Chapter 7

Pointers: Tic-Tac-Toe 2.0

Pointers are a powerful part of C++. In some ways, they behave like iterators from the STL. Often you can use them in place of references. But pointers offer functionality that no other part of the language can. In this chapter, you'll learn the basic mechanics of pointers and get an idea of what they're good for. Specifically, you'll learn to:

- Declare and initialize pointers
- Dereference pointers
- Use constants and pointers
- Pass and return pointers
- Work with pointers and arrays

Understanding Pointer Basics

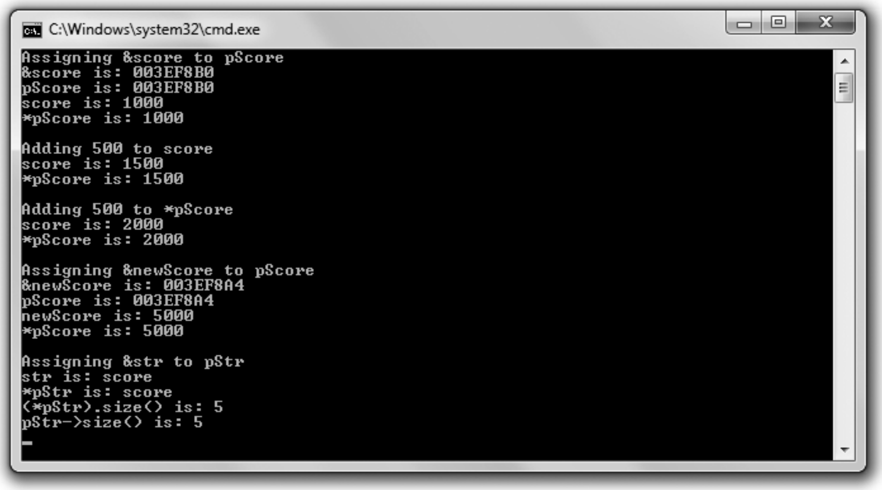
Pointers have a reputation for being difficult to understand. In reality, the essence of pointers is quite simple—a *pointer* is a variable that can contain a memory address. Pointers give you the ability to work directly and efficiently with computer memory. Like iterators from the STL, they're often used to access the contents of other variables. But before you can put pointers to good use in your game programs, you have to understand the basics of how they work.

Hint

Computer memory is a lot like a neighborhood, but instead of houses in which people store their stuff, you have memory locations where you can store data. Just like a neighborhood where houses sit side by side, labeled with addresses, chunks of computer memory sit side by side, labeled with addresses. In a neighborhood, you can use a slip of paper with a street address on it to get to a particular house (and to the stuff stored inside it). In a computer, you can use a pointer with a memory address in it to get to a particular memory location (and to the stuff stored inside it).

Introducing the Pointing Program

The Pointing program demonstrates the mechanics of pointers. The program creates a variable for a score and then creates a pointer to store the address of that variable. The program shows that you can change the value of a variable directly, and the pointer will reflect the change. It also shows that you can change the value of a variable through a pointer. It then demonstrates that you can change a pointer to point to another variable entirely. Finally, the program shows that pointers can work just as easily with objects. Figure 7.1 illustrates the results of the program.



```
Assigning &score to pScore
&score is: 003EF8B0
pScore is: 003EF8B0
score is: 1000
*pScore is: 1000

Adding 500 to score
score is: 1500
*pScore is: 1500

Adding 500 to *pScore
score is: 2000
*pScore is: 2000

Assigning &newScore to pScore
&newScore is: 003EF8A4
pScore is: 003EF8A4
newScore is: 5000
*pScore is: 5000

Assigning &str to pStr
str is: score
*pStr is: score
(*pStr).size() is: 5
pStr->size() is: 5
```

Figure 7.1
The pointer `pScore` first points to the variable `score` and then to the variable `newScore`, while the pointer `pStr` points to the variable `str`.
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 7 folder; the filename is `pointing.cpp`.

```
// Pointing
// Demonstrates using pointers

#include <iostream>
#include <string>

using namespace std;

int main()
{
    int* pAPointer;       //declare a pointer

    int* pScore = 0;      //declare and initialize a pointer

    int score = 1000;
    pScore = &score;      //assign pointer pScore address of variable score

    cout << "Assigning &score to pScore\n";
    cout << "&score is: " << &score << "\n";     //address of score variable
    cout << "pScore is: " << pScore << "\n";     //address stored in pointer
    cout << "score is: " << score << "\n";
    cout << "*pScore is: " << *pScore << "\n\n"; //value pointed to by pointer

    cout << "Adding 500 to score\n";
    score += 500;
    cout << "score is: " << score << "\n";
    cout << "*pScore is: " << *pScore << "\n\n";

    cout << "Adding 500 to *pScore\n";
    *pScore += 500;
    cout << "score is: " << score << "\n";
    cout << "*pScore is: " << *pScore << "\n\n";

    cout << "Assigning &newScore to pScore\n";
    int newScore = 5000;
    pScore = &newScore;
    cout << "&newScore is: " << &newScore << "\n";
    cout << "pScore is: " << pScore << "\n";
    cout << "newScore is: " << newScore << "\n";
    cout << "*pScore is: " << *pScore << "\n\n";

    cout << "Assigning &str to pStr\n";
    string str = "score";
    string* pStr = &str;     //pointer to string object
    cout << "str is: " << str << "\n";
    cout << "*pStr is: " << *pStr << "\n";
    cout << "(*pStr).size() is: " << (*pStr).size() << "\n";
    cout << "pStr->size() is: " << pStr->size() << "\n";

    return 0;
}
```

Declaring Pointers

With the first statement in `main()` I declare a pointer named `pAPointer`.

```
int* pAPointer;     //declare a pointer
```

Because pointers work in such a unique way, programmers often prefix pointer variable names with the letter “p” to remind them that the variable is indeed a pointer.

