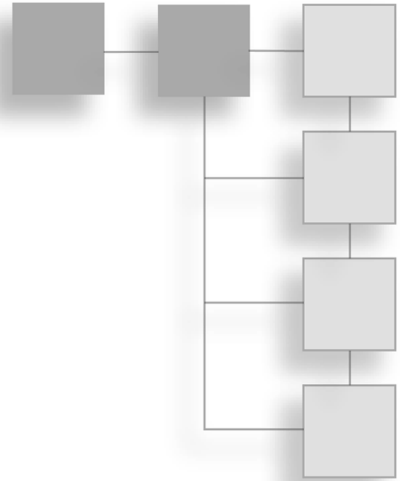


## CHAPTER 6



# REFERENCES: TIC-TAC-TOE

The concept of references is simple, but its implications are profound. In this chapter, you'll learn about references and how they can help you write more efficient game code. Specifically, you'll learn to:

- Create references
- Access and change referenced values
- Pass references to functions to alter argument values or for efficiency
- Return references from a function for efficiency or to alter values

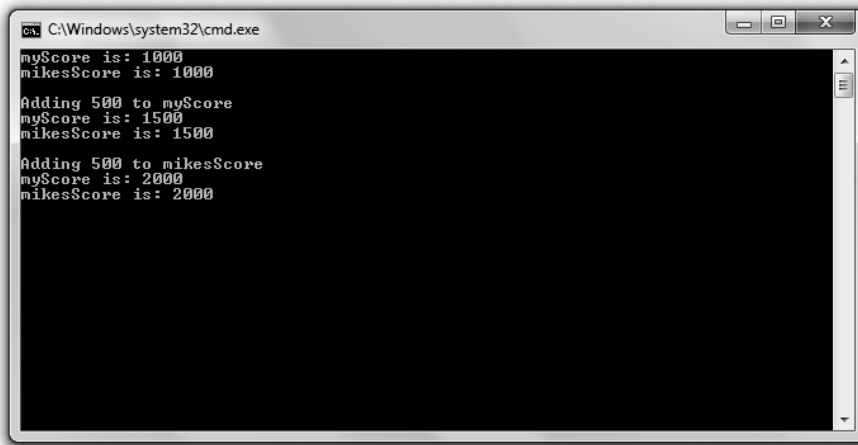
## USING REFERENCES

A *reference* provides another name for a variable. Whatever you do to a reference is done to the variable to which it refers. You can think of a reference as a nickname for a variable—another name that the variable goes by. In the first program in this chapter, I'll show you how to create references. Then, in the next few programs, I'll show you why you'd want to use references and how they can improve your game programs.

## Introducing the Referencing Program

The Referencing program demonstrates references. The program declares and initializes a variable to hold a score and then creates a reference that refers to the variable. The program displays the score using the variable and the reference to show that they access the

same single value. Next, the program shows that this single value can be altered through either the variable or the reference. Figure 6.1 illustrates the program.



```

C:\Windows\system32\cmd.exe
myScore is: 1000
mikesScore is: 1000
Adding 500 to myScore
myScore is: 1500
mikesScore is: 1500
Adding 500 to mikesScore
myScore is: 2000
mikesScore is: 2000

```

**Figure 6.1**

The variable `myScore` and the reference `mikesScore` are both names for the single score value.

Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website ([www.cengageptr.com/downloads](http://www.cengageptr.com/downloads)). The program is in the Chapter 6 folder; the filename is `referencing.cpp`.

```

// Referencing
// Demonstrates using references

#include <iostream>

using namespace std;

int main()
{
    int myScore = 1000;
    int& mikesScore = myScore; //create a reference

    cout << "myScore is: " << myScore << "\n";
    cout << "mikesScore is: " << mikesScore << "\n\n";

    cout << "Adding 500 to myScore\n";
    myScore += 500;
    cout << "myScore is: " << myScore << "\n";
}

```

```

cout << "mikesScore is: " << mikesScore << "\n\n";
cout << "Adding 500 to mikesScore\n";
mikesScore += 500;
cout << "myScore is: " << myScore << "\n";
cout << "mikesScore is: " << mikesScore << "\n\n";
return 0;
}

```

## Creating References

The first thing I do in `main()` is create a variable to hold my score.

```
int myScore = 1000;
```

Then I create a reference that refers to `myScore`.

```
int& mikesScore = myScore; //create a reference
```

The preceding line declares and initializes `mikesScore`, a reference that refers to `myScore`. `mikesScore` is an alias for `myScore`. `mikesScore` does not hold its own `int` value; it's simply another way to get at the `int` value that `myScore` holds.

To declare and initialize a reference, start with the type of value to which the reference will refer, followed by the reference operator (`&`), followed by the reference name, followed by `=`, followed by the variable to which the reference will refer.

### Trick

---

Sometimes programmers prefix a reference name with the letter "r" to remind them that they're working with a reference. A programmer might include the following lines:

```
int playerScore = 1000;
int& rScore = playerScore;
```

---

One way to understand references is to think of them as nicknames. For example, suppose you've got a friend named Eugene, and he (understandably) asks to be called by a nickname—Gibby (not much of an improvement, but it's what Eugene wants). So when you're at a party with your friend, you can call him over using either Eugene or Gibby. Your friend is only one person, but you can call him using either his name or a nickname. This is the same as how a variable and a reference to that variable work. You can

get to a single value stored in a variable by using its variable name or the name of a reference to that variable. Finally, whatever you do, try not to name your variables Eugene—for their sakes.

### Trap

---

Because a reference must always refer to another value, you must initialize the reference when you declare it. If you don't, you'll get a compile error. The following line is quite illegal:

```
int& mikesScore; //don't try this at home!
```

---

## Accessing Referenced Values

Next, I send both `myScore` and `mikesScore` to `cout`.

```
cout << "myScore is: " << myScore << "\n";
cout << "mikesScore is: " << mikesScore << "\n\n";
```

Both lines of code display 1000 because they each access the same single chunk of memory that stores the number 1000. Remember, there is only one value, and it is stored in the variable `myScore`. `mikesScore` simply provides another way to get to that value.

