02_04_C Program Control

Objectives

In this chapter, you'll learn:

- The essentials of counter-controlled iteration.
- To use the `for` and `do`...`while` iteration statements to execute statements repeatedly.
- To understand multiple selection using the `switch` selection statement.
- To use the `break` and `continue` statements to alter the flow of control.
- To use the logical operators to form complex conditional expressions in control statements.
- To avoid the consequences of confusing the equality and assignment operators.

```c
// Fig. 4.1: fig04_01.c
// Counter-controlled iteration.
#include <stdio.h>

int main(void)
{
   unsigned int counter = 1; // initialization

   while (counter <= 10) { // iteration condition
      printf ("%u\n", counter);
      ++counter; // increment
   }
}
```

```
1
2
3
4
5
6
7
8
9
10
```

Fig. 4.1 | Counter-controlled iteration.

```c
// Fig. 4.2: fig04_02.c
// Counter-controlled iteration with the for statement.
#include <stdio.h>

int main(void)
{
   // initialization, iteration condition, and increment
   //  are all included in the for statement header.
   for (unsigned int counter = 1; counter <= 10; ++counter) {
      printf("%u\n", counter);
   }
}
```

Fig. 4.2 | Counter-controlled iteration with the `for` statement.

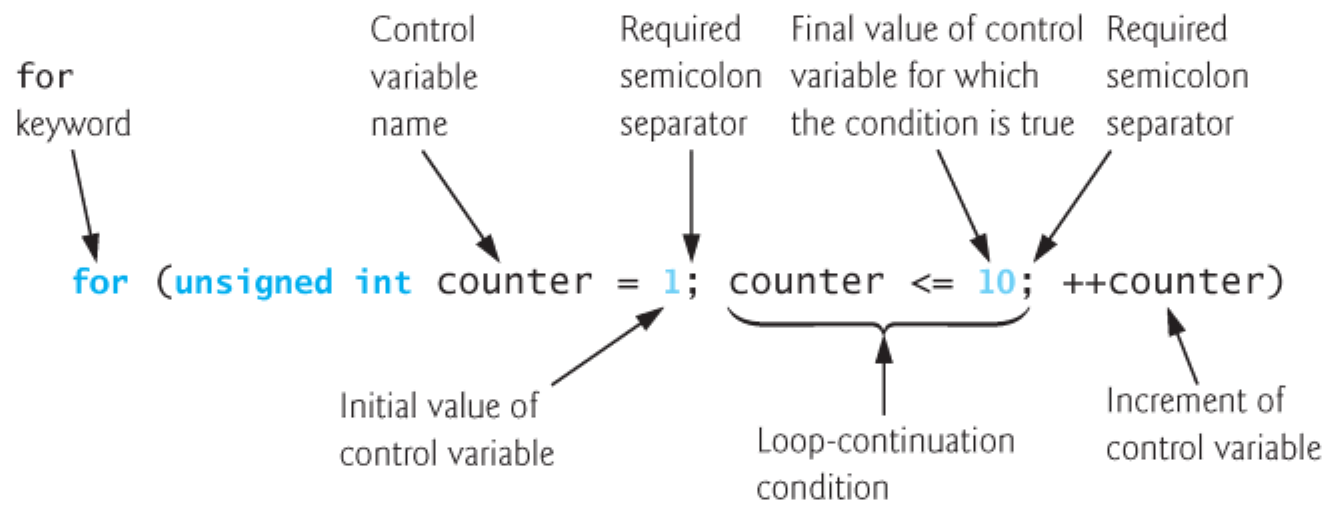

Fig. 4.3 | `for` statement header components.

```c
// Fig. 4.5: fig04_05.c
// Summation with for.
#include <stdio.h>

int main(void)
{
   unsigned int sum = 0; // initialize sum

   for (unsigned int number = 2; number <= 100; number += 2) {
      sum += number; // add number to sum
   }

   printf("Sum is %u\n", sum);
}
```

```
Sum is 2550
```

Fig. 4.5 | Summation with for.

Examples Using the `for` Statement

Application: Compound-Interest Calculations

- Consider the following problem statement:
  - A person invests $1000.00 in a savings account yielding 5% interest. Assuming that all interest is left on deposit in the account, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula for determining these amounts:

    $a = p(1 + r)^n$

    where

    p is the original amount invested (i.e., the principal)
    r is the annual interest rate
    n is the number of years
    a is the amount on deposit at the end of the n^{th} year.