Just like an iterator, a pointer is declared to point to a specific type of value. `pAPointer` is a pointer to `int`, which means that it can only point to an `int` value. `pAPointer` can’t point to a `float` or a `char`, for example. Another way to say this is that `pAPointer` can only store the address of an `int`.

To declare a pointer of your own, begin with the type of object to which the pointer will point, followed by an asterisk, followed by the pointer name. When you declare a pointer, you can put whitespace on either side of the asterisk. So `int* pAPointer;`, `int *pAPointer;`, and `int * pAPointer;` all declare a pointer named `pAPointer`.

Trap

When you declare a pointer, the asterisk only applies to the single variable name that immediately follows it. So the following statement declares `pScore` as a pointer to `int` and score as an `int`.

```
int* pScore, score;
```

`score` is not a pointer! It’s a variable of type `int`. One way to make this clearer is to play with the whitespace and rewrite the statement as:

```
int *pScore, score;
```

However, the clearest way to declare a pointer is to declare it in its own statement, as in the following lines.

```
int* pScore;
int score;
```

Initializing Pointers

As with other variables, you can initialize a pointer in the same statement you declare it. That's what I do next with the following line, which assigns `0` to `pScore`.

```
int* pScore = 0;  //declare and initialize a pointer
```

Assigning `0` to a pointer has special meaning. Loosely translated, it means, "Point to nothing." Programmers call a pointer with the value of zero a *null pointer*. You should always initialize a pointer with some value when you declare it, even if that value is zero.

Hint

Many programmers assign NULL to a pointer instead of 0 to make the pointer a null pointer. NULL is a constant defined in multiple library files, including `iostream`.

Assigning Addresses to Pointers

Because pointers store addresses of objects, you need a way to get addresses into the pointers. One way to do that is to get the memory address of an existing variable and assign it to a pointer. That's what I do in the following line, which gets the address of the variable `score` and assigns it to `pScore`.

```
pScore = &score;    //assign pointer address of variable score
```

I get the address of `score` by preceding the variable name with `&`, the *address of* operator. (Yes, you've seen the `&` symbol before, when it was used as the reference operator. However, in this context, the `&` symbol gets the address of an object.)

As a result of the preceding line of code, `pScore` contains the address of `score`. It's as if `pScore` knows exactly where `score` is located in the computer's memory. This means you can use `pScore` to get to `score` and manipulate the value stored in `score`. Figure 7.2 serves as a visual illustration of the relationship between `pScore` and `score`.

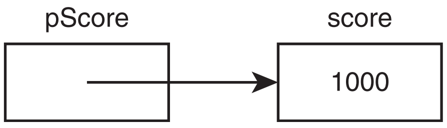


Figure 7.2
The pointer `pScore` points to `score`, which stores the value `1000`.

To prove that pScore contains the address of score, I display the address of the variable and the value of the pointer with the following lines.

```
cout << "&score is: " << &score << "\n";      //address of score variable
cout << "pScore is: " << pScore << "\n";      //address stored in pointer
```

As you can see from Figure 7.1, pScore contains 003EF8B0, which is the address of score. (The specific addresses displayed by the Pointing program might be different on your system. The important thing is that the values for pScore and &score are the same.)

Dereferencing Pointers

Just as you dereference an iterator to access the object to which it refers, you dereference a pointer to access the object to which it points. You accomplish the dereferencing the same way—with *, the dereference operator. I put the dereference operator to work with the following line, which displays 1000 because *pScore accesses the value stored in score.

```
cout << "*pScore is: " << *pScore << "\n\n"; //value pointed to by pointer
```

Remember, *pScore means, "the object to which pScore points."

Trap

Don't dereference a null pointer because it could lead to disastrous results.

Next, I add 500 to score with the following line.

```
score += 500;
```

When I send score to cout, 1500 is displayed, as you'd expect. When I send *pScore to cout, the contents of score are again sent to cout, and 1500 is displayed once more.

Next, I add 500 to the value to which pScore points with the following line.

```
*pScore += 500;
```

Because pScore points to score, the preceding line of code adds 500 to score. Therefore, when I next send score to cout, 2000 is displayed. Then, when I send *pScore to cout… you guessed it, 2000 is displayed again.

Trap

Don't change the value of a pointer when you want to change the value of the object to which the pointer points. For example, if I want to add 500 to the int that pScore points to, then the following line would be a big mistake.

```
pScore += 500;
```

The preceding code adds 500 to the address stored in pScore, not to the value to which pScore originally pointed. As a result, pScore now points to some address that might contain anything. Dereferencing a pointer like this can lead to disastrous results.

Reassigning Pointers

Unlike references, pointers can point to different objects at different times during the life of a program. Reassigning a pointer works like reassigning any other variable. Next, I reassign pScore with the following line.

```
pScore = &newScore;
```

As the result, pScore now points to newScore. To prove this, I display the address of newScore by sending &newScore to cout, followed by the address stored in pScore. Both statements display the same address. Then I send newScore and *pScore to cout. Both display 5000 because they both access the same chunk of memory that stores this value.

Trap

Don't change the value to which a pointer points when you want to change the pointer itself. For example, if I want to change pScore to point to newScore, then the following line would be a big mistake.

```
*pScore = newScore;
```

This code simply changes the value to which pScore currently points; it doesn't change pScore itself. If newScore is equal to 5000, then the previous code is equivalent to *pScore = 5000; and pScore still points to the same variable it pointed to before the assignment.

Using Pointers to Objects

So far, the Pointing program has worked only with values of a built-in type, int. But you can use pointers with objects just as easily. I demonstrate this next with the following lines, which create str, a string object equal to "score", and pStr, a pointer that points to that object.

```
string str = "score";
string* pStr = &str;    //pointer to string object
```

`pStr` is a pointer to `string`, meaning that it can point to any `string` object. Another way to say this is to say that `pStr` can store the address of any `string` object.

