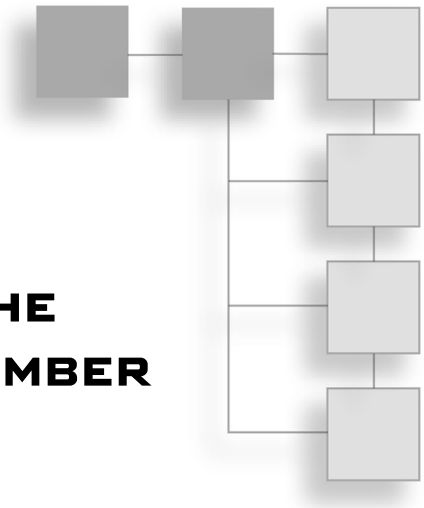
Chapter 2



Truth, Branching, and the Game Loop: Guess My Number

So far, the programs you've seen have been linear—each statement executes, in order, from top to bottom. However, to create interesting games, you need to write programs that execute (or skip) sections of code based on some condition. That's the main topic of this chapter. Specifically, you'll learn to:

- Understand truth (as C++ defines it)
- Use `if` statements to branch to sections of code
- Use `switch` statements to select a section of code to execute
- Use `while` and `do` loops to repeat sections of code
- Generate random numbers

Understanding Truth

Truth is black and white, at least as far as C++ is concerned. You can represent true and false with their corresponding keywords, `true` and `false`. You can store such a Boolean value with a `bool` variable, as you saw in Chapter 1, "Types, Variables, and Standard I/O: Lost Fortune." Here's a quick refresher:

```
bool fact = true, fiction = false;
```

This code creates two `bool` variables, `fact` and `fiction`. `fact` is `true` and `fiction` is `false`. Although the keywords `true` and `false` are handy, any expression or value can be interpreted as `true` or `false` too. Any non-zero value can be interpreted as `true`, while 0 can be interpreted as `false`.

Copyright © ${Date}, ${Publisher}. All rights reserved.

A common kind of expression interpreted as `true` or `false` involves comparing things. Comparisons are often made by using built-in relational operators. Table 2.1 lists the operators and a few sample expressions.

Table 2.1 Relational Operators

Operator	Meaning	Sample Expression	Evaluates To
==	equal to	5 == 5	true
		5 == 8	false
!=	not equal to	5 != 8	true
		5 != 5	false
>	greater than	8 > 5	true
		5 > 8	false
<	less than	5 < 8	true
		8 < 5	false
>=	greater than or equal to	8 >= 5	true
		5 >= 8	false
<=	less than or equal to	5 <= 8	true
		8 <= 5	false

Using the if Statement

Okay, it's time to put the concepts of true and false to work. You can use an `if` statement to test an expression for truth and execute some code based on it. Here's a simple form of the `if` statement:

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

If *`expression`* is `true`, then *`statement`* is executed. Otherwise, *`statement`* is skipped and the program branches to the statement after the `if` suite.

Hint

Whenever you see a generic *`statement`* like in the preceding code example, you can replace it with a single statement or a block of statements because a block is treated as a single unit.

Copyright © ${Date}. ${Publisher}. All rights reserved.

Introducing the Score Rater Program

The Score Rater program comments on a player's score using an `if` statement. Figure 2.1 shows the program in action.

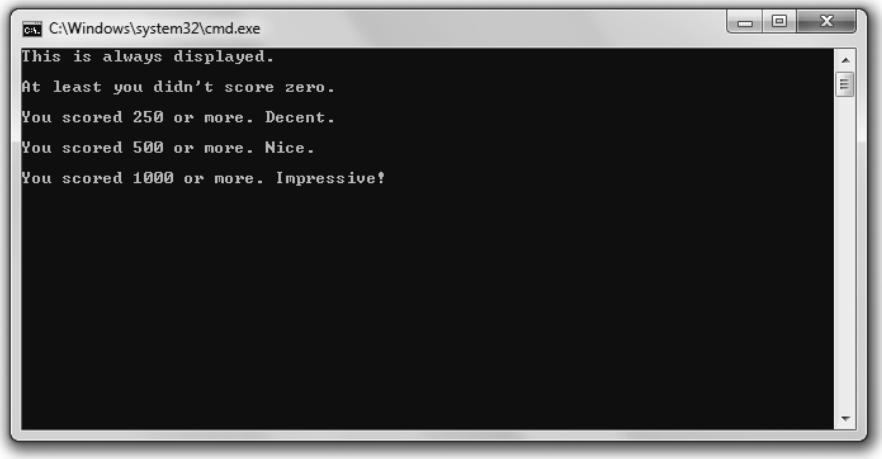


Figure 2.1
Messages are displayed (or not displayed) based on different `if` statements.
Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 2 folder; the filename is `score_rater.cpp`.

```
// Score Rater
// Demonstrates the if statement

#include <iostream>
using namespace std;

int main()
{
    if (true)
    {
        cout << "This is always displayed.\n\n";
    }

    if (false)
    {
```

Copyright © ${Date}. ${Publisher}. All rights reserved.

```
        cout << "This is never displayed.\n\n";
    }

    int score = 1000;

    if (score)
    {
        cout << "At least you didn't score zero.\n\n";
    }

    if (score >= 250)
    {
        cout << "You scored 250 or more. Decent.\n\n";
    }

    if (score >= 500)
    {
        cout << "You scored 500 or more. Nice.\n\n";

        if (score >= 1000)
        {
            cout << "You scored 1000 or more. Impressive!\n";
        }
    }

    return 0;
}
```

Testing true and false

In the first `if` statement I test `true`. Because `true` is, well, true, the program displays the message, "This is always displayed."

```
if (true)
{
    cout << "This is always displayed.\n\n";
}
```

In the next `if` statement I test `false`. Because `false` isn't true, the program doesn't display the message, "This is never displayed."

```
if (false)
{
    cout << "This is never displayed.\n\n";
}
```

Copyright © ${Date}. ${Publisher}. All rights reserved.

Trap

Notice that you don't use a semicolon after the closing parenthesis of the expression you test in an `if` statement. If you were to do this, you'd create an empty statement that would be paired with the `if` statement, essentially rendering the `if` statement useless. Here's an example:

```
if (false);
{
    cout << "This is never displayed.\n\n";
}
```

By adding the semicolon after `(false)`, I create an empty statement that's associated with the `if` statement. The preceding code is equivalent to:

```
if (false)
      ; // an empty statement, which does nothing
{
    cout << "This is never displayed.\n\n";
}
```

All I've done is play with the whitespace, which doesn't change the meaning of the code. Now the problem should be clear. The `if` statement sees the `false` value and skips the next statement (the empty statement). Then the program goes on its merry way to the statement after the `if` statement, which displays the message, "This is never displayed."

Be on guard for this error. It's an easy one to make and because it's not illegal, it won't produce a compile error.

Interpreting a Value as true or false

You can interpret any value as `true` or `false`. Any non-zero value can be interpreted as `true`, while `0` can be interpreted as `false`. I put this to the test in the next `if` statement:

```
if (score)
{
    cout << "At least you didn't score zero.\n\n";
}
```

`score` is 1000, so it's non-zero and interpreted as `true`. As a result, the message, "At least you didn't score zero," is displayed.

