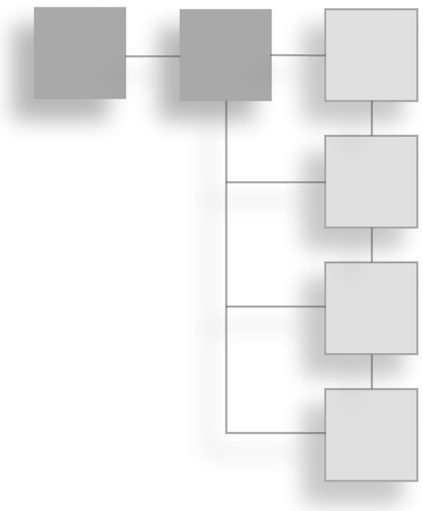
Chapter 4

The Standard Template Library: Hangman



So far, you've seen how to work with sequences of values using arrays. But there are more sophisticated ways to work with collections of values. In fact, working with collections is so common that part of standard C++ is dedicated to doing just that. In this chapter, you'll get an introduction to this important library. Specifically, you'll learn to:

- Use `vector` objects to work with sequences of values
- Use `vector` member functions to manipulate sequence elements
- Use iterators to move through sequences
- Use library algorithms to work with groups of elements
- Plan your programs with pseudocode

Introducing the Standard Template Library

Good game programmers are lazy. It's not that they don't want to work; it's just that they don't want to redo work that's already been done—especially if it has been done well. The *STL* (*Standard Template Library*) represents a powerful collection of programming work that's been done well. It provides a group of containers, algorithms, and iterators, among other things.

So what's a container and how can it help you write games? Well, containers let you store and access collections of values of the same type. Yes, arrays let you do the same thing, but the STL containers offer more flexibility and power than a simple but trusty array. The STL defines a variety of container types; each works in a different way to meet different needs.

The algorithms defined in the STL work with its containers. The *algorithms* are common functions that game programmers find themselves repeatedly applying to groups of values. They include algorithms for sorting, searching, copying, merging, inserting, and removing container elements. The cool thing is that the same algorithm can work its magic on many different container types.

Iterators are objects that identify elements in containers and can be manipulated to move among elements. They're great for, well, iterating through containers. In addition, iterators are required by the STL algorithms.

All of this makes a lot more sense when you see an actual implementation of one of the container types, so that's up next.

Using Vectors

The `vector` class defines one kind of container provided by the STL. It meets the general description of a *dynamic array*—an array that can grow and shrink in size as needed. In addition, `vector` defines member functions to manipulate vector elements. This means that the vector has all of the functionality of the array plus more.

At this point, you may be thinking to yourself: Why learn to use these fancy new vectors when I can already use arrays? Well, vectors have certain advantages over arrays, including:

- Vectors can grow as needed while arrays cannot. This means that if you use a vector to store objects for enemies in a game, the vector will grow to accommodate the number of enemies that are created. If you use an array, you have to create one that can store some maximum number of enemies. And if, during play, you need more room in the array than you thought, you're out of luck.
- Vectors can be used with the STL algorithms while arrays cannot. This means that by using vectors you get complex functionality like searching and sorting, built-in. If you use arrays, you have to write your own code to achieve this same functionality.

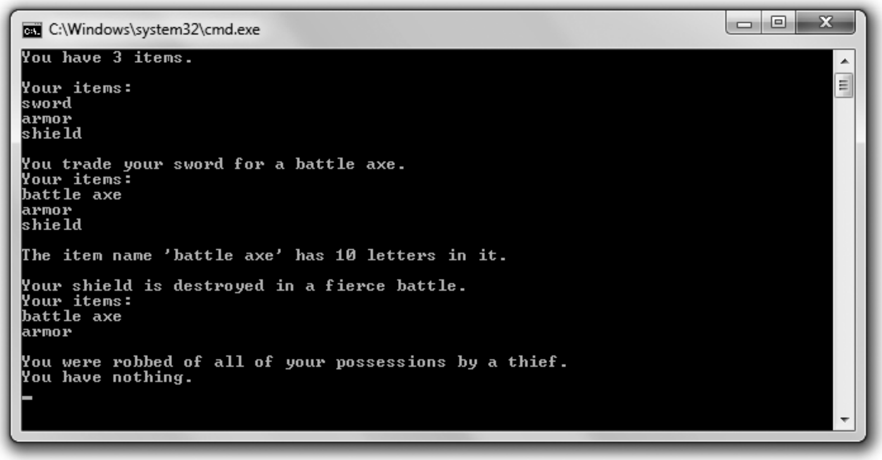
There are a few disadvantages to vectors when compared to arrays, including:

- Vectors require a bit of extra memory as overhead.
- There can be a performance cost when a vector grows in size.
- Vectors may not be available on some game console systems.

Overall, vectors (and the STL) can be a welcome tool in most any project.

Introducing the Hero's Inventory 2.0 Program

From the user's point of view, the Hero's Inventory 2.0 program is similar to its predecessor, the Hero's Inventory program from Chapter 3, "for Loops, Strings, and Arrays: Word Jumble." The new version stores and works with a collection of `string` objects that represent a hero's inventory. However, from the programmer's perspective the program is quite different. That's because the new program uses a vector instead of an array to represent the inventory. Figure 4.1 shows the results of the program.



```
You have 3 items.

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.

Your shield is destroyed in a fierce battle.
Your items:
battle axe
armor

You were robbed of all of your possessions by a thief.
You have nothing.
```

Figure 4.1
This time the hero's inventory is represented by a vector.
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 4 folder; the filename is `heros_inventory2.cpp`.

```
// Hero's Inventory 2.0
// Demonstrates vectors

#include <iostream>
#include <string>
#include <vector>

using namespace std;

int main()
{
    vector<string> inventory;
    inventory.push_back("sword");
    inventory.push_back("armor");
    inventory.push_back("shield");

    cout << "You have " << inventory.size() << " items.\n";

    cout << "\nYour items:\n";
    for (unsigned int i = 0; i < inventory.size(); ++i)
    {
        cout << inventory[i] << endl;
    }

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

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

    cout << "\nYour shield is destroyed in a fierce battle.";
    inventory.pop_back();
    cout << "\nYour items:\n";
    for (unsigned int i = 0; i < inventory.size(); ++i)
    {
        cout << inventory[i] << endl;
    }

    cout << "\nYou were robbed of all of your possessions by a thief.";
    inventory.clear();
    if (inventory.empty())
    {
        cout << "\nYou have nothing.\n";
    }
    else
    {
        cout << "\nYou have at least one item.\n";
    }

    return 0;
}
```

Preparing to Use Vectors

Before I can declare a vector, I have to include the file that contains its definition:

```
#include <vector>
```

All STL components live in the `std` namespace, so by using the following code (as I typically do) I can refer to `vector` without having to precede it with `std::`.

```
using namespace std;
```

Declaring a Vector

Okay, the first thing I do in `main()` is declare a new vector.

```
    vector<string> inventory;
```

The preceding line declared an empty vector named `inventory`, which can contain `string` object elements. Declaring an empty vector is fine because it grows in size when you add new elements.

