4;
   int e = X4;
   // Prints the values.
   cout << e << ", " << X1 << ", " << X2;
   cout << ", " << X3 << ", " << X4 << "\n";
   return 0;
}
```

The output from this program is

```
44   5   10   17   44
```

If you treated X1, X2, X3, and X4 as variables, you would not receive the correct answers. X1 through X4 are not variables; they are defined literals. Before your program is compiled, the preprocessor reads the first line and changes every occurrence of X1 to b+c. This occurs before the next #define is processed. Therefore, after the first #define, the source code looks like this:

```
// Filename: C6DEF2.CPP
// Illustrates that #define literals are not variables.

#include <iostream.h>

#define X2 b+c + b+c
#define X3 X2 * c + b+c - d
#define X4 2 * b+c + 3 * X2 + 4 * X3

main()
{
   int b=2;      // Declares and initializes four variables.
   int c=3;
   int d=4;
   int e=X4;

   // Prints the values.
   cout << e << ", " << b+c << ", " << X2;
   cout << ", " << X3 << ", " << X4 << "\n";
   return 0;
}
```

After the first #define finishes, the second one takes over and changes every occurrence of X2 to b+c + b+c. Your source code at that point becomes:

```
// Filename: C6DEF2.CPP
// Illustrates that #define literals are not variables.

#include <iostream.h>
```

```
#define X3 b+c + b+c * c + b+c - d
#define X4 2 * b+c + 3 * b+c + b+c + 4 * X3

main()
{
   int b=2;     // Declares and initializes four variables.
   int c=3;
   int d=4;
   int e=X4;

   // Prints the values.
   cout << e << ", " << b+c << ", " << b+c + b+c;
   cout << ", " << X3 << ", " << X4 << "\n";
   return 0;
}
```

After the second `#define` finishes, the third one takes over and changes every occurrence of `X3` to `b+c + b+c * c + b+c - d`. Your source code then becomes:

```
// Filename: C6DEF2.CPP
// Illustrates that #define literals are not variables.

#include <iostream.h>

#define X4 2 * b+c + 3 * b+c + b+c + 4 * b+c + b+c * c + b+c - d

main()
{
   int b=2;     // Declares and initializes four variables.
   int c=3;
   int d=4;
   int e=X4;

   // Prints the values.
   cout << e << ", " << b+c << ", " << b+c + b+c;
   cout << ", " << b+c + b+c * c + b+c - d
        << ", " << X4 << "\n";
   return 0;
}
```

The source code is growing rapidly! After the third `#define` finishes, the fourth and last one takes over and changes every occurrence of `X4` to `2 * b+c + 3 * b+c + b+c + 4 * b+c + b+c * c +b+c - d`. Your source code at this last point becomes:

```
// Filename: C6DEF2.CPP
// Illustrates that #define literals are not variables.

#include <iostream.h>

main()
{
   int b=2;      // Declares and initializes four variables.
   int c=3;
   int d=4;
   int e=2 * b+c + 3 * b+c + b+c + 4 * b+c + b+c * c + b+c - d;


   // Prints the values.
   cout << e << ", " << b+c << ", " << b+c + b+c;
   cout << ", " << b+c + b+c * c + b+c - d
        << ", " << 2 * b+c + 3 * b+c + b+c + 4 * b+c +
        b+c * c + b+c - d << "\n";
   return 0;
}
```

This is what your compiler actually reads. You did not type this complete listing; you typed the original listing (shown first). The preprocessor expanded your source code into this longer form, just as if you had typed it this way.

This is an extreme example, but it serves to illustrate how `#define` works on your source code and doesn't define any variables. The `#define` behaves like a word processor's search-and-replace command. Due to `#define`'s behavior, you can even rewrite the C++ language!

If you are used to BASIC, you might be more comfortable typing `PRINT` rather than C++'s `cout` when you want to print on-screen. If so, the following `#define` statement,

```
#define PRINT cout
```

enables you to print in C++ with these statements:

```
PRINT << "This is a new printing technique\n";
PRINT << "I could have used cout instead."\n;
```

This works because by the time your compiler reads the program, it reads only the following:

```
cout << "This is a new printing technique\n";
cout << "I could have used cout instead."\n;
```

In the next chapter, "Simple Input/Output," you learn about two functions sometimes used for input and output called `printf()` and `scanf()`. You can just as easily redefine function names using `#define` as you did with `cout`.

Also, remember that you cannot replace a defined literal if it resides in another string literal. For example, you cannot use the following `#define` statement:

```
#define AGE
```

to replace information in this `cout`:

