TRY IT OUT: USING LOGICAL OPERATORS WITHOUT CONFUSION

Suppose you want a program that will take applicant interviews for a large pharmaceutical corporation. The program should offer interviews to applicants who meet certain educational specifications. An applicant who meets any of the following criteria should be accepted for an interview:

1. Graduates over 25 who studied chemistry and who didn't graduate from Yale
2. Graduates from Yale who studied chemistry
3. Graduates from Harvard who studied economics and aren't older than 28
4. Graduates from Yale who are over 25 and who didn't study chemistry

One program to implement this policy is as follows:

```
/* Program 3.7 Confused recruiting policy  */
#include <stdio.h>

int main(void)
{
  int age = 0;              /* Age of the applicant              */
  int college = 0;          /* Code for college attended         */
  int subject = 0;          /* Code for subject studied          */
  bool interview = false;   /* true for accept, false for reject */

  /* Get data on the applicant */
  printf("\nWhat college? 1 for Harvard, 2 for Yale, 3 for other: ");
  scanf("%d",&college);
  printf("\nWhat subject? 1 for Chemistry, 2 for economics, 3 for other: ");
  scanf("%d", &subject);
  printf("\nHow old is the applicant? ");
  scanf("%d",&age);

  /* Check out the applicant */
  if((age>25 && subject==1) && (college==3 || college==1))
    interview = true;
  if(college==2 &&subject ==1)
    interview = true;
  if(college==1 && subject==2 && !(age>28))
    interview = true;
  if(college==2 && (subject==2 || subject==3) && age>25)
    interview = true;

  /* Output decision for interview */
  if(interview)
    printf("\n\nGive 'em an interview");
  else
    printf("\n\nReject 'em");
  return 0;
}
```

The output from this program should be something like this:

```
What college? 1 for Harvard, 2 for Yale, 3 for other: 2
What subject? 1 for Chemistry, 2 for Economics, 3 for other: 1
How old is the applicant? 24

Give 'em an interview
```

How It Works

The program works in a fairly straightforward way. The only slight complication is with the number of operators and `if` statements needed to check a candidate out:

```
if((age>25 && subject==1) && (college==3 || college==1))
  interview =true;
if(college==2 &&subject ==1)
  interview = true;
if(college==1 && subject==2 && !(age>28))
  interview = true;
if(college==2 && (subject==2 || subject==3) && age>25)
  interview = true;
```

The final `if` statement tells you whether to invite the applicant for an interview or not; it uses the variable `interview`:

```
if(interview)
  printf("\n\nGive 'em an interview");
else
  printf("\n\nReject 'em");
```

The variable `interview` is initialized to `false`, but if any of the criteria is met, you assign the value `true` to it. The `if` expression is just the variable `interview`, so the expression is `false` when `interview` is 0 and true when `interview` has any nonzero value.

This could be a lot simpler, though. Let's look at the conditions that result in an interview. You can specify each criterion with an expression as shown in the following table:

Expressions for Selecting Candidates

Criterion	Expression
Graduates over 25 who studied chemistry and who didn't graduate from Yale	`age>25 && college!=2`
Graduates from Yale who studied chemistry	`college==2 && subject==1`
Graduates from Harvard who studied economics and aren't older than 28	`college==1 && subject==2 && age<=28`
Graduates from Yale who are over 25 and who didn't study chemistry	`college==2 && age>25 && subject!=1`

The variable `interview` should be set to `true` if any of these four conditions is `true`, so you can now combine them using the `||` operator to set the value of the variable `interview`:

```
interview = (age>25 && college!=2) || (college==2 && subject==1) ||
                            (college==1 && subject==2 && age<=28) ||
                               (college==2 && age>25 && subject!=1);
```

Now you don't need the `if` statements to check the conditions at all. You just store the logical value, `true` or `false`, which arises from combining these expressions. In fact, you could dispense with the variable `interview` altogether by just putting the combined expression for the checks into the last `if`:

```
if((age>25 && college!=2) || (college==2 && subject==1) ||
                    (college==1 && subject==2 && age<=28) ||
                       (college==2 && age>25 && subject!=1))
  printf("\n\nGive 'em an interview");
else
  printf("\n\nReject 'em");
```

So you end up with a much shorter, if somewhat less readable program.

Multiple-Choice Questions

Multiple-choice questions come up quite often in programming. One example is selecting a different course of action depending on whether a candidate is from one or other of six different universities. Another example is when you want to choose to execute a particular set of statements depending on which day of the week it is. You have two ways to handle multiple-choice situations in C. One is a form of the `if` statement described as the `else-if` that provides the most general way to deal with multiple choices. The other is the `switch` statement, which is restricted in the way a particular choice is selected; but where it does apply, it provides a very neat and easily understood solution. Let's look at the `else-if` statement first.

Using else-if Statements for Multiple Choices

The use of the `else-if` statement for selecting one of a set of choices looks like this:

```
if(choice1)
  /* Statement or block for choice 1 */
else if(choice2)
  /* Statement or block for choice 2 */
else if(choice3)
  /* Statement or block for choice 2 */

/* … and so on …   */
else
  /* Default statement or block  */
```

Each `if` expression can be anything as long as the result is `true` or `false`. If the first `if` expression, `choice1`, is false, the next `if` is executed. If `choice2` is false, the next `if` is executed. This continues until an expression is found to be true, in which case the statement or block of statements for that `if` is executed. This ends the sequence, and the statement following the sequence of `else-if` statements is executed next.

