EXAMPLE 14.14 Consider the macro definition

```
#define  root(a, b)  sqrt(a*a + b*b)
```

Now suppose that this macro is utilized within a program in the following manner.

```
root(a+1, b+2)
```

The intent, of course, is to evaluate the formula

```
sqrt((a+1)*(a+1) + (b+2)*(b+2))
```

However, each appearance of a is replaced by the expression a + 1 (without parentheses), and each appearance of b is replaced by b + 1. Therefore, the result of the macro substitution will be

```
sqrt(a+1*a+1 + b+2*b+2)
```

This expression is equivalent to

```
sqrt(2*a+1 + 3*b+2) = sqrt(2*a + 3*b + 3)
```

which is clearly incorrect. The source of error can be corrected, however, by placing additional parentheses within the macro definition; i.e.,

```
#define  root(a, b)  sqrt((a)*(a) + (b)*(b))
```

A more subtle error occurs if we write

```
root(a++, b++)
```

The macro substitution results in the expression

```
sqrt(a*(a+1) + b*(b+1))
```

rather than

```
sqrt(a*a + b*b)
```

as intended. This is an example of an undesired side effect. The placement of additional parentheses within the macro definition will not correct this problem.

14.5 THE C PREPROCESSOR

The C preprocessor is a collection of special statements, called *directives*, that are executed at the beginning of the compilation process. The `#include` and `#define` statements considered earlier in this book are preprocessor directives. Additional preprocessor directives are `#if`, `#elif`, `#else`, `#endif`, `#ifdef`, `#ifndef`, `#line` and `#undef`. The preprocessor also includes three special operators: `defined`, `#`, and `##`.

Preprocessor directives usually appear at the beginning of a program, though this is not a firm requirement. Thus, a preprocessor directive may appear anywhere within a program. However, the directive will apply only to the portion of the program following its appearance.

For the beginning programmer, some of the preprocessor directives are relatively unimportant. Hence, we will not describe each preprocessor feature in detail. The more important features are discussed below.

The `#if`, `#elif`, `#else` and `#endif` directives are used most frequently. They permit conditional compilation of the source program, depending on the value of one or more true/false conditions. They are sometimes used in conjunction with the `defined` operator, which is used to determine whether or not a symbolic constant or a macro identifier has been defined within a program.

EXAMPLE 14.15 The following preprocessor directives illustrate the conditional compilation of a C program. The conditional compilation depends on the status of the symbolic constant FOREGROUND.

```
#if  defined(FOREGROUND)
    #define  BACKGROUND  0
#else
    #define  FOREGROUND  0
    #define  BACKGROUND  7
#endif
```

Thus, if FOREGROUND has already been defined, the symbolic constant BACKGROUND will represent the value 0. Otherwise, FOREGROUND and BACKGROUND will represent the values 0 and 7, respectively.

Here is another way to accomplish the same thing.

```
#ifdef  FOREGROUND
    #define  BACKGROUND  0
#else
    #define  FOREGROUND  0
    #define  BACKGROUND  7
#endif
```

The directive `#ifdef` is equivalent to `#if defined()`. Similarly, the directive `#ifndef` is equivalent to `#if !defined()`, i.e., "if not defined." The original approach, in which the `defined` operator appears explicitly, is the preferred form.

In each of these examples, the last directive is `#endif`. The preprocessor requires that any set of directives beginning with `#if`, `#ifdef` or `#ifndef` must end with `#endif`.

The directive `#elif` is analogous to an `else - if` clause using ordinary C control statements. An `#if` directive can be followed by any number of `#elif` directives, though there can be only one `#else` directive. The appearance of the `#else` directive is optional, as determined by the required program logic.

EXAMPLE 14.16 Here is another illustration of conditional compilation. In this situation the conditional compilation will depend on the value that is represented by the symbolic constant BACKGROUND.

```
#if BACKGROUND == 7
    #define  FOREGROUND  0
#elif BACKGROUND == 6
    #define  FOREGROUND  1
#else
    #define  FOREGROUND  6
#endif
```

In this example we see that FOREGROUND will represent 0 if BACKGROUND represents 7, and FOREGROUND will represent 1 if BACKGROUND represents 6. Otherwise, FOREGROUND will represent 6.

