E Preprocessor

Hold thou the good; define it well.
—Alfred, Lord Tennyson

I have found you an argument; but I am not obliged to find you an understanding.
—Samuel Johnson

A good symbol is the best argument, and is a missionary to persuade thousands.
—Ralph Waldo Emerson

Conditions are fundamentally sound.
—Herbert Hoover [December 1929]

Objectives

In this appendix you'll learn:

- To use `#include` for developing large programs.
- To use `#define` to create macros and macros with arguments.
- To understand conditional compilation.
- To display error messages during conditional compilation.
- To use assertions to test if the values of expressions are correct.

Outline

E.1 Introduction

This chapter introduces the **preprocessor**. Preprocessing occurs before a program is compiled. Some possible actions are inclusion of other files in the file being compiled, definition of **symbolic constants** and **macros**, **conditional compilation** of program code and **conditional execution of preprocessor directives.** All preprocessor directives begin with #, and only white-space characters may appear before a preprocessor directive on a line. Preprocessor directives are not C++ statements, so they do not end in a semicolon (;). Preprocessor directives are processed fully before compilation begins.

Common Programming Error E.1

Placing a semicolon at the end of a preprocessor directive can lead to a variety of errors, depending on the type of preprocessor directive.

Software Engineering Observation E.1

Many preprocessor features (especially macros) are more appropriate for C programmers than for C++ programmers. C++ programmers should familiarize themselves with the preprocessor, because they might need to work with C legacy code.

E.2 `#include` Preprocessor Directive

The **`#include` preprocessor directive** has been used throughout this text. The `#include` directive causes a copy of a specified file to be included in place of the directive. The two forms of the `#include` directive are

```
#include <filename>
#include "filename"
```

The difference between these is the location the preprocessor searches for the file to be included. If the filename is enclosed in angle brackets (< and >)—used for standard library header files—the preprocessor searches for the specified file in an implementation-dependent manner, normally through predesignated directories. If the file name is enclosed in quotes, the preprocessor searches first in the same directory as the file being compiled, then in the same implementation-dependent manner as for a file name enclosed in angle brackets. This method is normally used to include programmer-defined header files.

The `#include` directive is used to include standard header files such as `<iostream>` and `<iomanip>`. The `#include` directive is also used with programs consisting of several source files that are to be compiled together. A header file containing declarations and definitions common to the separate program files is often created and included in the file. Examples of such declarations and definitions are classes, structures, unions, enumerations, function prototypes, constants and stream objects (e.g., `cin`).

E.3 #define Preprocessor Directive: Symbolic Constants

The **#define preprocessor directive** creates **symbolic constants**—constants represented as symbols—and macros—operations defined as symbols. The `#define` preprocessor directive format is

```
#define  identifier  replacement-text
```

When this line appears in a file, all subsequent occurrences (except those inside a string) of *identifier* in that file will be replaced by *replacement-text* before the program is compiled. For example,

```
#define PI 3.14159
```

replaces all subsequent occurrences of the symbolic constant `PI` with the numeric constant `3.14159`. Symbolic constants enable you to create a name for a constant and use the name throughout the program. Later, if the constant needs to be modified throughout the program, it can be modified once in the `#define` preprocessor directive—and when the program is recompiled, all occurrences of the constant in the program will be modified. [*Note:* Everything to the right of the symbolic constant name replaces the symbolic constant. For example, `#define PI = 3.14159` causes the preprocessor to replace every occurrence of `PI` with `= 3.14159`. Such replacement is the cause of many subtle logic and syntax errors.] Redefining a symbolic constant with a new value without first undefining it is also an error. Note that `const` variables in C++ are preferred over symbolic constants. Constant variables have a specific data type and are visible by name to a debugger. Once a symbolic constant is replaced with its replacement text, only the replacement text is visible to a debugger. A disadvantage of `const` variables is that they might require a memory location of their data type size—symbolic constants do not require any additional memory.