If all of the `if` conditions are `false`, the statement or block following the final `else` will be executed. You can omit this final `else`, in which case the sequence will do nothing if all the `if` conditions are `false`. Here's a simple illustration of this:

```
if(salary<5000)
  printf("Your pay is very poor.");    /* pay < 5000            */
else if(salary<15000)
  printf("Your pay is not good.");     /* 5000 <= pay < 15000   */
else if(salary<50000)
  printf("Your pay is not bad.");      /* 15000 <= pay < 50000  */
else if(salary<100000)
  printf("Your pay is very good.");    /* 50000 <= pay < 100000 */
else
  printf("Your pay is exceptional.");  /* pay > 100000          */
```

Note that you don't need to test for lower limits in the if conditions after the first. This is because if you reach a particular if, the previous test must have been false.

Because any logical expressions can be used as the if conditions, this statement is very flexible and allows you to express a selection from virtually any set of choices. The switch statement isn't as flexible, but it's simpler to use in many cases. Let's take a look at the switch statement.

The switch Statement

The switch statement enables you to choose one course of action from a set of possible actions, based on the result of an integer expression. Let's start with a simple illustration of how it works.

Imagine that you're running a raffle or a sweepstakes. Suppose ticket number 35 wins first prize, number 122 wins second prize, and number 78 wins third prize. You could use the switch statement to check for a winning ticket number as follows:

```
switch(ticket_number)
{
  case 35:
    printf("Congratulations! You win first prize!");
    break;
  case 122:
    printf("You are in luck - second prize.");
    break;
  case 78:
    printf("You are in luck - third prize.");
    break;
  default:
    printf("Too bad, you lose.");
}
```

The value of the expression in parentheses following the keyword switch, which is ticket_number in this case, determines which of the statements between the braces will be executed. If the value of ticket_number matches the value specified after one of the case keywords, the following statements will be executed. If ticket_number has the value 122, for example, this message will be displayed:

```
You are in luck - second prize.
```

The effect of the break statement following the printf() is to skip over the other statements within that block and continue with whatever statement follows the closing brace. If you were to omit the break statement for a particular case, when the statements for that case are executed, execution would continue with the statements for the next case. If ticket_number has a value that doesn't correspond to any of the case values, the statements that follow the `default` keyword are executed, so you simply get the default message. Both `default` and `break` are keywords in C.

The general way of describing the `switch` statement is as follows:

```
switch(integer_expression)
{
  case constant_expression_1:
    statements_1;
    break;
    ....
  case constant_expression_n:
    statements_n;
    break;
  default:
    statements;
}
```

The test is based on the value of `integer_expression`. If that value corresponds to one of the case values defined by the associated `constant_expression_n` values, the statements following that case value are executed. If the value of `integer_expression` differs from every one of the case values, the statements following `default` are executed. Because you can't reasonably expect to select more than one case, all the case values must be different. If they aren't, you'll get an error message when you try to compile the program. The case values must all be **constant expressions**, which are expressions that can be evaluated at compile time. This means that a case value can't be dependent on a value that's determined when your program executes. Of course, the test expression `integer_expression` can be anything at all, as long as it evaluates to an integer.

You can leave out the `default` keyword and its associated statements. If none of the case values match, then nothing happens. Notice, however, that all of the case values for the associated `constant_expression` must be different. The `break` statement jumps to the statement after the closing brace.

Notice the punctuation and formatting. There's no semicolon at the end of the first `switch` expression. The body of the statement is enclosed within braces. The `constant_expression` value for a case is followed by a colon, and each subsequent statement ends with a semicolon, as usual.

Because an `enumeration` type is an `integer` type, you can use a variable of an `enumeration` type to control a `switch`. Here's an example:

```
enum Weekday {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};
enum Weekday today = Wednesday;
switch(today)
{
  case Sunday:
    printf("Today is Sunday.");
    break;
  case Monday:
    printf("Today is Monday.");
    break;
  case Tuesday:
    printf("Today is Tuesday.");
    break;
  case Wednesday:
    printf("Today is Wednesday.");
    break;
  case Thursday:
    printf("Today is Thursday.");
    break;
  case Friday:
    printf("Today is Friday.");
    break;
  case Saturday:
    printf("Today is Saturday.");
    break;
  }
```

This switch selects the case that corresponds to the value of the variable today, so in this case the message will be that today is Wednesday. There's no default case in the switch but you could put one in to guard against an invalid value for today.

You can associate several case values with one group of statements. You can also use an expression that results in a value of type char as the control expression for a switch. Suppose you read a character from the keyboard into a variable, ch, of type char. You can test this character in a switch like this:

```
switch(tolower(ch))
{
  case 'a': case 'e': case 'i': case 'o': case 'u':
    printf("The character is a vowel.");
    break;
  case 'b': case 'c': case 'd': case 'f': case 'g': case 'h': case 'j': case 'k':
  case 'l': case 'm': case 'n': case 'p': case 'q': case 'r': case 's': case 't':
  case 'v': case 'w': case 'x': case 'y': case 'z':
    printf("The character is a consonant.");
    break;
  default:
    printf("The character is not a letter.");
    break;
}
```

Because you use the function tolower() that is declared in the <ctype.h> header file to convert the value of ch to lowercase, you only need to test for lowercase letters. In the case in which ch contains the character code for a vowel, you output a message to that effect because for the five case values corresponding to vowels you execute the same printf() statement. Similarly, you output a suitable message when ch contains a consonant. If ch contains a code that's neither a consonant nor a vowel, the default case is executed.

