Chapter 10

# Pointers

## CHAPTER GOALS

- To learn how to declare, initialize, and use pointers
- To become familiar with dynamic memory allocation and deallocation
- To use pointers in common programming situations that involve optional and shared objects
- To avoid the common errors of dangling pointers and memory leaks
- To understand the relationship between arrays and pointers
- To be able to convert between string objects and character pointers

An object variable *contains* an object, but a pointer specifies *where* an object is located. In C++, pointers are important for several reasons. Pointers can refer to objects that are *dynamically* allocated whenever they are needed. Pointers can be used for *shared access* to objects. Furthermore, as you will see in Chapter 11, pointers are necessary for implementing *polymorphism*, an important concept in object-oriented programming.

In C++, there is a deep relationship between pointers and arrays. You will see in this chapter how this relationship explains a number of special properties and limitations of arrays. Finally, you will see how to convert between `string` objects and `char*` pointers, which is necessary when interfacing with legacy code.

CHAPTER CONTENTS



## 10.1 Pointers and Memory Allocation

The C++ run-time system can create new objects for us. When we ask for a

```
new Employee
```

then a *memory allocator* finds a storage location for a new employee object. The memory allocator keeps a large storage area, called the *heap*, for that purpose. The heap is a very flexible pool for memory. It can hold values of any type. You can equally ask for

```
new Time
new Product
```

See Syntax 10.1.

When you allocate a new heap object, the memory allocator tells you where the object is located, by giving you the object's *memory address*. To manipulate memory

### Syntax 10.1 : new Expression

```
new type_name
new type_name(expression1, expression2, . . ., expressionn)
```

**Example:**
```
new Time
new Employee("Lin, Lisa", 68000)
```

**Purpose:** Allocate and construct a value on the heap and return a pointer to the value.

### Syntax 10.2 : Pointer Variable Definition

*type_name* variable_name*;
*type_name* variable_name = expression*;

**Example:**
```
Employee* boss;
Product* p = new Product;
```

**Purpose:** Define a new pointer variable, and optionally supply an initial value.

addresses, you need to learn about a new C++ data type: the *pointer*. A pointer to an employee record,

```
Employee* boss;
```

contains the location or memory address for an employee object. A pointer to a time object,

```
Time* deadline;
```

stores the memory address for a time object. See Syntax 10.2.

The types `Employee*` and `Time*` denote pointers to employee and time objects. The `boss` and `deadline` variables of type `Employee*` and `Time*` store the locations or memory addresses of employee and time objects. They cannot store actual employee objects or time objects, however (see Figure 1).

When you create a new object on the heap, you usually want to initialize it. You can supply construction parameters, using the familiar syntax.

```
Employee* boss = new Employee("Lin, Lisa", 68000);
```

When you have a pointer to a value, you often want to access the value to which it points. That action—to go from the pointer to the value—is called *dereferencing*. In C++,

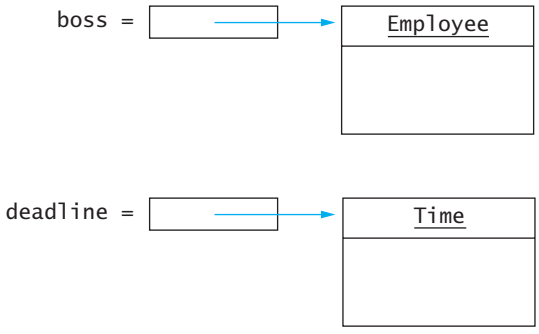



**Figure 1**

Pointers and the Objects to Which They Point

the * operator is used to indicate the value associated with a pointer. For example, if boss is an Employee*, then *boss is an Employee value:

```
Employee* boss = . . .;
raise_salary(*boss, 10);
```

Suppose you want to find out the name of the employee to which boss points:

```
Employee* boss =  . . .;
string name = *boss.get_name(); // Error
```

Unfortunately, that is a syntax error. The dot operator has a higher precedence than the * operator. That is, the compiler thinks that you mean

```
string name = *(boss.get_name()); // Error
```

However, boss is a pointer, not an object. You can't apply the dot (.) operator to a pointer, and the compiler reports an error. Instead, you must make it clear that you first want to apply the * operator, then the dot:

```
string name = (*boss).get_name(); // OK
```

Because this is such a common situation, the designers of C++ supply an operator to abbreviate the "dereference and access member" operation. That operator is written -> and usually pronounced as "arrow".

```
string name = boss->get_name(); // OK
```

Dereferencing of pointers and accessing members through pointers are summarized in Syntax 10.3.