## Altering Referenced Values

Next, I increase the value of `myScore` by 500.

```
myScore += 500;
```

When I send `myScore` to `cout`, 1500 is displayed, just as you'd expect. When I send `mikesScore` to `cout`, 1500 is also displayed. Again, that's because `mikesScore` is just another name for the variable `myScore`. In essence, I'm sending the same variable to `cout` both times.

Next, I increase `mikesScore` by 500.

```
mikesScore += 500;
```

Because `mikesScore` is just another name for `myScore`, the preceding line of code increases the value of `myScore` by 500. So when I next send `myScore` to `cout`, 2000 is displayed. When I send `mikesScore` to `cout`, 2000 is displayed again.

**Trap**

A reference always refers to the variable with which it was initialized. You can't reassign a reference to refer to another variable so, for example, the results of the following code might not be obvious.

```
int myScore = 1000;
int& mikesScore = myScore;
int larrysScore = 2500;
mikesScore = larrysScore; //may not do what you think!
```

The line `mikesScore = larrysScore;` does not reassign the reference `mikesScore` so it refers to `larrysScore` because a reference can't be reassigned. However, because `mikesScore` is just another name for `myScore`, the code `mikesScore = larrysScore;` is equivalent to `myScore = larrysScore;`, which assigns 2500 to `myScore`. And after all is said and done, `myScore` becomes 2500 and `mikesScore` still refers to `myScore`.

## PASSING REFERENCES TO ALTER ARGUMENTS

Now that you've seen how references work, you might be wondering why you'd ever use them. Well, references come in quite handy when you are passing variables to functions because when you pass a variable to a function, the function gets a copy of the variable. This means that the original variable you passed (called the *argument variable*) can't be changed. Sometimes this might be exactly what you want because it keeps the argument variable safe and unalterable. But other times you might want to change an argument variable from inside the function to which it was passed. You can accomplish this by using references.

### Introducing the Swap Program

The Swap program defines two variables—one that holds my pitifully low score and another that holds your impressively high score. After displaying the scores, the program calls a function meant to swap the scores. But because only copies of the score values are sent to the function, the argument variables that hold the scores are unchanged. Next, the program calls another swap function. This time, through the use of references, the argument variables' values are successfully exchanged—giving me the great big score and leaving you with the small one. Figure 6.2 shows the program in action.



```

C:\Windows\system32\cmd.exe
Original values
myScore: 150
yourScore: 1000

Calling badSwap()
myScore: 150
yourScore: 1000

Calling goodSwap()
myScore: 1000
yourScore: 150

```

**Figure 6.2**

Passing references allows `goodSwap()` to alter the argument variables.

Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website ([www.cengageptr.com/downloads](http://www.cengageptr.com/downloads)). The program is in the Chapter 6 folder; the filename is `swap.cpp`.

```

// Swap
// Demonstrates passing references to alter argument variables

#include <iostream>

using namespace std;

void badSwap(int x, int y);
void goodSwap(int& x, int& y);

int main()
{
    int myScore = 150;
    int yourScore = 1000;
    cout << "Original values\n";
    cout << "myScore: " << myScore << "\n";
    cout << "yourScore: " << yourScore << "\n\n";

    cout << "Calling badSwap()\n";
    badSwap(myScore, yourScore);
    cout << "myScore: " << myScore << "\n";
    cout << "yourScore: " << yourScore << "\n\n";
}

```

```

    cout << "Calling goodSwap()\n";
    goodSwap(myScore, yourScore);
    cout << "myScore: " << myScore << "\n";
    cout << "yourScore: " << yourScore << "\n";

    return 0;
}

void badSwap(int x, int y)
{
    int temp = x;
    x = y;
    y = temp;
}

void goodSwap(int& x, int& y)
{
    int temp = x;
    x = y;
    y = temp;
}

```

## Passing by Value

After declaring and initializing `myScore` and `yourScore`, I send them to `cout`. As you'd expect, 150 and 1000 are displayed. Next, I call `badSwap()`.

When you specify a parameter the way you've seen so far (as an ordinary variable, not as a reference), you're indicating that the argument for that parameter will be *passed by value*, meaning that the parameter will get a *copy* of the argument variable and not access to the argument variable itself. By looking at the function header of `badSwap()`, you can tell that a call to the function passes both arguments by value.

```
void badSwap(int x, int y)
```

This means that when I call `badSwap()` with the following line, copies of `myScore` and `yourScore` are sent to the parameters, `x` and `y`.

```
    badSwap(myScore, yourScore);
```

Specifically, `x` is assigned 150 and `y` is assigned 1000. As a result, nothing I do with `x` and `y` in the function `badSwap()` will have any effect on `myScore` and `yourScore`.

When the guts of `badSwap()` execute, `x` and `y` do exchange values—`x` becomes 1000 and `y` becomes 150. However, when the function ends, both `x` and `y` go out of scope and cease to exist. Control then returns to `main()`, where `myScore` and `yourScore` haven't changed. Then, when I send `myScore` and `yourScore` to `cout`, 150 and 1000 are displayed again. Sadly, I still have the small score and you still have the large one.

## Passing by Reference

It's possible to give a function access to an argument variable by passing a parameter a reference to the argument variable. As a result, anything done to the parameter will be done to the argument variable. To *pass by reference*, you must first declare the parameter as a reference.

