Chapter 3

for Loops, Strings, and Arrays: Word Jumble

You've seen how to work with single values, but in this chapter you'll learn how to work with sequences of data. You'll learn more about strings—objects for sequences of characters. You'll also see how to work with sequences of any type. And you'll discover a new type of loop that's perfect for use with these sequences. Specifically, you'll learn to:

- Use `for` loops to iterate over sequences
- Use objects, which combine data and functions
- Use `string` objects and their member functions to work with sequences of characters
- Use arrays to store, access, and manipulate sequences of any type
- Use multidimensional arrays to better represent certain collections of data

Using for Loops

You met one type of loop in Chapter 2, "Truth, Branching, and the Game Loop: Guess My Number,"—the `while` loop. Well, it's time to meet another—the `for` loop. Like its cousin the `while` loop, the `for` loop lets you repeat a section of code, but `for` loops are particularly suited for counting and moving through a sequence of things (like the items in an RPG character's inventory).

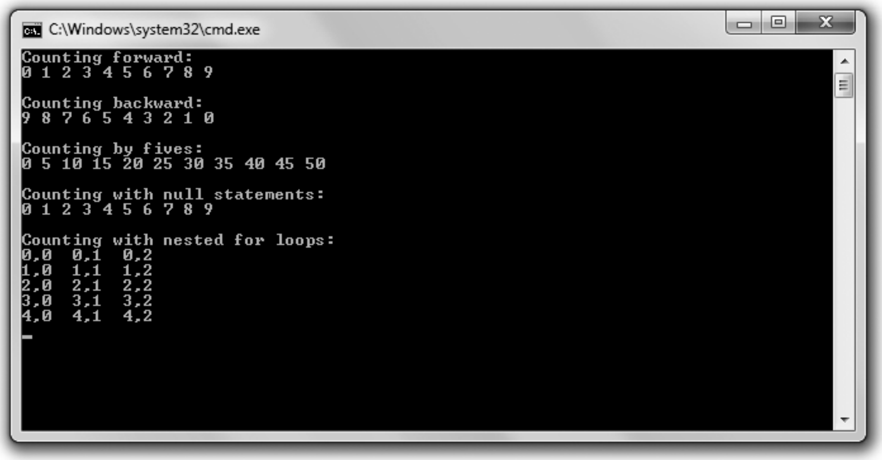
Here's the generic form of `for` loop:

```
for (initialization; test; action)
    statement;
```

`initialization` is a statement that sets up some initial condition for the loop. (For example, it might set a counter variable to 0.) The expression `test` is tested each time before the loop body executes, just as in a `while` loop. If `test` is `false`, the program moves on to the statement after the loop. If `test` is true, the program executes `statement`. Next, `action` is executed (which often involves incrementing a counter variable). The cycle repeats until `test` is `false`, at which point the loop ends.

Introducing the Counter Program

The Counter program counts forward, backward, and by fives. It even counts out a grid with rows and columns. It accomplishes all of this using `for` loops. Figure 3.1 shows the program in action.



```
Counting forward:
0 1 2 3 4 5 6 7 8 9

Counting backward:
9 8 7 6 5 4 3 2 1 0

Counting by fives:
0 5 10 15 20 25 30 35 40 45 50

Counting with null statements:
0 1 2 3 4 5 6 7 8 9

Counting with nested for loops:
0,0  0,1  0,2
1,0  1,1  1,2
2,0  2,1  2,2
3,0  3,1  3,2
4,0  4,1  4,2
```

Figure 3.1
`for` loops do all of the counting, while a pair of nested `for` loops displays the grid.
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 3 folder; the filename is `counter.cpp`.

```
// Counter
// Demonstrates for loops

#include <iostream>

using namespace std;
```

```
int main()
{
    cout << "Counting forward:\n";
    for (int i = 0; i < 10; ++i)
    {
        cout << i << " ";
    }

    cout << "\n\nCounting backward:\n";
    for (int i = 9; i >= 0; --i)
    {
        cout << i << " ";
    }

    cout << "\n\nCounting by fives:\n";
    for (int i = 0; i <= 50; i += 5)
    {
        cout << i << " ";
    }

    cout << "\n\nCounting with null statements:\n";
    int count = 0;
    for ( ; count < 10; )
    {
        cout << count << " ";
        ++count;
    }

    cout << "\n\nCounting with nested for loops:\n";
    const int ROWS = 5;
    const int COLUMNS = 3;
    for (int i = 0; i < ROWS; ++i)
    {
        for (int j = 0; j < COLUMNS; ++j)
        {
            cout << i << "," << j << " ";
        }

        cout << endl;
    }

    return 0;
}
```

Trap

If you're using an older compiler that doesn't fully implement the current C++ standard, when you try to compile this program, you might get an error that says something like: `error: 'i' : redefinition; multiple initialization`.

The best solution is to use a modern, compliant compiler. Luckily, if you're running Windows, you can download the popular (and free) Microsoft Visual Studio Express 2013 for Windows Desktop, which includes a modern compiler, from www.visualstudio.com/downloads/download-visual-studio-vs.

If you must use your old compiler, you should declare any `for` loop counter variables just once for all `for` loops in a scope. I cover the topic of scopes in Chapter 5, "Functions: Mad Lib."

Counting with for Loops

The first `for` loop counts from 0 to 9. The loop begins:

```
for (int i = 0; i < 10; ++i)
```

The initialization statement, `int i = 0`, declares `i` and initializes it to 0. The expression `i < 10` says that the loop will continue as long as `i` is less than 10. Lastly, the action statement, `++i`, says `i` is to be incremented each time the loop body finishes. As a result, the loop *iterates* 10 times—once for each of the values 0 through 9. And during each iteration, the loop body displays the value of `i`.

The next `for` loop counts from 9 down to 0. The loop begins:

```
for (int i = 9; i >= 0; --i)
```

Here, `i` is initialized to 9, and the loop continues as long as `i` is greater than or equal to 0. Each time the loop body finishes, `i` is decremented. As a result, the loop displays the values 9 through 0.

The next loop counts from 0 to 50, by fives. The loop begins:

```
for (int i = 0; i <= 50; i += 5)
```

Here, `i` is initialized to 0, and the loop continues as long as `i` is less than or equal to 50. But notice the action statement, `i += 5`. This statement increases `i` by five each time the loop body finishes. As a result, the loop displays the values 0, 5, 10, 15, and so on. The expression `i <= 50` says to execute the loop body as long as `i` is less than or equal to 50.

