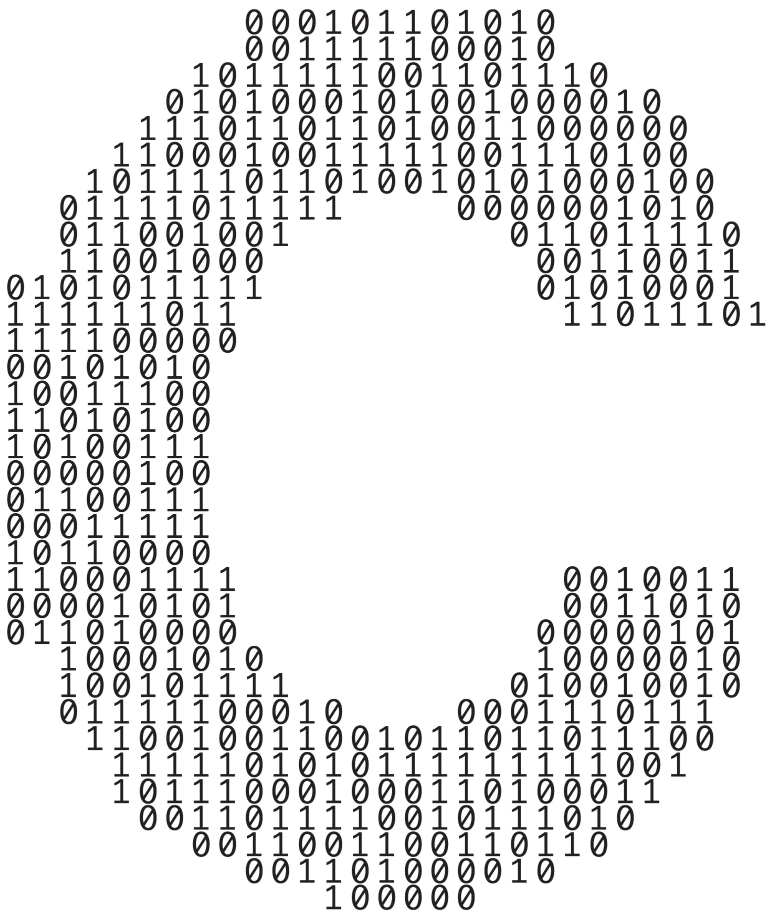
Preprocessor

14



Objectives

In this chapter, you'll:

- Use `#include` to help manage files in large programs.
- Use `#define` to create macros with and without arguments.
- Use conditional compilation to specify portions of a program that should not always be compiled, such as code that assists you in debugging.
- Display error messages during conditional compilation.
- Use assertions to test whether the expression values are correct.

Outline

14.1 Introduction

The C preprocessor executes *before* each program compiles. It:

- includes other files into the file being compiled,
- defines **symbolic constants** and **macros**,
- **conditionally compiles** program code and
- **conditionally executes preprocessor directives**.

Preprocessor directives begin with `#`. Only whitespace characters and comments delimited by `/*` and `*/` may appear before a preprocessor directive on a line.

C has perhaps the largest installed base of "legacy code" of any modern programming language. It's been in use for about five decades. As a professional C programmer, you're likely to encounter code written many years ago using older programming techniques. This chapter presents several of those techniques and recommends some newer techniques that can replace them.

✓ Self Check

1 *(Multiple Choice)* Which of the following are actions performed by the preprocessor?

a) The inclusion of other files into the file being compiled.
b) Definition of symbolic constants and macros.
c) Conditional compilation of program code and conditional execution of preprocessor directives.
d) All of the above.

Answer: d.

2 *(True/False)* C has perhaps the largest installed base of "legacy code" of any modern programming language. It's been in active use for about five decades.
Answer: *True.*

14.2 #include Preprocessor Directive

You've used the **#include preprocessor directive** throughout this book. When the preprocessor encounters a `#include`, it replaces the directive with a copy of the specified file. The two forms of the `#include` directive are:

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

The difference between these is the location where the preprocessor begins searching for the file. For filenames enclosed in angle brackets (< and >)—such as **standard library headers**—the preprocessor searches in *implementation-dependent* compiler and system folders. You typically use filenames enclosed in quotes ("") to include headers that you define for use with your program. In this case, the preprocessor begins searching in the *same* folder as the file in which the `#include` directive appears. If the compiler cannot find the specified file in the current folder, it searches the implementation-dependent compiler and system folders.