Note the break statement after the default case. This isn't necessary, but it does have a purpose. By always putting a break statement at the end of the last case, you ensure that the switch still works correctly if you later add a new case at the end.

You could simplify the switch by making use of another function that's declared in the <ctype.h> header. The isalpha() function will return a nonzero integer (thus true) if the character that's passed as the argument is an alphabetic character, and it will return 0 (false) if the character isn't an alphabetic character. You could therefore produce the same result as the previous switch with the following code:

```
if(!isalpha(ch))
    printf("The character is not a letter.");
else
  switch(tolower(ch))
  {
    case 'a': case 'e': case 'i': case 'o': case 'u':
      printf("The character is a vowel.");
    break;
  default:
    printf("The character is a consonant.");
    break;
  }
```

The `if` statement tests for `ch` not being a letter, and if this is so, it outputs a message. If `ch` is a letter, the `switch` statement will sort out whether it is a vowel or a consonant. The five vowel case values produce one output, and the default case produces the other. Because you know that `ch` contains a letter when the `switch` statement executes, if `ch` isn't a vowel, it must be a consonant.

As well as the `tolower()`, `toupper()`, and `isalpha()` functions that I've mentioned, the `<ctype.h>` header also declares several other useful functions for testing a character, as shown in Table 3-3.

Table 3-3. *Functions for Testing Characters*

Function	Tests For
`islower()`	Lowercase letter
`isupper()`	Uppercase letter
`isalnum()`	Uppercase or lowercase letter
`iscntrl()`	Control character
`isprint()`	Any printing character including space
`isgraph()`	Any printing character except space
`isdigit()`	Decimal digit (`'0'` to `'9'`)
`isxdigit()`	Hexadecimal digit (`'0'` to `'9'`, `'A'` to `'F'`, `'a'` to `'f'`)
`isblank()`	Standard blank characters (space, `'\t'`)
`isspace()`	Whitespace character (space, `'\n'`, `'\t'`, `'\v'`, `'\r'`, `'\f'`)
`ispunct()`	Printing character for which `isspace()` and `isalnum()` return `false`

In each case, the function returns a nonzero `integer` value (which is interpreted as `true`) if it finds what it's testing for and 0 (`false`) otherwise.

Let's look at the `switch` statement in action with an example.

TRY IT OUT: PICKING A LUCKY NUMBER

This example assumes that you're operating a lottery in which there are three winning numbers. Participants are required to guess a winning number, and the `switch` statement is designed to end the suspense and tell them about any valuable prizes they may have won:

```
/* Program 3.8 Lucky Lotteries    */
#include <stdio.h>

int main(void)
{
  int choice = 0;                 /* The number chosen              */
  /* Get the choice input */
  printf("\nPick a number between 1 and 10 and you may win a prize! ");
  scanf("%d",&choice);

  /* Check for an invalid selection */
  if((choice>10) || (choice <1))
    choice = 11;                  /* Selects invalid choice message */

  switch(choice)
  {
    case 7:
      printf("\nCongratulations!");
      printf("\nYou win the collected works of Amos Gruntfuttock.");
      break;                      /* Jumps to the end of the block  */

    case 2:
      printf("\nYou win the folding thermometer-pen-watch-umbrella.");
      break;                      /* Jumps to the end of the block  */

    case 8:
      printf("\nYou win the lifetime supply of aspirin tablets.");
      break;                      /* Jumps to the end of the block  */

    case 11:
      printf("\nTry between 1 and 10. You wasted your guess.");
                /* No break - so continue with the next statement */

    default:
      printf("\nSorry, you lose.\n");
      break;             /* Defensive break - in case of new cases */
  }
  return 0;
}
```

Typical output from this program will be the following:

```
Pick a number between 1 and 10 and you may win a prize! 3
Sorry, you lose.
```

or

```
Pick a number between 1 and 10 and you may win a prize! 7
Congratulations!
You win the collected works of Amos Gruntfuttock.
```

or, if you enter an invalid number

```
Pick a number between 1 and 10 and you may win a prize! 92
Try between 1 and 10. You wasted your guess.
Sorry, you lose.
```

How It Works

You do the usual sort of thing to start with. You declare an integer variable choice. Then you ask the user to enter a number between 1 and 10 and store the value the user types in choice:

```
int choice = 0;                /* The number chosen             */

/* Get the choice input */
printf("\nPick a number between 1 and 10 and you may win a prize! ");
scanf("%d",&choice);
```

Before you do anything else, you check that the user has really entered a number between 1 and 10:

```
/* Check for an invalid selection */
if((choice>10) || (choice <1))
  choice = 11;                 /* Selects invalid choice message */
```

If the value is anything else, you automatically change it to 11. You don't have to do this, but to ensure the user is advised of his or her mistake, you set the variable choice to 11, which produces the error message generated by the printf() for that case value.

Next, you have the switch statement, which will select from the cases between the braces that follow depending on the value of choice:

```
switch(choice)
{
  ...
}
```

If choice has the value 7, the case corresponding to that value will be executed:

```
  case 7:
    printf("\nCongratulations!");
    printf("\nYou win the collected works of Amos Gruntfuttock.");
    break;                     /* Jumps to the end of the block  */
```

The two printf() calls are executed, and the break will jump to the statement following the closing brace for the block (which ends the program, in this case, because there isn't one).