You can tell that a call to `goodSwap()` passes both arguments by reference by looking at the function header.

```
void goodSwap(int& x, int& y)
```

This means that when I call `goodSwap()` with the following line, the parameter `x` will refer to `myScore`, and the parameter `y` will refer to `yourScore`.

```
goodSwap(myScore, yourScore);
```

This means that `x` is just another name for `myScore`, and `y` is just another name for `yourScore`. When `goodSwap()` executes and `x` and `y` exchange values, what really happens is that `myScore` and `yourScore` exchange values.

After the function ends, control returns to `main()`, where I send `myScore` and `yourScore` to `cout`. This time 1000 and 150 are displayed. The variables have exchanged values. I've taken the large score and left you with the small one. Success at last!

## PASSING REFERENCES FOR EFFICIENCY

Passing a variable by value creates some overhead because you must copy the variable before you assign it to a parameter. When we're talking about variables of simple, built-in types, such as an `int` or a `float`, the overhead is negligible. But a large object, such as one that represents an entire 3D world, could be expensive to copy. Passing by reference, on the other hand, is efficient because you don't make a copy of an argument variable. Instead, you simply provide access to the existing object through a reference.



## Introducing the Inventory Displayer Program

The Inventory Displayer program creates a vector of strings that represents a hero's inventory. The program then calls a function that displays the inventory. The program passes the displayer function the vector of items as a reference, so it's an efficient call; the vector isn't copied. However, there's a new wrinkle. The program passes the vector as a special kind of reference that prohibits the displayer function from changing the vector. Figure 6.3 shows you the program.



**Figure 6.3**

The vector inventory is passed in a safe and efficient way to the function that displays the hero's items.

Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website ([www.cengageptr.com/downloads](http://www.cengageptr.com/downloads)). The program is in the Chapter 6 folder; the filename is `inventory_displayer.cpp`.

```
// Inventory Displayer
// Demonstrates constant references

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

using namespace std;

//parameter vec is a constant reference to a vector of strings
void display(const vector<string>& inventory);
```

```

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

    display(inventory);

    return 0;
}

//parameter vec is a constant reference to a vector of strings
void display(const vector<string>& vec)
{
    cout << "Your items:\n";
    for (vector<string>::const_iterator iter = vec.begin();
        iter != vec.end(); ++iter)
    {
        cout << *iter << endl;
    }
}

```

## Understanding the Pitfalls of Reference Passing

One way to efficiently give a function access to a large object is to pass it by reference. However, this introduces a potential problem. As you saw in the Swap program, it opens up an argument variable to being changed. But what if you don't want to change the argument variable? Is there a way to take advantage of the efficiency of passing by reference while protecting an argument variable's integrity? Yes, there is. The answer is to pass a constant reference.

### Hint

---

In general, you should avoid changing an argument variable. Try to write functions that send back new information to the calling code through a return value.

---

## Declaring Parameters as Constant References

The function `display()` shows the contents of the hero's inventory. In the function's header I specify one parameter—a constant reference to a vector of string objects named `vec`.

```
void display(const vector<string>& vec)
```

A *constant reference* is a restricted reference. It acts like any other reference, except you can't use it to change the value to which it refers. To create a constant reference, simply put the keyword `const` before the type in the reference declaration.

What does this all mean for the function `display()`? Because the parameter `vec` is a constant reference, it means `display()` can't change `vec`. In turn, this means that `inventory` is safe; it can't be changed by `display()`. In general, you can efficiently pass an argument to a function as a constant reference so it's accessible, but not changeable. It's like providing the function read-only access to the argument. Although constant references are very useful for specifying function parameters, you can use them anywhere in your program.

---

### Hint

A constant reference comes in handy in another way. If you need to assign a constant value to a reference, you have to assign it to a constant reference. (A non-constant reference won't do.)

---

## Passing a Constant Reference

Back in `main()`, I create `inventory` and then call `display()` with the following line, which passes the vector as a constant reference.

```
display(inventory);
```

This results in an efficient and safe function call. It's efficient because only a reference is passed; the vector is not copied. It's safe because the reference to the vector is a constant reference; `inventory` can't be changed by `display()`.

---

### Trap

You can't modify a parameter marked as a constant reference. If you try, you'll generate a compile error.

---

Next, `display()` lists the elements in the vector using a constant reference to `inventory`. Then control returns to `main()` and the program ends.

## DECIDING HOW TO PASS ARGUMENTS

At this point you've seen three different ways to pass arguments—by value, as a reference, and as a constant reference. So how do you decide which method to use? Here are some guidelines:

- **By value.** Pass by value when an argument variable is one of the fundamental built-in types, such as `bool`, `int`, or `float`. Objects of these types are so small that passing by reference doesn't result in any gain in efficiency. You should also pass by value when

you want the computer to make a copy of a variable. You might want to use a copy if you plan to alter a parameter in a function, but you don't want the actual argument variable to be affected.

- **As a constant reference.** Pass a constant reference when you want to efficiently pass a value that you don't need to change.
- **As a reference.** Pass a reference only when you want to alter the value of the argument variable. However, you should try to avoid changing argument variables whenever possible.

## RETURNING REFERENCES

Just like when you pass a value, when you return a value from a function, you're really returning a copy of the value. Again, for values of the basic built-in types, this isn't a big deal. However, it can be an expensive operation if you're returning a large object. Returning a reference is an efficient alternative.

## Introducing the Inventory Referencer Program

The Inventory Referencer program demonstrates returning references. The program displays the elements of a vector that holds a hero's inventory by using returned references. Then the program changes one of the items through a returned reference. Figure 6.4 shows the results of the program.

```

C:\Windows\system32\cmd.exe
Sending the returned reference to cout:
sword
Assigning the returned reference to another reference.
Sending the new reference to cout:
armor
Assigning the returned reference to a string object.
Sending the new string object to cout:
shield
Altering an object through a returned reference.
Sending the altered object to cout:
Healing Potion
-

```

**Figure 6.4**

The items in the hero's inventory are displayed and changed by using returned references.

Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website ([www.cengageptr.com/downloads](http://www.cengageptr.com/downloads)). The program is in the Chapter 6 folder; the filename is `inventory_referencer.cpp`.

```
// Inventory Referencer
// Demonstrates returning a reference

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

using namespace std;

//returns a reference to a string
string& refToElement(vector<string>& inventory, int i);

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

    //displays string that the returned reference refers to
    cout << "Sending the returned reference to cout:\n";
    cout << refToElement(inventory, 0) << "\n\n";

    //assigns one reference to another -- inexpensive assignment
    cout << "Assigning the returned reference to another reference.\n";
    string& rStr = refToElement(inventory, 1);
    cout << "Sending the new reference to cout:\n";
    cout << rStr << "\n\n";

    //copies a string object -- expensive assignment
    cout << "Assigning the returned reference to a string object.\n";
    string str = refToElement(inventory, 2);
    cout << "Sending the new string object to cout:\n";
    cout << str << "\n\n";

    //altering the string object through a returned reference
    cout << "Altering an object through a returned reference.\n";
    rStr = "Healing Potion";
    cout << "Sending the altered object to cout:\n";
    cout << inventory[1] << endl;

    return 0;
}
```

```
//returns a reference to a string
string& refToElement(vector<string>& vec, int i)
{
    return vec[i];
}
```

## Returning a Reference

Before you can return a reference from a function, you must specify that you're returning one. That's what I do in the `refToElement()` function header.

```
string& refToElement(vector<string>& inventory, int i)
```

By using the reference operator in `string&` when I specify the return type, I'm saying that the function will return a reference to a `string` object (not a `string` object itself). You can use the reference operator as I did to specify that a function returns a reference to an object of a particular type. Simply put the reference operator after the type name.

The body of the function `refToElement()` contains only one statement, which returns a reference to the element at position `i` in the vector.

```
    return vec[i];
```

Notice that there's nothing in the `return` statement to indicate that the function returns a reference. The function header and prototype determine whether a function returns an object or a reference to an object.

### Trap

---

Although returning a reference can be an efficient way to send information back to a calling function, you have to be careful not to return a reference to an out-of-scope object—an object that ceases to exist. For example, the following function returns a reference to a `string` object that no longer exists after the function ends—and that's illegal.

```
string& badReference()
{
    string local = "This string will cease to exist once the function ends.";
    return local;
}
```

One way to avoid this type of problem is to never return a reference to a local variable.

---

## Displaying the Value of a Returned Reference

After creating `inventory`, a vector of items, I display the first item through a returned reference.

```
cout << refToElement(inventory, 0) << "\n\n";
```

The preceding code calls `refToElement()`, which returns a reference to the element at position 0 of `inventory` and then sends that reference to `cout`. As a result, `sword` is displayed.

## Assigning a Returned Reference to a Reference

Next, I assign a returned reference to another reference with the following line, which takes a reference to the element in position 1 of `inventory` and assigns it to `rStr`.

```
string& rStr = refToElement(inventory, 1);
```

This is an efficient assignment because assigning a reference to a reference does not involve the copying of an object. Then I send `rStr` to `cout`, and `armor` is displayed.

## Assigning a Returned Reference to a Variable

Next, I assign a returned reference to a variable.

```
string str = refToElement(inventory, 2);
```

The preceding code doesn't assign a reference to `str`. It can't, because `str` is a `string` object. Instead, the code copies the element to which the returned reference refers (the element in position 2 of `inventory`) and assigns that new copy of the `string` object to `str`. Because this kind of assignment involves copying an object, it's more expensive than assigning one reference to another. Sometimes the cost of copying an object this way is perfectly acceptable, but you should be aware of the extra overhead associated with this kind of assignment and avoid it when necessary.

Next, I send the new `string` object, `str`, to `cout`, and `shield` is displayed.

## Altering an Object through a Returned Reference

You can also alter the object to which a returned reference refers. This means you can change the hero's inventory through `rStr`, as in the following line of code.

```
rStr = "Healing Potion";
```

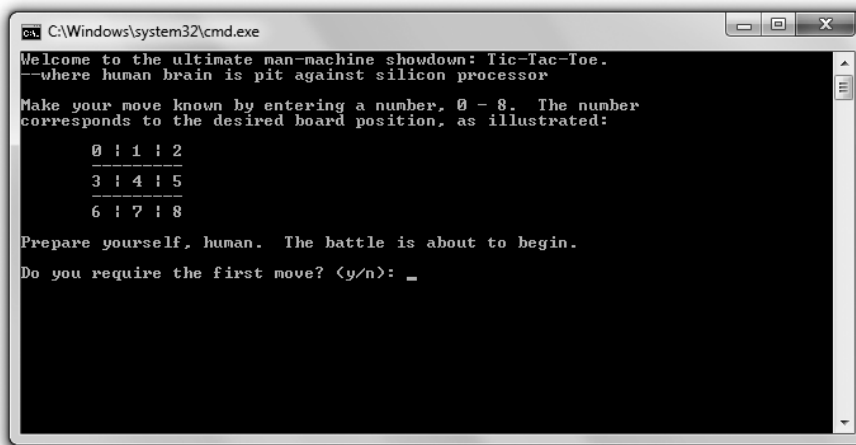
Because `rStr` refers to the element in position 1 of `inventory`, this code changes `inventory[1]` so it's equal to "Healing Potion". To prove it, I display the element using the following line, which does indeed show Healing Potion.

```
cout << inventory[1] << endl;
```

If I want to protect `inventory` so a reference returned by `refToElement()` can't be used to change the vector, I should specify the return type of the function as a constant reference.

## INTRODUCING THE TIC-TAC-TOE GAME

In this chapter project, you'll learn how to create a computer opponent using a dash of *AI* (*Artificial Intelligence*). In the game, the player and computer square off in a high-stakes, man-versus-machine showdown of Tic-Tac-Toe. The computer plays a formidable (although not perfect) game and comes with enough attitude to make any match fun. Figure 6.5 shows the start of a match.