There is one special value, NULL, that can be used to indicate a pointer that doesn't point anywhere. Instead of leaving pointer variables uninitialized, you should always set pointer variables to NULL when you define them.

```
Employee* boss = NULL; // will set later
. . .
if (boss != NULL) name = boss->get_name(); // OK
```

You cannot dereference the NULL pointer. That is, calling *boss or boss->get_name() is an error as long as boss is NULL.

```
Employee* boss = NULL;
string name = boss->get_name(); // NO!! Program will crash
```

The purpose of a NULL pointer is to test that it doesn't point to any valid object.

### Syntax 10.3 : Pointer Dereferencing

**pointer_expression*
*pointer_expression->class_member*

**Example:**
```
*boss
boss->set_salary(70000)
```

**Purpose:** Access the object to which a pointer points.

## Common Error 10.1

### Confusing Pointers with the Data to Which They Point

A pointer is a memory address—a number that tells where a value is located in memory. You can only carry out a small number of operations on a pointer:

- assign it to a pointer variable
- compare it with another pointer or the special value `NULL`
- dereference it to access the value to which it points

However, it is a common error to confuse the pointer with the value to which it points:

```
Employee* boss = . . .;
raise_salary(boss, 10); // ERROR
```

Remember that the pointer `boss` only describes *where* the employee object is. To actually refer to the employee object, use `*boss`:

```
raise_salary(*boss, 10); // OK
```

## Common Error 10.2

### Declaring Two Pointers on the Same Line

It is legal in C++ to define multiple variables together, like this:

```
int i = 0, j = 1;
```

This style does *not* work with pointers:

```
Employee* p, q;
```

For historical reasons, the * associates only with the first variable. That is, `p` is a `Employee*` pointer, and `q` is an `Employee` object. The remedy is to define each pointer variable separately:

```
Employee* p;
Employee* q;
```

You will see some programmers group the * with the variable:

```
Employee *p, *q;
```

While it is a legal declaration, don't use that style. It makes it harder to tell that `p` and `q` are variables of type `Employee*`.

## Advanced Topic 10.1

### The this Pointer

Each member function has a special parameter variable, called `this`, which is a pointer to the implicit parameter. For example, consider the `Product::is_better_than` function of Chapter 8. If you call

```
next.is_better_than(best)
```

then the `this` pointer has type `Product*` and points to the `next` object.

You can use the `this` pointer inside the definition of a method. For example,

```
bool Product::is_better_than(Product b)
{
   if (b.price == 0) return false;
   if (this->price == 0) return true;
   return this->score / this->price > b.score / b.price;
}
```

Here, the expression `this->price` refers to the `price` member of the object to which `this` points, that is, the `price` member of the implicit parameter, or `next.price`. The `this` pointer is not necessary, however, since by convention the expression `price` also refers to the field of the implicit parameter. Nevertheless, some programmers like to use the `this` pointer to make it explicit that `price` is a member and not a variable.

Note that `this` is a pointer whereas `b` is an object. Therefore, we access the `price` member of the implicit parameter as `this->price`, but for the explicit parameter we use `b.price`.

Very occasionally, a member function needs to pass the implicit parameter in its entirety to another function. Since `this` is a pointer to the implicit parameter, `*this` is the actual implicit parameter. For example, suppose someone defined a function

```
void debug_print(string message, Product p)
```

Then the code for the `is_better_than` function might start out with these statements:

```
debug_print("Implicit parameter:", *this);
debug_print("Explicit parameter:", b);
```

## 10.2 Deallocating Dynamic Memory

When you make a variable of type `Employee`, the memory for the employee object is allocated on the *run-time stack*. This memory automatically goes away when the program leaves the block in which the variable is allocated:

```
void f()
{
   Employee harry; // memory for employee allocated on the stack
   . . .
} // memory for employee automatically reclaimed
```

Values that are allocated on the heap do not follow this automatic allocation mechanism.