You can access an object through a pointer using the dereference operator. That's what I do next with the following line.

```
cout << "*pStr is: " << *pStr << "\n";
```

By using the dereference operator with `*pStr`, I send the object to which `pStr` points (`str`) to `cout`. As a result, the text `score` is displayed.

You can call the member functions of an object through a pointer the same way you can call the member functions of an object through an iterator. One way to do this is by using the dereference operator and the member access operator, which is what I do next with the following line.

```
cout << "(*pStr).size() is: " << (*pStr).size() << "\n";
```

The code `(*pStr).size()` says, "Take the result of dereferencing `pStr` and call that object's `size()` member function." Because `pStr` refers to the `string` object equal to `"score"`, the code returns `5`.

Hint

Whenever you dereference a pointer to access a data member or member function, surround the dereferenced pointer with a pair of parentheses. This ensures that the dot operator will be applied to the object to which the pointer points.

Just as with iterators, you can use the `->` operator with pointers for a more readable way to access object members. That's what I demonstrate next with the following line.

```
cout << "pStr->size() is: " << pStr->size() << "\n";
```

The preceding statement again displays the number of characters in the `string` object equal to `"score"`; however, I'm able to substitute `pStr->size()` for `(*pStr).size()` this time, making the code more readable.

Understanding Pointers and Constants

There are still some pointer mechanics you need to understand before you can start to use pointers effectively in your game programs. You can use the keyword `const` to restrict the way a pointer works. These restrictions can act as safeguards and can make your programming intentions clearer. Since pointers are quite versatile, restricting how a pointer can be used is in line with the programming mantra of asking only for what you need.

Using a Constant Pointer

As you've seen, pointers can point to different objects at different times in a program. However, by using the `const` keyword when you declare and initialize a pointer, you can restrict the pointer so it can only point to the object to which it was initialized to point. A pointer like this is called a *constant pointer*. Another way to say this is to say that the address stored in a constant pointer can never change—it's constant. Here's an example of creating a constant pointer:

```
int score = 100;
int* const pScore = &score;   //a constant pointer
```

The preceding code creates a constant pointer, `pScore`, which points to `score`. You create a constant pointer by putting `const` right before the name of the pointer when you declare it.

Like all constants, you must initialize a constant pointer when you first declare it. The following line is illegal and will produce a big, fat compile error.

```
int* const pScore;   //illegal -- you must initialize a constant pointer
```

Because `pScore` is a constant pointer, it can't ever point to any other memory location. The following code is also quite illegal.

```
pScore = &anotherScore;   //illegal -- pScore can't point to a different object
```

Although you can't change `pScore` itself, you can use `pScore` to change the value to which it points. The following line is completely legal.

```
*pScore = 500;
```

Confused? Don't be. It's perfectly fine to use a constant pointer to change the value to which it points. Remember, the restriction on a constant pointer is that its value—the address that the pointer stores—can't change.

The way a constant pointer works should remind you of something—a reference. Like a reference, a constant pointer can refer only to the object to which it was initialized to refer.

Hint

Although you can use a constant pointer instead of a reference in your programs, you should stick with references when possible. References have a cleaner syntax than pointers and can make your code easier to read.

Using a Pointer to a Constant

As you've seen, you can use pointers to change the values to which they point. However, by using the `const` keyword when you declare a pointer, you can restrict a pointer so it can't be used to change the value to which it points. A pointer like this is called a *pointer to a constant*. Here's an example of declaring such a pointer:

```
const int* pNumber;  //a pointer to a constant
```

The preceding code declares a pointer to a constant, `pNumber`. You declare a pointer to a constant by putting `const` right before the type of value to which the pointer will point.

You assign an address to a pointer to a constant as you did before.

```
int lives = 3;
pNumber = &lives;
```

However, you can't use the pointer to change the value to which it points. The following line is illegal.

```
*pNumber -= 1;  //illegal -- can't use pointer to a constant to change value
                //that pointer points to
```

Although you can't use a pointer to a constant to change the value to which it points, the pointer itself can change. This means that a pointer to a constant can point to different objects in a program. The following code is perfectly legal.

```
const int MAX_LIVES = 5;
pNumber = &MAX_LIVES;  //pointer itself can change
```

Using a Constant Pointer to a Constant

A *constant pointer to a constant* combines the restrictions of a constant pointer and a pointer to a constant. This means that a constant pointer to a constant can only point to the object to which it was initialized to point. In addition, it can't be used to change the value of the object to which it points. Here's the declaration and initialization of such a pointer:

```
const int* const pBONUS = &BONUS;  //a constant pointer to a constant
```

The preceding code creates a constant pointer to a constant named `pBONUS` that points to the constant `BONUS`.

Hint

Like a pointer to a constant, a constant pointer to a constant can point to either a non-constant or a constant value.

You can't reassign a constant pointer to a constant. The following line is not legal.

```
pBONUS = &MAX_LIVES;  //illegal -- pBONUS can't point to another object
```

You can't use a constant pointer to a constant to change the value to which it points. This means that the following line is illegal.

```
*pBONUS = MAX_LIVES;  //illegal -- can't change value through pointer
```

In many ways, a constant pointer to a constant acts like a constant reference, which can only refer to the value to which it was initialized to refer and which can't be used to change that value.

Hint

Although you can use a constant pointer to a constant instead of a constant reference in your programs, you should stick with constant references when possible. References have a cleaner syntax than pointers and can make your code easier to read.