To declare a vector of your own, write `vector` followed by the type of objects you want to use with the vector (surrounded by the < and > symbols), followed by the vector name.

Hint

There are additional ways to declare a vector. You can declare one with a starting size by specifying a number in parentheses after the vector name.

```
vector<string> inventory(10);
```

The preceding code declared a vector to hold `string` object elements with a starting size of 10. You can also initialize all of a vector's elements to the same value when you declare it. You simply supply the number of elements followed by the starting value, as in:

```
vector<string> inventory(10, "nothing");
```

The preceding code declared a vector with a size of 10 and initialized all 10 elements to `"nothing"`. Finally, you can declare a vector and initialize it with the contents of another vector.

```
vector<string> inventory(myStuff);
```

The preceding code created a new vector with the same contents as the vector `myStuff`.

Using the push_back( ) Member Function

Next, I give the hero the same three starting items as in the previous version of the program.

```
inventory.push_back("sword");
inventory.push_back("armor");
inventory.push_back("shield");
```

The `push_back()` member function adds a new element to the end of a vector. In the preceding lines, I added `"sword"`, `"armor"`, and `"shield"` to `inventory`. As a result, `inventory[0]` is equal to `"sword"`, `inventory[1]` is equal to `"armor"`, and `inventory[2]` is equal to `"shield"`.

Using the size( ) Member Function

Next, I display the number of items the hero has in his possession.

```
cout << "You have " << inventory.size() << " items.\n";
```

I get the size of `inventory` by calling the `size()` member function with `inventory.size()`. The `size()` member function simply returns the size of a vector. In this case, it returns 3.

Indexing Vectors

Next, I display all of the hero's items.

```
cout << "\nYour items:\n";
for (unsigned int i = 0; i < inventory.size(); ++i)
{
    cout << inventory[i] << endl;
}
```

Just as with arrays, you can index vectors by using the subscripting operator. In fact, the preceding code is nearly identical to the same section of code from the original Hero's Inventory program. The only difference is that I used `inventory.size()` to specify when the loop should end. Note that I made the loop variable `i` an `unsigned int` because the value returned by `size()` is an unsigned integer type.

Next, I replace the hero's first item.

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

Again, just as with arrays, I use the subscripting operator to assign a new value to an existing element position.

Trap

Although vectors are dynamic, you can't increase a vector's size by applying the subscripting operator. For example, the following highly dangerous code snippet does not increase the size of the vector `inventory`:

```
vector<string> inventory; //creating an empty vector
inventory[0] = "sword";   //may cause your program to crash!
```

Just as with arrays, you can attempt to access a nonexistent element position—but with potentially disastrous results. The preceding code changed some unknown section of your computer's memory and could cause your program to crash. To add a new element at the end of a vector, use the `push_back()` member function.

Calling Member Functions of an Element

Next, I show the number of letters in the name of the first item in the hero's inventory.

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

Just as with arrays, you can access the member functions of a vector element by writing the element, followed by the member selection operator, followed by the member function name. Because `inventory[0]` is equal to `"battle axe"`, `inventory[0].size()` returns 10.

Using the pop_back( ) Member Function

I remove the hero's shield using

```
inventory.pop_back();
```

The `pop_back()` member function removes the last element of a vector and reduces the vector size by one. In this case, `inventory.pop_back()` removes `"shield"` from `inventory` because that was the last element in the vector. Also, the size of `inventory` is reduced from 3 to 2.

Using the clear( ) Member Function

Next, I simulate the act of a thief robbing the hero of all of his items.

```
inventory.clear();
```

The `clear()` member function removes all of the items of a vector and sets its size to `0`. After the previous line of code executes, `inventory` is an empty vector.

Using the empty( ) Member Function

Finally, I check to see whether the hero has any items in his inventory.

```
if (inventory.empty())
{
    cout << "\nYou have nothing.\n";
}
else
{
    cout << "\nYou have at least one item.\n";
}
```

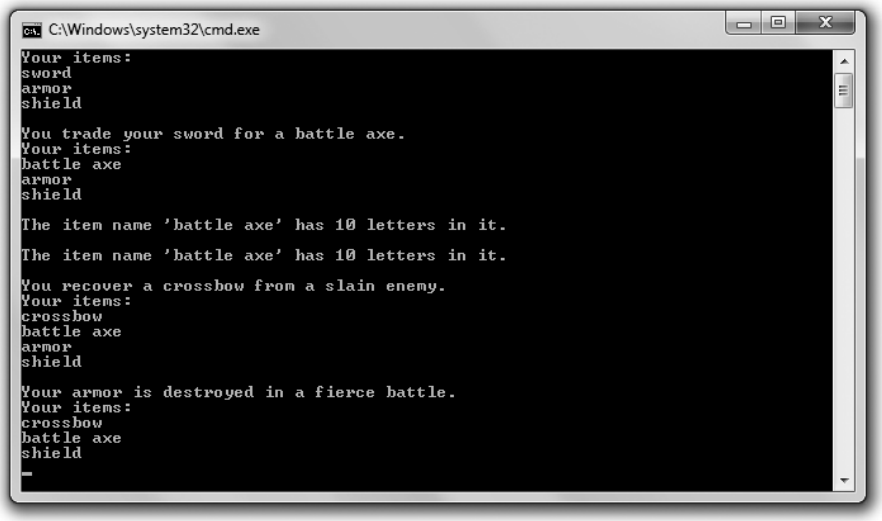
The `vector` member function `empty()` works just like the `string` member function `empty()`. It returns `true` if the `vector` object is empty; otherwise, it returns `false`. Because `inventory` is empty in this case, the program displays the message, "You have nothing."