You can initialize a counter variable, create a test condition, and update the counter variable with any values you want. However, the most common thing to do is to start the counter at 0 and increment it by 1 after each loop iteration.

Finally, the caveats regarding infinite loops that you learned about while studying `while` loops apply equally well to `for` loops. Make sure you create loops that can end; otherwise, you'll have a very unhappy gamer on your hands.

Using Empty Statements in for Loops

You can use empty statements in creating your `for` loop, as I did in the following loop:

```
for ( ; count < 10; )
```

I used an empty statement for the initialization and action statements. That's fine because I declared and initialized `count` before the loop and incremented it inside the loop body. This loop displays the same sequence of integers as the very first loop in the program. Although the loop might look odd, it's perfectly legal.

Hint

Different game programmers have different traditions. In the last chapter, you saw that you can create a loop that continues until it reaches an exit statement—such as a `break`—using `while (true)`. Well, some programmers prefer to create these kinds of loops using a `for` statement that begins with `for (;;)`. Because the test expression in this loop is the empty statement, the loop will continue until it encounters some exit statement.

Nesting for Loops

You can nest `for` loops by putting one inside the other. That's what I did in the following section of code, which counts out the elements of a grid. The outer loop, which begins:

```
for (int i = 0; i < ROWS; ++i)
```

simply executes its loop body `ROWS` (five) times. But it just so happens that there's another `for` loop inside this loop, which begins:

```
for (int j = 0; j < COLUMNS; ++j)
```

As a result, the inner loop executes in full for each iteration of the outer loop. In this case, that means the inner loop executes `COLUMNS` (three) times, for the `ROWS` (five) times the outer loop iterates, for a total of 15 times. Specifically, here's what happens:

1. The outer `for` loop declares `i` and initializes it to 0. Since `i` is less than `ROWS` (5), the program enters the outer loop's body.
2. The inner loop declares `j` and initializes it to 0. Since `j` is less than `COLUMNS` (3), the program enters its loop body, sending the values of `i` and `j` to `cout`, which displays 0, 0.

3. The program reaches the end of the body of the inner loop and increments `j` to 1. Since `j` is still less than `COLUMNS` (3), the program executes the inner loop's body again, displaying 0, 1.
4. The program reaches the end of the inner loop's body and increments `j` to 2. Since `j` is still less than `COLUMNS` (3), the program executes the inner loop's body again, displaying 0, 2.
5. The program reaches the end of the inner loop's body and increments `j` to 3. This time, however, `j` is not less than `COLUMNS` (3) and the inner loop ends.
6. The program finishes the first iteration of the outer loop by sending `endl` to `cout`, ending the first row.
7. The program reaches the end of the outer loop's body and increments `i` to 1. Since `i` is less than `ROWS` (5), the program enters the outer loop's body again.
8. The program reaches the inner loop, which starts from the beginning once again, by declaring and initializing `j` to 0. The program goes through the process described in Steps 2 through 7, displaying the second row of the grid. This process continues until all five rows have been displayed.

Again, the important thing to remember is that the inner loop is executed in full for each iteration of the outer loop.

Understanding Objects

So far, you've seen how to store individual pieces of information in variables and how to manipulate those variables using operators and functions. But most of the things you want to represent in games—such as, say, an alien spacecraft—are objects. They're encapsulated, cohesive things that combine qualities (such as an energy level) and abilities (for example, firing weapons). Often it makes no sense to talk about the individual qualities and abilities in isolation from each other.

Fortunately, most modern programming languages let you work with software objects (often just called *objects*) that combine data and functions. A data element of an object is called a *data member*, while a function of an object is called a *member function*. As a concrete example, think about that alien spacecraft. An alien spacecraft object might be of a new type called `Spacecraft`, defined by a game programmer, and might have a data member for its energy level and a member function to fire its weapons. In practice, an object's energy level might be stored in its data member `energy` as an `int`, and its ability to fire its weapons might be defined in a member function called `fireWeapons()`.

Every object of the same type has the same basic structure, so each object will have the same set of data members and member functions. However, as an individual, each object will have its own values for its data members. If you had a squadron of five alien spacecrafts, each would have its own energy level. One might have an energy level of 75, while another might have an energy level of only 10, and so on. Even if two crafts have the same energy level, each would belong to a unique spacecraft. Each craft could also fire its own weapons with a call to its member function, `fireWeapons()`. Figure 3.2 illustrates the concept of an alien spacecraft.

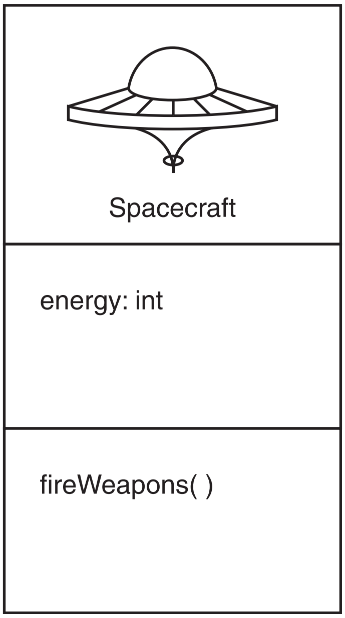


Figure 3.2
This representation of the definition of an alien spacecraft says that each object will have a data member called `energy` and a member function called `fireWeapons()`.

The cool thing about objects is that you don't need to know the implementation details to use them—just as you don't need to know how to build a car in order to drive one. You only have to know the object's data members and member functions—just as you only need to know where a car's steering wheel, gas pedal, and brake pedal are located.

You can store objects in variables, just like with built-in types. Therefore, you could store an alien spacecraft object in a variable of the `Spacecraft` type. You can access data members and member functions using the member selection operator (`.`), by placing the operator after the variable name of the object. So if you want your alien spacecraft, `ship`, to fire its weapons only if its energy level is greater than `10`, you could write:

```
// ship is an object of Spacecraft type
if (ship.energy > 10)
{
    ship.fireWeapons()
}
```

`ship.energy` accesses the object's `energy` data member, while `ship.fireWeapons()` calls the object's `fireWeapons()` member function.

