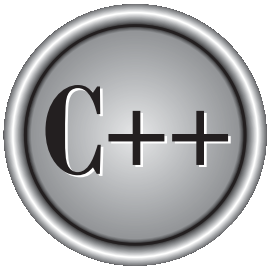
The Complete Reference



Chapter 3

Statements

This chapter discusses the statement. In the most general sense, a *statement* is a part of your program that can be executed. That is, a statement specifies an action. C and C++ categorize statements into these groups:

- Selection
- Iteration
- Jump
- Label
- Expression
- Block

Included in the selection statements are **if** and **switch**. (The term *conditional statement* is often used in place of "selection statement.") The iteration statements are **while**, **for**, and **do-while**. These are also commonly called *loop statements*. The jump statements are **break**, **continue**, **goto**, and **return**. The label statements include the **case** and **default** statements (discussed along with the **switch** statement) and the label statement (discussed with **goto**). Expression statements are statements composed of a valid expression. Block statements are simply blocks of code. (Remember, a block begins with a { and ends with a }.) Block statements are also referred to as *compound statements*.

Note *C++ adds two additional statement types: the **try** block (used by exception handling) and the declaration statement. These are discussed in Part Two.*

Since many statements rely upon the outcome of some conditional test, let's begin by reviewing the concepts of true and false.

True and False in C and C++

Many C/C++ statements rely upon a conditional expression that determines what course of action is to be taken. A conditional expression evaluates to either a true or false value. In C, a true value is any nonzero value, including negative numbers. A false value is 0. This approach to true and false allows a wide range of routines to be coded extremely efficiently.

C++ fully supports the zero/nonzero definition of true and false just described. But C++ also defines a Boolean data type called **bool,** which can have only the values **true** and **false**. As explained in Chapter 2, in C++, a 0 value is automatically converted into **false** and a nonzero value is automatically converted into **true**. The reverse also applies: **true** converts to 1 and **false** converts to 0. In C++, the expression that controls a conditional statement is technically of type **bool**. But since any nonzero value converts to **true** and any zero value converts to **false**, there is no practical difference between C and C++ on this point.

Note *C99 has added a Boolean type called* ***_Bool****, but it is incompatible with C++. See Part Two for a discussion on how to achieve compatibility between C99's* ***_Bool*** *and C++'s* ***bool*** *types.*

Selection Statements

C/C++ supports two types of selection statements: **if** and **switch**. In addition, the **?** operator is an alternative to **if** in certain circumstances.

if

The general form of the **if** statement is

if (*expression*) *statement*;
else *statement*;

where a *statement* may consist of a single statement, a block of statements, or nothing (in the case of empty statements). The **else** clause is optional.

If *expression* evaluates to true (anything other than 0), the statement or block that forms the target of **if** is executed; otherwise, the statement or block that is the target of **else** will be executed, if it exists. Remember, only the code associated with **if** or the code associated with **else** executes, never both.

In C, the conditional statement controlling **if** must produce a scalar result. A *scalar* is either an integer, character, pointer, or floating-point type. In C++, it may also be of type **bool**. It is rare to use a floating-point number to control a conditional statement because this slows execution time considerably. (It takes several instructions to perform a floating-point operation. It takes relatively few instructions to perform an integer or character operation.)

The following program contains an example of **if**. The program plays a very simple version of the "guess the magic number" game. It prints the message ** **Right** ** when the player guesses the magic number. It generates the magic number using the standard random number generator **rand( )**, which returns an arbitrary number between 0 and **RAND_MAX** (which defines an integer value that is 32,767 or larger). **rand( )** requires the header file **stdlib.h**. (A C++ program may also use the new-style header **<cstdlib>**.)

```
/* Magic number program #1. */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int magic; /* magic number */
```

```
  int guess; /* user's guess */

  magic = rand(); /* generate the magic number */

  printf("Guess the magic number: ");
  scanf("%d", &guess);

  if(guess == magic) printf("** Right **");

  return 0;
}
```

Taking the magic number program further, the next version illustrates the use of the **else** statement to print a message in response to the wrong number.

```
/* Magic number program #2. */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int magic; /* magic number */
  int guess; /* user's guess */

  magic = rand(); /* generate the magic number */

  printf("Guess the magic number: ");
  scanf("%d", &guess);

  if(guess == magic) printf("** Right **");
  else printf("Wrong");

  return 0;
}
```

Nested ifs

A nested **if** is an **if** that is the target of another **if** or **else**. Nested **if**s are very common in programming. In a nested **if**, an **else** statement always refers to the nearest **if** statement that is within the same block as the **else** and that is not already associated with an **else**. For example,

```
if(i)
{
  if(j) statement 1;
  if(k) statement 2; /* this if */
  else  statement 3; /* is associated with this else */
}
else statement 4; /* associated with if(i) */
```

As noted, the final **else** is not associated with **if(j)** because it is not in the same block. Rather, the final **else** is associated with **if(i)**. Also, the inner **else** is associated with **if(k)**, which is the nearest **if**.

