6

C Control Statements: Looping

You will learn about the following in this chapter:

- Keywords:

  `for`

  `while`

  `do while`

- Operators:

  `< > >=`

  `<= != == +=`

  `*= -= /= %=`

- Functions:

  `fabs()`

- C's three loop structures—`while`, `for`, and `do while`
- Using relational operators to construct expressions to control these loops
- Several other operators
- Arrays, which are often used with loops
- Writing functions that have return values

Powerful, intelligent, versatile, and useful! Most of us wouldn't mind being described that way. With C, there's at least the chance of having our programs described that way. The trick is controlling the flow of a program. According to computer science (which is the science of computers and not science by computers...yet), a good language should provide these three forms of program flow:

- Executing a sequence of statements
- Repeating a sequence of statements until some condition is met (looping)
- Using a test to decide between alternative sequences (branching)

The first form you know well; all the previous programs have consisted of a sequence of statements. The `while` loop is one example of the second form. This chapter takes a closer look at the `while` loop along with two other loop structures—`for` and `do while`. The final form, choosing between different possible courses of action, makes a program much more "intelligent" and increases the usefulness of a computer enormously. Sadly, you'll have to wait a chapter before being entrusted with such power. This chapter also introduces arrays because they give you something to do with your new knowledge of loops. In addition, this chapter continues your education about functions. Let's begin by reviewing the `while` loop.

Revisiting the `while` Loop

You are already somewhat familiar with the `while` loop, but let's review it with a program that sums integers entered from the keyboard (see Listing 6.1). This example makes use of the return value of `scanf()` to terminate input.

Listing 6.1 **The `summing.c` Program**

```
/* summing.c -- sums integers entered interactively */
#include <stdio.h>
int main(void)
{
    long num;
    long sum = 0L;      /* initialize sum to zero   */
    int status;

    printf("Please enter an integer to be summed ");
    printf("(q to quit): ");
    status = scanf("%ld", &num);
    while (status == 1) /* == means "is equal to"   */
    {
        sum = sum + num;
        printf("Please enter next integer (q to quit): ");
        status = scanf("%ld", &num);
    }
    printf("Those integers sum to %ld.\n", sum);

    return 0;
}
```

Listing 6.1 uses type long to allow for larger numbers. For consistency, the program initializes sum to 0L (type long zero) rather than to 0 (type int zero), even though C's automatic conversions enable you to use a plain 0.

Here is a sample run:

```
Please enter an integer to be summed (q to quit): 44
Please enter next integer (q to quit): 33
Please enter next integer (q to quit): 88
Please enter next integer (q to quit): 121
Please enter next integer (q to quit): q
Those integers sum to 286.
```

Program Comments

Let's look at the while loop first. The test condition for this loop is the following expression:

```
status == 1
```

The == operator is C's *equality operator*; that is, this expression tests whether status is equal to 1. Don't confuse it with status = 1, which assigns 1 to status. With the status == 1 test condition, the loop repeats as long as status is 1. For each cycle, the loop adds the current value of num to sum, so that sum maintains a running total. When status gets a value other than 1, the loop terminates, and the program reports the final value of sum.

For the program to work properly, it should get a new value for num on each loop cycle, and it should reset status on each cycle. The program accomplishes this by using two distinct features of scanf(). First, it uses scanf() to attempt to read a new value for num. Second, it uses the scanf() return value to report on the success of that attempt. Recall from Chapter 4, "Character Strings and Formatted Input/Output," that scanf() returns the number of items successfully read. If scanf() succeeds in reading an integer, it places the integer into num and returns the value 1, which is assigned to status. (Note that the input value goes to num, not to status.) This updates both num and the value of status, and the while loop goes through another cycle. If you respond with nonnumeric input, such as q, scanf() fails to find an integer to read, so its return value and status will be 0. That terminates the loop. The input character q, because it isn't a number, is placed back into the input queue; it does not get read. (Actually, any nonnumeric input, not just q, terminates the loop, but asking the user to enter q is a simpler instruction than asking the user to enter nonnumeric input.)

If scanf() runs into a problem before attempting to convert the value (for example, by detecting the end of the file or by encountering a hardware problem), it returns the special value EOF, which typically is defined as -1. This value, too, will cause the loop to terminate.

This dual use of scanf() gets around a troublesome aspect of interactive input to a loop: How do you tell the loop when to stop? Suppose, for instance, that scanf() did not have a return value. Then, the only thing that would change on each loop is the value of num. You could use the value of num to terminate the loop, using, say, num > 0 (num greater than 0) or num != 0 (num not equal to 0) as a test condition, but this prevents you from entering certain values,

such as –3 or 0, as input. Instead, you could add new code to the loop, such as asking "Do you wish to continue? <y/n>" at each cycle, and then test to see whether the user entered y. This is a bit clunky and slows down input. Using the return value of `scanf()` avoids these problems.

Now let's take a closer look at the program structure. We can summarize it as follows:

```
initialize sum to 0
prompt user
read input
while the input is an integer,
     add the input to sum,
     prompt user,
     then read next input
after input completes, print sum
```

This, incidentally, is an example of *pseudocode*, which is the art of expressing a program in simple English that parallels the forms of a computer language. Pseudocode is useful for working out the logic of a program. After the logic seems right, you can translate the pseudocode to the actual programming code. One advantage of pseudocode is that it enables you to concentrate on the logic and organization of a program and spares you from simultaneously worrying about how to express the ideas in a computer language. Here, for example, you can use indentation to indicate a block of code and not worry about C syntax requiring braces. Another advantage is that pseudocode is not tied to a particular language, so the same pseudocode can be translated into different computer languages.

Anyway, because the `while` loop is an entry-condition loop, the program must get the input and check the value of `status` *before* it goes to the body of the loop. That is why the program has a `scanf()` before the `while`. For the loop to continue, you need a read statement inside the loop so that it can find out the status of the next input. That is why the program also has a `scanf()` statement at the end of the `while` loop; it readies the loop for its next iteration. You can think of the following as a standard format for a loop:

```
get first value to be tested
while the test is successful
     process value
     get next value
```

C-Style Reading Loop

Listing 6.1 could be written in Pascal, BASIC, or FORTRAN along the same design displayed in the pseudocode. C, however, offers a shortcut. The construction

```
status = scanf("%ld", &num);
while (status == 1)
{
        /* loop actions */
        status = scanf("%ld", &num);
}
```

can be replaced by the following:

```
while (scanf("%ld", &num) == 1)
{
        /* loop actions */
}
```

The second form uses `scanf()` in two different ways simultaneously. First, the function call, if successful, places a value in `num`. Second, the function's return value (which is `1` or `0` and not the value of `num`) controls the loop. Because the loop condition is tested at each iteration, `scanf()` is called at each iteration, providing a new `num` and a new test. In other words, C's syntax features let you replace the standard loop format with the following condensed version:

```
while getting and testing the value succeeds
    process the value
```

Now let's take a more formal look at the `while` statement.

The `while` Statement

This is the general form of the `while` loop:

```
while (expression)
      statement
```

The *`statement`* part can be a simple statement with a terminating semicolon, or it can be a compound statement enclosed in braces.

So far, the examples have used relational expressions for the expression part; that is, *`expression`* has been a comparison of values. More generally, you can use any expression. If *`expression`* is true (or, more generally, nonzero), the statement is executed once and then the expression is tested again. This cycle of test and execution is repeated until *`expression`* becomes false (zero). Each cycle is called an *iteration* (see Figure 6.1).

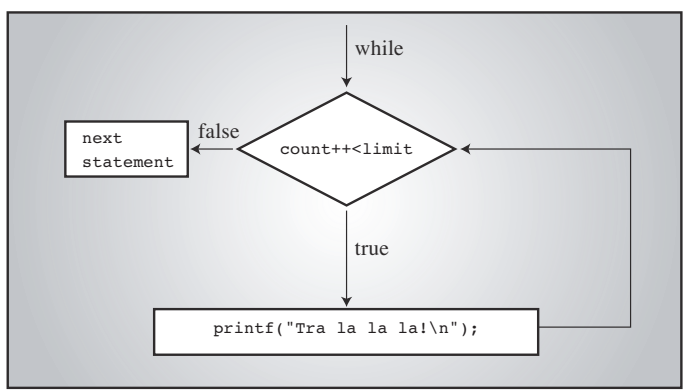

Figure 6.1 Structure of the `while` loop.

Terminating a `while` Loop

Here is a *crucial* point about `while` loops: When you construct a `while` loop, it must include something that changes the value of the test expression so that the expression eventually becomes false. Otherwise, the loop never terminates. (Actually, you can use `break` and an `if` statement to terminate a loop, but you haven't learned about them yet.) Consider this example:

```
index = 1;
while (index < 5)
   printf("Good morning!\n");
```

The preceding fragment prints its cheerful message indefinitely. Why? Because nothing within the loop changes the value of `index` from its initial value of `1`. Now consider this:

```
index = 1;
while (--index < 5)
   printf("Good morning!\n");
```

This last fragment isn't much better. It changes the value of `index`, but in the wrong direction! At least this version will terminate eventually when `index` drops below the most negative number that the system can handle and becomes the largest possible positive value. (The `toobig.c` program in Chapter 3, "Data and C," illustrates how adding 1 to the largest positive number typically produces a negative number; similarly, subtracting 1 from the most negative number typically yields a positive value.)

When a Loop Terminates

It is important to realize that the decision to terminate the loop or to continue takes place only when the test condition is evaluated. For example, consider the program shown in Listing 6.2.

Listing 6.2 **The `when.c` Program**

```
// when.c -- when a loop quits
#include <stdio.h>
int main(void)
{
    int n = 5;

    while (n < 7)                    // line 7
    {
        printf("n = %d\n", n);
        n++;                         // line 10
        printf("Now n = %d\n", n);   // line 11
    }
    printf("The loop has finished.\n");

    return 0;
}
```

Running Listing 6.2 produces the following output:

```
n = 5
Now n = 6
n = 6
Now n = 7
The loop has finished.
```

The variable `n` first acquires the value `7` on line 10 during the second cycle of the loop. However, the program doesn't quit then. Instead, it completes the loop (line 11) and quits the loop only when the test condition on line 7 is evaluated for the third time. (The variable `n` was `5` for the first test and `6` for the second test.)

`while`: An Entry-Condition Loop

The `while` loop is a *conditional* loop using an entry condition. It is called "conditional" because the execution of the statement portion depends on the condition described by the test expression, such as `(index < 5)`. The expression is an *entry condition* because the condition must be met before the body of the loop is entered. In a situation such as the following, the body of the loop is never entered because the condition is false to begin with:

```
index = 10;
while (index++ < 5)
    printf("Have a fair day or better.\n");
```

Change the first line to

```
index = 3;
```

and the loop will execute.