The same goes for the next two cases:

```
  case 2:
    printf("\nYou win the folding thermometer-pen-watch-umbrella.");
    break;                     /* Jumps to the end of the block  */

  case 8:
    printf("\nYou win the lifetime supply of aspirin tablets.");
    break;                     /* Jumps to the end of the block  */
```

These correspond to values for the variable choice of 2 or 8.

The next case is a little different:

```
  case 11:
    printf("\nTry between 1 and 10, you wasted your guess.");
                 /* No break – so continue with the next statement */
```

There's no break statement, so execution continues with the printf() for the default case after displaying the message. The upshot of this is that you get both lines of output if choice has been set to 11. This is entirely appropriate in this case, but usually you'll want to put a `break` statement at the end of each case. Remove the `break` statements from the program and try entering 7 to see why. You'll get all the output messages following any particular case.

The default case is as follows:

```
default:
  printf("\nSorry, you lose.\n");
  break;              /* Defensive break - in case of new cases */
```

This will be selected if the value of `choice` doesn't correspond to any of the other case values. You also have a `break` statement here. Although it isn't strictly necessary, many programmers always put a `break` statement after the default `case` statements or whichever is the last case in the `switch`. This provides for the possibility of adding further `case` statements to the `switch`. If you were to forget the break after the default case in such circumstances the `switch` won't do what you want. The `case` statements can be in any order in a `switch`, and `default` doesn't have to be the last.

TRY IT OUT: YES OR NO

Let's see the `switch` statement in action controlled by a variable of type `char` where the value is entered by the user. You'll prompt the user to enter the value `'y'` or `'Y'` for one action and `'n'` or `'N'` for another. On its own, this program may be fairly useless, but you've probably encountered many situations in which a program has asked just this question and then performed some action as a result (saving a file, for example):

```
/* Program 3.9 Testing cases */
#include <stdio.h>

int main(void)
{
  char answer = 0;                /* Stores an input character */

  printf("Enter Y or N: ");
  scanf(" %c", &answer);

  switch(answer)
  {
    case 'y': case 'Y':
      printf("\nYou responded in the affirmative.");
      break;

    case 'n': case 'N':
      printf("\nYou responded in the negative.");
      break;

    default:
      printf("\nYou did not respond correctly...");
      break;
  }
  return 0;
}
```

Typical output from this would be the following:

```
Enter Y or N: y
You responded in the affirmative.
```

How It Works

When you declare the variable `answer` as type `char`, you also take the opportunity to initialize it to 0. You then ask the user to type something in and store that value as usual:

```
  char answer = 0;                /* Stores an input character */

  printf("Enter Y or N: ");
  scanf(" %c", &answer);
```

The `switch` statement uses the character stored in `letter` to select a case:

```
switch(answer)
{
  ...
}
```

The first case in the `switch` provides for the possibility of the user entering an uppercase or a lowercase letter Y:

```
    case 'y': case 'Y':
      printf("\nYou responded in the affirmative.");
      break;
```

Both values `'y'` and `'Y'` will result in the same `printf()` being executed. In general, you can put as many cases together like this as you want. Notice the punctuation for this. The two cases just follow one another and each has a terminating colon after the case value.

The negative input is handled in a similar way:

```
    case 'n': case 'N':
      printf("\nYou responded in the negative.");
      break;
```

If the character entered doesn't correspond with any of the case values, the default case is selected:

```
    default:
      printf("\nYou did not respond correctly...");
      break;
```

Note the `break` statement after the `printf()` statements for the default case, as well as the legal case values. As before, this causes execution to break off at that point and continue after the end of the `switch` statement. Again, without it you'd get the statements for succeeding cases executed and, unless there's a `break` statement preceding the valid cases, you'd get the following statement (or statements), including the `default` statement, executed as well.

Of course, you could also use the `toupper()` or `tolower()` function to simplify the cases in the `switch`. By using one or the other you can nearly halve the number of cases:

```
switch(toupper(answer))
{
  case 'Y':
    printf("\nYou responded in the affirmative.");
    break;
  case 'N':
    printf("\nYou responded in the negative.");
    break;
  default:
    printf("\nYou did not respond correctly...");
    break;
 }
```

Remember, you need an `#include` directive for `<ctype.h>` if you want to use the `toupper()` function.

The goto Statement

The `if` statement provides you with the ability to choose one or the other of two blocks of statements, depending on a test. This is a powerful tool that enables you to alter the naturally sequential nature of a program. You no longer have to go from A to B to C to D. You can go to A and then decide whether to skip B and C and go straight to D.

The `goto` statement, on the other hand, is a blunt instrument. It directs the flow of statements to change *unconditionally*—do not pass Go, do not collect $200, go directly to jail. When your program hits a `goto`, it does just that. It goes to the place you send it, without checking any values or asking the user whether it is really what he or she wants.

I'm only going to mention the `goto` statement very briefly, because it isn't as great as it might at first seem. The problem with `goto` statements is that they seem too easy. This might sound perverse, but the important word is *seems*. It feels so simple that you can be tempted into using it all over the place, where it would be better to use a different statement. This can result in heavily tangled code.

When you use the `goto` statement, the position in the code to be moved to is defined by a statement label at that point. A statement label is defined in exactly the same way as a variable name, which is a sequence of letters and digits, the first of which must be a letter. The statement label is followed by a colon (:) to separate it from the statement it labels. If you think this sounds like a case label in a `switch`, you would be right. Case labels are statement labels.

