# Chapter 6. Decision and Control Statements

*Once a decision was made, I did not worry about it afterward.*

—Harry Truman

Calculations and expressions are only a small part of computer programming. Decision and control statements are needed. They specify the order in which statements are to be executed.

So far, we have constructed *linear programs*, that is, programs that execute in a straight line, one statement after another. In this chapter, we will see how to change the *control flow* of a program with *branching statements* and *looping statements*. Branching statements cause one section of code to be executed or not executed, depending on a *conditional clause*. Looping statements are used to repeat a section of code a number of times or until some condition occurs.

## 6.1 if Statement

The **if** statement allows us to put some decision-making into our programs. The general form of the **if** statement is:

```
if (condition)
    statement;
```

If the condition is true (nonzero), the statement will be executed. If the condition is false (0), the statement will not be executed. For example, suppose we are writing a billing program. At the end, if the customer owes us nothing or has a credit (owes us a negative amount), we want to print a message. In C, this program is written:

```
if (total_owed <= 0)
   printf("You owe nothing.\n");
```

The operator <= is a relational operator that represents *less than or equal to*. This statement reads "if the `total_owed` is less than or equal to zero, print the message." The complete list of relational operators is found in Table 6-1.

**Table 6-1. Relational Operators**

| Operator | Meaning |
|---|---|
| <= | Less than or equal to |
| < | Less than |
| > | Greater than |
| >= | Greater than or equal to |
| == | Equal[1] |
| != | Not equal |

[1] The equal test (==) is different from the assignment operator (=). One of the most common problems the C programmer faces is mixing them up.

Multiple statements may be grouped by putting them inside curly braces ({}). For example:

```
if (total_owed <= 0) {
    ++zero_count;
    printf("You owe nothing.\n");
}
```

For readability, the statements enclosed in {} are usually indented. This allows the programmer to quickly tell which statements are to be conditionally executed. As we will see later, mistakes in indentation can result in programs that are misleading and hard to read.

## 6.2 else Statement

An alternate form of the **if** statement is:

```
if (condition)
    statement;
else
    statement;
```

If the condition is true (nonzero), the first statement is executed. If it is false (0), the second statement is executed. In our accounting example, we wrote out a message only if nothing was owed. In real life, we probably would want to tell the customer how much is owed if there is a balance due:

```
if (total_owed <= 0)
```

```
    printf("You owe nothing.\n");
else
    printf("You owe %d dollars\n", total_owed);
```

Now, consider this program fragment (with incorrect indentation):

```
if (count < 10)     /* if #1 */
    if ((count % 4) == 2)   /* if #2 */
       printf("Condition:White\n");
  else
     printf("Condition:Tan\n");
```

Note to PASCAL programmers: unlike PASCAL, C requires you to put a semicolon at the end of the statement preceding **else**.

There are two **if** statements and one **else**. Which **if** does the **else** belong to?

1. It belongs to **if** #1.
2. It belongs to **if** #2.
3. If you never write code like this, don't worry about this situation.

The correct answer is "c." According to the C syntax rules, the **else** goes with the nearest **if**, so "b" is syntactically correct. But writing code like this violates the KISS principle (Keep It Simple, Stupid). We should write code as clearly and simply as possible. This code fragment should be written as:

```
if (count < 10) {       /* if #1 */
    if ((count % 4) == 2)   /* if #2 */
       printf("Condition:White\n");
    else
       printf("Condition:Tan\n");
}
```

In our original example, we could not clearly determine which **if** statement had the **else** clause; however, by adding an extra set of braces, we improve readability, understanding, and clarity.

## 6.3 How Not to Use strcmp

The function `strcmp` compares two strings, and then returns zero if they are equal or nonzero if they are different. To check if two strings are equal, we use the code:

```
/* Check to see if string1 == string2 */
```

```
if (strcmp(string1, string2) == 0)
    printf("Strings equal\n");
else
    printf("Strings not equal\n");
```

Some programmers omit the comment and the `== 0` clause. These omissions lead to the following confusing code:

```
if (strcmp(string1, string2))
    printf("......");
```

At first glance, this program obviously compares two strings and executes the `printf` statement if they are equal. Unfortunately, the obvious is wrong. If the strings are equal, `strcmp` returns 0, and the `printf` is not executed. Because of this backward behavior of `strcmp`, you should be very careful in your use of `strcmp` and always comment its use. (It also helps to put in a comment explaining what you're doing.)

## 6.4 Looping Statements

Looping statements allow the program to repeat a section of code any number of times or until some condition occurs. For example, loops are used to count the number of words in a document or to count the number of accounts that have past-due balances.

## 6.5 while Statement

The **while** statement is used when the program needs to perform repetitive tasks. The general form of a **while** statement is:

```
while (condition)
    statement;
```

The program will repeatedly execute the statement inside the **while** until the condition becomes false (0). (If the condition is initially false, the statement will not be executed.)