Using Relational Operators

Probably the most common expression you'll use with `if` statements involves comparing values using the relational operators. That's just what I'll demonstrate next. I test to see whether the score is greater than or equal to 250.

Copyright © ${Date}. ${Publisher}. All rights reserved.

```
if (score >= 250)
{
    cout << "You scored 250 or more. Decent.\n\n";
}
```

Because `score` is 1000, the block is executed, displaying the message that the player earned a decent score. If `score` had been less than 1000, the block would have been skipped and the program would have continued with the statement following the block.

Trap

The equal to relational operator is == (two equal signs in a row). Don't confuse it with = (one equal sign), which is the assignment operator.

While it's not illegal to use the assignment operator instead of the equal to relational operator, the results might not be what you expect. Take a look at this code:

```
int score = 500;
if (score = 1000)
{
        cout << " You scored 1000 or more. Impressive!\n";
}
```

As a result of this code, `score` is set to 1000 and the message, "You scored 1000 or more. Impressive!" is displayed. Here's what happens: Although `score` is 500 before the `if` statement, that changes. When the expression of the `if` statement, `(score = 1000)`, is evaluated, `score` is assigned 1000. The assignment statement evaluates to 1000, and because that's a non-zero value, the expression is interpreted as `true`. As a result, the string is displayed.

Be on guard for this type of mistake. It's easy to make, and in some cases (like this one) it won't cause a compile error.

Nesting if Statements

An `if` statement can cause a program to execute a statement or block of statements, including other `if` statements. When you write one `if` statement inside another, it's called *nesting*. In the following code, the `if` statement that begins `if (score >= 1000)` is nested inside the `if` statement that begins `if (score >= 500)`.

```
if (score >= 500)
{
    cout << "You scored 500 or more. Nice.\n\n";
```

Copyright © ${Date}. ${Publisher}. All rights reserved.

```
        if (score >= 1000)
        {
            cout << "You scored 1000 or more. Impressive!\n";
        }
    }
```

Because `score` is greater than 500, the program enters the statement block and displays the message, "You scored 500 or more. Nice." Then, in the inner `if` statement, the program compares `score` to 1000. Because `score` is greater than or equal to 1000, the program displays the message, "You scored 1000 or more. Impressive!"

Hint

You can nest as many levels as you want. However, if you nest code too deeply, it gets hard to read. In general, you should try to limit your nesting to a few levels at most.

Using the else Clause

You can add an `else` clause to an `if` statement to provide code that will only be executed if the tested expression is `false`. Here's the form of an `if` statement that includes an `else` clause:

```
if (expression)
    statement1;
else
    statement2;
```

If *`expression`* is `true`, *`statement1`* is executed. Then the program skips *`statement2`* and executes the statement following the `if` suite. If *`expression`* is `false`, *`statement1`* is skipped and *`statement2`* is executed. After *`statement2`* completes, the program executes the statement following the `if` suite.

Introducing the Score Rater 2.0 Program

The Score Rater 2.0 program also rates a score, which the user enters. But this time, the program uses an `if` statement with an `else` clause. Figures 2.2 and 2.3 show the two different messages that the program can display based on the score the user enters.

Copyright © ${Date}. ${Publisher}. All rights reserved.

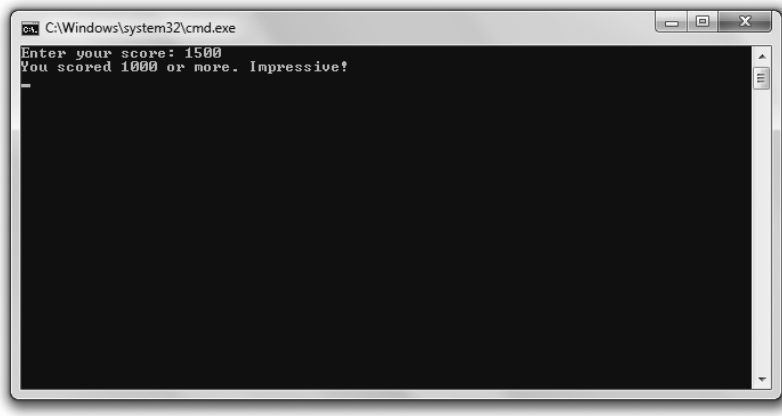


Figure 2.2
If the user enters a score that's 1000 or more, he is congratulated.
Used with permission from Microsoft.

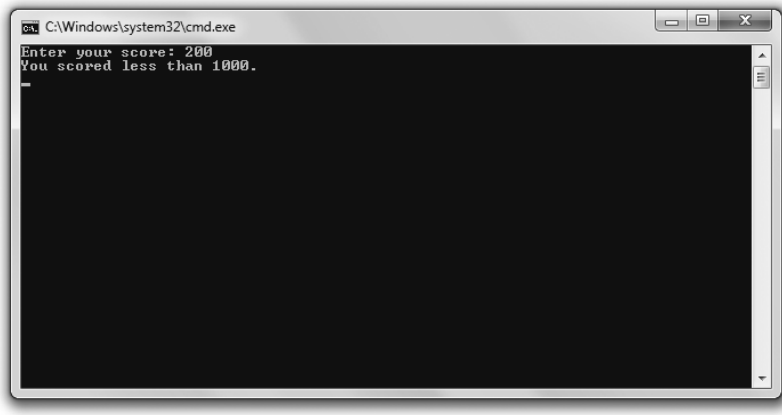


Figure 2.3
If the user enters a score that's less than 1000, there's no celebration.
Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 2 folder; the filename is `score_rater2.cpp`.

```
// Score Rater 2.0
// Demonstrates an else clause

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

Copyright © ${Date}. ${Publisher}. All rights reserved.

```
 int main()
{
    int score;
    cout << "Enter your score: ";
    cin >> score;

    if (score >= 1000)
    {
        cout << "You scored 1000 or more. Impressive!\n";
    }
    else
    {
        cout << "You scored less than 1000.\n";
    }

    return 0;
}
```

Creating Two Ways to Branch

You've seen the first part of the `if` statement already, and it works just as it did before. If `score` is greater than 1000, the message, "You scored 1000 or more. Impressive!" is displayed.

```
    if (score >= 1000)
    {
        cout << "You scored 1000 or more. Impressive!\n";
    }
```

Here's the twist. The `else` clause provides a statement for the program to branch to if the expression is `false`. So if `(score >= 1000)` is `false`, then the program skips the first message and instead displays the message, "You scored less than 1000."

```
    else
    {
        cout << "You scored less than 1000.\n";
    }
```

Using a Sequence of if Statements with else Clauses

You can chain together `if` statements with `else` clauses to create a sequence of expressions that are tested in order. The statement associated with the first expression to test true is executed; otherwise, the statement associated with the final (and optional) `else` clause is run. Here's the form such a series would take:

Copyright © ${Date}. ${Publisher}. All rights reserved.