Syntax Points

When using `while`, keep in mind that only the single statement, simple or compound, following the test condition is part of the loop. Indentation is an aid to the reader, not the computer. Listing 6.3 shows what can happen if you forget this.

Listing 6.3 **The `while1.c` Program**

```
/* while1.c -- watch your braces       */
/* bad coding creates an infinite loop */
#include <stdio.h>
int main(void)
{
    int n = 0;

    while (n < 3)
```

```
        printf("n is %d\n", n);
        n++;
    printf("That's all this program does\n");

    return 0;
}
```

Listing 6.3 produces the following output:

```
n is 0
n is 0
n is 0
n is 0
n is 0
```

...and so on, until you kill the program.

Although this example indents the `n++;` statement, it doesn't enclose it and the preceding statement within braces. Therefore, only the single print statement immediately following the test condition is part of the loop. The variable `n` is never updated, the condition `n < 3` remains eternally true, and you get a loop that goes on printing `n is 0` until you kill the program. This is an example of an *infinite loop*, one that does not quit without outside intervention.

Always remember that the `while` statement itself, even if it uses compound statements, counts syntactically as a single statement. The statement runs from the `while` to the first semicolon or, in the case of using a compound statement, to the terminating brace.

Be careful where you place your semicolons. For instance, consider the program in Listing 6.4.

Listing 6.4 **The `while2.c` Program**

```
/* while2.c -- watch your semicolons */
#include <stdio.h>
int main(void)
{
    int n = 0;

    while (n++ < 3);             /* line 7 */
        printf("n is %d\n", n);  /* line 8 */
    printf("That's all this program does.\n");

    return 0;
}
```

Listing 6.4 produces the following output:

```
n is 4
That's all this program does.
```

As we said earlier, the loop ends with the first statement, simple or compound, following the test condition. Because there is a semicolon immediately after the test condition on line 7, the loop ends there, because a lone semicolon counts as a statement. The print statement on line 8 is not part of the loop, so `n` is incremented on each loop, but it is printed only after the loop is exited.

In this example, the test condition is followed with the *null statement*, one that does nothing. In C, the lone semicolon represents the null statement. Occasionally, programmers intentionally use the `while` statement with a null statement because all the work gets done in the test. For example, suppose you want to skip over input to the first character that isn't whitespace or a digit. You can use a loop like this:

```
while (scanf("%d", &num) == 1)
  ;    /* skip integer input */
```

As long as `scanf()` reads an integer, it returns `1`, and the loop continues. Note that, for clarity, you should put the semicolon (the null statement) on the line below instead of on the same line. This makes it easier to see the null statement when you read a program and also reminds you that the null statement is there deliberately. Even better, use the `continue` statement discussed in the next chapter.

Which Is Bigger: Using Relational Operators and Expressions

`While` loops often rely on test expressions that make comparisons, comparison expressions merit a closer look. Such expressions are termed *relational expressions*, and the operators that appear in them are called *relational operators*. You have used several already, and Table 6.1 gives a complete list of C relational operators. This table pretty much covers all the possibilities for numerical relationships. (Numbers, even complex ones, are less complex than humans.)

Table 6.1 **Relational Operators**

Operator	Meaning
<	Is less than
<=	Is less than or equal to
==	Is equal to
>=	Is greater than or equal to
>	Is greater than
!=	Is not equal to

The relational operators are used to form the relational expressions used in `while` statements and in other C statements that we'll discuss later. These statements check to see whether the expression is true or false. Here are three unrelated statements containing examples of relational expressions. The meaning, we hope, is clear.

```
while (number < 6)
{
    printf("Your number is too small.\n");
    scanf("%d", &number);
}

while (ch != '$')
{
     count++;
     scanf("%c", &ch);
}

while (scanf("%f", &num) == 1)
    sum = sum + num;
```

Note in the second example that the relational expressions can be used with characters, too. The machine character code (which we have been assuming is ASCII) is used for the comparison. However, you can't use the relational operators to compare strings. Chapter 11, "Character Strings and String Functions," will show you what to use for strings.

The relational operators can be used with floating-point numbers, too. Beware, though: You should limit yourself to using only < and > in floating-point comparisons. The reason is that round-off errors can prevent two numbers from being equal, even though logically they should be. For example, certainly the product of 3 and 1/3 is 1.0. If you express 1/3 as a six-place decimal fraction, however, the product is .999999, which is not quite equal to 1. The `fabs()` function, declared in the `math.h` header file, can be handy for floating-point tests. This function returns the absolute value of a floating-point value—that is, the value without the algebraic sign. For example, you could test whether a number is close to a desired result with something like Listing 6.5.

Listing 6.5 **The `cmpflt.c` Program**

```
// cmpflt.c -- floating-point comparisons
#include <math.h>
#include <stdio.h>
int main(void)
{
    const double ANSWER = 3.14159;
    double response;

    printf("What is the value of pi?\n");
    scanf("%lf", &response);
```

```
    while (fabs(response - ANSWER) > 0.0001)
    {
        printf("Try again!\n");
        scanf("%lf", &response);
    }
    printf("Close enough!\n");

    return 0;
}
```

This loop continues to elicit a response until the user gets within 0.0001 of the correct value:

```
What is the value of pi?
3.14
Try again!
3.1416
Close enough!
```

Each relational expression is judged to be true or false (but never maybe). This raises an interesting question.

What Is Truth?

You can answer this age-old question, at least as far as C is concerned. Recall that an expression in C always has a value. This is true even for relational expressions, as the example in Listing 6.6 shows. That example prints the values of two relational expressions—one true and one false.

Listing 6.6 **The `t_and_f.c` Program**

```
/* t_and_f.c -- true and false values in C */
#include <stdio.h>
int main(void)
{
    int true_val, false_val;

    true_val = (10 > 2);    // value of a true relationship
    false_val = (10 == 2);  // value of a false relationship
    printf("true = %d; false = %d \n", true_val, false_val);

    return 0;
}
```

Listing 6.6 assigns the values of two relational expressions to two variables. Being straightforward, it assigns `true_val` the value of a true expression, and `false_val` the value of a false expression. Running the program produces the following simple output:

```
true = 1; false = 0
```

Aha! For C, a true expression has the value `1`, and a false expression has the value `0`. Indeed, some C programs use the following construction for loops that are meant to run forever because `1` always is true:

```
while (1)
{
  ...
}
```

What Else Is True?

If you can use a `1` or a `0` as a `while` statement test expression, can you use other numbers? If so, what happens? Let's experiment by trying the program in Listing 6.7.

Listing 6.7 **The `truth.c` Program**

```
// truth.c -- what values are true?
#include <stdio.h>
int main(void)
{
    int n = 3;

    while (n)
        printf("%2d is true\n", n--);
    printf("%2d is false\n", n);

    n = -3;
    while (n)
        printf("%2d is true\n", n++);
    printf("%2d is false\n", n);

    return 0;
}
```

Here are the results:

```
 3 is true
 2 is true
 1 is true
 0 is false
-3 is true
```

```
-2 is true
-1 is true
 0 is false
```

The first loop executes when `n` is `3`, `2`, and `1`, but terminates when `n` is `0`. Similarly, the second loop executes when `n` is `-3`, `-2`, and `-1`, but terminates when `n` is `0`. More generally, *all* nonzero values are regarded as true, and only `0` is recognized as false. C has a very tolerant notion of truth!

Alternatively, you can say that a `while` loop executes as long as its test condition evaluates to nonzero. This puts test conditions on a numeric basis instead of a true/false basis. Keep in mind that relational expressions evaluate to `1` if true and to `0` if false, so such expressions really are numeric.

Many C programmers make use of this property of test conditions. For example, the phrase `while (goats != 0)` can be replaced by `while (goats)` because the expression `(goats != 0)` and the expression `(goats)` both become `0`, or false, only when `goats` has the value `0`. The first form probably is clearer to those just learning the language, but the second form is the idiom most often used by C programmers. You should try to become sufficiently familiar with the `while (goats)` form so that it seems natural to you.

Troubles with Truth

C's tolerant notion of truth can lead to trouble. For example, let's make one subtle change to the program from Listing 6.1, producing the program shown in Listing 6.8.

Listing 6.8 **The `trouble.c` Program**

```
// trouble.c -- misuse of =
// will cause infinite loop
#include <stdio.h>
int main(void)
{
    long num;
    long sum = 0L;
    int status;

    printf("Please enter an integer to be summed ");
    printf("(q to quit): ");
    status = scanf("%ld", &num);
    while (status = 1)
    {
        sum = sum + num;
        printf("Please enter next integer (q to quit): ");
        status = scanf("%ld", &num);
    }
```

```
    printf("Those integers sum to %ld.\n", sum);

    return 0;
}
```

Listing 6.8 produces output like the following:

```
Please enter an integer to be summed (q to quit): 20
Please enter next integer (q to quit): 5
Please enter next integer (q to quit): 30
Please enter next integer (q to quit): q
Please enter next integer (q to quit):
Please enter next integer (q to quit):
Please enter next integer (q to quit):
Please enter next integer (q to quit):
```

...and so on until you kill the program—so perhaps you shouldn't actually try running this example.

This troublesome example made a change in the `while` test condition, replacing `status == 1` with `status = 1`. The second statement is an assignment statement, so it gives `status` the value `1`. Furthermore, the value of an assignment statement is the value of the left side, so `status = 1` has the same numerical value of `1`. So for all practical purposes, the `while` loop is the same as using `while (1)`; that is, it is a loop that never quits. You enter `q`, and `status` is set to `0`, but the loop test resets `status` to `1` and starts another cycle.

You might wonder why, because the program keeps looping, the user doesn't get a chance to type in any more input after entering `q`. When `scanf()` fails to read the specified form of input, it leaves the nonconforming input in place to be read the next time. When `scanf()` tries to read the `q` as an integer and fails, it leaves the `q` there. During the next loop cycle, `scanf()` attempts to read where it left off the last time—at the `q`. Once again, `scanf()` fails to read the `q` as an integer, so not only does this example set up an infinite loop, it also creates a loop of infinite failure, a daunting concept. It is fortunate that computers, as yet, lack feelings. Following stupid instructions eternally is no better or worse to a computer than successfully predicting the stock market for the next 10 years.