**Figure 6.5**  
The computer is full of...confidence.  
Used with permission from Microsoft.

## Planning the Game

This game is your most ambitious project yet. You certainly have all the skills you need to create it, but I'm going to go through a longer planning section to help you get the big picture and understand how to create a larger program. Remember, the most important



part of programming is planning to program. Without a roadmap, you'll never get to where you want to go (or it'll take you a lot longer as you travel the scenic route).

### In the Real World

---

Game designers work countless hours on concept papers, design documents, and prototypes before programmers write any game code. Once the design work is done, the programmers start their work—more planning. It's only after programmers write their own technical designs that they then begin coding in earnest. The moral of this story? Plan. It's easier to scrap a blueprint than a 50-story building.

---

### *Writing the Pseudocode*

It's back to your favorite language that's not really a language—pseudocode. Because I'll use functions for most of the tasks in the program, I can afford to think about the code at a rather abstract level. Each line of pseudocode should feel like one function call. Later, all I'll have to do is write the functions that the plan implies. Here's the pseudocode:

```
Create an empty Tic-Tac-Toe board
Display the game instructions
Determine who goes first
Display the board
While nobody has won and it's not a tie
    If it's the human's turn
        Get the human's move
        Update the board with the human's move
    Otherwise
        Calculate the computer's move
        Update the board with the computer's move
    Display the board
    Switch turns
Congratulate the winner or declare a tie
```

### *Representing the Data*

All right, I've got a good plan, but it is rather abstract and talks about throwing around different elements that aren't really defined in my mind yet. I see the idea of making a move as placing a piece on a game board. But how exactly am I going to represent the game board? Or a piece? Or a move?

Since I'm going to display the game board on the screen, why not just represent a piece as a single character—an X or an O? An empty piece could just be a space. Therefore, the board itself could be a vector of chars. There are nine squares on a Tic-Tac-Toe board,

so the vector should have nine elements. Each square on the board will correspond to an element in the vector. Figure 6.6 illustrates what I mean.

0	1	2
3	4	5
6	7	8

**Figure 6.6**  
Each square number corresponds to a position in the vector that represents the board.

Each square or position on the board is represented by a number, 0–8. That means the vector will have nine elements, giving it position numbers 0–8. Because each move indicates a square where a piece should be placed, a move is also just a number, 0–8. That means a move could be represented as an `int`.

The side the player and computer play could also be represented by a `char`—either an `'X'` or an `'O'`, just like a game piece. A variable to represent the side of the current turn would also be a `char`, either an `'X'` or an `'O'`.

**Creating a List of Functions**

The pseudocode inspires the different functions I’ll need. I created a list of them, thinking about what each will do, what parameters they’ll have, and what values they’ll return. Table 6.1 shows the results of my efforts.

Table 6.1 Tic-Tac-Toe Functions	
Function	Description
<code>void instructions()</code>	Displays the game instructions.
<code>char askYesNo(string question)</code>	Asks a yes or no question. Receives a question. Returns either a <code>'y'</code> or an <code>'n'</code> .
<code>int askNumber(string question, int high, int low = 0)</code>	Asks for a number within a range. Receives a question, a low number, and a high number. Returns a number in the range from <code>low</code> to <code>high</code> .

<code>char humanPiece()</code>	Determines the human's piece. Returns either an 'X' or an 'O'.
<code>char opponent(char <i>piece</i>)</code>	Calculates the opposing piece given a piece. Receives either an 'X' or an 'O'. Returns either an 'X' or an 'O'.
<code>void displayBoard(const vector&lt;char&gt; &amp; <i>board</i>)</code>	Displays the board on the screen. Receives a board.
<code>char winner(const vector&lt;char&gt;&amp; <i>board</i>)</code>	Determines the game winner. Receives a board. Returns an 'X', 'O', 'T' (to indicate a tie), or 'N' (to indicate that no one has won yet).
<code>bool isLegal(const vector&lt;char&gt;&amp; <i>board</i>, int <i>move</i>)</code>	Determines whether a move is legal. Receives a board and a move. Returns either true or false.
<code>int humanMove(const vector&lt;char&gt;&amp; <i>board</i>, char <i>human</i>)</code>	Gets the human's move. Receives a board and the human's piece. Returns the human's move.
<code>int computerMove(vector&lt;char&gt; <i>board</i>, char <i>computer</i>)</code>	Calculates the computer's move. Receives a board and the computer's piece. Returns the computer's move.
<code>void announceWinner(char <i>winner</i>, char <i>computer</i>, char <i>human</i>)</code>	Congratulates the winner or declares a tie. Receives the winning side, the computer's piece, and the human's piece.

## Setting Up the Program

You can download the code for this program from the Cengage Learning website ([www.cengageptr.com/downloads](http://www.cengageptr.com/downloads)). The program is in the Chapter 6 folder; the filename is `tic-tac-toe.cpp`. I'll go over the code here, section by section.

The first thing I do in the program is include the files I need, define some global constants, and write my function prototypes.

```
// Tic-Tac-Toe
// Plays the game of tic-tac-toe against a human opponent

#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
```

```

using namespace std;

// global constants
const char X = 'X';
const char O = 'O';
const char EMPTY = ' ';
const char TIE = 'T';
const char NO_ONE = 'N';

// function prototypes
void instructions();
char askYesNo(string question);
int askNumber(string question, int high, int low = 0);
char humanPiece();
char opponent(char piece);
void displayBoard(const vector<char>& board);
char winner(const vector<char>& board);
bool isLegal(const vector<char>& board, int move);
int humanMove(const vector<char>& board, char human);
int computerMove(vector<char> board, char computer);
void announceWinner(char winner, char computer, char human);

```

In the global constants section, `X` is shorthand for the char `'X'`, one of the two pieces in the game. `O` represents the char `'O'`, the other piece in the game. `EMPTY`, also a char, represents an empty square on the board. It's a space because when it's displayed, it will look like an empty square. `TIE` is a char that represents a tie game. And `NO_ONE` is a char used to represent neither side of the game, which I use to indicate that no one has won yet.

