

Module3

Program Control Statements

Table of Contents

CRITICAL SKILL 3.1: The if Statement	2
CRITICAL SKILL 3.2: The switch Statement.....	7
CRITICAL SKILL 3.3: The for Loop.....	13
CRITICAL SKILL 3.4: The while Loop	19
CRITICAL SKILL 3.5: The do-while Loop	21
CRITICAL SKILL 3.6: Using break to Exit a Loop	27
CRITICAL SKILL 3.7: Using continue.....	29
CRITICAL SKILL 3.8: Nested Loops	34
CRITICAL SKILL 3.9: Using the goto Statement	35

This module discusses the statements that control a program's flow of execution. There are three categories of : selection statements, which include the if and the switch; iteration statements, which include the for, while, and do-while loops; and jump statements, which include break, continue, return, and goto. Except for return, which is discussed later in this book, the remaining control statements, including the if and for statements to which you have already had a brief introduction, are examined here.

CRITICAL SKILL 3.1: The if Statement

Module 1 introduced the if statement. Now it is time to examine it in detail. The complete form of the if statement is

```
if(expression) statement;  
else statement;
```

where the targets of the if and else are single statements. The else clause is optional. The targets of both the if and else can also be blocks of statements. The general form of the if using blocks of statements is

```
if(expression) {  
    statement sequence  
}  
else {  
    statement sequence  
}
```

If the conditional expression is true, the target of the if will be executed; otherwise, the target of the else, if it exists, will be executed. At no time will both be executed. The conditional expression controlling the if may be any type of valid C++ expression that produces a true or false result.

The following program demonstrates the if by playing a simple version of the “guess the magic number” game. The program generates a random number, prompts for your guess, and prints the message **** Right **** if you guess the magic number. This program also introduces another C++ library function, called `rand()`, which returns a randomly selected integer value. It requires the `<cstdlib>` header.

```
// Magic Number program.  
  
#include <iostream>  
#include <cstdlib>  
using namespace std;  
  
int main()  
{  
    int magic; // magic number  
    int guess; // user's guess  
  
    magic = rand(); // get a random number  
  
    cout << "Enter your guess: ";  
    cin >> guess;  
  
    if(guess == magic) cout << "*** Right ***";  
    return 0;  
}
```

← If the guess matches the “magic number,” the message is displayed.

This program uses the 'if' statement to determine whether the user's guess matches the magic number. If it does, the message is printed on the screen. Taking the Magic Number program further, the next version uses the else to print a message when the wrong number is picked:

```
// Magic Number program: 1st improvement.

#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
    int magic; // magic number
    int guess; // user's guess

    magic = rand(); // get a random number

    cout << "Enter your guess: ";
    cin >> guess;

    if(guess == magic) cout << "*** Right ***";
    else cout << "...Sorry, you're wrong."; ← Indicate a wrong answer, too.

    return 0;
}
```

The Conditional Expression

Sometimes newcomers to C++ are confused by the fact that any valid C++ expression can be used to control the if. That is, the conditional expression need not be restricted to only those involving the relational and logical operators, or to operands of type bool. All that is required is that the controlling expression evaluate to either a true or false result. As you should recall from the previous module, a value of 0 is automatically converted into false, and all non-zero values are converted to true. Thus, any expression that results in a 0 or non-zero value can be used to control the if. For example, this program reads two integers from the keyboard and displays the quotient. To avoid a divide-by-zero error, an if statement, controlled by the second.

```
// Use an int value to control the if.
```

```
#include <iostream>
using namespace std;

int main()
{
    int a, b;
    cout << "Enter numerator: ";
    cin >> a;
    cout << "Enter denominator: ";
    cin >> b;
```

```

if(b) cout << "Result: " << a / b << '\n';
else cout << "Cannot divide by zero.\n";

return 0;
}

```

← Notice that **b** alone is sufficient to control this **if** statement. No relational operator is needed.

Here are two sample runs:

```

Enter numerator: 12
Enter denominator: 2
Result: 6

```

```

Enter numerator: 12
Enter denominator: 0
Cannot divide by zero.

```

Notice that **b** (the divisor) is tested for zero using `if(b)`. This approach works because when **b** is zero, the condition controlling the `if` is false and the `else` executes. Otherwise, the condition is true (non-zero) and the division takes place. It is not necessary (and would be considered bad style by many C++ programmers) to write this `if` as shown here:

```

if(b == 0) cout << a/Artifact << '\n';

```

This form of the statement is redundant and potentially inefficient.

Nested ifs

A nested `if` is an `if` statement that is the target of another `if` or `else`. Nested `ifs` are very common in programming. The main thing to remember about nested `ifs` in C++ is that an `else` statement always refers to the nearest `if` statement that is within the same block as the `else` and not already associated with an `else`. Here is an example:

```

if(i) {
    if(j) result = 1;
    if(k) result = 2;
    else result = 3; // this else is associated with if(k)
}
else result = 4; // this else is associated with if(i)

```

As the comments indicate, the final `else` is not associated with `if(j)` (even though it is the closest `if` without an `else`), because it is not in the same block. Rather, the final `else` is associated with `if(i)`. The inner `else` is associated with `if(k)` because that is the nearest `if`.

You can use a nested `if` to add a further improvement to the Magic Number program. This addition provides the player with feedback about a wrong guess.

```

// Magic Number program: 2nd improvement.

#include <iostream>
#include <cstdlib>

```

```

using namespace std;

int main()
{
    int magic; // magic number
    int guess; // user's guess

    magic = rand(); // get a random number

    cout << "Enter your guess: ";
    cin >> guess;
    if (guess == magic) {
        cout << "** Right **\n";
        cout << magic << " is the magic number.\n";
    }
    else {
        cout << "...Sorry, you're wrong.";
        if(guess > magic) cout <<" Your guess is too high.\n";
        else cout << " Your guess is too low.\n";
    }

    return 0;
}

```

Here, a nested if provides feedback to the player.