Don't use = for ==. Some computer languages (BASIC, for example) do use the same symbol for both the assignment operator and the relational equality operator, but the two operations are quite different (see Figure 6.2). The assignment operator assigns a value to the left variable. The relational equality operator, however, checks to see whether the left and right sides are already equal. It doesn't change the value of the left-hand variable, if one is present. Here's an example:

```
canoes = 5        ←Assigns the value 5 to canoes
canoes == 5       ←Checks to see whether canoes has the value 5
```

Be careful about using the correct operator. A compiler will let you use the wrong form, yielding results other than what you expect. (However, so many people have misused = so often that most compilers today will issue a warning to the effect that perhaps you didn't mean to use this.) If one of the values being compared is a constant, you can put it on the left side of the comparison to help catch errors:

```
5 = canoes          ←syntax error
5 == canoes         ←Checks to see whether canoes has the value 5
```

The point is that it is illegal to assign to a constant, so the compiler will tag the use of the assignment operator as a syntax error. Many practitioners put the constant first when constructing expressions that test for equality.

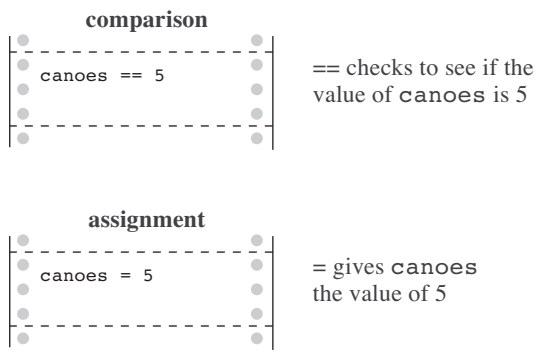

Figure 6.2 The relational operator == and the assignment operator =.

To sum up, the relational operators are used to form relational expressions. Relational expressions have the value 1 if true and 0 if false. Statements (such as `while` and `if`) that normally use relational expressions as tests can use any expression as a test, with nonzero values recognized as "true" and zero values as "false."

The New `_Bool` Type

Variables intended to represent true/false values traditionally have been represented by type `int` in C. C99 adds the `_Bool` type specifically for variables of this sort. The type is named after George Boole, the English mathematician who developed a system of algebra to represent and solve problems in logic. In programming, variables representing true or false have come to be known as *Boolean variables*, so `_Bool` is the C type name for a Boolean variable. A `_Bool` variable can only have a value of 1 (true) or 0 (false). If you try to assign a nonzero numeric value to a `_Bool` variable, the variable is set to 1, reflecting that C considers any nonzero value to be true.

Listing 6.9 fixes the test condition in Listing 6.8 and replaces the `int` variable `status` with the `_Bool` variable `input_is_good`. It's a common practice to give Boolean variables names that suggest true or false values.

Listing 6.9 **The `boolean.c` Program**

```
// boolean.c -- using a _Bool variable
#include <stdio.h>
int main(void)
{
    long num;
    long sum = 0L;
    _Bool input_is_good;

    printf("Please enter an integer to be summed ");
    printf("(q to quit): ");
    input_is_good = (scanf("%ld", &num) == 1);
    while (input_is_good)
    {
        sum = sum + num;
        printf("Please enter next integer (q to quit): ");
        input_is_good = (scanf("%ld", &num) == 1);
    }
    printf("Those integers sum to %ld.\n", sum);

    return 0;
}
```

Note how the code assigns the result of a comparison to the variable:

```
input_is_good = (scanf("%ld", &num) == 1);
```

This makes sense, because the == operator returns either a value of 1 or 0. Incidentally, the parentheses enclosing the == expression are not needed because the == operator has higher precedence than =; however, they may make the code easier to read. Also note how the choice of name for the variable makes the `while` loop test easy to understand:

```
while (input_is_good)
```

C99 also provides for a `stdbool.h` header file. This header file makes `bool` an alias for `_Bool` and defines `true` and `false` as symbolic constants for the values 1 and 0. Including this header file allows you to write code that is compatible with C++, which defines `bool`, `true`, and `false` as keywords.

If your system does not yet support the `_Bool` type, you can replace `_Bool` with `int`, and the example will work the same.

Precedence of Relational Operators

The precedence of the relational operators is less than that of the arithmetic operators, including + and –, and greater than that of assignment operators. This means, for example, that

```
x > y + 2
```

means the same as

```
x > (y + 2)
```

It also means that

```
x = y > 2
```

means

```
x = (y > 2)
```

In other words, `x` is assigned `1` if `y` is greater than `2` and is `0` otherwise; `x` is not assigned the value of `y`.

The relational operators have a greater precedence than the assignment operator. Therefore,

```
x_bigger = x > y;
```

means

```
x_bigger = (x > y);
```

The relational operators are themselves organized into two different precedences.

Higher precedence group:	`< <= > >=`
Lower precedence group:	`== !=`

Like most other operators, the relational operators associate from left to right. Therefore,

```
ex != wye == zee
```

is the same as

```
(ex != wye) == zee
```

First, C checks to see whether `ex` and `wye` are unequal. Then, the resulting value of `1` or `0` (true or false) is compared to the value of `zee`. We don't anticipate using this sort of construction, but we feel it is our duty to point out such sidelights.

Table 6.2 shows the priorities of the operators introduced so far, and Reference Section II, "C Operators," in Appendix B has a complete precedence ranking of all operators.

Table 6.2 **Operator Precedence**

Operators (From High to Low Precedence)	Associativity
( )	L–R
– + ++ —— sizeof	R–L (*type*) (all unary)
* / %	L–R
+ –	L–R
< > <= >=	L–R
== !=	L–R
=	R–L

Summary: The *while* Statement

Keyword:

`while`

General Comments:

The `while` statement creates a loop that repeats until the test expression becomes false, or zero. The `while` statement is an entry-condition loop—that is, the decision to go through one more pass of the loop is made before the loop is traversed. Therefore, it is possible that the loop is never traversed. The statement part of the form can be a simple statement or a compound statement.

Form:

```
while (expression)
        statement
```

The *statement* portion is repeated until the *expression* becomes false or 0.

Examples:

```
while (n++ < 100)
   printf(" %d %d\n",n, 2 * n + 1); // single statement

while (fargo < 1000)
{                                   // compound statement
   fargo = fargo + step;
   step = 2 * step;
}
```

Summary: Relational Operators and Expressions

Relational Operators:

Each relational operator compares the value at its left to the value at its right.

`<`	Is less than
`<=`	Is less than or equal to
`==`	Is equal to
`>=`	Is greater than or equal to
`>`	Is greater than
`!=`	Is unequal to

Relational Expressions:

A simple relational expression consists of a relational operator with an operand on each side. If the relation is true, the relational expression has the value `1`. If the relation is false, the relational expression has the value `0`.

Examples:

`5 > 2` is true and has the value `1`.

`(2 + a) == a` is false and has the value `0`.

Indefinite Loops and Counting Loops

Some of the `while` loop examples have been *indefinite* loops. That means we don't know in advance how many times the loop will be executed before the expression becomes false. For example, when Listing 6.1 used an interactive loop to sum integers, we didn't know beforehand how many integers would be entered. Other examples, however, have been *counting* loops. They execute a predetermined number of repetitions. Listing 6.10 is a short example of a `while` counting loop.

Listing 6.10 **The `sweetie1.c` Program**

```
// sweetie1.c -- a counting loop
#include <stdio.h>
int main(void)
{
    const int NUMBER = 22;
    int count = 1;                      // initialization

    while (count <= NUMBER)             // test
    {
        printf("Be my Valentine!\n");   // action
        count++;                        // update count
    }
```

```
    return 0;
}
```

Although the form used in Listing 6.10 works fine, it is not the best choice for this situation because the actions defining the loop are not all gathered together. Let's elaborate on that point.

Three actions are involved in setting up a loop that is to be repeated a fixed number of times:

1. A counter must be initialized.
2. The counter is compared with some limiting value.
3. The counter is incremented each time the loop is traversed.

The `while` loop condition takes care of the comparison. The increment operator takes care of the incrementing. In Listing 6.10, the incrementing is done at the end of the loop. This choice makes it possible to omit the incrementing accidentally. So it would be better to combine the test and update actions into one expression by using `count++ <= NUMBER`, but the initialization of the counter is still done outside the loop, making it possible to forget to initialize a counter. Experience teaches us that what might happen *will* happen eventually, so let's look at a control statement that avoids these problems.

The `for` Loop

The `for` loop gathers all three actions (initializing, testing, and updating) into one place. By using a `for` loop, you can replace the preceding program with the one shown in Listing 6.11.

Listing 6.11 **The `sweetie2.c` Program**

```
// sweetie2.c -- a counting loop using for
#include <stdio.h>
int main(void)
{
    const int NUMBER = 22;
    int count;

    for (count = 1; count <= NUMBER; count++)
        printf("Be my Valentine!\n");

    return 0;
}
```

The parentheses following the keyword `for` contain three expressions separated by two semicolons. The first expression is the initialization. It is done just once, when the `for` loop first

starts. The second expression is the test condition; it is evaluated before each potential execution of a loop. When the expression is false (when `count` is greater than `NUMBER`), the loop is terminated. The third expression, the change or update, is evaluated at the end of each loop. Listing 6.10 uses it to increment the value of `count`, but it needn't be restricted to that use. The `for` statement is completed by following it with a single simple or compound statement. Each of the three control expressions is a full expression, so any side effects in a control expression, such as incrementing a variable, take place before the program evaluates another expression. Figure 6.3 summarizes the structure of a `for` loop.

To show another example, Listing 6.12 uses the `for` loop in a program that prints a table of cubes.

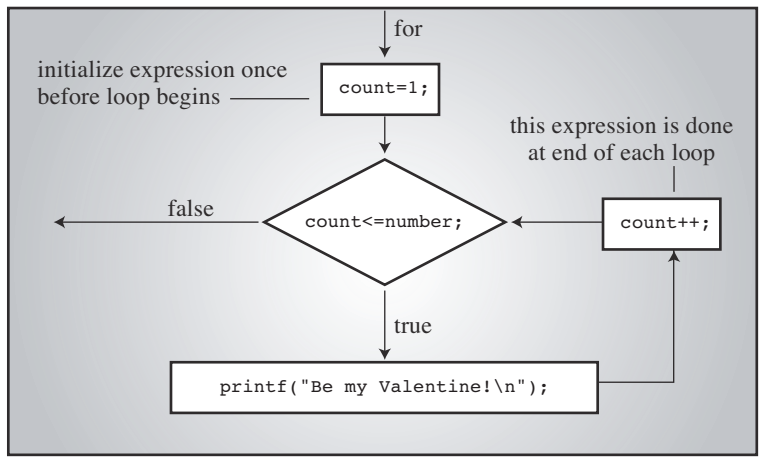

Figure 6.3 Structure of a `for` loop.