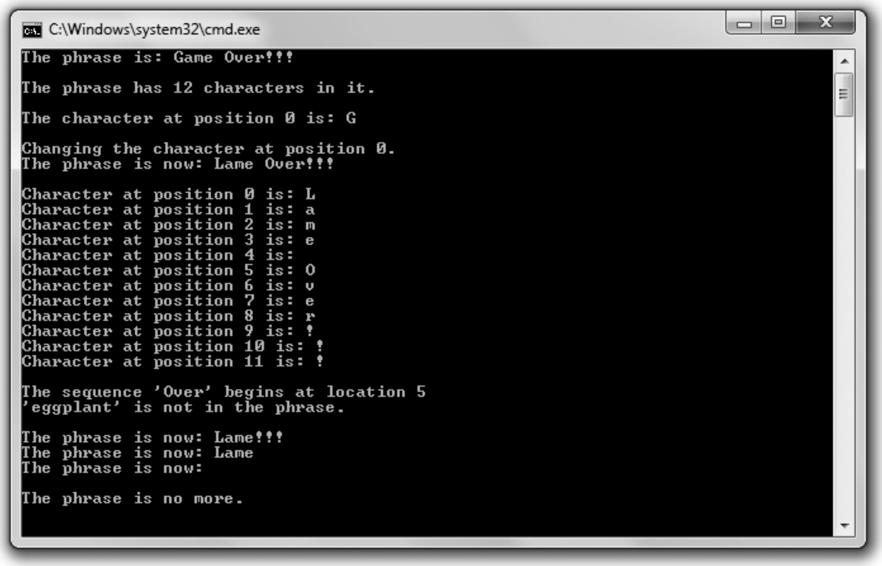
Although you can't make your own new types (like for an alien spacecraft) just yet, you can work with previously defined object types. And that's next on the agenda.

Using string Objects

`string` objects, which you met briefly in Chapter 1, "Types, Variables, and Standard I/O: Lost Fortune," are the perfect way to work with sequences of characters, whether you're writing a complete word puzzle game or simply storing a player's name. A `string` is actually an object, and it provides its own set of member functions that allow you to do a range of things with the `string` object—everything from simply getting its length to performing complex character substitutions. In addition, strings are defined so that they work intuitively with a few of the operators you already know.

Introducing the String Tester Program

The String Tester program uses the `string` object equal to `"Game Over!!!"` and tells you its length, the index (position number) of each character, and whether or not certain substrings can be found in it. In addition, the program erases parts of the `string` object. Figure 3.3 shows the results of the program.



```
The phrase is: Game Over!!!

The phrase has 12 characters in it.

The character at position 0 is: G

Changing the character at position 0.
The phrase is now: Lame Over!!!

Character at position 0 is: L
Character at position 1 is: a
Character at position 2 is: m
Character at position 3 is: e
Character at position 4 is:
Character at position 5 is: O
Character at position 6 is: v
Character at position 7 is: e
Character at position 8 is: r
Character at position 9 is: !
Character at position 10 is: !
Character at position 11 is: !

The sequence 'Over' begins at location 5
'eggplant' is not in the phrase.

The phrase is now: Lame!!!
The phrase is now: Lame
The phrase is now:

The phrase is no more.
```

Figure 3.3
`String` objects are combined, changed, and erased through familiar operators and `string` member functions.
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 3 folder; the filename is `string_tester.cpp`.

```
// String Tester
// Demonstrates string objects

#include <iostream>
#include <string>

using namespace std;

int main()
{
    string word1 = "Game";
    string word2("Over");
    string word3(3, '!');

    string phrase = word1 + " " + word2 + word3;
    cout << "The phrase is: " << phrase << "\n\n";

    cout << "The phrase has " << phrase.size() << " characters in it.\n\n";
    cout << "The character at position 0 is: " << phrase[0] << "\n\n";

    cout << "Changing the character at position 0.\n";
    phrase[0] = 'L';
    cout << "The phrase is now: " << phrase << "\n\n";

    for (unsigned int i = 0; i < phrase.size(); ++i)
    {
        cout << "Character at position " << i << " is: " << phrase[i] << endl;
    }

    cout << "\nThe sequence 'Over' begins at location ";
    cout << phrase.find("Over") << endl;

    if (phrase.find("eggplant") == string::npos)
    {
        cout << "'eggplant' is not in the phrase.\n\n";
    }

    phrase.erase(4, 5);
    cout << "The phrase is now: " << phrase << endl;

    phrase.erase(4);
    cout << "The phrase is now: " << phrase << endl;

    phrase.erase();
    cout << "The phrase is now: " << phrase << endl;

    if (phrase.empty())
    {
        cout << "\nThe phrase is no more.\n";
    }

    return 0;
}
```

Creating string Objects

The first thing I do in `main()` is create three strings in three different ways:

```
    string word1 = "Game";
    string word2("Over");
    string word3(3, '!');
```

In the first line of this group, I simply create the `string` object `word1` using the assignment operator in the same way you've seen for other variables. As a result, `word1` is `"Game"`.

Next, I create `word2` by placing the `string` object to which I want the variable set between a pair of parentheses. As a result, `word2` is `"Over"`.

Finally, I create `word3` by supplying between a pair of parentheses a number followed by a single character. This produces a `string` object made up of the provided character, which has a length equal to the number. As a result, `word3` is `"!!!"`.

Concatenating string Objects

Next, I create a new `string` object, `phrase`, by concatenating the first three `string` objects:

```
string phrase = word1 + " " + word2 + word3;
```

As a result, phrase is "Game Over!!!".

Notice that the + operator, which you've seen work only with numbers, also concatenates `string` objects. That's because the + operator has been overloaded. Now, when you first hear the term *overloaded*, you might think it's a bad thing—the operator is about to blow! But it's a good thing. Operator overloading redefines a familiar operator so it works differently when used in a new, previously undefined context. In this case, I use the + operator not to add numbers but to join `string` objects. I'm able to do this only because the `string` type specifically overloads the + operator and defines it so the operator means `string` object concatenation when used with strings.

Using the size( ) Member Function

Okay, it's time to take a look at a `string` member function. Next, I use the member function `size()`:

```
cout << "The phrase has " << phrase.size() << " characters in it.\n\n";
```

`phrase.size()` calls the member function `size()` of the `string` object `phrase` through the member selection operator . (the dot). The `size()` member function simply returns an unsigned integer value of the size of the `string` object—its number of characters. Because the `string` object is `"Game Over!!!"`, the member function returns 12. (Every character counts, including spaces.) Of course, calling `size()` for another `string` object might return a different result based on the number of characters in the `string` object.