Using Iterators

Iterators are the key to using containers to their fullest potential. With iterators you can, well, iterate through a sequence container. In addition, important parts of the STL require iterators. Many container member functions and STL algorithms take iterators as arguments. So if you want to reap the benefits of these member functions and algorithms, you must use iterators.

Introducing the Hero's Inventory 3.0 Program

The Hero's Inventory 3.0 program acts like its two predecessors, at least at the start. The program shows off a list of items, replaces the first item, and displays the number of letters in the name of an item. But then the program does something new: It inserts an item at the beginning of the group, and then it removes an item from the middle of the group. The program accomplishes all of this by working with iterators. Figure 4.2 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.

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

You recover a crossbow from a slain enemy.
Your items:
crossbow
battle axe
armor
shield

Your armor is destroyed in a fierce battle.
Your items:
crossbow
battle axe
shield
_
```

Figure 4.2
The program performs a few vector manipulations that you can accomplish only with iterators.
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 4 folder; the filename is `heros_inventory3.cpp`.

```
// Hero's Inventory 3.0
// Demonstrates iterators

#include <iostream>
#include <string>
#include <vector>

using namespace std;

int main()
{
    vector<string> inventory;
    inventory.push_back("sword");
    inventory.push_back("armor");
    inventory.push_back("shield");

    vector<string>::iterator myIterator;
    vector<string>::const_iterator iter;
    cout << "Your items:\n";
    for (iter = inventory.begin(); iter != inventory.end(); ++iter)
    {
        cout << *iter << endl;
    }

    cout << "\nYou trade your sword for a battle axe.";
    myIterator = inventory.begin();
    *myIterator = "battle axe";
    cout << "\nYour items:\n";
    for (iter = inventory.begin(); iter != inventory.end(); ++iter)
    {
        cout << *iter << endl;
    }

    cout << "\nThe item name '" << *myIterator << "' has ";
    cout << (*myIterator).size() << " letters in it.\n";

    cout << "\nThe item name '" << *myIterator << "' has ";
    cout << myIterator->size() << " letters in it.\n";

    cout << "\nYou recover a crossbow from a slain enemy.";
    inventory.insert(inventory.begin(), "crossbow");
    cout << "\nYour items:\n";
    for (iter = inventory.begin(); iter != inventory.end(); ++iter)
    {
        cout << *iter << endl;
    }

    cout << "\nYour armor is destroyed in a fierce battle.";
    inventory.erase((inventory.begin() + 2));
    cout << "\nYour items:\n";
    for (iter = inventory.begin(); iter != inventory.end(); ++iter)
    {
        cout << *iter << endl;
    }

    return 0;
}
```

Declaring Iterators

After I declare a vector for the hero's inventory and add the same three `string` objects from the previous incarnations of the program, I declare an iterator.

```
    vector<string>::iterator myIterator;
```

The preceding line declares an iterator named `myIterator` for a vector that contains `string` objects. To declare an iterator of your own, follow the same pattern. Write the container type, followed by the type of objects the container will hold (surrounded by the < and > symbols), followed by the scope resolution operator (the :: symbol), followed by `iterator`, followed by a name for your new iterator.

So what are iterators? *Iterators* are values that identify a particular element in a container. Given an iterator, you can access the value of the element. Given the right kind of iterator, you can change the value. Iterators can also move among elements via familiar arithmetic operators.

A way to think about iterators is to imagine them as Post-it notes that you can stick on a specific element in a container. An iterator is not one of the elements, but a way to refer to one. Specifically, I can use `myIterator` to refer to a particular element of the vector `inventory`. That is, I can stick the `myIterator` Post-it note on a specific element in `inventory`. Once I've done that, I can access the element or even change it through the iterator.

Next, I declare another iterator.

```
    vector<string>::const_iterator iter;
```

The preceding line of code creates a constant iterator named `iter` for a vector that contains `string` objects. A *constant iterator* is just like a regular iterator except that you can't use it to change the element to which it refers; the element must remain constant. You can think of a constant iterator as providing read-only access. However, the iterator itself can change. This means you can move `iter` all around the vector `inventory` as you see fit. You can't, however, change the value of any of the elements through `iter`. With a constant iterator the Post-It can change, but the thing it's stuck to can't.

Why would you want to use a constant iterator if it's a limited version of a regular iterator? First, it makes your intentions clearer. When you use a constant iterator, it's clear that you won't be changing any element to which it refers. Second, it's safer. You can use a constant iterator to avoid accidentally changing a container element. (If you attempt to change an element through a constant iterator, you'll generate a compile error.)

Trap

Using `push_back()` might invalidate all iterators referencing the vector.

Is all of this iterator talk a little too abstract for you? Are you tired of analogies about Post-it notes? Fear not—next, I put an actual iterator to work.

Looping through a Vector

Next, I loop through the contents of the vector and display the hero's inventory.

