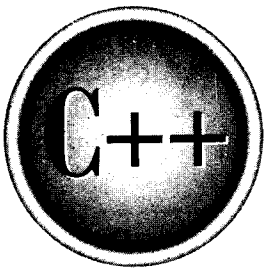
The Complete Reference



Chapter 10

The Preprocessor and Comments

You can include various instructions to the compiler in the source code of a C/C++ program. These are called *preprocessor directives*, and although not actually part of the C or C++ language per se, they expand the scope of the programming environment. This chapter also examines comments.

The Preprocessor

Before beginning, it is important to put the preprocessor in historical perspective. As it relates to C++, the preprocessor is largely a holdover from C. Moreover, the C++ preprocessor is virtually identical to the one defined by C. The main difference between C and C++ in this regard is the degree to which each relies upon the preprocessor. In C, each preprocessor directive is necessary. In C++, some features have been rendered redundant by newer and better C++ language elements. In fact, one of the long-term design goals of C++ is the elimination of the preprocessor altogether. But for now and well into the foreseeable future, the preprocessor will still be widely used.

The preprocessor contains the following directives:

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

As you can see, all preprocessor directives begin with a # sign. In addition, each preprocessing directive must be on its own line. For example,

```
#include <stdio.h>  #include <stdlib.h>
```

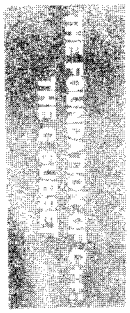
will not work.

#define

The **#define** directive defines an identifier and a character sequence (i.e., a set of characters) that will be substituted for the identifier each time it is encountered in the source file. The identifier is referred to as a *macro name* and the replacement process as *macro replacement*. The general form of the directive is

#define *macro-name char-sequence*

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



For example, if you wish to use the word **LEFT** for the value 1 and the word **RIGHT** for the value 0, you could declare these two **#define** directives:

```
#define LEFT 1
#define RIGHT 0
```

This causes the compiler to substitute a 1 or a 0 each time **LEFT** or **RIGHT** is encountered in your source file. For example, the following prints **0 1 2** on the screen:

```
printf("%d %d %d", RIGHT, LEFT, LEFT+1);
```

Once a macro name has been defined, it may be used as part of the definition of other macro names. For example, this code defines the values of **ONE**, **TWO**, and **THREE**:

```
#define ONE    1
#define TWO    ONE+ONE
#define THREE  ONE+TWO
```

Macro substitution is simply the replacement of an identifier by the character sequence associated with it. Therefore, if you wish to define a standard error message, you might write something like this:

```
#define E_MS "standard error on input\n"
/* ... */
printf(E_MS);
```

The compiler will actually substitute the string "standard error on input\n" when the identifier **E_MS** is encountered. To the compiler, the **printf( )** statement will actually appear to be

```
printf("standard error on input\n");
```

No text substitutions occur if the identifier is within a quoted string. For example,

```
#define XYZ this is a test

printf("XYZ");
```

does not print **this is a test**, but rather **XYZ**.

If the character sequence is longer than one line, you may continue it on the next by placing a backslash at the end of the line, as shown here:

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

C/C++ programmers commonly use uppercase letters for defined identifiers. This convention helps anyone reading the program know at a glance that a macro replacement will take place. Also, it is usually best to put all **#defines** at the start of the file or in a separate header file rather than sprinkling them throughout the program.

Macros are most frequently used to define names for "magic numbers" that occur in a program. For example, you may have a program that defines an array and has several routines that access that array. Instead of "hard-coding" the array's size with a constant, you can define the size using a **#define** statement and then use that macro name whenever the array size is needed. In this way, if you need to change the size of the array, you will only need to change the **#define** statement and then recompile your program. For example,

```
#define MAX_SIZE 100
/* ... */
float balance[MAX_SIZE];
/* ... */
for(i=0; i<MAX_SIZE; i++) printf("%f", balance[i]);
/* ... */
for(i=0; i<MAX_SIZE; i++) x =+ balance[i];
```

Since **MAX_SIZE** defines the size of the array **balance,** if the size of **balance** needs to be changed in the future, you need only change the definition of **MAX_SIZE.** All subsequent references to it will be automatically updated when you recompile your program.