## The main( ) Function

As you can see, the `main()` function is almost exactly the pseudocode I created earlier.

```

// main function
int main()
{
    int move;
    const int NUM_SQUARES = 9;
    vector<char> board(NUM_SQUARES, EMPTY);

    instructions();
    char human = humanPiece();
    char computer = opponent(human);
    char turn = X;

```

```

displayBoard(board);
while (winner(board) == NO_ONE)
{
    if (turn == human)
    {
        move = humanMove(board, human);
        board[move] = human;
    }
    else
    {
        move = computerMove(board, computer);
        board[move] = computer;
    }
    displayBoard(board);
    turn = opponent(turn);
}

announceWinner(winner(board), computer, human);

return 0;
}

```

## The instructions( ) Function

This function displays the game instructions and gives the computer opponent a little attitude.

```

void instructions()
{
    cout << "Welcome to the ultimate man-machine showdown: Tic-Tac-Toe.\n";
    cout << "--where human brain is pit against silicon processor\n\n";

    cout << "Make your move known by entering a number, 0 - 8. The number\n";
    cout << "corresponds to the desired board position, as illustrated:\n\n";

    cout << " 0 | 1 | 2\n";
    cout << " -----\n";
    cout << " 3 | 4 | 5\n";
    cout << " -----\n";
    cout << " 6 | 7 | 8\n\n";

    cout << "Prepare yourself, human. The battle is about to begin.\n\n";
}

```

## The askYesNo( ) Function

This function asks a yes or no question. It keeps asking the question until the player enters either a y or an n. It receives a question and returns either a 'y' or an 'n'.

```
char askYesNo(string question)
{
    char response;
    do
    {
        cout << question << " (y/n): ";
        cin >> response;
    } while (response != 'y' && response != 'n');
    return response;
}
```

## The askNumber( ) Function

This function asks for a number within a range and keeps asking until the player enters a valid number. It receives a question, a high number, and a low number. It returns a number within the range specified.

```
int askNumber(string question, int high, int low)
{
    int number;
    do
    {
        cout << question << " (" << low << " - " << high << "): ";
        cin >> number;
    } while (number > high || number < low);
    return number;
}
```

If you take a look at this function's prototype, you can see that the low number has a default value of 0. I take advantage of this fact when I call the function later in the program.

## The humanPiece( ) Function

This function asks the player if he wants to go first, and returns the human's piece based on that choice. As the great tradition of Tic-Tac-Toe dictates, the X goes first.

```

char humanPiece()
{
    char go_first = askYesNo("Do you require the first move?");
    if (go_first == 'y')
    {
        cout << "\nThen take the first move. You will need it.\n";
        return X;
    }
    else
    {
        cout << "\nYour bravery will be your undoing...I will go first.\n";
        return O;
    }
}

```

## The opponent( ) Function

This function gets a piece (either an 'X' or an 'O') and returns the opponent's piece (either an 'X' or an 'O').

```

char opponent(char piece)
{
    if (piece == X)
    {
        return O;
    }
    else
    {
        return X;
    }
}

```

## The displayBoard( ) Function

This function displays the board passed to it. Because each element in the board is a space, an 'X', or an 'O', the function can display each one. I use a few other characters on my keyboard to draw a decent-looking Tic-Tac-Toe board.

```

void displayBoard(const vector<char>& board)
{
    cout << "\n\t" << board[0] << " | " << board[1] << " | " << board[2];
    cout << "\n\t" << "-----";
}

```

```

cout << "\n\t" << board[3] << " | " << board[4] << " | " << board[5];
cout << "\n\t" << "-----";
cout << "\n\t" << board[6] << " | " << board[7] << " | " << board[8];
cout << "\n\n";
}

```

Notice that the vector that represents the board is passed through a constant reference. This means that the vector is passed efficiently; it is not copied. It also means that the vector is safeguarded against any changes. Since I plan to simply display the board and not change it in this function, this is perfect.

## The winner( ) Function

This function receives a board and returns the winner. There are four possible values for a winner. The function will return either `X` or `O` if one of the players has won. If every square is filled and no one has won, it returns `TIE`. Finally, if no one has won and there is at least one empty square, the function returns `NO_ONE`.

```

char winner(const vector<char>& board)
{
    // all possible winning rows
    const int WINNING_ROWS[8][3] = { {0, 1, 2},
                                       {3, 4, 5},
                                       {6, 7, 8},
                                       {0, 3, 6},
                                       {1, 4, 7},
                                       {2, 5, 8},
                                       {0, 4, 8},
                                       {2, 4, 6} };
}

```

The first thing to notice is that the vector that represents the board is passed through a constant reference. This means that the vector is passed efficiently; it is not copied. It also means that the vector is safeguarded against any change.

In this initial section of the function, I define a constant, two-dimensional array of ints called `WINNING_ROWS`, which represents all eight ways to get three in a row and win the game. Each winning row is represented by a group of three numbers—three board positions that form a winning row. For example, the group `{0, 1, 2}` represents the top row—board positions 0, 1, and 2. The next group, `{3, 4, 5}`, represents the middle row—board positions 3, 4, and 5. And so on....