The C language guarantees at least 15 levels of nesting. In practice, most compilers allow substantially more. More importantly, Standard C++ suggests that at least 256 levels of nested **if**s be allowed in a C++ program. However, nesting beyond a few levels is seldom necessary, and excessive nesting can quickly confuse the meaning of an algorithm.

You can use a nested **if** to further improve the magic number program by providing the player with feedback about a wrong guess.

```
/* Magic number program #3. */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int magic; /* magic number */
  int guess; /* user's guess */

  magic = rand(); /* get a random number */

  printf("Guess the magic number: ");
  scanf("%d", &guess);

  if (guess == magic) {
    printf("** Right **");
    printf(" %d is the magic number\n", magic);
  }
  else {
    printf("Wrong, ");
    if(guess > magic) printf("too high\n");
    else printf("too low\n");
  }
```

```
  return 0;
}
```

The if-else-if Ladder

A common programming construct is the *if-else-if ladder*, sometimes called the *if-else-if staircase* because of its appearance. Its general form is

```
if (expression) statement;
else
  if (expression) statement;
  else
    if (expression) statement;
    .
    .
    .
    else statement;
```

The conditions are evaluated from the top downward. As soon as a true condition is found, the statement associated with it is executed and the rest of the ladder is bypassed. If none of the conditions are true, the final **else** is executed. That is, if all other conditional tests fail, the last **else** statement is performed. If the final **else** is not present, no action takes place if all other conditions are false.

Although the indentation of the preceding if-else-if ladder is technically correct, it can lead to overly deep indentation. For this reason, the if-else-if ladder is generally indented like this:

```
if (expression)
  statement;
else if (expression)
  statement;
else if (expression)
  statement;
.
.
.
else
  statement;
```

Using an if-else-if ladder, the magic number program becomes

```
/* Magic number program #4. */
#include <stdio.h>
```

```
#include <stdlib.h>

int main(void)
{
  int magic; /* magic number */
  int guess; /* user's guess */

  magic = rand(); /* generate the magic number */

  printf("Guess the magic number: ");
  scanf("%d", &guess);

  if(guess == magic) {
    printf("** Right ** ");
    printf("%d is the magic number", magic);
  }
  else if(guess > magic)
    printf("Wrong, too high");
  else printf("Wrong, too low");

  return 0;
}
```

The ? Alternative

You can use the **?** operator to replace **if-else** statements of the general form:

if(*condition*) *expression*;
else *expression*;

However, the target of both **if** and **else** must be a single expression—not another statement.

The **?** is called a *ternary operator* because it requires three operands. It takes the general form

Exp1 ? Exp2 : Exp3

where *Exp1*, *Exp2*, and *Exp3* are expressions. Notice the use and placement of the colon.

The value of a **?** expression is determined as follows: *Exp1* is evaluated. If it is true, *Exp2* is evaluated and becomes the value of the entire **?** expression. If *Exp1* is false, then *Exp3* is evaluated and its value becomes the value of the expression. For example, consider

```
x = 10;
y = x>9 ? 100 : 200;
```

In this example, **y** is assigned the value 100. If **x** had been less than 9, **y** would have received the value 200. The same code written with the **if-else** statement would be

```
x = 10;
if(x>9) y = 100;
else y = 200;
```

The following program uses the **?** operator to square an integer value entered by the user. However, this program preserves the sign (10 squared is 100 and –10 squared is –100).

```
#include <stdio.h>

int main(void)
{
  int isqrd, i;

  printf("Enter a number: ");
  scanf("%d", &i);

  isqrd = i>0 ? i*i : -(i*i);

  printf("%d squared is %d", i, isqrd);

  return 0;
}
```

The use of the **?** operator to replace **if-else** statements is not restricted to assignments only. Remember, all functions (except those declared as **void**) return a value. Thus, you can use one or more function calls in a **?** expression. When the function's name is encountered, the function is executed so that its return value may be determined. Therefore, you can execute one or more function calls using the **?** operator by placing the calls in the expressions that form the **?**'s operands. Here is an example.

```
#include <stdio.h>

int f1(int n);
int f2(void);
```

```
int main(void)
{
  int t;

  printf("Enter a number: ");
  scanf("%d", &t);

  /* print proper message */
  t ? f1(t) + f2() : printf("zero entered.\n");

  return 0;
}

int f1(int n)
{
  printf("%d ", n);
  return 0;
}

int f2(void)
{
  printf("entered.\n");
  return 0;
}
```

Entering a 0 in this example calls the **printf( )** function and displays the message **zero entered**. If you enter any other number, both **f1( )** and **f2( )** execute. Note that the value of the **?** expression is discarded in this example. You don't need to assign it to anything.

A word of warning: Some C++ compilers rearrange the order of evaluation of an expression in an attempt to optimize the object code. This could cause functions that form the operands of the **?** operator to execute in an unintended sequence.