Note *C++ provides a better way of defining constants, which uses the* ***const*** *keyword. This is described in Part Two.*

Defining Function-like Macros

The **#define** directive has another powerful feature: the macro name can have arguments. Each time the macro name is encountered, the arguments used in its definition are replaced by the actual arguments found in the program. This form of a macro is called a *function-like macro.* For example,

```
#include <stdio.h>

#define ABS(a)   (a)<0 ? -(a) : (a)

int main(void)
{
  printf("abs of -1 and 1: %d %d", ABS(-1), ABS(1));

  return 0;
}
```

When this program is compiled, **a** in the macro definition will be substituted with the values –1 and 1. The parentheses that enclose **a** ensure proper substitution in all cases. For example, if the parentheses around **a** were removed, this expression

```
ABS(10-20)
```

would be converted to

```
10-20<0 ? -10-20 : 10-20
```

after macro replacement and would yield the wrong result.

The use of a function-like macro in place of real functions has one major benefit: It increases the execution speed of the code because there is no function call overhead. However, if the size of the function-like macro is very large, this increased speed may be paid for with an increase in the size of the program because of duplicated code.

Note *Although parameterized macros are a valuable feature, C++ has a better way of creating inline code, which uses the* ***inline*** *keyword.*

#error

The **#error** directive forces the compiler to stop compilation. It is used primarily for debugging. The general form of the **#error** directive is

#error *error-message*

The *error-message* is not between double quotes. When the **#error** directive is encountered, the error message is displayed, possibly along with other information defined by the compiler.

#include

The **#include** directive instructs the compiler to read another source file in addition to the one that contains the **#include** directive. The name of the additional source file must be enclosed between double quotes or angle brackets. For example,

```
#include "stdio.h"
#include <stdio.h>
```

both instruct the compiler to read and compile the header for the C I/O system library functions.

Include files can have **#include** directives in them. This is referred to as *nested includes*. The number of levels of nesting allowed varies between compilers. However, Standard C stipulates that at least eight nested inclusions will be available. Standard C++ recommends that at least 256 levels of nesting be supported.

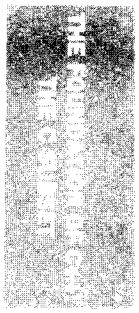
Whether the filename is enclosed by quotes or by angle brackets determines how the search for the specified file is conducted. If the filename is enclosed in angle brackets, the file is searched for in a manner defined by the creator of the compiler. Often, this means searching some special directory set aside for include files. If the filename is enclosed in quotes, the file is looked for in another implementation-defined manner. For many compilers, this means searching the current working directory. If the file is not found, the search is repeated as if the filename had been enclosed in angle brackets.

Typically, most programmers use angle brackets to include the standard header files. The use of quotes is generally reserved for including files specifically related to the program at hand. However, there is no hard and fast rule that demands this usage.

In addition to *files*, a C++ program can use the **#include** directive to include a C++ *header*. C++ defines a set of standard headers that provide the information necessary to the various C++ libraries. A header is a standard identifier that might, but need not, map to a filename. Thus, a header is simply an abstraction that guarantees that the appropriate information required by your program is included. Various issues associated with headers are described in Part Two.

Conditional Compilation Directives

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



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

Perhaps the most commonly used conditional compilation directives are the **#if**, **#else**, **#elif**, and **#endif**. These directives allow you to conditionally include portions of code based upon the outcome of a constant expression.

The general form of **#if** is

#if *constant-expression*
  statement sequence
#endif

If the constant expression following **#if** is true, the code that is between it and **#endif** is compiled. Otherwise, the intervening code is skipped. The **#endif** directive marks the end of an **#if** block. For example,

```
/* Simple #if example. */
#include <stdio.h>

#define MAX 100

int main(void)
{
#if MAX>99
  printf("Compiled for array greater than 99.\n");
#endif

  return 0;
}
```

This program displays the message on the screen because **MAX** is greater than 99. This example illustrates an important point. The expression that follows the **#if** is evaluated at compile time. Therefore, it must contain only previously defined identifiers and constants—no variables may be used.