In addition to using `#include` for standard library headers, you'll frequently use it in programs consisting of *multiple source files*. You'll often create headers for a program's common declarations, then include that file into multiple source files. Examples of such declarations are:

- `struct` and `union` declarations,
- `typedef`s,
- `enum`s, and
- function prototypes.

✓ Self Check

1 *(Fill-In)* The ________ preprocessor directive causes a copy of a specified file to be included in place of the directive.
Answer: `#include`.

2 *(Fill-In)* If the filename in an `#include` directive is enclosed in quotes, the preprocessor begins its search for the file in ________.
Answer: the same directory as the file being compiled.

14.3 #define Preprocessor Directive: Symbolic Constants

The **#define directive** creates:

- *symbolic constants*—constants represented as identifiers, and
- **macros**—operations defined as symbols.

The `#define` directive's format is

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

By convention, a symbolic constant's *identifier* should contain only uppercase letters and underscores. Using meaningful names for symbolic constants helps make programs self-documenting.

Replacing Symbolic Constants

When the preprocessor encounters a `#define` directive, it replaces *identifier* with *replacement text* throughout that source file, ignoring any occurrences of *identifier* in string literals or comments. For example,

```
#define PI 3.14159
```

replaces all subsequent occurrences of the symbolic constant `PI` with `3.14159`. Symbolic constants enable you to create named constants and use their names throughout the program.

Common Error with Symbolic Constants

Everything to the right of a symbolic constant's name replaces the symbolic constant. For example,

```
#define PI = 3.14159
```

causes the preprocessor to replace *every* occurrence of `PI` with `"= 3.14159"`. Incorrect `#define` directives cause many subtle logic and syntax errors. So, in preference to the preceding `#define`, you may prefer to use `const` variables, such as

ERR ⊗

```
const double PI = 3.14159;
```

These have the additional benefit that they're defined in C, so the compiler can check them for proper syntax and type safety.

✓ Self Check

1 *(Fill-In)* The `#define` directive creates ________ constants and ________.
Answer: symbolic, macros.

2 *(True/False)* The statement

```
#define PI 3.14159;
```

replaces all subsequent occurrences of the symbolic constant `PI` with the numeric constant `3.14159`.
Answer: *False*. Actually, this `#define` is a common error. Preprocessor directives are not C statements and generally should not end in semicolons. The above statement would replace all occurrences of `PI` with `3.14159;` (including the semicolon), probably causing one or more compilation errors.

14.4 #define Preprocessor Directive: Macros

Technically, any identifier defined in a `#define` preprocessor directive is a macro. As with symbolic constants, the macro-identifier is replaced with replacement-text before the program is compiled. Macros may be defined with or without arguments.

A macro without arguments is a symbolic constant. When the preprocessor encounters a macro with arguments, it substitutes the arguments in the replacement text, then expands the macro—that is, it replaces the macro with its replacement-text and argument list.

14.4.1 Macro with One Argument

Consider the following one-argument macro definition that calculates a circle's area:

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

Expanding a Macro with an Argument

Wherever CIRCLE_AREA(*argument*) appears in the file, the preprocessor:

- substitutes *argument* for x in the replacement-text,
- replaces PI with its value 3.14159 (from Section 14.3), and
- expands the macro in the program.

For example, the preprocessor expands

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

to

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

At compile time, the compiler evaluates the preceding expression and assigns the result to the variable area.

Importance of Parentheses

The *parentheses* around each x in the replacement-text force the proper evaluation order when a macro's argument is an expression. Consider the statement

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

which expands to

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

This evaluates correctly because the parentheses force the proper evaluation order. If you omit the macro definition's parentheses, the macro expansion is

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

which evaluates *incorrectly* as ⊗ ERR

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

because of C's operator precedence rules. For this reason, you should always enclose macro arguments in parentheses in the replacement-text to prevent logic errors.