Summarizing Constants and Pointers

I've presented a lot of information on constants and pointers, so I want to provide a summary to help crystallize the new concepts. Here are three examples of the different ways in which you can use the keyword `const` when declaring pointers:

- `int* const p = &i;`
- `const int* p;`
- `const int* const p = &I;`

The first example declares and initializes a constant pointer. A constant pointer can only point to the object to which it was initialized to point. The value—the memory address—stored in the pointer itself is constant and can't change. A constant pointer can only point to a non-constant value; it can't point to a constant.

The second example declares a pointer to a constant. A pointer to a constant can't be used to change the value to which it points. A pointer to a constant can point to different objects during the life of a program. A pointer to a constant can point to a constant or non-constant value.

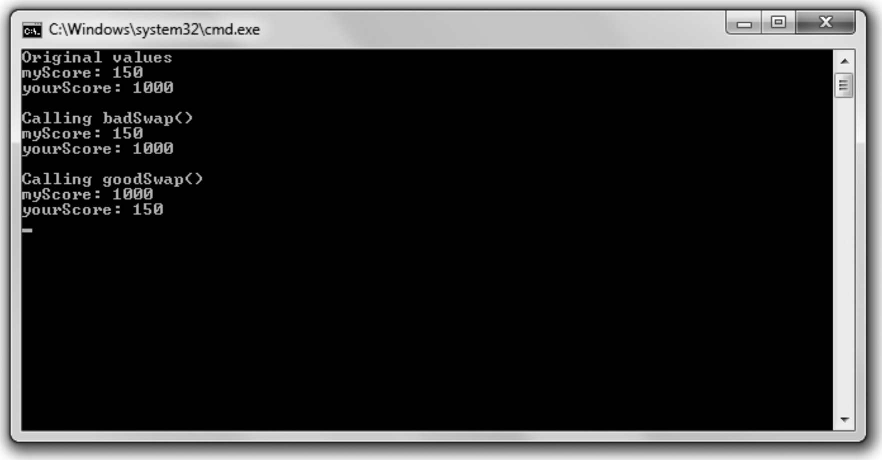
The third example declares a constant pointer to a constant. A constant pointer to a constant can only point to the value to which it was initialized to point. In addition, it can't be used to change the value to which it points. A constant pointer to a constant can be initialized to point to a constant or a non-constant value.

Passing Pointers

Even though references are the preferred way to pass arguments because of their cleaner syntax, you still might need to pass objects through pointers. For example, suppose you're using a graphics engine that returns a pointer to a 3D object. If you want another function to use this object, you'll probably want to pass the pointer to the object for efficiency. Therefore, it's important to know how to pass pointers as well as references.

Introducing the Swap Pointer Version Program

The Swap Pointer Version program works just like the Swap program from Chapter 6, "References: Tic-Tac-Toe," except that the Swap Pointer Version program uses pointers instead of references. The Swap Pointer Version 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. Because only copies of the score values are sent to the function, the original variables are unaltered. Next, the program calls another swap function. This time, using constant pointers, the original variables' values are successfully exchanged (giving me the great big score and leaving you with the small one). Figure 7.3 shows the program in action.



```
Original values
myScore: 150
yourScore: 1000

Calling badSwap()
myScore: 150
yourScore: 1000

Calling goodSwap()
myScore: 1000
yourScore: 150
```

Figure 7.3
Passing pointers allows a function to alter variables outside of the function's scope.
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 7 folder; the filename is `swap_pointer_ver.cpp`.

```
// Swap Pointer
// Demonstrates passing constant pointers to alter argument variables

#include <iostream>

using namespace std;

void badSwap(int x, int y);
void goodSwap(int* const pX, int* const pY);

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* const pX, int* const pY)
{
    //store value pointed to by pX in temp
    int temp = *pX;
    //store value pointed to by pY in address pointed to by pX
    *pX = *pY;
    //store value originally pointed to by pX in address pointed to by pY
    *pY = temp;
}
```

Passing by Value

After I declare and initialize `myScore` and `yourScore`, I send them to `cout`. As you'd expect, `150` and `1000` are displayed. Next I call `badSwap()`, which passes both arguments by value. This means that when I call the function 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 `badSwap()` will have any effect on `myScore` and `yourScore`.

When `badSwap()` executes, `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. Control then returns to `main()`, in which `myScore` and `yourScore` haven't changed. When I then send `myScore` and `yourScore` to `cout`, `150` and `1000` are displayed again. Sadly, I still have the tiny score and you still have the large one.

Passing a Constant Pointer

You've seen that it's possible to give a function access to variables by passing references. It's also possible to accomplish this using pointers. When you pass a pointer, you pass only the address of an object. This can be quite efficient, especially if you're working with objects that occupy large chunks of memory. Passing a pointer is like e-mailing a friend the URL of a website instead of trying to send him the entire site.

Before you can pass a pointer to a function, you need to specify function parameters as pointers. That's what I do in the `goodSwap()` header.

```
void goodSwap(int* const pX, int* const pY)
```

This means that `pX` and `pY` are constant pointers and will each accept a memory address. I made the parameters constant pointers because, although I plan to change the values they point to, I don't plan to change the pointers themselves. Remember, this is just how references work. You can change the value to which a reference refers, but not the reference itself.

In `main()`, I pass the addresses of `myScore` and `yourScore` when I call `goodSwap()` with the following line.

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

Notice that I send the addresses of the variables to `goodSwap()` by using the *address of* operator. When you pass an object to a pointer, you need to send the address of the object.

In `goodSwap()`, `pX` stores the address of `myScore` and `pY` stores the address of `yourScore`. Anything done to `*pX` will be done to `myScore`; anything done to `*pY` will be done to `yourScore`.

The first line of `goodSwap()` takes the value that `pX` points to and assigns it to `temp`.

```
int temp = *pX;
```

Because `pX` points to `myScore`, `temp` becomes `150`.

