7

C Control Statements: Branching and Jumps

You will learn about the following in this chapter:

- Keywords

  `if`, `else`, `switch`, `continue`

  `break`, `case`, `default`, `goto`

- Operators

  `&& || ?:`

- Functions

  `getchar()`, `putchar()`, the `ctype.h` family

- How to use the `if` and `if else` statements and how to nest them
- Using logical operators to combine relational expressions into more involved test expressions
- C's conditional operator
- The `switch` statement
- The `break`, `continue`, and `goto` jumps
- Using C's character I/O functions—`getchar()` and `putchar()`
- The family of character-analysis functions provided by the `ctype.h` header file

As you grow more comfortable with C, you will probably want to tackle more complex tasks. When you do, you'll need ways to control and organize these projects. C has the tools to meet these needs. You've already learned to use loops to program repetitive tasks. In this chapter, you'll learn about branching structures such as `if` and `switch`, which allow a program to base its actions on conditions it checks. Also, you are introduced to C's logical operators, which enable you to test for more than one relationship in a `while` or `if` condition, and you look at

C's jump statements, which shift the program flow to another part of a program. By the end of this chapter, you'll have all the basic information you need to design a program that behaves the way you want.

The `if` Statement

Let's start with a simple example of an `if` statement, shown in Listing 7.1. This program reads in a list of daily low temperatures (in Celsius) and reports the total number of entries and the percentage that were below freezing (that is, below zero degrees Celsius). It uses `scanf()` in a loop to read in the values. Once during each loop cycle, it increments a counter to keep track of the number of entries. An `if` statement identifies temperatures below freezing and keeps track of the number of below-freezing days separately.

Listing 7.1 **The `colddays.c` Program**

```
// colddays.c -- finds percentage of days below freezing
#include <stdio.h>
int main(void)
{
    const int FREEZING = 0;
    float temperature;
    int cold_days = 0;
    int all_days = 0;

    printf("Enter the list of daily low temperatures.\n");
    printf("Use Celsius, and enter q to quit.\n");
    while (scanf("%f", &temperature) == 1)
    {
        all_days++;
        if (temperature < FREEZING)
            cold_days++;
    }
    if (all_days != 0)
        printf("%d days total: %.1f%% were below freezing.\n",
               all_days, 100.0 * (float) cold_days / all_days);
    if (all_days == 0)
        printf("No data entered!\n");

    return 0;
}
```

Here is a sample run:

```
Enter the list of daily low temperatures.
Use Celsius, and enter q to quit.
```

```
12 5 -2.5 0 6 8 -3 -10 5 10 q
10 days total: 30.0% were below freezing.
```

The `while` loop test condition uses the return value of `scanf()` to terminate the loop when `scanf()` encounters nonnumeric input. By using `float` instead of `int` for `temperature`, the program is able to accept input such as `-2.5` as well as `8`.

Here is the new statement in the `while` block:

```
if (temperature < FREEZING)
    cold_days++;
```

This `if` statement instructs the computer to increase `cold_days` by 1 *if* the value just read (`temperature`) is less than zero. What happens if `temperature` is not less than zero? Then the `cold_days++;` statement is skipped, and the `while` loop moves on to read the next temperature value.

The program uses the `if` statement two more times to control the output. If there is data, the program prints the results. If there is no data, the program reports that fact. (Soon you'll see a more elegant way to handle this part of the program.)

To avoid integer division, the example uses the cast to `float` when the percentage is being calculated. You don't really need the type cast because in the expression `100.0 * cold_days / all_days`, the subexpression `100.0 * cold_days` is evaluated first and is forced into floating point by the automatic type conversion rules. Using the type cast documents your intent, however, and helps protect the program against misguided revisions. The `if` statement is called a *branching statement* or *selection statement* because it provides a junction where the program has to select which of two paths to follow. The general form is this:

```
if (expression)
      statement
```

If *`expression`* evaluates to true (nonzero), *`statement`* is executed. Otherwise, it is skipped. As with a `while` loop, *`statement`* can be either a single statement or a single block (also termed a compound statement). The structure is very similar to that of a `while` statement. The chief difference is that in an `if` statement, the test and (possibly) the execution are done just once, but in the `while` loop, the test and execution can be repeated several times.

Normally, *`expression`* is a relational expression; that is, it compares the magnitude of two quantities, as in the expressions `x > y` and `c == 6`. If *`expression`* is true (`x` is greater than `y`, or `c` does equal `6`), the statement is executed. Otherwise, the statement is ignored. More generally, any expression can be used, and an expression with a `0` value is taken to be false.

The statement portion can be a simple statement, as in the example, or it can be a compound statement or block, marked off by braces:

```
if (score > big)
    printf("Jackpot!\n");  // simple statement

if (joe > ron)
```

```
{                           // compound statement
    joecash++;
    printf("You lose, Ron.\n");
}
```

Note that the entire `if` structure counts as a single statement, even when it uses a compound statement.

Adding `else` to the `if` Statement

The simple form of an `if` statement gives you the choice of executing a statement (possibly compound) or skipping it. C also enables you to choose between two statements by using the `if else` form. Let's use the `if else` form to fix an awkward segment from Listing 7.1.

```
if (all_days != 0)
    printf("%d days total: %.1f%% were below freezing.\n",
           all_days, 100.0 * (float) cold_days / all_days);
if (all_days == 0)
    printf("No data entered!\n");
```

If the program finds that `all_days` is not equal to `0`, it should know that `days` must be `0` without retesting, and it does. With `if else`, you can take advantage of that knowledge by rewriting the fragment this way:

```
if (all_days!= 0)
    printf("%d days total: %.1f%% were below freezing.\n",
           all_days, 100.0 * (float) cold_days / all_days);
else
    printf("No data entered!\n");
```

Only one test is made. If the `if` test expression is true, the temperature data is printed. If it's false, the warning message is printed.

Note the general form of the `if else` statement:

```
if (expression)
    statement1
else
    statement2
```

If *`expression`* is true (nonzero), *`statement1`* is executed. If *`expression`* is false or zero, the single statement following the `else` is executed. The statements can be simple or compound. C doesn't require indentation, but it is the standard style. Indentation shows at a glance the statements that depend on a test for execution.

If you want more than one statement between the `if` and the `else`, you must use braces to create a single block. The following construction violates C syntax, because the compiler expects just one statement (single or compound) between the `if` and the `else`:

```
if (x > 0)
    printf("Incrementing x:\n");
    x++;
else           // will generate an error
    printf("x <= 0 \n");
```

The compiler sees the `printf()` statement as part of the `if` statement, and it sees the `x++;` statement as a separate statement, not as part of the `if` statement. It then sees the `else` as being unattached to an `if`, which is an error. Instead, use this:

```
if (x > 0)
{
    printf("Incrementing x:\n");
    x++;
}
else
    printf("x <= 0 \n");
```

The `if` statement enables you to choose whether to do one action. The `if else` statement enables you to choose between two actions. Figure 7.1 compares the two statements.

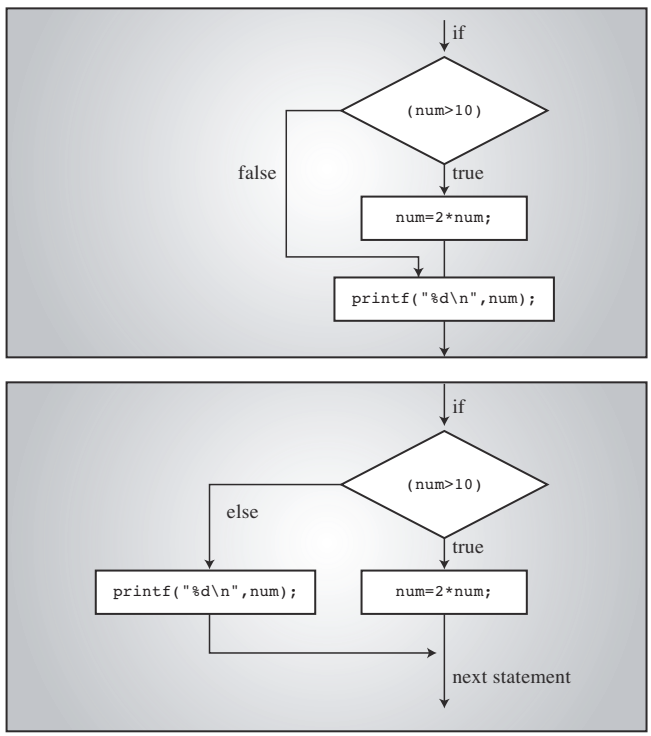


Figure 7.1 `if` versus `if else`.

Another Example: Introducing `getchar()` and `putchar()`

Most of the examples so far have used numeric input. To give you practice with other types, let's look at a character-oriented example. You already know how to use `scanf()` and `printf()` with the `%c` specifier to read and write characters; but now you'll meet a pair of C functions specifically designed for character-oriented I/O—`getchar()` and `putchar()`.

The `getchar()` function takes no arguments, and it returns the next character from input. For example, the following statement reads the next input character and assigns its value to the variable `ch`:

```
ch = getchar();
```

This statement has the same effect as the following statement:

```
scanf("%c", &ch);
```

The `putchar()` function prints its argument. For example, the next statement prints as a character the value previously assigned to `ch`:

```
putchar(ch);
```

This statement has the same effect as the following:

```
printf("%c", ch);
```

Because these functions deal only with characters, they are faster and more compact than the more general `scanf()` and `printf()` functions. Also, note that they don't need format specifiers; that's because they work with characters only. Both functions are typically defined in the `stdio.h` file. (Also, typically, they are preprocessor *macros* rather than true functions; we'll talk about function-like macros in Chapter 16, "The C Preprocessor and the C Library.")

Let's see how these functions work by writing a program that repeats an input line but replaces each non-space character with the character that follows it in the ASCII code sequence. Spaces will be reproduced as spaces. You can state the desired response as, "If the character is a space, print it; otherwise, print the next character in the ASCII sequence."

The C code looks much like this statement, as you can see in Listing 7.2.

Listing 7.2 **The `cypher1.c` Program**

```
// cypher1.c -- alters input, preserving spaces
#include <stdio.h>
#define SPACE ' '             // that's quote-space-quote
int main(void)
{
    char ch;

    ch = getchar();           // read a character
    while (ch != '\n')        // while not end of line
    {
```

```
        if (ch == SPACE)      // leave the space
            putchar(ch);      // character unchanged
        else
            putchar(ch + 1);  // change other characters
        ch = getchar();       // get next character
    }
    putchar(ch);              // print the newline

    return 0;
}
```

(If your compiler complains about possible data loss due to conversion, don't worry. Chapter 8, "Character Input/Output and Input Validation," will explain all when it introduces `EOF`.)

