CHAPTER 3

Making Decisions

In Chapter 2 you learned how to do calculations in your programs. In this chapter, you'll take great leaps forward in the range of programs you can write and the flexibility you can build into them. You'll add one of the most powerful programming tools to your inventory: the ability to compare the values of expressions and, based on the outcome, choose to execute one set of statements or another.

What this means is that you'll be able to control the sequence in which statements are executed in a program. Up until now, all the statements in your programs have been executed strictly in sequence. In this chapter you're going to change all that.

You are going to learn the following:

- How to make decisions based on arithmetic comparisons
- What logical operators are and how you can use them
- More about reading data from the keyboard
- How you can write a program that can be used as a calculator

The Decision-Making Process

You'll start with the essentials of in a program. Decision making in a program is concerned with choosing to execute one set of program statements rather than another. In everyday life you do this kind of thing all the time. Each time you wake up you have to decide whether it's a good idea to go to work. You may go through these questions:

> Do I feel well?
>
> If the answer is no, stay in bed. If the answer is yes, go to work.

You could rewrite this as follows:

> If I feel well, I will go to work. Otherwise, I will stay in bed.

That was a straightforward decision. Later, as you're having breakfast, you notice it's raining, so you think:

> If it is raining as hard as it did yesterday, I will take the bus. If it is raining harder than yesterday, I will drive to work. Otherwise, I will risk it and walk.
>
> This is a more complex decision process. It's a decision based on several levels in the amount of rain falling, and it can have any of three different results.

As the day goes on, you're presented with more of these decisions. Without them you'd be stuck with only one course of action. Until now, in this book, you've had exactly the same problem with

your programs. All the programs will run a straight course to a defined end, without making any decisions. This is a severe constraint on what your programs can do and one that you'll relieve now. First, you'll set up some basic building blocks of knowledge that will enable you to do this.

Arithmetic Comparisons

To make a decision, you need a mechanism for comparing things. This involves some new operators. Because you're dealing with numbers, comparing numerical values is basic to decision making. You have three fundamental **relational operators** that you use to compare values:

- < is less than
- == is equal to
- > is greater than

■ **Note** The `equal to` operator has *two* successive equal signs (==). You'll almost certainly use one equal sign on occasions by mistake. This will cause considerable confusion until you spot the problem. Look at the difference. If you type `my_weight = your_weight`, it's an assignment that puts the value from the variable `your_weight` into the variable `my_weight`. If you type the expression `my_weight == your_weight`, you're comparing the two values: you're asking whether they're exactly the same—you're not making them the same. If you use = where you intended to use == the compiler cannot determine that it is an error because either is usually valid.

Expressions Involving Relational Operators

Have a look at these examples:

```
5 < 4      1 == 2      5 > 4
```

These expressions are called **logical expressions** or **Boolean expressions** because each of them can result in just one of two values: either `true` or `false`. As you saw in the previous chapter, the value `true` is represented by 1; `false` is represented by 0. The first expression is false because 5 is patently not less than 4. The second expression is also false because 1 is not equal to 2. The third expression is true because 5 is greater than 4.

Because a relational operator produces a Boolean result, you can store the result in a variable of type `_Bool`. For example

```
_Bool result = 5 < 4;        /* result will be false */
```

If you `#include` the `<stdbool.h>` header file in the source file, you can use `bool` instead of the keyword `_Bool`, so you could write the statement like this:

```
bool result = 5 < 4;         /* result will be false */
```

Keep in mind that any nonzero numerical value will result in `true` when it is converted to type `_Bool`. This implies that you can assign the result of an arithmetic expression to a `_Bool` variable and store `true` if it is nonzero and `false` otherwise.

The Basic if Statement

Now that you have the relational operators for making comparisons, you need a statement allowing you to make a decision. The simplest is the `if` statement. If you want to compare your weight with that of someone else and print a different sentence depending on the result, you could write the body of a program as follows:

```
if(your_weight > my_weight)
  printf("You are heavier than me.\n");

if(your_weight < my_weight)
  printf("I am heavier than you.\n");

if(your_weight == my_weight)
  printf("We are exactly the same weight.\n");
```

Note how the statement following each `if` is indented. This is to show that it's dependent on the result of the `if` test. Let's go through this and see how it works. The first `if` tests whether the value in `your_weight` is greater than the value in `my_weight`. The expression for the comparison appears between the parentheses that immediately follow the keyword `if`. If the result of the comparison is true, the statement immediately after the `if` will be executed. This just outputs the following message:

```
You are heavier than me.
```

Execution will then continue with the next `if`.

What if the expression between the parentheses in the first `if` is false? In this case, the statement immediately following the `if` will be skipped, so the message won't be displayed. It will be displayed only if `your_weight` is greater than `my_weight`.

The second `if` works in essentially the same way. If the expression between parentheses after the keyword `if` is true, the following statement will be executed to output this message:

```
I am heavier than you.
```

This will be the case if `your_weight` is less than `my_weight`. If this isn't so, the statement will be skipped and the message won't be displayed. The third `if` is again the same. The effect of these statements is to print one message that will depend on whether `your_weight` is greater than, less than, or equal to `my_weight`. Only one message will be displayed because only one of these can be true.

The general form or syntax of the `if` statement is as follows:

```
if(expression)
  Statement1;

Next_statement;
```

Notice that the expression that forms the test (the `if`) is enclosed between parentheses and that there is no semicolon at the end of the first line. This is because both the line with the `if` keyword and the following line are tied together. The second line could be written directly following the first, like this:

```
if(expression) Statement1;
```

But for the sake of clarity, people usually put `Statement1` on a new line.

The `expression` in parentheses can be any expression that results in a value of `true` or `false`. If the expression is true, `Statement1` is executed, after which the program continues with `Next_statement`. If the expression is false, `Statement1` is skipped and execution continues immediately with `Next_statement`. This is illustrated in Figure 3-1.