The if-else-if Ladder

A common programming construct that is based upon nested ifs is the if-else-if ladder, also referred to as the if-else-if staircase. It looks like this:

```

if(condition)
    statement;
else if(condition)
    statement;
else if(condition)
    statement;
.
.
.
else
    statement;

```

The conditional expressions are evaluated from the top downward. As soon as a true condition is found, the statement associated with it is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed. The final else often acts as a default condition; that is, if all other conditional tests fail, then the last else statement is performed. If there is no final else and all other conditions are false, then no action will take place.

The following program demonstrates the if-else-if ladder:

```
// Demonstrate an if-else-if ladder.
#include <iostream>
using namespace std;

int main()
{
    int x;

    for(x=0; x<6; x++) {
        if(x==1) cout << "x is one\n";
        else if(x==2) cout << "x is two\n";
        else if(x==3) cout << "x is three\n";
        else if(x==4) cout << "x is four\n";
        else cout << "x is not between 1 and 4\n";
    }

    return 0;
}
```

— An if-else-if ladder

The program produces the following output:

```
x is not between 1 and 4
x is one
x is two
x is three
x is four
x is not between 1 and 4
```

As you can see, the default else is executed only if none of the preceding if statements succeeds.



Progress Check

1. The condition controlling the if must use a relational operator. True or false?
2. To what if does an else always associate?
3. What is an if-else-if ladder?

Answer Key:

1. The condition controlling the if must use a relational operator. True or false?
2. To what if does an else always associate?
3. What is an if-else-if ladder?

CRITICAL SKILL 3.2: The switch Statement

The second of C++'s selection statements is the switch. The switch provides for a multiway branch. Thus, it enables a program to select among several alternatives. Although a series of nested if statements can perform multiway tests, for many situations the switch is a more efficient approach. It works like this: the value of an expression is successively tested against a list of constants. When a match is found, the statement sequence associated with that match is executed. The general form of the switch statement is

```
switch(expression) {  
    case constant1:  
        statement sequence  
        break;  
    case constant2:  
        statement sequence  
        break;  
    case constant3:  
        statement sequence  
        break;  
    .  
    .  
    .  
    default:  
        statement sequence  
}
```

The switch expression must evaluate to either a character or an integer value. (Floatingpoint expressions, for example, are not allowed.) Frequently, the expression controlling the switch is simply a variable. The case constants must be integer or character literals.

The default statement sequence is performed if no matches are found. The default is optional; if it is not present, no action takes place if all matches fail. When a match is found, the statements associated with that case are executed until the break is encountered or, in a concluding case or default statement, until the end of the switch is reached.

There are four important things to know about the switch statement:

The switch differs from the if in that switch can test only for equality (that is, for matches between the switch expression and the case constants), whereas the if conditional expression can be of any type.

No two case constants in the same switch can have identical values. Of course, a switch statement enclosed by an outer switch may have case constants that are the same.

A switch statement is usually more efficient than nested ifs.

The statement sequences associated with each case are not blocks. However, the entire switch statement does define a block. The importance of this will become apparent as you learn more about C++.


The following program demonstrates the switch. It asks for a number between 1 and 3, inclusive. It then displays a proverb linked to that number. Any other number causes an error message to be displayed.

```
/*
   A simple proverb generator that
   demonstrates the switch.
*/

#include <iostream>
using namespace std;

int main()
{
    int num;

    cout << "Enter a number from 1 to 3: ";
    cin >> num;

    switch(num) {  The value of num determines the case sequence executed.
        case 1:
            cout << "A rolling stone gathers no moss.\n";
            break;
        case 2:
            cout << "A bird in hand is worth two in the bush.\n";
            break;
        case 3:
            cout << "A fool and his money are soon parted.\n";
            break;
        default:
            cout << "You must enter either 1, 2, or 3.\n";
    }

    return 0;
}
```

Here are two sample runs:

```
Enter a number from 1 to 3: 1
A rolling stone gathers no moss.
```

```
Enter a number from 1 to 3: 5
You must enter either 1, 2, or 3.
```

Technically, the break statement is optional, although most applications of the switch will use it. When encountered within the statement sequence of a case, the break statement causes program flow to exit from the entire switch statement and resume at the next statement outside the switch. However, if a

break statement does not end the statement sequence associated with a case, then all the statements at and below the matching case will be executed until a break (or the end of the switch) is encountered. For example, study the following program carefully. Can you figure out what it will display on the screen?


```
// A switch without break statements.

#include <iostream>
using namespace std;

int main()
{
    int i;

    for(i=0; i<5; i++) {
        switch(i) {
            case 0: cout << "less than 1\n";
            case 1: cout << "less than 2\n";
            case 2: cout << "less than 3\n";
            case 3: cout << "less than 4\n";
            case 4: cout << "less than 5\n";
        }
        cout << '\n';
    }

    return 0;
}
```



This program displays the following output:

```
less than 1
less than 2
less than 3
less than 4
less than 5

less than 2
less than 3
less than 4
less than 5

less than 3
less than 4
less than 5

less than 4
less than 5

less than 5
```

As this program illustrates, execution will continue into the next case if no break statement is present. You can have empty cases, as shown in this example:

```
switch(i) {
    case 1:
    case 2: ← Empty case sequences
    case 3:
        cout << "i is less than 4";
        break;
    case 4:
        cout << "i is 4";
        break;
```

In this fragment, if i has the value 1, 2, or 3, then the message

i is less than 4

is displayed. If it is 4, then

i is 4

is displayed. The “stacking” of cases, as shown in this example, is very common when several cases share common code.