Hint

`string` objects also have a member function `length()`, which, just like `size()`, returns the number of characters in the `string` object.

Indexing a string Object

A `string` object stores a sequence of `char` values. You can access any individual `char` value by providing an index number with the subscripting operator (`[]`). That's what I do next:

```
cout << "The character at position 0 is: " << phrase[0] << "\n\n";
```

The first element in a sequence is at position 0. In the previous statement, `phrase[0]` is the character G. And because counting begins at 0, the last character in the `string` object is `phrase[11]`, even though the `string` object has 12 characters in it.

Trap

It's a common mistake to forget that indexing begins at position 0. Remember, a `string` object with *n* characters in it can be indexed from position 0 to position *n*–1.

Not only can you access characters in a `string` object with the subscripting operator, but you can also reassign them. That's what I do next:

```
phrase[0] = 'L';
```

I change the first character of `phrase` to the character L, which means `phrase` becomes `"Lame Over!!!"`

Trap

C++ compilers do not perform bounds checking when working with `string` objects and the subscripting operator. This means that the compiler doesn't check to see whether you're attempting to access an element that doesn't exist. Accessing an invalid sequence element can lead to disastrous results because it's possible to write over critical data in your computer's memory. By doing this, you can crash your program, so take care when using the subscripting operator.

Iterating through string Objects

Given your new knowledge of `for` loops and `string` objects, it's a snap to iterate through the individual characters of a `string` object. That's what I do next:

```
for (unsigned int i = 0; i < phrase.size(); ++i)
{
    cout << "Character at position " << i << " is: " << phrase[i] << endl;
}
```

The loop iterates through all of the valid positions of phrase. It starts with 0 and goes through 11. During each iteration, a character of the string object is displayed with phrase[i]. Note that I made the loop variable, i, an unsigned int because the value returned by size() is an unsigned integral type.

In the Real World

Iterating through a sequence is a powerful and often-used technique in games. You might, for example, iterate through hundreds of individual units in a strategy game, updating their status and order. Or you might iterate through the list of vertices of a 3D model to apply some geometric transformation.

Using the find( ) Member Function

Next, I use the member function find() to check whether either of two string literals are contained in phrase. First, I check for the string literal "Over":

```
cout << "\nThe sequence 'Over' begins at location ";
cout << phrase.find("Over") << endl;
```

The find() member function searches the calling string object for the string supplied as an argument. The member function returns the position number of the first occurrence where the string object for which you are searching begins in the calling string object. This means that phrase.find("Over") returns the position number where the first occurrence of "Over" begins in phrase. Since phrase is "Lame Over!!!", find() returns 5. (Remember, position numbers begin at 0, so 5 means the sixth character.)

But what if the string for which you are searching doesn't exist in the calling string? I tackle that situation next:

```
if (phrase.find("eggplant") == string::npos)
{
    cout << "'eggplant' is not in the phrase.\n\n";
}
```

Because "eggplant" does not exist in phrase, find() returns a special constant defined in the file string, which I access with string::npos. As a result, the screen displays the message, "'eggplant' is not in the phrase."

The constant I access through string::npos represents the largest possible size of a string object, so it is greater than any possible valid position number in a string object. Informally, it means "a position number that can't exist." It's the perfect return value to indicate that one string couldn't be found in another.

Hint

When using `find()`, you can supply an optional argument that specifies a character number for the program to start looking for the substring. The following line will start looking for the string literal `"eggplant"` beginning at position 5 in the `string` object `phrase`.

```
location = phrase.find("eggplant", 5);
```

Using the erase( ) Member Function

The `erase()` member function removes a specified substring from a `string` object. One way to call the member function is to specify the beginning position and the length of the substring, as I did in this code:

```
phrase.erase(4, 5);
```

The previous line removes the five-character substring starting at position 4. Because `phrase` is `"Lame Over!!!"`, the member function removes the substring `Over` and, as a result, `phrase` becomes `"Lame!!!"`.

Another way to call `erase()` is to supply just the beginning position of the substring. This removes all of the characters starting at that position number to the end of the `string` object. That's what I do next:

```
phrase.erase(4);
```

This line removes all of the characters of the `string` object starting at position 4. Since `phrase` is `"Lame!!!"`, the member function removes the substring `!!!` and, as a result, `phrase` becomes `"Lame"`.

Yet another way to call `erase()` is to supply no arguments, as I did in this code:

```
phrase.erase();
```

The previous line erases every character in `phrase`. As a result, `phrase` becomes the empty string, which is equal to `""`.

Using the empty( ) Member Function

The `empty()` member function returns a `bool` value—`true` if the `string` object is empty and `false` otherwise. I use `empty()` in the following code:

```
if (phrase.empty())
{
    cout << "\nThe phrase is no more.\n";
}
```

Because `phrase` is equal to the empty string, `phrase().empty` returns `true`, and the screen displays the message, "The phrase is no more."

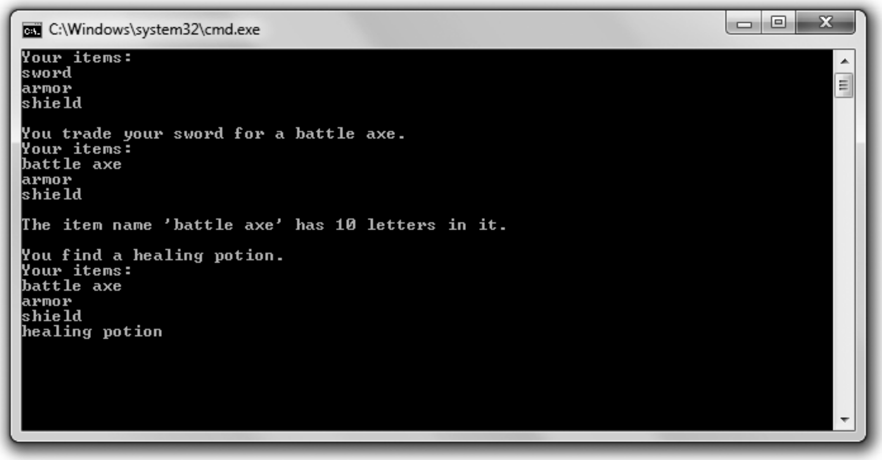
Using Arrays

While `string` objects provide a great way to work with a sequence of characters, arrays provide a way to work with elements of any type. That means you can use an array to store a sequence of integers for, say, a high-score list. But it also means that you can use arrays to store elements of programmer-defined types, such as a sequence of items that an RPG character might carry.