```
if (expression1)
    statement1;
else if (expression2)
    statement2;

...

else if (expressionN)
    statementN;
else
    statementN+1;
```

If *expression1* is `true`, *statement1* is executed and the rest of the code in the sequence is skipped. Otherwise, *expression2* is tested and if `true`, *statement2* is executed and the rest of the code in the sequence is skipped. The computer continues to check each expression in order (through *expressionN*) and will execute the statement associated with the first expression that is true. If no expression is true, then the statement associated with the final `else` clause, *statementN+1*, is executed.

Introducing the Score Rater 3.0 Program

The Score Rater 3.0 program also rates a score, which the user enters. But this time, the program uses a sequence of `if` statements with `else` clauses. Figure 2.4 shows the results of the program.

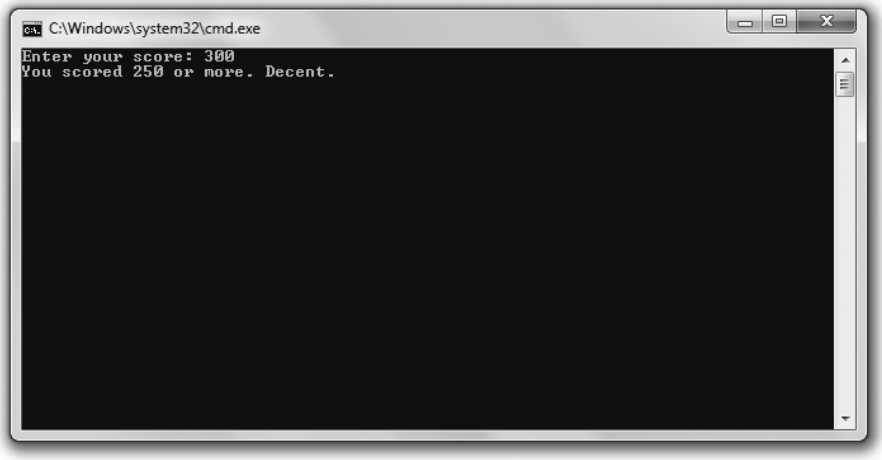


Figure 2.4
The user can get one of multiple messages, depending on his score.
Used with permission from Microsoft.

Copyright © ${Date}. ${Publisher}. All rights reserved.

You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 2 folder; the filename is `score_rater3.cpp`.

```
// Score Rater 3.0
// Demonstrates if else-if else suite

#include <iostream>
using namespace std;

int main()
{
    int score;
    cout << "Enter your score: ";
    cin >> score;

    if (score >= 1000)
    {
       cout << "You scored 1000 or more. Impressive!\n";
    }
    else if (score >= 500)
    {
       cout << "You scored 500 or more. Nice.\n";
    }
    else if (score >= 250)
    {
       cout << "You scored 250 or more. Decent.\n";
    }
    else
    {
       cout << "You scored less than 250. Nothing to brag about.\n";
    }

    return 0;
}
```

Creating a Sequence of if Statements with else Clauses

You've seen the first part of this sequence twice already, and it works just the same this time around. If `score` is greater than or equal to 1000, the message, "You scored 1000 or more. Impressive!" is displayed and the computer branches to the `return` statement.

```
    if (score >= 1000)
```

Copyright © ${Date}. ${Publisher}. All rights reserved.

However, if the expression is `false`, then we know that `score` is less than 1000 and the computer evaluates the next expression in the sequence:

```
else if (score >= 500)
```

If `score` is greater than or equal to 500, the message, "You scored 500 or more. Nice." is displayed and the computer branches to the `return` statement. However, if that expression is `false`, then we know that `score` is less than 500 and the computer evaluates the next expression in the sequence:

```
else if (score >= 250)
```

If `score` is greater than or equal to 250, the message, "You scored 250 or more. Decent." is displayed and the computer branches to the `return` statement. However, if that expression is `false`, then we know that `score` is less than 250 and the statement associated with the final `else` clause is executed and the message, "You scored less than 250. Nothing to brag about." is displayed.

Hint

While the final `else` clause in an `if else-if` suite isn't required, you can use it as a way to execute code if none of the expressions in the sequence are true.

Using the switch Statement

You can use a `switch` statement to create multiple branching points in your code. Here's a generic form of the `switch` statement:

```
switch (choice)
{
   case value1:
          statement1;
          break;
   case value2:
          statement2;
          break;
   case value3:
          statement3;
          break;
               .
               .
               .
```

Copyright © ${Date}. ${Publisher}. All rights reserved.

```
        case valueN:
              statementN;
              break;
        default:
              statementN + 1;
  }
```

The statement tests *choice* against the possible values—*value1*, *value2*, and *value3*—in order. If *choice* is equal to a value, then the program executes the corresponding *statement*. When the program hits a `break` statement, it exits the `switch` structure. If *choice* doesn't match any value, then the statement associated with the optional `default` is executed.

The use of `break` and `default` are optional. If you leave out a `break`, however, the program will continue through the remaining statements until it hits a `break` or a `default` or until the `switch` statement ends. Usually you want one `break` statement to end each `case`.

Hint

Although a `default` case isn't required, it's usually a good idea to have one as a catchall.

Here's an example to cement the ideas. Suppose *choice* is equal to *value2*. The program will first test *choice* against *value1*. Because they're not equal, the program will continue. Next, the program will test *choice* against *value2*. Because they are equal, the program will execute *statement2*. Then the program will hit the `break` statement and exit the `switch` structure.

Trap

You can use the `switch` statement only to test an `int` (or a value that can be treated as an `int`, such as a `char` or an `enumerator`). A `switch` statement won't work with any other type.

Introducing the Menu Chooser Program

The Menu Chooser program presents the user with a menu that lists three difficulty levels and asks him to make a choice. If the user enters a number that corresponds to a listed choice, then he is shown a message confirming the choice. If the user makes some other choice, he is told that the choice is invalid. Figure 2.5 shows the program in action.

Copyright © ${Date}, ${Publisher}. All rights reserved.

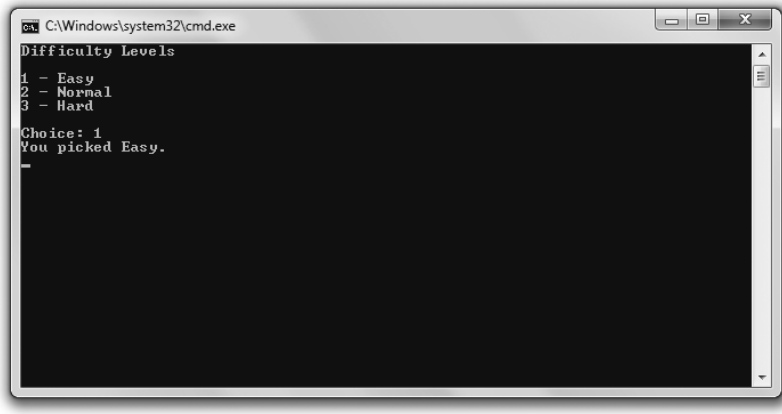


Figure 2.5
Looks like I took the easy way out.
Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 2 folder; the filename is `menu_chooser.cpp`.

```
// Menu Chooser
// Demonstrates the switch statement

#include <iostream>

using namespace std;

int main()
{
    cout << "Difficulty Levels\n\n";
    cout << "1 - Easy\n";
    cout << "2 - Normal\n";
    cout << "3 - Hard\n\n";

    int choice;
    cout << "Choice: ";
    cin >> choice;

    switch (choice)
    {
        case 1:
            cout << "You picked Easy.\n";
            break;
        case 2:
            cout << "You picked Normal.\n";
            break;
```

Copyright © ${Date}. ${Publisher}. All rights reserved.