Like other statements, the `goto` statement ends with a semicolon:

```
goto there;
```

The destination statement must have the same label as appears in the `goto` statement, which is `there` in this case. As I said, the label is written preceding the statement it applies to, with a colon separating the label from the rest of the statement, as in this example:

```
there: x=10;                      /* A labeled statement */
```

The `goto` statement can be used in conjunction with an `if` statement, as in the following example:

```
...
if(dice == 6)
  goto Waldorf;
else
  goto Jail;                     /* Go to the statement labeled Jail */

Waldorf:
  comfort = high;
  ...
  /* Code to prevent falling through to Jail */

Jail:                     /* The label itself. Program control is sent here */
  comfort = low;
  ...
```

You roll the dice. If you get 6, you go to the `Waldorf`; otherwise, you go to `Jail`. This might seem perfectly fine but, at the very least, it's confusing. To understand the sequence of execution, you need to hunt for the destination labels. Imagine your code was littered with `goto`s. It would be very difficult to follow and perhaps even more difficult to fix when things go wrong. So it's best to avoid the `goto` statement as far as possible. In theory it's always possible to avoid using the `goto` statement, but there are one or two instances in which it's a useful option. You'll look into loops in Chapter 4, but for now, know that exiting from the innermost loop of a deeply nested set of loops can be much simpler with a `goto` statement than with other mechanisms.

Bitwise Operators

Before you come to the big example for the chapter, you'll examine a group of operators that look something like the logical operators you saw earlier but in fact are quite different. These are called the **bitwise operators**, because they operate on the bits in integer values. There are six bitwise operators, as shown in Table 3-4.

Table 3-4. *Bitwise Operators*

Operator	Description
&	Bitwise AND operator
|	Bitwise OR operator
^	Bitwise Exclusive OR (EOR) operator
~	Bitwise NOT operator, also called the 1's complement operator
>>	Bitwise shift right operator
<<	Bitwise shift left operator

All of these only operate on integer types. The ~ operator is a unary operator—it applies to one operand—and the others are binary operators.

The bitwise AND operator, &, combines the corresponding bits of its operands in such a way that if both bits are 1, the resulting bit is 1; otherwise, the resulting bit is 0. Suppose you declare the following variables:

```
int x = 13;
int y = 6;
int z = x&y;                  /* AND the bits of x and y */
```

After the third statement, z will have the value 4 (binary 100). This is because the corresponding bits in x and y are combined as follows:

x	0	0	0	0	1	1	0	1
y	0	0	0	0	0	1	1	0
x&y	0	0	0	0	0	1	0	0

Obviously the variables would have more bits than I have shown here, but the additional bits would all be 0. There is only one instance where corresponding bits in the variables x and y are both 1 and that is the third bit from the right; this is the only case where the result of ANDing the bits is 1.

Caution It's important not to get the bitwise operators and the logical operators muddled. The expression x & y will produce quite different results from x && y in general. Try it out and see.

The bitwise OR operator, |, results in 1 if either or both of the corresponding bits are 1; otherwise, the result is 0. Let's look at a specific example. If you combine the same values using the | operator in a statement such as this

```
int z = x|y;                  /* OR the bits of x and y */
```

the result would be as follows:

x	0	0	0	0	1	1	0	1
y	0	0	0	0	0	1	1	0
x|y	0	0	0	0	1	1	1	1

The value stored in z would therefore be 15 (binary 1111).

The bitwise EOR operator, ^, produces a 1 if both bits are different, and 0 if they're the same. Again, using the same initial values, the statement

```
int z = x^y;                  /*Exclusive OR the bits of x and y */
```

would result in z containing the value 11 (binary 1011), because the bits combine as follows:

x	0	0	0	0	1	1	0	1
y	0	0	0	0	0	1	1	0
x^y	0	0	0	0	1	0	1	1

The unary operator, ~, flips the bits of its operand, so 1 becomes 0, and 0 becomes 1. If you apply this operator to x with the value 13 as before, and you write

```
int z = ~x;                             /* Store 1's complement of x */
```

then z will have the value 14. The bits are set as follows:

x	0	0	0	0	1	1	0	1
~x	1	1	1	1	0	0	1	0

The value 11110010 is 14 in 2's complement representation of negative integers. If you're not familiar with the 2's complement form, and you want to find out about it, it is described in Appendix A.

The `shift` operators shift the bits in the left operand by the number of positions specified by the right operand. You could specify a shift-left operation with the following statements:

```
int value = 12;
int shiftcount = 3;                   /* Number of positions to be shifted */
int result = value << shiftcount;     /* Shift left shiftcount positions   */
```

The variable `result` will contain the value 96. The binary number in `value` is 00001100. The bits are shifted to the left three positions, and 0s are introduced on the right, so the value of `value << shiftcount`, as a binary number, will be 01100000.

The right `shift` operator moves the bits to the right, but it's a little more complicated than left shift. For `unsigned` values, the bits that are introduced on the left (in the vacated positions as the bits are shifted right) are filled with zeros. Let's see how this works in practice. Suppose you declare a variable:

```
unsigned int value = 65372U;
```

As a binary value in a 2-byte variable, this is:

Suppose you now execute the following statement:

1111 1111 0101 1100

```
unsigned int result = value >> 2;     /* Shift right two bits */
```

The bits in `value` will be shifted two places to the right, introducing zeros at the left end, and the resultant value will be stored in `result`. In binary this will be 0, which is the decimal value 16343.

0011 1111 1101 0111

For signed values that are negative, where the leftmost bit will be 1, the result depends on your system. In most cases, the sign bit is propagated, so the bits introduced on the right are 1 bits, but on some systems zeros are introduced in this case too. Let's see how this affects the result.