Using the **?** operator, you can rewrite the magic number program yet again.

```
/* Magic number program #5. */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int magic;
  int guess;
```

```
  magic = rand(); /* generate the magic number */

  printf("Guess the magic number: ");
  scanf("%d", &guess);

  if(guess == magic) {
    printf("** Right ** ");
    printf("%d is the magic number", magic);
  }
  else
    guess > magic ? printf("High") : printf("Low");

  return 0;
}
```

Here, the **?** operator displays the proper message based on the outcome of the test **guess > magic**.

The Conditional Expression

Sometimes newcomers to C/C++ are confused by the fact that you can use any valid expression to control the **if** or the **?** operator. That is, you are not restricted to expressions involving the relational and logical operators (as is the case in languages like BASIC or Pascal). The expression must simply evaluate to either a true or false (zero or nonzero) value. For example, the following program reads two integers from the keyboard and displays the quotient. It uses an **if** statement, controlled by the second number, to avoid a divide-by-zero error.

```
/* Divide the first number by the second. */

#include <stdio.h>

int main(void)
{
  int a, b;

  printf("Enter two numbers: ");
  scanf("%d%d", &a, &b);

  if(b) printf("%d\n", a/b);
  else printf("Cannot divide by zero.\n");

  return 0;
}
```

This approach works because if **b** is 0, the condition controlling the **if** is false and the **else** executes. Otherwise, the condition is true (nonzero) and the division takes place.

One other point: Writing the **if** statement as shown here

```
if(b != 0) printf("%d\n", a/b);
```

is redundant, potentially inefficient, and is considered bad style. Since the value of **b** alone is sufficient to control the **if**, there is no need to test it against 0.

switch

C/C++ has a built-in multiple-branch selection statement, called **switch**, which successively tests the value of an expression against a list of integer or character constants. When a match is found, the statements associated with that constant are executed. The general form of the **switch** statement is

```
switch (expression) {
  case constant1:
    statement sequence
    break;
  case constant2:
    statement sequence
    break;
  case constant3:
    statement sequence
    break;
  .
  .
  .
  default
    statement sequence
}
```

The *expression* must evaluate to a character or integer value. Floating-point expressions, for example, are not allowed. The value of *expression* is tested, in order, against the values of the constants specified in the **case** statements. When a match is found, the statement sequence associated with that **case** is executed until the **break** statement or the end of the **switch** statement is reached. The **default** statement is executed if no matches are found. The **default** is optional and, if it is not present, no action takes place if all matches fail.

In C, a **switch** can have at least 257 **case** statements. Standard C++ recommends that *at least* 16,384 **case** statements be supported! In practice, you will want to limit the number of **case** statements to a smaller amount for efficiency. Although **case** is a label statement, it cannot exist by itself, outside of a **switch**.

The **break** statement is one of C/C++'s jump statements. You can use it in loops as well as in the **switch** statement (see the section "Iteration Statements"). When **break** is encountered in a **switch**, program execution "jumps" to the line of code following the **switch** statement.

There are three important things to know about the **switch** statement:

- The **switch** differs from the **if** in that **switch** can only test for equality, whereas **if** can evaluate any type of relational or logical expression.
- No two **case** constants in the same **switch** can have identical values. Of course, a **switch** statement enclosed by an outer **switch** may have **case** constants that are the same.
- If character constants are used in the **switch** statement, they are automatically converted to integers.

The **switch** statement is often used to process keyboard commands, such as menu selection. As shown here, the function **menu( )** displays a menu for a spelling-checker program and calls the proper procedures:

```
void menu(void)
{
  char ch;

  printf("1. Check Spelling\n");
  printf("2. Correct Spelling Errors\n");
  printf("3. Display Spelling Errors\n");
  printf("Strike Any Other Key to Skip\n");
  printf("      Enter your choice: ");

  ch = getchar(); /* read the selection from
                     the keyboard */

  switch(ch) {
    case '1':
      check_spelling();
      break;
    case '2':
      correct_errors();
      break;
    case '3':
      display_errors();
      break;
    default :
```

```
      printf("No option selected");
  }
}
```

Technically, the **break** statements inside the **switch** statement are optional. They terminate the statement sequence associated with each constant. If the **break** statement is omitted, execution will continue on into the next **case**'s statements until either a **break** or the end of the **switch** is reached. For example, the following function uses the "drop through" nature of the **case**s to simplify the code for a device-driver input handler:

```
/* Process a value */
void inp_handler(int i)
{
  int flag;

  flag = -1;

  switch(i) {
    case 1:  /* These cases have common */
    case 2:  /* statement sequences. */
    case 3:
      flag = 0;
      break;
    case 4:
      flag = 1;
    case 5:
      error(flag);
      break;
    default:
      process(i);
  }
}
```

This example illustrates two aspects of **switch**. First, you can have **case** statements that have no statement sequence associated with them. When this occurs, execution simply drops through to the next **case**. In this example, the first three **case**s all execute the same statements, which are

```
flag = 0;
break;
```

Second, execution of one statement sequence continues into the next **case** if no **break** statement is present. If **i** matches 4, **flag** is set to 1 and, because there is no **break** statement at the end of that **case**, execution continues and the call to **error(flag)** is executed. If **i** had matched 5, **error(flag)** would have been called with a flag value of –1 (rather than 1).