Here is a sample run:

```
CALL ME HAL.
DBMM NF IBM/
```

Compare this loop to the one from Listing 7.1. Listing 7.1 uses the status returned by `scanf()` instead of the value of the input item to determine when to terminate the loop. Listing 7.2, however, uses the value of the input item itself to decide when to terminate the loop. This difference results in a slightly different loop structure, with one read statement before the loop and one read statement at the end of each loop. C's flexible syntax, however, enables you to emulate Listing 7.1 by combining reading and testing into a single expression. That is, you can replace a loop of the form

```
ch = getchar();            /* read a character         */
while (ch != '\n')         /* while not end of line    */
{
    ...                      /* process character        */
    ch = getchar();        /* get next character       */
}
```

with one that looks like this:

```
while ((ch = getchar()) != '\n')
{
    ...                      /* process character        */
}
```

The critical line is

```
while ((ch = getchar()) != '\n')
```

It demonstrates a characteristic C programming style—combining two actions in one expression. C's free-formatting facility can help to make the separate components of the line clearer:

```
while (
       (ch = getchar())             // assign a value to ch
                        != '\n')    // compare ch to \n
```

The actions are assigning a value to `ch` and comparing this value to the newline character. The parentheses around `ch = getchar()` make it the left operand of the `!=` operator. To evaluate this expression, the computer must first call the `getchar()` function and then assign its return value to `ch`. Because the value of an assignment expression is the value of the left member, the value of `ch = getchar()` is just the new value of `ch`. Therefore, after `ch` is read, the test condition boils down to `ch != '\n'` (that is, to `ch` *not* being the newline character).

This particular idiom is very common in C programming, so you should be familiar with it. You also should make sure you remember to use parentheses to group the subexpressions properly.

All the parentheses are necessary. Suppose that you mistakenly used this:

```
while (ch = getchar() != '\n')
```

The `!=` operator has higher precedence than `=`, so the first expression to be evaluated is `getchar() != '\n'`. Because this is a relational expression, its value is `1` or `0` (true or false). Then this value is assigned to `ch`. Omitting the parentheses means that `ch` is assigned `0` or `1` rather than the return value of `getchar()`; this is not desirable.

The statement

```
putchar(ch + 1);   /* change other characters  */
```

illustrates once again that characters really are stored as integers. In the expression `ch + 1`, `ch` is expanded to type `int` for the calculation, and the resulting `int` is passed to `putchar()`, which takes an `int` argument but only uses the final byte to determine which character to display.

The `ctype.h` Family of Character Functions

Notice that the output for Listing 7.2 shows a period being converted to a slash; that's because the ASCII code for the slash character is one greater than the code for the period character. But if the point of the program is to convert only letters, it would be nice to leave all nonletters, not just spaces, unaltered. The logical operators, discussed later in this chapter, provide a way to test whether a character is not a space, not a comma, and so on, but it would be rather cumbersome to list all the possibilities. Fortunately, C has a standard set of functions for analyzing characters; the `ctype.h` header file contains the prototypes. These functions take a character as an argument and return nonzero (true) if the character belongs to a particular category and zero (false) otherwise. For example, the `isalpha()` function returns a nonzero value if its argument is a letter. Listing 7.3 generalizes Listing 7.2 by using this function; it also incorporates the shortened loop structure we just discussed.

Listing 7.3 **The `cypher2.c` Program**

```
// cypher2.c -- alters input, preserving non-letters
#include <stdio.h>
#include <ctype.h>            // for isalpha()
int main(void)
{
    char ch;

    while ((ch = getchar()) != '\n')
    {
        if (isalpha(ch))      // if a letter,
            putchar(ch + 1);  // display next letter
        else                  // otherwise,
            putchar(ch);      // display as is
    }
    putchar(ch);              // display the newline

    return 0;
}
```

Here is a sample run; note how both lowercase and uppercase letters are enciphered, but spaces and punctuation are not:

```
Look! It's a programmer!
Mppl! Ju't b qsphsbnnfs!
```

Tables 7.1 and 7.2 list several functions provided when you include the `ctype.h` header file. Some mention a locale; this refers to C's facility for specifying a locale that modifies or extends basic C usage. (For example, many nations use a comma instead of a decimal point when writing decimal fractions, and a particular locale could specify that C use the comma in the same way for floating-point output, thus displaying `123.45` as `123,45`.) Note that the mapping functions don't modify the original argument; instead, they return the modified value. That is,

```
tolower(ch);       // no effect on ch
```

doesn't change `ch`. To change `ch`, do this:

```
ch = tolower(ch);  // convert ch to lowercase
```

Table 7.1 **The `ctype.h` Character-Testing Functions**

Name	True If the Argument Is
`isalnum()`	Alphanumeric (alphabetic or numeric)
`isalpha()`	Alphabetic

Name	True If the Argument Is
`isblank()`	A standard blank character (space, horizontal tab, or newline) or any additional locale-specific character so specified
`iscntrl()`	A control character, such as Ctrl+B
`isdigit()`	A digit
`isgraph()`	Any printing character other than a space
`islower()`	A lowercase character
`isprint()`	A printing character
`ispunct()`	A punctuation character (any printing character other than a space or an alphanumeric character)
`isspace()`	A whitespace character (a space, newline, formfeed, carriage return, vertical tab, horizontal tab, or, possibly, other locale-defined character)
`isupper()`	An uppercase character
`isxdigit()`	A hexadecimal-digit character

Table 7.2 **The `ctype.h` Character-Mapping Functions**

Name	Action
`tolower()`	If the argument is an uppercase character, this function returns the lowercase version; otherwise, it just returns the original argument.
`toupper()`	If the argument is a lowercase character, this function returns the uppercase version; otherwise, it just returns the original argument.

Multiple Choice `else if`

Life often offers us more than two choices. You can extend the `if else` structure with `else if` to accommodate this fact. Let's look at a particular example. Utility companies often have charges that depend on the amount of energy the customer uses. Here are the rates one company charges for electricity, based on kilowatt-hours (kWh):

First 360 kWh:	$0.13230 per kWh
Next 108 kWh:	$0.15040 per kWh
Next 252 kWh:	$0.30025 per kWh
Over 720 kWh:	$0.34025 per kWh

If you worry about your energy management, you might want to prepare a program to calculate your energy costs. The program in Listing 7.4 is a first step in that direction.

Listing 7.4 **The `electric.c` Program**

```
// electric.c -- calculates electric bill
#include <stdio.h>
#define RATE1   0.13230       // rate for first 360 kwh
#define RATE2   0.15040       // rate for next 108 kwh
#define RATE3   0.30025       // rate for next 252 kwh
#define RATE4   0.34025       // rate for over 720 kwh
#define BREAK1  360.0         // first breakpoint for rates
#define BREAK2  468.0         // second breakpoint for rates
#define BREAK3  720.0         // third breakpoint for rates
#define BASE1   (RATE1 * BREAK1)
// cost for 360 kwh
#define BASE2  (BASE1 + (RATE2 * (BREAK2 - BREAK1)))
// cost for 468 kwh
#define BASE3   (BASE1 + BASE2 + (RATE3 *(BREAK3 - BREAK2)))
//cost for 720 kwh
int main(void)
{
    double kwh;               // kilowatt-hours used
    double bill;              // charges

    printf("Please enter the kwh used.\n");
    scanf("%lf", &kwh);       // %lf for type double
    if (kwh <= BREAK1)
        bill = RATE1 * kwh;
    else if (kwh <= BREAK2)   // kwh between 360 and 468
        bill = BASE1 + (RATE2 * (kwh - BREAK1));
    else if (kwh <= BREAK3)   // kwh betweent 468 and 720
        bill = BASE2 + (RATE3 * (kwh - BREAK2));
    else                      // kwh above 680
        bill = BASE3 + (RATE4 * (kwh - BREAK3));
    printf("The charge for %.1f kwh is $%1.2f.\n", kwh, bill);

    return 0;
}
```

Here's some sample output:

```
Please enter the kwh used.
580
The charge for 580.0 kwh is $97.50.
```

Listing 7.4 uses symbolic constants for the rates so that the constants are conveniently gathered in one place. If the power company changes its rates (it's possible), having the rates in one place makes them easy to update. The listing also expresses the rate breakpoints symbolically. They, too, are subject to change. `BASE1` and `BASE2` are expressed in terms of the rates and breakpoints. Then, if the rates or breakpoints change, the bases are updated automatically. You may recall that the preprocessor does not do calculations. Where `BASE1` appears in the program, it will be replaced by 0.13230 `* 360.0`. Don't worry; the compiler does evaluate this expression to its numerical value (`47.628`) so that the final program code uses `47.628` rather than a calculation.

The flow of the program is straightforward. The program selects one of three formulas, depending on the value of `kwh`. You should pay particular attention to the fact that the only way the program can reach the first `else` is if `kwh` is equal to or greater than `360`. Therefore, the `else if (kwh <= BREAK2)` line really is equivalent to demanding that `kwh` be between `360` and `482`, as the program comment notes. Similarly, the final `else` can be reached only if `kwh` exceeds `720`. Finally, note that `BASE1`, `BASE2`, and `BASE3` represent the total charges for the first 360, 468, and 720 kilowatt-hours, respectively. Therefore, you need to add on only the additional charges for electricity in excess of those amounts.

Actually, the `else if` is a variation on what you already knew. For example, the core of the program is just another way of writing

```
if (kwh <= BREAK1)
    bill = RATE1 * kwh;
else
    if (kwh <= BREAK2)   // kwh between 360 and 468
        bill = BASE1 + (RATE2 * (kwh - BREAK1));
    else
        if (kwh <= BREAK3)   // kwh betweent 468 and 720
            bill = BASE2 + (RATE3 * (kwh - BREAK2));
        else                       // kwh above 680
            bill = BASE3 + (RATE4 * (kwh - BREAK3));
```

That is, the program consists of an `if else` statement for which the statement part of the `else` is another `if else` statement. The second `if else` statement is said to be *nested* inside the first and the third inside the second. Recall that the entire `if else` structure counts as a single statement, which is why we didn't have to enclose the nested `if else` statements in braces. However, using braces would clarify the intent of this particular format.

These two forms are perfectly equivalent. The only differences are in where you put spaces and newlines, and these differences are ignored by the compiler. Nonetheless, the first form is better because it shows more clearly that you are making a four-way choice. This form makes it easier to skim the program and see what the choices are. Save the nested forms of indentation for when they are needed—for example, when you must test two separate quantities. An example of such a situation is having a 10% surcharge for kilowatt-hours in excess of 720 during the summer only.