Listing 6.12 **The `for_cube.c` Program**

```
/* for_cube.c -- using a for loop to make a table of cubes */
#include <stdio.h>
int main(void)
{
    int num;

    printf("    n   n cubed\n");
    for (num = 1; num <= 6; num++)
        printf("%5d %5d\n", num, num*num*num);

    return 0;
}
```

Listing 6.12 prints the integers 1 through 6 and their cubes.

```
n   n cubed
1     1
2     8
3    27
4    64
5   125
6   216
```

The first line of the `for` loop tells us immediately all the information about the loop parameters: the starting value of `num`, the final value of `num`, and the amount that `num` increases on each looping.

Using `for` for Flexibility

Although the `for` loop looks similar to the FORTRAN `DO` loop, the Pascal `FOR` loop, and the BASIC `FOR...NEXT` loop, it is much more flexible than any of them. This flexibility stems from how the three expressions in a `for` specification can be used. The examples so far have used the first expression to initialize a counter, the second expression to express the limit for the counter, and the third expression to increase the value of the counter by 1. When used this way, the C `for` statement is very much like the others we have mentioned. However, there are many more possibilities; here are nine variations:

- You can use the decrement operator to count down instead of up:

```
/* for_down.c */
#include <stdio.h>
int main(void)
{
    int secs;

    for (secs = 5; secs > 0; secs--)
        printf("%d seconds!\n", secs);
    printf("We have ignition!\n");
    return 0;
}
```

Here is the output:

```
5 seconds!
4 seconds!
3 seconds!
2 seconds!
1 seconds!
We have ignition!
```

- You can count by twos, tens, and so on, if you want:

```
/* for_13s.c */
#include <stdio.h>
int main(void)
{
    int n;        // count by 13s from 2

    for (n = 2;  n < 60; n = n + 13)
        printf("%d \n", n);
    return 0;
}
```

  This would increase `n` by 13 during each cycle, printing the following:

```
 2
15
28
41
54
```

- You can count by characters instead of by numbers:

```
/* for_char.c */
#include <stdio.h>
int main(void)
{
    char ch;

    for (ch = 'a'; ch <= 'z'; ch++)
        printf("The ASCII value for %c is %d.\n", ch, ch);
    return 0;
}
```

  The program assumes the system uses ASCII code for characters. Here's the abridged output:

```
The ASCII value for a is 97.
The ASCII value for b is 98.
...
The ASCII value for x is 120.
The ASCII value for y is 121.
The ASCII value for z is 122.
```

  The program works because characters are stored as integers, so this loop really counts by integers anyway.

- You can test some condition other than the number of iterations. In the `for_cube` program, you can replace

```
for (num = 1; num <= 6; num++)
```

with

```
for (num = 1; num*num*num <= 216; num++)
```

You would use this test condition if you were more concerned with limiting the size of the cube than with limiting the number of iterations.

- You can let a quantity increase geometrically instead of arithmetically; that is, instead of adding a fixed amount each time, you can multiply by a fixed amount:

```
/* for_geo.c */
#include <stdio.h>
int main(void)
{
    double debt;

    for (debt = 100.0; debt < 150.0; debt = debt * 1.1)
        printf("Your debt is now $%.2f.\n", debt);
    return 0;
}
```

This program fragment multiplies `debt` by 1.1 for each cycle, increasing it by 10% each time. The output looks like this:

```
Your debt is now $100.00.
Your debt is now $110.00.
Your debt is now $121.00.
Your debt is now $133.10.
Your debt is now $146.41.
```

- You can use any legal expression you want for the third expression. Whatever you put in will be updated for each iteration.

```
/* for_wild.c */
#include <stdio.h>
int main(void)
{
    int x;
    int y = 55;

    for (x = 1; y <= 75; y = (++x * 5) + 50)
        printf("%10d %10d\n", x, y);
    return 0;
}
```

This loop prints the values of `x` and of the algebraic expression `++x * 5 + 50`. The output looks like this:

```
1         55
2         60
3         65
```

```
4          70
5          75
```

Notice that the test involves `y`, not `x`. Each of the three expressions in the `for` loop control can use different variables. (Note that although this example is valid, it does not show good style. The program would have been clearer if we hadn't mixed the updating process with an algebraic calculation.)

- You can even leave one or more expressions blank (but don't omit the semicolons). Just be sure to include within the loop itself some statement that eventually causes the loop to terminate.

```
/* for_none.c */
#include <stdio.h>
int main(void)
{
    int ans, n;

    ans = 2;
    for (n = 3; ans <= 25; )
        ans = ans * n;
    printf("n = %d; ans = %d.\n", n, ans);
    return 0;
}
```

Here is the output:

```
n = 3; ans = 54.
```

The loop keeps the value of `n` at 3. The variable `ans` starts with the value 2, and then increases to 6 and 18 and obtains a final value of 54. (The value 18 is less than 25, so the `for` loop goes through one more iteration, multiplying 18 by 3 to get 54.) Incidentally, an empty middle control expression is considered to be true, so the following loop goes on forever:

```
for (; ; )
     printf("I want some action\n");
```

- The first expression need not initialize a variable. It could, instead, be a `printf()` statement of some sort. Just remember that the first expression is evaluated or executed only once, before any other parts of the loop are executed.

```
/* for_show.c */
#include <stdio.h>
int main(void)
{
    int num = 0;

    for (printf("Keep entering numbers!\n"); num != 6;  )
        scanf("%d", &num);
```

```
    printf("That's the one I want!\n");
    return 0;
}
```

This fragment prints the first message once and then keeps accepting numbers until you enter 6:

```
Keep entering numbers!
3
5
8
6
That's the one I want!
```

- The parameters of the loop expressions can be altered by actions within the loop. For example, suppose you have the loop set up like this:

  ```
  for (n = 1; n < 10000; n = n + delta)
  ```

  If after a few iterations your program decides that `delta` is too small or too large, an `if` statement (see Chapter 7, "C Control Statements: Branching and Jumps") inside the loop can change the size of `delta`. In an interactive program, `delta` can be changed by the user as the loop runs. This sort of adjustment is a bit on the dangerous side; for example, setting `delta` to `0` gets you (and the loop) nowhere.

In short, the freedom you have in selecting the expressions that control a `for` loop makes this loop able to do much more than just perform a fixed number of iterations. The usefulness of the `for` loop is enhanced further by the operators we will discuss shortly.

Summary: The *for* Statement

Keyword: `for`

General Comments:

The `for` statement uses three control expressions, separated by semicolons, to control a looping process. The `initialize` expression is executed once, before any of the loop statements are executed. Then the `test` expression is evaluated and, if it is true (or nonzero), the loop is cycled through once. Then the `update` expression is evaluated, and it is time to check the `test` expression again. The `for` statement is an entry-condition loop—the decision to go through one more pass of the loop is made before the loop is traversed. Therefore, it is possible that the loop is never traversed. The `statement` part of the form can be a simple statement or a compound statement.

Form:

```
for (initialize ; test ; update)
     statement
```

The loop is repeated until `test` becomes false or zero.

Example:

```
for (n = 0;  n < 10 ; n++)
      printf(" %d %d\n", n, 2 * n + 1);
```

More Assignment Operators: +=, -=, *=, /=, %=

C has several assignment operators. The most basic one, of course, is =, which simply assigns the value of the expression at its right to the variable at its left. The other assignment operators update variables. Each is used with a variable name to its left and an expression to its right. The variable is assigned a new value equal to its old value adjusted by the value of the expression at the right. The exact adjustment depends on the operator. For example,

`scores += 20` is the same as `scores = scores + 20`.

`dimes -= 2` is the same as `dimes = dimes - 2`.

`bunnies *= 2` is the same as `bunnies = bunnies * 2`.

`time /= 2.73` is the same as `time = time / 2.73`.

`reduce %= 3` is the same as `reduce = reduce % 3`.

The preceding list uses simple numbers on the right, but these operators also work with more elaborate expressions, such as the following:

`x *= 3 * y + 12` is the same as `x = x * (3 * y + 12)`.

The assignment operators we've just discussed have the same low priority that = does—that is, less than that of + or *. This low priority is reflected in the last example in which `12` is added to `3 * y` before the result is multiplied by `x`.

You are not required to use these forms. They are, however, more compact, and they may produce more efficient machine code than the longer form. The combination assignment operators are particularly useful when you are trying to squeeze something complex into a `for` loop specification.

The Comma Operator

The comma operator extends the flexibility of the `for` loop by enabling you to include more than one initialization or update expression in a single `for` loop specification. For example, Listing 6.13 shows a program that prints first-class postage rates. (At the time of this writing, the rate is 46 cents for the first ounce and 20 cents for each additional ounce. You can check the Internet for the current rates.)

Listing 6.13 **The `postage.c` Program**

```
// postage.c -- first-class postage rates
#include <stdio.h>
int main(void)
{
    const int FIRST_OZ = 46; // 2013 rate
    const int NEXT_OZ = 20;  // 2013 rate
    int ounces, cost;

    printf(" ounces  cost\n");
    for (ounces=1, cost=FIRST_OZ; ounces <= 16; ounces++,
         cost += NEXT_OZ)
        printf("%5d   $%4.2f\n", ounces, cost/100.0);

    return 0;
}
```

The first five lines of the output look like this:

```
ounces  cost
    1   $0.46
    2   $0.66
    3   $0.86
    4   $1.06
```

The program uses the comma operator in the initialize and the update expressions. Its presence in the first expression causes `ounces` and `cost` to be initialized. Its second occurrence causes `ounces` to be increased by 1 and `cost` to be increased by 20 (the value of `NEXT_OZ`) for each iteration. All the calculations are done in the `for` loop specifications (see Figure 6.4).

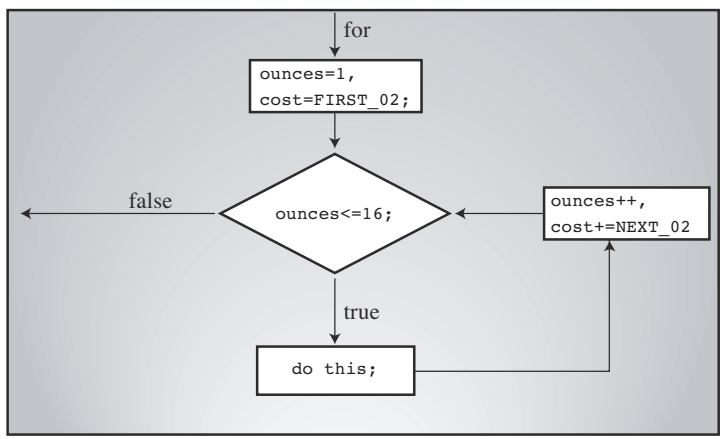

Figure 6.4 The comma operator and the `for` loop.