The fact that **case**s can run together when no **break** is present prevents the unnecessary duplication of statements, resulting in more efficient code.

Nested switch Statements

You can have a **switch** as part of the statement sequence of an outer **switch**. Even if the **case** constants of the inner and outer **switch** contain common values, no conflicts arise. For example, the following code fragment is perfectly acceptable:

```
switch(x) {
  case 1:
    switch(y) {
      case 0: printf("Divide by zero error.\n");
              break;
      case 1: process(x,y);
    }
    break;
  case 2:
    .
    .
    .
```

Iteration Statements

In C/C++, and all other modern programming languages, iteration statements (also called *loops*) allow a set of instructions to be executed repeatedly until a certain condition is reached. This condition may be predefined (as in the **for** loop), or open-ended (as in the **while** and **do-while** loops).

The for Loop

The general design of the **for** loop is reflected in some form or another in all procedural programming languages. However, in C/C++, it provides unexpected flexibility and power.

The general form of the **for** statement is

for(*initialization*; *condition*; *increment*) *statement*;

The **for** loop allows many variations, but its most common form works like this. The *initialization* is an assignment statement that is used to set the loop control variable. The

condition is a relational expression that determines when the loop exits. The *increment* defines how the loop control variable changes each time the loop is repeated. You must separate these three major sections by semicolons. The **for** loop continues to execute as long as the condition is true. Once the condition becomes false, program execution resumes on the statement following the **for**.

In the following program, a **for** loop is used to print the numbers 1 through 100 on the screen:

```
#include <stdio.h>

int main(void)
{
  int x;

  for(x=1; x <= 100; x++) printf("%d ", x);

  return 0;
}
```

In the loop, **x** is initially set to 1 and then compared with 100. Since **x** is less than 100, **printf( )** is called and the loop iterates. This causes **x** to be increased by 1 and again tested to see if it is still less than or equal to 100. If it is, **printf( )** is called. This process repeats until **x** is greater than 100, at which point the loop terminates. In this example, **x** is the loop control variable, which is changed and checked each time the loop repeats.

The following example is a **for** loop that iterates multiple statements:

```
for(x=100; x != 65; x -= 5) {
  z = x*x;
  printf("The square of %d, %f", x, z);
}
```

Both the squaring of **x** and the call to **printf( )** are executed until **x** equals 65. Note that the loop is *negative running*: **x** is initialized to 100 and 5 is subtracted from it each time the loop repeats.

In **for** loops, the conditional test is always performed at the top of the loop. This means that the code inside the loop may not be executed at all if the condition is false to begin with. For example, in

```
x = 10;
for(y=10; y!=x; ++y) printf("%d", y);
printf("%d", y);  /* this is the only printf()
                     statement that will execute */
```

the loop will never execute because **x** and **y** are equal when the loop is entered. Because this causes the conditional expression to evaluate to false, neither the body of the loop nor the increment portion of the loop executes. Hence, **y** still has the value 10, and the only output produced by the fragment is the number 10 printed once on the screen.

for Loop Variations

The previous discussion described the most common form of the **for** loop. However, several variations of the **for** are allowed that increase its power, flexibility, and applicability to certain programming situations.

One of the most common variations uses the comma operator to allow two or more variables to control the loop. (Remember, you use the comma operator to string together a number of expressions in a "do this and this" fashion. See Chapter 2.) For example, the variables **x** and **y** control the following loop, and both are initialized inside the **for** statement:

```
for(x=0, y=0; x+y<10; ++x) {
  y = getchar();
  y = y - '0'; /* subtract the ASCII code for 0
                  from y */
    .
    .
    .
}
```

Commas separate the two initialization statements. Each time the loop repeats, **x** is incremented and **y**'s value is set by keyboard input. Both **x** and **y** must be at the correct value for the loop to terminate. Even though **y**'s value is set by keyboard input, **y** must be initialized to 0 so that its value is defined before the first evaluation of the conditional expression. (If **y** were not defined, it could by chance contain the value 10, making the conditional test false and preventing the loop from executing.)

