C Programming

Updated for C11

ABSOLUTE BEGINNER'S GUIDE

No experience necessary!

Third Edition

que

Greg Perry and Dean Miller

C Programming

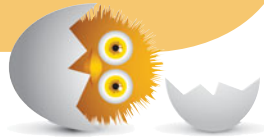
Greg Perry and Dean Miller

que®
800 East 96th Street
Indianapolis, Indiana 46240

IN THIS CHAPTER

- Testing data
- Using `if`
- Using `else`

11



THE FORK IN THE ROAD—TESTING DATA TO PICK A PATH

C provides an extremely useful statement called `if`. `if` lets your programs make decisions and execute certain statements based on the results of those decisions. By testing contents of variables, your programs can produce different output, given different input.

This chapter also describes *relational operators*. Combined with `if`, relational operators make C a powerful data-processing language. Computers would really be boring if they couldn't test data; they would be little more than calculators if they had no capability to decide courses of action based on data.

Testing Data

The C `if` statement works just like it does in spoken language: *If something is true, do one thing; otherwise, do something else.* Consider these statements:

If I make enough money, we'll go to Italy.

If the shoes don't fit, take them back.

If it's hot outside, water the lawn.

Table 11.1 lists the C relational operators, which permit testing of data. Notice that some of the relational operators consist of two symbols.

TABLE 11.1 C Relational Operators

Relational Operator	Description
`==`	Equal to
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal to
`<=`	Less than or equal to
`!=`	Not equal to

NOTE Relational operators compare two values. You always put a variable, literal, or expression—or a combination of any two of them—on either side of a relational operator.

Before delving into `if`, let's look at a few relational operators and see what they really mean. A regular operator produces a mathematical result. A relational operator produces a *true* or *false* result. When you compare two data values, the data values either produce a true comparison or they don't. For example, given the following values:

```
int i = 5;
int j = 10;
int k = 15;
int l = 5;
```

the following statements are *true:*

```
i == l;
j < k;
k > i;
j != l;
```

The following statements are *not* true, so they are *false:*

```
i > j;
k < j;
k == l
```

TIP To tell the difference between = and ==, remember that you need two equals signs to double-check whether something is equal.

WARNING Only like values should go on either side of the relational operator. In other words, don't compare a character to a float. If you have to compare two unlike data values, use a typecast to keep the values the same data type.

Every time C evaluates a relational operator, a value of `1` or `0` is produced. True always results in `1`, and false always results in `0`. The following statements assign a `1` to the variable `a` and a `0` to the variable `b`:

```
a = (4 < 10);  // (4 < 10) is true, so a 1 is put in a
b = (8 == 9);  // (8 == 9) is false, so a 0 is put in b
```

You will often use relational operators in your programs because you'll often want to know whether sales (stored in a variable) is more than a set goal, whether payroll calculations are in line, and whether a product is in inventory or needs to be ordered, for example. You have seen only the beginning of relational operators. The next section explains how to use them.

Using `if`

The `if` statement uses relational operators to perform data testing. Here's the format of the `if` statement:

```
if (condition)
{ block of one or more C statements; }
```

The parentheses around the *condition* are required. The *condition* is a relational test like those described in the preceding section. The *block of one or more C statements* is called the *body* of the `if` statement. The braces around the *block of one or more C statements* are required if the body of the `if` contains more than a single statement.

TIP Even though braces aren't required, if an `if` contains just one statement, always use the braces. If you later add statements to the body of the `if`, the braces will be there. If the braces enclose more than one statement, the braces enclose what is known as a *compound statement*.

Here is a program with two `if` statements:

```
// Example program #1 from Chapter 11 of Absolute Beginner's Guide
// to C, 3rd Edition
// File Chapter11ex1.c

/* This program asks the user for their birth year and calculates
how old they will be in the current year. (it also checks to make
sure a future year has not been entered.) It then tells the user if
they were born in a leap year. */

#include <stdio.h>
#define CURRENTYEAR 2013

main()
{

    int yearBorn, age;

    printf("What year were you born?\n");
    scanf(" %d", &yearBorn);

    // This if statement can do some data validation, making sure
    // the year makes sense
    // The statements will only execute if the year is after the
    // current year
```

```
    if (yearBorn > CURRENTYEAR)
    {
        printf("Really? You haven't been born yet?\n");
        printf("Want to try again with a different year?\n");
        printf("What year were you born?\n");
        scanf(" %d", &yearBorn);
    }

    age = CURRENTYEAR - yearBorn;

    printf("\nSo this year you will turn %d on your birthday!\n",
    age);

    // The second if statment uses the modulus operator to test if
    // the year of birth was a Leap Year. Again, only if it is will
    // the code execute

    if ((yearBorn % 4) == 0)
    {
        printf("\nYou were born in a Leap Year--cool!\n");
    }

    return 0;
}
```

Consider a few notes about this program. If you use the current year in your program, that's a good variable to set with a `#define` statement before `main()`. That way, you can simply change that one line later if you run this program any year in the future.

The first `if` statement is an example of how to potentially use `if` as a form of data validation. The statement tests whether the user has entered a year later than the current year and, if so, executes the section of code that follows in the braces. If the user has entered a proper year, the program skips down to the line that calculates the user's age. The section in the braces reminds the reader that he or she couldn't possibly be born in the year entered and gives the user a chance to enter a new year. The program then proceeds as normal.

Here you might have noticed a limitation to this plan. If the user enters an incorrect year a second time, the program proceeds and even tells the age in negative years! A second style of conditional statements, a `do...while` loop, keeps hounding the user until he or she enters correct data. This is covered in Chapter 14, "Code Repeat—Using Loops to Save Time and Effort."

TIP You can change the relational operator to not accept the data entry if the user types in a year greater than or equal to the current year, but maybe the user is helping a recent newborn!

After calculating what the user's age will be on his or her birthday this year, a second `if` statement tests the year of the user's birth to see whether he or she was born in a leap year by using the modulus operator. Only leap years are divisible by 4 without a remainder, so only people who were born in one of those years will see the message noting the one-in-four odds of their birth year. For the rest, that section of code is skipped and the program reaches its termination point.

NOTE The `main()` function in the Draw Poker program in Appendix B, "The Draw Poker Program," asks the player which cards to keep and which cards to replace. An `if` is used to determine exactly what the user wants to do.

Otherwise...: Using `else`

In the preceding section, you saw how to write a course of action that executes if the relational test is true. If the relational test is false, nothing happens. This section explains the `else` statement that you can add to `if`. Using `else`, you can specify exactly what happens when the relational test is false. Here is the format of the combined `if...else`:

```
if (condition)
{ block of one or more C statements; }
else
{ block of one or more C statements; }
```

So in the case of `if...else`, one of the two segments of code will run, depending on whether the condition tested is true (in which case, the `if` code will run) or false (in which case, the `else` code will run). This is perfect if you have two possible outcomes and need to run different code for each.

Here is an example of `if...else` that moves the previous program to an `if...else` construction. In this version, the user does not have the opportunity to re-enter a year, but it does congratulate the user on coming back from the future.