Common Programming Error E.2

Using symbolic constants in a file other than the file in which the symbolic constants are defined is a compilation error (unless they are `#included` from a header file).

Good Programming Practice E.1

Using meaningful names for symbolic constants makes programs more self-documenting.

E.4 #define Preprocessor Directive: Macros

[*Note:* This section is included for the benefit of C++ programmers who will need to work with C legacy code. In C++, macros can often be replaced by templates and inline functions.] A macro is an operation defined in a `#define` preprocessor directive. As with symbolic constants, the ***macro-identifier*** is replaced with the ***replacement-text*** before the program is compiled. Macros may be defined with or without ***arguments***. A macro without arguments is processed like a symbolic constant. In a macro with arguments, the arguments are substituted in the *replacement-text*, then the macro is expanded—i.e., the *replacement-text* replaces the macro-identifier and argument list in the program. There is no data type checking for macro arguments. A macro is used simply for text substitution.

Consider the following macro definition with one argument for the area of a circle:

```
#define CIRCLE_AREA( x ) ( PI * ( x ) * ( x ) )
```

Wherever `CIRCLE_AREA( y )` appears in the file, the value of `y` is substituted for `x` in the replacement text, the symbolic constant `PI` is replaced by its value (defined previously) and the macro is expanded in the program. For example, the statement

```
area = CIRCLE_AREA( 4 );
```

is expanded to

```
area = ( 3.14159 * ( 4 ) * ( 4 ) );
```

Because the expression consists only of constants, at compile time the value of the expression can be evaluated, and the result is assigned to `area` at runtime. The parentheses around each `x` in the replacement text and around the entire expression force the proper order of evaluation when the macro argument is an expression. For example, the statement

```
area = CIRCLE_AREA( c + 2 );
```

is expanded to

```
area = ( 3.14159 * ( c + 2 ) * ( c + 2 ) );
```

which evaluates correctly, because the parentheses force the proper order of evaluation. If the parentheses are omitted, the macro expansion is

```
area = 3.14159 * c + 2 * c + 2;
```

which evaluates incorrectly as

```
area = ( 3.14159 * c ) + ( 2 * c ) + 2;
```

because of the rules of operator precedence.

Common Programming Error E.3

Forgetting to enclose macro arguments in parentheses in the replacement text is an error.

Macro `CIRCLE_AREA` could be defined as a function. Function `circleArea`, as in

```
double circleArea( double x ) { return 3.14159 * x * x; }
```

performs the same calculation as `CIRCLE_AREA`, but the overhead of a function call is associated with function `circleArea`. The advantages of `CIRCLE_AREA` are that macros insert code directly in the program—avoiding function overhead—and the program remains readable because `CIRCLE_AREA` is defined separately and named meaningfully. A disadvantage is that its argument is evaluated twice. Also, every time a macro appears in a program, the macro is expanded. If the macro is large, this produces an increase in program size. Thus, there is a trade-off between execution speed and program size (if disk space is low).

Note that `inline` functions (see Chapter 6) are preferred to obtain the performance of macros and the software engineering benefits of functions.

Performance Tip E.1

Macros can sometimes be used to replace a function call with `inline` code prior to execution time. This eliminates the overhead of a function call. Inline functions are preferable to macros because they offer the type-checking services of functions.

The following is a macro definition with two arguments for the area of a rectangle:

```
#define RECTANGLE_AREA( x, y )  ( ( x ) * ( y ) )
```

Wherever `RECTANGLE_AREA( a, b )` appears in the program, the values of `a` and `b` are substituted in the macro replacement text, and the macro is expanded in place of the macro name. For example, the statement

```
rectArea = RECTANGLE_AREA( a + 4, b + 7 );
```

is expanded to

```
rectArea = ( ( a + 4 ) * ( b + 7 ) );
```

The value of the expression is evaluated and assigned to variable `rectArea`.