The **converge( )** function, shown next, demonstrates multiple loop control variables in action. The **converge( )** function copies the contents of one string into another by moving characters from both ends, converging in the middle.

```
/* Demonstrate multiple loop control variables. */
#include <stdio.h>
#include <string.h>

void converge(char *targ, char *src);

int main(void)
{
```

```
  char target[80] = "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";

  converge(target, "This is a test of converge().");
  printf("Final string: %s\n", target);

  return 0;
}

/* This function copies one string into another.
   It copies characters to both the ends,
   converging at the middle. */
void converge(char *targ, char *src)
{
  int i, j;

  printf("%s\n", targ);
  for(i=0, j=strlen(src); i<=j; i++, j--) {
    targ[i] = src[i];
    targ[j] = src[j];
    printf("%s\n", targ);
  }
}
```

Here is the output produced by the program.

```
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
TXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
ThXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX.
ThiXXXXXXXXXXXXXXXXXXXXXXXXXXXX).
ThisXXXXXXXXXXXXXXXXXXXXXXXXXX().
This XXXXXXXXXXXXXXXXXXXXXXe().
This iXXXXXXXXXXXXXXXXXXXge().
This isXXXXXXXXXXXXXXXXXrge().
This is XXXXXXXXXXXXXXerge().
This is aXXXXXXXXXXXXverge().
This is a XXXXXXXXXXnverge().
This is a tXXXXXXXXonverge().
This is a teXXXXXXconverge().
This is a tesXXXX converge().
This is a testXXf converge().
This is a test of converge().
Final string: This is a test of converge().
```

In **converge( )**, the **for** loop uses two loop control variables, **i** and **j**, to index the string from opposite ends. As the loop iterates, **i** is increased and **j** is decreased. The loop stops when **i** is greater than **j**, thus ensuring that all characters are copied.

The conditional expression does not have to involve testing the loop control variable against some target value. In fact, the condition may be any relational or logical statement. This means that you can test for several possible terminating conditions.

For example, you could use the following function to log a user onto a remote system. The user has three tries to enter the password. The loop terminates when the three tries are used up or the user enters the correct password.

```
void sign_on(void)
{
  char str[20] = "";
  int x;

  for(x=0; x<3 && strcmp(str, "password"); ++x) {
    printf("Enter password please:");
    gets(str);
  }

  if(x==3) return;
  /* else log user in ... */
}
```

This function uses **strcmp( )**, the standard library function that compares two strings and returns 0 if they match.

Remember, each of the three sections of the **for** loop may consist of any valid expression. The expressions need not actually have anything to do with what the sections are generally used for. With this in mind, consider the following example:

```
#include <stdio.h>

int sqrnum(int num);
int readnum(void);
int prompt(void);

int main(void)
{
  int t;
```

```
  for(prompt(); t=readnum(); prompt())
    sqrnum(t);

  return 0;
}

int prompt(void)
{
  printf("Enter a number: ");
  return 0;
}

int readnum(void)
{
  int t;

  scanf("%d", &t);
  return t;
}

int sqrnum(int num)
{
  printf("%d\n", num*num);
  return num*num;
}
```

Look closely at the **for** loop in **main( )**. Notice that each part of the **for** loop is composed of function calls that prompt the user and read a number entered from the keyboard. If the number entered is 0, the loop terminates because the conditional expression will be false. Otherwise, the number is squared. Thus, this **for** loop uses the initialization and increment portions in a nontraditional but completely valid sense.

Another interesting trait of the **for** loop is that pieces of the loop definition need not be there. In fact, there need not be an expression present for any of the sections—the expressions are optional. For example, this loop will run until the user enters **123**:

```
for(x=0; x!=123; ) scanf("%d", &x);
```

Notice that the increment portion of the **for** definition is blank. This means that each time the loop repeats, **x** is tested to see if it equals 123, but no further action takes place. If you type **123** at the keyboard, however, the loop condition becomes false and the loop terminates.

The initialization of the loop control variable can occur outside the **for** statement. This most frequently happens when the initial condition of the loop control variable must be computed by some complex means as in this example:

```
gets(s);  /* read a string into s */
if(*s) x = strlen(s); /* get the string's length */
else x = 10;

for( ; x<10; ) {
  printf("%d", x);
  ++x;
}
```

The initialization section has been left blank and **x** is initialized before the loop is entered.

The Infinite Loop

Although you can use any loop statement to create an infinite loop, **for** is traditionally used for this purpose. Since none of the three expressions that form the **for** loop are required, you can make an endless loop by leaving the conditional expression empty:

```
for( ; ; ) printf("This loop will run forever.\n");
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C++ programmers more commonly use the **for(;;)** construct to signify an infinite loop.

Actually, the **for(;;)** construct does not guarantee an infinite loop because a **break** statement, encountered anywhere inside the body of a loop, causes immediate termination. (**break** is discussed in detail later in this chapter.) Program control then resumes at the code following the loop, as shown here:

```
ch = '\0';

for( ; ; ) {
  ch = getchar(); /* get a character */
  if(ch=='A') break; /* exit the loop */
}

printf("you typed an A");
```

This loop will run until the user types an **A** at the keyboard.

for Loops with No Bodies

A statement may be empty. This means that the body of the **for** loop (or any other loop) may also be empty. You can use this fact to improve the efficiency of certain algorithms and to create time delay loops.

Removing spaces from an input stream is a common programming task. For example, a database program may allow a query such as "show all balances less than 400." The database needs to have each word fed to it separately, without leading spaces. That is, the database input processor recognizes "**show**" but not " **show**". The following loop shows one way to accomplish this. It advances past leading spaces in the string pointed to by **str**.