```
        case 3:
               cout << "You picked Hard.\n";
               break;
        default:
               cout << "You made an illegal choice.\n";
    }
    return 0;
}
```

Creating Multiple Ways to Branch

The `switch` statement creates four possible branching points. If the user enters `1`, then code associated with `case 1` is executed and "You picked Easy" is displayed. If the user enters `2`, then code associated with `case 2` is executed and "You picked Normal" is displayed. If the user enters `3`, then code associated with `case 3` is executed and "You picked Hard" is displayed. If the user enters any other value, then `default` kicks in and "You made an illegal choice" is displayed.

Trap

You'll almost always want to end each case with a `break` statement. Don't forget them; otherwise, your code might do things you never intended.

Using while Loops

`while` loops let you repeat sections of code as long as an expression is `true`. Here's a generic form of the `while` loop:

```
while (expression)
    statement;
```

If *`expression`* is `false`, the program moves on to the statement after the loop. If *`expression`* is `true`, the program executes *`statement`* and loops back to test *`expression`* again. This cycle repeats until *`expression`* tests `false`, at which point the loop ends.

Introducing the Play Again Program

The Play Again program simulates the play of an exciting game. (Okay, by "simulates the play of an exciting game," I mean the program displays the message "**Played an exciting game**.") Then the program asks the user if he wants to play again. The user continues to play as long as he enters `y`. The program accomplishes this repetition using a `while` loop. Figure 2.6 shows the program in action.

Copyright © ${Date}, ${Publisher}. All rights reserved.

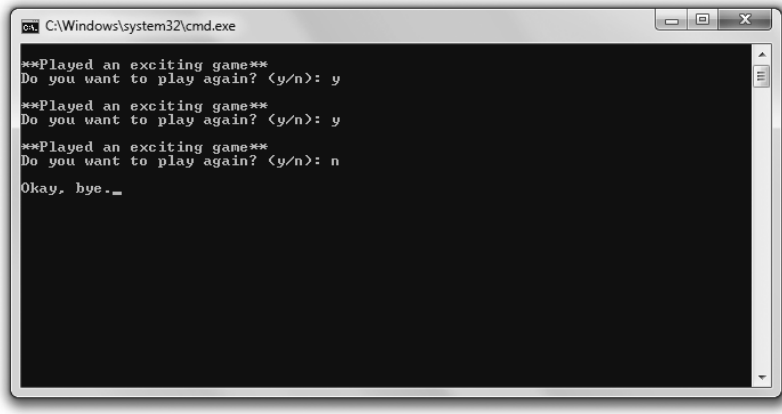


Figure 2.6
The repetition is accomplished using a `while` loop.
Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 2 folder; the filename is `play_again.cpp`.

```
// Play Again
// Demonstrates while loops

#include <iostream>
using namespace std;

int main()
{
    char again = 'y';
    while (again == 'y')
    {
        cout << "\n**Played an exciting game**";
        cout << "\nDo you want to play again? (y/n): ";
        cin >> again;
    }

    cout << "\nOkay, bye.";

    return 0;
}
```

Looping with a while Loop

The first thing the program does in the `main()` function is declare the `char` variable named `again` and initialize it to `'y'`. Then the program begins the `while` loop by testing `again` to see whether it's equal to `'y'`. Because it is, the program displays the message "**Played an

Copyright © ${Date}. ${Publisher}. All rights reserved.

exciting game**," asks the user whether he wants to play again, and stores the reply in `again`. The loop continues as long as the user enters `y`.

You'll notice that I had to initialize `again` before the loop because the variable is used in the loop expression. Because a `while` loop evaluates its expressions before its *loop body* (the group of statements that repeat), you have to make sure that any variables in the expression have a value before the loop begins.

Using do Loops

Like `while` loops, `do` loops let you repeat a section of code based on an expression. The difference is that a `do` loop tests its expression after each loop iteration. This means that the loop body is always executed at least once. Here's a generic form of a `do` loop:

```
do
        statement;
while (expression)
```

The program executes *`statement`* and then, as long as *`expression`* tests `true`, the loop repeats. Once *`expression`* tests `false`, the loop ends.

Introducing the Play Again 2.0 Program

The Play Again 2.0 program looks exactly the same to the user as the original Play Again program. Play Again 2.0, like its predecessor, simulates the play of an exciting game by displaying the message "**Played an exciting game**" and asking the user whether he wants to play again. The user continues to play as long as he enters `y`. This time, though, the program accomplishes the repetition using a `do` loop. Figure 2.7 shows off the program.

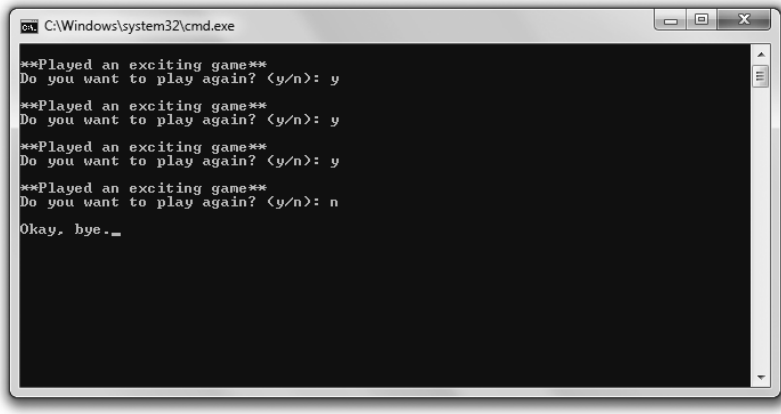


Figure 2.7
Each repetition is accomplished using a `do` loop.
Used with permission from Microsoft.

Copyright © ${Date}. ${Publisher}. All rights reserved.

You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 2 folder; the filename is `play_again2.cpp`.

```
// Play Again 2.0
// Demonstrates do loops

#include <iostream>

using namespace std;

int main()
{
    char again;
    do
    {
        cout << "\n**Played an exciting game**";
        cout << "\nDo you want to play again? (y/n): ";
        cin >> again;
    } while (again == 'y');

    cout << "\nOkay, bye.";

    return 0;
}
```

Looping with a do Loop

Before the `do` loop begins, I declare the character `again`. However, I don't need to initialize it because it's not tested until after the first iteration of the loop. I get a new value for `again` from the user in the loop body. Then I test `again` in the loop expression. If `again` is equal to `'y'`, the loop repeats; otherwise, the loop ends.

In the Real World

Even though you can use `while` and `do` loops pretty interchangeably, most programmers use the `while` loop. Although a `do` loop might seem more natural in some cases, the advantage of a `while` loop is that its expression appears right at the top of the loop; you don't have to go hunting to the bottom of the loop to find it.

Copyright © ${Date}. ${Publisher}. All rights reserved.

Trap

If you've ever had a game get stuck in the same endless cycle, you might have experienced an *infinite loop*—a loop without end. Here's a simple example of an infinite loop:

```
int test = 10;
while (test == 10)
{
    cout << test;
}
```

In this case, the loop is entered because `test` is `10`. But because `test` never changes, the loop will never stop. As a result, the user will have to kill the running program to end it. The moral of this story? Make sure that the expression of a loop can eventually become `false` or that there's another way for the loop to end, such as described in the following section, "Using break and continue Statements."

Using break and continue Statements

It's possible to alter the behavior you've seen in loops. You can immediately exit a loop with the `break` statement, and you can jump directly to the top of a loop with a `continue` statement. Although you should use these powers sparingly, they do come in handy sometimes.

Introducing the Finicky Counter Program

The Finicky Counter program counts from 1 to 10 through a `while` loop. It's finicky because it doesn't like the number 5—it skips it. Figure 2.8 shows a run of the program.

Figure 2.8
The number 5 is skipped with a `continue` statement, and the loop ends with a `break` statement.
Used with permission from Microsoft.

Copyright © ${Date}. ${Publisher}. All rights reserved.

You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 2 folder; the filename is `finicky_counter.cpp`.