Suppose you define a variable with this statement:

```
int new_value = -164;
```

This happens to be the same bit pattern as the unsigned value that you used earlier; remember that this is the 2's complement representation of the value:

1111 1111 0101 1100

Suppose you now execute this statement:

```
int new_result = new_value >> 2;      /* Shift right two bits */
```

This will shift the value in `new_value` two bit positions to the right and the result will be stored in `new_result`. If, as is usually the case, the sign bit is propagated, 1s will be inserted on the left as the bits are shifted to the right, so `new_result` will end up as

1111 1111 1101 0111

This is the decimal value –41, which is what you might expect because it amounts to –164/4. If the sign bit isn't propagated, however, as can occur on some computers, the value in `new_result` will be

0011 1111 1101 0111

So shifting right two bits in this case has changed the value –164 to +16343—perhaps a rather unexpected result.

The op= Use of Bitwise Operators

You can use all of the binary bitwise operators in the `op=` form of assignment. The exception is the operator ~, which is a unary operator. As you saw in Chapter 2, a statement of the form

```
lhs op= rhs;
```

is equivalent to the statement

```
lhs = lhs op (rhs);
```

This means that if you write

```
value <<= 4;
```

the effect is to shift the contents of the integer variable, `value`, left four bit positions. It's exactly the same as the following:

```
value = value << 4;
```

You can do the same kind of thing with the other binary operators. For example, you could write the following statement:

```
value &= 0xFF;
```

where `value` is an integer variable. This is equivalent to the following:

```
value = value & 0xFF;
```

The effect of this is to keep the rightmost eight bits unchanged and to set all the others to zero.

Using Bitwise Operators

The bitwise operators look interesting in an academic kind of way, but what use are they? They don't come up in everyday programs, but in some areas they become very useful. One major use of the bitwise `AND`, `&`, and the bitwise `OR`, `|`, is in operations to test and set individual bits in an integer variable. With this capability you can use individual bits to store data that involves one of two choices. For example, you could use a single integer variable to store several characteristics of a person. You could store whether the person is male or female with one bit, and you could use three other bits to specify whether the person can speak French, German, or Italian. You might use another bit to record whether the person's salary is $50,000 or more. So in just four bits you have a substantial set of data recorded. Let's see how this would work out.

The fact that you only get a 1 bit when both of the bits being combined are 1 means that you can use the `&` operator to select a part of an integer variable or even just a single bit. You first define a value, usually called a **mask**, that you use to select the bit or bits that you want. It will contain a bit value of 1 for the bit positions you want to keep and a bit value of 0 for the bit positions you want to discard. You can then `AND` this mask with the value that you want to select from. Let's look at an example. You can define masks with the following statements:

```
unsigned int male       = 0x1;    /* Mask selecting first (rightmost) bit */
unsigned int french     = 0x2;    /* Mask selecting second bit            */
unsigned int german     = 0x4;    /* Mask selecting third bit             */
unsigned int italian    = 0x8;    /* Mask selecting fourth bit            */
unsigned int payBracket = 0x10;   /* Mask selecting fifth bit             */
```

In each case, a 1 bit will indicate that the particular condition is true. These masks in binary each pick out an individual bit, so you could have an `unsigned int` variable, `personal_data`, which would store five items of information about a person. If the first bit is 1, the person is male, and if the first bit is 0, the person is female. If the second bit is 1, the person speaks French, and if it is 0, the person doesn't speak French, and so on for all five bits at the right end of the data value.

You could therefore test the variable, `personal_data`, for a German speaker with the following statement:

```
if(personal_data & german)
  /* Do something because they speak German */
```

The expression `personalData & german` will be nonzero—that is, true—if the bit corresponding to the mask, `german`, is 1; otherwise, it will be 0.

Of course, there's nothing to prevent you from combining several expressions involving using masks to select individual bits with the logical operators. You could test whether someone is a female who speaks French or Italian with the following statement:

```
if(!(personal_data & male) && ((personal_data & french) ||
                                                    (personal_data & italian)))
  /* We have a French or Italian speaking female */
```

As you can see, it's easy enough to test individual bits or combinations of bits. The only other thing you need to understand is how to set individual bits. The `OR` operator swings into action here.

You can use the `OR` operator to set individual bits in a variable using the same mask as you use to test the bits. If you want to set the variable `personal_data` to record a person as speaking French, you can do it with this statement:

```
personal_data |= french;               /* Set second bit to 1 */
```

Just to remind you, the preceding statement is exactly the same as the following statement:

```
personal_data = personal_data|french;  /* Set second bit to 1 */
```

The second bit from the right in `personal_data` will be set to 1, and all the other bits will remain as they were. Because of the way the `|` operator works, you can set multiple bits in a single statement:

```
personal_data |= french|german|male;
```

This sets the bits to record a French- and German-speaking male. If the variable `personalData` previously recorded that the person spoke Italian, that bit would still be set, so the `OR` operator is additive. If a bit is already set, it will stay set.

What about resetting a bit? Suppose you want to change the male bit to female. This amounts to resetting a 1 bit to 0, and it requires the use of the `!` operator with the bitwise `AND`:

```
personal_data &= !male;            /* Reset male to female */
```

This works because `!male` will have a 0 bit set for the bit that indicates male and all the other bits as 1. Thus, the bit corresponding to male will be set to 0: 0 `AND`ed with anything is 0, and all the other bits will be as they were. If another bit is 1, then `1&1` will still be 1. If another bit is 0, then `0&1` will still be 0.