```
for( ; *str == ' '; str++) ;
```

As you can see, this loop has no body—and no need for one either.

Time delay loops are often used in programs. The following code shows how to create one by using **for**:

```
for(t=0; t<SOME_VALUE; t++) ;
```

The while Loop

The second loop available in C/C++ is the **while** loop. Its general form is

while(*condition*) *statement*;

where *statement* is either an empty statement, a single statement, or a block of statements. The *condition* may be any expression, and true is any nonzero value. The loop iterates while the condition is true. When the condition becomes false, program control passes to the line of code immediately following the loop.

The following example shows a keyboard input routine that simply loops until the user types **A**:

```
char wait_for_char(void)
{
  char ch;

  ch = '\0';  /* initialize ch */
  while(ch != 'A') ch = getchar();
  return ch;
}
```

First, **ch** is initialized to null. As a local variable, its value is not known when **wait_for_char( )** is executed. The **while** loop then checks to see if **ch** is not equal to **A**. Because **ch** was initialized to null, the test is true and the loop begins. Each time you press a key, the condition is tested again. Once you enter an **A**, the condition becomes false because **ch** equals **A**, and the loop terminates.

Like **for** loops, **while** loops check the test condition at the top of the loop, which means that the body of the loop will not execute if the condition is false to begin with. This feature may eliminate the need to perform a separate conditional test before the loop. The **pad( )** function provides a good illustration of this. It adds spaces to the end of a string to fill the string to a predefined length. If the string is already at the desired length, no spaces are added.

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

void pad(char *s, int length);

int main(void)
{
  char str[80];

  strcpy(str, "this is a test");
  pad(str, 40);
  printf("%d", strlen(str));

  return 0;
}

/* Add spaces to the end of a string. */
void pad(char *s, int length)
{
  int l;

  l = strlen(s); /* find out how long it is */

  while(l<length) {
    s[l] = ' '; /* insert a space */
    l++;
  }
  s[l]= '\0'; /* strings need to be
                 terminated in a null */
}
```

The two arguments of **pad( )** are **s**, a pointer to the string to lengthen, and **length**, the number of characters that **s** should have. If the length of string **s** is already equal to or greater than **length**, the code inside the **while** loop does not execute. If **s** is shorter than **length**, **pad( )** adds the required number of spaces. The **strlen( )** function, part of the standard library, returns the length of the string.

If several separate conditions need to terminate a **while** loop, a single variable commonly forms the conditional expression. The value of this variable is set at various points throughout the loop. In this example,

```
void func1(void)
{
  int working;

  working = 1; /* i.e., true */

  while(working) {
    working = process1();
    if(working)
      working = process2();
    if(working)
      working = process3();
  }
}
```

any of the three routines may return false and cause the loop to exit.

There need not be any statements in the body of the **while** loop. For example,

```
while((ch=getchar()) != 'A') ;
```

will simply loop until the user types **A**. If you feel uncomfortable putting the assignment inside the **while** conditional expression, remember that the equal sign is just an operator that evaluates to the value of the right-hand operand.

The do-while Loop

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do-while** loop checks its condition at the bottom of the loop. This means that a **do-while** loop always executes at least once. The general form of the **do-while** loop is

```
do {
  statement;
} while(condition);
```

Although the curly braces are not necessary when only one statement is present, they are usually used to avoid confusion (to you, not the compiler) with the **while**. The **do-while** loop iterates until *condition* becomes false.

The following **do-while** loop will read numbers from the keyboard until it finds a number less than or equal to 100.

```
do {
  scanf("%d", &num);
} while(num > 100);
```

Perhaps the most common use of the **do-while** loop is in a menu selection function. When the user enters a valid response, it is returned as the value of the function. Invalid responses cause a reprompt. The following code shows an improved version of the spelling-checker menu developed earlier in this chapter:

```
void menu(void)
{
  char ch;

  printf("1. Check Spelling\n");
  printf("2. Correct Spelling Errors\n");
  printf("3. Display Spelling Errors\n");
  printf("      Enter your choice: ");

  do {
    ch = getchar(); /* read the selection from
                       the keyboard */
    switch(ch) {
      case '1':
        check_spelling();
        break;
      case '2':
        correct_errors();
        break;
      case '3':
        display_errors();
        break;
    }
  } while(ch!='1' && ch!='2' && ch!='3');
}
```

Here, the **do-while** loop is a good choice because you will always want a menu function to display the menu at least once. After the options have been displayed, the program will loop until a valid option is selected.

Declaring Variables within Selection and Iteration Statements

In C++ (but not C89), it is possible to declare a variable within the conditional expression of an **if** or **switch**, within the conditional expression of a **while** loop, or within the initialization portion of a **for** loop. A variable declared in one of these places has its scope limited to the block of code controlled by that statement. For example, a variable declared within a **for** loop will be local to that loop.