The replacement text for a macro or symbolic constant is normally any text on the line after the identifier in the `#define` directive. If the replacement text for a macro or symbolic constant is longer than the remainder of the line, a backslash (\) must be placed at the end of each line of the macro (except the last line), indicating that the replacement text continues on the next line.

Symbolic constants and macros can be discarded using the **`#undef` preprocessor directive**. Directive `#undef` “undefines” a symbolic constant or macro name. The scope of a symbolic constant or macro is from its definition until it is either undefined with `#undef` or the end of the file is reached. Once undefined, a name can be redefined with `#define`.

Note that expressions with side effects (e.g., variable values are modified) should not be passed to a macro, because macro arguments may be evaluated more than once.

Common Programming Error E.4

Macros often replace a name that wasn’t intended to be a use of the macro but just happened to be spelled the same. This can lead to exceptionally mysterious compilation and syntax errors.

E.5 Conditional Compilation

Conditional compilation enables you to control the execution of preprocessor directives and the compilation of program code. Each of the conditional preprocessor directives evaluates a constant integer expression that will determine whether the code will be compiled. Cast expressions, `sizeof` expressions and enumeration constants cannot be evaluated in preprocessor directives because these are all determined by the compiler and preprocessing happens before compilation.

The conditional preprocessor construct is much like the `if` selection structure. Consider the following preprocessor code:

```
#ifndef NULL
   #define NULL 0
#endif
```

which determines whether the symbolic constant `NULL` is already defined. The expression `#ifndef NULL` includes the code up to `#endif` if `NULL` is not defined, and skips the code if `NULL` is defined. Every **`#if`** construct ends with **`#endif`**. Directives **`#ifdef`** and **`#ifndef`** are shorthand for `#if defined(`*name*`)` and `#if !defined(`*name*`)`. A multiple-part conditional preprocessor construct may be tested using the `#elif` (the equivalent of `else if` in an `if` structure) and the `#else` (the equivalent of `else` in an `if` structure) directives.

During program development, programmers often find it helpful to "comment out" large portions of code to prevent it from being compiled. If the code contains C-style comments, `/*` and `*/` cannot be used to accomplish this task, because the first `*/` encountered would terminate the comment. Instead, you can use the following preprocessor construct:

```
#if 0
   code prevented from compiling
#endif
```

To enable the code to be compiled, simply replace the value `0` in the preceding construct with the value `1`.

Conditional compilation is commonly used as a debugging aid. Output statements are often used to print variable values and to confirm the flow of control. These output statements can be enclosed in conditional preprocessor directives so that the statements are compiled only until the debugging process is completed. For example,

```
#ifdef DEBUG
   cerr << "Variable x = " << x << endl;
#endif
```

causes the `cerr` statement to be compiled in the program if the symbolic constant `DEBUG` has been defined before directive `#ifdef DEBUG`. This symbolic constant is normally set by a command-line compiler or by settings in the IDE (e.g., Visual Studio) and not by an explicit `#define` definition. When debugging is completed, the `#define` directive is removed from the source file, and the output statements inserted for debugging purposes are ignored during compilation. In larger programs, it might be desirable to define several different symbolic constants that control the conditional compilation in separate sections of the source file.

Common Programming Error E.5

Inserting conditionally compiled output statements for debugging purposes in locations where C++ currently expects a single statement can lead to syntax errors and logic errors. In this case, the conditionally compiled statement should be enclosed in a compound statement. Thus, when the program is compiled with debugging statements, the flow of control of the program is not altered.

E.6 #error and #pragma Preprocessor Directives

The **`#error` directive**

```
#error tokens
```

prints an implementation-dependent message including the *tokens* specified in the directive. The tokens are sequences of characters separated by spaces. For example,

```
    #error 1 - Out of range error
```

contains six tokens. In one popular C++ compiler, for example, when a `#error` directive is processed, the tokens in the directive are displayed as an error message, preprocessing stops and the program does not compile.