```
// Finicky Counter
// Demonstrates break and continue statements

#include <iostream>

using namespace std;

int main()
{
    int count = 0;
    while (true)
    {
        count += 1;

        //end loop if count is greater than 10
        if (count > 10)
        {
            break;
        }

        //skip the number 5
        if (count == 5)
        {
            continue;
        }

        cout << count << endl;
    }

    return 0;
}
```

Creating a while (true) Loop

I set up the loop with the following line:

```
while (true)
```

Technically, this creates an infinite loop. This might seem odd coming so soon after a warning to avoid infinite loops, but this particular loop isn't really infinite because I put an exit condition in the loop body.

Copyright © ${Date}, ${Publisher}. All rights reserved.

Hint

Although a `while (true)` loop sometimes can be clearer than a traditional loop, you should also try to minimize your use of these loops.

Using the break Statement to Exit a Loop

This is the exit condition I put in the loop:

```
//end loop if count is greater than 10
if (count > 10)
{
    break;
}
```

Because `count` is increased by 1 each time the loop body begins, it will eventually reach 11. When it does, the `break` statement (which means "break out of the loop") is executed and the loop ends.

Using the continue Statement to Jump Back to the Top of a Loop

Just before `count` is displayed, I included the lines:

```
//skip the number 5
if (count == 5)
{
    continue;
}
```

The `continue` statement means "jump back to the top of the loop." At the top of the loop, the `while` expression is tested and the loop is entered again if it's true. So when `count` is equal to `5`, the program does not get to the `cout << count << endl;` statement. Instead, it goes right back to the top of the loop. As a result, 5 is skipped and never displayed.

Understanding When to Use break and continue

You can use `break` and `continue` in any loop you create; they aren't just for `while (true)` loops. But you should use them sparingly. Both `break` and `continue` can make it harder for programmers to see the flow of a loop.

Copyright © ${Date}, ${Publisher}. All rights reserved.

Using Logical Operators

So far you've seen fairly simple expressions evaluated for their truth or falsity. However, you can combine simpler expressions with *logical operators* to create more complex expressions. Table 2.2 lists the logical operators.

Table 2.2 Logical Operators

Operator	Description	Sample Expression
!	Logical NOT	`!expression`
&&	Logical AND	`expression1 && expression2`
||	Logical OR	`expression1 || expression2`

Introducing the Designers Network Program

The Designers Network program simulates a computer network in which only a select group of game designers are members. Like real-world computer systems, each member must enter a username and a password to log in. With a successful login, the member is personally greeted. To log in as a guest, all a user needs to do is enter `guest` at either the username or password prompt. Figures 2.9 through 2.11 show the program.

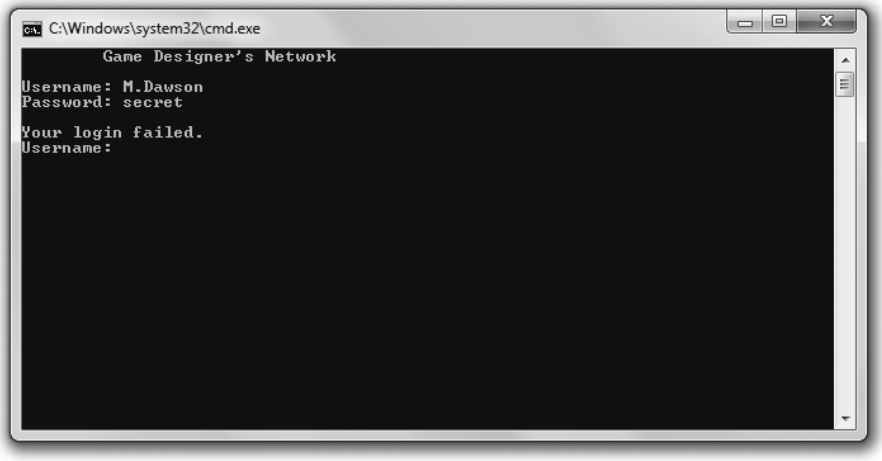


Figure 2.9
If you're not a member or a guest, you can't get in.
Used with permission from Microsoft.

Copyright © ${Date}. ${Publisher}. All rights reserved.

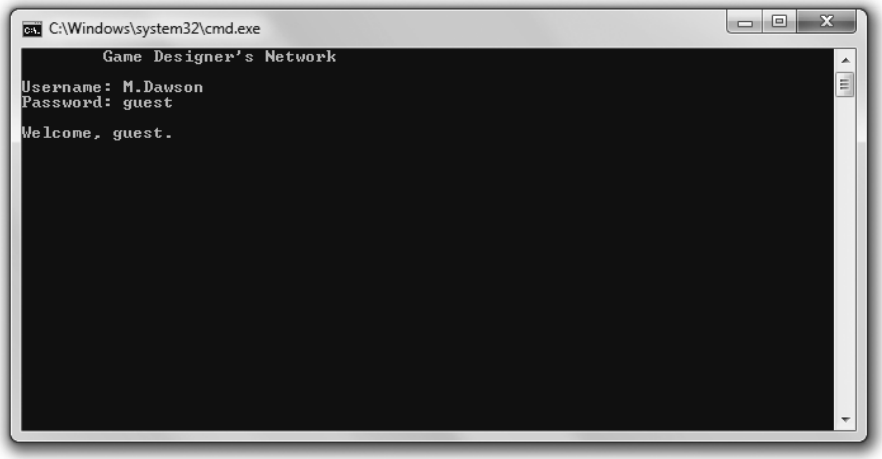


Figure 2.10
You can log in as a guest.
Used with permission from Microsoft.

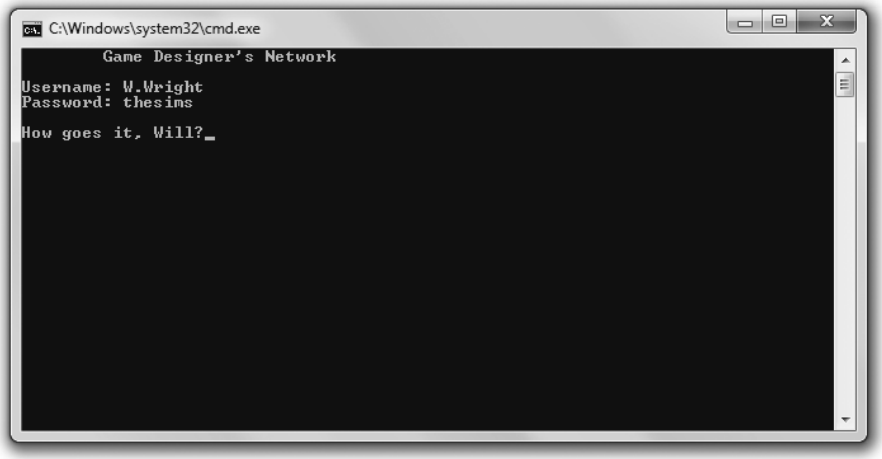


Figure 2.11
Looks like one of the elite logged in today.
Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 2 folder; the filename is `designers_network.cpp`.

Copyright © ${Date}. ${Publisher}. All rights reserved.