- This problem involves a loop that performs the indicated calculation for each of the 10 years the money remains on deposit.

```c
// Fig. 4.6: fig04_06.c
// Calculating compound interest.
#include <stdio.h>
#include <math.h>

int main(void)
{
   double principal = 1000.0; // starting principal
   double rate = .05; // annual interest rate

   // output table column heads
   printf("%4s%21s\n", "Year", "Amount on deposit");

   // calculate amount on deposit for each of ten years
   for (unsigned int year = 1; year <= 10; ++year) {

      // calculate new amount for specified year
      double amount = principal * pow(1.0 + rate, year);

      // output one table row
      printf("%4u%21.2f\n", year, amount);
   }
}
```

Fig. 4.6 | Calculating compound interest. (Part 1 of 2.)

```
Year    Amount on deposit
   1              1050.00
   2              1102.50
   3              1157.63
   4              1215.51
   5              1276.28
   6              1340.10
   7              1407.10
   8              1477.46
   9              1551.33
  10              1628.89
```

Fig. 4.6 | Calculating compound interest. (Part 2 of 2.)

Examples Using the `for` Statement

Formatting Numeric Output

- The conversion specifier **`%21.2f`** is used to print the value of the variable amount in the program.
- The **`21`** in the conversion specifier denotes the *field width* in which the value will be printed.
- A field width of **`21`** specifies that the value printed will appear in **`21`** print positions.
- The **`2`** specifies the *precision* (i.e., the number of decimal positions).

Examples Using the `for` Statement (Cont.)

- If the number of characters displayed is less than the field width, then the value will automatically be *right justified* in the field.

- This is particularly useful for aligning floating-point values with the same precision (so that their decimal points align vertically).

- To *left justify* a value in a field, place a - (minus sign) between the % and the field width.

- The minus sign may also be used to left justify integers (such as in **`%-6d`**) and character strings (such as in **`%-8s`**).

```c
// Fig. 4.7: fig04_07.c
// Counting letter grades with switch.
#include <stdio.h>

int main(void)
{
   unsigned int aCount = 0;
   unsigned int bCount = 0;
   unsigned int cCount = 0;
   unsigned int dCount = 0;
   unsigned int fCount = 0;

   puts("Enter the letter grades.");
   puts("Enter the EOF character to end input.");
   int grade; // one grade

```

Fig. 4.7 | Counting letter grades with `switch`. (Part 1 of 5.)

```
    // loop until user types end-of-file key sequence
    while ((grade = getchar()) != EOF) {

        // determine which grade was input
        switch (grade) { // switch nested in while

            case 'A': // grade was uppercase A
            case 'a': // or lowercase a
                ++aCount;
                break; // necessary to exit switch

            case 'B': // grade was uppercase B
            case 'b': // or lowercase b
                ++bCount;
                break;

            case 'C': // grade was uppercase C
            case 'c': // or lowercase c
                ++cCount;
                break;

```

Fig. 4.7 | Counting letter grades with `switch`. (Part 2 of 5.)

```c
         case 'D': // grade was uppercase D
         case 'd': // or lowercase d
            ++dCount;
            break;

         case 'F': // grade was uppercase F
         case 'f': // or lowercase f
            ++fCount;
            break;

         case '\n': // ignore newlines,
         case '\t': // tabs,
         case ' ': // and spaces in input
            break;

         default: // catch all other characters
            printf("%s", "Incorrect letter grade entered.");
            puts(" Enter a new grade.");
            break; // optional; will exit switch anyway
      }
   } // end while

```

Fig. 4.7 | Counting letter grades with `switch`. (Part 3 of 5.)

```c
    // output summary of results
    puts("\nTotals for each letter grade are:");
    printf("A: %u\n", aCount);
    printf("B: %u\n", bCount);
    printf("C: %u\n", cCount);
    printf("D: %u\n", dCount);
    printf("F: %u\n", fCount);
}
```

Fig. 4.7 | Counting letter grades with `switch`. (Part 4 of 5.)

```
Enter the letter grades.
Enter the EOF character to end input.
a
b
c
C
A
d
f
C
E
Incorrect letter grade entered. Enter a new grade.
D
A
b
^Z
```

Not all systems display a representation of the EOF character

```
Totals for each letter grade are:
A: 3
B: 2
C: 3
D: 2
F: 1
```

Fig. 4.7 | Counting letter grades with `switch`. (Part 5 of 5.)

`switch` Multiple-Selection Statement

Reading Character Input

- Unlike some other languages you may have used, chars in C are integers. **`char`** is just another integer type, usually 8 bits and smaller than **`int`**, but still an integer type.
- In C you can convert between char and other integer types using a cast, or just by assigning.
- Unless EOF occurs**,** **`getchar()`** is defined to return "an unsigned char converted to an int" (same as fgetc), so if it helps you can imagine that it reads some **`char`**, **`c`**, then returns **`(int)(unsigned char)c`**.

`switch` Multiple-Selection Statement

Reading Character Input

- The **`getchar`** function (from **`<stdio.h>`**) reads one character from the keyboard and returns as an **`int`** the character that the user entered.
- Characters are normally stored in variables of type **`char`**.
- However, an important feature of C is that characters can be stored in any integer data type because they’re usually represented as one-byte integers in the computer.

`switch` Multiple-Selection Statement (Cont.)