The **`#pragma` directive**

```
    #pragma tokens
```

causes an implementation-defined action. A pragma not recognized by the implementation is ignored. A particular C++ compiler, for example, might recognize pragmas that enable you to take advantage of that compiler's specific capabilities. For more information on `#error` and `#pragma`, see the documentation for your C++ implementation.

E.7 Operators # and

The **`#`** and **`##`** preprocessor operators are available in C++ and ANSI/ISO C. The `#` operator causes a replacement-text token to be converted to a string surrounded by quotes. Consider the following macro definition:

```
    #define HELLO( x ) cout << "Hello, " #x << endl;
```

When `HELLO(John)` appears in a program file, it is expanded to

```
    cout << "Hello, " "John" << endl;
```

The string `"John"` replaces `#x` in the replacement text. Strings separated by white space are concatenated during preprocessing, so the above statement is equivalent to

```
    cout << "Hello, John" << endl;
```

Note that the `#` operator must be used in a macro with arguments, because the operand of `#` refers to an argument of the macro.

The `##` operator concatenates two tokens. Consider the following macro definition:

```
    cout << "Hello, John" << endl;
    #define TOKENCONCAT( x, y )  x ## y
```

When `TOKENCONCAT` appears in the program, its arguments are concatenated and used to replace the macro. For example, `TOKENCONCAT(O, K)` is replaced by `OK` in the program. The `##` operator must have two operands.

E.8 Predefined Symbolic Constants

There are six **predefined symbolic constants** (Fig. E.1). The identifiers for each of these begin and (except for `__cplusplus`) end with *two* underscores. These identifiers and preprocessor operator `defined` (Section E.5) cannot be used in `#define` or `#undef` directives.

Symbolic constant	Description
`__LINE__`	The line number of the current source-code line (an integer constant).
`__FILE__`	The presumed name of the source file (a string).
`__DATE__`	The date the source file is compiled (a string of the form `"Mmm dd yyyy"` such as `"Aug 19 2002"`).
`__STDC__`	Indicates whether the program conforms to the ANSI/ISO C standard. Contains value 1 if there is full conformance and is undefined otherwise.
`__TIME__`	The time the source file is compiled (a string literal of the form `"hh:mm:ss"`).
`__cplusplus`	Contains the value `199711L` (the date the ISO C++ standard was approved) if the file is being compiled by a C++ compiler, undefined otherwise. Allows a file to be set up to be compiled as either C or C++.

Fig. E.1 | The predefined symbolic constants.

E.9 Assertions

The **`assert` macro**—defined in the **`<cassert>`** header file—tests the value of an expression. If the value of the expression is `0` (false), then `assert` prints an error message and calls function **`abort`** (of the general utilities library—`<cstdlib>`) to terminate program execution. This is a useful debugging tool for testing whether a variable has a correct value. For example, suppose variable `x` should never be larger than `10` in a program. An assertion may be used to test the value of `x` and print an error message if the value of `x` is incorrect. The statement would be

```
assert( x <= 10 );
```

If `x` is greater than `10` when the preceding statement is encountered in a program, an error message containing the line number and file name is printed, and the program terminates. You may then concentrate on this area of the code to find the error. If the symbolic constant `NDEBUG` is defined, subsequent assertions will be ignored. Thus, when assertions are no longer needed (i.e., when debugging is complete), we insert the line

```
#define NDEBUG
```

in the program file rather than deleting each assertion manually. As with the `DEBUG` symbolic constant, `NDEBUG` is often set by compiler command-line options or through a setting in the IDE.

Most C++ compilers now include exception handling. C++ programmers prefer using exceptions rather than assertions. But assertions are still valuable for C++ programmers who work with C legacy code.

E.10 Wrap-Up

This appendix discussed the `#include` directive, which is used to develop larger programs. You also learned about the `#define` directive, which is used to create macros. We introduced conditional compilation, displaying error messages and using assertions.