You can string together as many `else if` statements as you need (within compiler limits, of course), as illustrated by this fragment:

```
if (score < 1000)
     bonus = 0;
else if (score < 1500)
     bonus = 1;
else if (score < 2000)
     bonus = 2;
else if (score < 2500)
     bonus = 4;
else
     bonus = 6;
```

(This might be part of a game program, in which `bonus` represents how many additional photon bombs or food pellets you get for the next round.)

Speaking of compiler limits, the C99 standard requires that a compiler support a minimum of 127 levels of nesting.

Pairing `else` with `if`

When you have a lot of `if`s and `else`s, how does the computer decide which `if` goes with which `else`? For example, consider the following program fragment:

```
if (number > 6)
    if (number < 12)
       printf("You're close!\n");
else
    printf("Sorry, you lose a turn!\n");
```

When is `Sorry, you lose a turn!` printed? When `number` is less than or equal to `6`, or when `number` is greater than `12`? In other words, does the `else` go with the first `if` or the second? The answer is, the `else` goes with the second `if`. That is, you would get these responses:

Number	Response
`5`	None
`10`	You're close!
`15`	Sorry, you lose a turn!

The rule is that an `else` goes with the most recent `if` unless braces indicate otherwise (see Figure 7.2).

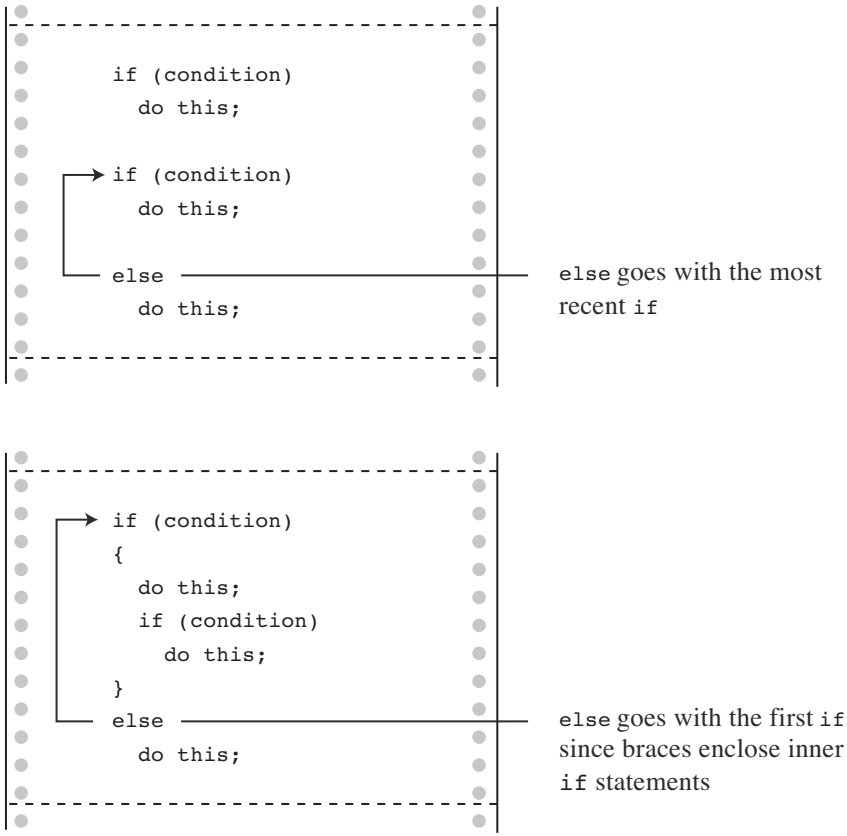


Figure 7.2 The rule for `if else` pairings.

Note: Indent the next-to-last "do this;" two spaces and terminate the last "do this" with a semicolon. Move the } and { two spaces to the left.

The indentation of the first example makes it look as though the `else` goes with the first `if`, but remember that the compiler ignores indentation. If you really want the `else` to go with the first `if`, you could write the fragment this way:

```
if (number > 6)
{
     if (number < 12)
         printf("You're close!\n");
}
else
   printf("Sorry, you lose a turn!\n");
```

Now you would get these responses:

Number	Response
5	Sorry, you lose a turn!
10	You're close!
15	None

More Nested `if`s

You've already seen that the `if...else if...else` sequence is a form of nested `if`, one that selects from a series of alternatives. Another kind of nested `if` is used when choosing a particular selection leads to an additional choice. For example, a program could use an `if else` to select between males and females. Each branch within the `if else` could then contain another `if else` to distinguish between different income groups.

Let's apply this form of nested `if` to the following problem. Given an integer, print all the integers that divide into it evenly; if there are no divisors, report that the number is prime.

This problem requires some forethought before you whip out the code. First, you need an overall design for the program. For convenience, the program should use a loop to enable you to input numbers to be tested. That way, you don't have to run the program again each time you want to examine a new number. We've already developed a model for this kind of loop:

```
prompt user
while the scanf() return value is 1
   analyze the number and report results
   prompt user
```

Recall that by using `scanf()` in the loop test condition, the program attempts both to read a number and to check to see whether the loop should be terminated.

Next, you need a plan for finding divisors. Perhaps the most obvious approach is something like this:

```
for (div = 2; div < num; div++)
  if (num % div == 0)
       printf("%d is divisible by %d\n", num, div);
```

The loop checks all the numbers between `2` and `num` to see whether they divide evenly into `num`. Unfortunately, this approach is wasteful of computer time. You can do much better. Consider, for example, finding the divisors of 144. You find that 144 % 2 is 0, meaning 2 goes into 144 evenly. If you then actually divide 2 into 144, you get 72, which also is a divisor, so you can get two divisors instead of one divisor out of a successful `num % div` test. The real payoff, however, comes in changing the limits of the loop test. To see how this works, look at the pairs of divisors you get as the loop continues: 2,72, 3,48, 4,36, 6,24, 8,18, 9,16, 12,12, 16,9, 18,8, and so on. Ah! After you get past the 12,12 pair, you start getting the same divisors

(in reverse order) that you already found. Instead of running the loop to 143, you can stop after reaching 12. That saves a lot of cycles!

Generalizing this discovery, you see that you have to test only up to the square root of `num` instead of to `num`. For numbers such as 9, this is not a big savings, but the difference is enormous for a number such as 10,000. Instead of messing with square roots, however, you can express the test condition as follows:

```
for (div = 2; (div * div) <= num; div++)
    if (num % div == 0)
         printf("%d is divisible by %d and %d.\n",
                   num, div, num / div);
```

If `num` is `144`, the loop runs through `div = 12`. If `num` is `145`, the loop runs through `div = 13`.

There are two reasons for using this test rather than a square root test. First, integer multiplication is faster than extracting a square root. Second, the square root function hasn't been formally introduced yet.

We need to address just two more problems, and then you'll be ready to program. First, what if the test number is a perfect square? Reporting that 144 is divisible by 12 and 12 is a little clumsy, but you can use a nested `if` statement to test whether `div` equals `num / div`. If so, the program will print just one divisor instead of two.

```
for (div = 2; (div * div) <= num; div++)
{
   if (num % div == 0)
   {
      if (div * div != num)
         printf("%d is divisible by %d and %d.\n",
             num, div, num / div);
      else
         printf("%d is divisible by %d.\n", num, div);
   }
}
```

> **Note**
>
> Technically, the `if else` statement counts as a single statement, so the braces around it are not needed. The outer `if` is a single statement also, so the braces around it are not needed. However, when statements get long, the braces make it easier to see what is happening, and they offer protection if later you add another statement to an `if` or to the loop.

Second, how do you know if a number is prime? If `num` is prime, program flow never gets inside the `if` statement. To solve this problem, you can set a variable to some value, say `1`, outside the loop and reset the variable to `0` inside the `if` statement. Then, after the loop is completed, you can check to see whether the variable is still `1`. If it is, the `if` statement was never entered, and the number is prime. Such a variable is often called a *flag*.

Traditionally, C has used the `int` type for flags, but the new `_Bool` type matches the requirements perfectly. Furthermore, by including the `stdbool.h` header file, you can use `bool` instead of the keyword `_Bool` for the type and use the identifiers `true` and `false` instead of `1` and `0`.

Listing 7.5 incorporates all these ideas. To extend the range, the program uses type `long` instead of type `int`. (If your system doesn't support the `_Bool` type, you can use the `int` type for `isPrime` and use `1` and `0` instead of `true` and `false`.)

Listing 7.5 **The `divisors.c` Program**

```
// divisors.c -- nested ifs display divisors of a number
#include <stdio.h>
#include <stdbool.h>
int main(void)
{
    unsigned long num;          // number to be checked
    unsigned long div;          // potential divisors
    bool isPrime;               // prime flag

    printf("Please enter an integer for analysis; ");
    printf("Enter q to quit.\n");
    while (scanf("%lu", &num) == 1)
    {
        for (div = 2, isPrime = true; (div * div) <= num; div++)
        {
            if (num % div == 0)
            {
                if ((div * div) != num)
                printf("%lu is divisible by %lu and %lu.\n",
                        num, div, num / div);
                else
                    printf("%lu is divisible by %lu.\n",
                           num, div);
                isPrime= false; // number is not prime
            }
        }
        if (isPrime)
            printf("%lu is prime.\n", num);
        printf("Please enter another integer for analysis; ");
        printf("Enter q to quit.\n");
    }
    printf("Bye.\n");

    return 0;
}
```

Note that the program uses the comma operator in the `for` loop control expression to enable you to initialize `isPrime` to `true` for each new input number.

Here's a sample run:

```
Please enter an integer for analysis; Enter q to quit.
123456789
123456789 is divisible by 3 and 41152263.
123456789 is divisible by 9 and 13717421.
123456789 is divisible by 3607 and 34227.
123456789 is divisible by 3803 and 32463.
123456789 is divisible by 10821 and 11409.
Please enter another integer for analysis; Enter q to quit.
149
149 is prime.
Please enter another integer for analysis; Enter q to quit.
2013
2013 is divisible by 3 and 671.
2013 is divisible by 11 and 183.
2013 is divisible by 33 and 61.
Please enter another integer for analysis; Enter q to quit.
q
Bye.
```

The program will identify 1 as prime, which, technically, it isn't. The logical operators, coming up in the next section, would let you exclude 1 from the prime list.

Summary: Using `if` Statements for Making Choices

Keywords:

`if`, `else`

General Comments:

In each of the following forms, the statement can be either a simple statement or a compound statement. A true expression means one with a nonzero value.

Form 1:

```
if (expression)
  statement
```

The *`statement`* is executed if the *`expression`* is true.