Nested switch Statements

It is possible to have a switch as part of the statement sequence of an outer switch. Even if the case constants of the inner and outer switch contain common values, no conflicts will arise. For example, the following code fragment is perfectly acceptable:

```
switch(ch1) {
    case 'A': cout << "This A is part of outer switch";
        switch(ch2) { ← A nested switch
            case 'A':
                cout << "This A is part of inner switch";
                break;
            case 'B': // ...
        }
        break;
    case 'B': // ...
```



1. The expression controlling the switch must be of what type?
2. When the switch expression matches a case constant, what happens?
3. When a case sequence is not terminated by a break, what happens?

Answer Key:

Q: Under what conditions should I use an if-else-if ladder rather than a switch when coding a multiway branch?

A: In general, use an if-else-if ladder when the conditions controlling the selection process do not rely upon a single value. For example, consider the following if-else-if sequence:

```
if(x < 10) // ... else if(y > 0) // ... else if(!done) // ...
```

This sequence cannot be recoded into a switch because all three conditions involve different variables—and differing types. What variable would control the switch? Also, you will need to use an if-else-if ladder when testing floating-point values or other objects that are not of types valid for use in a switch expression.

Project 3-1 Start Building a C++ Help System

This project builds a simple help system that displays the syntax for the C++ control

Help.cpp

statements. The program displays a menu containing the control statements and then waits for you to choose one. After one is chosen, the syntax of the statement is displayed. In this first version of the program, help is available for only the if and switch statements. The other control statements are added by subsequent projects.

Step by Step

1. Create a file called Help.cpp.
2. The program begins by displaying the following menu:

Help on:

1. if

2. switch Choose one:

To accomplish this, you will use the statement sequence shown here:

```
cout << "Help on:\n"; cout << " 1. if\n"; cout << " 2. switch\n"; cout << "Choose one: ";
```

3. Next, the program obtains the user's selection, as shown here:

```
cin >> choice;
```

4. Once the selection has been obtained, the program uses this switch statement to display the syntax for the selected statement:

```
switch(choice) {
    case '1':
        cout << "The if:\n\n";
        cout << "if(condition) statement;\n";
        cout << "else statement;\n";
        break;
    case '2':
        cout << "The switch:\n\n";
        cout << "switch(expression) {\n";
        cout << "    case constant:\n";
        cout << "        statement sequence\n";
        cout << "        break;\n";
        cout << "    // ... \n";
        cout << "}\n";
        break;
    default:
        cout << "Selection not found.\n";
}
```

Notice how the default clause catches invalid choices. For example, if the user enters 3, no case constants will match, causing the default sequence to execute.

5. Here is the entire Help.cpp program listing:

```
/*
    Project 3-1

    A simple help system.
*/

#include <iostream>
using namespace std;

int main() {
    char choice;

    cout << "Help on:\n";
    cout << " 1. if\n";
    cout << " 2. switch\n";
    cout << "Choose one: ";
    cin >> choice;
```

```

cout << "\n";

switch(choice) {
    case '1':
        cout << "The if:\n\n";
        cout << "if(condition) statement;\n";
        cout << "else statement;\n";
        break;
    case '2':
        cout << "The switch:\n\n";
        cout << "switch(expression) {\n";
        cout << "    case constant:\n";
        cout << "        statement sequence\n";
        cout << "        break;\n";
        cout << "    // ... \n";
        cout << "}\n";
        break;
    default:
        cout << "Selection not found.\n";
}

return 0;
}

```

Here is a sample run:

```

Help on:
  1. if
  2. switch
Choose one: 1

The if:

if(condition) statement;
else statement;

```

CRITICAL SKILL 3.3: The for Loop

You have been using a simple form of the for loop since Module 1. You might be surprised at just how powerful and flexible the for loop is. Let's begin by reviewing the basics, starting with the most traditional forms of the for.

The general form of the for loop for repeating a single statement is

```
for(initialization; expression; increment) statement;
```

For repeating a block, the general form is

```
for(initialization; expression; increment) {
    statement sequence
}
```

The initialization is usually an assignment statement that sets the initial value of the loop control variable, which acts as the counter that controls the loop. The expression is a conditional expression that determines whether the loop will repeat. The increment defines the amount by which the loop control variable will change each time the loop is repeated. Notice that these three major sections of the loop must be separated by semicolons. The for loop will continue to execute as long as the conditional expression tests true. Once the condition becomes false, the loop will exit, and program execution will resume on the statement following the for block.

The following program uses a for loop to print the square roots of the numbers between 1 and 99. Notice that in this example, the loop control variable is called num.

```
// Show square roots of 1 to 99.

#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    int num;
    double sq_root;

    for(num=1; num < 100; num++) {
        sq_root = sqrt((double) num);
        cout << num << " " << sq_root << '\n';
    }

    return 0;
}
```

This program uses the standard function `sqrt()`. As explained in Module 2, the `sqrt()` function returns the square root of its argument. The argument must be of type `double`, and the function returns a value of type `double`. The header `<cmath>` is required.