```
cout << "Your items:\n";
for (iter = inventory.begin(); iter != inventory.end(); ++iter)
    cout << *iter << endl;
```

In the preceding code, I use a `for` loop to move from the first to the last element of `inventory`. At this general level, this is exactly how I looped through the contents of the vector in Hero's Inventory 2.0. But instead of using an integer and the subscripting operator to access each element, I used an iterator. Basically, I moved the Post-it note through the entire sequence of elements and displayed the value of each element to which the note was stuck. There are a lot of new ideas in this little loop, so I'll tackle them one at a time.

Calling the begin( ) Vector Member Function

In the initialization statement of the loop, I assign the return value of `inventory.begin()` to `iter`. The `begin()` member function returns an iterator that refers to a container's first element. So in this case, the statement assigns an iterator that refers to the first element of `inventory` (the `string` object equal to `"sword"`) to `iter`. Figure 4.3 shows an abstract view of the iterator returned by a call to `inventory.begin()`. (Note that the figure is abstract because the vector `inventory` doesn't contain the string literals `"sword"`, `"armor"`, and `"shield"`; it contains `string` objects.)

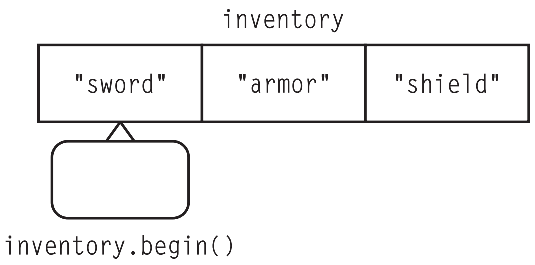


Figure 4.3
A call to `inventory.begin()` returns an iterator that refers to the first element in the vector.

Calling the end( ) Vector Member Function

In the test statement of the loop, I test the return value of `inventory.end()` against `iter` to make sure the two are not equal. The `end()` member function returns an iterator one past the last element in a container. This means the loop will continue until `iter` has moved through all of the elements in `inventory`. Figure 4.4 shows an abstract view of the iterator returned by a call to this member function. (Note that the figure is abstract because the vector `inventory` doesn't contain the string literals `"sword"`, `"armor"`, and `"shield"`; it contains string objects.)

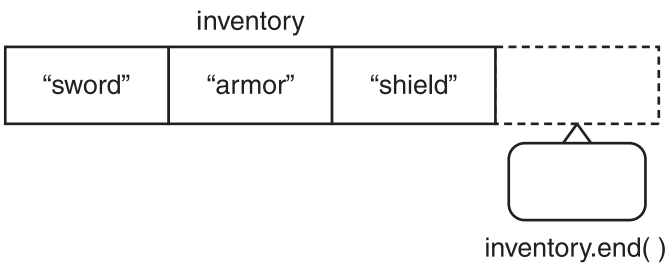


Figure 4.4
A call to `inventory.end()` returns an iterator one past the last element of the vector.

Trap

The `end()` `vector` member function returns an iterator that's one *past* the last element in the vector—not the last element. Therefore, you can't get a value from the iterator returned by `end()`. This might seem counterintuitive, but it works well for loops that move through a container.

Altering an Iterator

The action statement in the loop, `++iter`, increments `iter`, which moves it to the next element in the vector. Depending upon the iterator, you can perform other mathematical operations on iterators to move them around a container. Most often, though, you'll find that you simply want to increment an iterator.

Dereferencing an Iterator

In the loop body, I send `*iter` to `cout`. By placing the dereference operator (`*`) in front of `iter`, I display the value of the element to which the iterator refers (not the iterator itself). By placing the dereference operator in front of an iterator, you're saying, "Treat this as the thing that the iterator references, not as the iterator itself."

Changing the Value of a Vector Element

Next, I change the first element in the vector from the `string` object equal to `"sword"` to the `string` object equal to `"battle axe"`. First, I set `myIterator` to reference the first element of `inventory`.

```
myIterator = inventory.begin();
```

Then I change the value of the first element.

```
*myIterator = "battle axe";
```

Remember, by dereferencing `myIterator` with `*`, the preceding assignment statement says, "Assign `"battle axe"` to the element that `myIterator` references." It does not change `myIterator`. After the assignment statement, `myIterator` still refers to the first element in the vector.

Just to prove that the assignment worked, I then display all of the elements in `inventory`.

Accessing Member Functions of a Vector Element

Next, I display the number of characters in the name of the first item in the hero's inventory.

```
cout << "\nThe item name '" << *myIterator << "' has ";
cout << (*myIterator).size() << " letters in it.\n";
```

The code `(*myIterator).size()` says, "Take the result of dereferencing `myIterator` and call that object's `size()` member function." Because `myIterator` refers to the `string` object equal to `"battle axe"`, the code returns 10.

Hint

Whenever you dereference an iterator to access a data member or member function, surround the dereferenced iterator by a pair of parentheses. This ensures that the dot operator will be applied to the object the iterator references.

The code `(*myIterator).size()` is not the prettiest, so C++ offers an alternative, more intuitive way to express the same thing, which I demonstrate in the next two lines of the program.

```
cout << "\nThe item name '" << *myIterator << "' has ";
cout << myIterator->size() << " letters in it.\n";
```

The preceding code does exactly the same thing the first pair of lines I presented in this section do; it displays the number of characters in `"battle axe"`. However, notice that I substitute `myIterator->size()` for `(*myIterator).size()`. You can see that this version (with the `->` symbol) is more readable. The two pieces of code mean exactly the same thing to the computer, but this new version is easier for humans to use. In general, you can use the indirect member selection operator, `->`, to access the member functions or data members of an object that an iterator references.

Hint

Syntactic sugar is a nicer, alternative syntax. It replaces harsh syntax with something that's a bit easier to swallow. As an example, instead of writing the code `(*myIterator).size()`, I can use the syntactic sugar provided by the `->` operator and write `myIterator->size()`.

Using the insert( ) Vector Member Function

Next, I add a new item to the hero's inventory. This time, though, I don't add the item to the end of the sequence; instead, I insert it at the beginning.

```
    inventory.insert(inventory.begin(), "crossbow");
```

One form of the `insert()` member function inserts a new element into a vector just before the element referred to by a given iterator. You supply two arguments to this version of `insert()`—the first is an iterator, and the second is the element to be inserted. In this case, I inserted `"crossbow"` into `inventory` just before the first element. As a result, all of the other elements will move down by one. This version of the `insert()` member function returns an iterator that references the newly inserted element. In this case, I don't assign the returned iterator to a variable.

Trap

Calling the `insert()` member function on a vector invalidates all of the iterators that reference elements after the insertion point because all of the elements after the insertion point are shifted down by one.

Next, I show the contents of the vector to prove the insertion worked.

Using the erase( ) Vector Member Function

Next, I remove an item from the hero's inventory. However, this time I don't remove the item at the end of the sequence; instead, I remove one from the middle.