Form 2:

```
if (expression)
     statement1
else
     statement2
```

If the *expression* is true, *statement1* is executed. Otherwise, *statement2* is executed.

Form 3:

```
if (expression1)
     statement1
else if (expression2)
     statement2
else
     statement3
```

If *expression1* is true, *statement1* is executed. If *expression1* is false but *expression2* is true, *statement2* is executed. Otherwise, if both expressions are false, *statement3* is executed.

Example:

```
if (legs == 4)
     printf("It might be a horse.\n");
else if (legs > 4)
     printf("It is not a horse.\n");
else    /* case of legs < 4 */
{
     legs++;
     printf("Now it has one more leg.\n");
}
```

Let's Get Logical

You've seen how `if` and `while` statements often use relational expressions as tests. Sometimes you will find it useful to combine two or more relational expressions. For example, suppose you want a program that counts how many times the characters other than single or double quotes appear in an input sentence. You can use logical operators to meet this need, and you can use the period character (`.`) to identify the end of a sentence. Listing 7.6 presents a short program illustrating this method.

Listing 7.6 **The `chcount.c` Program**

```
// chcount.c  -- use the logical AND operator
#include <stdio.h>
#define PERIOD '.'
int main(void)
{
    char ch;
    int charcount = 0;

    while ((ch = getchar()) != PERIOD)
```

```
    {
        if (ch != '"' && ch != '\'')
            charcount++;
    }
    printf("There are %d non-quote characters.\n", charcount);

    return 0;
}
```

The following is a sample run:

```
I didn't read the "I'm a Programming Fool" best seller.
There are 50 non-quote characters.
```

The action begins as the program reads a character and checks to see whether it is a period, because the period marks the end of a sentence. Next comes something new, a statement using the logical AND operator, `&&`. You can translate the `if` statement as, "If the character is not a double quote AND if it is not a single quote, increase `charcount` by 1."

Both conditions must be true if the whole expression is to be true. The logical operators have a lower precedence than the relational operators, so it is not necessary to use additional parentheses to group the subexpressions.

C has three logical operators:

Operator	Meaning
`&&`	and
`||`	or
`!`	not

Suppose `exp1` and `exp2` are two simple relational expressions, such as `cat > rat` and `debt == 1000`. Then you can state the following:

- `exp1 && exp2` is true only if both `exp1` and `exp2` are true.
- `exp1 || exp2` is true if either `exp1` or `exp2` is true or if both are true.
- `!exp1` is true if `exp1` is false, and it's false if `exp1` is true.

Here are some concrete examples:

`5 > 2 && 4 > 7` is false because only one subexpression is true.

`5 > 2 || 4 > 7` is true because at least one of the subexpressions is true.

`!(4 > 7)` is true because `4` is not greater than `7`.

The last expression, incidentally, is equivalent to the following:

```
4 <= 7
```

If you are unfamiliar or uncomfortable with logical operators, remember that

```
(practice && time) == perfection
```

Alternate Spellings: The `iso646.h` Header File

C was developed in the United States on systems using the standard U.S. keyboards. But in the wider world, not all keyboards have the same symbols as U.S. keyboards do. Therefore, the C99 standard added alternative spellings for the logical operators. They are defined in the `iso646.h` header file. If you use this header file, you can use `and` instead of `&&`, `or` instead of `||`, and `not` instead of `!`. For example, you can rewrite

```
if (ch != '"' && ch != '\'')
    charcount++;
```

this way:

```
if (ch != '"' and ch != '\'')
    charcount++;
```

Table 7.3 lists your choices; they are pretty easy to remember. In fact, you might wonder why C didn't simply use the new terms. The answer probably is that C historically has tried to keep the number of keywords small. Reference Section V, "The Standard ANSI C Library with C99 and C11 Additions," lists additional alternative spellings for some operators you haven't met yet.

Table 7.3 **Alternative Representations of Logical Operators**

Traditional	**`iso646.h`**
`&&`	`and`
`||`	`or`
!	not

Precedence

The `!` operator has a very high precedence—higher than multiplication, the same as the increment operators, and just below that of parentheses. The `&&` operator has higher precedence than `||`, and both rank below the relational operators and above assignment in precedence. Therefore, the expression

```
a > b && b > c || b > d
```

would be interpreted as

```
((a > b) && (b > c)) || (b > d)
```

That is, `b` is between `a` and `c`, or `b` is greater than `d`.

Many programmers would use parentheses, as in the second version, even though they are not needed. That way, the meaning is clear even if the reader doesn't quite remember the precedence of the logical operators.

Order of Evaluation

Aside from those cases in which two operators share an operand, C ordinarily does not guarantee which parts of a complex expression are evaluated first. For example, in the following statement, the expression `5 + 3` might be evaluated before `9 + 6`, or it might be evaluated afterward:

```
apples = (5 + 3) * (9 + 6);
```

This ambiguity was left in the language so that compiler designers could make the most efficient choice for a particular system. One exception to this rule (or lack of rule) is the treatment of logical operators. C guarantees that logical expressions are evaluated from left to right. The `&&` and `||` operators are sequence points, so all side effects take place before a program moves from one operand to the next. Furthermore, it guarantees that as soon as an element is found that invalidates the expression as a whole, the evaluation stops. These guarantees make it possible to use constructions such as the following:

```
while ((c = getchar()) != ' ' && c != '\n')
```

This construction sets up a loop that reads characters up to the first space or newline character. The first subexpression gives a value to `c`, which then is used in the second subexpression. Without the order guarantee, the computer might try to test the second expression before finding out what value `c` has.

Here is another example:

```
if (number != 0 && 12/number == 2)
     printf("The number is 5 or 6.\n");
```

If `number` has the value `0`, the first subexpression is false, and the relational expression is not evaluated any further. This spares the computer the trauma of trying to divide by zero. Many languages do not have this feature. After seeing that `number` is 0, they still plunge ahead to check the next condition.

Finally, consider this example:

```
while ( x++ < 10 && x + y < 20)
```

The fact that the `&&` operator is a sequence point guarantees that `x` is incremented before the expression on the right is evaluated.

Summary: Logical Operators and Expressions

Logical Operators:

Logical operators normally take relational expressions as operands. The `!` operator takes one operand. The rest take two—one to the left, one to the right.

Operator	Meaning
`&&`	and
`||`	or
`!`	not

Logical Expressions:

`expression1 && expression2` is true if and only if both expressions are true. `expression1 || expression2` is true if either one or both expressions are true. `!expression` is true if the expression is false, and vice versa.

Order of Evaluation:

Logical expressions are evaluated from left to right. Evaluation stops as soon as something is discovered that renders the expression false.

Examples:

`6 > 2 && 3 == 3`	True.
`! (6 > 2 && 3 == 3)`	False.
`x != 0 && (20 / x) < 5`	The second expression is evaluated only if `x` is nonzero.

Ranges

You can use the `&&` operator to test for ranges. For example, to test for `score` being in the range 90 to 100, you can do this:

```
if (range >= 90 && range <= 100)
    printf("Good show!\n");
```

It's important to avoid imitating common mathematical notation, as in the following:

```
if (90 <= range <= 100)    // NO! Don't do it!
    printf("Good show!\n");
```

The problem is that the code is a semantic error, not a syntax error, so the compiler will not catch it (although it might issue a warning). Because the order of evaluation for the <= operator is left-to-right, the test expression is interpreted as follows:

```
(90 <= range) <= 100
```

The subexpression `90 <= range` either has the value `1` (for true) or `0` (for false). Either value is less than 100, so the whole expression is always true, regardless of the value of `range`. So use `&&` for testing for ranges.

A lot of code uses range tests to see whether a character is, say, a lowercase letter. For instance, suppose `ch` is a `char` variable:

```
if (ch >= 'a' && ch <= 'z')
    printf("That's a lowercase character.\n");
```

This works for character codes such as ASCII, in which the codes for consecutive letters are consecutive numbers. However, this is not true for some codes, including EBCDIC. The more portable way of doing this test is to use the `islower()` function from the `ctype.h` family (refer to Table 7.1):

```
if (islower(ch))
    printf("That's a lowercase character.\n");
```

The `islower()` function works regardless of the particular character code used. (However, some ancient implementations lack the `ctype.h` family.)

A Word-Count Program

Now you have the tools to make a word-counting program (that is, a program that reads input and reports the number of words it finds). You may as well count characters and lines while you are at it. Let's see what such a program involves.

First, the program should read input character-by-character, and it should have some way of knowing when to stop. Second, it should be able to recognize and count the following units: characters, lines, and words. Here's a pseudocode representation:

```
read a character
while there is more input
     increment character count
     if a line has been read, increment line count
     if a word has been read, increment word count
     read next character
```

You already have a model for the input loop:

```
while ((ch = getchar()) != STOP)
{
  ...
}
```

Here, `STOP` represents some value for `ch` that signals the end of the input. The examples so far have used the newline character and a period for this purpose, but neither is satisfactory for a general word-counting program. For the present, choose a character (such as `|`) that is not

common in text. In Chapter 8, "Character Input/Output and Input Validation," we'll present a better solution that also allows the program to be used with text files as well as keyboard input.

Now let's consider the body of the loop. Because the program uses `getchar()` for input, it can count characters by incrementing a counter during each loop cycle. To count lines, the program can check for newline characters. If a character is a newline, the program should increment the line count. One question to decide is what to do if the `STOP` character comes in the middle of a line. Should that count as a line or not? One answer is to count it as a partial line—that is, a line with characters but no newline. You can identify this case by keeping track of the previous character read. If the last character read before the `STOP` character isn't a newline, you have a partial line.

The trickiest part is identifying words. First, you have to define what you mean by a word. Let's take a relatively simple approach and define a word as a sequence of characters that contains no whitespace (that is, no spaces, tabs, or newlines). Therefore, "glymxck" and "r2d2" are words. A word starts when the program first encounters non-whitespace, and then it ends when the next whitespace character shows up. Here is the most straightforward test expression for detecting non-whitespace:

```
c != ' ' && c != '\n' && c != '\t'   /* true if c is not whitespace */
```

And the most straightforward test for detecting whitespace is

```
c == ' ' || c == '\n' || c == '\t'   /* true if c is whitespace */
```

However, it is simpler to use the `ctype.h` function `isspace()`, which returns true if its argument is a whitespace character. So `isspace(c)` is true if `c` is whitespace, and `!isspace(c)` is true if `c` isn't whitespace.