The for loop can proceed in a positive or negative fashion, and it can increment the loop control variable by any amount. For example, the following program prints the numbers 50 to -50, in decrements of 10:

```
// A negatively running for loop.

#include <iostream>
using namespace std;

int main()
{
    int i;

    for(i=50; i >= -50; i = i-10) cout << i << ' ';
    return 0;
}
```

←
A negatively running for loop

Here is the output from the program:

```
50 40 30 20 10 0 -10 -20 -30 -40 -50
```

An important point about for loops is that the conditional expression is always tested at the top of the loop. This means that the code inside the loop may not be executed at all if the condition is false to begin with. Here is an example:

```
for(count=10; count < 5; count++)  
    cout << count; // this statement will not execute
```

This loop will never execute, because its control variable, count, is greater than 5 when the loop is first entered. This makes the conditional expression, count<5, false from the outset; thus, not even one iteration of the loop will occur.

Some Variations on the for Loop

The for is one of the most versatile statements in the C++ language because it allows a wide range of variations. For example, multiple loop control variables can be used. Consider the following fragment of code:

```
for(x=0, y=10; x <= y; ++x, --y) ← Multiple loop control variables  
    cout << x << ' ' << y << '\n';
```

Here, commas separate the two initialization statements and the two increment expressions. This is necessary in order for the compiler to understand that there are two initialization and two increment statements. In C++, the comma is an operator that essentially means “do this and this.” Its most common use is in the for loop. You can have any number of initialization and increment statements, but in practice, more than two or three make the for loop unwieldy.

Ask the Expert

Q: Does C++ support mathematical functions other than sqrt()?

A: Yes! In addition to sqrt(), C++ supports an extensive set of mathematical library functions. For example, sin(), cos(), tan(), log(), pow(), ceil(), and floor() are just a few. If mathematical programming is your interest, you will want to explore the C++ math functions. All C++ compilers support these functions, and their descriptions will be found in your compiler’s documentation. They all require the header <cmath>.

The condition controlling the loop may be any valid C++ expression. It does not need to involve the loop control variable. In the next example, the loop continues to execute until the rand() function produces a value greater than 20,000.

```
/*  
    Loop until a random number that is  
    greater than 20,000.  
*/
```

```

#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
    int i;
    int r;

    r = rand();

    for(i=0; r <= 20000; i++) ← This conditional expression does not use
        r = rand();           the loop control variable.

    cout << "Number is " << r <<
        ". It was generated on try " << i << ".";

    return 0;
}

```

Here is a sample run:

```
Number is 26500. It was generated on try 3.
```

Each time through the loop, a new random number is obtained by calling `rand()`. When a value greater than 20,000 is generated, the loop condition becomes false, terminating the loop.

Missing Pieces

Another aspect of the for loop that is different in C++ than in many computer languages is that pieces of the loop definition need not be there. For example, if you want to write a loop that runs until the number 123 is typed in at the keyboard, it could look like this:

```

// A for loop with no increment.

#include <iostream>
using namespace std;

int main()
{
    int x;

    for(x=0; x != 123; ) { ← No increment expression
        cout << "Enter a number: ";
        cin >> x;
    }

    return 0;
}

```


Here, the increment portion of the for definition is blank. This means that each time the loop repeats, x is tested to see whether it equals 123, but no further action takes place. If, however, you type 123 at the keyboard, the loop condition becomes false and the loop exits. The for loop will not modify the loop control variable if no increment portion of the loop is present.

Another variation on the for is to move the initialization section outside of the loop, as shown in this fragment:

`x = 0;` ← `x` is initialized outside the loop.

```
for( ; x<10; )
{
    cout << x << ' ';
    ++x;
}
```

Here, the initialization section has been left blank, and x is initialized before the loop is entered. Placing the initialization outside of the loop is generally done only when the initial value is derived through a complex process that does not lend itself to containment inside the for statement. Notice that in this example, the increment portion of the for is located inside the body of the loop.

The Infinite for Loop

You can create an infinite loop (a loop that never terminates) using this for construct:

```
for(;;) {
//... }
```

This loop will run forever. Although there are some programming tasks, such as operating system command processors, that require an infinite loop, most “infinite loops” are really just loops with special termination requirements. Near the end of this module, you will see how to halt a loop of this type. (Hint: It’s done using the break statement.)

Loops with No Body

In C++, the body associated with a for loop can be empty. This is because the null statement is syntactically valid. Bodiless loops are often useful. For example, the following program uses one to sum the numbers from 1 to 10:

```
// The body of a for loop can be empty.

#include <iostream>
#include <cstdlib>
using namespace std;
```

```

int main()
{
    int i;
    int sum = 0;

    // sum the numbers from 1 through 10
    for(i=1; i <= 10; sum += i++) ; ← This loop has no body.

    cout << "Sum is " << sum;

    return 0;
}

```

The output from the program is shown here:

```
Sum is 55
```

Notice that the summation process is handled entirely within the for statement and no body is needed. Pay special attention to the increment expression:

```
sum += i++
```

Don't be intimidated by statements like this. They are common in professionally written C++ programs and are easy to understand if you break them down into their parts. In words, this statement says, "add to sum the value of sum plus i, then increment i." Thus, it is the same as this sequence of statements:

```
sum = sum + i; i++;
```

Declaring Loop Control Variables Inside the for Loop

Often, the variable that controls a for loop is needed only for the purposes of the loop and is not used elsewhere. When this is the case, it is possible to declare the variable inside the initialization portion of the for. For example, the following program computes both the summation and the factorial of the numbers 1 through 5. It declares its loop control variable i inside the for:

```

// Declare loop control variable inside the for.

#include <iostream>
using namespace std;

int main() {
    int sum = 0;
    int fact = 1;

    // compute the factorial of the numbers through 5
    for(int i = 1; i <= 5; i++) { ← Here, i is declared inside
        sum += i; // i is known throughout the loop the for loop.
        fact *= i;
    }
}

```

```

// but, i is not known here.

cout << "Sum is " << sum << "\n";
cout << "Factorial is " << fact;

return 0;
}

```

Here is the output produced by the program:

```

Sum is 15
Factorial is 120

```

When you declare a variable inside a for loop, there is one important point to remember: the variable is known only within the for statement. Thus, in the language of programming, the scope of the variable is limited to the for loop. Outside the for loop, the variable will cease to exist. Therefore, in the preceding example, `i` is not accessible outside the for loop. If you need to use the loop control variable elsewhere in your program, you will not be able to declare it inside the for loop.

