The **switch** and **goto** Statements

This chapter focuses on the `switch` statement. It also improves the `if` and `else-if` constructs by streamlining the multiple-choice decisions your programs make. The `switch` statement does not replace the `if` statement, but it is better to use `switch` when your programs must perform one of many different actions.

The `switch` and `break` statements work together. Almost every `switch` statement you use includes at least one `break` statement in the body of the `switch`. To conclude this chapter—and this section of the book on C++ constructs—you learn the `goto` statement, although it is rarely used.

This chapter introduces the following:

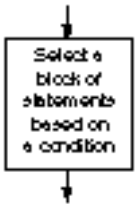
- The `switch` statement used for selection
- The `goto` statement used for branching from one part of your program to another

If you have mastered the `if` statement, you should have little trouble with the concepts presented here. By learning the `switch` statement, you should be able to write menus and multiple-choice data-entry programs with ease.

The **switch** Statement

Use the switch statement when your program makes a multiple-choice selection.

The switch statement is sometimes called the *multiple-choice statement*. The switch statement enables your program to choose from several alternatives. The format of the switch statement is a little longer than the format of other statements you have seen. Here is the switch statement:




```
switch (expression)
   { case (expression1): { one or more C++ statements; }
     case (expression2): { one or more C++ statements; }
     case (expression3): { one or more C++ statements; }
      .
      .
      .
     default: { one or more C++ statements; }
   }
```

The expression can be an integer expression, a character, a literal, or a variable. The *sub*expressions (expression1, expression2, and so on) can be any other integer expression, character, literal, or variable. The number of case expressions following the switch line is determined by your application. The one or more C++ statements is any block of C++ code. If the block is only one statement long, you do not need the braces, but they are recommended.

The default line is optional; most (but not all) switch statements include the default. The default line does not have to be the last line of the switch body.

If expression matches expression1, the statements to the right of expression1 execute. If expression matches expression2, the statements to the right of expression2 execute. If none of the expressions match the switch expression, the default case block executes. The case expression does not need parentheses, but the parentheses sometimes make the value easier to find.

TIP: Use a break statement after each case block to keep execution from "falling through" to the remaining case statements.

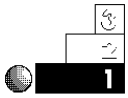
Using the switch statement is easier than its format might lead you to believe. Anywhere an if-else-if combination of statements can go, you can usually put a clearer switch statement. The switch statement is much easier to follow than an if-in-an-if-in-an-if statement, as you have had to write previously.

However, the if and else-if combinations of statements are not difficult to follow. When the relational test that determines the choice is complex and contains many && and || operators, the if statement might be a better candidate. The switch statement is preferred whenever multiple-choice possibilities are based on a single literal, variable, or expression.

TIP: Arrange case statements in the most-often to least-often executed order to improve your program's speed.

The following examples clarify the switch statement. They compare the switch statement to if statements to help you see the difference.

Examples



1. Suppose you are writing a program to teach your child how to count. Your program will ask the child for a number. It then beeps (rings the computer's alarm bell) as many times as necessary to match that number.

   The following program assumes the child presses a number key from 1 to 5. This program uses the if-else-if combination to accomplish this counting-and-beeping teaching method.

   Identify the program and include the necessary header file. You want to sound a beep and move the cursor to the next line, so define a global variable called BEEP *that does this. You need a variable to hold the user's answer, so make* num *an integer variable.*

   Ask the user for a number. Assign the user's number to num*. If* num *is 1, call* BEEP *once. If* num *is 2, call* BEEP *twice. If* num *is 3, call* BEEP *three times. If* num *is 4, call* BEEP *four times. If* num *is 5, call* BEEP *five times.*

```
// Filename: C15BEEP1.CPP
// Beeps a designated number of times.
#include <iostream.h>

// Define a beep cout to save repeating printf()s
// throughout the program.
#define BEEP cout << "\a \n"
main()
{
   int num;

   // Request a number from the child
   // (you might have to help).
   cout << "Please enter a number ";
   cin >> num;

   // Use multiple if statements to beep.
   if (num == 1)
     { BEEP; }
   else if (num == 2)
          { BEEP; BEEP; }
        else if (num == 3)
                 { BEEP; BEEP; BEEP; }
               else if (num == 4)
                      { BEEP; BEEP; BEEP; BEEP; }
                    else if (num == 5)
                           { BEEP; BEEP; BEEP; BEEP; BEEP; }
   return 0;
}
```

No beeps are sounded if the child enters something other than 1 through 5. This program takes advantage of the `#define` preprocessor directive to define a shortcut to an alarm `cout` function. In this case, the `BEEP` is a little clearer to read, as long as you remember that `BEEP` is not a command, but is replaced with the `cout` everywhere it appears.