The next line assigns the value pointed to by `pY` to the object to which `pX` points.

```
*pX = *pY;
```

This statement copies the value stored in `yourScore`, `1000`, and assigns it to the memory location of `myScore`. As a result, `myScore` becomes `1000`.

The last statement in the function stores the value of `temp`, `150`, in the address pointed to by `pY`.

```
*pY = temp;
```

Because `pY` points to `yourScore`, `yourScore` becomes `150`.

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. Success at last!

Hint

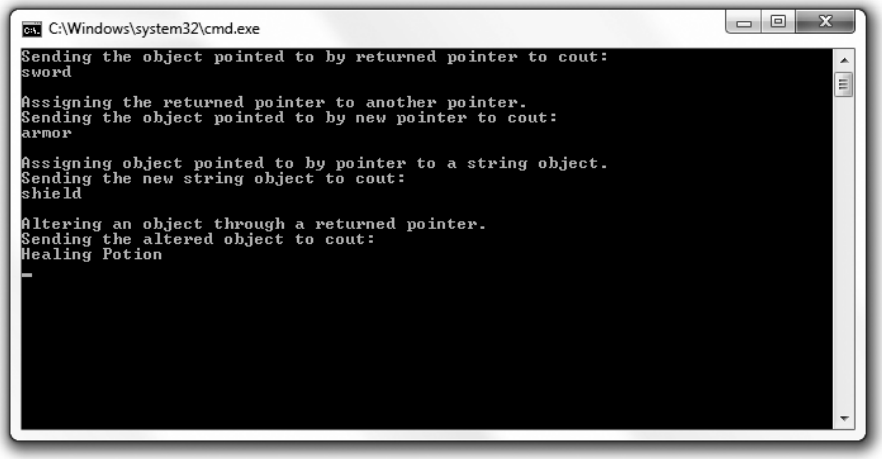
You can also pass a constant pointer to a constant. This works much like passing a constant reference, which is done to efficiently pass an object that you don't need to change. I've adapted the Inventory Displayer program from Chapter 6, which demonstrates passing constant references, to pass a constant pointer to a constant. You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 7 folder; the filename is `inventory_displayer_pointer_ver.cpp`.

Returning Pointers

Before references, the only option game programmers had for returning objects efficiently from functions was using pointers. And even though using references provides a cleaner syntax than using pointers, you might still need to return objects through pointers.

Introducing the Inventory Pointer Program

The Inventory Pointer program demonstrates returning pointers. Through returned pointers, the program displays and even alters the values of a vector that holds a hero's inventory. Figure 7.4 shows the results of the program.



```
Sending the object pointed to by returned pointer to cout:
sword

Assigning the returned pointer to another pointer.
Sending the object pointed to by new pointer to cout:
armor

Assigning object pointed to by pointer to a string object.
Sending the new string object to cout:
shield

Altering an object through a returned pointer.
Sending the altered object to cout:
Healing Potion
_
```

Figure 7.4
A function returns a pointer (not a `string` object) to each item in the hero's inventory.
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 7 folder; the filename is `inventory_pointer.cpp`.

```
// Inventory Pointer
// Demonstrates returning a pointer

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

```
using namespace std;

//returns a pointer to a string element
string* ptrToElement(vector<string>* const pVec, int i);

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

    //displays string object that the returned pointer points to
    cout << "Sending the object pointed to by returned pointer to cout:\n";
    cout << *(ptrToElement(&inventory, 0)) << "\n\n";
    //assigns one pointer to another -- inexpensive assignment
    cout << "Assigning the returned pointer to another pointer.\n";
    string* pStr = ptrToElement(&inventory, 1);
    cout << "Sending the object pointed to by new pointer to cout:\n";
    cout << *pStr << "\n\n";

    //copies a string object -- expensive assignment
    cout << "Assigning object pointed to by pointer to a string object.\n";
    string str = *(ptrToElement(&inventory, 2));
    cout << "Sending the new string object to cout:\n";
    cout << str << "\n\n";

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

    return 0;
}

string* ptrToElement(vector<string>* const pVec, int i)
{
    //returns address of the string in position i of vector that pVec points to
    return &((*pVec)[i]);
}
```

Returning a Pointer

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

```
string* ptrToElement(vector<string>* const pVec, int i)
```

By starting the header with `string*`, I'm saying that the function will return a pointer to a `string` object (and not a `string` object itself). To specify that a function returns a pointer to an object of a particular type, put an asterisk after the type name of the return type.

The body of the function `ptrToElement()` contains only one statement, which returns a pointer to the element at position `i` in the vector pointed to by `pVec`.

```
    return &((*pVec)[i]);
```

The `return` statement might look a little cryptic, so I'll step through it. Whenever you come upon a complex expression, evaluate it like the computer does—by starting with the innermost part. I'll start with `(*pVec)[i]`, which means the element in position `i` of the vector pointed to by `pVec`. By applying the *address of* operator (`&`) to the expression, it becomes the address of the element in position `i` of the vector pointed to by `pVec`.

Trap

Although returning a pointer can be an efficient way to send information back to a calling function, you have to be careful not to return a pointer that points to an out-of-scope object. For example, the following function returns a pointer that, if used, could crash the program.

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

The program could crash because `badPointer()` returns a pointer to a string that no longer exists after the function ends. A pointer to a non-existent object is called a *dangling pointer*. Attempting to dereference a dangling pointer can lead to disastrous results. One way to avoid dangling pointers is to never return a pointer to a local variable.

Using a Returned Pointer to Display a Value

After I create `inventory`, a vector of items, I display a value with a returned pointer.