Next, I check to see whether either player has won.



```

const int TOTAL_ROWS = 8;
// if any winning row has three values that are the same (and not EMPTY),
// then we have a winner
for(int row = 0; row < TOTAL_ROWS; ++row)
{
    if ( (board[WINNING_ROWS[row][0]] != EMPTY) &&
        (board[WINNING_ROWS[row][0]] == board[WINNING_ROWS[row][1]]) &&
        (board[WINNING_ROWS[row][1]] == board[WINNING_ROWS[row][2]]) )
    {
        return board[WINNING_ROWS[row][0]];
    }
}

```

I loop through each possible way a player can win to see whether either player has three in a row. The `if` statement checks to see whether the three squares in question all contain the same value and are not `EMPTY`. If so, it means that the row has either three Xs or three Os in it, and one side has won. The function then returns the piece in the first position of this winning row.

If neither player has won, I check for a tie game.

```

// since nobody has won, check for a tie (no empty squares left)
if (count(board.begin(), board.end(), EMPTY) == 0)
    return TIE;

```

If there are no empty squares on the board, then the game is a tie. I use the STL `count()` algorithm, which counts the number of times a given value appears in a group of container elements, to count the number of `EMPTY` elements in `board`. If the number is equal to 0, the function returns `TIE`.

Finally, if neither player has won and the game isn't a tie, then there is no winner yet. Thus, the function returns `NO_ONE`.

```

// since nobody has won and it isn't a tie, the game ain't over
return NO_ONE;
}

```

## The `isLegal()` Function

This function receives a board and a move. It returns `true` if the move is a legal one on the board or `false` if the move is not legal. A legal move is represented by the number of an empty square.

```
inline bool isLegal(int move, const vector<char>& board)
{
    return (board[move] == EMPTY);
}
```

Again, notice that the vector that represents the board is passed through a constant reference. This means that the vector is passed efficiently; it is not copied. It also means that the vector is safeguarded against any change.

You can see that I inlined `isLegal()`. Modern compilers are quite good at optimizing on their own; however, since this function is just one line, it's a good candidate for inlining.

## The `humanMove( )` Function

This next function receives a board and the human's piece. It returns the square number for where the player wants to move. The function asks the player for the square number to which he wants to move until the response is a legal move. Then the function returns the move.

```
int humanMove(const vector<char>& board, char human)
{
    int move = askNumber("Where will you move?", (board.size() - 1));
    while (!isLegal(move, board))
    {
        cout << "\nThat square is already occupied, foolish human.\n";
        move = askNumber("Where will you move?", (board.size() - 1));
    }
    cout << "Fine...\n";
    return move;
}
```

Again, notice that the vector that represents the board is passed through a constant reference. This means that the vector is passed efficiently; it is not copied. It also means that the vector is safeguarded against any change.

## The `computerMove( )` Function

This function receives the board and the computer's piece. It returns the computer's move. The first thing to notice is that I do not pass the board by reference.

```
int computerMove(vector<char> board, char computer)
```

Instead, I choose to pass by value, even though it's not as efficient as passing by reference. I pass by value because I need to work with and modify a copy of the board as I place pieces in empty squares to determine the best computer move. By working with a copy, I keep the original vector that represents the board safe.

Now on to the guts of the function. Okay, how do I program a bit of AI so the computer puts up a decent fight? Well, I came up with a basic three-step strategy for choosing a move.

1. If the computer can win on this move, make that move.
2. Otherwise, if the human can win on his next move, block him.
3. Otherwise, take the best remaining open square. The best square is the center.  
The next best squares are the corners, and then the rest of the squares.

The next section of the function implements Step 1.

```
{
    unsigned int move = 0;
    bool found = false;

    //if computer can win on next move, that's the move to make
    while (!found && move < board.size())
    {
        if (isLegal(move, board))
        {
            board[move] = computer;
            found = winner(board) == computer;
            board[move] = EMPTY;
        }

        if (!found)
        {
            ++move;
        }
    }
}
```

I begin to loop through all of the possible moves, 0–8. For each move, I test to see whether the move is legal. If it is, I put the computer's piece in the corresponding square and check to see whether the move gives the computer a win. Then I undo the move by making that square empty again. If the move didn't result in a win for the computer, I go on to the next empty square. However, if the move did give the computer a win, then the loop

ends—and I’ve found the move (found is true) that I want the computer to make (square number move) to win the game.

Next, I check to see if I need to go on to Step 2 of my AI strategy. If I haven’t found a move yet (found is false), then I check to see whether the human can win on his next move.

```
//otherwise, if human can win on next move, that's the move to make
if (!found)
{
    move = 0;
    char human = opponent(computer);
    while (!found && move < board.size())
    {
        if (isLegal(move, board))
        {
            board[move] = human;
            found = winner(board) == human;
            board[move] = EMPTY;
        }
        if (!found)
        {
            ++move;
        }
    }
}
```

I begin to loop through all of the possible moves, 0–8. For each move, I test to see whether the move is legal. If it is, I put the human’s piece in the corresponding square and check to see whether the move gives the human a win. Then I undo the move by making that square empty again. If the move didn’t result in a win for the human, I go on to the next empty square. However, if the move did give the human a win, then the loop ends—and I’ve found the move (found is true) that I want the computer to make (square number move) to block the human from winning on his next move.

Next, I check to see if I need to go on to Step 3 of my AI strategy. If I haven’t found a move yet (found is false) then I look through the list of best moves, in order of desirability, and take the first legal one.

```
//otherwise, moving to the best open square is the move to make
if (!found)
{
    move = 0;
    unsigned int i = 0;
    const int BEST_MOVES[] = {4, 0, 2, 6, 8, 1, 3, 5, 7};
    //pick best open square
    while (!found && i < board.size())
    {
        move = BEST_MOVES[i];
        if (isLegal(move, board))
        {
            found = true;
        }
        ++i;
    }
}
```

At this point in the function, I've found the move I want the computer to make—whether that's a move that gives the computer a win, blocks a winning move for the human, or is simply the best empty square available. So, I have the computer announce the move and return the corresponding square number.

```
cout << "I shall take square number " << move << endl;
return move;
}
```

## In the Real World

---

The Tic-Tac-Toe game considers only the next possible move. Programs that play serious games of strategy, such as chess, look far deeper into the consequences of individual moves and consider many levels of moves and countermoves. In fact, good computer chess programs can consider literally millions of board positions before making a move.