```
// Designers Network
// Demonstrates logical operators

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

int main()
{
    string username;
    string password;
    bool success;

    cout << "\tGame Designer's Network\n";

    do
    {
         cout << "\nUsername: ";
         cin >> username;

         cout << "Password: ";
         cin >> password;

        if (username == "S.Meier" && password == "civilization")
        {
            cout << "\nHey, Sid.";
            success = true;
        }

        else if (username == "S.Miyamoto" && password == "mariobros")
        {
            cout << "\nWhat's up, Shigeru?";
            success = true;
        }

        else if (username == "W.Wright" && password == "thesims")
        {
            cout << "\nHow goes it, Will?";
            success = true;
        }

        else if (username == "guest" || password == "guest")
        {
            cout << "\nWelcome, guest.";
            success = true;
        }

        else
```

Copyright © ${Date}. ${Publisher}. All rights reserved.

```
        {
            cout << "\nYour login failed.";
            success = false;
        }
    } while (!success);
    return 0;
}
```

Using the Logical AND Operator

The logical AND operator, `&&`, lets you join two expressions to form a larger one, which can be evaluated to `true` or `false`. The new expression is `true` only if the two expressions it joins are `true`; otherwise, it is `false`. Just as in English, "and" means both. Both original expressions must be `true` for the new expression to be `true`. Here's a concrete example from the Designers Network program:

```
if (username == "S.Meier" && password == "civilization")
```

The expression `username == "S.Meier" && password == "civilization"` is `true` only if both `username == "S.Meier"` and `password == "civilization"` are `true`. This works perfectly because I only want to grant Sid access if he enters both his username and his password. Just one or the other won't do.

Another way to understand how `&&` works is to look at all of the possible combinations of truth and falsity (see Table 2.3).

Table 2.3 Possible Login Combinations Using the AND Operator

username == "S.Meier"	password == "civilization"	username == "S.Meier" && password == "civilization"
true	true	true
true	false	false
false	true	false
false	false	false

Of course, the Designers Network program works for other users besides Sid Meier. Through a series of `if` statements with `else` clauses using the `&&` operator, the program checks three different `username` and `password` pairs. If a user enters a recognized pair, he is personally greeted.

Copyright © ${Date}. ${Publisher}. All rights reserved.

Using the Logical OR Operator

The logical OR operator, `||`, lets you join two expressions to form a larger one, which can be evaluated to `true` or `false`. The new expression is `true` if the first expression *or* the second expression is `true`; otherwise, it is `false`. Just as in English, "or" means either. If either the first or second expression is `true`, then the new expression is `true`. (If both are `true`, then the larger expression is still `true`.) Here's a concrete example from the Designers Network program:

```
else if (username == "guest" || password == "guest")
```

The expression `username == "guest" || password == "guest"` is `true` if `username == "guest"` is `true` or if `password == "guest"` is `true`. This works perfectly because I want to grant a user access as a guest as long as he enters `guest` for the username or password. If the user enters `guest` for both, that's fine too.

Another way to understand how `||` works is to look at all of the possible combinations of truth and falsity (see Table 2.4).

Table 2.4 Possible Login Combinations Using the OR Operator

username == "guest"	password == "guest"	username == "guest" || password == "guest"
true	true	true
true	false	true
false	true	true
false	false	false

Using the Logical NOT Operator

The logical NOT operator, `!`, lets you switch the truth or falsity of an expression. The new expression is `true` if the original is `false`; the new expression is `false` if the original is `true`. Just as in English, "not" means the opposite. The new expression has the opposite value of the original.

I use the NOT operator in the Boolean expression of the `do` loop:

```
} while (!success);
```

Copyright © ${Date}. ${Publisher}. All rights reserved.

The expression `!success` is `true` when `success` is `false`. That works perfectly because `success` is `false` only when there has been a failed login. In that case, the block associated with the `do` loop executes again and the user is asked for his username and password once more.

The expression `!success` is `false` when `success` is `true`. That works perfectly because when `success` is `true`, the user has successfully logged in and the loop ends.

Another way to understand how `!` works is to look at all of the possible combinations of truth and falsity (see Table 2.5).

Table 2.5 Possible Login Combinations Using the NOT Operator

security	!security
`true`	`false`
`false`	`true`

Understanding Order of Operations

Just like arithmetic operators, logical operators have precedence levels that affect the order in which an expression is evaluated. Logical NOT, `!`, has a higher level of precedence than logical AND, `&&`, which has a higher precedence than logical OR, `||`.

Just as with arithmetic operators, if you want an operation with lower precedence to be evaluated first, you can use parentheses. You can create complex expressions that involve arithmetic operators, relational operators, and logical operators. Operator precedence will define the exact order in which elements of the expression are evaluated. However, it's best to try to create expressions that are clear and simple rather than expressions that require a mastery of the operator precedence list to decipher.

For a list of C++ operators and their precedence levels, see Appendix B, "Operator Precedence."

Hint

Although you can use parentheses in a larger expression to change the way in which it's evaluated, you can also use *redundant parentheses*—parentheses that don't change the value of the expressions—to make the expression clearer. Let me give you a simple example. Check out the following expression from the Designers Network program:

```
(username == "S.Meier" && password == "civilization")
```

Copyright © ${Date}. ${Publisher}. All rights reserved.

Now, here's the expression with some redundant parentheses:

```
( (username == "S.Meier") && (password == "civilization") )
```

While the extra parentheses don't change the meaning of the expression, they really help the two smaller expressions, joined by the `&&` operator, stand out.

Using redundant parentheses is a bit of an art form. Are they helpful or just plain redundant? That's a call you as the programmer have to make.

Generating Random Numbers

A sense of unpredictability can add excitement to a game. Whether it's the sudden change in a computer opponent's strategy in an RTS (real-time strategy) or an alien creature bursting from an arbitrary door in an FPS (first-person shooter), players thrive on a certain level of surprise. Generating random numbers is one way to achieve this kind of surprise.

Introducing the Die Roller Program

The Die Roller program simulates the roll of a six-sided die. The computer calculates the roll by generating a random number. Figure 2.12 shows the results of the program.

Figure 2.12
The die roll is based on a random number generated by the program.
Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 2 folder; the filename is `die_roller.cpp`.

Copyright © ${Date}. ${Publisher}. All rights reserved.