```
    cout << *(ptrToElement(&inventory, 0)) << "\n\n";
```

The preceding code calls `ptrToElement()`, which returns a pointer to `inventory[0]`. (Remember, `ptrToElement()` doesn't return a copy of one of the elements of `inventory`; it returns a pointer to one of them.) The line then sends the `string` object pointed to by the pointer to `cout`. As a result, `sword` is displayed.

Assigning a Returned Pointer to a Pointer

Next, I assign a returned pointer to another pointer with the following line.

```
string* pStr = ptrToElement(&inventory, 1);
```

The call to `prtToElement()` returns a pointer to `inventory[1]`. The statement assigns that pointer to `pStr`. This is an efficient assignment because assigning a pointer to a pointer does not involve copying the `string` object.

To help you understand the results of this line of code, look at Figure 7.5, which shows a representation of `pStr` after the assignment. (Note that the figure is abstract because the vector `inventory` doesn't contain the string literals `"sword"`, `"armor"`, and `"shield"`; instead, it contains `string` objects.)

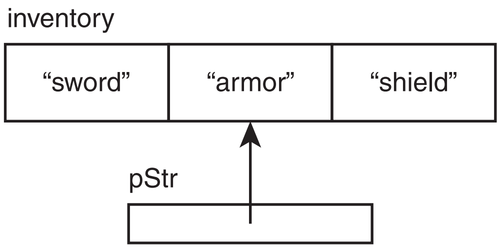


Figure 7.5
`pStr` points to the element at position 1 of `inventory`.

Next, I send `*pStr` to `cout`, and `armor` is displayed.

Assigning to a Variable the Value Pointed to by a Returned Pointer

Next, I assign the value pointed to by a returned pointer to a variable.

```
string str = *(ptrToElement(&inventory, 2));
```

The call to `ptrToElement()` returns a pointer to `inventory[2]`. However, the preceding statement doesn't assign this pointer to `str`—it can't because `str` is a `string` object. Instead, the computer quietly makes a copy of the `string` object to which the pointer points and assigns that object to `str`. To help drive this point home, check out Figure 7.6, which provides an abstract representation of the results of this assignment.

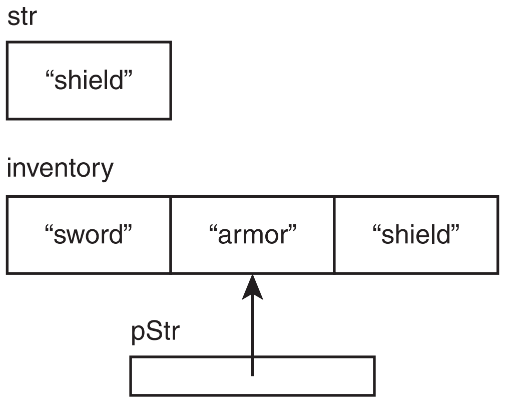


Figure 7.6
`str` is a new `string` object, totally independent from `inventory`.

An assignment like this one, where an object is copied, is more expensive than the assignment of one pointer to another. Sometimes the cost of copying an object is perfectly acceptable, but you should be aware of the extra overhead associated with this kind of assignment and avoid it when necessary.

Altering an Object through a Returned Pointer

You can also alter the object to which a returned pointer points. This means that I can change the hero's inventory through `pStr`.

```
*pStr = "Healing Potion";
```

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

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

For an abstract representation, check out Figure 7.7, which shows the status of the variables after the assignment.

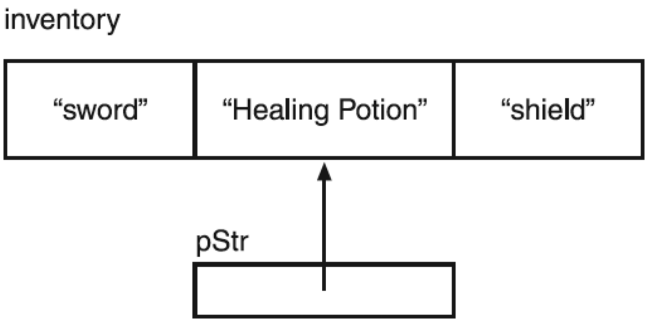


Figure 7.7
`inventory[1]` is changed through the returned pointer stored in `pStr`.

Hint

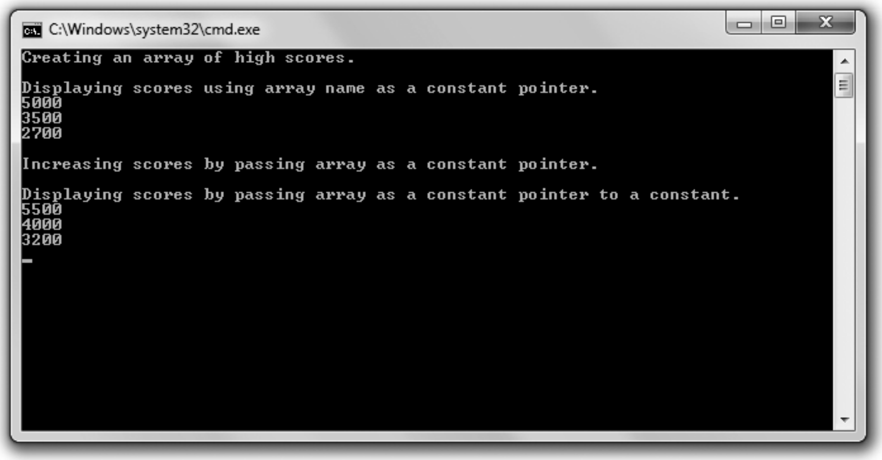
If you want to protect an object pointed to by a returned pointer, make sure to restrict the pointer. Return either a pointer to a constant or a constant pointer to a constant.

Understanding the Relationship between Pointers and Arrays

Pointers have an intimate relationship with arrays. In fact, an array name is really a constant pointer to the first element of the array. Because the elements of an array are stored in a contiguous block of memory, you can use the array name as a pointer for random access to elements. This relationship also has important implications for how you can pass and return arrays, as you'll soon see.

Introducing the Array Passer Program

The Array Passer program creates an array of high scores and then displays them, using the array name as a constant pointer. Next, the program passes the array name as a constant pointer to a function that increases the scores. Finally, the program passes the array name to a function as a constant pointer to a constant to display the new high scores. Figure 7.8 shows the results of the program.


C:\Windows\system32\cmd.exe

```
Creating an array of high scores.