You allocate values on the heap with `new`, and you must reclaim them using the `delete` operator:

```
void g()
{
   Employee* boss;
   boss = new Employee(. . .);
      // memory for employee allocated on the heap
   . . .
   delete boss; // memory for employee manually reclaimed
}
```

Actually, the foregoing example is a little more complex than that. There are two allocations: one on the stack and one on the heap. The variable `boss` is allocated on the stack. It is of type `Employee*`; that is, `boss` can hold the address of an employee object. Defining the pointer variable does not yet create an `Employee` object. The next line of code allocates an `Employee` object on the heap and stores its address in the pointer variable.

At the end of the block, the storage space for the variable `boss` on the stack is automatically reclaimed. Reclaiming the pointer variable does not automatically reclaim the object to which it points. The memory address is merely forgotten. (That can be a problem—see Common Error 10.4). Therefore, you must manually delete the memory block holding the object.

Note that the pointer variable on the stack has a *name*, namely `boss`. But the employee object, allocated on the heap with `new Employee`, has no name! It can be reached only through the `boss` pointer. Values on the stack always have names; heap values do not.

When a pointer variable is first defined, it contains a random address. Using that random address is an error. In practice, your program will likely crash or mysteriously misbehave if you use an uninitialized pointer:

```
Employee* boss;
string name = boss->get_name(); // NO!! boss contains a random address
```

You must always initialize a pointer so that it points to an actual value before you can use it:

```
Employee* boss = new Employee("Lin, Lisa", 68000);
string name = boss->get_name(); // OK
```

After you delete the value attached to a pointer, you can no longer use that address! The storage space may already be reassigned to another value.

```
delete boss;
string name = boss->get_name(); // NO!! boss points to a deleted element
```

### Syntax 10.4 : `delete` Expression

`delete` *pointer_expression*;

**Example:** `delete boss;`

**Purpose:** Deallocate a value that is stored on the heap and allow the memory to be reallocated.

## Common Error 10.3

### Dangling Pointers

The most common pointer error is to use a pointer that has not been initialized, or that has already been deleted. Such a pointer is called a *dangling* pointer, because it does point somewhere, just not to a valid object. You can create real damage by writing to the location to which it points. Even reading from the location can crash your program.

An uninitialized pointer has a good chance of pointing to an address that your program doesn't own. On most operating systems, attempting to access such a location causes a processor error, and the operating system shuts down the program. You may have seen that happen to other programs—a dialog with a bomb icon or a message such as "general protection fault" or "segmentation fault" comes up, and the program is terminated.

If a dangling pointer points to a valid address inside your program, then writing to it will damage some part of your program. You will change the value of one of your variables, or perhaps damage the control structures of the heap so that after several calls to `new` something crazy happens.

When your program crashes and you restart it, the problem may not reappear, or it may manifest itself in different ways because the random pointer is now initialized with a different random address. Programming with pointers requires iron discipline, because you can create true damage with dangling pointers.

Always initialize pointer variables. If you can't initialize them with the return value of `new`, then set them to `NULL`.

*Never* use a pointer that has been deleted. Some people immediately set every pointer to `NULL` after deleting it. That is certainly helpful:

```
delete first;
first = NULL;
```

However, it is not a complete solution.

```
second = first;
. . .
delete first;
first = NULL;
```

You must still remember that `second` is now dangling. As you can see, you must carefully keep track of all pointers and the corresponding heap objects to avoid dangling pointers.

## Common Error 10.4

### Memory Leaks

The second most common pointer error is to allocate memory on the heap and never deallocate it. A memory block that is never deallocated is called a *memory leak*.

If you allocate a few small blocks of memory and forget to deallocate them, this is not a huge problem. When the program exits, all allocated memory is returned to the operating system.

But if your program runs for a long time, or if it allocates lots of memory (perhaps in a loop), then it can run out of memory. Memory exhaustion will cause your program to crash. In extreme cases, the computer may freeze up if you exhausted all available memory. Avoiding memory leaks is particularly important in programs that need to run for months or years, without restarting.

Even if you write short-lived programs, you should make it a habit to avoid memory leaks. Make sure that every call to the `new` operator has a corresponding call to the `delete` operator.