```
// Die Roller
// Demonstrates generating random numbers

#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

int main()
{
    srand(static_cast<unsigned int>(time(0))); //seed random number generator

    int randomNumber = rand(); //generate random number

    int die = (randomNumber % 6) + 1; // get a number between 1 and 6
    cout << "You rolled a " << die << endl;

    return 0;
}
```

Calling the rand( ) Function

One of the first things I do in the program is include a new file:

```
#include <cstdlib>
```

The file `cstdlib` contains (among other things) functions that deal with generating random numbers. Because I've included the file, I'm free to call the functions it contains, including the function `rand()`, which is exactly what I do in `main()`:

```
    int randomNumber = rand(); //generate random number
```

As you learned in Chapter 1, functions are pieces of code that can do some work and return a value. You call or invoke a function by using its name followed by a pair of parentheses. If a function returns a value, you can assign that value to a variable. That's what I do here with my use of the assignment statement. I assign the value returned by `rand()` (a random number) to `randomNumber`.

Hint

The `rand()` function generates a random number between 0 and at least 32767. The exact upper limit depends on your implementation of C++. The upper limit is stored in the constant `RAND_MAX`, which is defined in `cstdlib`. So if you want to know the maximum random number `rand()` can generate, just send `RAND_MAX` to `cout`.

Copyright © ${Date}, ${Publisher}. All rights reserved.

Functions can also take values to use in their work. You provide these values by placing them between the parentheses after the function name, separated by commas. These values are called *arguments*, and when you provide them, you *pass* them to the function. I didn't pass any values to `rand()` because the function doesn't take any arguments.

Seeding the Random Number Generator

Computers generate *pseudorandom* numbers—not truly random numbers—based on a formula. One way to think about this is to imagine that the computer reads from a huge book of predetermined numbers. By reading from this book, the computer can appear to produce a sequence of random numbers.

But there's a problem: The computer always starts reading the book from the beginning. Because of this, the computer will always produce the same series of "random" numbers in a program. In games, this isn't something we'd want. We wouldn't, for example, want the same series of dice rolls in a game of craps every time we played.

A solution to this problem is to tell the computer to start reading from some arbitrary place in the book when a game program begins. This process is called *seeding* the random number generator. Game programmers give the random number generator a number, called a *seed*, to determine the starting place in this sequence of pseudorandom numbers.

The following code seeds the random number generator:

```
srand(static_cast<unsigned int>(time(0)));  //seed random number generator
```

Wow, that's a pretty cryptic looking line, but what it does is simple. It seeds the random number generator based on the current date and time, which is perfect since the current date and time will be different for each run of the program.

In terms of the actual code, the `srand()` function seeds the random number generator—you just have to pass it an `unsigned int` as a seed. What gets passed to the function here is the return value of `time(0)`—a number based on the current system date and time. The code `static_cast<unsigned int>` just converts (or *casts*) this value to an `unsigned int`. Now, you don't have to understand all the nuances of this line; the least you need to know is that if you want a program to generate a series of random numbers that are different

Copyright © ${Date}, ${Publisher}. All rights reserved.

each time the program is run, your program should execute this line once before making calls to `rand()`.

Hint

A comprehensive explanation of the various forms of casting a value from one type to another is beyond the scope of this book.

Calculating a Number within a Range

After generating a random number, `randomNumber` holds a value between 0 and 32767 (based on my implementation of C++). But I need a number between 1 and 6, so next I use the modulus operator to produce a number in that range.

```
int die = (randomNumber % 6) + 1; // get a number between 1 and 6
```

Any positive number divided by 6 will give a remainder between 0 and 5. In the preceding code, I take this remainder and add 1, giving me the possible range of 1 through 6—exactly what I wanted. You can use this technique to convert a random number to a number within a range you're looking for.

Trap

Using the modulus operator to create a number within a range from a random number might not always produce uniform results. Some numbers in the range might be more likely to appear than others. However, this isn't a problem for simple games.

Understanding the Game Loop

The *game loop* is a generalized representation of the flow of events in a game. The core of the events repeats, which is why it's called a loop. Although the implementation might be quite different from game to game, the fundamental structure is the same for almost all games across genres. Whether you're talking about a simple space shooter or a complex role-playing game (RPG), you can usually break the game down into the same repeating components of the game loop. Figure 2.13 provides a visual representation of the game loop.

Copyright © ${Date}. ${Publisher}. All rights reserved.

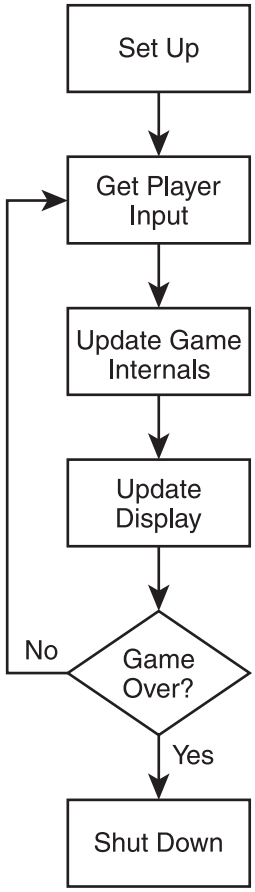


Figure 2.13
The game loop describes a basic flow of events that fits just about any game.

Here's an explanation of the parts of the game loop:

- **Setup.** This often involves accepting initial settings or loading game assets, such as sound, music, and graphics. The player might also be presented with the game backstory and his objectives.
- **Getting player input.** Whether it comes from the keyboard, mouse, joystick, trackball, or some other device, input from the player is captured.
- **Updating game internals.** The game logic and rules are applied to the game world, taking into account player input. This might take the shape of a physics system determining the interaction of objects or it might involve calculations of enemy AI, for example.

Copyright © ${Date}. ${Publisher}. All rights reserved.

- **Updating the display.** In the majority of games, this process is the most taxing on the computer hardware because it often involves drawing graphics. However, this process can be as simple as displaying a line of text.
- **Checking whether the game is over.** If the game isn't over (if the player's character is still alive and the player hasn't quit, for example), control branches back to the getting player input stage. If the game is over, control falls through to the shutting down stage.
- **Shutting down.** At this point, the game is over. The player is often given some final information, such as his score. The program frees any resources, if necessary, and exits.

Introducing Guess My Number

The final project for this chapter, Guess My Number, is the classic number-guessing game. For those who missed out on this game in their childhood, it goes like this: The computer chooses a random number between 1 and 100, and the player tries to guess the number in as few attempts as possible. Each time the player enters a guess, the computer tells him whether the guess is too high, too low, or right on the money. Once the player guesses the number, the game is over. Figure 2.14 shows Guess My Number in action. You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 2 folder; the filename is `guess_my_number.cpp`.

Figure 2.14
I guessed the computer's number in just three tries.
Used with permission from Microsoft.

Copyright © ${Date}. ${Publisher}. All rights reserved.

Applying the Game Loop

It's possible to examine even this simple game through the construct of the game loop. Figure 2.15 shows how nicely the game loop paradigm fits the flow of the game.

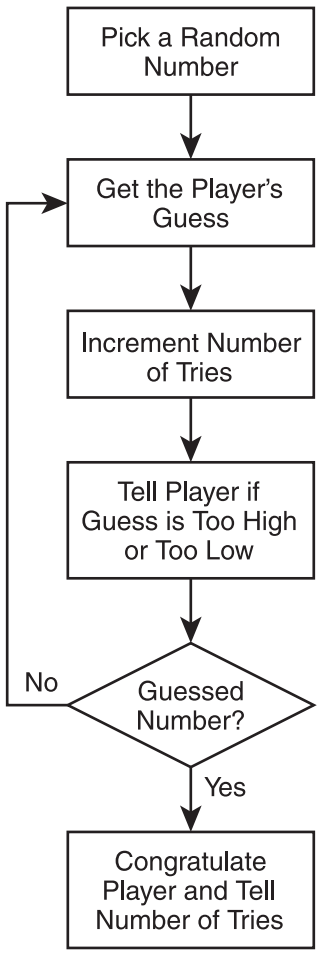


Figure 2.15
The game loop applied to Guess My Number.

Setting Up the Game

As always, I start off with some comments and include the necessary files.