To keep track of whether a character is in a word, you can set a flag (call it `inword`) to `1` when the first character in a word is read. You can also increment the word count at that point. Then, as long as `inword` remains `1` (or true), subsequent non-whitespace characters don't mark the beginning of a word. At the next whitespace character, you must reset the flag to `0` (or false) and then the program will be ready to find the next word. Let's put that into pseudocode:

```
if c is not whitespace and inword is false
     set inword to true and count the word
if c is whitespace and inword is true
     set inword to false
```

This approach sets `inword` to `1` (true) at the beginning of each word and to `0` (false) at the end of each word. Words are counted only at the time the flag setting is changed from `0` to `1`. If you have the `_Bool` type available, you can include the `stdbool.h` header file and use `bool` for the `inword` type and `true` and `false` for the values. Otherwise, use the `int` type and `1` and `0` as the values.

If you do use a Boolean variable, the usual idiom is to use the value of the variable itself as a test condition. That is, use

```
if (inword)
```

instead of

```
if (inword == true)
```

and use

```
if (!inword)
```

instead of

```
if (inword == false)
```

The reasoning is that the expression `inword == true` evaluates to `true` if `inword` is `true` and to `false` if `inword` is `false`, so you may as well just use `inword` as the test. Similarly, `!inword` has the same value as the expression `inword == false` (not true is `false`, and not false is `true`).

Listing 7.7 translates these ideas (identifying lines, identifying partial lines, and identifying words) into C.

Listing 7.7 **The `wordcnt.c` Program**

```
// wordcnt.c -- counts characters, words, lines
#include <stdio.h>
#include <ctype.h>         // for isspace()
#include <stdbool.h>       // for bool, true, false
#define STOP '|'
int main(void)
{
    char c;                 // read in character
    char prev;              // previous character read
    long n_chars = 0L;      // number of characters
    int n_lines = 0;        // number of lines
    int n_words = 0;        // number of words
    int p_lines = 0;        // number of partial lines
    bool inword = false;    // == true if c is in a word

    printf("Enter text to be analyzed (| to terminate):\n");
    prev = '\n';            // used to identify complete lines
    while ((c = getchar()) != STOP)
    {
        n_chars++;          // count characters
        if (c == '\n')
            n_lines++;      // count lines
        if (!isspace(c) && !inword)
        {
            inword = true;  // starting a new word
            n_words++;      // count word
        }
```

```
        if (isspace(c) && inword)
            inword = false; // reached end of word
        prev = c;           // save character value
    }

    if (prev != '\n')
        p_lines = 1;
    printf("characters = %ld, words = %d, lines = %d, ",
          n_chars, n_words, n_lines);
    printf("partial lines = %d\n", p_lines);

    return 0;
}
```

Here is a sample run:

```
Enter text to be analyzed (| to terminate):
Reason is a
powerful servant but
an inadequate master.
|
characters = 55, words = 9, lines = 3, partial lines = 0
```

The program uses logical operators to translate the pseudocode to C. For example,

```
if c is not whitespace and inword is false
```

gets translated into the following:

```
if (!isspace(c) && !inword)
```

Note again that `!inword` is equivalent to `inword == false`. The entire test condition certainly is more readable than testing for each whitespace character individually:

```
if (c != ' ' && c != '\n' && c != '\t' && !inword)
```

Either form says, "If `c` is *not* whitespace *and* if you are *not* in a word." If both conditions are met, you must be starting a new word, and `n_words` is incremented. If you are in the middle of a word, the first condition holds, but `inword` will be `true`, and `n_words` is not incremented. When you reach the next whitespace character, `inword` is set equal to `false` again. Check the coding to see whether the program gets confused when there are several spaces between one word and the next. Chapter 8 shows how to modify this program to count words in a file.

The Conditional Operator: `?:`

C offers a shorthand way to express one form of the `if else` statement. It is called a *conditional expression* and uses the `?:` conditional operator. This is a two-part operator that has three operands. Recall that operators with one operand are called *unary* operators and that operators

with two operands are called *binary* operators. In that tradition, operators with three operands are called *ternary* operators, and the conditional operator is C's only example in that category. Here is an example that yields the absolute value of a number:

```
x = (y < 0) ? -y : y;
```

Everything between the = and the semicolon is the conditional expression. The meaning of the statement is "If `y` is less than zero, `x = -y`; otherwise, `x = y`." In `if else` terms, the meaning can be expressed as follows:

```
if (y < 0)
    x = -y;
else
    x = y;
```

The following is the general form of the conditional expression:

```
expression1 ? expression2 : expression3
```

If *`expression1`* is true (nonzero), the whole conditional expression has the same value as *`expression2`*. If *`expression1`* is `false` (zero), the whole conditional expression has the same value as *`expression3`*.

You can use the conditional expression when you have a variable to which you want to assign one of two possible values. A typical example is setting a variable equal to the maximum of two values:

```
max = (a > b) ? a : b;
```

This sets `max` to `a` if it is greater than `b`, and to `b` otherwise.

Usually, an `if else` statement can accomplish the same thing as the conditional operator. The conditional operator version, however, is more compact and, depending on the compiler, may result in more compact program code.

Let's look at a paint program example, shown in Listing 7.8. The program calculates how many cans of paint are needed to paint a given number of square feet. The basic algorithm is simple: Divide the square footage by the number of square feet covered per can. However, suppose the answer is 1.7 cans. Stores sell whole cans, not fractional cans, so you would have to buy two cans. Therefore, the program should round up to the next integer when a fractional paint can is involved. The conditional operator is used to handle that situation, and it's also used to print *cans* or *can*, as appropriate.

Listing 7.8 **The `paint.c` Program**

```
/* paint.c -- uses conditional operator */
#include <stdio.h>
#define COVERAGE 350       // square feet per paint can
int main(void)
{
```

```
    int sq_feet;
    int cans;

    printf("Enter number of square feet to be painted:\n");
    while (scanf("%d", &sq_feet) == 1)
    {
        cans = sq_feet / COVERAGE;
        cans += ((sq_feet % COVERAGE == 0)) ? 0 : 1;
        printf("You need %d %s of paint.\n", cans,
                cans == 1 ? "can" : "cans");
        printf("Enter next value (q to quit):\n");
    }

    return 0;
}
```

Here's a sample run:

```
Enter number of square feet to be painted:
349
You need 1 can of paint.
Enter next value (q to quit):
351
You need 2 cans of paint.
Enter next value (q to quit):
q
```

Because the program is using type `int`, the division is truncated; that is, `351/350` becomes `1`. Therefore, `cans` is rounded down to the integer part. If `sq_feet % COVERAGE` is 0, `COVERAGE` divides evenly into `sq_feet` and `cans` is left unchanged. Otherwise, there is a remainder, so `1` is added. This is accomplished with the following statement:

```
cans += ((sq_feet % COVERAGE == 0)) ? 0 : 1;
```

It adds the value of the expression to the right of `+=` to `cans`. The expression to the right is a conditional expression having the value `0` or `1`, depending on whether `COVERAGE` divides evenly into `sq_feet`.

The final argument to the `printf()` function is also a conditional expression:

```
cans == 1 ? "can" : "cans");
```

If the value of `cans` is `1`, the string `"can"` is used. Otherwise, `"cans"` is used. This demonstrates that the conditional operator can use strings for its second and third operands.

Summary: The Conditional Operator

The Conditional Operator:

`?:`

General Comments:

This operator takes three operands, each of which is an expression. They are arranged as follows:

expression1 `?` *expression2* `:` *expression3*

The value of the whole expression equals the value of *expression2* if *expression1* is true. Otherwise, it equals the value of *expression3*.

Examples:

`(5 > 3) ? 1 : 2` has the value `1`.

`(3 > 5) ? 1 : 2` has the value `2`.

`(a > b) ? a : b` has the value of the larger of `a` or `b`.

Loop Aids: `continue` and `break`

Normally, after the body of a loop has been entered, a program executes all the statements in the body before doing the next loop test. The `continue` and `break` statements enable you to skip part of a loop or even terminate it, depending on tests made in the body of the loop.

The `continue` Statement

This statement can be used in the three loop forms. When encountered, it causes the rest of an iteration to be skipped and the next iteration to be started. If the `continue` statement is inside nested structures, it affects only the innermost structure containing it. Let's try `continue` in the short program in Listing 7.9.

Listing 7.9 **The `skippart.c` Program**

```
/* skippart.c  -- uses continue to skip part of loop */
#include <stdio.h>
int main(void)
{
    const float MIN = 0.0f;
    const float MAX = 100.0f;

    float score;
    float total = 0.0f;
    int n = 0;
    float min = MAX;
```

```
    float max = MIN;

    printf("Enter the first score (q to quit): ");
    while (scanf("%f", &score) == 1)
    {
        if (score < MIN || score > MAX)
        {
            printf("%0.1f is an invalid value. Try again: ",
                    score);
            continue;  // jumps to while loop test condition
        }
        printf("Accepting %0.1f:\n", score);
        min = (score < min)? score: min;
        max = (score > max)? score: max;
        total += score;
        n++;
        printf("Enter next score (q to quit): ");
    }
    if (n > 0)
    {
        printf("Average of %d scores is %0.1f.\n", n, total / n);
        printf("Low = %0.1f, high = %0.1f\n", min, max);
    }
    else
        printf("No valid scores were entered.\n");
    return 0;
}
```

In Listing 7.9, the `while` loop reads input until you enter nonnumeric data. The `if` statement within the loop screens out invalid score values. If, say, you enter `188`, the program tells you `188 is an invalid value`. Then the `continue` statement causes the program to skip over the rest of the loop, which is devoted to processing valid input. Instead, the program starts the next loop cycle by attempting to read the next input value.

Note that there are two ways you could have avoided using `continue`. One way is omitting the `continue` and making the remaining part of the loop an `else` block:

```
if (score < 0 || score > 100)
    /* printf() statement */
else
{
    /* statements */
}
```

Alternatively, you could have used this format instead:

```
if (score >= 0 && score <= 100)
{
```

```
    /* statements */
}
```

An advantage of using `continue` in this case is that you can eliminate one level of indentation in the main group of statements. Being concise can enhance readability when the statements are long or are deeply nested already.

Another use for `continue` is as a placeholder. For example, the following loop reads and discards input up to, and including, the end of a line:

```
while (getchar() != '\n')
    ;
```

Such a technique is handy when a program has already read some input from a line and needs to skip to the beginning of the next line. The problem is that the lone semicolon is hard to spot. The code is much more readable if you use `continue`:

```
while (getchar() != '\n')
    continue;
```

Don't use `continue` if it complicates rather than simplifies the code. Consider the following fragment, for example:

```
while ((ch = getchar() ) != '\n')
{
    if (ch == '\t')
        continue;
    putchar(ch);
}
```

This loop skips over the tabs and quits only when a newline character is encountered. The loop could have been expressed more economically as this:

```
while ((ch = getchar()) != '\n')
    if (ch != '\t')
        putchar(ch);
```

Often, as in this case, reversing an `if` test eliminates the need for a `continue`.