Displaying scores using array name as a constant pointer.
5000
3500
2700

Increasing scores by passing array as a constant pointer.

Displaying scores by passing array as a constant pointer to a constant.
5500
4000
3200
_
```

Figure 7.8
Using an array name as a pointer, the high scores are displayed, altered, and passed to functions.
Used with permission from Microsoft.

You can download the code for this program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 7 folder; the filename is `array_passer.cpp`.

```
//Array Passer
//Demonstrates relationship between pointers and arrays

#include <iostream>

using namespace std;

void increase(int* const array, const int NUM_ELEMENTS);
void display(const int* const array, const int NUM_ELEMENTS);

int main()
{
    cout << "Creating an array of high scores.\n\n";
    const int NUM_SCORES = 3;
    int highScores[NUM_SCORES] = {5000, 3500, 2700};

    cout << "Displaying scores using array name as a constant pointer.\n";
    cout << *highScores << endl;
    cout << *(highScores + 1) << endl;
    cout << *(highScores + 2) << "\n\n";

    cout << "Increasing scores by passing array as a constant pointer.\n\n";
    increase(highScores, NUM_SCORES);
    cout << "Displaying scores by passing array as a constant pointer to a constant.\n";
    display(highScores, NUM_SCORES);

    return 0;
}

void increase(int* const array, const int NUM_ELEMENTS)
{
    for (int i = 0; i < NUM_ELEMENTS; ++i)
    {
        array[i] += 500;
    }
}

void display(const int* const array, const int NUM_ELEMENTS)
{
    for (int i = 0; i < NUM_ELEMENTS; ++i)
    {
        cout << array[i] << endl;
    }
}
```

Using an Array Name as a Constant Pointer

Because an array name is a constant pointer to the first element of the array, you can dereference the name to get at the first element. That's what I do after I create an array of high scores, called `highScores`.

```
    cout << *highScores << endl;
```

I dereference `highScores` to access the first element in the array and send it to `cout`. As a result, `5000` is displayed.

You can randomly access array elements using an array name as a pointer through simple addition. All you have to do is add the position number of the element you want to access to the pointer before you dereference it. This is simpler than it sounds. For example, I next access the score at position `1` in `highScores` with the following line, which displays `3500`.

```
    cout << *(highScores + 1) << endl;
```

In the preceding code, `*(highScores + 1)` is equivalent to `highScores[1]`. Both return the element in position `1` of `highScores`.

Next, I access the score at position 2 in highScores with the following line, which displays 2700.

```
    cout << *(highScores + 2) << endl;
```

In the preceding code, *(highScores + 2) is equivalent to highScores[2]. Both return the element in position 2 of highScores. In general, you can write arrayName[i] as *(arrayName + i), where arrayName is the name of an array.

Passing and Returning Arrays

Because an array name is a constant pointer, you can use it to efficiently pass an array to a function. That's what I do next with the following line, which passes to increase() a constant pointer to the first element of the array and the number of elements in the array.

```
    increase(highScores, NUM_SCORES);
```

Hint

When you pass an array to a function, it's usually a good idea to also pass the number of elements in the array so the function can use this to avoid attempting to access an element that doesn't exist.

As you can see from the function header of increase(), the array name is accepted as a constant pointer.

```
void increase(int* const array, const int NUM_ELEMENTS)
```

The function body adds 500 to each score.

```
    for (int i = 0; i < NUM_ELEMENTS; ++i)
    {
        array[i] += 500;
    }
```

I treat array just like any array and use the subscripting operator to access each of its elements. Alternatively, I could have treated array as a pointer and substituted *(array + i) += 500 for the expression array[i] += 500, but I opted for the more readable version.

After increase() ends, control returns to main(). To prove that increase() did in fact increase the high scores, I call a function to show the scores.

```
    display(highScores, NUM_SCORES);
```

The function `display()` also accepts `highScore` as a pointer. However, as you can see from the function's header, the function accepts it as a constant pointer to a constant.