Summary

Section E.2 `#include` Preprocessor Directive

- All preprocessor directives begin with `#` and are processed before the program is compiled.
- Only white-space characters may appear before a preprocessor directive on a line.
- The `#include` directive includes a copy of the specified file. If the filename is enclosed in quotes, the preprocessor begins searching in the same directory as the file being compiled for the file to be included. If the filename is enclosed in angle brackets (`<` and `>`), the search is performed in an implementation-defined manner.

Section E.3 `#define` Preprocessor Directive: Symbolic Constants

- The `#define` preprocessor directive is used to create symbolic constants and macros.
- A symbolic constant is a name for a constant.

Section E.4 `#define` Preprocessor Directive: Macros

- A macro is an operation defined in a `#define` preprocessor directive. Macros may be defined with or without arguments.
- The replacement text for a macro or symbolic constant is any text remaining on the line after the identifier (and, if any, the macro argument list) in the `#define` directive. If the replacement text for a macro or symbolic constant is too long to fit on one line, a backslash (\) is placed at the end of the line, indicating that the replacement text continues on the next line.
- Symbolic constants and macros can be discarded using the `#undef` preprocessor directive. Directive `#undef` "undefines" the symbolic constant or macro name.
- The scope of a symbolic constant or macro is from its definition until it is either undefined with `#undef` or the end of the file is reached.

Section E.5 Conditional Compilation

- Conditional compilation enables you to control the execution of preprocessor directives and the compilation of program code.
- The conditional preprocessor directives evaluate constant integer expressions. Cast expressions, `sizeof` expressions and enumeration constants cannot be evaluated in preprocessor directives.
- Every `#if` construct ends with `#endif`.
- Directives `#ifdef` and `#ifndef` are provided as shorthand for `#if defined(`*name*`)` and `#if !defined(`*name*`)`.
- A multiple-part conditional preprocessor construct is tested with directives `#elif` and `#else`.

Section E.6 `#error` and `#pragma` Preprocessor Directives

- The `#error` directive prints an implementation-dependent message that includes the tokens specified in the directive and terminates preprocessing and compiling.
- The `#pragma` directive causes an implementation-defined action. If the pragma is not recognized by the implementation, the pragma is ignored.

Section E.7 Operators # and ##

- The `#` operator causes the following replacement text token to be converted to a string surrounded by quotes. The `#` operator must be used in a macro with arguments, because the operand of `#` must be an argument of the macro.
- The `##` operator concatenates two tokens. The `##` operator must have two operands.

Section E.8 Predefined Symbolic Constants

- There are six predefined symbolic constants. Constant `__LINE__` is the line number of the current source-code line (an integer). Constant `__FILE__` is the presumed name of the file (a string). Constant `__DATE__` is the date the source file is compiled (a string). Constant `__TIME__` is the time the source file is compiled (a string). Note that each of the predefined symbolic constants begins (and, with the exception of `__cplusplus`, ends) with two underscores.

Section E.9 Assertions

- The `assert` macro—defined in the `<cassert>` header file—tests the value of an expression. If the value of the expression is `0` (false), then `assert` prints an error message and calls function `abort` to terminate program execution.

Self-Review Exercises

E.1 Fill in the blanks in each of the following:

a) Every preprocessor directive must begin with ________.
b) The conditional compilation construct may be extended to test for multiple cases by using the ________ and the ________ directives.
c) The ________ directive creates macros and symbolic constants.
d) Only ________ characters may appear before a preprocessor directive on a line.
e) The ________ directive discards symbolic constant and macro names.
f) The ________ and ________ directives are provided as shorthand notation for `#if defined(`*name*`)` and `#if !defined(`*name*`)`.
g) ________ enables you to control the execution of preprocessor directives and the compilation of program code.
h) The ________ macro prints a message and terminates program execution if the value of the expression the macro evaluates is `0`.
i) The ________ directive inserts a file in another file.
j) The ________ operator concatenates its two arguments.
k) The ________ operator converts its operand to a string.
l) The character ________ indicates that the replacement text for a symbolic constant or macro continues on the next line.