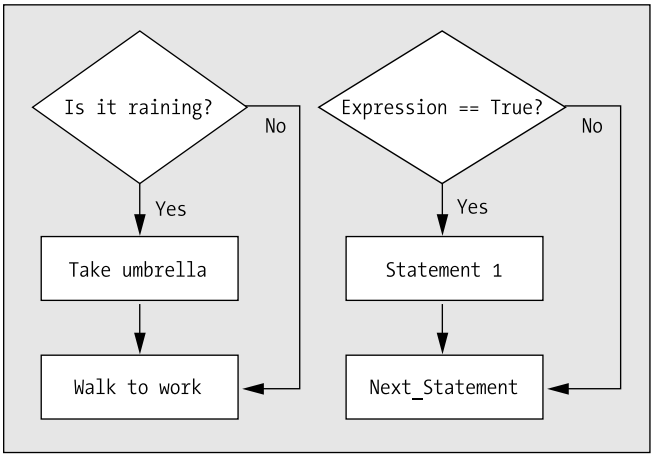

Figure 3-1. *The operation of the* `if` *statement*

You could have used the basic `if` statement to add some politically incorrect comments in the program that calculated the height of a tree at the end of the previous chapter. For example, you could have added the following code just after you'd calculated the height of the shortest person:

```
if(Shorty < 36)
  printf("\nMy, you really are on the short side, aren't you?");
```

Here, you have used the `if` statement to add a gratuitously offensive remark, should the individual be less than 36 inches tall.

Don't forget what I said earlier about what happens when a numerical value is converted to type `_Bool`. Because the control expression for an `if` statement is expected to produce a Boolean result, the compiler will arrange to convert the result of an `if` expression that produces a numerical result to type `_Bool`. You'll sometimes see this used in programs to test for a nonzero result of a calculation. Here's a statement that illustrates this:

```
if(count)
  printf("The value of count is not zero.");
```

This will only produce output if `count` is not 0, because a 0 value for `count` will mean the `if` expression is `false`.

TRY IT OUT: CHECKING CONDITIONS

Let's see the `if` statement in action. This program gets the user to enter a number between 1 and 10 and then tells the user how big that number is:

```
/* Program 3.1 A simple example of the if statement */
#include <stdio.h>

int main(void)
{
  int number = 0;
  printf("\nEnter an integer between 1 and 10: ");
  scanf("%d",&number);
```

```
  if(number > 5)
    printf("You entered %d which is greater than 5\n", number);

  if(number < 6)
    printf("You entered %d which is less than 6\n", number);
  return 0;
}
```

Sample output from this program is as follows:

```
Enter an integer between 1 and 10: 7
You entered 7 which is greater than 5
```

or

```
Enter an integer between 1 and 10: 3
You entered 3 which is less than 6
```

How It Works

As usual, you include a comment at the beginning as a reminder of what the program does. You include the `stdio.h` header file to allow you to use the `printf()` statement. You then have the beginning of the `main()` function of the program. This function doesn't return a value, as indicated by the keyword `void`:

```
/* Program 3.1 A simple example of the if statement*/
#include <stdio.h>

int main(void)
{
```

In the first three statements in the body of `main()`, you read an integer from the keyboard after prompting the user for the data:

```
  int number = 0;
  printf("\nEnter an integer between 1 and 10: \n");
  scanf("%d",&number);
```

You declare an integer variable called `number` that you initialize to 0, and then you prompt the user to enter a number between 1 and 10. This value is then read using the `scanf()` function and stored in the variable `number`.

The next statement is an `if` that tests the value that was entered:

```
  if(number > 5)
   printf("You entered %d which is greater than 5", number);
```

You compare the value in `number` with the value 5. If `number` is greater than 5, you execute the next statement, which displays a message, and you go to the next part of the program. If `number` isn't greater than 5, `printf()` is simply skipped. You've used the `%d` conversion specifier for integer values to output the number the user typed in.

You then have another `if` statement:

```
  if(number < 6)
   printf("You entered %d which is less than 6", number);
```

This compares the value entered with 6 and, if it's smaller, you execute the next statement to display a message. Otherwise, the `printf()` is skipped and the program ends. Only one of the two possible messages will be displayed because the number will always be less than 6 or greater than 5.

The `if` statement enables you to be selective about what input you accept and what you finally do with it. For instance, if you have a variable and you want to have its value specifically limited at some point, even though higher values may arise somehow in the program, you could write this:

```
if(x > 90)
  x = 90;
```

This would ensure that if anyone entered a value of `x` that was larger than 90, your program would automatically change it to 90. This would be invaluable if you had a program that could only specifically deal with values within a range. You could also check whether a value was lower than a given number and, if not, set it to that number. In this way, you could ensure that the value was within the given range.

Finally you have the return statement that ends the program and returns control to the operating system:

```
  return 0;
```

Extending the if Statement: if-else

You can extend the `if` statement with a small addition that gives you a lot more flexibility. Imagine it rained a little yesterday. You could write the following:

If the rain today is worse than the rain yesterday,

I will take my umbrella.

Else

I will take my jacket.

Then I will go to work.

This is exactly the kind of decision-making the `if-else` statement provides. The syntax of the `if-else` statement is as follows:

```
if(expression)
  Statement1;
else
  Statement2;

Next_statement;
```

Here, you have an either-or situation. You'll always execute either `Statement1` or `Statement2` depending on whether `expression` results in the value `true` or `false`:

If `expression` evaluates to `true`, `Statement1` is executed and the program continues with `Next_statement`.

If `expression` evaluates to `false`, `Statement2` following the `else` keyword is executed, and the program continues with `Next_statement`.

The sequence of operations involved here is shown in Figure 3-2.