The **#else** directive works much like the **else** that is part of the C++ language: it establishes an alternative if **#if** fails. The previous example can be expanded as shown here:

```
/* Simple #if/#else example. */
#include <stdio.h>
```

```
#define MAX 10

int main(void)
{
#if MAX>99
  printf("Compiled for array greater than 99.\n");
#else
  printf("Compiled for small array.\n");
#endif

  return 0;
}
```

In this case, **MAX** is defined to be less than 99, so the **#if** portion of the code is not compiled. The **#else** alternative is compiled, however, and the message **Compiled for small array** is displayed.

Notice that **#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** directive means "else if" and establishes an if-else-if chain for multiple compilation options. **#elif** is followed by a constant expression. If the expression is true, that block of code is compiled and no other **#elif** expressions are tested. Otherwise, the next block in the series is checked. The general form for **#elif** is

```
#if expression
  statement sequence
#elif expression 1
  statement sequence
#elif expression 2
  statement sequence
#elif expression 3
  statement sequence
#elif expression 4
.
.
.
#elif expression N
  statement sequence
#endif
```

For example, the following fragment uses the value of **ACTIVE_COUNTRY** to define the currency sign:

```
#define US 0
#define ENGLAND 1
#define JAPAN 2

#define ACTIVE_COUNTRY US

#if ACTIVE_COUNTRY == US
  char currency[] = "dollar";
#elif ACTIVE_COUNTRY == ENGLAND
  char currency[] = "pound";
#else
  char currency[] = "yen";
#endif
```

Standard C states that **#if**s and **#elif**s may be nested at least eight levels. Standard C++ suggests that at least 256 levels of nesting be allowed. When nested, each **#endif**, **#else**, or **#elif** associates with the nearest **#if** or **#elif**. For example, the following is perfectly valid:

```
#if MAX>100
  #if SERIAL_VERSION
    int port=198;
  #elif
    int port=200;
  #endif
#else
  char out_buffer[100];
#endif
```

#ifdef and #ifndef

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

#ifdef *macro-name*
 statement sequence
#endif

If *macro-name* has been previously defined in a **#define** statement, the block of code 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, the block of code is compiled.

Both **#ifdef** and **#ifndef** may use an **#else** or **#elif** statement. For example,

```
#include <stdio.h>

#define TED 10

int main(void)
{
#ifdef TED
  printf("Hi Ted\n");
#else
  printf("Hi anyone\n");
#endif
#ifndef RALPH
  printf("RALPH not defined\n");
#endif

  return 0;
}
```

will print **Hi Ted** and **RALPH not defined**. However, if **TED** were not defined, **Hi anyone** would be displayed, followed by **RALPH not defined**.

You may nest **#ifdef**s and **#ifndef**s to at least eight levels in Standard C. Standard C++ suggests that at least 256 levels of nesting be supported.

#undef

The **#undef** directive removes a previously defined definition of the macro name that follows it. That is, it "undefines" a macro. The general form for **#undef** is

```
#undef macro-name
```

For example,

```
#define LEN 100
#define WIDTH 100

char array[LEN][WIDTH];

#undef LEN
#undef WIDTH
/* at this point both LEN and WIDTH are undefined */
```

Both **LEN** and **WIDTH** are defined until the **#undef** statements are encountered.

#undef is used principally 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. The **defined** operator has this general form:

defined *macro-name*

If *macro-name* is currently defined, then the expression is true. Otherwise, it is false. 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 may 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
  printf("Final version!\n");
#endif
```

One reason for using **defined** is that it allows the existence of a macro name to be determined by a **#elif** statement.

#line

The **#line** directive changes the contents of __LINE__ and __FILE__, which are predefined identifiers in the compiler. The __LINE__ identifier contains the line number of the currently compiled line of code. The __FILE__ identifier is a string that contains the name of the source file being compiled. The general form for **#line** is

#line *number "filename"*

where *number* is any positive integer and becomes the new value of __LINE__, and the optional *filename* is any valid file identifier, which becomes the new value of __FILE__. **#line** is primarily used for debugging and special applications.

For example, the following code specifies that the line count will begin with 100. The **printf( )** statement displays the number 102 because it is the third line in the program after the **#line 100** statement.