## Advanced Topic 10.2

### The Address Operator

The `new` operator returns the memory address of a value that is allocated on the heap. You can also obtain the address of a local or global variable, by applying the address (`&`) operator. For example,

```
Employee harry;
Employee* p = &harry;
```

See Figure 2. However, you should never delete an address that you obtained from the `&` operator. Doing so would corrupt the heap, leading to errors in subsequent calls to `new`.

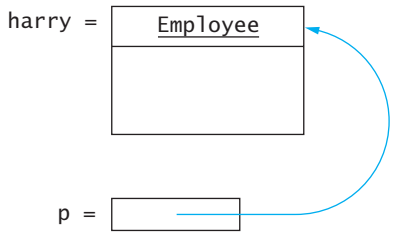



**Figure 2**

The Address Operator

## 10.3 Common Uses for Pointers

In the preceding sections, you have seen how to define pointer variables, and how to make them point to dynamically allocated values. In this section, you will learn how pointers can be useful for solving common programming problems.

In our first example, we will model a `Department` class that describes a department in a company or university, such as the Shipping Department or the Computer Science Department. In our model, a department has

- a name of type `string` (such as `"Shipping"` )
- an *optional* receptionist of type `Employee`

We will use a pointer to model the fact that the receptionist is optional:

```
class Department
{
   . . .
private:
   string name;
   Employee* receptionist;
};
```

If a particular department has a receptionist, then the pointer will be set to the address of an employee object. Otherwise, the pointer will be the special value `NULL`. In the constructor, we set the value to `NULL`:

```
Department::Department(String n)
{
   name = n;
   receptionist = NULL;
}
```

The `set_receptionist` function sets the pointer to the address of an employee object:

```
void Department::set_receptionist(Employee* r)
{
   receptionist = r;
}
```

The `print` function prints either the name of the receptionist or the string `"None"`.

```
void Department::print() const
{
   cout << "Name: " << name
      << "\nReceptionist: ";
   if (receptionist == NULL)
      cout << "None";
   else
       cout << receptionist->get_name()
  cout << "\n";
}
```

Note the use of the `->` operator when calling the `get_name` function. Since `receptionist` is a pointer, and not an object, it would be an error to use the dot operator.

Here take advantage of pointers to model a relationship in which one object may refer to 0 or 1 occurrences of another object. Without pointers, it would have been more difficult and less efficient to express the optional nature of the employee object. You might use a Boolean variable and an object, like this:

```
class Department // modeled without pointers
{
   . . .
private:
   string name;
   boolean has_receptionist;
   Employee receptionist;
};
```

Now those department objects that don't have a receptionist still use up storage space for an unused employee object. Clearly, pointers offer a better solution.

Another common use of pointers is *sharing*. Some departments may have a receptionist and a secretary; in others, one person does double duty. Rather than duplicating objects, we can use pointers to share the object (see Figure 3).

```
class Department
{
   ...
private:
   string name;
   Employee* receptionist;
   Employee* secretary;
};
```

Sharing is particularly important when changes to the object need to be observed by all users of the object. Consider, for example, the following code sequence:

```
Employee* tina = new Employee("Tester, Tina", 50000);
Department qc("Quality Control");
qc.set_receptionist(tina);
qc.set_secretary(tina);
tina->set_salary(55000);
```

Now there are three pointers to the employee object: `tina` and the `receptionist` and `secretary` pointers in the `qc` object. When raising the salary, the new salary is set in the shared object, and the changed salary is visible from all three pointers.

In contrast, we might have modeled the department with two employee objects, like this:

```
class Department // modeled without pointers
{
   . . .
private:
   string name;
   Employee receptionist;
   Employee secretary;
};
```

Now consider the equivalent code:

```
Employee tina("Tester, Tina", 50000);
Department qc("Quality Control");
qc.set_receptionist(tina);
qc.set_secretary(tina);
tina.set_salary(55000);
```