```
inventory.erase((inventory.begin() + 2));
```

One form of the `erase()` member function removes an element from a vector. You supply one argument to this version of `erase()`—the iterator that references the element you want to remove. In this case, I passed `(inventory.begin() + 2)`, which is equal to the iterator that references the third element in `inventory`. This removes the `string` object equal to `"armor"`. As a result, all of the following elements will move up by one. This version of the `erase()` member function returns an iterator that references the element after the element that was removed. In this case, I don't assign the returned iterator to a variable.

Trap

Calling the `erase()` member function on a vector invalidates all of the iterators that reference elements after the removal point because all of the elements after the removal point are shifted up by one.

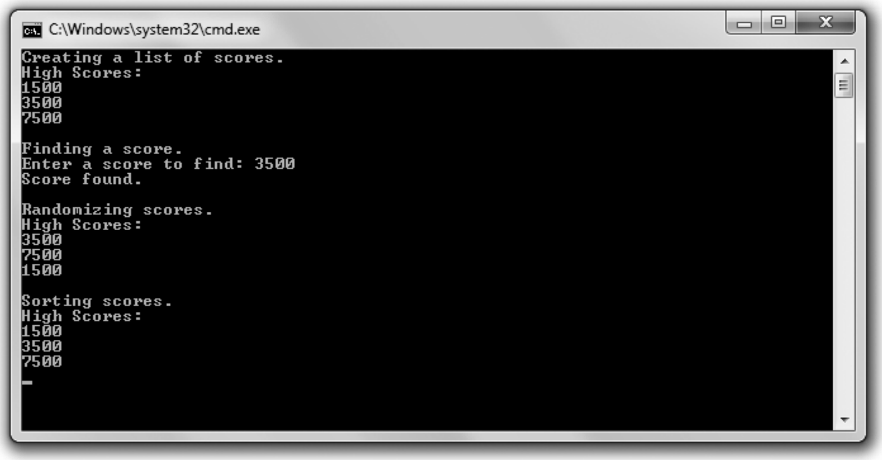
Next, I show the contents of the vector to prove the removal worked.

Using Algorithms

The STL defines a group of algorithms that allow you to manipulate elements in containers through iterators. Algorithms exist for common tasks such as searching, randomizing, and sorting. These algorithms are your built-in arsenal of flexible and efficient weapons. By using them, you can leave the mundane task of manipulating container elements in common ways to the STL so you can concentrate on writing your game. The powerful thing about these algorithms is that they are generic—the same algorithm can work with elements of different container types.

Introducing the High Scores Program

The High Scores program creates a vector of high scores. It uses STL algorithms to search, shuffle, and sort the scores. Figure 4.5 illustrates the program.



```
Creating a list of scores.
High Scores:
1500
3500
7500

Finding a score.
Enter a score to find: 3500
Score found.

Randomizing scores.
High Scores:
3500
7500
1500

Sorting scores.
High Scores:
1500
3500
7500
```

Figure 4.5
STL algorithms search, shuffle, and sort elements of a vector of high scores.
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 4 folder; the filename is `high_scores.cpp`.

```
// High Scores
// Demonstrates algorithms

#include <iostream>
#include <vector>
#include <algorithm>
#include <ctime>
#include <cstdlib>

using namespace std;

int main()
{
    vector<int>::const_iterator iter;

    cout << "Creating a list of scores.";
    vector<int> scores;
    scores.push_back(1500);
    scores.push_back(3500);
    scores.push_back(7500);

    cout << "\nHigh Scores:\n";
    for (iter = scores.begin(); iter != scores.end(); ++iter)
    {
        cout << *iter << endl;
    }

    cout << "\nFinding a score.";
    int score;
    cout << "\nEnter a score to find: ";
    cin >> score;
    iter = find(scores.begin(), scores.end(), score);
    if (iter != scores.end())
    {
        cout << "Score found.\n";
    }
    else
    {
        cout << "Score not found.\n";
    }

    cout << "\nRandomizing scores.";
    srand(static_cast<unsigned int>(time(0)));
    random_shuffle(scores.begin(), scores.end());
    cout << "\nHigh Scores:\n";
    for (iter = scores.begin(); iter != scores.end(); ++iter)
    {
        cout << *iter << endl;
    }

    cout << "\nSorting scores.";
    sort(scores.begin(), scores.end());
    cout << "\nHigh Scores:\n";
    for (iter = scores.begin(); iter != scores.end(); ++iter)
    {
        cout << *iter << endl;
    }

    return 0;
}
```

Preparing to Use Algorithms

In order to use the STL algorithms, I include the file with their definitions.

```
#include <algorithm>
```

As you know, all STL components live in the `std` namespace. By using the following code (as I typically do), I can refer to algorithms without having to precede them with `std::`.

```
using namespace std;
```

Using the find( ) Algorithm

After I display the contents of the vector `scores`, I get a value from the user to find and store it in the variable `score`. Then I use the `find()` algorithm to search the vector for the value:

```
iter = find(scores.begin(), scores.end(), score);
```

The `find()` STL algorithm searches a specified range of a container's elements for a value. It returns an iterator that references the first matching element. If no match is found, it returns an iterator to the end of the range. You must pass the starting point as an iterator, the ending point as an iterator, and a value to find. The algorithm searches from the starting iterator up to but not including the ending iterator. In this case, I passed `scores.begin()` and `scores.end()` as the first and second arguments to search the entire vector. I passed `score` as the third argument to search for the value the user entered.

Next, I check to see if the value `score` was found:

```
if (iter != scores.end())
{
    cout << "Score found.\n";
}
else
{
    cout << "Score not found.\n";
}
```

Remember, `iter` will reference the first occurrence of `score` in the vector, if the value was found. So, as long as `iter` is not equal to `scores.end()`, I know that score was found and I display a message saying so. Otherwise, `iter` will be equal to `scores.end()` and I know `score` was not found.

Using the random_shuffle( ) Algorithm

Next, I prepare to randomize the scores using the `random_shuffle()` algorithm. Just as when I generate a single random number, I seed the random number generator before I call `random_shuffle()`, so the order of the scores might be different each time I run the program.

```
srand(static_cast<unsigned int>(time(0)));
```

Then I reorder the scores in a random way.

```
random_shuffle(scores.begin(), scores.end());
```

The `random_shuffle()` algorithm randomizes the elements of a sequence. You must supply as iterators the starting and ending points of the sequence to shuffle. In this case, I passed the iterators returned by `scores.begin()` and `scores.end()`. These two iterators indicate that I want to shuffle all of the elements in `scores`. As a result, `scores` contains the same scores, but in some random order.

Then I display the scores to prove the randomization worked.

Trick

Although you might not want to randomize a list of high scores, `random_shuffle()` is a valuable algorithm for games. You can use it for everything from shuffling a deck of cards to mixing up the order of the enemies a player will encounter in a game level.

Using the sort( ) Algorithm

Next, I sort the scores.