The comma operator is not restricted to `for` loops, but that's where it is most often used. The operator has two further properties. First, it guarantees that the expressions it separates are evaluated in a left-to-right order. (In other words, the comma is a sequence point, so all side effects to the left of the comma take place before the program moves to the right of the comma.) Therefore, `ounces` is initialized before `cost`. The order is not important for this example, but it would be important if the expression for `cost` contained `ounces`. Suppose, for instance, that you had this expression:

```
ounces++, cost = ounces * FIRST_OZ
```

This would increment `ounces` and then use the new value for `ounces` in the second subexpression. The comma being a sequence point guarantees that the side effects of the left subexpression occur before the right subexpression is evaluated.

Second, the value of the whole comma expression is the value of the right-hand member. The effect of the statement

```
x = (y = 3, (z = ++y + 2) + 5);
```

is to first assign `3` to `y`, increment `y` to `4`, and then add `2` to `4` and assign the resulting value of `6` to `z`, next add `5` to `z`, and finally assign the resulting value of `11` to `x`. Why anyone would do this is beyond the scope of this book. On the other hand, suppose you get careless and use comma notation in writing a number:

```
houseprice = 249,500;
```

This is not a syntax error. Instead, C interprets this as a comma expression, with `houseprice = 249` being the left subexpression and `500` the right subexpression. Therefore, the value of the whole comma expression is the value of the right-hand expression, and the left substatement assigns the value `249` to the `houseprice` variable. Therefore, the effect is the same as the following code:

```
houseprice = 249;
500;
```

Remember that any expression becomes a statement with the addition of a semicolon, so `500;` is a statement that does nothing.

On the other hand, the statement

```
houseprice = (249,500);
```

assigns 500, the value of the right subexpression, to `houseprice`.

The comma also is used as a separator, so the commas in

```
char ch, date;
```

and

```
printf("%d %d\n", chimps, chumps);
```

are separators, not comma operators.

Summary: The New Operators

Assignment Operators:

Each of these operators updates the variable at its left by the value at its right, using the indicated operation:

`+=`	Adds the right-hand quantity to the left-hand variable
`-=`	Subtracts the right-hand quantity from the left-hand variable
`*=`	Multiplies the left-hand variable by the right-hand quantity
`/=`	Divides the left-hand variable by the right-hand quantity
`%=`	Gives the remainder obtained from dividing the left-hand variable by the right-hand quantity

Example:

```
rabbits *= 1.6;
```

is the same as

```
rabbits = rabbits * 1.6;
```

These combination assignment operators have the same low precedence as the regular assignment operator, lower than arithmetic operators. Therefore, a statement such as

```
contents *= old_rate + 1.2;
```

has the same final effect as this:

```
contents = contents * (old_rate + 1.2);
```

The Comma Operator:

The comma operator links two expressions into one and guarantees that the leftmost expression is evaluated first. It is typically used to include more information in a `for` loop control expression. The value of the whole expression is the value of the right-hand expression.

Example:

```
for (step = 2, fargo = 0; fargo < 1000; step *= 2)
     fargo += step;
```

Zeno Meets the `for` Loop

Let's see how the `for` loop and the comma operator can help solve an old paradox. The Greek philosopher Zeno once argued that an arrow will never reach its target. First, he said, the arrow covers half the distance to the target. Then it has to cover half of the remaining distance. Then it still has half of what's left to cover, ad infinitum. Because the journey has an infinite number of parts, Zeno argued, it would take the arrow an infinite amount of time to reach its journey's

end. We doubt, however, that Zeno would have volunteered to be a target on the strength of this argument.

Let's take a quantitative approach and suppose that it takes the arrow 1 second to travel the first half. Then it would take 1/2 second to travel half of what was left, 1/4 second to travel half of what was left next, and so on. You can represent the total time by the following infinite series:

```
1 + 1/2 + 1/4 + 1/8 + 1/16 +....
```

The short program in Listing 6.14 finds the sum of the first few terms. The variable `power_of_two` takes on the values `1.0`, `2.0`, `4.0`, `8.0`, and so on.

Listing 6.14 **The `zeno.c` Program**

```
/* zeno.c -- series sum */
#include <stdio.h>

int main(void)
{
    int t_ct;       // term count
    double time, power_of_2;
    int limit;

    printf("Enter the number of terms you want: ");
    scanf("%d", &limit);
    for (time=0, power_of_2=1, t_ct=1; t_ct <= limit;
                            t_ct++, power_of_2 *= 2.0)
    {
        time += 1.0/power_of_2;
        printf("time = %f when terms = %d.\n", time, t_ct);
    }

    return 0;
}
```

Here is the output for 15 terms:

```
Enter the number of terms you want: 15
time = 1.000000 when terms = 1.
time = 1.500000 when terms = 2.
time = 1.750000 when terms = 3.
time = 1.875000 when terms = 4.
time = 1.937500 when terms = 5.
time = 1.968750 when terms = 6.
time = 1.984375 when terms = 7.
time = 1.992188 when terms = 8.
time = 1.996094 when terms = 9.
```

```
time = 1.998047 when terms = 10.
time = 1.999023 when terms = 11.
time = 1.999512 when terms = 12.
time = 1.999756 when terms = 13.
time = 1.999878 when terms = 14.
time = 1.999939 when terms = 15.
```

You can see that although you keep adding more terms, the total seems to level out. Indeed, mathematicians have proven that the total approaches 2.0 as the number of terms approaches infinity, just as this program suggests. Here's one demonstration. Suppose you let `S` represent the sum:

```
S = 1 + 1/2 + 1/4 + 1/8 + ...
```

Here the ellipsis mean "and so on." Then dividing by 2 gives

```
S/2 = 1/2 + 1/4 + 1/8 + 1/16 + ...
```

Subtracting the second expression from the first gives

```
S - S/2 = 1 +1/2 -1/2 + 1/4 -1/4 +...
```

Except for the initial value of `1`, each other value occurs in pairs, one positive and one negative, so those terms cancel each other, leaving

```
S/2 = 1.
```

Then, multiplying both sides by 2 gives

```
S = 2.
```

One possible moral to draw from this is that before doing an involved calculation, check to see whether mathematicians have an easier way to do it.

What about the program itself? It shows that you can use more than one comma operator in an expression. You initialized `time`, `power_of_2`, and `count`. After you set up the conditions for the loop, the program itself is extremely brief.

An Exit-Condition Loop: `do while`

The `while` loop and the `for` loop are both entry-condition loops. The test condition is checked *before* each iteration of the loop, so it is possible for the statements in the loop to never execute. C also has an *exit-condition* loop, in which the condition is checked after each iteration of the loop, guaranteeing that statements are executed at least once. This variety is called a `do while` loop. Listing 6.15 shows an example.

Listing 6.15 **The `do_while.c` Program**

```
/* do_while.c -- exit condition loop */
#include <stdio.h>
int main(void)
{
    const int secret_code = 13;
    int code_entered;

    do
    {
        printf("To enter the triskaidekaphobia therapy club,\n");
        printf("please enter the secret code number: ");
        scanf("%d", &code_entered);
    } while (code_entered != secret_code);
    printf("Congratulations! You are cured!\n");

    return 0;
}
```

The program in Listing 6.15 reads input values until the user enters `13`. The following is a sample run:

```
To enter the triskaidekaphobia therapy club,
please enter the secret code number: 12
To enter the triskaidekaphobia therapy club,
please enter the secret code number: 14
To enter the triskaidekaphobia therapy club,
please enter the secret code number: 13
Congratulations! You are cured!
```

An equivalent program using a `while` loop would be a little longer, as shown in Listing 6.16.

Listing 6.16 **The `entry.c` Program**

```
/* entry.c -- entry condition loop */
#include <stdio.h>
int main(void)
{
    const int secret_code = 13;
    int code_entered;

    printf("To enter the triskaidekaphobia therapy club,\n");
    printf("please enter the secret code number: ");
    scanf("%d", &code_entered);
    while (code_entered != secret_code)
    {
```

```
        printf("To enter the triskaidekaphobia therapy club,\n");
        printf("please enter the secret code number: ");
        scanf("%d", &code_entered);
    }
    printf("Congratulations! You are cured!\n");

    return 0;
}
```

Here is the general form of the `do while` loop:

```
do
    statement
while ( expression );
```

The statement can be simple or compound. Note that the `do while` loop itself counts as a statement and, therefore, requires a terminating semicolon. Also, see Figure 6.5.

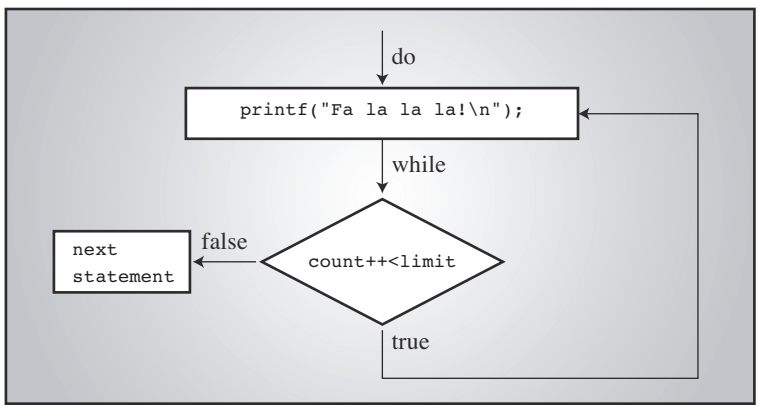

Figure 6.5 Structure of a `do while` loop.

A `do while` loop is always executed at least once because the test is made after the body of the loop has been executed. A `for` loop or a `while` loop, on the other hand, can be executed zero times because the test is made before execution. You should restrict the use of `do while` loops to cases that require at least one iteration. For example, a password program could include a loop along these pseudocode lines:

```
do
{
    prompt for password
    read user input
} while (input not equal to password);
```

Avoid a `do while` structure of the type shown in the following pseudocode:

```
do
{
   ask user if he or she wants to continue
   some clever stuff
} while (answer is yes);
```

Here, after the user answers "no," some clever stuff gets done anyway because the test comes too late.