It's Better to Use a Function

Defining the CIRCLE_AREA macro as a function is safer. The circleArea function

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

performs the same calculation as macro CIRCLE_AREA, but a function's argument is evaluated only once when the function is called. Also, the compiler performs type checking on functions. The preprocessor does not support type checking. In the past, programmers often used macros to replace function calls with inline code to eliminate PERF the function-call overhead. Today's optimizing compilers often inline function calls for you, so many programmers no longer use macros for this purpose. You can also use the C standard's inline keyword (see Appendix C).

14.4.2 Macro with Two Arguments

The following two-argument macro calculates a rectangle's area:

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

Wherever RECTANGLE_AREA(x, y) appears in the program, the preprocessor substitutes the values of x and y in the macro's replacement-text and expands the macro in the program. For example, the statement

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

is expanded to

```
int rectangleArea = ((a + 4) * (b + 7));
```

14.4.3 Macro Continuation Character

A macro's or symbolic constant's replacement-text is everything to the identifier's right in the #define directive. If the replacement-text is longer than the remainder of the line, you can place a backslash (\) continuation character at the end of the line to continue the replacement-text on the next line.

14.4.4 #undef Preprocessor Directive

Symbolic constants and macros can be discarded for the remainder of a source file using the #undef preprocessor directive. Directive #undef *undefines* a symbolic constant or macro name. A macro's or symbolic constant's scope is from its definition until it's undefined with #undef, or until the end of the source file. Once undefined, a macro or symbolic constant can be redefined with #define.

14.4.5 Standard-Library Macros

Some standard-library functions actually are defined as macros, based on other library functions. A macro commonly defined in the <stdio.h> header is

```
#define getchar() getc(stdin)
```

The macro definition of getchar uses function getc to get one character from the standard input stream. The <stdio.h> header's putchar function and the <ctype.h> header's character-handling functions often are implemented as macros as well.

14.4.6 Do Not Place Expressions with Side Effects in Macros

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. We'll show an example of this in Section 14.11.

✓ Self Check

1 *(True/False)* The following two-argument macro calculates a rectangle's area:

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

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

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

is expanded to

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

The value of the expression is evaluated at runtime and assigned to variable `rectArea`.
Answer: *False*. Actually, the correct expansion is:

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

2 *(Fill-In)* Expressions with ________ should not be passed to a macro because macro arguments may be evaluated more than once.
Answer: side effects.

14.5 Conditional Compilation

Conditional compilation enables you to control which preprocessor directives execute and whether parts of your C code compile. Each conditional preprocessor directive evaluates a constant integer expression. Cast expressions, `sizeof` expressions and enumeration constants *cannot* be evaluated in preprocessor directives.

14.5.1 `#if...#endif` Preprocessor Directive

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

```
#if !defined(MY_CONSTANT)
   #define MY_CONSTANT 0
#endif
```

This determines whether `MY_CONSTANT` is *defined*—that is, whether `MY_CONSTANT` has already appeared in an earlier `#define` directive within the current source file. The expression `defined(MY_CONSTANT)` evaluates to `1` (true) if `MY_CONSTANT` is defined; otherwise, it evaluates to `0` (false). If the result is `0`, `!defined(MY_CONSTANT)` evaluates to `1`, indicating that `MY_CONSTANT` was not defined previously, so the `#define` directive executes. Otherwise, the preprocessor skips the `#define` directive.

Every `#if` construct ends with `#endif`. The directives `#ifdef` and `#ifndef` are shorthand for `#if defined(`*name*`)` and `#if !defined(`*name*`)`. You can test a multiple-part conditional preprocessor construct by using

- `#elif` (the equivalent of `else if` in an `if` statement) and
- `#else` (the equivalent of `else` in an `if` statement) directives.

Conditional preprocessor directives are frequently used to *prevent header files from being included multiple times in the same source file*. These directives frequently are used to enable and disable code that makes software compatible with a range of platforms.