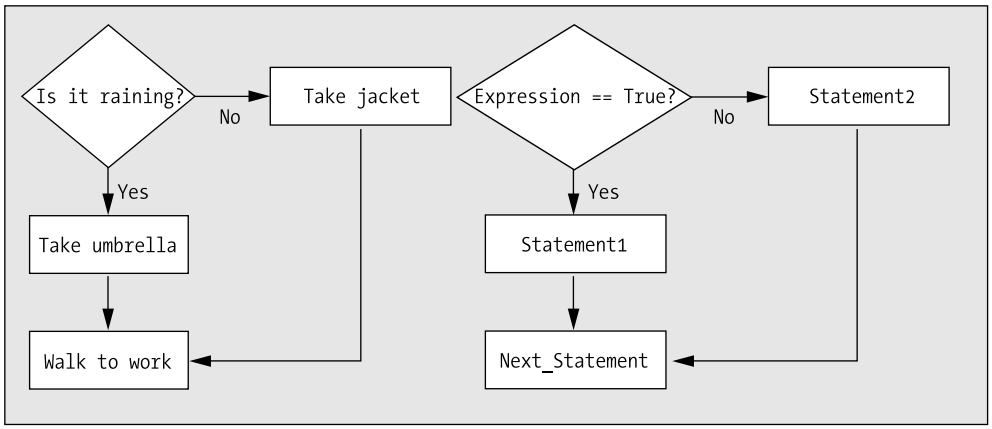

Figure 3-2. *The operation of the* `if-else` *statement*

TRY IT OUT: USING IF TO ANALYZE NUMBERS

Let's suppose that you're selling a product at a single-unit price of $3.50, and for order quantities greater than ten you offer a 5 percent discount. You can use the `if-else` statement to calculate and output the price for a given quantity.

```
/* Program 3.2 Using if statements to decide on a discount */
#include <stdio.h>

int main(void)
{
  const double unit_price = 3.50;           /* Unit price in dollars */
  int quantity = 0;
  printf("Enter the number that you want to buy:"); /* Prompt message */
  scanf(" %d", &quantity);                  /* Read the input        */

  /* Test for order quantity qualifying for a discount */
  if(quantity>10)                           /* 5% discount           */
    printf("The price for %d is $%.2f\n", quantity, quantity*unit_price*0.95);
  else                                      /* No discount           */
    printf("The price for %d is $%.2f\n", quantity, quantity*unit_price);
  return 0;
}
```

Typical output from this program is as follows:

```
Enter the number that you want to buy:20
The price for 20 is $66.50
```

How It Works

Once your program has read the order quantity, the `if-else` statement does all the work:

```
if(quantity>10)                          /* 5% discount          */
  printf("\nThe price for %d is $%.2f\n", quantity, quantity*unit_price*0.95);
else                                     /* No discount          */
  printf("\nThe price for %d is $%.2f\n", quantity, quantity*unit_price);
```

If `quantity` is greater than ten, the first `printf()` will be executed that applies a 5 percent discount. Otherwise, the second `printf()` will be executed that applies no discount to the price.

There are a few more things I could say on this topic, though. First of all, you can also solve the problem with a simple `if` statement by replacing the `if-else` statement with the following code:

```
double discount = 0.0;                   /* Discount allowed */
if(quantity>10)
   discount = 0.05;                      /* 5% discount      */
printf("\nThe price for %d is $%.2f\n", quantity,
                                    quantity*unit_price*(1.0-discount));
```

This considerably simplifies the code. You now have a single `printf()` call that applies the discount that is set, either 0 or 5 percent. With a variable storing the discount value, it's also clearer what is happening in the code.

The second point worth making is that floating-point variables aren't ideal for calculations involving money because of the potential rounding that can occur. Providing that the amounts of money are not extremely large, one alternative is to use integer values and just store cents, for example

```
const long unit_price = 350L;            /* Unit price in cents */
int quantity = 0;
printf("Enter the number that you want to buy:");  /* Prompt message */
scanf(" %d", &quantity);                /* Read the input      */

long discount = 0L;                      /* Discount allowed    */
if(quantity>10)
  discount = 5L;                         /* 5% discount         */
long total_price = quantity*unit_price*(100-discount)/100;
long dollars = total_price/100;
long cents = total_price%100;
printf("\nThe price for %d is $%ld.%ld\n", quantity, dollars,cents);
```

Of course, you also have the possibility of storing the dollars and cents for each monetary value in separate integer variables. It gets a little more complicated because you then have to keep track of when the cents value reaches or exceeds 100 during arithmetic operations.

Using Blocks of Code in if Statements

You can also replace either `Statement1` or `Statement2`, or even both, by a block of statements enclosed between braces {}. This means that you can supply many instructions to the computer after testing the value of an expression using an `if` statement simply by placing these instructions together between braces. I can illustrate the mechanics of this by considering a real-life situation:

If the weather is sunny,

I will walk to the park, eat a picnic, and walk home.

Else

I will stay in, watch football, and drink beer.

The syntax for an `if` statement that involves statement blocks is as follows:

```
if(expression)
{
  StatementA1;
  StatementA2;
  ...
}
else
{
  StatementB1;
  StatementB2;
  ...
}

Next_statement;
```

All the statements that are in the block between the braces following the `if` condition will be executed if `expression` evaluates to `true`. If `expression` evaluates to `false`, all the statements between the braces following the `else` will be executed. In either case, execution continues with `Next_statement`. Have a look at the indentation. The braces aren't indented, but the statements between the braces are. This makes it clear that all the statements between an opening and a closing brace belong together.

■ **Note** Although I've been talking about using a block of statements in place of a single statement in an `if` statement, this is just one example of a general rule. Wherever you can have a single statement, you can equally well have a block of statements between braces. This also means that you can nest one block of statements inside another.

Nested if Statements

It's also possible to have `if`s within `if`s. These are called **nested** `if`**s**. For example

If the weather is good,

I will go out in the yard.

And if it's cool enough,

I will sit in the sun.

Else

I will sit in the shade.

Else

I will stay indoors.

I will then drink some lemonade.

In programming terms, this corresponds to the following:

```
if(expression1)              /* Weather is good?            */
{
  StatementA;                /* Yes - Go out in the yard    */
  if(expression2)            /* Cool enough?                */
    StatementB;              /* Yes - Sit in the sun        */
  else
    StatementC;              /* No - Sit in the shade       */
}
else
  StatementD;                /* Weather not good - stay in  */
Statement E;                 /* Drink lemonade in any event */
```

Here, the second `if` condition, `expression2`, is only checked if the first `if` condition, `expression1`, is true. The braces enclosing `StatementA` and the second `if` are necessary to make both of these statements a part of what is executed when `expression1` is true. Note how the `else` is aligned with the `if` it belongs to. The logic of this is illustrated in Figure 3-3.