Summary: The *do while* Statement

Keywords:

`do while`

General Comments:

The `do while` statement creates a loop that repeats until the test *expression* becomes false or zero. The `do while` statement is an exit-condition loop—the decision to go through one more pass of the loop is made after the loop has been traversed. Therefore, the loop must be executed at least once. The *statement* part of the form can be a simple statement or a compound statement.

Form:

```
do
    statement
while (expression);
```

The *statement* portion is repeated until the *expression* becomes false or zero.

Example:

```
do
    scanf("%d", &number);
while (number != 20);
```

Which Loop?

When you decide you need a loop, which one should you use? First, decide whether you need an entry-condition loop or an exit-condition loop. Your answer should usually be an entry-condition loop. There are several reasons computer scientists consider an entry-condition loop to be superior. One is the general principle that it is better to look before you leap (or loop) than after. A second is that a program is easier to read if the loop test is found at the beginning of the loop. Finally, in many uses, it is important that the loop be skipped entirely if the test is not initially met.

Assume that you need an entry-condition loop. Should it be a `for` or a `while`? This is partly a matter of taste, because what you can do with one, you can do with the other. To make a `for` loop like a `while`, you can omit the first and third expressions. For example,

```
for ( ;test; )
```

is the same as

```
while (test)
```

To make a `while` like a `for`, preface it with an initialization and include update statements. For example,

```
initialize;
while (test)
{
  body;
  update;
}
```

is the same as

```
for (initialize; test; update)
   body;
```

In terms of prevailing style, a `for` loop is appropriate when the loop involves initializing and updating a variable, and a `while` loop is better when the conditions are otherwise. A `while` loop is natural for the following condition:

```
while (scanf("%ld", &num) == 1)
```

The `for` loop is a more natural choice for loops involving counting with an index:

```
for (count = 1; count <= 100; count++)
```

Nested Loops

A *nested loop* is one loop inside another loop. A common use for nested loops is to display data in rows and columns. One loop can handle, say, all the columns in a row, and the second loop handles the rows. Listing 6.17 shows a simple example.

Listing 6.17 **The `rows1.c` Program**

```
/* rows1.c -- uses nested loops */
#include <stdio.h>
#define ROWS  6
#define CHARS 10
int main(void)
{
```

```
    int row;
    char ch;

    for (row = 0; row < ROWS; row++)              /* line 10 */
    {
        for (ch = 'A'; ch < ('A' + CHARS); ch++)  /* line 12 */
            printf("%c", ch);
        printf("\n");
    }

    return 0;
}
```

Running the program produces this output:

```
ABCDEFGHIJ
ABCDEFGHIJ
ABCDEFGHIJ
ABCDEFGHIJ
ABCDEFGHIJ
ABCDEFGHIJ
```

Program Discussion

The `for` loop beginning on line 10 is called an *outer* loop, and the loop beginning on line 12 is called an *inner* loop because it is inside the other loop. The outer loop starts with `row` having a value of `0` and terminates when `row` reaches `6`. Therefore, the outer loop goes through six cycles, with `row` having the values `0` through `5`. The first statement in each cycle is the inner `for` loop. This loop goes through 10 cycles, printing the characters `A` through `J` on the same line. The second statement of the outer loop is `printf("\n");`. This statement starts a new line so that the next time the inner loop is run, the output is on a new line.

Note that, with a nested loop, the inner loop runs through its full range of iterations for each single iteration of the outer loop. In the last example, the inner loop prints 10 characters to a row, and the outer loop creates six rows.

A Nested Variation

In the preceding example, the inner loop did the same thing for each cycle of the outer loop. You can make the inner loop behave differently each cycle by making part of the inner loop depend on the outer loop. Listing 6.18, for example, alters the last program slightly by making the starting character of the inner loop depend on the cycle number of the outer loop. It also uses the newer comment style and `const` instead of `#define` to help you get comfortable with both approaches.

Listing 6.18 **The `rows2.c` Program**

```
// rows2.c -- using dependent nested loops
#include <stdio.h>
int main(void)
{
    const int ROWS = 6;
    const int CHARS = 6;
    int row;
    char ch;

    for (row = 0; row < ROWS; row++)
    {
        for (ch = ('A' + row);  ch < ('A' + CHARS); ch++)
            printf("%c", ch);
        printf("\n");
    }

    return 0;
}
```

Here's the output this time:

```
ABCDEF
BCDEF
CDEF
DEF
EF
F
```

Because `row` is added to `'A'` during each cycle of the outer loop, `ch` is initialized in each row to one character later in the alphabet. The test condition, however, is unaltered, so each row still ends on `F`. This results in one fewer character being printed in each row.

Introducing Arrays

Arrays are important features in many programs. They enable you to store several items of related information in a convenient fashion. We will devote all of Chapter 10, "Arrays and Pointers," to arrays, but because arrays are often used with loops, we want to introduce them now.

An *array* is a series of values of the same type, such as 10 `char`s or 15 `int`s, stored sequentially. The whole array bears a single name, and the individual items, or *elements*, are accessed by using an integer index. For example, the declaration

```
float debts[20];
```

announces that `debts` is an array with 20 elements, each of which can hold a type `float` value. The first element of the array is called `debts[0]`, the second element is called `debts[1]`, and so on, up to `debts[19]`. Note that the numbering of array elements starts with 0, not 1. Each element can be assigned a `float` value. For example, you can have the following:

```
debts[5] = 32.54;
debts[6] = 1.2e+21;
```

In fact, you can use an array element the same way you would use a variable of the same type. For example, you can read a value into a particular element:

```
scanf("%f", &debts[4]);  // read a value into the 5th element
```

One potential pitfall is that, in the interest of speed of execution, C doesn't check to see whether you use a correct subscript. Each of the following, for example, is bad code:

```
debts[20] = 88.32;   // no such array element
debts[33] = 828.12;  // no such array element
```

However, the compiler doesn't look for such errors. When the program runs, these statements would place data in locations possibly used for other data, potentially corrupting the output of the program or even causing it to abort.

An array can be of any data type.

```
int nannies[22];   /* an array to hold 22 integers        */
char actors[26];   /* an array to hold 26 characters      */
long big[500];     /* an array to hold 500 long integers  */
```

Earlier, for example, we talked about strings, which are a special case of what can be stored in a `char` array. (A `char` array, in general, is one whose elements are assigned `char` values.) The contents of a `char` array form a string if the array contains the null character, `\0`, which marks the end of the string (see Figure 6.6).

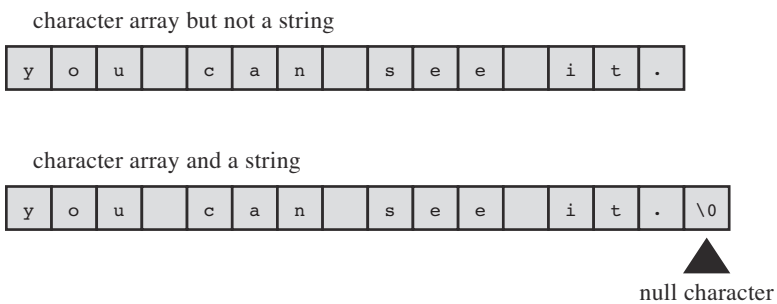

Figure 6.6 Character arrays and strings.

The numbers used to identify the array elements are called *subscripts*, *indices*, or *offsets*. The subscripts must be integers, and, as mentioned, the subscripting begins with 0. The array elements are stored next to each other in memory, as shown in Figure 6.7.

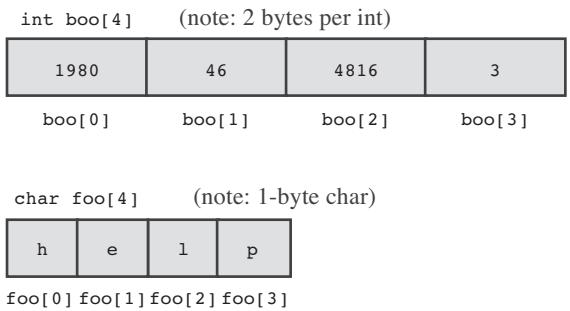

Figure 6.7 The `char` and `int` arrays in memory.

Using a `for` Loop with an Array

There are many, many uses for arrays. Listing 6.19 is a relatively simple one. It's a program that reads in 10 golf scores that will be processed later. By using an array, you avoid the need to invent 10 different variable names, one for each score. Also, you can use a `for` loop to do the reading. The program goes on to report the sum of the scores and their average and a handicap, which is the difference between the average and a standard score, or par.

Listing 6.19 **The `scores_in.c` Program**

```
// scores_in.c -- uses loops for array processing
#include <stdio.h>
#define SIZE 10
#define PAR 72
int main(void)
{
    int index, score[SIZE];
    int sum = 0;
    float average;

    printf("Enter %d golf scores:\n", SIZE);
    for (index = 0; index < SIZE; index++)
        scanf("%d", &score[index]);  // read in the ten scores
    printf("The scores read in are as follows:\n");
    for (index = 0; index < SIZE; index++)
        printf("%5d", score[index]); // verify input
    printf("\n");
```

```
    for (index = 0; index < SIZE; index++)
        sum += score[index];         // add them up
    average = (float) sum / SIZE;    // time-honored method
    printf("Sum of scores = %d, average = %.2f\n", sum, average);
    printf("That's a handicap of %.0f.\n", average - PAR);

    return 0;
}
```

Let's see if Listing 6.19 works; then we can make a few comments. Here is the output:

```
Enter 10 golf scores:
99 95 109 105 100
96 98 93 99 97 98
The scores read in are as follows:
   99   95  109  105  100   96   98   93   99   97
Sum of scores = 991, average = 99.10
That's a handicap of 27.
```

It works, so let's check out some of the details. First, note that although the example shows 11 numbers typed, only 10 were read because the reading loop reads just 10 values. Because `scanf()` skips over whitespace, you can type all 10 numbers on one line, place each number on its own line, or, as in this case, use a mixture of newlines and spaces to separate the input. (Because input is buffered, the numbers are sent to the program only when you press the Enter key.)

Next, using arrays and loops is much more convenient than using 10 separate `scanf()` statements and 10 separate `printf()` statements to read in and verify the 10 scores. The `for` loop offers a simple and direct way to use the array subscripts. Notice that an element of an `int` array is handled like an `int` variable. To read the `int` variable `fue`, you would use `scanf("%d", &fue)`. Listing 6.19 is reading the `int` element `score[index]`, so it uses `scanf("%d", &score[index])`.