14.5.2 Commenting Out Blocks of Code with #if…#endif

During program development, it's often helpful to "comment out" portions of your code to prevent them from being compiled. If the code contains multiline comments, `/*` and `*/` cannot do this because you cannot nest multiline comments. Instead, you can use the following preprocessor construct:

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

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

14.5.3 Conditionally Compiling Debug Code

Conditional compilation is sometimes used as a *debugging* aid. For example, some programmers use `printf` statements to print variable values and to confirm a program's flow of control. You can enclose such `printf` statements in conditional preprocessor directives so the statements are compiled only while you're still debugging your code. For example,

```
#ifdef DEBUG
   printf("Variable x = %d\n", x);
#endif
```

compiles the `printf` statement if the symbolic constant `DEBUG` is defined with

```
#define DEBUG
```

before `#ifdef DEBUG`. When you complete your debugging phase, you remove or comment out the `#define` directive in the source file, and the `printf` statements inserted for debugging purposes are ignored during compilation. In larger programs, you might define several symbolic constants that control the conditional compilation in separate sections of the source file.

Many compilers allow you to define and undefine symbolic constants like `DEBUG` with a compiler flag that you supply each time you compile the code so that you do not need to change the code. When inserting conditionally compiled `printf` statements in locations where C expects a single statement (e.g., a control statement's body), ensure that the conditionally compiled statements are enclosed in braces (`{}`).

Self Check

1 *(True/False)* The conditional compilation statement

```
#ifdef DEBUG
   printf("Variable x = %d\n", x);
#endif
```

compiles the `printf` statement if the symbolic constant `DEBUG` is defined (`#define DEBUG`) before `#ifdef DEBUG`.
Answer: *True*.

2 *(True/False)* During program development, it's often helpful to "comment out" portions of code to prevent them from being compiled. If the code contains multiline comments, `/*` and `*/` should be used to accomplish this task.
Answer: *False*. Actually, if the code contains multiline comments, `/*` and `*/` cannot be used to accomplish this task, because such comments cannot be nested. Instead, you can use the following preprocessor construct:

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

14.6 #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 6 tokens. When the `#error` directive is processed on some systems, the tokens 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. For more information on `#error` and `#pragma`, see the documentation for your C compiler.

Self Check

1 *(Fill-In)* When a(n) ________ preprocessor directive is processed on some systems, the tokens in the directive are displayed as an error message, preprocessing stops and the program does not compile.
Answer: `#error`.

2 *(Fill-In)* The `#pragma` directive causes a(n) ________ action
Answer: implementation-defined.

14.7 # and ## Operators

The # operator converts a replacement-text token to a string surrounded by quotes. Consider the following macro definition:

```
#define HELLO(x) puts("Hello, " #x);
```

When HELLO(John) appears in a program file, the preprocessor expands it to

```
puts("Hello, " "John");
```

replacing #x with the string "John". Strings separated by whitespace are concatenated during preprocessing, so the preceding statement is equivalent to

```
puts("Hello, John");
```

The # operator must be used in a macro with arguments because #'s operand refers to one of the macro's arguments.

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

```
#define TOKENCONCAT(x, y)  x ## y
```

When TOKENCONCAT appears in a file, the preprocessor concatenates the arguments and uses the result to replace the macro. For example, TOKENCONCAT(O, K) is replaced by OK in the program. The ## operator must have two operands.

✓ Self Check

1 *(Fill-In)* The ________ preprocessor operator causes a replacement-text token to be converted to a string surrounded by quotes.
Answer: #.

2 *(Code)* The ## preprocessor operator concatenates two tokens. Write the macro definition described by, "When SIDEBYSIDE appears in the program, its arguments are concatenated and used to replace the macro. So, SIDEBYSIDE(GOOD, BYE) is replaced by GOODBYE in the program."
Answer: `#define SIDEBYSIDE(a, b) a ## b`

14.8 Line Numbers

The #line preprocessor directive causes the subsequent source-code lines to be renumbered, starting with the specified constant integer value. The directive

```
#line 100
```

starts line numbering from 100 beginning with the next source-code line. Including a filename in the #line directive, as in

```
#line 100 "file1.c"
```

indicates that lines are numbered from 100 beginning with the next source-code line and that the filename for the purpose of any compiler messages is "file1.c". This version of the #line directive normally helps make the messages produced by syntax errors and compiler warnings more meaningful. The line numbers do not appear in the source file.