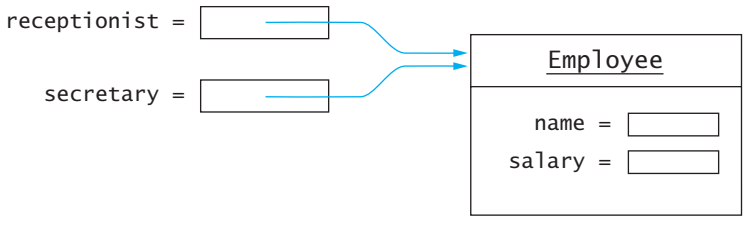
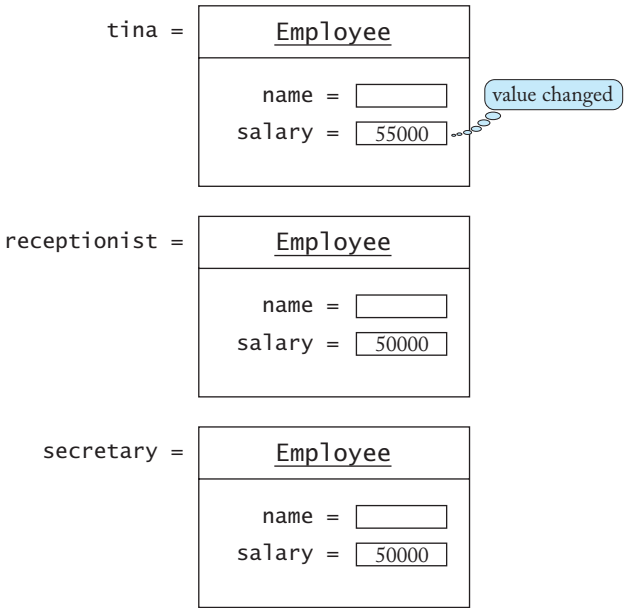



**Figure 3**

Two Pointers Share an Employee Object

**Figure 4**

Three Separate Employee Objects




The department object contains two copies of the `tina` object. When raising the salary, the copies are not affected (see Figure 4).

This example shows that pointers are very useful to model a "*n* : 1" relationship, in which a number of different variables share the same object.

In Chapter 11, you will see another use of pointers, in which a pointer can refer to objects of varying types. That phenomenon, called *polymorphism*, is an important part of object-oriented programming.

The following program gives a complete implementation of the `Department` class. Note how the pointers are used to express optional and shared objects.

## File department.cpp

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

using namespace std;

#include "ccc_empl.h"

/**
   A department in an organization.
*/
class Department
{
public:
   Department(string n);
   void set_receptionist(Employee* e);
   void set_secretary(Employee* e);
   void print() const;
private:
```

```
19    string name;
20    Employee* receptionist;
21    Employee* secretary;
22 };
23
24 /**
25    Constructs a department with a given name.
26    @param n the department name
27 */
28 Department::Department(string n)
29 {
30    name = n;
31    receptionist = NULL;
32    secretary = NULL;
33 }
34
35 /**
36    Sets the receptionist for this department.
37    @param e the receptionist
38 */
39 void Department::set_receptionist(Employee* e)
40 {
41    receptionist = e;
42 }
43
44 /**
45    Sets the secretary for this department.
46    @param e the secretary
47 */
48 void Department::set_secretary(Employee* e)
49 {
50    secretary = e;
51 }
52
53 /**
54    Prints a description of this department.
55 */
56 void Department::print() const
57 {
58    cout << "Name: " << name
59         << "\nReceptionist: ";
60    if (receptionist == NULL)
61       cout << "None";
62    else
63       cout << receptionist->get_name() << " "
64            << receptionist->get_salary();
65    cout << "\nSecretary: ";
66    if (secretary == NULL)
67       cout << "None";
68    else if (secretary == receptionist)
69       cout << "Same";
70    else
71       cout << secretary->get_name() << " "
72            << secretary->get_salary();
73    cout << "\n";
74 }
```

```
75
76 int main()
77 {
78    Department shipping("Shipping");
79    Department qc("Quality Control");
80    Employee* harry = new Employee("Hacker, Harry", 45000);
81    shipping.set_secretary(harry);
82    Employee* tina = new Employee("Tester, Tina", 50000);
83    qc.set_receptionist(tina);
84    qc.set_secretary(tina);
85    tina->set_salary(55000);
86    shipping.print();
87    qc.print();
88
89    return 0;
90 }
```

## Advanced Topic 10.3

### References

In Section 5.8, you saw how to use *reference parameters* in functions that modify variables. For example, consider the function

```
void raise_salary(Employee& e, double by)
{
   double new_salary = e.get_salary() * (1 + by / 100);
   e.set_salary(new_salary);
}
```

This function modifies the first parameter but not the second. That is, if you call the function as