```
void display(const int* const array, const int NUM_ELEMENTS)
```

By passing the array in this way, I keep it safe from changes. Because all I want to do is display each element, it's the perfect way to go.

Finally, the body of `display()` runs and all of the scores are listed, showing that they've each increased by 500.

Hint

You can pass a C-style string to a function, just like any other array. In addition, you can pass a string literal to a function as a constant pointer to a constant.

Because an array name is a pointer, you can return an array using the array name, just as you would any other pointer to an object.

Introducing the Tic-Tac-Toe 2.0 Game

The project for this chapter is a modified version of the project from Chapter 6, the Tic-Tac-Toe game. From the player's perspective, the Tic-Tac-Toe 2.0 game looks exactly the same as the original because the changes are under the hood—I've replaced all of the references with pointers. This means that objects such as the Tic-Tac-Toe board are passed as constant pointers instead of as references. This has other implications, including the fact that the address of a Tic-Tac-Toe board must be passed instead of the board itself.

You can download the code for the new version of the program from the Cengage Learning website (www.cengageptr.com/downloads). The program is in the Chapter 7 folder; the filename is `tic-tac-toe2.cpp`. I won't go over the code because most of it remains the same. But even though the number of changes isn't great, the changes are significant. This is a good program to study because, although you should use references whenever you can, you should be equally comfortable with pointers.

Summary

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

- Computer memory is organized in an ordered way, where each chunk of memory has its own unique address.
- A pointer is a variable that contains a memory address.
- In many ways, pointers act like iterators from the STL. For example, just as with iterators, you use pointers to indirectly access an object.
- To declare a pointer, you list a type, followed by an asterisk, followed by a name.
- Programmers often prefix pointer variable names with the letter "p" to remind them that the variable is indeed a pointer.
- Just like an iterator, a pointer is declared to refer to a value of a specific type.
- It's good programming practice to initialize a pointer when you declare it.
- If you assign `0` to a pointer, the pointer is called a null pointer.
- To get the address of a variable, put the *address of* operator (`&`) before the variable name.
- When a pointer contains the address of an object, it's said to point to the object.
- Unlike references, you can reassign pointers. That is, a pointer can point to different objects at different times during the life of a program.
- Just as with iterators, you dereference a pointer to access the object it points to with `*`, the dereference operator.
- Just as with iterators, you can use the `->` operator with pointers for a more readable way to access object data members and member functions.
- A constant pointer can only point to the object to which it was initialized to point. You declare a constant pointer by putting the keyword `const` right before the pointer name, as in `int* const p = &i;`.
- You can't use a pointer to a constant to change the value to which it points. You declare a pointer to a constant by putting the keyword `const` before the type name, as in `const int* p;`.
- A constant pointer to a constant can only point to the value to which it was initialized to point, and it can't be used to change that value. You declare a constant pointer to a constant by putting the keyword `const` before the type name and right before the pointer name, as in `const int* const p = &I;`.

- You can pass pointers for efficiency or to provide direct access to an object.
- If you want to pass a pointer for efficiency, you should pass a pointer to a constant or a constant pointer to a constant so the object you're passing access to can't be changed through the pointer.
- A dangling pointer is a pointer to an invalid memory address. Dangling pointers are often caused by deleting an object to which a pointer pointed. Dereferencing such a pointer can lead to disastrous results.
- You can return a pointer from a function, but be careful not to return a dangling pointer.

Questions and Answers

Q: How is a pointer different from the variable to which it points?
A: A pointer stores a memory address. If a pointer points to a variable, it stores the address of that variable.

Q: What good is it to store the address of a variable that already exists?
A: One big advantage of storing the address of an existing variable is that you can pass a pointer to the variable for efficiency instead of passing the variable by value.

Q: Does a pointer always have to point to an existing variable?
A: No. You can create a pointer that points to an unnamed chunk of computer memory as you need it. You'll learn more about allocating memory in this dynamic fashion in Chapter 9, "Advanced Classes and Dynamic Memory: Game Lobby."

Q: Why should I pass variables using references instead of pointers whenever possible?
A: Because of the sweet, syntactic sugar that references provide. Passing a reference or a pointer is an efficient way to provide access to objects, but pointers require extra syntax (like the dereference operator) to access the object itself.

Q: Why should I initialize a pointer when I declare it or soon thereafter?
A: Because dereferencing an uninitialized pointer can lead to disastrous results, including a program crash.

Q: What's a dangling pointer?
A: A pointer that points to an invalid memory location, where any data could exist.

Q: What's so dangerous about a dangling pointer?
A: Like using an uninitialized pointer, using a dangling pointer can lead to disastrous results, including a program crash.

Q: Why should I initialize a pointer to `0`?
A: By initializing a pointer to `0`, you create a null pointer, which is understood as a pointer to nothing.

Q: So then it's safe to dereference a null pointer, right?
A: No! Although it's good programming practice to assign `0` to a pointer that doesn't point to an object, dereferencing a null pointer is as dangerous as dereferencing a dangling pointer.

Q: What will happen if I dereference a null pointer?
A: Just like dereferencing a dangling pointer or an uninitialized pointer, the results are unpredictable. Most likely, you'll crash your program.

Q: What good are null pointers?
A: They're often returned by functions as a sign of failure. For example, if a function is supposed to return a pointer to an object that represents the graphics screen, but that function couldn't initialize the screen, it might return a null pointer.

Q: How does using the keyword `const` when declaring a pointer affect the pointer?
A: It depends on how you use it. Generally, you use `const` when you are declaring a pointer to restrict what the pointer can do.

Q: What kinds of restrictions can I impose on a pointer by declaring it with `const`?
A: You can restrict a pointer so it can only point to the object it was initialized to point to, or you can restrict a pointer so it can't change the value of the object it points to, or both.

Q: Why would I want to restrict what a pointer can do?
A: For safety. For example, you might be working with an object that you know you don't want to change.

Q: To what type of pointers can I assign a constant value?
A: A pointer to a constant or a constant pointer to a constant.

Q: How can I safely return a pointer from a function?
A: One way is by returning a pointer to an object that you received from the calling function. This way, you're returning a pointer to an object that exists back in the calling code. (In Chapter 9, you'll discover another important way when you learn about dynamic memory.)

Discussion Questions

1. What are the advantages and disadvantages of passing a pointer?
2. What kinds of situations call for a constant pointer?
3. What kinds of situations call for a pointer to a constant?
4. What kinds of situations call for a constant pointer to a constant?
5. What kinds of situations call for a non-constant pointer to a non-constant object?

Exercises

1. Write a program with a pointer to a pointer to a `string` object. Use the pointer to the pointer to call the `size()` member function of the `string` object.
2. Rewrite the Mad Lib Game project from Chapter 5, "Functions: Mad Lib," so that no `string` objects are passed to the function that tells the story. Instead, the function should accept pointers to `string` objects.
3. Will the three memory addresses displayed by the following program all be the same? Explain what's going on in the code.

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

int main()
{
    int a = 10;
    int& b = a;
    int* c = &b;

    cout << &a << endl;
    cout << &b << endl;
    cout << &(*c) << endl;

    return 0;
}
```

This page intentionally left blank