One drawback to this type of `if`-in-an-`if` program is its readability. By the time you indent the body of each `if` and `else`, the program is too far to the right. There is no room for more than five or six possibilities. More importantly, this

type of logic is difficult to follow. Because it involves a multiple-choice selection, a `switch` statement is much better to use, as you can see with the following, improved version.


```
// Filename: C15BEEP2.CPP
// Beeps a certain number of times using a switch.
#include <iostream.h>

// Define a beep cout to save repeating couts
// throughout the program.
#define BEEP cout << "\a \n"
main()
{
   int num;

   // Request from the child (you might have to help).
   cout << "Please enter a number ";
   cin >> num;

   switch (num)
   { case (1): { BEEP;
                 break; }
     case (2): { BEEP; BEEP;
                 break; }
     case (3): { BEEP; BEEP; BEEP;
                 break; }
     case (4): { BEEP; BEEP; BEEP; BEEP;
                 break; }
     case (5): { BEEP; BEEP; BEEP; BEEP; BEEP;
                 break; }
   }
   return 0;
}
```

This example is much clearer than the previous one. The value of `num` controls the execution—only the `case` that matches `num` executes. The indention helps separate each `case`.

If the child enters a number other than 1 through 5, no beeps are sounded because there is no `case` expression to match any other value and there is no default `case`.

Because the BEEP preprocessor directive is so short, you can put more than one on a single line. This is not a requirement, however. The block of statements following a case can also be more than one statement long.

If more than one case expression is the same, only the first expression executes.

2. If the child does not enter a 1, 2, 3, 4, or 5, nothing happens in the previous program. What follows is the same program modified to take advantage of the default option. The default block of statements executes if none of the previous cases match.

```
// Filename: C15BEEP3.CPP
// Beeps a designated number of times using a switch.
#include <iostream.h>

// Define a beep cout to save repeating couts
// throughout the program.
#define BEEP cout << "\a \n"
main()
{
   int num;

 // Request a number from the child (you might have to help).
   cout << "Please enter a number ";
   cin >> num;

   switch (num)
   { case (1): { BEEP;
                 break; }
     case (2): { BEEP; BEEP;
                 break; }
     case (3): { BEEP; BEEP; BEEP;
                 break; }
     case (4): { BEEP; BEEP; BEEP; BEEP;
                 break; }
     case (5): { BEEP; BEEP; BEEP; BEEP; BEEP;
                 break; }
     default:  { cout << "You must enter a number from " <<
                         "1 to 5\n";
```

```
                cout << "Please run this program again\n";
                break; }
   }
   return 0;
}
```

The `break` at the end of the `default` case might seem redundant. After all, no other `case` statements execute by "falling through" from the default case. It is a good habit to put a `break` after the `default` case anyway. If you move the `default` higher in the `switch` (it doesn't have to be the last `switch` option), you are more inclined to move the `break` with it (where it is then needed).

3. To show the importance of using `break` statements in each `case` expression, here is the same beeping program without any `break` statements.

```
// Filename: C15BEEP4.CPP
// Incorrectly beeps using a switch.
#include <iostream.h>

// Define a beep printf() to save repeating couts
// throughout the program.
#define BEEP cout << "\a \n"
main()
{
   int num;

   // Request a number from the child
   // (you might have to help).
   cout << "Please enter a number ";
   cin >> num;

   switch (num)                          // Warning!
   { case (1): { BEEP; }     // Without a break, this code
     case (2): { BEEP; BEEP; }     // falls through to the
     case (3): { BEEP; BEEP; BEEP; } // rest of the beeps!
     case (4): { BEEP; BEEP; BEEP; BEEP; }
     case (5): { BEEP; BEEP; BEEP; BEEP; BEEP; }
     default:  { cout << "You must enter a number " <<
                         "from 1 to 5\n";
```

```
                    cout << "Please run this program again\n"; }
    }
    return 0;
}
```

If the user enters a 1, the program beeps 15 times! The `break` is not there to stop the execution from falling through to the other cases. Unlike other programming languages such as Pascal, C++'s `switch` statement requires that you insert `break` statements between each `case` if you want only one `case` executed. This is not necessarily a drawback. The trade-off of having to specify `break` statements gives you more control in how you handle specific `case`s, as shown in the next example.