You've seen that the `continue` statement causes the remaining body of a loop to be skipped. Where exactly does the loop resume? For the `while` and `do while` loops, the next action taken after the `continue` statement is to evaluate the loop test expression. Consider the following loop, for example:

```
count = 0;
while (count < 10)
{
    ch = getchar();
    if (ch == '\n')
```

```
        continue;
    putchar(ch);
    count++;
}
```

It reads 10 characters (excluding newlines, because the `count++;` statement gets skipped when `ch` is a newline) and echoes them, except for newlines. When the `continue` statement is executed, the next expression evaluated is the loop test condition.

For a `for` loop, the next actions are to evaluate the update expression and then the loop test expression. Consider the following loop, for example:

```
for (count = 0; count < 10; count++)
{
    ch = getchar();
    if (ch == '\n')
        continue;
    putchar(ch);
}
```

In this case, when the `continue` statement is executed, first `count` is incremented and then it's compared to `10`. Therefore, this loop behaves slightly differently from the `while` example. As before, only non-newline characters are displayed. However, this time, newline characters are included in the count, so it reads 10 characters, including newlines.

The `break` Statement

A `break` statement in a loop causes the program to break free of the loop that encloses it and to proceed to the next stage of the program. In Listing 7.9, replacing `continue` with `break` would cause the loop to quit when, say, 188 is entered, instead of just skipping to the next loop cycle. Figure 7.3 compares `break` and `continue`. If the `break` statement is inside nested loops, it affects only the innermost loop containing it.

Sometimes `break` is used to leave a loop when there are two separate reasons to leave. Listing 7.10 uses a loop that calculates the area of a rectangle. The loop terminates if you respond with nonnumeric input for the rectangle's length or width.

Listing 7.10 **The `break.c` Program**

```
/* break.c -- uses break to exit a loop */
#include <stdio.h>
int main(void)
{
    float length, width;

    printf("Enter the length of the rectangle:\n");
    while (scanf("%f", &length) == 1)
```

```
    {
        printf("Length = %0.2f:\n", length);
        printf("Enter its width:\n");
        if (scanf("%f", &width) != 1)
            break;
        printf("Width = %0.2f:\n", width);
        printf("Area = %0.2f:\n", length * width);
        printf("Enter the length of the rectangle:\n");
    }
    printf("Done.\n");

    return 0;
}
```

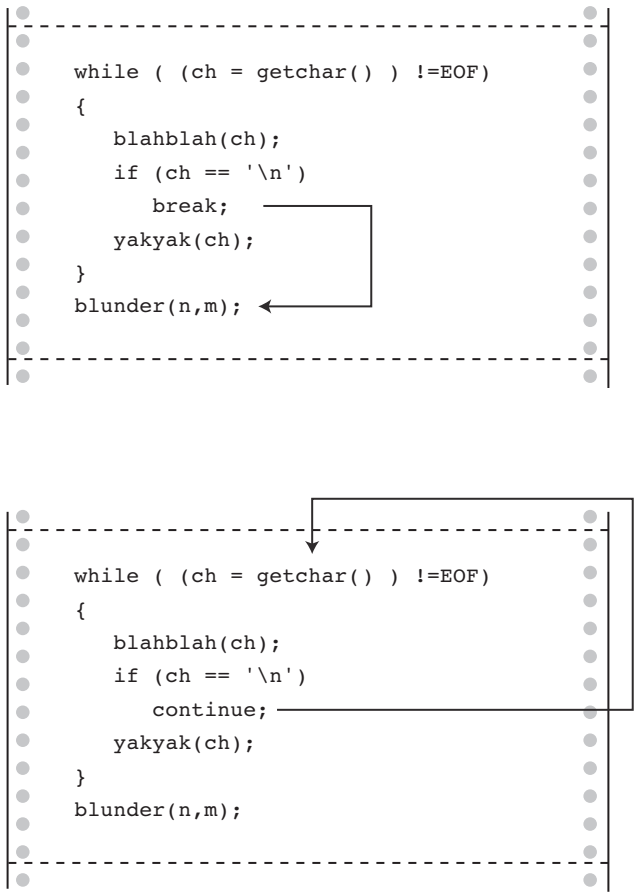


Figure 7.3 Comparing `break` and `continue`.

You could have controlled the loop this way:

```
while (scanf("%f %f", &length, &width) == 2)
```

However, using `break` makes it simple to echo each input value individually.

As with `continue`, don't use `break` when it complicates code. For example, consider the following loop:

```
while ((ch = getchar()) != '\n')
{
   if (ch == '\t')
         break;
   putchar(ch);
}
```

The logic is clearer if both tests are in the same place:

```
while ((ch = getchar() ) != '\n' && ch != '\t')
      putchar(ch);
```

The `break` statement is an essential adjunct to the `switch` statement, which is coming up next.

A `break` statement takes execution directly to the first statement following the loop; unlike the case for `continue` in a `for` loop, the update part of the control section is skipped. A `break` in a nested loop just takes the program out of the inner loop; to get out of the outer loop requires a second `break`:

```
int p, q;

scanf("%d", &p);
while ( p > 0)
{
    printf("%d\n", p);
    scanf("%d", &q);
    while( q > 0)
    {
        printf("%d\n",p*q);
        if (q > 100)
            break;          // break from inner loop
        scanf("%d", &q);
    }
    if (q > 100)
        break;              // break from outer loop
    scanf("%d", &p);
}
```

Multiple Choice: `switch` and `break`

The conditional operator and the `if else` construction make it easy to write programs that choose between two alternatives. Sometimes, however, a program needs to choose one of several alternatives. You can do this by using `if else if...else`. However, in many cases, it is more convenient to use the C `switch` statement. Listing 7.11 shows how the `switch` statement works. This program reads in a letter and then responds by printing an animal name that begins with that letter.

Listing 7.11 **The `animals.c` Program**

```
/* animals.c -- uses a switch statement */
#include <stdio.h>
#include <ctype.h>
int main(void)
{
    char ch;

    printf("Give me a letter of the alphabet, and I will give ");
    printf("an animal name\nbeginning with that letter.\n");
    printf("Please type in a letter; type # to end my act.\n");
    while ((ch = getchar()) != '#')
    {
        if('\n' == ch)
            continue;
        if (islower(ch))     /* lowercase only          */
            switch (ch)
            {
                case 'a' :
                      printf("argali, a wild sheep of Asia\n");
                      break;
                case 'b' :
                      printf("babirusa, a wild pig of Malay\n");
                      break;
                case 'c' :
                      printf("coati, racoonlike mammal\n");
                      break;
                case 'd' :
                      printf("desman, aquatic, molelike critter\n");
                      break;
                case 'e' :
                      printf("echidna, the spiny anteater\n");
                      break;
                case 'f' :
                      printf("fisher, brownish marten\n");
                      break;
```

```
                default :
                      printf("That's a stumper!\n");
            }               /* end of switch           */
        else
            printf("I recognize only lowercase letters.\n");
        while (getchar() != '\n')
              continue;     /* skip rest of input line */
        printf("Please type another letter or a #.\n");
    }                       /* while loop end          */
    printf("Bye!\n");

    return 0;
}
```

We got a little lazy and stopped at *f*, but we could have continued in the same manner. Let's look at a sample run before explaining the program further:

```
Give me a letter of the alphabet, and I will give an animal name
beginning with that letter.
Please type in a letter; type # to end my act.
a [enter]
argali, a wild sheep of Asia
Please type another letter or a #.
dab [enter]
desman, aquatic, molelike critter
Please type another letter or a #.
r [enter]
That's a stumper!
Please type another letter or a #.
Q [enter]
I recognize only lowercase letters.
Please type another letter or a #.
# [enter]
Bye!
```

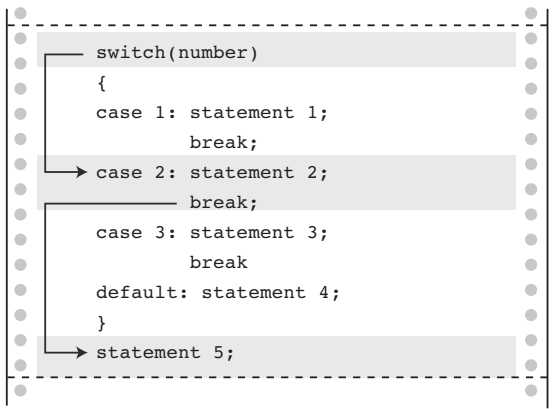
The program's two main features are its use of the `switch` statement and its handling of input. We'll look first at how `switch` works.

Using the `switch` Statement

The expression in the parentheses following the word `switch` is evaluated. In this case, it has whatever value you last entered for `ch`. Then the program scans the list of *labels* (here, `case 'a' :`, `case 'b' :`, and so on) until it finds one matching that value. The program then jumps to that line. What if there is no match? If there is a line labeled `default :`, the program jumps there. Otherwise, the program proceeds to the statement following the `switch`.

What about the `break` statement? It causes the program to break out of the `switch` and skip to the next statement after the `switch` (see Figure 7.4). Without the `break` statement, every statement from the matched label to the end of the `switch` would be processed. For example, if you removed all the `break` statements from the program and then ran the program using the letter *d*, you would get this exchange:

```
Give me a letter of the alphabet, and I will give an animal name
beginning with that letter.
Please type in a letter; type # to end my act.
d [enter]
desman, aquatic, molelike critter
echidna, the spiny anteater
fisher, a brownish marten
That's a stumper!
Please type another letter or a #.
# [enter]
Bye!
```



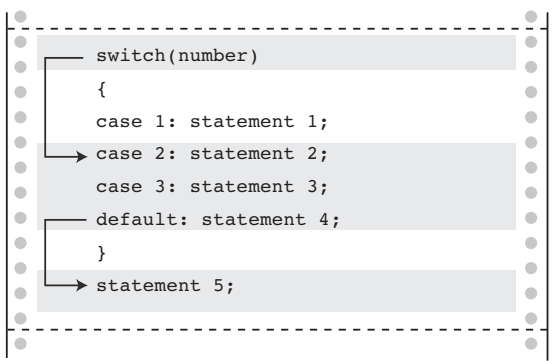


Figure 7.4 Program flow in `switch`es, with and without `break`s.

All the statements from `case 'd' :` to the end of the `switch` were executed.