I've used the example of using bits to record specific items of personal data. If you want to program a PC using the Windows application programming interface (API), you'll often use individual bits to record the status of various window parameters, so the bitwise operators can be very useful in this context.

TRY IT OUT: USING BITWISE OPERATORS

Let's exercise some of the bitwise operators in a slightly different example, but using the same principles discussed previously. This example illustrates how you can use a mask to select multiple bits from a variable. You'll write a program that sets a value in a variable and then uses the bitwise operators to reverse the sequence of hexadecimal digits. Here's the code:

```
/* Program 3.10 Exercising bitwise operators */
#include <stdio.h>

int main(void)
{
  unsigned int original = 0xABC;
  unsigned int result = 0;
  unsigned int mask = 0xF;    /* Rightmost four bits                      */

  printf("\n original = %X", original);

  /* Insert first digit in result */
  result |= original&mask;    /* Put right 4 bits from original in result */

  /* Get second digit */
  original >>= 4;             /* Shift original right four positions      */
  result <<= 4;               /* Make room for next digit                 */
  result |= original&mask;    /* Put right 4 bits from original in result */

  /* Get third digit */
  original >>= 4;             /* Shift original right four positions      */
  result <<= 4;               /* Make room for next digit                 */
  result |= original&mask;    /* Put right 4 bits from original in result */
  printf("\t result = %X\n", result);
  return 0;
}
```

This will produce the following output:

```
 original = ABC  result = CBA
```

How It Works

This program uses the idea of masking, previously discussed. The rightmost hexadecimal digit in `original` is obtained by ANDing the value with `mask` in the expression `original&mask`. This sets all the other hexadecimal digits to 0. Because the value of `mask` as a binary number is

0000 0000 0000 1111

you can see that only the first four bits on the right are kept. Any of these four bits that is 1 in `original` will stay as 1 in the result, and any that are 0 will stay as 0. All the other bits will be 0 because 0 ANDed with anything is 0.

Once you've selected the rightmost four bits, you then store the result with the following statement:

```
  result |= original&mask;    /* Put right 4 bits from original in result */
```

The content of `result` is `OR`ed with the hexadecimal digit that's produced by the expression on the right side.

To get at the second digit in `original`, you need to move it to where the first digit was. You do this by shifting `original` right by four bit positions:

```
original >>= 4;             /* Shift original right four positions       */
```

The first digit is shifted out and is lost.

To make room for the next digit from `original`, you shift the contents of `result` left by four bit positions with this statement:

```
result <<= 4;               /* Make room for next digit                 */
```

Now you want to insert the second digit from `original`, which is now in the first digit position, into `result`. You do this with the following statement:

```
result |= original&mask;    /* Put right 4 bits from original in result */
```

To get the third digit you just repeat the process. Clearly, you could repeat this for as many digits as you want.

Designing a Program

You've reached the end of Chapter 3 successfully, and now you'll apply what you've learned so far to build a useful program.

The Problem

The problem is to write a simple calculator that can add, subtract, multiply, divide, and find the remainder when one number is divided by another. The program must allow the calculation that is to be performed to be keyed in a natural way, such as 5.6 * 27 or 3 + 6.

The Analysis

All the math involved is simple, but the processing of the input adds a little complexity. You need to make checks on the input to make sure that the user hasn't asked the computer to do something impossible. You must allow the user to input a calculation in one go, for example

```
34.87 + 5
```

or

```
9 * 6.5
```

The steps involved in writing this program are as follows:

1. Get the user's input for the calculation that the user wants the computer to perform.
2. Check that input to make sure that it's understandable.
3. Perform the calculation.
4. Display the result.

The Solution

This section outlines the steps you'll take to solve the problem.

Step 1

Getting the user input is quite easy. You'll be using `printf()` and `scanf()`, so you need the `<stdio.h>` header file. The only new thing I'll introduce is in the way in which you'll get the input. As I said earlier, rather than asking the user for each number individually and then asking for the operation to be performed, you'll get the user to type it in more naturally. You can do this because of the way `scanf()` works, but I'll discuss the details of that after you've seen the first part of the program. Let's kick off the program with the code to read the input:

```
/*Program 3.11 A calculator*/
#include <stdio.h>

int main(void)
{
  double number1 = 0.0;          /* First operand value a decimal number  */
  double number2 = 0.0;          /* Second operand value a decimal number */
  char operation = 0;            /* Operation - must be +, -, *, /, or %  */

  printf("\nEnter the calculation\n");
  scanf("%lf %c %lf", &number1, &operation, &number2);

  /* Plus the rest of the code for the program */
  return 0;
}
```

The `scanf()` function is fairly clever when it comes to reading data. You don't actually need to enter each input data item on a separate line. All that's required is one or more whitespace characters between each item of input. (You create a whitespace character by pressing the spacebar, the Tab key, or the Enter key.)

Step 2

Next, you must check to make sure that the input is correct. The most obvious check to perform is that the operation to be performed is valid. You've already decided that the valid operations are `+`, `-`, `/`, `*`, and `%`, so you need to check that the operation is one of these.