- Thus, we can treat a character as either an integer or a character, depending on its use.
- For example, the statement

```
printf("The character (%c) has the value %d.\n", 'a', 'a');
```

- uses the conversion specifiers **`%c`** and **`%d`** to print the character **`a`** and its integer value, respectively.
- The result is

```
The character (a) has the value 97.
```

- The integer 97 is the character's numerical representation in the computer.

`switch` Multiple-Selection Statement (Cont.)

- Many computers today use the ASCII (American Standard Code for Information Interchange) character set in which 97 represents the lowercase letter 'a'.

- Characters can be read with **`scanf`** by using the conversion specifier **`%c`**.

- In the program, the value of the assignment **`grade = getchar()`** is compared with the value of **`EOF`** (a symbol whose acronym stands for “end of file”).

`switch` Multiple-Selection Statement (Cont.)

- We use **`EOF`** (which normally has the value `-1`) as the sentinel value.
- The user types a system-dependent keystroke combination to mean “end of file”—i.e., “I have no more data to enter.”
- On Linux, ctrl+d signals EOF, and on Windows it's ctrl+z.
- **`EOF`** is a symbolic integer constant defined in the **`<stdio.h>`** header (we’ll see in Chapter 6 how symbolic constants are defined).
- If the value assigned to **`grade`** is equal to **`EOF`**, the program terminates.
- We’ve chosen to represent characters in this program as **`int`**s because **`EOF`** has an integer value (normally **`-1`**).

```c
// Fig. 4.9: fig04_09.c
// Using the do...while iteration statement.
#include <stdio.h>

int main(void)
{
   unsigned int counter = 1; // initialize counter

   do {
      printf("%u  ", counter);
   } while (++counter <= 10);
}
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 4.9 | Using the do...while iteration statement.

```c
// Fig. 4.11: fig04_11.c
// Using the break statement in a for statement.
#include <stdio.h>

int main(void)
{
   unsigned int x; // declared here so it can be used after loop

   // loop 10 times
   for (x = 1; x <= 10; ++x) {

      // if x is 5, terminate loop
      if (x == 5) {
         break; // break loop only if x is 5
      }

      printf("%u ", x);
   }

   printf("\nBroke out of loop at x == %u\n", x);
}
```

```
1 2 3 4
Broke out of loop at x == 5
```

Fig. 4.11 | Using the `break` statement in a `for` statement.

```c
// Fig. 4.12: fig04_12.c
// Using the continue statement in a for statement.
#include <stdio.h>

int main(void)
{
   // loop 10 times
   for (unsigned int x = 1; x <= 10; ++x) {

      // if x is 5, continue with next iteration of loop
      if (x == 5) {
         continue; // skip remaining code in loop body
      }

      printf("%u ", x);
   }

   puts("\nUsed continue to skip printing the value 5");
}
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing the value 5
```

Fig. 4.12 | Using the `continue` statement in a `for` statement.

Operators	Associativity	Type
++ *(postfix)* -- *(postfix)*	right to left	postfix
+ - ! ++ *(prefix)* -- *(prefix)* *(type)*	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
&&	left to right	logical AND
||	left to right	logical OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment
,	left to right	comma

Fig. 4.16 | Operator precedence and associativity.

Confusing Equality (==) and Assignment (=) Operators

- There’s one type of error that C programmers, no matter how experienced, tend to make so frequently that we felt it was worth a separate section.

- That error is accidentally swapping the operators == (equality) and = (assignment).

- *What makes these swaps so damaging is the fact that they do not ordinarily cause compilation errors.*

- Rather, statements with these errors ordinarily compile correctly, allowing programs to run to completion while likely generating incorrect results through *runtime logic errors*.

Confusing Equality (==) and Assignment (=) Operators (Cont.)

- Two aspects of C cause these problems.

- One is that any expression in C that produces a value can be used in the decision portion of any control statement.

- If the value is 0, it's treated as false, and if the value is nonzero, it's treated as true.

- The second is that assignments in C produce a value, namely the value that's assigned to the variable on the left side of the assignment operator.

Confusing Equality (==) and Assignment (=) Operators (Cont.)

- For example, suppose we intend to write

```
if (payCode == 4)
    printf(“%s“, "You get a bonus!");
```

but we accidentally write

```
if (payCode = 4)
    printf(“%s“, "You get a bonus!");
```

- The first `if` statement properly awards a bonus to the person whose `paycode` is equal to `4`.
- The second `if` statement—the one with the error—evaluates the assignment expression in the `if` condition.

Confusing Equality (==) and Assignment (=) Operators (Cont.)

- This expression is a simple assignment whose value is the constant 4.

- Because any nonzero value is interpreted as “true,” the condition in this **`if`** statement is always true, and not only is the value of **`payCode`** inadvertently set to 4, but the person always receives a bonus regardless of what the actual paycode is!