NOTE

Whether a variable declared within the initialization portion of a **for** loop is restricted to that loop or not has changed over time. Originally, the variable was available after the **for**, but this was changed during the C++ standardization process. Today, the ANSI/ISO Standard C++ restricts the variable to the scope of the **for** loop. Some compilers, however, do not. You will need to check this feature in the environment you are using.

Before moving on, you might want to experiment with your own variations on the for loop. As you will find, it is a fascinating loop.



Progress Check

1. Can portions of a **for** statement be empty?
2. Show how to create an infinite loop using **for**.
3. What is the scope of a variable declared within a **for** statement?

CRITICAL SKILL 3.4: The while Loop

Another loop is the while. The general form of the while loop is `while(expression) statement`; where `statement` may be a single statement or a block of statements. The `expression` defines the condition that controls the loop, and it can be any valid expression. The statement is performed while the condition is true. When the condition becomes false, program control passes to the line immediately following the loop.

The next program illustrates the while in a short but sometimes fascinating program. Virtually all computers support an extended character set beyond that defined by ASCII. The extended characters, if they exist, often include special characters such as foreign language symbols and scientific notations. The ASCII characters use values that are less than 128. The extended character set begins at 128 and continues to 255. This program prints all characters between 32 (which is a space) and 255. When you run this program, you will most likely see some very interesting characters.

```
/*
   This program displays all printable characters,
   including the extended character set, if one exists.
*/

#include <iostream>
using namespace std;

int main()
{
    unsigned char ch;

    ch = 32;
    while(ch) { ← Use a while loop.
        cout << ch;
        ch++;
    }

    return 0;
}
```

Examine the loop expression in the preceding program. You might be wondering why only `ch` is used to control the while. Since `ch` is an unsigned character, it can only hold the values 0 through 255. When it holds the value 255 and is then incremented, its value will “wrap around” to zero. Therefore, the test for `ch` being zero serves as a convenient stopping condition.

As with the for loop, the while checks the conditional expression at the top of the loop, which means that the loop code may not execute at all. This eliminates the need to perform a separate test before the loop. The following program illustrates this characteristic of the while loop. It displays a line of periods. The number of periods displayed is equal to the value entered by the user. The program does not allow lines longer than 80 characters. The test for a permissible number of periods is performed inside the loop’s conditional expression, not outside of it.

```

#include <iostream>
using namespace std;

int main()
{
    int len;

    cout << "Enter length (1 to 79): ";
    cin >> len;

    while(len>0 && len<80) {
        cout << '.';
        len--;
    }

    return 0;
}

```

If len is out of range, then the while loop will not execute even once. Otherwise, the loop executes until len reaches zero. There need not be any statements at all in the body of the while loop. Here is an example:

```
while(rand() != 100) ;
```

This loop iterates until the random number generated by rand() equals 100.

CRITICAL SKILL 3.5: The do-while Loop

The last of C++'s loops is the do-while. Unlike the for and the while loops, in which the condition is tested at the top of the loop, the do-while loop checks its condition at the bottom of the loop. This means that a do-while loop will always execute at least once. The general form of the do-while loop is

```
do { statements; } while(condition);
```

Although the braces are not necessary when only one statement is present, they are often used to improve readability of the do-while construct, thus preventing confusion with the while. The do-while loop executes as long as the conditional expression is true.

The following program loops until the number 100 is entered:

```

#include <iostream>
using namespace std;

int main()
{
    int num;
    do {
        cout << "Enter a number (100 to stop): ";
        cin >> num;
    } while(num != 100);

    return 0;
}

```

A do-while loop always executes at least once.

Using a do-while loop, we can further improve the Magic Number program. This time, the program loops until you guess the number.

```

// Magic Number program: 3rd improvement.

#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
    int magic; // magic number
    int guess; // user's guess

    magic = rand(); // get a random number

    do {
        cout << "Enter your guess: ";
        cin >> guess;
        if(guess == magic) {
            cout << "** Right ** ";
            cout << magic << " is the magic number.\n";
        }
        else {
            cout << "...Sorry, you're wrong.";
            if(guess > magic)
                cout << " Your guess is too high.\n";
            else cout << " Your guess is too low.\n";
        }
    } while(guess != magic);

    return 0;
}

```

Here is a sample run:

```
Enter your guess: 10
...Sorry, you're wrong. Your guess is too low.
Enter your guess: 100
...Sorry, you're wrong. Your guess is too high.
Enter your guess: 50
...Sorry, you're wrong. Your guess is too high.
Enter your guess: 41
** Right ** 41 is the magic number.
```

One last point: Like the `for` and `while`, the body of the `do-while` loop can be empty, but this is seldom the case in practice.

Progress Check

1. What is the main difference between the **while** and the **do-while** loops?
2. Can the body of a **while** loop be empty?