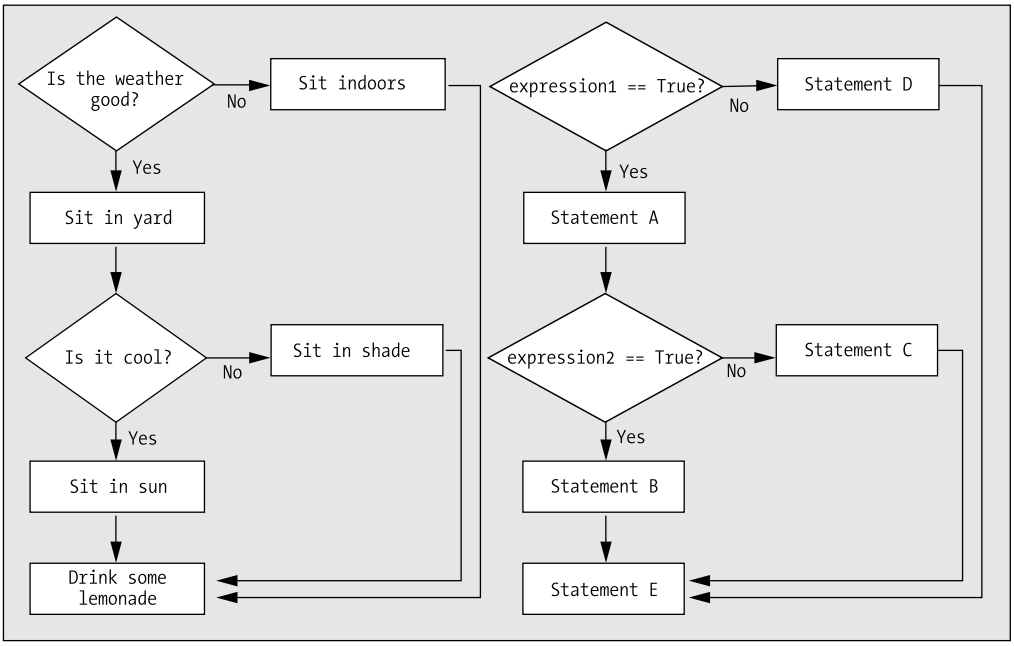

Figure 3-3. *Nested* `if` *statements*

TRY IT OUT: ANALYZING NUMBERS

You'll now exercise your `if` skills with a couple more examples. This program tests to see whether you enter an odd or an even number, and if the number is even, it then tests to see whether half that number is also even:

```
/* Program 3.3 Using nested ifs to analyze numbers */
#include <stdio.h>
#include <limits.h>            /* For LONG_MAX */
```

```
int main(void)
{
  long test = 0L;              /* Stores the integer to be checked */

  printf("Enter an integer less than %ld:", LONG_MAX);
  scanf(" %ld", &test);

  /* Test for odd or even by checking the remainder after dividing by 2 */
  if(test % 2L == 0L)
  {
    printf("The number %ld is even", test);

    /* Now check whether half the number is also even */
    if((test/2L) % 2L == 0L)
    {
      printf("\nHalf of %ld is also even", test);
      printf("\nThat's interesting isn't it?\n");
    }
  }
  else
    printf("The number %ld is odd\n", test);
  return 0;
}
```

The output will look something like this:

```
Enter an integer less than 2147483647:20
The number 20 is even
Half of 20 is also even
That's interesting isn't it?
```

or this

```
Enter an integer less than 2147483647:999
The number 999 is odd
```

How It Works

The prompt for input makes use of the `LONG_MAX` symbol that's defined in the `<limits.h>` header file. This specifies the maximum value of type `long`. You can see from the output that on my system the upper limit for `long` values is 2147483647.

The first `if` condition tests for an even number:

```
if(test % 2L == 0L)
```

If you were to use `0` instead of `0L` here, your compiler may insert code to convert 0, which is of type `int`, to a value of type `long` to allow the comparison for equality to be made. Using the constant `0L` of type `long` avoids this unnecessary operation. For any even number, the remainder after dividing by 2 will be 0. If the expression is `true`, the block that follows will be executed:

```
{
  printf("The number %ld is even", test);

  /* Now check whether half the number is also even */
  if((test/2L) % 2L == 0L)
  {
    printf("\nHalf of %ld is also even", test);
    printf("\nThat's interesting isn't it?\n");
  }
}
```

After outputting a message where the value is even, you have another `if` statement. This is called a nested `if` because it's inside the first `if`. The nested `if` condition divides the original value by 2 and tests whether the result is even, using the same mechanism as in the first `if` statement. There's an extra pair of parentheses in the nested `if` condition around the expression `test/2L`. These aren't strictly necessary, but they help to make what's going on clear. Making programs easier to follow is the essence of good programming style. If the result of the nested `if` condition is true, the two further `printf()` statements in the block following the nested `if` will be executed.

Try adding code to make the nested `if` an `if-else` that will output `"Half of %ld is odd"`.

If the original input value isn't even, the statement following the `else` keyword will be executed:

```
else
  printf("The number %ld is odd\n", test);
```

Note You can nest `if`s anywhere inside another `if`, but I don't recommend this as a technique that you should use extensively. If you do, your program is likely to end up being very hard to follow and you are more likely to make mistakes.