You also need to check the second number to see if it's `0` if the operation is either `/` or `%`. If the right operand is 0, these operations are invalid. You could do all these checks using `if` statements, but a `switch` statement provides a far better way of doing this because it is easier to understand than a sequence of `if` statements:

```
/*Program 3.11 A calculator*/
#include <stdio.h>

int main(void)
{
  double number1 = 0.0;          /* First operand value a decimal number  */
  double number2 = 0.0;          /* Second operand value a decimal number */
  char operation = 0;            /* Operation - must be +, -, *, /, or %  */
  printf("\nEnter the calculation\n");
  scanf("%lf %c %lf", &number1, &operation, &number2);

  /* Code to check the input goes here */
  switch(operation)
  {
    case '+':                    /* No checks necessary for add      */
      break;

    case '-':                    /* No checks necessary for subtract */
      break;

    case '*':                    /* No checks necessary for multiply */
      break;

    case '/':
      if(number2 == 0)          /* Check second operand for zero    */
        printf("\n\n\aDivision by zero error!\n");
      break;

    case '%':                   /* Check second operand for zero    */
      if((long)number2 == 0)
         printf("\n\n\aDivision by zero error!\n");
      break;

    default:                    /* Operation is invalid if we get to here */
      printf("\n\n\aIllegal operation!\n");
      break;
  }
  /* Plus the rest of the code for the program */
  return 0;
}
```

Because you're casting the second operand to an integer when the operator is %, it isn't sufficient to just check the second operand against 0—you must check that number2 doesn't have a value that will result in 0 when it's cast to type long.

Steps 3 and 4

So now that you've checked the input, you can calculate the result. You have a choice here. You could calculate each result in the switch and store it to be output after the switch, or you could simply output the result for each case. Let's go for the latter approach. The code you need to add is as follows:

```
/*Program 3.11 A calculator*/
#include <stdio.h>

int main(void)
{
  double number1 = 0.0;         /* First operand value a decimal number  */
  double number2 = 0.0;         /* Second operand value a decimal number */
  char operation = 0;           /* Operation - must be +, -, *, /, or %  */
  printf("\nEnter the calculation\n");
  scanf("%lf %c %lf", &number1, &operation, &number2);

  /* Code to check the input goes here */
  switch(operation)
  {
    case '+':                   /* No checks necessary for add       */
      printf("= %lf\n", number1 + number2);
      break;

    case '-':                   /* No checks necessary for subtract */
      printf("= %lf\n", number1 - number2);
      break;

    case '*':                   /* No checks necessary for multiply */
      printf("= %lf\n", number1 * number2);
      break;

    case '/':
      if(number2 == 0)          /* Check second operand for zero    */
        printf("\n\n\aDivision by zero error!\n");
      else
        printf("= %lf\n", number1 / number2);
       break;

    case '%':                   /* Check second operand for zero    */
      if((long)number2 == 0)
        printf("\n\n\aDivision by zero error!\n");
      else
        printf("= %ld\n", (long)number1 % (long)number2);
      break;

    default:                    /* Operation is invalid if we get to here */
      printf("\n\n\aIllegal operation!\n");
      break;
  }
  return 0;
}
```

Notice how you cast the two numbers from `double` to `long` when you calculate the modulus. This is because the `%` operator only works with integers in C.

All that's left is to try it out! Here's some typical output:

```
Enter the calculation
25*13
= 325.000000
```

Here's another example:

```
Enter the calculation
999/3.3
= 302.727273
```

And just one more

```
Enter the calculation
7%0


Division by zero error!
```

Summary

This chapter ends with quite a complicated example. In the first two chapters you really just looked at the groundwork for C programs. You could do some reasonably useful things, but you couldn't control the sequence of operations in the program once it had started. In this chapter you've started to feel the power of the language and how you can use data entered by the user or results calculated during execution to determine what happens next.

You've learned how to compare variables and then use `if`, `if-else`, `else-if`, and `switch` statements to affect the outcome. You also know how to use logical operators to combine comparisons between your variables. You should now understand a lot more about making decisions and taking different paths through your program code.

In the next chapter, you'll learn how to write even more powerful programs: programs that can repeat a set of statements until some condition is met. By the end of Chapter 4, you'll think your calculator is small-fry.

Exercises

The following exercises enable you to try out what you've learned in this chapter. If you get stuck, look back over the chapter for help. If you're still stuck, you can download the solutions from the Source Code/Download area of the Apress web site (`http://www.apress.com`), but that really should be a last resort.

Exercise 3-1. Write a program that will first allow a user to choose one of two options:

1. Convert a temperature from degrees Celsius to degrees Fahrenheit.
2. Convert a temperature from degrees Fahrenheit to degrees Celsius.

The program should then prompt for the temperature value to be entered and output the new value that results from the conversion. To convert from Celsius to Fahrenheit you can multiply the value by 1.8 and then add 32. To convert from Fahrenheit to Celsius, you can subtract 32 from the value, then multiply by 5, and divide the result by 9.

Exercise 3-2. Write a program that prompts the user to enter the date as three integer values for the month, the day in the month, and the year. The program should then output the date in the form 31st December 2003 when the user enters 12 31 2003, for example.

You will need to work out when *th*, *nd*, *st*, and *rd* need to be appended to the day value. Don't forget 1st, 2nd, 3rd, 4th; but 11th, 12th, 13th, 14th; and 21st, 22nd, 23rd, and 24th.

Exercise 3-3. Write a program that will calculate the price for a quantity entered from the keyboard, given that the unit price is $5 and there is a discount of 10 percent for quantities over 30 and a 15 percent discount for quantities over 50.

Exercise 3-4. Modify the last example in the chapter that implemented a calculator so that the user is given the option to enter `y` or `Y` to carry out another calculation, and `n` or `N` to end the program. (Note: You'll have to use a `goto` statement for this at this time, but you'll learn a better way of doing this in the next chapter).