Ask the Expert

Q: Given the flexibility inherent in all of C++'s loops, what criteria should I use when selecting a loop? That is, how do I choose the right loop for a specific job?

A: Use a `for` loop when performing a known number of iterations. Use the `do-while` when you need a loop that will always perform at least one iteration. The `while` is best used when the loop will repeat an unknown number of times.

The `while` checks its condition at the top of the loop. The `do-while` checks its condition at the bottom of the loop. Thus, a `do-while` will always execute at least once.

Yes, the body of a `while` loop (or any other C++ loop) can be empty.

Project 3-2 Improve the C++ Help System

This project expands on the C++ help system that was created in Project 3-1. This version adds the syntax for the `for`, `while`, and `do-while` loops. It also checks the user's menu selection, looping until a valid response is entered. To do this, it uses a `do-while` loop. In general, using a `do-while` loop to handle menu selection is common because you will always want the loop to execute at least once.

Step by Step

1. Copy `Help.cpp` to a new file called `Help2.cpp`.

2. Change the portion of the program that displays the choices so that it uses the do-while loop shown here:

```
do {
    cout << "Help on:\n";
    cout << "  1. if\n";
    cout << "  2. switch\n";
    cout << "  3. for\n";
    cout << "  4. while\n";
    cout << "  5. do-while\n";
    cout << "Choose one: ";

    cin >> choice;

} while( choice < '1' || choice > '5');
```

After making this change, the program will loop, displaying the menu until the user enters a response that is between 1 and 5. You can see how useful the do-while loop is in this context.

3. Expand the switch statement to include the for, while, and do-while loops, as shown here:

```
switch(choice) {
    case '1':
        cout << "The if:\n\n";
        cout << "if(condition) statement;\n";
        cout << "else statement;\n";
        break;
    case '2':
        cout << "The switch:\n\n";
        cout << "switch(expression) {\n";
        cout << "  case constant:\n";
        cout << "    statement sequence\n";
```



```

        cout << "    break;\n";
        cout << "    // ... \n";
        cout << "}\n";
        break;
    case '3':
        cout << "The for:\n\n";
        cout << "for(init; condition; increment) ";
        cout << " statement;\n";
        break;
    case '4':
        cout << "The while:\n\n";
        cout << "while(condition) statement;\n";
        break;
    case '5':
        cout << "The do-while:\n\n";
        cout << "do {\n";
        cout << "    statement;\n";
        cout << "} while (condition);\n";
        break;
}

```

Notice that no default statement is present in this version of the switch. Since the menu loop ensures that a valid response will be entered, it is no longer necessary to include a default statement to handle an invalid choice.

4. Here is the entire Help2.cpp program listing:

```

/*
    Project 3-2

    An improved Help system that uses a
    a do-while to process a menu selection.
*/

#include <iostream>
using namespace std;

int main() {
    char choice;

    do {
        cout << "Help on:\n";
        cout << "    1. if\n";
        cout << "    2. switch\n";
        cout << "    3. for\n";
        cout << "    4. while\n";
    }
}

```

```

    cout << "Choose one: ";

    cin >> choice;

} while( choice < '1' || choice > '5');

cout << "\n\n";

switch(choice) {
    case '1':
        cout << "The if:\n\n";
        cout << "if(condition) statement;\n";
        cout << "else statement;\n";
        break;
    case '2':
        cout << "The switch:\n\n";
        cout << "switch(expression) {\n";
        cout << "    case constant:\n";
        cout << "        statement sequence\n";
        cout << "        break;\n";
        cout << "    // ... \n";
        cout << "}\n";
        break;
    case '3':
        cout << "The for:\n\n";
        cout << "for(init; condition; increment)";
        cout << " statement;\n";
        break;
    case '4':
        cout << "The while:\n\n";
        cout << "while(condition) statement;\n";
        break;
    case '5':
        cout << "The do-while:\n\n";
        cout << "do {\n";
        cout << "    statement;\n";
        cout << "} while (condition);\n";
        break;
}

return 0;
}

```

Ask the Expert

Q: Earlier you showed how a variable could be declared in the initialization portion of the for loop. Can variables be declared inside any other C++ control statement?

A: Yes. In C++, it is possible to declare a variable within the conditional expression of an if or switch, within the conditional expression of a while loop, or within the initialization portion of a for loop. A variable declared in one of these places has its scope limited to the block of code controlled by that statement.

You have already seen an example of declaring a variable within a for loop. Here is an example that declares a variable within an if:

```
if(int x = 20) { x = x - y; if(x > 10) y = 0;
}
```

The if declares x and assigns it the value 20. Since this is a true value, the target of the if executes. Because variables declared within a conditional statement have their scope limited to the block of code controlled by that statement, x is not known outside the if.

As mentioned in the discussion of the for loop, whether a variable declared within a control statement is known only to that statement or is available after that statement may vary between compilers. You should check the compiler that you are using before assuming a specific behavior in this regard. Of course, the ANSI/ISO C++ standard stipulates that the variable is known only within the statement in which it is declared.

Most programmers do not declare variables inside any control statement other than the for. In fact, the declaration of variables within the other statements is controversial, with some programmers suggesting that to do so is bad practice.

CRITICAL SKILL 3.6: Using break to Exit a Loop

It is possible to force an immediate exit from a loop, bypassing the loop's conditional test, by using the break statement. When the break statement is encountered inside a loop, the loop is immediately terminated, and program control resumes at the next statement following the loop. Here is a simple example:

```
#include <iostream>
using namespace std;
```

```

int main()
{
    int t;

    // Loops from 0 to 9, not to 100!
    for(t=0; t<100; t++) {
        if(t==10) break; ← Break out of the for when t equals 10.
        cout << t << ' ';
    }

    return 0;
}

```

The output from the program is shown here:

```
0 1 2 3 4 5 6 7 8 9
```

As the output illustrates, this program prints only the numbers 0 through 9 on the screen before ending. It will not go to 100, because the break statement will cause it to terminate early.