```
sort(scores.begin(), scores.end());
```

The `sort()` algorithm sorts the elements of a sequence in ascending order. You must supply as iterators the starting and ending points of the sequence to sort. In this particular case, I passed the iterators returned by `scores.begin()` and `scores.end()`. These two iterators indicate that I want to sort all of the elements in `scores`. As a result, `scores` contains all of the scores in ascending order.

Finally, I display the scores to prove the sorting worked.

Trick

A very cool property of STL algorithms is that they can work with containers defined outside of the STL. These containers only have to meet certain requirements. For example, even though `string` objects are not part of the STL, you can use appropriate STL algorithms on them. The following code snippet demonstrates this:

```
string word = "High Scores";
random_shuffle(word.begin(), word.end());
```

The preceding code randomly shuffles the characters in `word`. As you can see, `string` objects have both `begin()` and `end()` member functions, which return iterators to the first character and one past the last character, respectively. That's part of the reason why STL algorithms work with `string`s—because they're designed to.

Understanding Vector Performance

Like all STL containers, vectors provide game programmers with sophisticated ways to work with information, but this level of sophistication can come at a performance cost. And if there's one thing game programmers obsess about, it's performance. But fear not, vectors and other STL containers are incredibly efficient. In fact, they've already been used in published PC and console games. However, these containers have their strengths and weaknesses; a game programmer needs to understand the performance characteristics of the various container types so that he can choose the right one for the job.

Examining Vector Growth

Although vectors grow dynamically as needed, every vector has a specific size. When a new element added to a vector pushes the vector beyond its current size, the computer reallocates memory and might even copy all of the vector elements to this newly seized chunk of memory real estate. This can cause a performance hit.

The most important thing to keep in mind about program performance is whether you need to care. For example, vector memory reallocation might not occur at a performance-critical part of your program. In that case, you can safely ignore the cost of reallocation. Also, with small vectors, the reallocation cost might be insignificant so, again, you can safely ignore it. However, if you need greater control over when these memory reallocations occur, you have it.

Using the capacity( ) Member Function

The `capacity()` `vector` member function returns the capacity of a vector—in other words, the number of elements that a vector can hold before a program must reallocate more memory for it. A vector's capacity is not the same thing as its size (the number of elements a vector currently holds). Here's a code snippet to help drive this point home:

```
cout << "Creating a 10 element vector to hold scores.\n";
vector<int> scores(10, 0);   //initialize all 10 elements to 0
cout << "Vector size is :" << scores.size() << endl;
cout << "Vector capacity is:" << scores.capacity() << endl;

cout << "Adding a score.\n";
scores.push_back(0);   //memory is reallocated to accommodate growth
cout << "Vector size is :" << scores.size() << endl;
cout << "Vector capacity is:" << scores.capacity() << endl;
```

Right after I declare and initialize the vector, this code reports that its size and capacity are both 10. However, after an element is added, the code reports that the vector's size is 11 while its capacity is 20. That's because the capacity of a vector doubles every time a program reallocates additional memory for it. In this case, when a new score was added, memory was reallocated, and the capacity of the vector doubled from 10 to 20.

Using the reserve( ) Member Function

The `reserve()` member function increases the capacity of a vector to the number supplied as an argument. Using `reserve()` gives you control over when a reallocation of additional memory occurs. Here's an example:

```
cout << "Creating a list of scores.\n";
vector<int> scores(10, 0);  //initialize all 10 elements to 0
cout << "Vector size is :" << scores.size() << endl;
cout << "Vector capacity is:" << scores.capacity() << endl;

cout << "Reserving more memory.\n";
scores.reserve(20);  //reserve memory for 10 additional elements
cout << "Vector size is :" << scores.size() << endl;
cout << "Vector capacity is:" << scores.capacity() << endl;
```

Right after I declare and initialize the vector, this code reports that its size and capacity are both 10. However, after I reserve memory for 10 additional elements, the code reports that the vector's size is still 10 while its capacity is 20.

By using `reserve()` to keep a vector's capacity large enough for your purposes, you can delay memory reallocation to a time of your choosing.

Hint

As a beginning game programmer, it's good to be aware of how vector memory allocation works; however, don't obsess over it. The first game programs you'll write probably won't benefit from a more manual process of vector memory allocation.

Examining Element Insertion and Deletion

Adding or removing an element from the end of a vector using the `push_back()` or `pop_back()` member functions is extremely efficient. However, adding or removing an element at any other point in a vector (for example, using `insert()` or `erase()`) can require more work because you might have to move multiple elements to accommodate the insertion or deletion. With small vectors the overhead is usually insignificant, but with larger vectors (with, say, thousands of elements), inserting or erasing elements from the middle of a vector can cause a performance hit.

Fortunately, the STL offers another sequence container type, `list`, which allows for efficient insertion and deletion regardless of the sequence size. The important thing to remember is that one container type isn't the solution for every problem. Although `vector` is versatile and perhaps the most popular STL container type, there are times when another container type might make more sense.

Trap

Just because you want to insert or delete elements from the middle of a sequence, that doesn't mean you should abandon the vector. It might still be a good choice for your game program. It really depends on how you use the sequence. If your sequence is small or there are only a few insertions and deletions, then a vector might still be your best bet.

Examining Other STL Containers

The STL defines a variety of container types that fall into two basic categories: sequential and associative. With a *sequential container*, you can retrieve values in sequence, while an *associative container* lets you retrieve values based on keys. `vector` is an example of a sequential container.

How might you use these different container types? Consider an online, turn-based strategy game. You could use a sequential container to store a group of players that you want to cycle through in, well, sequence. On the other hand, you could use an associative container to retrieve player information in a random-access fashion by looking up a unique identifier, such as a player's IP address.

Finally, the STL defines container adaptors that adapt one of the sequence containers. *Container adaptors* represent standard computer science data structures. Although they are not official containers, they look and feel just like them. Table 4.1 lists the container types offered by the STL.

Table 4.1 STL Containers

Container	Type	Description
`deque`	Sequential	Double-ended queue
`list`	Sequential	Linear list
`map`	Associative	Collection of key/value pairs in which each key is associated with exactly one value

(*Continued*)

Table 4.1 STL Containers (*Continued*)

Container	Type	Description
`multimap`	Associative	Collection of key/value pairs in which each key may be associated with more than one value
`multiset`	Associative	Collection in which each element is not necessarily unique
`priority_queue`	Adaptor	Priority queue
`queue`	Adaptor	Queue
`set`	Associative	Collection in which each element is unique
`stack`	Adaptor	Stack
`vector`	Sequential	Dynamic array

Planning Your Programs

So far, all the programs you've seen have been pretty simple. The idea of formally planning any of them on paper probably seems like overkill. It's not. Planning your programs (even the small ones) will almost always result in time (and frustration) saved.

Programming is a lot like construction. Imagine a contractor building a house for you without a blueprint. Yikes! You might end up with a house that has 12 bathrooms, no windows, and a front door on the second floor. Plus, it probably would cost you 10 times the estimated price. Programming is the same way. Without a plan, you'll likely struggle through the process and waste time. You might even end up with a program that doesn't quite work.

Using Pseudocode

Many programmers sketch out their programs using *pseudocode*—a language that falls somewhere between English and a formal programming language. Anyone who understands English should be able to follow pseudocode. Here's an example: Suppose I want to make a million dollars. A worthy goal, but what do I do to achieve it? I need a plan. So I come up with one and put it in pseudocode.