Introducing the Hero's Inventory Program

The Hero's Inventory program maintains the inventory of a hero from a typical RPG. Like in most RPGs, the hero is from a small, insignificant village, and his father was killed by an evil warlord. (What's a quest without a dead father?) Now that the hero has come of age, it's time for him to seek his revenge.

In this program, the hero's inventory is represented by an array. The array is a sequence of `string` objects—one for each item in the hero's possession. The hero trades and even finds new items. Figure 3.4 shows the program in action.



```
Your items:
sword
armor
shield

You trade your sword for a battle axe.
Your items:
battle axe
armor
shield

The item name 'battle axe' has 10 letters in it.

You find a healing potion.
Your items:
battle axe
armor
shield
healing potion
```

Figure 3.4
The hero's inventory is a sequence of `string` objects stored in an array.
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 3 folder; the filename is `heros_inventory.cpp`.

```
// Hero's Inventory
// Demonstrates arrays

#include <iostream>
#include <string>

using namespace std;

int main()
{
    const int MAX_ITEMS = 10;
    string inventory[MAX_ITEMS];

    int numItems = 0;
    inventory[numItems++] = "sword";
    inventory[numItems++] = "armor";
    inventory[numItems++] = "shield";

    cout << "Your items:\n";
    for (int i = 0; i < numItems; ++i)
    {
        cout << inventory[i] << endl;
    }

    cout << "\nYou trade your sword for a battle axe.";
    inventory[0] = "battle axe";
    cout << "\nYour items:\n";
    for (int i = 0; i < numItems; ++i)
    {
        cout << inventory[i] << endl;
    }

    cout << "\nThe item name '" << inventory[0] << "' has ";
    cout << inventory[0].size() << " letters in it.\n";

    cout << "\nYou find a healing potion.";
    if (numItems < MAX_ITEMS)
    {
        inventory[numItems++] = "healing potion";
    }
    else
    {
        cout << "You have too many items and can't carry another.";
    }
    cout << "\nYour items:\n";
    for (int i = 0; i < numItems; ++i)
    {
        cout << inventory[i] << endl;
    }

    return 0;
}
```

Creating Arrays

It's often a good idea to define a constant for the number of elements in an array. That's what I did with `MAX_ITEMS`, which represents the maximum number of items the hero can carry:

```
const int MAX_ITEMS = 10;
```

You declare an array much the same way you would declare any variable you've seen so far: You provide a type followed by a name. In addition, your compiler must know the size of the array so it can reserve the necessary memory space. You can provide that information following the array name, surrounded by square brackets. Here's how I declare the array for the hero's inventory:

```
string inventory[MAX_ITEMS];
```

The preceding code declares an array `inventory` of `MAX_ITEMS` `string` objects. (Because `MAX_ITEMS` is 10, that means 10 `string` objects.)

Trick

You can initialize an array with values when you declare it by providing an *`initializer list`*—a sequence of elements separated by commas and surrounded by curly braces. Here's an example:

```
string inventory[MAX_ITEMS] = {"sword", "armor", "shield"};
```

The preceding code declares an array of `string` objects, `inventory`, that has a size of `MAX_ITEMS`. The first three elements of the array are initialized to `"sword"`, `"armor"`, and `"shield"`.

If you omit the number of elements when using an initializer list, the array will be created with a size equal to the number of elements in the list. Here's an example:

```
string inventory[] = {"sword", "armor", "shield"};
```

Because there are three elements in the initializer list, the preceding line creates an array, `inventory`, that is three elements in size. Its elements are `"sword"`, `"armor"`, and `"shield"`.

Indexing Arrays

You index arrays much like you index `string` objects. You can access any individual element by providing an index number with the subscripting operator (`[]`).

Next, I add three items to the hero's inventory using the subscripting operator:

```
int numItems = 0;
inventory[numItems++] = "sword";
inventory[numItems++] = "armor";
inventory[numItems++] = "shield";
```

I start by defining `numItems` for the number of items the hero is carrying at the moment. Next, I assign `"sword"` to position 0 of the array. Because I use the postfix increment operator, `numItems` is incremented after the assignment to the array. The next two lines add `"armor"` and `"shield"` to the array, leaving `numItems` at the correct value of 3 when the code finishes.

Now that the hero is stocked with some items, I display his inventory:

```
cout << "Your items:\n";
for (int i = 0; i < numItems; ++i)
{
    cout << inventory[i] << endl;
}
```

This should remind you of string indexing. The code loops through the first three elements of `inventory`, displaying each `string` object in order.

Next, the hero trades his sword for a battle axe. I accomplish this through the following line:

```
inventory[0] = "battle axe";
```

The previous code reassigns the element at position 0 in `inventory` the `string` object `"battle axe"`. Now the first three elements of `inventory` are `"battle axe"`, `"armor"`, and `"shield"`.

Trap

Array indexing begins at 0, just as you saw with `string` objects. This means that the following code defines a five-element array.

```
int highScores[5];
```

Valid position numbers are 0 through 4, inclusive. There is no element `highScores[5]`! An attempt to access `highScores[5]` could lead to disastrous results, including a program crash.

Accessing Member Functions of an Array Element

You can access the member functions of an array element by writing the array element, followed by the member selection operator, followed by the member function name. This sounds a bit complicated, but it's not. Here's an example:

```
cout << inventory[0].size() << " letters in it.\n";
```

The code `inventory[0].size()` means the program should call the `size()` member function of the element `inventory[0]`. In this case, because `inventory[0]` is `"battle axe"`, the call returns 10, the number of characters in the `string` object.

Being Aware of Array Bounds

As you learned, you have to be careful when you index an array. Because an array has a fixed size, you can create an integer constant to store the size of an array. Again, that's just what I did in the beginning of the program:

```
const int MAX_ITEMS = 10;
```

In the following lines, I use `MAX_ITEMS` to protect myself before adding another item to the hero's inventory:

```
if (numItems < MAX_ITEMS)
{
    inventory[numItems++] = "healing potion";
}
else
{
    cout << "You have too many items and can't carry another.";
}
```

In the preceding code, I first checked to see whether `numItems` is less than `MAX_ITEMS`. If it is, then I can safely use `numItems` as an index and assign a new `string` object to the array.