4. This program controls the printing of end-of-day sales totals. It first asks for the day of the week. If the day is Monday through Thursday, a daily total is printed. If the day is a Friday, a weekly total and a daily total are printed. If the day happens to be the end of the month, a monthly sales total is printed as well.

   In a real application, these totals would come from the disk drive rather than be assigned at the top of the program. Also, rather than individual sales figures being printed, a full daily, weekly, and monthly report of many sales totals would probably be printed. You are on your way to learning more about expanding the power of your C++ programs. For now, concentrate on the `switch` statement and its possibilities.

   Each type of report for sales figures is handled through a hierarchy of `case` statements. Because the daily amount is the last `case`, it is the only report printed if the day of the week is Monday through Thursday. If the day of the week is Friday, the second `case` prints the weekly sales total and then falls through to the daily total (because Friday's daily total must be printed as well). If it is the end of the month, the first `case` executes, falling through to the weekly total, then to the daily sales total as well. Other languages that do not offer this "fall through" flexibility are more limiting.

```
// Filename: C15SALE.CPP
// Prints daily, weekly, and monthly sales totals.
#include <iostream.h>
#include <stdio.h>

main()
{
   float daily=2343.34;    // Later, these figures
   float weekly=13432.65;  // come from a disk file
   float monthly=43468.97; // instead of being assigned
                           // as they are here.
   char ans;
   int day;        // Day value to trigger correct case.

   // Month is assigned 1 through 5 (for Monday through
   // Friday) or 6 if it is the end of the month. Assume
   // a weekly and a daily prints if it is the end of the
   // month, no matter what the day is.
   cout << "Is this the end of the month? (Y/N) ";
   cin >> ans;
   if ((ans=='Y') || (ans=='y'))
     { day=6; }                          // Month value
   else
     { cout << "What day number, 1 through 5 (for Mon-Fri)" <<
              " is it? ";
       cin >> day; }

   switch (day)
     { case (6): printf("The monthly total is %.2f \n",
                        monthly);
       case (5): printf("The weekly total is %.2f \n",
                        weekly);
       default:  printf("The daily total is %.2f \n", daily);
     }
   return 0;
}
```

5. The order of the `case` statements is not fixed. You can rearrange the statements to make them more efficient. If only one or two cases are being selected most of the time, put those cases near the top of the `switch` statement.

For example, in the previous program, most of the company's reports are daily, but the daily option is third in the case statements. By rearranging the case statements so the daily report is at the top, you can speed up this program because C++ does not have to scan two case expressions that it rarely executes.

```
// Filename: C15DEPT1.CPP
// Prints message depending on the department entered.
#include <iostream.h>
main()
{
   char choice;

   do   // Display menu and ensure that user enters a
        // correct option.
     { cout << "\nChoose your department: \n";
       cout << "S - Sales \n";
       cout << "A - Accounting \n";
       cout << "E - Engineering \n";
       cout << "P - Payroll \n";
       cout << "What is your choice? ";
       cin >> choice;
       // Convert choice to uppercase (if they
       // entered lowercase) with the ASCII table.
       if ((choice>=97) && (choice<=122))
         { choice -= 32; }        // Subtract enough to make
                                  // uppercase.
     } while ((choice!='S')&&(choice!='A')&&
              (choice!='E')&&(choice!='P'));

   // Put Engineering first because it occurs most often.
   switch (choice)
   { case ('E') : { cout << "\n Your meeting is at 2:30";
                    break; }
     case ('S') : { cout << "\n Your meeting is at 8:30";
                    break; }
     case ('A') : { cout << "\n Your meeting is at 10:00";
                    break; }
     case ('P') : { cout << "\n Your meeting has been " <<
                            "canceled";
```

```
                    break; }
    }
    return 0;
}
```

The goto Statement

Early programming languages did not offer the flexible constructs that C++ gives you, such as `for` loops, `while` loops, and `switch` statements. Their only means of looping and comparing was with the `goto` statement. C++ still includes a `goto`, but the other constructs are more powerful, flexible, and easier to follow in a program.

The `goto` causes execution to jump to some statement other than the next one.