To make the nested `if` statement output a message when the condition is `false`, you would need to insert the following after the closing brace:

```
else
  printf("\nHalf of %ld is odd", test);
```

More Relational Operators

You can now add a few more relational operators that you can use to compare expressions in `if` statements. These three additional operators make up the complete set:

- `>=` is greater than or equal to
- `<=` is less than or equal to
- `!=` is not equal to

These are fairly self-explanatory, but let's consider some examples anyway, starting with a few arithmetic examples:

```
6 >= 5      5 <= 5      4 <= 5      4 != 5      10 != 10
```

These all result in the value `true`, except for the last one, which is `false` because 10 most definitely *is* equal to 10. These operators can be applied to values of type `char` and `wchar_t` as well as the other numerical types. If you remember, character types also have a numeric value associated with them. The ASCII table in Appendix B provides a full listing of all the standard ASCII characters and their numeric codes. Table 3-1 is an extract from Appendix B as a reminder for the next few examples.

Table 3-1. *Characters and ASCII Codes*

Character	ASCII Code (Decimal)
A	65
B	66
P	80
Q	81
Z	90
b	98

A `char` value may be expressed either as an integer or as a keyboard character between quotes, such as `'A'`. Don't forget, numeric values stored as type `char` may be signed or unsigned, depending on how your compiler implements the type. When type `char` is unsigned, values can be from 128 to +127. When `char` is an unsigned type, values can be from 0 to 255. Here are a few examples of comparing values of type `char`:

```
'Z' >= 'A'      'Q' <= 'P'      'B' <= 'b'      'B' != 66
```

With the ASCII values of the characters in mind, the first expression is true, because `'Z'`, which has the code value 90, comes after `'A'`, which has the code value 65. The second is false, as `'Q'` doesn't come before `'P'`. The third is true. This is because in ASCII code lowercase letters are 32 higher than their uppercase equivalents. The last is false. The value 66 is indeed the decimal ASCII representation for the character `'B'`.

TRY IT OUT: CONVERTING UPPERCASE TO LOWERCASE

Let's exercise the new logical operators in an example. Here you have a program that will convert any uppercase letter that is entered to a lowercase letter:

```
/* Program 3.4 Converting uppercase to lowercase */
#include <stdio.h>

int main(void)
{
  char letter = 0;                      /* Stores a character              */

  printf("Enter an uppercase letter:"); /* Prompt for input                */
  scanf("%c", &letter);                 /* Read a character                */
```

```
  /* Check whether the input is uppercase */
  if(letter >= 'A')                    /* Is it A or greater?             */
    if(letter <= 'Z')                  /* and is it Z or lower?           */
    {                                  /* It is uppercase                 */
      letter = letter - 'A'+ 'a';      /* Convert from upper- to lowercase */
      printf("You entered an uppercase %c\n", letter);
    }
    else                               /* It is not an uppercase letter   */
      printf("Try using the shift key, Bud! I want a capital letter.\n");
  return 0;
}
```

Sample output from this program might be the following:

```
Enter an uppercase letter:G
You entered an uppercase g
```

or

```
Enter an uppercase letter:s
Try using the shift key, Bud! I want a capital letter.
```

How It Works

In the first three statements, you declare a variable of type `char` called `letter`, you prompt the user to input a capital letter, and you store the character entered in the variable `letter`:

```
  char letter = 0;                     /* Stores a character              */

  printf("Enter an uppercase letter:");/* Prompt for input                */
  scanf("%c", &letter);                /* Read a character                */
```

If a capital letter is entered, the character in the `letter` variable must be between `'A'` and `'Z'`, so the next `if` checks whether the character is greater than or equal to `'A'`:

```
  if(letter >= 'A')                      /* Is it A or greater?           */
```

If the expression is true, you continue with the nested `if` that tests whether `letter` is less than or equal to `'Z'`:

```
    if(letter <= 'Z')                    /* and is it Z or lower?         */
```

If this expression is true, you convert the character to lowercase and output a message by executing the block of statements following the `if`:

```
    {                                    /* It is uppercase               */
      letter = letter - 'A'+ 'a';        /* Convert from upper- to lowercase */
      printf("You entered an uppercase %c\n", letter);
    }
```

To convert to lowercase, you subtract the character code for `'A'` from `letter` and add the character code for `'a'`. If `letter` contained `'A'`, subtracting `'A'` would produce 0, and adding `'a'` would result in `'a'`. If `letter` contained `'B'`, subtracting `'A'` would produce 1, and adding `'a'` would result in `'b'`. You can see this conversion

works for any uppercase letter. Note that although this works fine for ASCII, there are coding systems (such as EBCDIC) in which this won't work, because the letters don't have a contiguous sequence of codes. If you want to be sure that the conversion works for any code, you can use the standard library function `tolower()`. This converts the character passed as an argument to lowercase if it's an uppercase letter; otherwise, it returns the character code value unchanged. To use this function, you need to include the header file `ctype.h` in your program. This header file also declares the complementary function, `toupper()`, that will convert lowercase letters to uppercase.

If the expression `letter <= 'Z'` is false, you go straight to the statement following `else` and display a different message:

```
    else                                /* It is not an uppercase letter    */
      printf("Try using the shift key, Bud! I want a capital letter.\n");
```

There's something wrong, though. What if the character that was entered was less than `'A'`? There's no `else` clause for the first `if`, so the program just ends without outputting anything. To deal with this, you must add another `else` clause at the end of the program. The complete nested `if` would then become the following:

```
  if(letter >= 'A')                     /* Is it A or greater?              */
    if(letter <= 'Z')                   /* and is it Z or lower?            */
    {                                   /* It is uppercase                  */
      letter = letter - 'A'+ 'a';       /* Convert from upper- to lowercase */
      printf("You entered an uppercase %c\n", letter);
    }
    else                                /* It is not an uppercase letter    */
      printf("Try using the shift key, Bud! I want a capital letter.\n");
  else
    printf("You didn't enter an uppercase letter\n");
```

Now you always get a message. Note the indentation to show which `else` belongs to which `if`. The indentation doesn't determine what belongs to what. It just provides a visual cue. An `else` always belongs to the `if` that immediately precedes it that isn't already spoken for by another `else`.