For example, Example 6 -1 later in this chapter will compute all the Fibonacci numbers that are less than 100. The Fibonacci sequence is:

```
1 1 2 3 5 8
```

The terms are computed from the equations:

```
1
1
2 = 1 + 1
3 = 1 + 2
5 = 2 + 3
etc.
```

In general terms this is:

$f_n = f_{n-1} + f_{n-2}$

This is a mathematical equation using mathematical variable names ($f_n$). Mathematicians use this very terse style of naming variables. In programming, terse is dangerous, so we translate these names into something verbose for C. Table 6-2 shows this translation.

**Table 6-2. Math to C Name Translation**

| Math-style name | C-style name |
|---|---|
| $f_n$ | `next_number` |
| $f_{n-1}$ | `current_number` |
| $f_{n-2}$ | `old_number` |

In C code, the equation is expressed as:

```
next_number = current_number + old_number;
```

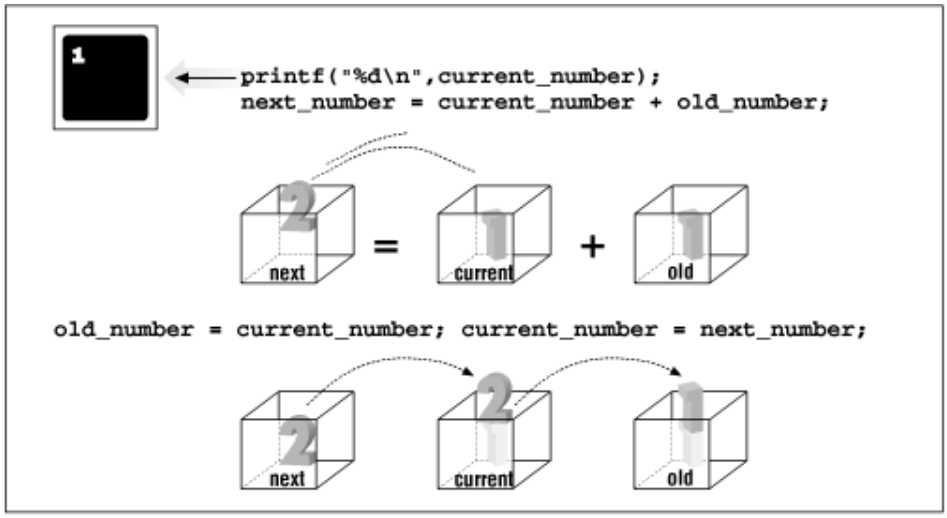
We want to loop until our current term is 100 or larger. The **while** loop:

```
while (current_number < 100)
```

will repeat our computation and printing until we reach this limit.

Figure 6 -1 shows what happens to the variable during the execution of the program. At the beginning, `current_number` and `old_number` are 1. We print the value of the current term. Then the variable `next_number` is computed (value 2). Next we advance one term by putting `next_number` into `current_number` and `current_number` into `old_number`. This process is repeated until we compute the last term and the **while** loop exits.

## Figure 6-1. Fibonacci execution




This completes the body of the loop. The first two terms of the Fibonacci sequence are 1 and 1. We initialize our first two terms to these values. Putting it all together, we get the code in Example 6-1.

## Example 6-1. *fib/fib.c*

```
#include <stdio.h>
int   old_number;     /* previous Fibonacci number */
int   current_number; /* current Fibonacci number */
int   next_number;    /* next number in the series */

int main()
{
    /* start things out */
    old_number = 1;
    current_number = 1;

    printf("1\n");    /* Print first number */

    while (current_number < 100) {

        printf("%d\n", current_number);
        next_number = current_number + old_number;

        old_number = current_number;
        current_number = next_number;
```

```
    }
    return (0);
}
```

## 6.6 break Statement

We have used a **while** statement to compute the Fibonacci numbers less than 100. The loop exits when the condition after the **while** becomes false (0). Loops can be exited at any point through the use of a **break** statement.

Suppose we want to add a series of numbers, but we don't know how many numbers are to be added together. We need some way of letting the program know that we have reached the end of our list. In Example 6-2, we use the number zero (0) to signal the end-of-list.

Note that the **while** statement begins with:

```
while (1) {
```

Left to its own devices, the program will loop forever because the **while** will exit only when the expression 1 is 0. The only way to exit this loop is through a **break** statement.

When we see the end of the list indicator (0), we use the statement:

```
if (item == 0)
     break;
```

to exit the loop.

## Example 6-2. *total/total.c*

```
#include <stdio.h>
char  line[100];/* line of data for input */
int   total;    /* Running total of all numbers so far */
int   item;     /* next item to add to the list */

int main()
{
    total = 0;
    while (1) {
        printf("Enter # to add \n");
        printf("  or 0 to stop:");
```

```
        fgets(line, sizeof(line), stdin);
        sscanf(line, "%d", &item);

        if (item == 0)
            break;

        total += item;
        printf("Total: %d\n", total);
    }
    printf("Final total %d\n", total);
    return (0);
}
```

## 6.7 continue Statement

The **continue** statement is very similar to the **break** statement, except that instead of terminating the loop, **continue** starts reexecuting the body of the loop from the top. For example, if we want to modify the previous program to total only numbers larger than 0, we could write a program such as Example 6-3.