The `#undef` directive "undefines" a symbolic constant or a macro identifier; i.e., it negates the effect of a `#define` directive that may have appeared earlier in the program.

EXAMPLE 14.17 The following example illustrates the use of the `#undefine` directive within a C program.

```
#define  FOREGROUND  7
#define  BACKGROUND  0

main()

{

    . . . . .

    #undef  FOREGROUND

    . . . . .

    #undef  BACKGROUND

    . . . . .

}
```

The symbolic constants `FOREGROUND` and `BACKGROUND` are defined by the first two directives. These definitions are then negated by the `#undef` directives, when they appear later in the program. Prior to the `#undef` directives, any references to `FOREGROUND` or `BACKGROUND` will be associated with the values 7 and 0, respectively. After the `#undef` directives, any references to the corresponding identifiers will be ignored.

The "stringizing" operator # allows a formal argument within a macro definition to be converted to a string. If a formal argument in a macro definition is preceded by this operator, the corresponding actual argument will automatically be enclosed in double quotes. Consecutive whitespace characters inside the actual argument will be replaced by a single blank space, and any special characters, such as ', " and \, will be replaced by their corresponding escape sequences; e.g., \', \" and \\. In addition, the resulting string will automatically be concatenated (i.e., combined) with any adjacent strings.

EXAMPLE 14.18 Here is an illustration of the use of the "stringizing" operator, #.

```
#define  display(text)  printf(#text "\n")

main()
{
    . . . . .
    display(Please do not sleep in class.);
    . . . . .
    display(Please   -   don't snore during the professor's lecture!);
}
```

Within `main`, the macros are equivalent to

```
printf("Please do not sleep in class.\n");
```

and

```
printf("Please - don\'t snore during the professor\'s lecture!\n");
```

Notice that each actual argument is converted to a string within the `printf` function. Each argument is concatenated with a newline character (`\n`), which is written as a separate string within the macro definition. Also, notice that the consecutive blank spaces appearing in the second argument are replaced by single blank spaces, and each apostrophe (') is replaced by its corresponding escape sequence (`\'`).

Execution of this program will result in the following output:

```
Please do not sleep in class.

Please - don't snore during the professor's lecture!
```

The "token-pasting" operator ## causes individual items within a macro definition to be concatenated, thus forming a single item. The various rules governing the use of this operator are somewhat complicated. However, the general purpose of the token-pasting operator is illustrated in the following example.

EXAMPLE 14.19 A C program contains the following macro definition.

```
#define  display(i)  printf("x" #i " = %f\n", x##i)
```

Suppose this macro is accessed by writing

```
display(3);
```

The result will be

```
printf("x3 = %f\n", x3);
```

Thus, the expression `x##i` becomes the variable `x3`, since `3` is the current value of the argument `i`.

Notice that this example illustrates the use of both the stringizing operator (#) and the token-pasting operator (##).

Refer to the programmer's reference manual for your particular C compiler for more information on the use of the C preprocessor.

Review Questions

14.1 What is an enumeration? How is an enumeration defined?

14.2 What are enumeration constants? In what form are they written?

14.3 Summarize the rules for assigning names to enumeration constants.

14.4 Summarize the rules for assigning numerical values to enumeration constants. What default values are assigned to enumeration constants?

14.5 Can two or more enumeration constants have the same numerical value? Explain.

14.6 What are enumeration variables? How are they declared?

14.7 In what ways can enumeration variables be processed? What restrictions apply to the processing of enumeration variables?

14.8 What advantage is there in using enumeration variables within a program?

14.9 Summarize the rules for assigning initial values to enumeration variables. Compare your answer with that for Prob. 14.4.

14.10 Most C programs recognize two formal arguments in the definition of function `main`. What are they traditionally called? What are their respective data types?

14.11 Describe the information represented by each formal argument in function `main`. Is information passed explicitly to each argument?