✓ Self Check

1 *(True/False)* The preprocessor directive

```
#line 100 "file1.c"
```

indicates that lines are numbered from `100` beginning with the next source-code line and that the name of the file for the purpose of any compiler messages is `"file1.c"`.
Answer: *True*.

14.9 Predefined Symbolic Constants

Standard C provides predefined symbolic constants, several of which are shown in the following table:

Symbolic constant	Explanation
`__LINE__`	The line number of the current source-code line (an integer constant).
`__FILE__`	The name of the source file (a string).
`__DATE__`	The date the source file was compiled (in the form `"Mmm dd yyyy"`, such as `"Jan 19 2002"`).
`__TIME__`	The time the source file was compiled (in the form `"hh:mm:ss"`).
`__STDC__`	The value `1` if the compiler supports Standard C; `0` otherwise. Requires the compiler flag `/Za` in Visual C++.

The remaining predefined symbolic constants are in Section 6.10.8 of the C standard. These identifiers begin and end with *two* underscores and often are useful for including additional information in error messages. These identifiers and the `defined` identifier (used in Section 14.5) cannot be used in `#define` or `#undef` directives.

✓ Self Check

1 *(Fill-In)* The predefined symbolic constant ________ is described by, "The value `1` if the compiler supports Standard C; `0` otherwise."
Answer: `__STDC__`.

14.10 Assertions

The `assert` macro—defined in `<assert.h>`—tests an expression's value at execution time. If the value is false (`0`), `assert` prints an error message and terminates the program by calling function `abort` of the general utilities library (`<stdlib.h>`).

The `assert` macro 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. You can use an assertion to test `x`'s value and print an error message if it's greater than 10, as in

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

If `x` is greater than `10` when this statement executes, the program displays an error message containing the line number and filename where the `assert` statement appears, then terminates. You'd then focus on this area of the code to find the error.

If the symbolic constant `NDEBUG` is defined, subsequent assertions in the source file are *ignored*. So, when assertions are no longer needed, rather than deleting each assertion manually, you can insert the following line in the source file:

```
#define NDEBUG
```

Many compilers have debug and release modes that automatically define and undefine `NDEBUG`.

Assertions are not meant as a substitute for error handling during normal runtime conditions. You should use them only to find logic errors during program development. The C standard also includes a capability called `_Static_assert`, which is essentially a compile-time version of `assert` that produces a *compilation error* if the assertion fails. We discuss `_Static_assert` in Appendix C.

Self Check

1 *(True/False)* The `assert` macro—defined in `<assert.h>`—tests the value of an expression at compile time.
Answer: *False*. Actually, the `assert` macro tests the value of an expression at *execution time*. `_Static_assert` is essentially a compile-time version of `assert` that produces a compilation error if the assertion fails.

2 *(Fill-In)* When assertions are no longer needed, you can insert the line ________ in the code file rather than delete each assertion manually.
Answer: `#define NDEBUG`.

14.11 Secure C Programming

The `CIRCLE_AREA` macro defined in Section 14.4:

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

is an *unsafe* macro, because it evaluates its argument `x` *more than once*. This can cause subtle errors. If the macro argument contains *side effects*—such as incrementing a variable or calling a function that modifies a variable's value—those side effects would be performed *multiple* times.

For example, if we call `CIRCLE_AREA` as follows:

```
double result = CIRCLE_AREA(++radius);
```

The preprocessor expands this to

```
double result = ((3.14159) * (++radius) * (++radius));
```

which increments `radius` *twice*. Also, the preceding statement's result is *undefined*, because C allows a variable to be modified *only once* in a statement. In a function call, the argument is *evaluated only once* before it's passed to the function. So, functions are always preferred to unsafe macros.

✓ Self Check

1 *(Fill-In)* The `CIRCLE_AREA` macro

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

is considered unsafe because it ________. This can cause subtle errors.
Answer: evaluates its argument `x` more than once.