```
// Guess My Number
// The classic number guessing game
```

Copyright © ${Date}. ${Publisher}. All rights reserved.

```
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;
```

I include `cstdlib` because I plan to generate a random number. I include `ctime` because I want to seed the random number generator with the current time.

Next, I start the `main()` function by picking a random number, setting the number of tries to 0, and establishing a variable for the player's guess:

```
int main()
{
    srand(static_cast<unsigned int>(time(0)));   //seed random number generator

    int secretNumber = rand() % 100 + 1;            // random number between 1 and 100
    int tries = 0;
    int guess;

    cout << "\tWelcome to Guess My Number\n\n";
```

Creating the Game Loop

Next, I write the game loop.

```
    do
    {
        cout << "Enter a guess: ";
        cin >> guess;
        ++tries;

        if (guess > secretNumber)
        {
            cout << "Too high!\n\n";
        }
        else if (guess < secretNumber)
        {
            cout << "Too low!\n\n";
        }
        else
        {
            cout << "\nThat's it! You got it in " << tries << " guesses!\n";
        }
    } while (guess != secretNumber);
```

Copyright © ${Date}. ${Publisher}. All rights reserved.

I get the player's guess, increment the number of tries, and then tell the player if his guess is too high, too low, or right on the money. If the player's guess is correct, the loop ends. Notice that the `if` statements are nested inside the `while` loop.

Wrapping Up the Game

Once the player has guessed the secret number, the loop and game are over. All that's left to do is end the program.

```
    return 0;
}
```

Summary

In this chapter, you learned the following concepts:

- You can use the truth or falsity of an expression to branch to (or skip) sections of code.
- You can represent truth or falsity with the keywords, `true` and `false`.
- You can evaluate any value or expression for truth or falsity.
- Any non-zero value can be interpreted as `true`, while 0 can be interpreted as `false`.
- A common way to create an expression to be evaluated as `true` or `false` is to compare values with the relational operators.
- The `if` statement tests an expression and executes a section of code only if the expression is `true`.
- The `else` clause of an `if` statement specifies code that should be executed only if the expression tested in the `if` statement is `false`.
- The `switch` statement tests a value that can be treated as an `int` and executes a section of code labeled with the corresponding value.
- The `default` keyword, when used in a `switch` statement, specifies code to be executed if the value tested in the `switch` statement matches no listed values.
- The `while` loop executes a section of code if an expression is `true` and repeats the code as long as the expression is `true`.
- A `do` loop executes a section of code and then repeats the code as long as the expression is `true`.

Copyright © ${Date}. ${Publisher}. All rights reserved.

- Used in a loop, the `break` statement immediately ends the loop.
- Used in a loop, the `continue` statement immediately causes the control of the program to branch to the top of the loop.
- The `&&` (AND) operator combines two simpler expressions to create a new expression that is `true` only if both simpler expressions are `true`.
- The `||` (OR) operator combines two simpler expressions to create a new expression that is `true` if either simpler expression is `true`.
- The `!` (NOT) operator creates a new expression that is the opposite truth value of the original.
- The game loop is a generalized representation of the flow of events in a game, the core of which repeats.
- The file `cstdlib` contains functions that deal with generating random numbers.
- The function `srand()`, defined in `cstdlib`, seeds the random number generator.
- The function `rand()`, defined in `cstdlib`, returns a random number.

Questions and Answers

Q: Do you have to use the keywords `true` and `false`?
A: No, but it's a good idea. Before the advent of the keywords `true` and `false`, programmers often used 1 to represent true and 0 to represent false. However, now that `true` and `false` are available, it's best to use them instead of the old-fashioned 1 and 0.

Q: Can you assign a `bool` variable something other than `true` or `false`?
A: Yes. You can assign an expression to a `bool` variable, which will store the truth or falsity of the expression.

Q: Can you use a `switch` statement to test some non-integer value?
A: No. `switch` statements only work with values that can be interpreted as integers (including `char` values).

Q: How can you test a single non-integer value against multiple values if you can't use a `switch` statement?
A: You can use a series of `if` statements.

Q: What's an infinite loop?
A: A loop that will never end, regardless of user input.

Copyright © ${Date}. ${Publisher}. All rights reserved.

Q: Why are infinite loops considered bad?
A: Because a program stuck in an infinite loop will never end on its own. It has to be shut down by the operating system. In the worst case, a user will have to shut his computer off to end a program stuck in an infinite loop.

Q: Won't a compiler catch an infinite loop and flag it as an error?
A: No. An infinite loop is a logical error—the kind of error a programmer must track down.

Q: If infinite loops are a bad thing, then isn't a `while (true)` loop a bad thing?
A: No. When a programmer creates a `while (true)` loop, he should provide a way for the loop to end (usually through a `break` statement).

Q: Why would a programmer create a `while (true)` loop?
A: `while (true)` loops are often used for the main loop of a program, like the game loop.

Q: Why do some people feel that using a `break` statement to exit a loop is poor programming?
A: Because indiscriminate use of `break` statements can make it hard to understand the conditions under which a loop ends. However, sometimes the use of a `while (true)` loop along with a `break` statement can be clearer than creating the same loop in a more traditional way.

Q: What's a pseudorandom number?
A: A random number that's usually generated by a formula. As a result, a series of pseudorandom numbers is not truly random, but good enough for most purposes.

Q: What is seeding a random number generator?
A: It's giving the random number generator a seed, such as an integer, which affects the way the generator produces random numbers. If you don't seed a random number generator, it will produce the same series of numbers each time it's run from the beginning of a program.

Q: Don't you always want to seed the random number generator before using it?
A: Not necessarily. You might want a program to produce the exact same sequence of "random" numbers each time it runs for testing purposes, for example.

Copyright © ${Date}. ${Publisher}. All rights reserved.

Q: How can I generate more truly random numbers?
A: There are third-party libraries that produce better pseudorandom numbers than the ones that typically come with C++ compilers.

Q: Do all games use the game loop?
A: The game loop is just a way of looking at a typical game's flow of events. And just because this paradigm fits a particular game, that doesn't necessarily mean that the game is implemented with a loop around the bulk of its code.

Discussion Questions

1. What kinds of things would be difficult to program without loops?
2. What are the advantages and disadvantages of the `switch` statement versus a series of `if` statements?
3. When might you omit a `break` statement from the end of a `case` in a `switch` statement?
4. When should you use a `while` loop over a `do` loop?
5. Describe your favorite game in terms of the game loop. Is the game loop a good fit?

Exercises

1. Rewrite the Menu Chooser program from this chapter using an enumeration to represent difficulty levels. The variable choice will still be of type `int`.
2. What's wrong with the following loop?

```
int x = 0;
while (x)
{
    ++x;
    cout << x << endl;
}
```

3. Write a new version of the Guess My Number program in which the player and the computer switch roles. That is, the player picks a number and the computer must guess what it is.

Copyright © ${Date}. ${Publisher}. All rights reserved.

This page intentionally left blank

Copyright © ${Date}. ${Publisher}. All rights reserved.