In this case, `numItems` is 3, so I assign the string `"healing potion"` to array position 3. If this hadn't been the case, then I would have displayed the message, "You have too many items and can't carry another."

So what happens if you do attempt to access an array element outside the bounds of the array? It depends, because you'd be accessing some unknown part of the computer's memory. At worst, if you attempt to assign some value to an element outside the bounds of an array, you could cause your program to do unpredictable things, and it might even crash.

Testing to make sure that an index number is a valid array position before using it is called *bounds checking*. It's critical for you to perform bounds checking when there's a chance that an index you want to use might not be valid.

Understanding C-Style Strings

Before `string` objects came along, C++ programmers represented strings with arrays of characters terminated by a null character. These arrays of characters are now called *C-style strings* because the practice began in C programs. You can declare and initialize a C-style string as you would any other array:

```
char phrase[] = "Game Over!!!";
```

C-style strings terminate with a character called the *null character* to signify their end. You can write the null character as `'\0'`. I didn't need to use the null character in the previous code because it is stored at the end of the string for me. So technically, `phrase` has 13 elements. (However, functions that work with C-style strings will say that `phrase` has a length of `12`, which makes sense and is in line with how `string` objects work.)

As with any other type of array, you can specify the array size when you define it. So another way to declare and initialize a C-style string is

```
char phrase[81] = "Game Over!!!";
```

The previous code creates a C-style string that can hold 80 printable characters (plus its terminating null character).

C-style strings don't have member functions. But the `cstring` file, which is part of the standard library, contains a variety of functions for working with C-style strings.

A nice thing about `string` objects is that they're designed to work seamlessly with C-style strings. For example, all of the following are completely valid uses of C-style strings with `string` objects:

```
string word1 = "Game";
char word2[] = " Over";

string phrase = word1 + word2;

if (word1 != word2)
{
    cout << "word1 and word2 are not equal.\n";
}

if (phrase.find(word2) != string::npos)
{
    cout << "word2 is contained in phrase.\n";
}
```

You can concatenate `string` objects and C-style strings, but the result is always a `string` object (so the code `char phrase2[] = word1 + word2;` would produce an error). You can compare `string` objects and C-style strings using the relational operators. And you can even use C-style strings as arguments in `string` object member functions.

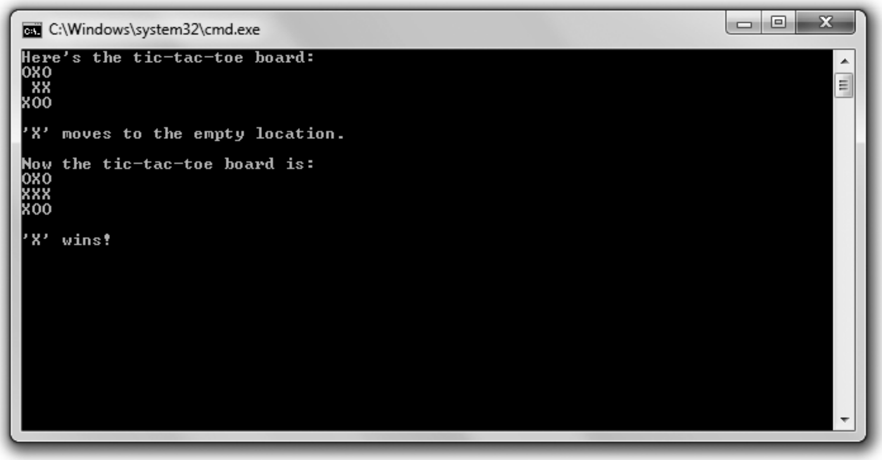
C-style strings have the same shortcomings as arrays. One of the biggest is that their lengths are fixed. So the moral is: Use `string` objects whenever possible, but be prepared to work with C-style strings if necessary.

Using Multidimensional Arrays

As you've seen, sequences are great for games. You can use them in the form of a string to store a player's name, or you can use them in the form of any array to store a list of items in an RPG. But sometimes part of a game cries out for more than a linear list of things. Sometimes part of a game literally requires more dimension. For example, while you could represent a chessboard with a 64-element array, it really is much more intuitive to work with it as a two-dimensional entity of 8 × 8 elements. Fortunately, you can create an array of two or three (or even more dimensions) to best fit your game's needs.

Introducing the Tic-Tac-Toe Board Program

The Tic-Tac-Toe Board program displays a tic-tac-toe board. The program displays the board and declares X the winner. Although the program could have been written using a one-dimensional array, it uses a two-dimensional array to represent the board. Figure 3.5 illustrates the program.



```
Here's the tic-tac-toe board:
OXO
 XX
XOO

'X' moves to the empty location.

Now the tic-tac-toe board is:
OXO
XXX
XOO

'X' wins!
```

Figure 3.5
The tic-tac-toe board is represented by a two-dimensional array.
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 3 folder; the filename is `tic-tac-toe_board.cpp`.

```
// Tic-Tac-Toe Board
// Demonstrates multidimensional arrays

#include <iostream>

using namespace std;

int main()
{
    const int ROWS = 3;
    const int COLUMNS = 3;
    char board[ROWS][COLUMNS] = { {'O', 'X', 'O'},
                                  {' ', 'X', 'X'},
                                  {'X', 'O', 'O'} };

    cout << "Here's the tic-tac-toe board:\n";
    for (int i = 0; i < ROWS; ++i)
    {
        for (int j = 0; j < COLUMNS; ++j)
        {
            cout << board[i][j];
        }

        cout << endl;
    }

    cout << "\n'X' moves to the empty location.\n\n";
    board[1][0] = 'X';

    cout << "Now the tic-tac-toe board is:\n";
    for (int i = 0; i < ROWS; ++i)
    {
        for (int j = 0; j < COLUMNS; ++j)
        {
            cout << board[i][j];
        }

        cout << endl;
    }

    cout << "\n'X' wins!";

    return 0;
}
```

Creating Multidimensional Arrays

One of the first things I do in the program is declare and initialize an array for the tic-tac-toe board.

```
    char board[ROWS][COLUMNS] = { {'O', 'X', 'O'},
                                  {' ', 'X', 'X'},
                                  {'X', 'O', 'O'} };
```

The preceding code declares a 3 × 3 (since `ROWS` and `COLUMNS` are both 3) two-dimensional character array. It also initializes all of the elements.