The `goto` statement causes your program to jump to a different location, rather than execute the next statement in sequence. The format of the `goto` statement is

```
goto statement label
```

A `statement label` is named just as variables are (see Chapter 4, "Variables and Literals"). A `statement label` cannot have the same name as a C++ command, a C++ function, or another variable in the program. If you use a `goto` statement, there must be a `statement label` elsewhere in the program to which the `goto` branches. Execution then continues at the statement with the `statement label`.

The `statement label` precedes a line of code. Follow all `statement labels` with a colon (:) so C++ recognizes them as labels, not variables. You have not seen statement labels in the C++ programs so far in this book because none of the programs needed them. A `statement label` is optional unless you have a `goto` statement.

The following four lines of code each has a different `statement label`. This is not a program, but individual lines that might be included in a program. Notice that the `statement labels` are on the left.

```
pay: cout << "Place checks in the printer \n";

Again: cin >> name;

EndIt: cout << "That is all the processing. \n";

CALC: amount = (total / .5) * 1.15;
```

The statement labels are not intended to replace comments, although their names reflect the code that follows. Statement labels give `goto` statements a tag *to go to*. When your program finds the `goto`, it branches to the statement labeled by the `statement label`. The program then continues to execute sequentially until the next `goto` changes the order again (or until the program ends).

TIP: Use identifying line labels. A repetitive calculation deserves a label such as `CalcIt` and not `x15z`. Even though both are allowed, the first one is a better indication of the code's purpose.

Use goto Judiciously

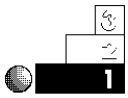
The `goto` is not considered a good programming statement when overused. There is a tendency, especially for beginning programmers, to include too many `goto` statements in a program. When a program branches all over the place, it becomes difficult to follow. Some people call programs with many `goto` statements "spaghetti code."

To eliminate `goto` statements and write better structured programs, use the other looping and `switch` constructs seen in the previous few chapters.

The `goto` is not necessarily a bad statement—if used judiciously. Starting with the next chapter, you begin to break your programs into smaller modules called functions, and the `goto` becomes less and less important as you write more and more functions.

For now, become familiar with `goto` so you can understand programs that use it. Some day, you might have to correct the code of someone who used the `goto`.

Examples



1. The following program has a problem that is a direct result of the `goto`, but it is still one of the best illustrations of the `goto` statement. The program consists of an *endless loop* (or an *infinite* loop). The first three lines (after the opening brace) execute, then the `goto` in the fourth line causes execution to loop back to the beginning and repeat the first three lines. The `goto` continues to do this until you press Ctrl-Break or ask your system administrator to cancel the program.

Identify the program and include the input/output header file. You want to print a message, but split it over three lines. You want the message to keep repeating, so label the first line, then use a `goto` *to jump back to that line.*

```
// Filename: C15GOTO1.CPP
// Program to show use of goto. This program ends
// only when the user presses Ctrl-Break.
#include <iostream.h>
main()
{
   Again: cout << "This message \n";
   cout << "\t keeps repeating \n";
   cout << "\t\t over and over \n";

   goto Again;   // Repeat continuously.

   return 0;
}
```

Notice the `statement label` (`Again` in the previous example) has a colon to separate it from the rest of the line, but there is not a colon with the label at the `goto` statement. Here is the result of running this program.

```
This message
         keeps repeating
                  over and over
This message
         keeps repeating
                  over and over
```

```
This message
         keeps repeating
                  over and over
This message
         keeps repeating
                  over and over
This message
         keeps repeating
                  over and over
This message
         keeps repeating
                  over and over
This message
         keeps repeating
                  over and over
This message
```

2. It is sometimes easier to read your program's code when you write the statement labels on separate lines. Remember that writing maintainable programs is the goal of every good programmer. Making your programs easier to read is a prime consideration when you write them. The following program is the same repeating program shown in the previous example, except the statement label is placed on a separate line.

```
// Filename: C15GOTO2.CPP
// Program to show use of goto. This program ends
// only when the user presses Ctrl-Break.
#include <iostream.h>
main()
{

Again:
   cout << "This message \n";
   cout << "\t keeps repeating \n";
   cout << "\t\t over and over \n";

   goto Again;   // Repeat continuously

   return 0;
}
```

The line following the statement label is the one that executes next, after control is passed (by the `goto`) to the label.

Of course, these are silly examples. You probably don't want to write programs with infinite loops. The `goto` is a statement best preceded with an `if`; this way the `goto` eventually stops branching without intervention from the user.