So how would this look if you were working with wide characters? Not that different really:

```
/* Program 3.4A Converting uppercase to lowercase using wide characters */
#include <stdio.h>

int main(void)
{
  wchar_t letter = 0;                   /* Stores a character               */

  printf("Enter an uppercase letter:");  /* Prompt for input                */
  scanf("%lc", &letter);                /* Read a character                 */

  /* Check whether the input is uppercase */
  if(letter >= L'A')                    /* Is it A or greater?              */
    if(letter <= L'Z')                  /* and is it Z or lower?            */
    {                                   /* It is uppercase                  */
      letter = letter - L'A'+ L'a';     /* Convert from upper- to lowercase */
      printf("You entered an uppercase %lc\n", letter);
    }
    else                                /* It is not an uppercase letter    */
      printf("Try using the shift key, Bud! I want a capital letter.\n");
  return 0;
}
```

The type of the variable letter is now wchar_t and the character constants all have L in front to make them wide characters. The only other differences are the format specifications for input and output where you use %lc instead of %c.

Of course, you might want to be sure here that the conversion from uppercase to lowercase operation works regardless of the code values, but the tolower() and toupper() functions I mentioned earlier won't work with wide characters. However, the <wctype.h> header file defines the towlower() and towupper() functions that will. With the header file included you could write the statement that does the conversion as:

```
letter = towlower(letter); /* Convert from upper- to lowercase */
```

You used a nested if statement to check for two conditions in the example but, as you can imagine, this could get very confusing when you've got a lot of different criteria that you need to check for. The good news is that C allows you to use logical operators to simplify the situation.

Logical Operators

Sometimes it just isn't enough to perform a single test for a decision. You may want to combine two or more checks on values and, if they're all true, perform a certain action. Or you may want to perform a calculation if one or more of a set of conditions are true.

For example, you may only want to go to work if you're feeling well *and* it's a weekday. Just because you feel great doesn't mean you want to go in on a Saturday or a Sunday. Alternatively, you could say that you'll stay at home if you feel ill *or* if it's a weekend day. These are exactly the sorts of circumstances for which the logical operators are intended.

The AND Operator &&

You can look first at the logical AND operator, &&. This is another binary operator because it operates on two items of data. The && operator combines two logical expressions—that is, two expressions that have a value true or false. Consider this expression:

```
Test1 && Test2
```

This expression evaluates to true if both expressions Test1 and Test2 evaluate to true. If either or both of the operands for the && operator are false, the result of the operation is false.

The obvious place to use the && operator is in an if expression. Let's look at an example:

```
if(age > 12 && age < 20)
  printf("You are officially a teenager.");
```

The printf() statement will be executed only if age has a value between 13 and 19 inclusive.

Of course, the operands of the && operator can be _Bool variables. You could replace the previous statement with the following:

```
_Bool test1 = age > 12;
_Bool test2 = age < 20;
if(test1 && test2)
  printf("You are officially a teenager.");
```

The values of the two logical expressions checking the value of age are stored in the variables test1 and test2. The if expression is now much simpler using the _Bool variables as operands.

Naturally, you can use more than one of these logical operators in an expression:

```
if(age > 12 && age < 20 && savings > 5000)
  printf("You are a rich teenager.");
```

All three conditions must be true for the `printf()` to be executed. That is, the `printf()` will be executed only if the value of `age` is between 13 and 19 inclusive, and the value of `savings` is greater than 5000.

The OR Operator ||

The logical `OR` operator, `||`, covers the situation in which you want to check for any of two or more conditions being true. If either or both operands of the `||` operator is `true`, the result is `true`. The result is `false` only when both operands are false. Here's an example of using this operator:

```
if(a < 10 || b > c || c > 50)
  printf("At least one of the conditions is true.");
```

The `printf()` will be executed only if *at least* one of the three conditions, `a<10`, `b>c`, or `c<50`, is true. When `a`, `b`, and `c` all have the value 9, for instance, this will be the case. Of course, the `printf()` will also be executed when two of the conditions are `true`, as well as all three.

You can use the `&&` and `||` logical operators in combination, as in the following code fragment:

```
 if((age > 12 && age < 20) || savings > 5000)
   printf ("Either you're a teenager, or you're rich, or possibly both.");
```

The `printf()` statement will be executed if the value of `age` is between 12 and 20 or the value of `savings` is greater than 5000, or both. As you can see, when you start to use more operators, things can get confusing. The parentheses around the expression that is the left operand of the `||` operator are not strictly necessary but I put them in to make the condition easier to understand. Making use of Boolean variables can help. You could replace the previous statement with the following:

```
bool age_test1 = age > 12;
bool age_test2 = age < 20;
bool age_check = test1 && test2;
bool savings_check = savings > 5000;
if((age_check || savings_check)
 printf ("Either you're a teenager, or you're rich, or possibly both.");
```

Now you have declared four Boolean variables using `bool`, which assumes the `<stdbool.h>` header has been included into the source file. You should be able to see that the `if` statement works with essentially the same test as before. Of course, you could define the value of `age_check` in a single step, like this:

```
bool age_check = age > 12 && age < 20;
bool savings_check = savings > 5000;
if((age_check || savings_check)
 printf ("Either you're a teenager, or you're rich, or possibly both.");
```

This reduces the number of variables you use and still leaves the code reasonably clear.

The NOT Operator !

Last but not least is the logical `NOT` operator, represented by `!`. The `!` operator is a unary operator, because it applies to just one operand. The logical `NOT` operator reverses the value of a logical expression: `true` becomes `false`, and `false` becomes `true`. Suppose you have two variables, `a` and `b`, with the values 5 and 2 respectively; then the expression `a>b` is `true`. If you use the logical `NOT` operator, the expression `!(a>b)` is `false`. I recommend that you avoid using this operator as far as possible; it

tends to result in code that becomes difficult to follow. As an illustration of how not to use NOT, you can rewrite the previous example as follows:

```
if((!(age >= 12) && !(age >= 20)) || !(savings <= 5000))
{
  printf("\nYou're either not a teenager and rich ");
  printf("or not rich and a teenager,\n");
  printf("or neither not a teenager nor not rich.");
}
```

As you can see, it becomes incredibly difficult to unravel the nots!