This example illustrates several style points. First, it's a good idea to use a `#define` directive to create a manifest constant (`SIZE`) to specify the size of the array. You use this constant in defining the array and in setting the loop limits. If you later need to expand the program to handle 20 scores, simply redefine `SIZE` to be 20. You don't have to change every part of the program that uses the array size. Second, the idiom

```
for (index = 0; index < SIZE; index++)
```

is a handy one for processing an array of size `SIZE`. It's important to get the right array limits. The first element has index `0`, and the loop starts by setting `index` to `0`. Because the numbering starts with `0`, the element index for the last element is `SIZE - 1`. That is, the tenth element is `score[9]`. Using the test condition `index < SIZE` accomplishes this, making the last value of `index` used in the loop `SIZE - 1`.

Third, a good practice is to have a program repeat or "echo" the values it has just read in. This helps ensure that the program is processing the data you think it is.

Finally, note that Listing 6.19 uses three separate `for` loops. You might wonder if this is really necessary. Could you have combined some of the operations in one loop? The answer is yes, you could have done so. That would have made the program more compact. However, you should be swayed by the principle of *modularity*. The idea behind this term is that a program should be broken into separate units, with each unit having one task to perform. This makes a program easier to read. Perhaps even more important, modularity makes it much easier to update or modify a program if different parts of the program are not intermingled. When you know enough about functions, you could make each unit into a function, enhancing the modularity of the program.

A Loop Example Using a Function Return Value

The last example in this chapter uses a function that calculates the result of raising a number to an integer power. (For the serious number-cruncher, the `math.h` library provides a more powerful power function called `pow()` that allows floating-point exponents.) The three main tasks in this exercise are devising the algorithm for calculating the answer, expressing the algorithm in a function that returns the answer, and providing a convenient way of testing the function.

First, let's look at an algorithm. We'll keep the function simple by restricting it to positive integer powers. Then, to raise `n` to the `p` power, just multiply `n` times itself `p` times. This is a natural task for a loop. You can set the variable `pow` to `1` and then repeatedly multiply it by `n`:

```
for(i = 1; i <= p; i++)
     pow *= n;
```

Recall that the `*=` operator multiplies the left side by the right side. After the first loop cycle, `pow` is `1` times `n`, or `n`. After the second cycle, `pow` is its previous value (`n`) times `n`, or `n` squared, and so on. The `for` loop is natural in this context because the loop is executed a predetermined (after `p` is known) number of times.

Now that we have an algorithm, we can decide which data types to use. The exponent `p`, being an integer, should be type `int`. To allow ample range in values for `n` and its power, make `n` and `pow` type `double`.

Next, let's consider how to put the function together. We need to give the function two values, and the function should give back one. To get information to the function, we can use two arguments, one `double` and one `int`, specifying which number to raise to what power. How do we arrange for the function to return a value to the calling program? To write a function with a return value, do the following:

1. When you define a function, state the type of value it returns.
2. Use the keyword `return` to indicate the value to be returned.

For example, we can do this:

```
double power(double n, int p)  // returns a double
{
    double pow = 1;
    int i;

    for (i = 1; i <= p; i++)
        pow *= n;

    return pow;                // return the value of pow
}
```

To declare the function type, preface the function name with the type, just as you do when declaring a variable. The keyword `return` causes the function to return the following value to the calling function. Here the function returns the value of a variable, but it can return the value of expressions, too. For instance, the following is a valid statement:

```
return 2 * x + b;
```

The function would compute the value of the expression and return it. In the calling function, the return value can be assigned to another variable, can be used as a value in an expression, can be used as an argument to another function—as in `printf("%f", power(6.28, 3))`—or can be ignored.

Now let's use the function in a program. To test the function, it would be convenient to be able to feed several values to the function to see how it reacts. This suggests setting up an input loop. The natural choice is the `while` loop. You can use `scanf()` to read in two values at a time. If successful in reading two values, `scanf()` returns the value `2`, so you can control the loop by comparing the `scanf()` return value to 2. One more point: To use the `power()` function in your program, you need to declare it, just as you declare variables that the program uses. Listing 6.20 shows the program.

Listing 6.20 **The `power.c` Program**

```
// power.c -- raises numbers to integer powers
#include <stdio.h>
double power(double n, int p); // ANSI prototype
int main(void)
{
    double x, xpow;
    int exp;

    printf("Enter a number and the positive integer power");
    printf(" to which\nthe number will be raised. Enter q");
    printf(" to quit.\n");
    while (scanf("%lf%d", &x, &exp) == 2)
    {
```

```
        xpow = power(x,exp);   // function call
        printf("%.3g to the power %d is %.5g\n", x, exp, xpow);
        printf("Enter next pair of numbers or q to quit.\n");
    }
    printf("Hope you enjoyed this power trip -- bye!\n");

    return 0;
}

double power(double n, int p)  // function definition
{
    double pow = 1;
    int i;

    for (i = 1; i <= p; i++)
        pow *= n;

    return pow;                // return the value of pow
}
```

Here is a sample run:

```
Enter a number and the positive integer power to which
the number will be raised. Enter q to quit.
1.2 12
1.2 to the power 12 is 8.9161
Enter next pair of numbers or q to quit.
2
16
2 to the power 16 is 65536
Enter next pair of numbers or q to quit.
q
Hope you enjoyed this power trip -- bye!
```

Program Discussion

The `main()` program is an example of a *driver*, a short program designed to test a function.

The `while` loop is a generalization of a form we've used before. Entering `1.2 12` causes `scanf()` to read two values successfully and to return `2`, and the loop continues. Because `scanf()` skips over whitespace, input can be spread over more than one line, as the sample output shows, but entering `q` produces a return value of `0` because `q` can't be read using the `%lf` specifier. This causes `scanf()` to return `0`, thus terminating the loop. Similarly, entering `2.8 q` would produce a `scanf()` return value of `1`; that, too, would terminate the loop.

Now let's look at the function-related matters. The `power()` function appears three times in this program. The first appearance is this:

```
double power(double n, int p); // ANSI prototype
```

This statement announces, or *declares*, that the program will be using a function called `power()`. The initial keyword `double` indicates that the `power()` function returns a type `double` value. The compiler needs to know what kind of value `power()` returns so that it will know how many bytes of data to expect and how to interpret them; this is why you have to declare the function. The `double n, int p` within the parentheses means that `power()` takes two arguments. The first should be a type `double` value, and the second should be type `int`.

The second appearance is this:

```
xpow = power(x,exp);           // function call
```

Here the program calls the function, passing it two values. The function calculates `x` to the `exp` power and returns the result to the calling program, where the return value is assigned to the variable `xpow`.

The third appearance is in the head of the function definition:

```
double power(double n, int p)  // function definition
```

Here `power()` takes two parameters, a `double` and an `int`, represented by the variables `n` and `p`. Note that `power()` is not followed by a semicolon when it appears in a function definition, but is followed by a semicolon when in a function declaration. After the function heading comes the code that specifies what `power()` does.

Recall that the function uses a `for` loop to calculate the value of `n` to the `p` power and assign it to `pow`. The following line makes the value of `pow` the function return value:

```
return pow;                    // return the value of pow
```

Using Functions with Return Values

Declaring the function, calling the function, defining the function, using the `return` keyword—these are the basic elements in defining and using a function with a return value.

At this point, you might have some questions. For example, if you are supposed to declare functions before you use their return values, how come you used the return value of `scanf()` without declaring `scanf()`? Why do you have to declare `power()` separately when your definition of it says it is type `double`?

Let's take the second question first. The compiler needs to know what type `power()` is when it first encounters `power()` in the program. At this point, the compiler has not yet encountered the definition of `power()`, so it doesn't know that the definition says the return type is `double`. To help out the compiler, you preview what is to come by using a *forward declaration*. This declaration informs the compiler that `power()` is defined elsewhere and that it will return type `double`. If you place the `power()` function definition ahead of `main()` in the file, you can

omit the forward declaration because the compiler will know all about `power()` before reaching `main()`. However, that is not standard C style. Because `main()` usually provides the overall framework for a program, it's best to show `main()` first. Also, functions often are kept in separate files, so a forward declaration is essential.

Next, why didn't you declare `scanf()`? Well, you did. The `stdio.h` header file has function declarations for `scanf()`, `printf()`, and several other I/O functions. The `scanf()` declaration states that it returns type `int`.

Key Concepts

The loop is a powerful programming tool. You should pay particular attention to three aspects when setting up a loop:

- Clearly defining the condition that causes the loop to terminate
- Making sure the values used in the loop test are initialized before the first use
- Making sure the loop does something to update the test each cycle

C handles test conditions by evaluating them numerically. A result of `0` is false, and any other value is true. Expressions using the relational operators often are used as tests, and they are a bit more specific. Relational expressions evaluate to `1` if true and to `0` if false, which is consistent with the values allowed for the new `_Bool` type.

Arrays consist of adjacent memory locations all of the same type. You need to keep in mind that array element numbering starts with 0 so that the subscript of the last element is always one less than the number of elements. C doesn't check to see if you use valid subscript values, so the responsibility is yours.

Employing a function involves three separate steps:

1. Declare the function with a function prototype.
2. Use the function from within a program with a function call.
3. Define the function.

The prototype allows the compiler to see whether you've used the function correctly, and the definition sets down how the function works. The prototype and definition are examples of the contemporary programming practice of separating a program element into an interface and an implementation. The interface describes how a feature is used, which is what a prototype does, and the implementation sets forth the particular actions taken, which is what the definition does.

Summary

The main topic of this chapter has been program control. C offers you many aids for structuring your programs. The `while` and the `for` statements provide entry-condition loops. The `for` statements are particularly suited for loops that involve initialization and updating. The comma operator enables you to initialize and update more than one variable in a `for` loop. For the less common occasion when an exit-condition loop is needed, C has the `do while` statement.

A typical `while` loop design looks like this:

```
get first value
while (value meets test)
{
    process the value
    get next value
}
```

A `for` loop doing the same thing would look like this:

```
for (get first value; value meets test; get next value)
    process the value
```

All these loops use a test condition to determine whether another loop cycle is to be executed. In general, the loop continues if the test expression evaluates to a nonzero value; otherwise, it terminates. Often, the test condition is a relational expression, which is an expression formed by using a relational operator. Such an expression has a value of `1` if the relation is true and a value of `0` otherwise. Variables of the `_Bool` type, introduced by C99, can only hold the value `1` or `0`, signifying true or false.

In addition to relational operators, this chapter looked at several of C's arithmetic assignment operators, such as `+=` and `*=`. These operators modify the value of the left-hand operand by performing an arithmetic operation on it.