When loops are nested (that is, when one loop encloses another), a break will cause an exit from only the innermost loop. Here is an example:

```

#include <iostream>
using namespace std;

int main()
{
    int t, count;

    for(t=0; t<10; t++) {
        count = 1;
        for(;;) {
            cout << count << ' ';
            count++;
            if(count==10) break;
        }
        cout << '\n';
    }

    return 0;
}

```

Here is the output produced by this program:

```
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
```

As you can see, this program prints the numbers 1 through 9 on the screen ten times. Each time the break is encountered in the inner for loop, control is passed back to the outer for loop. Notice that the inner for is an infinite loop that uses the break statement to terminate.

The break statement can be used with any loop statement. It is most appropriate when a special condition can cause immediate termination. One common use is to terminate infinite loops, as the foregoing example illustrates.

One other point: A break used in a switch statement will affect only that switch, and not any loop the switch happens to be in.

CRITICAL SKILL 3.7: Using continue

It is possible to force an early iteration of a loop, bypassing the loop's normal control structure. This is accomplished using continue. The continue statement forces the next iteration of the loop to take place, skipping any code between itself and the conditional expression that controls the loop. For example, the following program prints the even numbers between 0 and 100:

```
#include <iostream>
using namespace std;

int main()
{
    int x;

    for(x=0; x<=100; x++) {
        if(x%2) continue; ← Continue the loop early when x is odd.
        cout << x << ' ';
    }

    return 0;
}
```

Only even numbers are printed, because an odd number will cause the continue statement to execute, resulting in early iteration of the loop, bypassing the cout statement. Remember that the % operator

produces the remainder of an integer division. Thus, when *x* is odd, the remainder is 1, which is true. When *x* is even, the remainder is 0, which is false.

In while and do-while loops, a continue statement will cause control to go directly to the conditional expression and then continue the looping process. In the case of the for, the increment part of the loop is performed, then the conditional expression is executed, and then the loop continues.



Progress Check

1. Within a loop, what happens when a **break** is executed?
2. What does **continue** do?
3. Assuming a loop that encloses a **switch**, does a **break** in the **switch** cause the loop to terminate?

Project 3-3 Finish the C++ Help System

This project puts the finishing touches on the C++ help system that was created by the previous projects. This version adds the syntax for break, continue, and goto. It also allows the user to request the syntax for more than one statement. It does this by adding an outer loop that runs until the user enters a q as a menu selection.

Step by Step

1. Copy Help2.cpp to a new file called Help3.cpp.
2. Surround all of the program code with an infinite for loop. Break out of this loop, using break, when a q is entered. Since this loop surrounds all of the program code, breaking out of this loop causes the program to terminate.
3. Change the menu loop, as shown here:

```
do {  
    cout << "Help on:\n";
```

```

    cout << "  1. if\n";
    cout << "  2. switch\n";
    cout << "  3. for\n";
    cout << "  4. while\n";
    cout << "  5. do-while\n";
    cout << "  6. break\n";
    cout << "  7. continue\n";
    cout << "  8. goto\n";
    cout << "Choose one (q to quit): ";
    cin >> choice;
} while( choice < '1' || choice > '8' && choice != 'q');

```

Notice that this loop now includes the break, continue, and goto statements. It also accepts a q as a valid choice.

4. Expand the switch statement to include the break, continue, and goto statements, as shown here:

```

case '6':
    cout << "The break:\n\n";
    cout << "break;\n";
    break;
case '7':
    cout << "The continue:\n\n";
    cout << "continue;\n";
    break;
case '8':
    cout << "The goto:\n\n";
    cout << "goto label;\n";
    break;

```

5. Here is the entire Help3.cpp program listing:

```

/*
   Project 3-3

   The finished Help system that processes multiple requests.
*/

#include <iostream>
using namespace std;

int main() {
    char choice;

    for(;;) {
        do {

```

```

    cout << "Help on:\n";
    cout << "  1. if\n";
    cout << "  2. switch\n";
    cout << "  3. for\n";
    cout << "  4. while\n";
    cout << "  5. do-while\n";
    cout << "  6. break\n";
    cout << "  7. continue\n";
    cout << "  8. goto\n";
    cout << "Choose one (q to quit): ";
    cin >> choice;
} while( choice < '1' || choice > '8' && choice != 'q');

if(choice == 'q') break;

cout << "\n\n";

switch(choice) {
    case '1':
        cout << "The if:\n\n";
        cout << "if(condition) statement;\n";
        cout << "else statement;\n";
        break;
    case '2':
        cout << "The switch:\n\n";
        cout << "switch(expression) {\n";
        cout << "    case constant:\n";
        cout << "        statement sequence\n";
        cout << "        break;\n";
        cout << "    // ... \n";
        cout << "}\n";
        break;
    case '3':
        cout << "The for:\n\n";
        cout << "for(init; condition; increment)";
        cout << " statement;\n";
        break;
    case '4':
        cout << "The while:\n\n";
        cout << "while(condition) statement;\n";
        break;
    case '5':
        cout << "The do-while:\n\n";
        cout << "do {\n";
        cout << "    statement;\n";