```
If you can think of a new and useful product
    Then that's your product
```

```
Otherwise
    Repackage an existing product as your product
Make an infomercial about your product
Show the infomercial on TV
Charge $100 per unit of your product
Sell 10,000 units of your product
```

Even though anyone, even a non-programmer, can understand my plan, my pseudocode feels vaguely like a program. The first four lines resemble an `if` statement with an `else` clause, and that's intentional. When you write your plan, you should try to incorporate the feel of the code that you're representing with pseudocode.

Using Stepwise Refinement

Your programming plan might not be finished after only one draft. Often pseudocode needs multiple passes before it can be implemented in programming code. *Stepwise refinement* is one process used to rewrite pseudocode to make it ready for implementation. Stepwise refinement is pretty simple. Basically, it means, "Make it more detailed." By taking each step described in pseudocode and breaking it down into a series of simpler steps, the plan becomes closer to programming code. Using stepwise refinement, you keep breaking down each step until you feel the entire plan could be fairly easily translated into a program. As an example, take a step from my master plan to make a million dollars:

```
Create an infomercial about your product
```

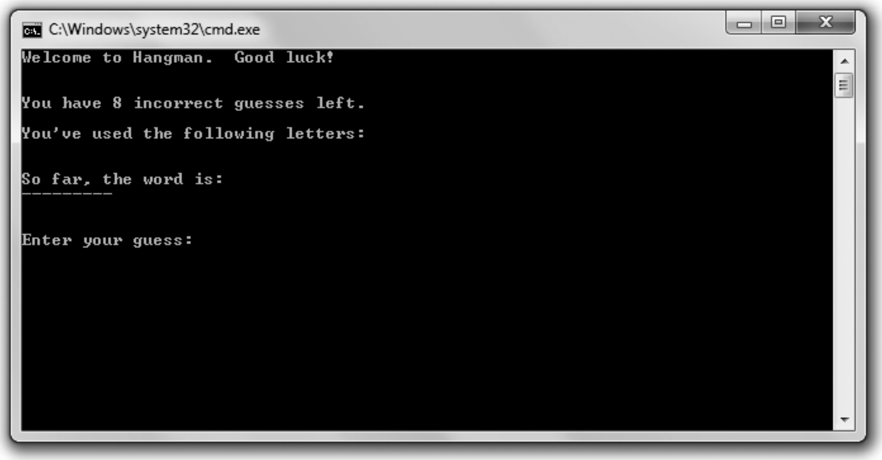
This might seem like too vague of a task. How do you create an infomercial? Using stepwise refinement, you can break down the single step into several others so it becomes:

```
Write a script for an infomercial about your product
Rent a TV studio for a day
Hire a production crew
Hire an enthusiastic audience
Film the infomercial
```

If you feel these five steps are clear and achievable, then that part of the pseudocode has been thoroughly refined. If you're still unclear about a step, refine it some more. Continue with this process and you will have a complete plan—and a million dollars.

Introducing Hangman

In the Hangman program, the computer picks a secret word and the player tries to guess it one letter at a time. The player is allowed eight incorrect guesses. If he or she fails to guess the word in time, the player is hanged and the game is over. Figure 4.6 shows the game.



```
Welcome to Hangman.  Good luck!

You have 8 incorrect guesses left.
You've used the following letters:

So far, the word is:
---------

Enter your guess:
```

Figure 4.6
The Hangman game in action.
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 4 folder; the filename is `hangman.cpp`.

Planning the Game

Before I write a single line in C++, I plan the game program using pseudocode.

```
Create a group of words
Pick a random word from the group as the secret word
While player hasn't made too many incorrect guesses and hasn't guessed the secret word
    Tell player how many incorrect guesses he or she has left
    Show player the letters he or she has guessed
    Show player how much of the secret word he or she has guessed
    Get player's next guess
    While player has entered a letter that he or she has already guessed
        Get player's guess
    Add the new guess to the group of used letters
    If the guess is in the secret word
        Tell the player the guess is correct
        Update the word guessed so far with the new letter
    Otherwise
        Tell the player the guess is incorrect
        Increment the number of incorrect guesses the player has made
```

```
If the player has made too many incorrect guesses
    Tell the player that he or she has been hanged
Otherwise
    Congratulate the player on guessing the secret word
```

Although the pseudocode doesn't account for every line of C++ I'll write, I think it does a good job describing what I need to do. Then I begin writing the program.

Setting Up the Program

As usual, I start with some comments and include the files I need.

```
// Hangman
// The classic game of hangman

#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
#include <ctime>
#include <cctype>

using namespace std;
```

Notice that I include a new file: `cctype`. It's part of the standard library, and it includes functions for converting characters to uppercase, which I use so I can compare apples to apples (uppercase to uppercase) when I compare individual characters.

Initializing Variables and Constants

Next, I start the `main()` function and initialize variables and constants for the game.

```
int main()
{
    //setup
    const int MAX_WRONG = 8;  //maximum number of incorrect guesses allowed

    vector<string> words;  //collection of possible words to guess
    words.push_back("GUESS");
    words.push_back("HANGMAN");
    words.push_back("DIFFICULT");

    srand(static_cast<unsigned int>(time(0)));
    random_shuffle(words.begin(), words.end());
    const string THE_WORD = words[0];            //word to guess
    int wrong = 0;                               //number of incorrect guesses
    string soFar(THE_WORD.size(), '-');          //word guessed so far
    string used = "";                            //letters already guessed

    cout << "Welcome to Hangman. Good luck!\n";
```

`MAX_WRONG` is the maximum number of incorrect guesses the player can make. `words` is a vector of possible words to guess. I randomize `words` using the `random_shuffle()` algorithm, and then I assign the first word in the vector to `THE_WORD`, which is the secret word the player must guess. `wrong` is the number of incorrect guesses the player has made. `soFar` is the word guessed so far by the player. `soFar` starts out as a series of dashes—one for each letter in the secret word. When the player guesses a letter that's in the secret word, I replace the dash at the corresponding position with the letter.

Entering the Main Loop

Next, I enter the main loop, which continues until the player has made too many incorrect guesses or has guessed the word.