Here is an example that declares a variable within the initialization portion of a **for** loop:

```
/* i is local to for loop; j is known outside loop. */
int j;
for(int i = 0; i<10; i++)
  j = i * i;

/* i = 10; // *** Error *** -- i not known here! */
```

Here, **i** is declared within the initialization portion of the **for** and is used to control the loop. Outside the loop, **i** is unknown.

Since often a loop control variable in a **for** is needed only by that loop, the declaration of the variable in the initialization portion of the **for** is becoming common practice. Remember, however, that this is not supported by C89. (This restriction was removed from C by C99.)

Tip *Whether a variable declared within the initialization portion of a **for** loop is local to that loop has changed over time. Originally, the variable was available after the **for**. However, Standard C++ restricts the variable to the scope of the **for** loop as just described.*

If your compiler fully complies with Standard C++, then you can also declare a variable within any conditional expression, such as those used by the **if** or a **while**. For example, this fragment,

```
if(int x = 20) {
  x = x - y;
```

```
  if(x>10) y = 0;
}
```

declares **x** and assigns it the value 20. Since this is a true value, the target of the **if** executes. Variables declared within a conditional statement have their scope limited to the block of code controlled by that statement. Thus, in this case, **x** is not known outside the **if**. Frankly, not all programmers believe that declaring variables within conditional statements is good practice, and this technique will not be used in this book.

Jump Statements

C/C++ has four statements that perform an unconditional branch: **return**, **goto**, **break**, and **continue**. Of these, you may use **return** and **goto** anywhere in your program. You may use the **break** and **continue** statements in conjunction with any of the loop statements. As discussed earlier in this chapter, you can also use **break** with **switch**.

The return Statement

The **return** statement is used to return from a function. It is categorized as a jump statement because it causes execution to return (jump back) to the point at which the call to the function was made. A **return** may or may not have a value associated with it. If **return** has a value associated with it, that value becomes the return value of the function. In C89, a non-**void** function does not technically have to return a value. If no return value is specified, a garbage value is returned. However, in C++ (and in C99), a non-**void** function *must* return a value. That is, in C++, if a function is specified as returning a value, any **return** statement within it must have a value associated with it. (Even in C89, if a function is declared as returning a value, it is good practice to actually return one!)

The general form of the **return** statement is

return *expression*;

The *expression* is present only if the function is declared as returning a value. In this case, the value of *expression* will become the return value of the function.

You can use as many **return** statements as you like within a function. However, the function will stop executing as soon as it encounters the first **return**. The } that ends a function also causes the function to return. It is the same as a **return** without any specified value. If this occurs within a non-**void** function, then the return value of the function is undefined.

A function declared as **void** may not contain a **return** statement that specifies a value. Since a **void** function has no return value, it makes sense that no **return** statement within a **void** function can return a value.

See Chapter 6 for more information on **return**.

The goto Statement

Since C/C++ has a rich set of control structures and allows additional control using **break** and **continue**, there is little need for **goto**. Most programmers' chief concern about the **goto** is its tendency to render programs unreadable. Nevertheless, although the **goto** statement fell out of favor some years ago, it occasionally has its uses. There are no programming situations that require **goto**. Rather, it is a convenience, which, if used wisely, can be a benefit in a narrow set of programming situations, such as jumping out of a set of deeply nested loops. The **goto** is not used outside of this section.

The **goto** statement requires a label for operation. (A *label* is a valid identifier followed by a colon.) Furthermore, the label must be in the same function as the **goto** that uses it—you cannot jump between functions. The general form of the **goto** statement is

goto *label*;
.
.
.
label:

where *label* is any valid label either before or after **goto**. For example, you could create a loop from 1 to 100 using the **goto** and a label, as shown here:

```
x = 1;
loop1:
  x++;
  if(x<100) goto loop1;
```

The break Statement

The **break** statement has two uses. You can use it to terminate a **case** in the **switch** statement (covered in the section on **switch** earlier in this chapter). You can also use it to force immediate termination of a loop, bypassing the normal loop conditional test.

When the **break** statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop. For example,

```
#include <stdio.h>

int main(void)
{
  int t;
```

```
  for(t=0; t<100; t++) {
    printf("%d ", t);
    if(t==10) break;
  }

  return 0;
}
```

prints the numbers 0 through 10 on the screen. Then the loop terminates because **break** causes immediate exit from the loop, overriding the conditional test **t<100**.

Programmers often use the **break** statement in loops in which a special condition can cause immediate termination. For example, here a keypress can stop the execution of the **look_up( )** function:

```
void look_up(char *name)
{
  do {
    /* look up names ... */
    if(kbhit()) break;
  } while(!found);
  /* process match */
}
```

The **kbhit( )** function returns 0 if you do not press a key. Otherwise, it returns a nonzero value. Because of the wide differences between computing environments, neither Standard C nor Standard C++ defines **kbhit( ),** but you will almost certainly have it (or one with a slightly different name) supplied with your compiler.