```



```

        cout << "} while (condition);\n";
        break;
    case '6':
        cout << "The break:\n\n";
        cout << "break;\n";
        break;
    case '7':
        cout << "The continue:\n\n";
        cout << "continue;\n";
        break;
    case '8':
        cout << "The goto:\n\n";
        cout << "goto label;\n";
        break;
    }
    cout << "\n";
}

return 0;
}

```

6. Here is a sample run:

Help on:

1. if
2. switch
3. for
4. while
5. do-while
6. break
7. continue
8. goto

Choose one (q to quit): 1

The if:

```

if(condition) statement;
else statement;

```

Help on:

1. if
2. switch
3. for
4. while
5. do-while
6. break
7. continue
8. goto

Choose one (q to quit): 6

```
The break:
break;
```

```
Help on:
 1. if
 2. switch
 3. for
 4. while
 5. do-while
 6. break
 7. continue
 8. goto
```

```
Choose one (q to quit): q
```

CRITICAL SKILL 3.8: Nested Loops

As you have seen in some of the preceding examples, one loop can be nested inside of another. Nested loops are used to solve a wide variety of problems and are an essential part of programming. So, before leaving the topic of C++'s loop statements, let's look at one more nested loop example. The following program uses a nested for loop to find the factors of the numbers from 2 to 100:

```
/*
   Use nested loops to find factors of numbers
   between 2 and 100.
*/

#include <iostream>
using namespace std;

int main() {

    for(int i=2; i <= 100; i++) {
        cout << "Factors of " << i << ": ";

        for(int j = 2; j < i; j++)
            if((i%j) == 0) cout << j << " ";

        cout << "\n";
    }

    return 0;
}
```

Here is a portion of the output produced by the program:

```
Factors of 2:
Factors of 3:
Factors of 4: 2
Factors of 5:
```

```

Factors of 6: 2 3
Factors of 7:
Factors of 8: 2 4
Factors of 9: 3
Factors of 10: 2 5
Factors of 11:
Factors of 12: 2 3 4 6
Factors of 13:
Factors of 14: 2 7
Factors of 15: 3 5
Factors of 16: 2 4 8
Factors of 17:
Factors of 18: 2 3 6 9
Factors of 19:
Factors of 20: 2 4 5 10

```

In the program, the outer loop runs *i* from 2 through 100. The inner loop successively tests all numbers from 2 up to *i*, printing those that evenly divide *i*.

CRITICAL SKILL 3.9: Using the goto Statement

The `goto` is C++'s unconditional jump statement. Thus, when encountered, program flow jumps to the location specified by the `goto`. The statement fell out of favor with programmers many years ago because it encouraged the creation of “spaghetti code.” However, the `goto` is still occasionally—and sometimes effectively—used. This book will not make a judgment regarding its validity as a form of program control. It should be stated, however, that there are no programming situations that require the use of the `goto` statement; it is not needed to make the language complete. Rather, it is a convenience which, if used wisely, can be of benefit in certain programming situations. As such, the `goto` is not used in this book outside of this section. The chief concern most programmers have about the `goto` is its tendency to clutter a program and render it nearly unreadable. However, there are times when the use of the `goto` can clarify program flow rather than confuse it.

The `goto` requires a label for operation. A label is a valid C++ identifier followed by a colon. Furthermore, the label must be in the same function as the `goto` that uses it. For example, a loop from 1 to 100 could be written using a `goto` and a label, as shown here:

```

x = 1;
loop1: ←—————|
    x++;
    if(x < 100) goto loop1; —————→ Execution jumps to loop1.

```

One good use for the `goto` is to exit from a deeply nested routine. For example, consider the following code fragment:

```

for(...) {
    for(...) {
        while(...) {
            if(...) goto stop;
            .
            .
            .
        }
    }
}
stop:
    cout << "Error in program.\n";

```

Eliminating the goto would force a number of additional tests to be performed. A simple break statement would not work here, because it would only cause the program to exit from the innermost loop.

Module 3 Mastery Check

1. Write a program that reads characters from the keyboard until a \$ is typed. Have the program count the number of periods. Report the total at the end of the program.
2. In the switch, can the code sequence from one case run into the next? Explain.
3. Show the general form of the if-else-if ladder.

4. Given

```

if(x < 10)
    if(y > 100) {
        if(!done) x = z;
        else y = z;
    }
else cout << "error"; // what if?

```

to what if does the last else associate?

5. Show the for statement for a loop that counts from 1000 to 0 by -2.
6. Is the following fragment valid?

```

for(int i = 0; i < num; i++)
    sum += i;

count = i;

```

7. Explain what break does.
8. In the following fragment, after the break statement executes, what is displayed?

```

for(i = 0; i < 10; i++) {
    while(running) {
        if(x<y) break;
        // ...
    }
    cout << "after while\n";
}
cout << "After for\n";

```

9. What does the following fragment print?

```

for(int i = 0; i<10; i++) {
    cout << i << " ";
    if(!(i%2)) continue;
    cout << "\n";
}

```

10. The increment expression in a for loop need not always alter the loop control variable by a fixed amount. Instead, the loop control variable can change in any arbitrary way. Using this concept, write a program that uses a for loop to generate and display the progression 1, 2, 4, 8, 16, 32, and so on.
11. The ASCII lowercase letters are separated from the uppercase letters by 32. Thus, to convert a lowercase letter to uppercase, subtract 32 from it. Use this information to write a program that reads characters from the keyboard. Have it convert all lowercase letters to uppercase, and all uppercase letters to lowercase, displaying the result. Make no changes to any other character. Have the program stop when the user enters a period. At the end, have the program display the number of case changes that have taken place.
12. What is C++'s unconditional jump statement?