Hint

It's possible to simply declare a multidimensional array without initializing it. Here's an example:

```
char chessBoard[8][8];
```

The preceding code declares an 8 × 8, two-dimensional character array, `chessBoard`. By the way, multidimensional arrays aren't required to have the same size for each dimension. The following is a perfectly valid declaration for a game map represented by individual characters:

```
char map[12][20];
```

Indexing Multidimensional Arrays

The next thing I do in the program is display the tic-tac-toe board. But before I get into the details of that, I want to explain how to index an individual array element. You index an individual element of a multidimensional array by supplying a value for each dimension of the array. That's what I do to place an X in the array where a space was:

```
board[1][0] = 'X';
```

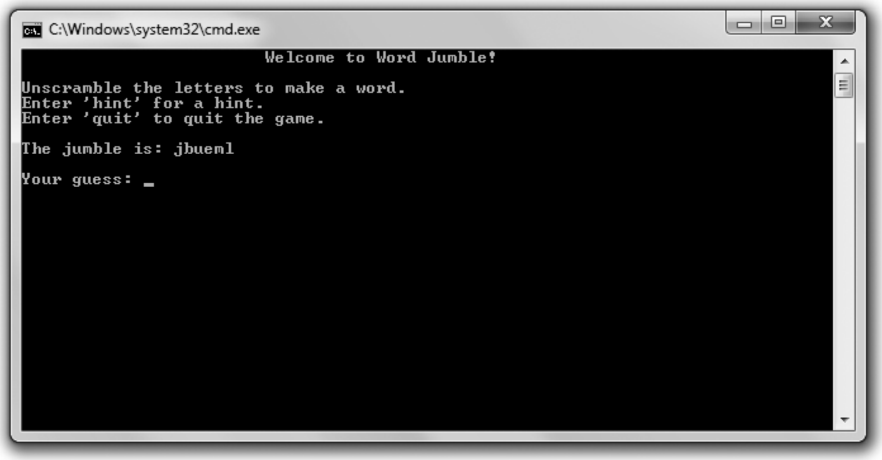
The previous code assigns the character to the element at `board[1][0]` (which was `' '`). Then I display the tic-tac-toe board after the move the same way I displayed it before the move.

```
for (int i = 0; i < ROWS; ++i)
{
    for (int j = 0; j < COLUMNS; ++j)
    {
        cout << board[i][j];
    }
    cout << endl;
}
```

By using a pair of nested `for` loops, I move through the two-dimensional array and display the character elements as I go, forming a tic-tac-toe board.

Introducing Word Jumble

Word Jumble is a puzzle game in which the computer creates a version of a word where the letters are in random order. The player has to guess the word to win the game. If the player is stuck, he or she can ask for a hint. Figure 3.6 shows the game.



```
                              Welcome to Word Jumble!

Unscramble the letters to make a word.
Enter 'hint' for a hint.
Enter 'quit' to quit the game.

The jumble is: jbueml

Your guess: _
```

Figure 3.6
Hmm…the word looks "jumbled."
Used with permission from Microsoft.

In the Real World

Even though puzzle games don't usually break into the top-ten list of games, major companies still publish them year after year. Why? For one simple reason: They're profitable. Puzzle games, while not usually blockbusters, can still sell well. Many gamers out there (casual and hardcore) are drawn to the Zen of a well-designed puzzle game. And puzzle games cost much less to produce than the high-profile games that require large production teams and years of development time.

Setting Up the Program

As usual, I start with some comments and include the files I need. You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 3 folder; the filename is `word_jumble.cpp`.

```
// Word Jumble
// The classic word jumble game where the player can ask for a hint

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

using namespace std;
```

Picking a Word to Jumble

My next task is to pick a word to jumble—the word the player will try to guess. First, I create a list of words and hints:

```
int main()
{
    enum fields {WORD, HINT, NUM_FIELDS};
    const int NUM_WORDS = 5;
    const string WORDS[NUM_WORDS][NUM_FIELDS] =
    {
        {"wall", "Do you feel you're banging your head against something?"},
        {"glasses", "These might help you see the answer."},
        {"labored", "Going slowly, is it?"},
        {"persistent", "Keep at it."},
        {"jumble", "It's what the game is all about."}
    };
```

I declare and initialize a two-dimensional array with words and corresponding hints. The enumeration defines enumerators for accessing the array. For example, `WORDS[x][WORD]` is always a `string` object that is one of the words, while `WORDS[x][HINT]` is the corresponding hint.

Trick

You can list a final enumerator in an enumeration as a convenient way to store the number of elements. Here's an example:

```
enum difficulty {EASY, MEDIUM, HARD, NUM_DIFF_LEVELS};
cout << "There are " << NUM_DIFF_LEVELS << " difficulty levels.";
```

In the previous code, `NUM_DIFF_LEVELS` is 3, the exact number of difficulty levels in the enumeration. As a result, the second line of code displays the message, "There are 3 difficulty levels."

Next, I pick a random word from my choices.

```
    srand(static_cast<unsigned int>(time(0)));
    int choice = (rand() % NUM_WORDS);
    string theWord = WORDS[choice][WORD];   //word to guess
    string theHint = WORDS[choice][HINT];   //hint for word
```

I generate a random index based on the number of words in the array. Then I assign both the random word at that index and its corresponding hint to the variables `theWord` and `theHint`.

Jumbling the Word

Now that I have the word for the player to guess, I need to create a jumbled version of it.

```
string jumble = theWord;   //jumbled version of word
int length = jumble.size();
for (int i = 0; i < length; ++i)
{
    int index1 = (rand() % length);
    int index2 = (rand() % length);
    char temp = jumble[index1];
    jumble[index1] = jumble[index2];
    jumble[index2] = temp;
}
```

In the preceding code, I created a copy of the word `jumble` to…well, jumble. I generated two random positions in the `string` object and swapped the characters at those positions. I did this a number of times equal to the length of the word.

Welcoming the Player

Now it's time to welcome the player, which is what I do next.

```
cout << "\t\t\tWelcome to Word Jumble!\n\n";
cout << "Unscramble the letters to make a word.\n";
cout << "Enter 'hint' for a hint.\n";
cout << "Enter 'quit' to quit the game.\n\n";
cout << "The jumble is: " << jumble;

string guess;
cout << "\n\nYour guess: ";
cin >> guess;
```

I gave the player instructions on how to play, including how to quit and how to ask for a hint.