2 *(True/False)* Macros are always preferred to functions.
Answer: *False*. Actually, functions are always preferred to unsafe macros.

Summary

Section 14.1 Introduction

- The preprocessor executes before a program compiles.
- All **preprocessor directives** (p. 737) begin with `#`.
- Only whitespace characters and comments may appear before a preprocessor directive on a line.

Section 14.2 `#include` Presprocessor Directive

- The **`#include` directive** (p. 737) includes a copy of the specified file. If the filename is enclosed in quotes, the preprocessor begins searching in the same folder as the file being compiled. If the filename is enclosed in angle brackets (< and >), as is the case for C standard library headers, the search is performed in an implementation-defined manner.

Section 14.3 `#define` Preprocessor Directive: Symbolic Constants

- The **`#define` preprocessor directive** (p. 737) creates symbolic constants and macros.
- A **symbolic constant** (p. 737) is a name for a constant.

Section 14.4 `#define` Preprocessor Directive: Macros

- A **macro** is an operation defined in a `#define` preprocessor directive. Macros may be defined with or without arguments.
- **Replacement-text** (p. 738) is specified after a symbolic constant's identifier or after the closing right parenthesis of a macro's argument list. If the replacement-text for a macro or symbolic constant is longer than the remainder of the line, use a **backslash** (\; p. 740) at the end of the line to indicate that the replacement-text continues on the next line.
- Symbolic constants and macros can be discarded using the **`#undef` preprocessor directive** (p. 740). Directive `#undef` "undefines" the symbolic constant or macro name.
- The **scope** (p. 740) of a symbolic constant or macro is from its definition until it's undefined with `#undef` or until the end of the file.

Section 14.5 Conditional Compilation

- **Conditional compilation** (p. 742) enables you to control whether preprocessor directives execute and whether program code compiles.
- The **conditional preprocessor directives** evaluate constant integer expressions. Cast expressions, `sizeof` expressions and `enum` constants cannot be evaluated in preprocessor directives.
- Every **`#if`** construct ends with **`#endif`** (p. 742).

- Directives **#ifdef** and **#ifndef** (p. 742) are provided as shorthand for **#if defined(*name*)** and **#if !defined(*name*)**.
- **Multiple-part conditional preprocessor constructs** may be tested with directives **#elif** and **#else** (p. 742).

Section 14.6 #error and #pragma Preprocessor Directives

- The **#error directive** (p. 743) terminates preprocessing, prevents compilation and prints an implementation-dependent message that includes the tokens specified in the directive.
- The **#pragma directive** (p. 743) causes an implementation-defined action. If the #pragma is not recognized by the implementation, it's ignored.

Section 14.7 # and ## Operators

- The **# operator** converts a *replacement-text* token to a string surrounded by quotes. The # operator must be used in a macro with arguments because #'s operand must be one of the macro's arguments.
- The **## operator** concatenates two tokens. The ## operator must have two operands.

Section 14.8 Line Numbers

- The **#line preprocessor directive** (p. 744) causes the subsequent source-code lines to be renumbered, starting with the specified constant integer value. This directive also enables you to specify the filename used for that source-code file in compiler error messages.

Section 14.9 Predefined Symbolic Constants

- Constant **__LINE__** (p. 745) is the line number (an integer) of the current source-code line.
- Constant **__FILE__** (p. 745) is the name of the file (a string).
- Constant **__DATE__** (p. 745) is the date the source file is compiled (a string).
- Constant **__TIME__** (p. 745) is the time the source file is compiled (a string).
- Constant **__STDC__** (p. 745) indicates whether the compiler supports Standard C.
- Each of the predefined symbolic constants begins and ends with two underscores.

Section 14.10 Assertions

- Macro **assert** (p. 745; **<assert.h> header**) tests the value of an expression. If the value is 0 (false), assert prints an error message and calls **function abort** (p. 745) to terminate program execution.

Self-Review Exercises

14.1 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 ________ 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 whether preprocessor directives execute and whether code compiles.
h) The ________ macro prints a message and terminates program execution if the macro's expression evaluates is `0`.
i) The ________ directive inserts a file in another file.
j) The ________ preprocessor operator concatenates its two arguments.
k) The ________ preprocessor 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.
m) The ________ directive causes the source-code lines to be numbered from the indicated value beginning with the next source-code line.