### Example 6-3. *totalb/totalb.c*

```
[File: totalb/totalb.c]
#include <stdio.h>
char  line[100];   /* line from input */
int   total;       /* Running total of all numbers so far */
int   item;        /* next item to add to the list */
int   minus_items; /* number of negative items */

int main()
{
    total = 0;
    minus_items = 0;

    while (1) {
        printf("Enter # to add\n");
        printf("  or 0 to stop:");

        fgets(line, sizeof(line), stdin);
        sscanf(line, "%d", &item);

        if (item == 0)
            break;
```

```
        if (item < 0) {
           ++minus_items;
           continue;
        }
        total += item;
        printf("Total: %d\n", total);
    }

    printf("Final total %d\n", total);
    printf("with %d negative items omitted\n",
                minus_items);
    return (0);
}
```

## 6.8 Assignment Anywhere Side Effect

C allows the use of assignment statements almost anywhere. For example, you can put assignment statements inside assignment statements:

```
/* don't program like this */
average = total_value / (number_of_entries = last - first);
```

This is the equivalent of saying:

```
/* program like this */
number_of_entries = last - first;
average = total_value / number_of_entries;
```

The first version buries the assignment of `number_of_entries` inside the expression. Programs should be clear and simple and should not hide anything. The most important rule of programming is *keep it simple*.

C also allows the programmer to put assignment statements in the **while** conditional. For example:

```
/* do not program like this */
while ((current_number = last_number + old_number) < 100)
    printf("Term %d\n", current_number);
```

Avoid this type of programming. Notice how much clearer the logic is in the version below:

```
/* program like this */
```

```
while (1) {
    current_number = last_number + old_number;
    if (current_number >= 100)
       break;
    printf("Term %d\n", current_number);
}
```

**Question 6 -1**: *For some strange reason, Example 6 -4 thinks that everyone owes a balance of dollars. Why?* (Click here for the answer Section 6.9 )

## Example 6-4. *owe0/owe0.c*

```
#include <stdio.h>
char  line[80];        /* input line */
int   balance_owed;     /* amount owed */

int main()
{
    printf("Enter number of dollars owed:");
    fgets(line, sizeof(line), stdin);
    sscanf(line, "%d", &balance_owed);

    if (balance_owed = 0)
       printf("You owe nothing.\n");
    else
       printf("You owe %d dollars.\n", balance_owed);

    return (0);
}
```

Sample output:

```
Enter number of dollars owed: 12
You owe 0 dollars.
```

## 6.9 Answer

**Answer 6 -1** : This program illustrates one of the most common and frustrating of C errors. The problem is that C allows assignment statements inside **if** conditionals. The statement:

```
if (balance_owed = 0)
```

uses a single equal sign (=) instead of the double equal sign (==). C will assign balance_owed the value and test the result (which is 0). If the result was nonzero (true), the **if** clause would be executed. Because the result is (false), the **else** clause is executed and the program prints the wrong answer.

The statement:

```
if (balance_owed = 0)
```

is equivalent to:

```
balance_owed = 0;
if (balanced_owed != 0)
```

The statement should be written:

```
if (balance_owed == 0)
```

This error is the most common error that beginning C programmers make.

## 6.10 Programming Exercises

**Exercise 6-1**: Write a program to find the square of the distance between two points. (For a more advanced problem, find the actual distance. This problem involves using the standard function sqrt. Use your help system to find out more about how to use this function.)

**Exercise 6-2**: A professor generates letter grades using Table 6-3.

**Table 6-3. Grade Values**

| % Right | Grade |
|---|---|
| 0-60 | F |
| 61-70 | D |
| 71-80 | C |
| 81-90 | B |
| 91-100 | A |

Given a numeric grade, print the letter.

Programmers frequently have to modify code that someone else wrote. A good exercise is to take someone else's code, such as the program that someone wrote for Chapter 6, and then modify it.

**Exercise 6-3**: Modify the previous program to print a + or - after the letter grade, based on the last digit of the score. The modifiers are listed in Table 6-4.

**Table 6-4. Grade Modification Values**

| Last digit | Modifier |
|---|---|
| 1-3 | - |
| 4-7 | <blank> |
| 8-0 | + |

For example, 81=B-, 94=A, and 68=D+. Note: An F is only an F. There is no F+ or F-.

**Exercise 6-4**: Given an amount of money (less than $1.00), compute the number of quarters, dimes, nickels, and pennies needed.

**Exercise 6-5**: A leap year is any year divisible by 4, unless the year is divisible by 100, but not 400. Write a program to tell if a year is a leap year.

**Exercise 6-6**: Write a program that, given the number of hours an employee worked and the hourly wage, computes the employee's weekly pay. Count any hours over 40 as overtime at time and a half.