E.2 Write a program to print the values of the predefined symbolic constants `__LINE__`, `__FILE__`, `__DATE__` and `__TIME__` listed in Fig. E.1.

E.3 Write a preprocessor directive to accomplish each of the following:

a) Define symbolic constant `YES` to have the value `1`.
b) Define symbolic constant `NO` to have the value `0`.
c) Include the header file `common.h`. The header is found in the same directory as the file being compiled.
d) If symbolic constant `TRUE` is defined, undefine it, and redefine it as `1`. Do not use `#ifdef`.
e) If symbolic constant `TRUE` is defined, undefine it, and redefine it as `1`. Use the `#ifdef` preprocessor directive.
f) If symbolic constant `ACTIVE` is not equal to `0`, define symbolic constant `INACTIVE` as `0`. Otherwise, define `INACTIVE` as `1`.
g) Define macro `CUBE_VOLUME` that computes the volume of a cube (takes one argument).

Answers to Self-Review Exercises

E.1 a) `#`. b) `#elif`, `#else`. c) `#define`. d) white-space. e) `#undef`. f) `#ifdef`, `#ifndef`. g) Conditional compilation. h) `assert`. i) `#include`. j) `##`. k) `#`. l) `\`.

E.2 (See below.)

```
// exF_02.cpp
// Self-Review Exercise E.2 solution.
#include <iostream>
using namespace std;

int main()
{
   cout << "__LINE__ = " << __LINE__ << endl
        << "__FILE__ = " << __FILE__ << endl
        << "__DATE__ = " << __DATE__ << endl
        << "__TIME__ = " << __TIME__ << endl
        << "__cplusplus = " << __cplusplus << endl;
}  // end main
```

```
__LINE__ = 9
__FILE__ = c:\cpp4e\ch19\ex19_02.CPP
__DATE__ = Jul 17 2002
__TIME__ = 09:55:58
__cplusplus = 199711L
```

E.3
a) `#define YES 1`
b) `#define NO 0`
c) `#include "common.h"`
d)
```
#if defined(TRUE)
   #undef TRUE
   #define TRUE 1
#endif
```
e)
```
#ifdef TRUE
   #undef TRUE
   #define TRUE 1
#endif
```
f)
```
#if ACTIVE
   #define INACTIVE 0
#else
   #define INACTIVE 1
#endif
```
g) `#define CUBE_VOLUME( x )  ( ( x ) * ( x ) * ( x ) )`

Exercises

E.4 Write a program that defines a macro with one argument to compute the volume of a sphere. The program should compute the volume for spheres of radii from 1 to 10 and print the results in tabular format. The formula for the volume of a sphere is

$$(4.0 / 3) * \pi * r^3$$

where π is 3.14159.

E.5 Write a program that produces the following output:

```
The sum of x and y is 13
```

The program should define macro `SUM` with two arguments, `x` and `y`, and use `SUM` to produce the output.

E.6 Write a program that uses macro `MINIMUM2` to determine the smaller of two numeric values. Input the values from the keyboard.

E.7 Write a program that uses macro `MINIMUM3` to determine the smallest of three numeric values. Macro `MINIMUM3` should use macro `MINIMUM2` defined in Exercise E.6 to determine the smallest number. Input the values from the keyboard.

E.8 Write a program that uses macro `PRINT` to print a string value.

E.9 Write a program that uses macro `PRINTARRAY` to print an array of integers. The macro should receive the array and the number of elements in the array as arguments.

E.10 Write a program that uses macro `SUMARRAY` to sum the values in a numeric array. The macro should receive the array and the number of elements in the array as arguments.

E.11 Rewrite the solutions to Exercises E.4–E.10 as `inline` functions.

E.12 For each of the following macros, identify the possible problems (if any) when the preprocessor expands the macros:

a) `#define SQR( x ) x * x`
b) `#define SQR( x ) ( x * x )`
c) `#define SQR( x ) ( x ) * ( x )`
d) `#define SQR( x ) ( ( x ) * ( x ) )`