```
#include <stdio.h>

#line 100                    /* reset the line counter */
int main(void)               /* line 100 */
{                            /* line 101 */
  printf("%d\n",__LINE__); /* line 102 */

  return 0;
}
```

#pragma

#pragma is an implementation-defined directive that allows various instructions to be given to the compiler. For example, a compiler may have an option that supports program execution tracing. A trace option would then be specified by a **#pragma** statement. You must check the compiler's documentation for details and options.

The # and ## Preprocessor Operators

There are two preprocessor operators: # and ##. These operators are used with the **#define** statement.

The # operator, which is generally called the *stringize* operator, turns the argument it precedes into a quoted string. For example, consider this program.

```
#include <stdio.h>

#define mkstr(s)  # s

int main(void)
{
  printf(mkstr(I like C++));

  return 0;
}
```

The preprocessor turns the line

```
printf(mkstr(I like C++));
```

into

```
printf("I like C++");
```

The ## operator, called the *pasting* operator, concatenates two tokens. For example,

```
#include <stdio.h>

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

int main(void)
{
  int xy = 10;

  printf("%d", concat(x, y));

  return 0;
}
```

The preprocessor transforms

```
printf("%d", concat(x, y));
```

into

```
printf("%d", 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 the preprocessor to handle some special cases.

Predefined Macro Names

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

__LINE__
__FILE__
__DATE__
__TIME__
__STDC__
__cplusplus

The C language defines the first five of these. Each will be described here, in turn.

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, the compiler will accept only standard C/C++ code that does not contain any nonstandard extensions.

A compiler conforming to Standard C++ will define__**cplusplus** as a value containing at least six digits. Nonconforming compilers will use a value with five or less digits.

Comments

C89 defines only one style of comment, which begins with the character pair /* and ends with */. There must be no spaces between the asterisk and the slash. The compiler

ignores any text between the beginning and ending comment symbols. For example, this program prints only **hello** on the screen:

```
#include <stdio.h>

int main(void)
{
  printf("hello");
  /* printf("there"); */

  return 0;
}
```

This style of comment is commonly called a *multiline comment* because the text of the comment may extend over two or more lines. For example,

```
/* this is a
multi-line
comment */
```

Comments may be placed anywhere in a program, as long as they do not appear in the middle of a keyword or identifier. For example, this comment is valid:

```
x = 10+ /* add the numbers */5;
```

while

```
swi/*this will not work*/tch(c) { ...
```

is incorrect because a keyword cannot contain a comment. However, you should not generally place comments in the middle of expressions because it obscures their meaning.

Multiline comments may not be nested. That is, one comment may not contain another comment. For example, this code fragment causes a compile-time error:

```
/* this is an outer comment
  x = y/a;
  /* this is an inner comment - and causes an error */
*/
```

Single-Line Comments

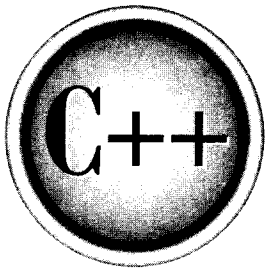
C++ (and C99) supports two types of comments. The first is the multiline comment. The second is the *single-line comment*. Single-line comments begin with a // and end at the end of the line. For example,

```
// this is a single-line comment
```

Single line comments are especially useful when short, line-by-line descriptions are needed. Although they are not technically supported by C89, many C compilers will accept them anyway, and single-line comments were added to C by C99. One last point: a single-line comment can be nested within a multiline comment.

You should include comments whenever they are needed to explain the operation of the code. All but the most obvious functions should have a comment at the top that states what the function does, how it is called, and what it returns.

The Complete Reference



Part II

C++

Part One examined the C subset of C++. Part Two describes those features of the language specific to C++. That is, it discusses those features of C++ that it does not have in common with C. Because many of the C++ features are designed to support object-oriented programming (OOP), Part Two also provides a discussion of its theory and merits. We will begin with an overview of C++.