Incidentally, a `break` statement works with loops and with `switch`, but `continue` works just with loops. However, `continue` can be used as part of a `switch` statement if the statement is in a loop. In that situation, as with other loops, `continue` causes the program to skip over the rest of the loop, including other parts of the `switch`.

If you are familiar with Pascal, you will recognize the `switch` statement as being similar to the Pascal `case` statement. The most important difference is that the `switch` statement requires the use of a `break` if you want only the labeled statement to be processed. Also, you can't use a range as a C case.

The `switch` test expression in the parentheses should be one with an integer value (including type `char`). The `case` labels must be integer-type (including `char`) constants or integer constant expressions (expressions containing only integer constants). You can't use a variable for a `case` label. Here, then, is the structure of a `switch`:

```
switch (integer expression)
{
     case constant1:
              statements    <--optional
     case constant2:
              statements    <--optional
     default :              <--optional
              statements    <--optional
}
```

Reading Only the First Character of a Line

The other new feature incorporated into `animals.c` is how it reads input. As you might have noticed in the sample run, when `dab` was entered, only the first character was processed. This behavior of disposing of the rest of the line is often desirable in interactive programs looking for single-character responses. The following code produced this behavior:

```
while (getchar() != '\n')
    continue;         /* skip rest of input line */
```

This loop reads characters from input up to and including the newline character generated by the Enter key. Note that the function return value is not assigned to `ch`, so the characters are merely read and discarded. Because the last character discarded is the newline character, the next character to be read is the first character of the next line. It gets read by `getchar()` and assigned to `ch` in the outer `while` loop.

Suppose a user starts out by pressing Enter so that the first character encountered is a newline. The following code takes care of that possibility:

```
if (ch == '\n')
    continue;
```

Multiple Labels

You can use multiple `case` labels for a given statement, as shown in Listing 7.12.

Listing 7.12 **The `vowels.c` Program**

```
// vowels.c -- uses multiple labels
#include <stdio.h>
int main(void)
{
    char ch;
    int a_ct, e_ct, i_ct, o_ct, u_ct;

    a_ct = e_ct = i_ct = o_ct = u_ct = 0;

    printf("Enter some text; enter # to quit.\n");
    while ((ch = getchar()) != '#')
    {
        switch (ch)
        {
            case 'a' :
            case 'A' :  a_ct++;
                        break;
            case 'e' :
            case 'E' :  e_ct++;
                        break;
            case 'i' :
            case 'I' :  i_ct++;
                        break;
            case 'o' :
            case 'O' :  o_ct++;
                        break;
            case 'u' :
            case 'U' :  u_ct++;
                        break;
            default :   break;
          }                     // end of switch
    }                           // while loop end
    printf("number of vowels:   A    E    I    O    U\n");
    printf("                %4d %4d %4d %4d %4d\n",
          a_ct, e_ct, i_ct, o_ct, u_ct);

    return 0;
}
```

If `ch` is, say, the letter `i`, the `switch` statement goes to the location labeled `case 'i' :`. Because there is no `break` associated with that label, program flow goes to the next statement, which is `i_ct++;`. If `ch` is `I`, program flow goes directly to that statement. In essence, both labels refer to the same statement.

Strictly speaking, the `break` statement for `case 'U'` isn't needed, because in its absence, program flow goes to the next statement in the `switch`, which is the `break` for the `default` case. So the `case 'U'` break could be dropped, thus shortening the code. On the other hand, if other cases might be added later (you might want to count the letter *y* as a sometimes vowel), having the `break` already in place protects you from forgetting to add one.

Here's a sample run:

```
Enter some text; enter # to quit.
I see under the overseer.#
number of vowels:   A    E    I    O    U
                    0    7    1    1    1
```

In this particular case, you can avoid multiple labels by using the `toupper()` function from the `ctype.h` family (refer to Table 7.2) to convert all letters to uppercase before testing:

```
while ((ch = getchar()) != '#')
{
    ch = toupper(ch);
    switch (ch)
    {
      case 'A' :  a_ct++;
                  break;
      case 'E' :  e_ct++;
                  break;
      case 'I' :  i_ct++;
                  break;
      case 'O' :  o_ct++;
                  break;
      case 'U' :  u_ct++;
                  break;
      default :   break;
    }                           // end of switch
}                               // while loop end
```

Or, if you want to, you could leave `ch` unchanged and use `toupper(ch)` as the test condition:

```
switch(toupper(ch))
```

Summary: Multiple Choice with *switch*

Keyword:

`switch`

General Comments:

Program control jumps to the `case` label bearing the value of *expression*. Program flow then proceeds through all the remaining statements unless redirected again with a `break` statement. Both *expression* and `case` labels must have integer values (type `char` is included), and the labels must be constants or expressions formed solely from constants. If no `case` label matches the expression value, control goes to the statement labeled `default`, if present. Otherwise, control passes to the next statement following the `switch` statement.

Form:

```
switch (expression)
{
    case label1 : statement1 // use break to skip to end
    case label2 : statement2
    default     : statement3
}
```

There can be more than two labeled statements, and the `default` case is optional.

Example:

```
switch (choice)
    {
    case 1  :
    case 2  : printf("Darn tootin'!\n");  break;
    case 3  : printf("Quite right!\n");
    case 4  : printf("Good show!\n"); break;
    default : printf("Have a nice day.\n");
    }
```

If `choice` has the integer value `1` or `2`, the first message is printed. If it is `3`, the second and third messages are printed. (Flow continues to the following statement because there is no `break` statement after `case 3`.) If it is `4`, the third message is printed. Other values print only the last message.

`switch` and `if else`

When should you use a `switch` and when should you use the `if else` construction? Often you don't have a choice. You can't use a `switch` if your choice is based on evaluating a floating-point variable or expression. Nor can you conveniently use a `switch` if a variable must fall into a certain range. It is simple to write the following:

```
if (integer < 1000 && integer > 2)
```

Unhappily, covering this range with a `switch` would involve setting up `case` labels for each integer from 3 to 999. However, if you can use a `switch`, your program often runs a little faster and takes less code.

The `goto` Statement

The `goto` statement, bulwark of the older versions of BASIC and FORTRAN, is available in C. However, C, unlike those two languages, can get along quite well without it. Kernighan and Ritchie refer to the `goto` statement as "infinitely abusable" and suggest that it "be used sparingly, if at all." First, we will show you how to use `goto`. Then, we will show why you usually don't need to.

The `goto` statement has two parts—the `goto` and a label name. The label is named following the same convention used in naming a variable, as in this example:

```
goto part2;
```

For the preceding statement to work, the function must contain another statement bearing the `part2` label. This is done by beginning a statement with the label name followed by a colon:

```
part2: printf("Refined analysis:\n");
```

Avoiding `goto`

In principle, you never need to use the `goto` statement in a C program, but if you have a background in older versions of FORTRAN or BASIC, both of which require its use, you might have developed programming habits that depend on using `goto`. To help you get over that dependence, we will outline some familiar `goto` situations and then show you a more C-like approach:

- Handling an `if` situation that requires more than one statement:

  ```
  if (size > 12)
      goto a;
  goto b;
  a: cost = cost * 1.05;
  flag = 2;
  b: bill = cost * flag;
  ```

  In old-style BASIC and FORTRAN, only the single statement immediately following the `if` condition is attached to the `if`. No provision is made for blocks or compound statements. We have translated that pattern into the equivalent C. The standard C approach of using a compound statement or block is much easier to follow:

  ```
  if (size > 12)
  {
      cost = cost * 1.05;
      flag = 2;
  ```

```
}
bill = cost * flag;
```

- Choosing from two alternatives:

```
if (ibex > 14)
    goto a;
sheds = 2;
goto b;
a: sheds= 3;
b: help = 2 * sheds;
```

  Having the `if else` structure available allows C to express this choice more cleanly:

```
if (ibex > 14)
    sheds = 3;
else
    sheds = 2;
help = 2 * sheds;
```

  Indeed, newer versions of BASIC and FORTRAN have incorporated `else` into their syntax.

- Setting up an indefinite loop:

```
readin: scanf("%d", &score);
if (score < 0)
    goto stage2;
lots of statements
goto readin;
stage2: more stuff;
```

  Use a `while` loop instead:

```
scanf("%d", &score);
while (score <= 0)
{
    lots of statements
    scanf("%d", &score);
}
more stuff;
```

- Skipping to the end of a loop and starting the next cycle. Use `continue` instead.
- Leaving a loop. Use `break` instead. Actually, `break` and `continue` are specialized forms of `goto`. The advantages of using them are that their names tell you what they are supposed to do and that, because they don't use labels, there is no danger of putting a label in the wrong place.
- Leaping madly about to different parts of a program. In a word, *don't!*

There is a use of goto tolerated by many C practitioners—getting out of a nested set of loops if trouble shows up (a single break gets you out of the innermost loop only):

```
while (funct > 0)
    {
    for (i = 1, i <= 100; i++)
        {
        for (j = 1; j <= 50; j++)
            {
            statements galore;
            if (bit trouble)
                goto help;
            statements;
            }
        more statements;
        }
    yet more statements;
    }
and more statements;
help : bail out;
```

As you can see from the other examples, the alternative forms are clearer than the goto forms. This difference grows even greater when you mix several of these situations. Which gotos are helping ifs, which are simulating if elses, which are controlling loops, which are just there because you have programmed yourself into a corner? By using gotos excessively, you create a labyrinth of program flow. If you aren't familiar with gotos, keep it that way. If you are used to using them, try to train yourself not to. Ironically, C, which doesn't need a goto, has a better goto than most languages because it enables you to use descriptive words for labels instead of numbers.

Summary: Program Jumps

Keywords:

break, continue, goto

General Comments:

These three instructions cause program flow to jump from one location of a program to another location.

The break Command:

The break command can be used with any of the three loop forms and with the switch statement. It causes program control to skip the rest of the loop or the switch containing it and to resume with the next command following the loop or switch.

Example:

```
switch (number)
{
     case 4:  printf("That's a good choice.\n");
```

```
            break;
     case 5:  printf("That's a fair choice.\n");
            break;
     default: printf("That's a poor choice.\n");
}
```

The `continue` Command:

The `continue` command can be used with any of the three loop forms but not with a `switch`. It causes program control to skip the remaining statements in a loop. For a `while` or `for` loop, the next loop cycle is started. For a `do while` loop, the exit condition is tested and then, if necessary, the next loop cycle is started.

Example:

```
while ((ch = getchar())  != '\n')
{
    if (ch == ' ')
        continue;
    putchar(ch);
    chcount++;
}
```

This fragment echoes and counts non-space characters.

The `goto` Command:

A `goto` statement causes program control to jump to a statement bearing the indicated label. A colon is used to separate a labeled statement from its label. Label names follow the rules for variable names. The labeled statement can come either before or after the `goto`.