A **break** causes an exit from only the innermost loop. For example,

```
for(t=0; t<100; ++t) {
  count = 1;
  for(;;) {
    printf("%d ", count);
    count++;
    if(count==10) break;
  }
}
```

prints the numbers 1 through 10 on the screen 100 times. Each time execution encounters **break**, control is passed back to the outer **for** loop.

A **break** used in a **switch** statement will affect only that **switch**. It does not affect any loop the **switch** happens to be in.

The exit( ) Function

Although **exit( )** is not a program control statement, a short digression that discusses it is in order at this time. Just as you can break out of a loop, you can break out of a program by using the standard library function **exit( )**. This function causes immediate termination of the entire program, forcing a return to the operating system. In effect, the **exit( )** function acts as if it were breaking out of the entire program.

The general form of the **exit( )** function is

void exit(int *return_code*);

The value of *return_code* is returned to the calling process, which is usually the operating system. Zero is generally used as a return code to indicate normal program termination. Other arguments are used to indicate some sort of error. You can also use the macros **EXIT_SUCCESS** and **EXIT_FAILURE** for the *return_code*. The **exit( )** function requires the header **stdlib.h**. A C++ program may also use the C++-style header **<cstdlib>**.

Programmers frequently use **exit( )** when a mandatory condition for program execution is not satisfied. For example, imagine a virtual reality computer game that requires a special graphics adapter. The **main( )** function of this game might look like this:

```
#include <stdlib.h>

int main(void)
{
   if(!virtual_graphics()) exit(1);
   play();
   /* ... */
}
/* .... */
```

where **virtual_graphics( )** is a user-defined function that returns true if the virtual-reality graphics adapter is present. If the adapter is not in the system, **virtual_graphics( )** returns false and the program terminates.

As another example, this version of **menu( )** uses **exit( )** to quit the program and return to the operating system:

```
void menu(void)
{
```

```
  char ch;

  printf("1. Check Spelling\n");
  printf("2. Correct Spelling Errors\n");
  printf("3. Display Spelling Errors\n");
  printf("4. Quit\n");
  printf("      Enter your choice: ");

  do {
    ch = getchar(); /* read the selection from
                       the keyboard */
      switch(ch) {
        case '1':
          check_spelling();
          break;
        case '2':
          correct_errors();
          break;
        case '3':
          display_errors();
          break;
        case '4':
          exit(0); /* return to OS */
      }
    } while(ch!='1' && ch!='2' && ch!='3');
  }
```

The continue Statement

The **continue** statement works somewhat like the **break** statement. Instead of forcing termination, however, **continue** forces the next iteration of the loop to take place, skipping any code in between. For the **for** loop, **continue** causes the conditional test and increment portions of the loop to execute. For the **while** and **do-while** loops, program control passes to the conditional tests. For example, the following program counts the number of spaces contained in the string entered by the user:

```
/* Count spaces */
#include <stdio.h>

int main(void)
{
```

```
  char s[80], *str;
  int space;

  printf("Enter a string: ");
  gets(s);
  str = s;

  for(space=0; *str; str++) {
    if(*str != ' ') continue;
    space++;
  }
  printf("%d spaces\n", space);

  return 0;
}
```

Each character is tested to see if it is a space. If it is not, the **continue** statement forces the **for** to iterate again. If the character *is* a space, **space** is incremented.

The following example shows how you can use **continue** to expedite the exit from a loop by forcing the conditional test to be performed sooner:

```
void code(void)
{
  char done, ch;

  done = 0;
  while(!done) {
    ch = getchar();
    if(ch=='$') {
      done = 1;
      continue;
    }
    putchar(ch+1); /* shift the alphabet one
                      position higher */
  }
}
```

This function codes a message by shifting all characters you type one letter higher. For example, an **A** becomes a **B**. The function will terminate when you type a **$**. After a **$** has been input, no further output will occur because the conditional test, brought into effect by **continue**, will find **done** to be true and will cause the loop to exit.

Expression Statements

Chapter 2 covered expressions thoroughly. However, a few special points are mentioned here. Remember, an expression statement is simply a valid expression followed by a semicolon, as in

```
func();  /* a function call */
a = b+c; /* an assignment statement */
b+f();   /* a valid, but strange statement */
;        /* an empty statement */
```

The first expression statement executes a function call. The second is an assignment. The third expression, though strange, is still evaluated by the C++ compiler and the function **f( )** is called. The final example shows that a statement can be empty (sometimes called a *null statement*).

Block Statements

Block statements are simply groups of related statements that are treated as a unit. The statements that make up a block are logically bound together. Block statements are also called *compound statements*. A block is begun with a { and terminated by its matching }. Programmers use block statements most commonly to create a multistatement target for some other statement, such as **if**. However, you may place a block statement anywhere you would put any other statement. For example, this is perfectly valid (although unusual) C/C++ code:

```
#include <stdio.h>

int main(void)
{
  int i;

  {  /* a block statement */
     i = 120;
     printf("%d", i);
  }

  return 0;
}
```