```
cout << "AGE";
```

because `AGE` is a string literal, and it prints literally just as it appears inside the double quotation marks. The preprocessor can replace only defined literals that do not appear in quotation marks.

Do Not Overdo `#define`

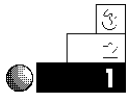
Many early C programmers enjoyed redefining parts of the language to suit whatever they were used to in another language. The `cout` to `PRINT` example was only one example of this. You can redefine virtually any C++ statement or function to "look" any way you like.

There is a danger to this, however, so be wary of using `#define` for this purpose. Your redefining the language becomes confusing to others who modify your program later. Also, as you become more familiar with C++, you will naturally use the true

C++ language more and more. When you are comfortable with C++, older programs that you redefined will be confusing—even to you!

If you are programming in C++, use the language conventions that C++ provides. Shy away from trying to redefine commands in the language. Think of the `#define` directive as a way to define numeric and string literals. If those literals ever change, you have to change only one line in your program. "Just say no" to any temptation to redefine commands and built-in functions. Better yet, modify any older C code that uses `#define`, and replace the `#define` preprocessor directive with the more useful `const` command.

Examples



1. Suppose you want to keep track of your company's target sales amount of $55,000.00. That target amount has not changed for the previous two years. Because it probably will not change soon (sales are flat), you decide to start using a defined literal to represent this target amount. Then, if target sales do change, you just have to change the amount on the `#define` line to:

   ```
   #define TARGETSALES 55000.00
   ```

   which defines a floating-point literal. You can then assign `TARGETSALES` to floating-point variables and print its value, just as if you had typed 55000.00 throughout your program, as these lines show:

   ```
   amt = TARGETSALES
   cout << TARGETSALES;
   ```

2. If you find yourself defining the same literals in many programs, file the literals on disk and include them. Then, you don't have to type your defined literals at the beginning

of every program. If you store these literals in a file called MYDEFS.C in your program's directory, you can include the file with the following `#include` statement:

```
#include "mydefs.c"
```

(To use angled brackets, you have to store the file in your system's include directory.)

3. Defined literals are appropriate for array sizes. For example, suppose you declare an array for a customer's name. When you write the program, you know you don't have a customer whose name is longer than 22 characters (including the null). Therefore, you can do this:

```
#define CNMLENGTH 22
```

When you define the array, you can use this:

```
char cust_name[CNMLENGTH]
```

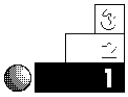
Other statements that need the array size also can use `CNMLENGTH`.

4. Many C++ programmers define a list of error messages. Once they define the messages with an easy-to-remember name, they can print those literals if an error occurs and still maintain consistency in their programs. The following error messages (or a similar form) often appear at the beginning of C++ programs.


```
#define DISKERR "Your disk drive seems not to be working"
#define PRNTERR "Your printer is not responding"
#define AGEERR  "You cannot enter an age that small"
#define NAMEERR "You must enter a full name"
```

Review Questions

The answers to the review questions are in Appendix B.



1. True or false: You can define variables with the preprocessor directives.

2. Which preprocessor directive merges another file into your program?
3. Which preprocessor directive defines literals throughout your program?

4. True or false: You can define character, string, integer, and floating-point literals with the `#define` directive.
5. Which happens first: your program is compiled or preprocessed?
6. What C++ keyword is used to replace the `#define` preprocessor directive?

7. When do you use the angled brackets in an `#include`, and when do you use double quotation marks?
8. Which are easier to change: defined literals or literals that you type throughout a program? Why?
9. Which header file should you include in almost every C++ program you write?
10. True or false: The `#define` in the following:

```
#define MESSAGE "Please press Enter to continue..."
```

changes this statement:

```
cout << "MESSAGE";
```

11. What is the output from the following program?

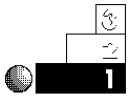
```
// Filename: C6EXER.C

#include <iostream.h>
#define AMT1 a+a+a
#define AMT2 AMT1 - AMT1

main()
{
   int a=1;
   cout << "Amount is " << AMT2 << "\n";
   return 0;
}
```

Even if you get this right, you will appreciate the side effects of `#define`. The `const` keyword (discussed in Chapter 4, "Variables and Literals") before a constant variable has none of the side effects that `#define` has.

Review Exercises



1. Write a program that prints your name to the screen. Use a defined literal for the name. Do not use a character array, and don't type your actual name inside the `cout`.

2. Suppose your boss wanted you to write a program that produced an "exception report." If the company's sales are less than $100,000.00 or more than $750,000.00, your boss wants your program to print the appropriate message. You learn how to produce these types of reports later in the book, but for now just write the `#define` statements that define these two floating-point literals.

3. Write the `cout` statements that print your name and birth date to the screen. Store these statements in their own file. Write a second program that includes the first file and prints your name and birth date. Be sure also to include <iostream.h>, because the included file contains `cout` statements.

4. Write a program that defines the ten digits, 0 through 9, as literals `ZERO` through `NINE`. Add these ten defined digits and print the result.

Summary

This chapter taught you the `#include` and `#define` preprocessor directives. Despite the fact that these directives are not executed, they temporarily change your source code by merging and defining literals into your program.

The next chapter, “Simple Input/Output,” explains input and output in more detail. There are ways to control precision when using `cin` and `cout`, as well as built-in functions that format input and output.