Arrays were the next subject. Arrays are declared using brackets to indicate the number of elements. The first element of an array is numbered 0; the second is numbered 1, and so forth. For example, the declaration

```
double hippos[20];
```

creates an array of 20 elements, and the individual elements range from `hippos[0]` through `hippos[19]`. The subscripts used to number arrays can be manipulated conveniently by using loops.

Finally, the chapter showed how to write and use a function with a return value.

Review Questions

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

1. Find the value of `quack` after each line; each of the final five statements uses the value of `quack` produced by the preceding statement.

```
int quack = 2;
quack += 5;
quack *= 10;
quack -= 6;
quack /= 8;
quack %= 3;
```

2. Given that `value` is an `int`, what output would the following loop produce?

```
for ( value = 36; value > 0; value /= 2)
      printf("%3d", value);
```

   What problems would there be if `value` were `double` instead of `int`?

3. Represent each of the following test conditions:
   a. `x` is greater than `5`.
   b. `scanf()` attempts to read a single `double` (called `x`) and fails.
   c. `x` has the value `5`.

4. Represent each of the following test conditions:
   a. `scanf()` succeeds in reading a single integer.
   b. `x` is not `5`.
   c. `x` is `20` or greater.

5. You suspect that the following program is not perfect. What errors can you find?

```
#include <stdio.h>
int main(void)
{                                       /* line 3  */
  int i, j, list(10);                   /* line 4  */

  for (i = 1, i <= 10,  i++)            /* line 6  */
  {                                     /* line 7  */
      list[i] = 2*i + 3;                /* line 8  */
      for (j = 1, j > = i, j++)         /* line 9  */
          printf(" %d", list[j]);       /* line 10 */
```

```
        printf("\n");                          /* line 11 */
}                                              /* line 12 */
```

6. Use nested loops to write a program that produces this pattern:

```
$$$$$$$$
$$$$$$$$
$$$$$$$$
$$$$$$$$
```

7. What will each of the following programs print?

   a.

```
#include <stdio.h>

int main(void)
{
    int i = 0;

    while (++i < 4)
       printf("Hi! ");
    do
       printf("Bye! ");
    while (i++ < 8);
    return 0;
}
```

   b.

```
#include <stdio.h>

int main(void)
{
     int i;
     char ch;

     for (i = 0, ch = 'A'; i < 4; i++, ch += 2 * i)
            printf("%c", ch);
     return 0;
}
```

8. Given the input `Go west, young man!`, what would each of the following programs produce for output? (The `!` follows the space character in the ASCII sequence.)

   a.

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

```
int main(void)
{
    char ch;

    scanf("%c", &ch);
    while ( ch != 'g' )
    {
         printf("%c", ch);
         scanf("%c", &ch);
    }
    return 0;
}
```

b.

```
#include <stdio.h>

int main(void)
{
    char ch;

    scanf("%c", &ch);
    while ( ch != 'g' )
    {
         printf("%c", ++ch);
         scanf("%c", &ch);
    }
    return 0;
}
```

c.

```
#include <stdio.h>

int main(void)
{
    char ch;

    do {
         scanf("%c", &ch);
         printf("%c", ch);
    } while ( ch != 'g' );
    return 0;
}
```

d.

```
#include <stdio.h>

int main(void)
```

```
{
    char ch;

    scanf("%c", &ch);
    for ( ch = '$'; ch != 'g'; scanf("%c", &ch) )
          printf("%c", ch);
    return 0;
}
```

9. What will the following program print?

```
#include <stdio.h>
int main(void)
{
     int n, m;

     n = 30;
     while (++n <= 33)
          printf("%d|",n);

     n = 30;
     do
          printf("%d|",n);
     while (++n <= 33);

     printf("\n***\n");

     for (n = 1; n*n < 200; n += 4)
          printf("%d\n", n);

     printf("\n***\n");

     for (n = 2, m = 6; n < m; n *= 2, m+= 2)
          printf("%d %d\n", n, m);

     printf("\n***\n");

     for (n = 5; n > 0; n--)
     {
          for (m = 0; m <= n; m++)
               printf("=");
          printf("\n");
     }
     return 0;
}
```

10. Consider the following declaration:

```
double mint[10];
```

    a. What is the array name?
    b. How many elements does the array have?
    c. What kind of value can be stored in each element?
    d. Which of the following is a correct usage of `scanf()` with this array?
        i. `scanf("%lf", mint[2])`
        ii. `scanf("%lf", &mint[2])`
        iii. `scanf("%lf", &mint)`

11. Mr. Noah likes counting by twos, so he's written the following program to create an array and to fill it with the integers 2, 4, 6, 8, and so on. What, if anything, is wrong with this program?

```
#include <stdio.h>
#define SIZE 8
int main(void)
{
  int by_twos[SIZE];
  int index;

  for (index = 1; index <= SIZE; index++)
       by_twos[index] = 2 * index;
  for (index = 1; index <= SIZE; index++)
       printf("%d ", by_twos);
  printf("\n");
  return 0;
}
```

12. You want to write a function that returns a `long` value. What should your definition of the function include?

13. Define a function that takes an `int` argument and that returns, as a `long`, the square of that value.

14. What will the following program print?

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

```
    for(k = 1, printf("%d: Hi!\n", k); printf("k = %d\n",k),
        k*k < 26; k+=2, printf("Now k is %d\n", k) )
            printf("k is %d in the loop\n",k);
    return 0;
}
```

Programming Exercises

1. Write a program that creates an array with 26 elements and stores the 26 lowercase letters in it. Also have it show the array contents.

2. Use nested loops to produce the following pattern:

   ```
   $
   $$
   $$$
   $$$$
   $$$$$
   ```

3. Use nested loops to produce the following pattern:

   ```
   F
   FE
   FED
   FEDC
   FEDCB
   FEDCBA
   ```

   Note: If your system doesn't use ASCII or some other code that encodes letters in numeric order, you can use the following to initialize a character array to the letters of the alphabet:

   ```
   char lets[27] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   ```

   Then you can use the array index to select individual letters; for example, `lets[0]` is `'A'`, and so on.

4. Use nested loops to produce the following pattern:

   ```
   A
   BC
   DEF
   GHIJ
   KLMNO
   PQRSTU
   ```

If your system doesn't encode letters in numeric order, see the suggestion in programming exercise 3.

5. Have a program request the user to enter an uppercase letter. Use nested loops to produce a pyramid pattern like this:

```
    A
   ABA
  ABCBA
 ABCDCBA
ABCDEDCBA
```

The pattern should extend to the character entered. For example, the preceding pattern would result from an input value of `E`. Hint: Use an outer loop to handle the rows. Use three inner loops in a row, one to handle the spaces, one for printing letters in ascending order, and one for printing letters in descending order. If your system doesn't use ASCII or a similar system that represents letters in strict number order, see the suggestion in programming exercise 3.

6. Write a program that prints a table with each line giving an integer, its square, and its cube. Ask the user to input the lower and upper limits for the table. Use a `for` loop.

7. Write a program that reads a single word into a character array and then prints the word backward. Hint: Use `strlen()` (Chapter 4) to compute the index of the last character in the array.

8. Write a program that requests two floating-point numbers and prints the value of their difference divided by their product. Have the program loop through pairs of input values until the user enters nonnumeric input.

9. Modify exercise 8 so that it uses a function to return the value of the calculation.

10. Write a program that requests lower and upper integer limits, calculates the sum of all the integer squares from the square of the lower limit to the square of the upper limit, and displays the answer. The program should then continue to prompt for limits and display answers until the user enters an upper limit that is equal to or less than the lower limit. A sample run should look something like this:

```
Enter lower and upper integer limits: 5 9
The sums of the squares from 25 to 81 is 255
Enter next set of limits: 3 25
The sums of the squares from 9 to 625 is 5520
Enter next set of limits: 5 5
Done
```

11. Write a program that reads eight integers into an array and then prints them in reverse order.

12. Consider these two infinite series:

```
1.0 + 1.0/2.0 + 1.0/3.0 + 1.0/4.0 + ...
1.0 - 1.0/2.0 + 1.0/3.0 - 1.0/4.0 + ...
```

Write a program that evaluates running totals of these two series up to some limit of number of terms. Hint: –1 times itself an odd number of times is –1, and –1 times itself an even number of times is 1. Have the user enter the limit interactively; let a zero or negative value terminate input. Look at the running totals after 100 terms, 1000 terms, 10,000 terms. Does either series appear to be converging to some value?

13. Write a program that creates an eight-element array of `int`s and sets the elements to the first eight powers of 2 and then prints the values. Use a `for` loop to set the values, and, for variety, use a `do while` loop to display the values.

14. Write a program that creates two eight-element arrays of `double`s and uses a loop to let the user enter values for the eight elements of the first array. Have the program set the elements of the second array to the cumulative totals of the elements of the first array. For example, the fourth element of the second array should equal the sum of the first four elements of the first array, and the fifth element of the second array should equal the sum of the first five elements of the first array. (It's possible to do this with nested loops, but by using the fact that the fifth element of the second array equals the fourth element of the second array plus the fifth element of the first array, you can avoid nesting and just use a single loop for this task.) Finally, use loops to display the contents of the two arrays, with the first array displayed on one line and with each element of the second array displayed below the corresponding element of the first array.

15. Write a program that reads in a line of input and then prints the line in reverse order. You can store the input in an array of `char`; assume that the line is no longer than 255 characters. Recall that you can use `scanf()` with the `%c` specifier to read a character at a time from input and that the newline character (`\n`) is generated when you press the Enter key.

16. Daphne invests $100 at 10% simple interest. (That is, every year, the investment earns an interest equal to 10% of the original investment.) Deirdre invests $100 at 5% interest compounded annually. (That is, interest is 5% of the current balance, including previous addition of interest.) Write a program that finds how many years it takes for the value of Deirdre's investment to exceed the value of Daphne's investment. Also show the two values at that time.

17. Chuckie Lucky won a million dollars (after taxes), which he places in an account that earns 8% a year. On the last day of each year, Chuckie withdraws $100,000. Write a program that finds out how many years it takes for Chuckie to empty his account.

18. Professor Rabnud joined a social media group. Initially he had five friends. He noticed that his friend count grew in the following fashion. The first week one friend dropped out and the remaining number of friends doubled. The second week two friends dropped out and the remaining number of friends doubled. In general, in the Nth week, N friends dropped out and the remaining number doubled. Write a program that computes and displays the number of friends each week. The program should continue until the count exceeds Dunbar's number. Dunbar's number is a rough estimate of the maximum size of a cohesive social group in which each member knows every other member and how they relate to one another. Its approximate value is 150.