Form:

```
goto label;
    .
    .
    .
label : statement
```

Example:

```
top : ch = getchar();
      .
      .
      .
if (ch != 'y')
    goto top;
```

Key Concepts

One aspect of intelligence is the ability to adjust one's responses to the circumstances. Therefore, selection statements are the foundation for developing programs that behave intelligently. In C, the `if`, `if else`, and `switch` statements, along with the conditional operator (`?:`), implement selection.

The `if` and `if else` statements use a test condition to determine which statements are executed. Any nonzero value is treated as `true`, whereas zero is treated as `false`. Typically, tests involve relational expressions, which compare two values, and logical expressions, which use logical operators to combine or modify other expressions.

One general principle to keep in mind is that if you want to test for two conditions, you should use a logical operator together with two complete test expressions. For instance, the following two attempts are faulty:

```
if (a < x < z)           // wrong --no logical operator
...
if (ch != 'q' && != 'Q') // wrong -- missing a complete test
...
```

Remember, the correct way is to join two relational expressions with a logical operator:

```
if (a < x && x < z)          // use && to combine two expressions
...
if (ch != 'q' && ch != 'Q')  // use && to combine two expressions
...
```

The control statements presented in these last two chapters will enable you to tackle programs that are much more powerful and ambitious than those you worked with before. For evidence, just compare some of the examples in these chapters to those of the earlier chapters.

Summary

This chapter has presented quite a few topics to review, so let's get to it. The `if` statement uses a test condition to control whether a program executes the single simple statement or block following the test condition. Execution occurs if the test expression has a nonzero value and doesn't occur if the value is zero. The `if else` statement enables you to select from two alternatives. If the test condition is nonzero, the statement before the `else` is executed. If the test expression evaluates to zero, the statement following the `else` is executed. By using another `if` statement to immediately follow the `else`, you can set up a structure that chooses between a series of alternatives.

The test condition is often a *relational expression*—that is, an expression formed by using one of the relational operators, such as < or ==. By using C's logical operators, you can combine relational expressions to create more complex tests.

The *conditional operator* (`? :`) creates an expression that, in many cases, provides a more compact alternative to an `if else` statement.

The `ctype.h` family of character functions, such as `isspace()` and `isalpha()`, offers convenient tools for creating test expressions based on classifying characters.

The `switch` statement enables you to select from a series of statements labeled with integer values. If the integer value of the test condition following the `switch` keyword matches a label, execution goes to the statement bearing that label. Execution then proceeds through the statements following the labeled statement unless you use a `break` statement.

Finally, `break`, `continue`, and `goto` are jump statements that cause program flow to jump to another location in the program. A `break` statement causes the program to jump to the next statement following the end of the loop or `switch` containing the `break`. The `continue` statement causes the program to skip the rest of the containing loop and to start the next cycle.

Review Questions

You'll find answers to the review questions in Appendix A, "Answers to the Review Questions."

1. Determine which expressions are `true` and which are `false`.
   a. `100 > 3 && 'a'>'c'`
   b. `100 > 3 || 'a'>'c'`
   c. `!(100>3)`

2. Construct an expression to express the following conditions:
   a. `number` is equal to or greater than 90 but smaller than 100.
   b. `ch` is not a `q` or a `k` character.
   c. `number` is between 1 and 9 (including the end values) but is not a 5.
   d. `number` is not between 1 and 9.

3. The following program has unnecessarily complex relational expressions as well as some outright errors. Simplify and correct it.

```
#include <stdio.h>
int main(void)                                        /* 1  */
{                                                     /* 2  */
  int weight, height;  /* weight in lbs, height in inches */
                                                      /* 4  */
  scanf("%d , weight, height);                        /* 5  */
  if (weight < 100 && height > 64)                    /* 6  */
     if (height >= 72)                                /* 7  */
        printf("You are very tall for your weight.\n");
```

```
   else if (height < 72 &&  > 64)                /* 9  */
      printf("You are tall for your weight.\n");  /* 10 */
  else if (weight > 300 && ! (weight <= 300)       /* 11 */
           && height < 48)                         /* 12 */
   if (!(height >= 48) )                          /* 13 */
       printf(" You are quite short for your weight.\n");
  else                                            /* 15 */
   printf("Your weight is ideal.\n");             /* 16 */
                                                  /* 17 */
  return 0;
}
```

4. What is the numerical value of each of the following expressions?

   a. `5 > 2`

   b. `3 + 4 > 2 && 3 < 2`

   c. `x >= y || y > x`

   d. `d = 5 + ( 6 > 2 )`

   e. `'X' > 'T' ? 10 : 5`

   f. `x > y ? y > x : x > y`

5. What will the following program print?

```
#include <stdio.h>
int main(void)
{
  int num;
  for (num = 1; num <= 11; num++)
  {
       if (num % 3 == 0)
            putchar('$');
       else
            putchar('*');
            putchar('#');
       putchar('%');
  }
  putchar('\n');
  return 0;
}
```

6. What will the following program print?

```
#include <stdio.h>
int main(void)
```

```
{
    int i = 0;
    while ( i < 3) {
       switch(i++) {
           case 0 : printf("fat ");
           case 1 : printf("hat ");
           case 2 : printf("cat ");
           default: printf("Oh no!");
       }
       putchar('\n');
    }
    return 0;
}
```

7. What's wrong with this program?

```
#include <stdio.h>
int main(void)
{
  char ch;
  int lc = 0;    /* lowercase char count
  int uc = 0;    /* uppercase char count
  int oc = 0;    /* other char count

  while ((ch = getchar()) != '#')
  {
       if ('a' <= ch >= 'z')
            lc++;
       else if (!(ch < 'A') || !(ch > 'Z')
            uc++;
       oc++;
  }
  printf(%d lowercase, %d uppercase, %d other, lc, uc, oc);
  return 0;
}
```

8. What will the following program print?

```
/* retire.c   */
#include <stdio.h>
int main(void)
{
  int age = 20;
```

```
  while (age++ <= 65)
  {
     if (( age % 20) == 0) /* is age divisible by 20? */
         printf("You are %d. Here is a raise.\n", age);
     if (age = 65)
         printf("You are %d. Here is your gold watch.\n", age);
  }
  return 0;
}
```

9. What will the following program print when given this input?

```
q
c
h
b
#include <stdio.h>
int main(void)
{
  char ch;

  while ((ch = getchar()) != '#')
  {
       if (ch == '\n')
            continue;
       printf("Step 1\n");
       if (ch == 'c')
            continue;
       else if (ch == 'b')
            break;
       else if (ch == 'h')
            goto laststep;
       printf("Step 2\n");
  laststep:  printf("Step 3\n");
  }
  printf("Done\n");
  return 0;
}
```

10. Rewrite the program in Review Question 9 so that it exhibits the same behavior but does not use a `continue` or a `goto`.

Programming Exercises

1. Write a program that reads input until encountering the # character and then reports the number of spaces read, the number of newline characters read, and the number of all other characters read.

2. Write a program that reads input until encountering #. Have the program print each input character and its ASCII decimal code. Print eight character-code pairs per line. Suggestion: Use a character count and the modulus operator (`%`) to print a newline character for every eight cycles of the loop.

3. Write a program that reads integers until 0 is entered. After input terminates, the program should report the total number of even integers (excluding the 0) entered, the average value of the even integers, the total number of odd integers entered, and the average value of the odd integers.

4. Using `if else` statements, write a program that reads input up to #, replaces each period with an exclamation mark, replaces each exclamation mark initially present with two exclamation marks, and reports at the end the number of substitutions it has made.

5. Redo exercise 4 using a `switch`.

6. Write a program that reads input up to # and reports the number of times that the sequence `ei` occurs.

> **Note**
>
> The program will have to "remember" the preceding character as well as the current character. Test it with input such as "Receive your eieio award."

7. Write a program that requests the hours worked in a week and then prints the gross pay, the taxes, and the net pay. Assume the following:

   a. Basic pay rate = $10.00/hr

   b. Overtime (in excess of 40 hours) = time and a half

   c. Tax rate: #15% of the first $300

   20% of the next $150

   25% of the rest

   Use `#define` constants, and don't worry if the example does not conform to current tax law.

8. Modify assumption a. in exercise 7 so that the program presents a menu of pay rates from which to choose. Use a `switch` to select the pay rate. The beginning of a run should look something like this:

```
*************************************************************
Enter the number corresponding to the desired pay rate or action:
1) $8.75/hr                          2) $9.33/hr
3) $10.00/hr                         4) $11.20/hr
5) quit
*************************************************************
```

   If choices 1 through 4 are selected, the program should request the hours worked. The program should recycle until 5 is entered. If something other than choices 1 through 5 is entered, the program should remind the user what the proper choices are and then recycle. Use `#defined` constants for the various earning rates and tax rates.

9. Write a program that accepts a positive integer as input and then displays all the prime numbers smaller than or equal to that number.

10. The 1988 United States Federal Tax Schedule was the simplest in recent times. It had four categories, and each category had two rates. Here is a summary (dollar amounts are taxable income):

Category	Tax
Single	15% of first $17,850 plus 28% of excess
Head of Household	15% of first $23,900 plus 28% of excess
Married, Joint	15% of first $29,750 plus 28% of excess
Married, Separate	15% of first $14,875 plus 28% of excess

    For example, a single wage earner with a taxable income of $20,000 owes 0.15 × $17,850 + 0.28 × ($20,000–$17,850). Write a program that lets the user specify the tax category and the taxable income and that then calculates the tax. Use a loop so that the user can enter several tax cases.

11. The ABC Mail Order Grocery sells artichokes for $2.05 per pound, beets for $1.15 per pound, and carrots for $1.09 per pound. It gives a 5% discount for orders of $100 or more prior to adding shipping costs. It charges $6.50 shipping and handling for any order of 5 pounds or under, $14.00 shipping and handling for orders over 5 pounds and under 20 pounds, and $14.00 plus $0.50 per pound for shipments of 20 pounds or more. Write a program that uses a `switch` statement in a loop such that a response of `a` lets the user enter the pounds of artichokes desired, `b` the pounds of beets, `c` the pounds of carrots, and `q` allows the user to exit the ordering process. The program should keep track of cumulative totals. That is, if the user enters 4 pounds of beets and later enters 5 pounds of beets, the program should use report 9 pounds of beets. The program then

should compute the total charges, the discount, if any, the shipping charges, and the grand total. The program then should display all the purchase information: the cost per pound, the pounds ordered, and the cost for that order for each vegetable, the total cost of the order, the discount (if there is one), the shipping charge, and the grand total of all the charges.