Hint

As enthralling as you think your game is, you should always provide a way for the player to exit it.

Entering the Game Loop

Next, I enter the game loop.

```
    while ((guess != theWord) && (guess != "quit"))
    {
        if (guess == "hint")
        {
            cout << theHint;
        }
        else
        {
            cout << "Sorry, that's not it.";
        }

        cout <<"\n\nYour guess: ";
        cin >> guess;
    }
```

The loop continues to ask the player for a guess until the player either guesses the word or asks to quit.

Saying Goodbye

When the loop ends, the player has either won or quit, so it's time to say goodbye.

```
    if (guess == theWord)
    {
        cout << "\nThat's it! You guessed it!\n";
    }

    cout << "\nThanks for playing.\n";

    return 0;
}
```

If the player has guessed the word, I congratulate him or her. Finally, I thank the player for playing.

Summary

In this chapter, you learned the following concepts:

- The `for` loop lets you repeat a section of code. In a `for` loop, you can provide an initialization statement, an expression to test, and an action to take after each loop iteration.
- `for` loops are often used for counting or looping through a sequence.
- Objects are encapsulated, cohesive entities that combine data (called *data members*) and functions (called *member functions*).
- `string` objects (often just called *strings*) are defined in the file `string`, which is part of the standard library. `string` objects allow you to store a sequence of characters and also have member functions.
- `string` objects are defined so that they work intuitively with familiar operators, such as the concatenation operator and the relational operators.
- All `string` objects have member functions, including those for determining a `string` object's length, determining whether a `string` object is empty, finding substrings, and removing substrings.
- Arrays provide a way to store and access sequences of any type.
- A limitation of arrays is that they have a fixed length.
- You can access individual elements of `string` objects and arrays through the subscripting operator.
- Bounds checking is not enforced when attempts are made to access individual elements of `string` objects or arrays. Therefore, bounds checking is up to the programmer.
- C-style strings are character arrays terminated with the null character. They are the standard way to represent strings in the C language. And even though C-style strings are perfectly legal in C++, `string` objects are the preferred way to work with sequences of characters.
- Multidimensional arrays allow for access to array elements using multiple subscripts. For example, a chessboard can be represented as a two-dimensional array, 8 × 8 elements.

Questions and Answers

Q: Which is better, a `while` loop or a `for` loop?
A: Neither is inherently better than the other. Use the loop that best fits your needs.

Q: When might it be better to use a `for` loop than a `while` loop?
A: You can create a `while` loop to do the job of any `for` loop; however, there are some cases that cry out for a `for` loop. Those include counting and iterating through a sequence.

Q: Can I use `break` and `continue` statements with `for` loops?
A: Sure. And they behave just like they do in `while` loops: `break` ends the loop and `continue` jumps control back to the top of the loop.

Q: Why do programmers tend to use variable names such as `i`, `j`, and `k` as counters in `for` loops?
A: Believe it or not, programmers use `i`, `j`, and `k` mainly out of tradition. The practice started in early versions of the FORTRAN language, in which integer variables had to start with certain letters, including `i`, `j`, and `k`.

Q: I don't have to include a file to use `int` or `char` types, so why do I have to include the `string` file to use strings?
A: `int` and `char` are built-in types. They're always accessible in any C++ program. The `string` type, on the other hand, is not a built-in type. It's defined as part of the standard library in the file `string`.

Q: How did C-style strings get their name?
A: In the C programming language, programmers represent strings with arrays of characters terminated by a null character. This practice carried over to C++. After the new `string` type was introduced in C++, programmers needed a way to differentiate between the two. Therefore, the old method was dubbed C-style strings.

Q: Why should I use `string` objects instead of C-style strings?
A: `string` objects have advantages over C-style strings. The most obvious is that they are dynamically sizeable. You don't have to specify a length limit when you create one.

Q: Should I ever use C-style strings?
A: You should opt for `string` objects whenever possible. If you're working on an existing project that uses C-style strings, then you might have to work with C-style strings.

Q: What is operator overloading?

A: It's a process that allows you to define the use of familiar operators in different contexts with different but predictable results. For example, the + operator that is used to add numbers is overloaded by the `string` type to join strings.

Q: Can't operator overloading be confusing?

A: It's true that by overloading an operator you give it another meaning. But the new meaning applies only in a specific new context. For example, it's clear in the expression `4 + 6` that the + operator adds numbers, while in the expression `myString1 + myString2`, the + operator joins strings.

Q: Can I use the += operator to concatenate strings?

A: Yes, the += operator is overloaded so it works with strings.

Q: To get the number of characters in a `string` object, should I use the `length()` member function or the `size()` member function?

A: Both `length()` and `size()` return the same value, so you can use either.

Q: What's a predicate function?

A: A function that returns either `true` or `false`. The `string` object member function `empty()` is an example of a predicate function.

Q: What happens if I try to assign a value to an element beyond the bounds of an array?

A: C++ will allow you to make the assignment. However, the results are unpredictable and might cause your program to crash. That's because you're altering some unknown part of your computer's memory.

Q: Why should I use multidimensional arrays?

A: To make working with a group of elements more intuitive. For example, you could represent a chessboard with a one-dimensional array, as in `chessBoard[64]`, or you could represent it with a more intuitive, two-dimensional array, as in `chessBoard[8][8]`.

Discussion Questions

1. What are some of the things from your favorite game that you could represent as objects? What might their data members and member functions be?
2. What are the advantages of using an array over a group of individual variables?
3. What are some limitations imposed by a fixed array size?
4. What are the advantages and disadvantages of operator overloading?
5. What kinds of games could you create using string objects, arrays, and `for` loops as your main tools?

Exercises

1. Improve the Word Jumble game by adding a scoring system. Make the point value for a word based on its length. Deduct points if the player asks for a hint.
2. What's wrong with the following code?

```
for (int i = 0; i <= phrase.size(); ++i)
{
    cout << "Character at position " << i << " is: " << phrase[i] << endl;
}
```

3. What's wrong with the following code?

```
const int ROWS = 2;
const int COLUMNS = 3;
char board[COLUMNS][ROWS] = { {'O', 'X', 'O'},
                              {' ', 'X', 'X'} };
```