```
    //main loop
    while ((wrong < MAX_WRONG) && (soFar != THE_WORD))
    {
        cout << "\n\nYou have " << (MAX_WRONG - wrong);
        cout << " incorrect guesses left.\n";
        cout << "\nYou've used the following letters:\n" << used << endl;
        cout << "\nSo far, the word is:\n" << soFar << endl;
```

Getting the Player's Guess

Next, I get the player's guess.

```
        char guess;
        cout << "\n\nEnter your guess: ";
        cin >> guess;
        guess = toupper(guess); //make uppercase since secret word in uppercase
        while (used.find(guess) != string::npos)
        {
            cout << "\nYou've already guessed " << guess << endl;
            cout << "Enter your guess: ";
            cin >> guess;
            guess = toupper(guess);
        }
        used += guess;

        if (THE_WORD.find(guess) != string::npos)
        {
            cout << "That's right! " << guess << " is in the word.\n";

            //update soFar to include newly guessed letter
            for (int i = 0; i < THE_WORD.length(); ++i)
            {
                if (THE_WORD[i] == guess)
                {
                    soFar[i] = guess;
                }
            }
        }
        else
        {
            cout << "Sorry, " << guess << " isn't in the word.\n";
            ++wrong;
        }
    }
```

I convert the guess to uppercase using the function `uppercase()`, which is defined in the file `cctype`. I do this so I can compare uppercase letters to uppercase letters when I'm checking a guess against the letters of the secret word.

If the player guesses a letter that he or she has already guessed, I make the player guess again. If the player guesses a letter correctly, I update the word guessed so far. Otherwise, I tell the player the guess is not in the secret word and I increase the number of incorrect guesses the player has made.

Ending the Game

At this point, the player has guessed the word or has made one too many incorrect guesses. Either way, the game is over.

```
    //shut down
    if (wrong == MAX_WRONG)
    {
        cout << "\nYou've been hanged!";
    }
    else
    {
        cout << "\nYou guessed it!";
    }

    cout << "\nThe word was " << THE_WORD << endl;

    return 0;
}
```

I congratulate the player or break the bad news that he or she has been hanged. Then I reveal the secret word.

Summary

In this chapter, you learned the following concepts:

- The Standard Template Library (STL) is a powerful collection of programming code that provides containers, algorithms, and iterators.
- Containers are objects that let you store and access collections of values of the same type.
- Algorithms defined in the STL can be used with their containers and provide common functions for working with groups of objects.
- Iterators are objects that identify elements in containers and can be manipulated to move among elements.
- Iterators are the key to using containers to their fullest. Many of the container member functions require iterators, and the STL algorithms require them too.
- To get the value referenced by an iterator, you must dereference the iterator using the dereference operator (*).
- A vector is one kind of sequential container provided by the STL. It acts like a dynamic array.
- It's very efficient to iterate through a vector. It's also very efficient to insert or remove an element from the end of a vector.
- It can be inefficient to insert or delete elements from the middle of a vector, especially if the vector is large.
- Pseudocode, which falls somewhere between English and a programming language, is used to plan programs.
- Stepwise refinement is a process used to rewrite pseudocode to make it ready for implementation.

Questions and Answers

Q: Why is the STL important?
A: Because it saves game programmers time and effort. The STL provides commonly used container types and algorithms.

Q: Is the STL fast?
A: Definitely. The STL has been honed by hundreds of programmers to eke out as much performance as possible on each supported platform.

Q: When should I use a vector instead of an array?
A: Almost always. Vectors are efficient and flexible. They do require a little more memory than arrays, but this tradeoff is almost always worth the benefits.

Q: Is a vector as fast as an array?
A: Accessing a vector element can be just as fast as accessing an array element. Also, iterating through a vector can be just as fast as iterating through an array.

Q: If I can use the subscripting operator with vectors, why would I ever need iterators?
A: There are several reasons. First, many of the `vector` member functions require iterators. (`insert()` and `erase()` are two examples.) Second, STL algorithms require iterators. And third, you can't use the subscripting operator with most of the STL containers, so you'll need to learn to use iterators sooner or later.

Q: Which is the best way to access elements of a vector—through iterators or through the subscripting operator?
A: It depends. If you need random-element access, then the subscripting operator is a natural fit. If you need to use STL algorithms, then you must use iterators.

Q: What about iterating through the elements of a vector? Should I use the subscripting operator or an iterator?
A: You can use either method. However, an advantage of using an iterator is that it gives you the flexibility to substitute a different STL container in place of a vector (such as a list) without much code changing.

Q: Why does the STL define more than one sequential container type?
A: Different sequential container types have different performance properties. They're like tools in a toolbox; each tool is best suited for a different job.

Q: What are container adaptors?
A: Container adaptors are based on one of the STL sequence containers; they represent standard computer data structures. Although they are not official containers, they look and feel just like them.

Q: What's a stack?
A: A data structure in which elements are removed in the reverse order from how they were added. This means that the last element added is the first one removed. This is just like a real-life stack, from which you remove the last item you placed on the top of the stack.

Q: What's a queue?
A: A data structure in which elements are removed in the same order they were added. This is just like a real-life queue, such as a line of people in which the first person in line is served first.

Q: What's a double-ended queue?
A: A queue in which elements can be added or removed from either end.

Q: What's a priority queue?
A: A data structure that supports finding and removing the element with the highest priority.

Q: When would I use pseudocode?
A: Any time you want to plan a program or section of code.

Q: When would I use stepwise refinement?
A: When you want to get even more detailed with your pseudocode.

Discussion Questions

1. Why should a game programmer use the STL?
2. What are the advantages of a vector over an array?
3. What types of game objects might you store with a vector?
4. How do performance characteristics of a container type affect the decision to use it?
5. Why is program planning important?

Exercises

1. Write a program using vectors and iterators that allows a user to maintain a list of his or her favorite games. The program should allow the user to list all game titles, add a game title, and remove a game title.
2. Assuming that `scores` is a vector that holds elements of type `int`, what's wrong with the following code snippet (meant to increment each element)?

   ```
   vector<int>::iterator iter;
   //increment each score
   for (iter = scores.begin(); iter != scores.end(); ++iter)
   {
       iter++;
   }
   ```
3. Write pseudocode for the Word Jumble game from Chapter 3.

This page intentionally left blank