3. The following program is one of the worst-written programs ever! It is the epitome of spaghetti code! However, do your best to follow it and understand its output. By understanding the flow of this output, you can hone your understanding of the `goto`. You might also appreciate the fact that the rest of this book uses the `goto` only when needed to make the program clearer.

```
// Filename: C15GOTO3.CPP
// This program demonstrates the overuse of goto.
#include <iostream.h>
main()
{
   goto Here;

   First:
   cout << "A \n";
   goto Final;

   There:
   cout << "B \n";
   goto First;

   Here:
   cout << "C \n";
   goto There;

   Final:
   return 0;
}
```

At first glance, this program appears to print the first three letters of the alphabet, but the `goto` statements make them print in the reverse order, *C, B, A*. Although the program is

not a well-designed program, some indention of the lines without statement labels make it a little more readable. This enables you to quickly separate the statement labels from the remaining code, as you can see from the following program.

```
// Filename: C15GOTO4.CPP
// This program demonstrates the overuse of goto.
#include <iostream.h>
main()
{
   goto Here;

First:
   cout << "A \n";
   goto Final;

There:
   cout << "B \n";
   goto First;

Here:
   cout << "C \n";
   goto There;

Final:
   return 0;
}
```

This program's listing is slightly easier to follow than the previous one, even though both do the same thing. The remaining programs in this book with statement labels also use such indention.

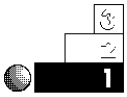
You certainly realize that this output is better produced by the following three lines.

```
cout << "C \n";
cout << "B \n";
cout << "A \n";
```

The `goto` warning is worth repeating: Use `goto` sparingly and only when its use makes your program more readable and maintainable. Usually, you can use much better commands.

Review Questions

The answers to the review questions are in Appendix B.



1. How does `goto` change the order in which a program normally executes?
2. What statement can substitute for an `if-else-if` construct?
3. Which statement almost always ends each `case` statement in a `switch`?

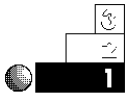
4. True or false: The order of your `case` statements has no bearing on the efficiency of your program.
5. Rewrite the following section of code using a `switch` statement.

```
if (num == 1)
   { cout << "Alpha"; }
else if (num == 2)
       { cout << "Beta"; }
     else if (num == 3)
            { cout << "Gamma"; }
          else
            { cout << "Other"; }
```

6. Rewrite the following program using a `do-while` loop.

```
Ask:
   cout << "What is your first name? ";
   cin >> name;
   if ((name[0] < 'A') || (name[0] > 'Z'))
      { goto Ask; }    // Keep asking until the user
                       // enters a valid letter.
```

Review Exercises



1. Write a program using the `switch` statement that asks users for their age, then prints a message saying "You can vote!" if they are 18, "You can adopt!" if they are 21, or "Are you really that young?" for any other age.

2. Write a menu-driven program for your local TV cable company. Here is how to assess charges: If you are within 20 miles outside the city limits, you pay $12.00 per month; 21 to 30 miles outside the city limits, you pay $23.00 per month; 31 to 50 miles outside the city limits, you pay $34.00. No one outside 50 miles receives the service. Prompt the users with a menu for their residence's distance from the city limits.

3. Write a program that calculates parking fees for a multilevel parking garage. Ask whether the driver is in a car or a truck. Charge the driver $2.00 for the first hour, $3.00 for the second, and $5.00 for more than 2 hours. If it is a truck, add $1.00 to the total fee. (*Hint:* Use one `switch` and one `if` statement.)

4. Modify the previous parking problem so the charge depends on the time of day the vehicle is parked. If the vehicle is parked before 8 a.m., charge the fees in Exercise 3. If the vehicle is parked after 8 a.m. and before 5 p.m., charge an extra usage fee of 50 cents. If the vehicle is parked after 5 p.m., deduct 50 cents from the computed price. You must prompt users for the starting time in a menu, as follows.

```
1. Before 8 a.m.
2. Before 5 p.m.
3. After 5 p.m.
```

Summary

You now have seen the `switch` statement and its options. With it, you can improve the readability of a complicated `if-else-if` selection. The `switch` is especially good when several outcomes are possible, based on the user's choice.

The `goto` statement causes an unconditional branch, and can be difficult to follow at times. The `goto` statement is not used much now, and you can almost always use a better construct. However, you should be acquainted with as much C++ as possible in case you have to work on programs others have written.

This ends the section on program control. The next section introduces user-written functions. So far, you have been using C++'s built-in functions, such as `strcpy()` and `printf()`. Now it's time to write your own.