TRY IT OUT: A BETTER WAY TO CONVERT LETTERS

Earlier in this chapter you tried a program in which the user was prompted to enter an uppercase character. The program used a nested if to ensure that the input was of the correct type, and then wrote the small-letter equivalent or a remark indicating that the input was of the wrong type to the command line.

You can now see that all this was completely unnecessary, because you can achieve the same result like this:

```
/* Program 3.5   Testing letters the easy way */
#include <stdio.h>
int main(void)
{
  char letter =0;                          /* Stores an input character */

  printf("Enter an upper case letter:");   /* Prompt for input          */
  scanf(" %c", &letter);                   /* Read the input character  */

  if((letter >= 'A') && (letter <= 'Z'))   /* Verify uppercase letter  */
  {
    letter += 'a'-'A';                     /* Convert to lowercase     */
    printf("You entered an uppercase %c.\n", letter);
  }
  else
    printf("You did not enter an uppercase letter.\n");
  return 0;
}
```

The output will be similar to that from the earlier example.

How It Works

The output is similar but not exactly the same as the original program. In the corrected version of the program, you generated a different message when the input was less than 'A'. This version is rather better, though. Compare the mechanism to test the input in the two programs and you'll see how much neater the second solution is. This is the original version:

```
if(letter >= 'A')
  if(letter <= 'Z')
```

This is the new version:

```
  if((letter >= 'A') && (letter <= 'Z'))   /* Verify uppercase letter  */
```

Rather than having confusing nested `if` statements, here you've checked that the character entered is greater than `'A'` *and* less than `'Z'` in one statement. Notice that you put extra parentheses around the two expressions to be checked. They aren't really needed in this case, but they don't hurt, and they leave you or any other programmer in no doubt as to the order of execution.

There's also a slightly simpler way of expressing the conversion to lowercase:

```
letter += 'a'-'A';                    /* Convert to lowercase     */
```

Now you use the `+=` operator to add the difference between `'a'` and `'A'` to the character code value stored in `letter`.

If you add an `#include` directive for the `<ctype.h>` standard header file to the source, you could use the `tolower()` function to do the same thing:

```
letter = tolower(letter);
```

The lowercase letter that the `tolower()` function returns is stored back in the variable `letter`. The `toupper()` function that is also declared in `<ctype.h>` converts the argument to uppercase.

The Conditional Operator

There's another operator called the **conditional operator** that you can use to test data. It evaluates one of two expressions depending on whether a logical expression evaluates `true` or `false`.

Because three operands are involved—the logical expression plus two other expressions—this operator is also referred to as the **ternary operator**. The general representation of an expression using the conditional operator looks like this:

```
condition ? expression1 : expression2
```

Notice how the operator is arranged in relation to the operands. There is `?` following the logical expression, `condition`, to separate it from the next operand, `expression1`. This is separated from the third operand, `expression2`, by a colon. The value that results from the operation will be produced by evaluating `expression1` if `condition` evaluates to `true`, or by evaluating `expression2` if `condition` evaluates to `false`. Note that only one of `expression1` and `expression2` will be evaluated. Normally this is of little significance, but sometimes this is important.

You can use the conditional operator in a statement such as this:

```
x = y > 7 ? 25 : 50;
```

Executing this statement will result in `x` being set to 25 if `y` is greater than 7, or to 50 otherwise. This is a nice shorthand way of producing the same effect as this:

```
if(y > 7)
  x = 25;
else
  x = 50;
```

The conditional operator enables you to express some things economically. An expression for the minimum of two variables can be written very simply using the conditional operator. For example, you could write an expression that compared two salaries and obtained the greater of the two, like this:

```
your_salary > my_salary ? your_salary : my_salary
```

Of course, you can use the conditional operator in a more complex expression. Earlier in Program 3.2 you calculated a quantity price for a product using an `if-else` statement. The price was

$3.50 per item with a discount of 5 percent for quantities over ten. You can do this sort of calculation in a single step with the conditional operator:

```
total_price = unit_price*quantity*(quantity>10 ? 1.0 : 0.95);
```

TRY IT OUT: USING THE CONDITIONAL OPERATOR

This discount business could translate into a short example. Suppose you have the unit price of the product still at $3.50, but you now offer three levels of discount: 15 percent for more than 50, 10 percent for more than 20, and the original 5 percent for more than 10. Here's how you can handle that:

```
/* Program 3.6 Multiple discount levels */
#include <stdio.h>

int main(void)
{
  const double unit_price = 3.50; /* Unit price in dollars     */
  const double discount1 = 0.05;  /* Discount for more than 10 */
  const double discount2 = 0.1;   /* Discount for more than 20 */
  const double discount3 = 0.15;  /* Discount for more than 50 */
  double total_price = 0.0;
  int quantity = 0;

  printf("Enter the number that you want to buy:");
  scanf(" %d", &quantity);

  total_price = quantity*unit_price*(1.0 -
                  (quantity>50 ? discount3 : (
                          quantity>20 ? discount2 : (
                                 quantity>10 ? discount1 : 0.0))));

  printf("The price for %d is $%.2f\n", quantity, total_price);
  return 0;
}
```

Some typical output from the program is as follows:

```
Enter the number that you want to buy:60
The price for 60 is $178.50
```

How It Works

The interesting bit is the statement that calculates the total price for the quantity that's entered. The statement uses three conditional operators, so it takes a little unraveling:

```
  total_price = quantity*unit_price*(1.0 -
                  (quantity>50 ? discount3 : (
                          quantity>20 ? discount2 : (
                                 quantity>10 ? discount1 : 0.0))));
```

You can understand how this produces the correct result by breaking it into pieces. The basic price is produced by the expression `quantity*unit_price`, which simply multiplies the unit price by the quantity ordered. The result of this has to be multiplied by a factor that's determined by the quantity. If the quantity is over 50, the basic price must be multiplied by `(1.0-discount3)`. This is determined by an expression like the following:

```
(1.0 - quantity > 50 ? discount3 : something_else)
```

If `quantity` is greater than 50 here, the expression will amount to `(1.0-discount3)`, and the right side of the assignment is complete. Otherwise, it will be `(1.0-something_else)`, where `something_else` is the result of another conditional operator.