14.2 Write a program to print the values of the predefined symbolic constants listed in Section 14.9.

14.3 Write a preprocessor directive to accomplish each of the following:
a) Define the symbolic constant `YES` to have the value `1`.
b) Define the symbolic constant `NO` to have the value `0`.
c) Include the header `common.h`. The header is found in the same directory as the file being compiled.
d) Renumber the remaining lines in the file beginning with line number `3000`.
e) If the symbolic constant `TRUE` is defined, undefine it and redefine it as `1`. Do not use `#ifdef`.
f) If the symbolic constant `TRUE` is defined, undefine it and redefine it as `1`. Use the `#ifdef` preprocessor directive.
g) If the symbolic constant `TRUE` is not equal to `0`, define symbolic constant `FALSE` as `0`. Otherwise define `FALSE` as `1`.
h) Define the macro `CUBE_VOLUME` that computes the volume of a cube. The macro takes one argument.

Answers to Self-Review Exercises

14.1 a) `#`. b) `#elif`, `#else`. c) `#define`. d) whitespace. e) `#undef`. f) `#ifdef`, `#ifndef`. g) Conditional compilation. h) `assert`. i) `#include`. j) `##`. k) `#`. l) `\`. m) `#line`.

14.2 See below. [*Note:* In Visual Studio, `__STDC__` works requires the `/Za` compiler flag.]

```c
// ex14_02.c
// Print the values of the predefined macros
#include <stdio.h>
int main(void) {
   printf("__LINE__ = %d\n", __LINE__);
   printf("__FILE__ = %s\n", __FILE__);
   printf("__DATE__ = %s\n", __DATE__);
   printf("__TIME__ = %s\n", __TIME__);
   printf("__STDC__ = %d\n", __STDC__);
}
```

```
__LINE__ = 5
__FILE__ = ex14_02.c
__DATE__ = Jan 01 2021
__TIME__ = 11:39:12
__STDC__ = 1
```

14.3 See the answers below:

a) `#define YES 1`

b) `#define NO 0`

c) `#include "common.h"`

d) `#line 3000`

e)
```
#if defined(TRUE)
   #undef TRUE
   #define TRUE 1
#endif
```

f)
```
#ifdef TRUE
   #undef TRUE
   #define TRUE 1
#endif
```

g)
```
#if TRUE
   #define FALSE 0
#else
   #define FALSE 1
#endif
```

h) `#define CUBE_VOLUME(x)  ((x) * (x) * (x))`

Exercises

14.4 *(Subtracting Two Numbers)* Write a program that defines a macro `SUBTRACT` with two arguments, `x` and `y`, to subtract `y` from `x`. Your program should print the following output:

```
x subtract y is 17
```

14.5 *(Volume of a Cube)* Write a program that defines macro `CUBE` with one argument to compute a cube's volume. The argument represents the length of the cube. The formula for a cube's volume is length raised to the power of three. Test your program with examples.

14.6 *(Smaller of Two Numbers)* Write a program that defines and uses a macro named `MINIMUM2` to determine the smaller of two numeric values.

14.7 *(Smallest of Three Numbers)* Write a program that defines and uses a macro named `MINIMUM3` to determine the smallest of three numeric values. Macro `MINIMUM3` should use macro `MINIMUM2` from Exercise 14.6 to determine the smallest number.

14.8 *(Printing Repeated Strings)* Write a program that defines and uses macro `PRINT3`, which accepts a string value and prints three iterations of the same string together.

For example, `PRINT3(“hello”)` should produce the output “`hellohellohello`”.

14.9 *(Traversing Array and Compute Average)* Write a program that defines and uses macro `AVERAGEARRAY` to compute, and print the average of an array of floating point numbers. The macro should receive the array and its number of elements as arguments.

14.10 *(Power)* Write a program that defines and uses macro `POW`, which computes the power of a number. The macro should accept two arguments, `x` and `y`, and print the result of `x` to the power of `y`.

This page is intentionally left blank