---

## The announceWinner( ) Function

This function receives the winner of the game, the computer's piece, and the human's piece. The function announces the winner or declares a tie.

```

void announceWinner(char winner, char computer, char human)
{
    if (winner == computer)
    {
        cout << winner << "'s won!\n";
        cout << "As I predicted, human, I am triumphant once more -- proof\n";
        cout << "that computers are superior to humans in all regards.\n";
    }
    else if (winner == human)
    {
        cout << winner << "'s won!\n";
        cout << "No, no! It cannot be! Somehow you tricked me, human.\n";
        cout << "But never again! I, the computer, so swear it!\n";
    }
    else
    {
        cout << "It's a tie.\n";
        cout << "You were most lucky, human, and somehow managed to tie me.\n";
        cout << "Celebrate...for this is the best you will ever achieve.\n";
    }
}

```

## SUMMARY

In this chapter, you should have learned the following concepts:

- A reference is an alias; it's another name for a variable.
- You create a reference using &—the referencing operator.
- A reference must be initialized when it's defined.
- A reference can't be changed to refer to a different variable.
- Whatever you do to a reference is done to the variable to which the reference refers.
- When you assign a reference to a variable, you create a new copy of the referenced value.
- When you pass a variable to a function by value, you pass a copy of the variable to the function.
- When you pass a variable to a function by reference, you pass access to the variable.
- Passing by reference can be more efficient than passing by value, especially when you are passing large objects.

- Passing a reference provides direct access to the argument variable passed to a function. As a result, the function can make changes to the argument variable.
- A constant reference can't be used to change the value to which it refers. You declare a constant reference by using the keyword `const`.
- You can't assign a constant reference or a constant value to a non-constant reference.
- Passing a constant reference to a function protects the argument variable from being changed by that function.
- Changing the value of an argument variable passed to a function can lead to confusion, so game programmers consider passing a constant reference before passing a non-constant reference.
- Returning a reference can be more efficient than returning a copy of a value, especially when you are returning large objects.
- You can return a constant reference to an object so the object can't be changed through the returned reference.
- A basic technique of game AI is to have the computer consider all of its legal moves and all of its opponent's legal replies before deciding which move to take next.

## QUESTIONS AND ANSWERS

**Q:** Different programmers put the reference operator (&) in different places when declaring a reference. Where should I put it?

**A:** Three basic styles exist with regard to using the referencing operator. Some programmers opt for `int& ref = var;`, while others opt for `int&ref = var;`. Still others opt for `int &ref = var;`. The computer is fine with all three. There are cases to be made for each style; however, the most important thing is to be consistent.

**Q:** Why can't I initialize a non-constant reference with a constant value?

**A:** Because a non-constant reference allows you to change the value to which it refers.

**Q:** If I initialize a constant reference with a non-constant variable, can I change the value of the variable?

**A:** Not through the constant reference, because when you declare a constant reference, you're saying that the reference can't be used to change the value to which it refers (even if that value can be changed by other means).

**Q:** How does passing a constant reference save overhead?

**A:** When you pass a large object to a function by value, your program makes a copy of the object. This can be an expensive operation depending on the size of the object. Passing a reference is like passing only access to the large object; it is an inexpensive operation.

**Q:** Can I make a reference to a reference?

**A:** Not exactly. You can assign one reference to another reference, but the new reference will simply refer to the value to which the original reference refers.

**Q:** What happens if I declare a reference without initializing it?

**A:** Your compiler should complain because it's illegal.

**Q:** Why should I avoid changing the value of a variable that I pass through a reference?

**A:** Because it could lead to confusion. It's impossible to tell from only a function call whether a variable is being passed to change its value.

**Q:** Does that mean I should always pass a constant reference?

**A:** No. You can pass a non-constant reference to a function, but to most game programmers, this signals that you intend to change the argument variable's value.

**Q:** If I don't change the argument variables passed to functions, how should I get new information back to the calling code?

**A:** Use return values.

**Q:** Can I pass a literal as a non-constant reference?

**A:** No. If you try to pass a literal as a non-constant reference, you'll generate a compile error.

**Q:** Is it impossible to pass a literal to a parameter that accepts a reference?

**A:** No, you can pass a literal as a constant reference.

**Q:** What happens when I return an object from a function?

**A:** Normally, your program creates a copy of the object and returns that. This can be an expensive operation, depending on the size of the object.

**Q:** Why return a reference?

**A:** It can be more efficient because returning a reference doesn't involve copying an object.

**Q:** How can I lose the efficiency of returning a reference?

**A:** By assigning the returned reference to a variable. When you assign a reference to a variable, the computer must make a copy of the object to which the reference refers.

**Q:** What's wrong with returning a reference to a local variable?

**A:** The local variable doesn't exist once the function ends, which means that you're returning a reference to a non-existent object, which is illegal.



## DISCUSSION QUESTIONS

1. What are the advantages and disadvantages of passing an argument by value?
2. What are the advantages and disadvantages of passing a reference?
3. What are the advantages and disadvantages of passing a constant reference?
4. What are the advantages and disadvantages of returning a reference?
5. Should game AI cheat in order to create a more worthy opponent?

## EXERCISES

1. Improve the Mad Lib game from Chapter 5, “Functions: Mad Lib,” by using references to make the program more efficient.
2. What’s wrong with the following program?

```
int main()
{
    int score;
    score = 1000;
    float& rScore = score;
    return 0;
}
```

3. What’s wrong with the following function?

```
int& plusThree(int number)
{
    int threeMore = number + 3;
    return threeMore;
}
```

*This page intentionally left blank*