Of course, if `quantity` isn't greater than 50, it may still be greater than 20, in which case you want `something_else` to be `discount2`. This is produced by the conditional operator that appears in the `something_else` position in the statement:

```
(quantity>20 ? discount2 : something_else_again)
```

This will result in `something_else` being `discount2` if the value of `quantity` is over 20, which is precisely what you want, and `something_else_again` if it isn't. You want `something_else_again` to be `discount1` if `quantity` is over 10, and 0 if it isn't. The last conditional operator that occupies the `something_else_again` position in the statement does this:

```
(quantity>10 ? discount1 : 0.0)
```

And that's it!

In spite of its odd appearance, you'll see the conditional operator crop up quite frequently in C programs. A very handy application of this operator that you'll see in examples in this book and elsewhere is to vary the contents of a message or prompt depending on the value of an expression. For example, if you want to display a message indicating the number of pets that a person has, and you want the message to change between singular and plural automatically, you could write this:

```
printf("You have %d pet%s.", pets, pets == 1 ? "" : "s" );
```

You use the `%s` specifier when you want to output a string. If `pets` is equal to 1, an empty string will be output in place of the `%s`; otherwise, `"s"` will be output. Thus, if `pets` has the value 1, the statement will output this message:

```
You have 1 pet.
```

However, if the variable `pets` is 5, you will get this output:

```
You have 5 pets.
```

You can use this mechanism to vary an output message depending on the value of an expression in many different ways: `she` instead of `he`, `wrong` instead of `right`, and so on.

Operator Precedence: Who Goes First?

With all the parentheses you've used in the examples in this chapter, now is a good time to come back to operator precedence. Operator precedence determines the sequence in which operators in an expression are executed. You have the logical operators &&, ==, !=, and ||, plus the comparison operators and the arithmetic operators. When you have more than one operator in an expression, how do you know which ones are used first? This order of precedence can affect the result of an expression substantially.

For example, suppose you are to process job applications and you want to only accept applicants who are 25 or older and have graduated from Harvard or Yale. Here's the age condition you can represent by this conditional expression:

```
Age >= 25
```

Suppose that you represent graduation by the variables `Yale` and `Harvard`, which may be true or false. Now you can write the condition as follows:

```
Age >= 25 && Harvard || Yale
```

Unfortunately, this will result in howls of protest because you'll now accept Yale graduates who are under 25. In fact, this statement will accept Yale graduates of any age. But if you're from Harvard, you must be 25 or over to be accepted. Because of operator precedence, this expression is effectively the following:

```
(Age >= 25 && Harvard) || Yale
```

So you take anybody at all from Yale. I'm sure those wearing a Y-front sweatshirt will claim that this is as it should be, but what you really meant was this:

```
Age >= 25 && (Harvard || Yale)
```

Because of operator precedence, you must put the parentheses in to force the order of operations to be what you want.

In general, the precedence of the operators in an expression determines whether it is necessary for you to put parentheses in to get the result you want, but if you are unsure of the precedence of the operators you are using, it does no harm to put the parentheses in. Table 3-2 shows the order of precedence for all the operators in C, from highest at the top to lowest at the bottom.

There are quite a few operators in the table that we haven't addressed yet. You'll see the operators ~, <<, >>, &, ^, and | later in this chapter in the "Bitwise Operators" section and you'll learn about the rest later in the book.

All the operators that appear in the same row in the table are of equal precedence. The sequence of execution for operators of equal precedence is determined by their associativity, which determines whether they're selected from left to right or from right to left. Naturally, parentheses around an expression come at the very top of the list of operators because they're used to override the natural priorities defined.

Table 3-2. *Operator Order of Precedence*

Operators	Description	Associativity
`( )`	Parenthesized expression	Left-to-right
`[]`	Array subscript	
`.`	Member selection by object	
`->`	Member selection by pointer	
`+ -`	Unary + and -	Right-to-left
`++ --`	Prefix increment and prefix decrement	
`! ~`	Logical `NOT` and bitwise complement	
`*`	Dereference	
`&`	Address-of	
`sizeof`	Size of expression or type	
`(type)`	Explicit cast to `type` such as `(int)` or `(double)`	
	Type casts such as `(int)` or `(double)`	
`* / %`	Multiplication and division and modulus (remainder)	Left-to-right
`+ -`	Addition and subtraction	Left-to-right
`<< >>`	Bitwise shift left and bitwise shift right	Left-to-right
`< <=`	Less than and less than or equal to	Left-to-right
`> >=`	Greater than and greater than or equal to	
`== !=`	Equal to and not equal to	Left-to-right
`&`	Bitwise `AND`	Left-to-right
`^`	Bitwise exclusive `OR`	Left-to-right
`|`	Bitwise `OR`	Left-to-right
`&&`	Logical `AND`	Left-to-right
`||`	Logical `OR`	Left-to-right
`?:`	Conditional operator	Right-to-left
`=`	Assignment	Right-to-left
`+= -=`	Addition assignment and subtraction assignment	
`/= *=`	Division assignment and multiplication assignment	
`%=`	Modulus assignment	
`<<= >>=`	Bitwise shift left assignment and bitwise shift right assignment	
`&= |=`	Bitwise `AND` assignment and bitwise `OR` assignment	
`^=`	Bitwise exclusive `OR` assignment	
`,`	Comma operator	Left-to-right

As you can see from Table 3-2, all the comparison operators are below the binary arithmetic operators in precedence, and the binary logical operators are below the comparison operators. As a result, arithmetic is done first, then comparisons, and then logical combinations. Assignments come last in this list, so they're only performed once everything else has been completed. The conditional operator squeezes in just above the assignment operators.

Note that the ! operator is highest within the set of logical operators. Consequently the parentheses around logical expressions are essential when you want to negate the value of a logical expression.