```
raise_salary(harry, percent);
```

then the value of `harry` may change, but the value of `percent` is unaffected.

A reference is a pointer in disguise. The function receives two parameters: the address of an `Employee` object and a copy of a `double` value. The function is logically equivalent to

```
void raise_salary(Employee* pe, double by)
{
   double new_salary = pe->get_salary() * (1 + by / 100);
   pe->set_salary(new_salary);
}
```

The function call is equivalent to the call

```
raise_salary(&harry, percent);
```

This is an example of sharing: the pointer variable in the function modifies the original object, and not a copy.

When you use references, the compiler automatically passes parameter addresses and dereferences the pointer parameters in the function body. For that reason, references are more convenient for the programmer than explicit pointers.

## 10.4 Arrays and Pointers

There is an intimate connection between arrays and pointers in C++. Consider this declaration of an array:

```
int a[10];
```

The value of `a` is a pointer to the starting element (see Figure 5).

```
int* p = a; // now p points to a[0]
```

You can dereference `a` by using the `*` operator: The statement

```
*a = 12;
```

has the same effect as the statement

```
a[0] = 12;
```

Moreover, pointers into arrays support *pointer arithmetic*. You can add an integer offset to the pointer to point at another array location. For example,

```
a + 3
```

is a pointer to the array element with index 3. Dereferencing that pointer yields the element `a[3]`. In fact, for any integer `n`, it is true that

```
a[n] == *(a + n)
```

This relationship is called the *array/pointer duality law*.

This law explains why all C++ arrays start with an index of zero. The pointer `a` (or `a + 0`) points to the starting element of the array. That element must therefore be `a[0]`.

The connection between arrays and pointers becomes even more important when considering array parameters of functions. Consider the `maximum` function from Section 9.5.2.

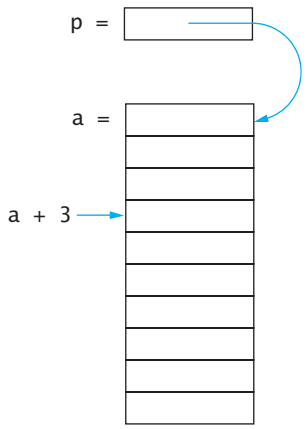



**Figure 5**

Pointers into an Array

```
double maximum(const double a[], int a_size)
{
   if (a_size == 0) return 0;
   double highest = a[0];
   int i;
   for (i = 0; i < a_size; i++)
      if (a[i] > highest)
         highest = a[i];
   return highest;
}
```

Call this function with a particular array:

```
double data[10];
. . . // initialize data
double m = maximum(data, 10);
```

Note the value `data` that is passed to the `maximum` function. It is actually a pointer to the starting element of the array. In other words, the `maximum` function could have equally well been declared as

```
double maximum(const double* a, int a_size)
{
   . . .
}
```

The `const` modifier indicates that the pointer `a` can only be used for reading, not for writing.

The parameter declaration of the first example

```
const double a[]
```

is merely another way of declaring a pointer parameter. The declaration gives the illusion that an entire array is passed to the function, but in fact the function receives only the starting address for the array.

It is essential that the function also knows where the array ends. The second parameter `a_size` indicates the size of the array that starts at `a`.

## Advanced Topic 10.4

### Using a Pointer to Step Through an Array

Now that you know that the first parameter of the `maximum` function is a pointer, you can implement the function in a slightly different way. Rather than incrementing an integer index, you can increment a pointer variable to visit all array elements in turn:

```
double maximum(const double* a, int a_size)
{
   if (a_size == 0) return 0;
   double highest = *a;
   const double* p = a + 1;
   int count = a_size - 1;
   while (count > 0)
   {
```

```
         if (*p > highest)
            highest = *p;
         p++;
         count--;
      }
      return highest;
   }
```

Initially, the pointer `p` points to the element `a[1]`. The increment

```
p++;
```

moves it to point to the next element (see Figure 6).

It is a tiny bit more efficient to dereference and increment a pointer than to access an array element as `a[i]`. For this reason, some programmers routinely use pointers instead of indexes to access array elements. However, the efficiency gain is quite insignificant, and the resulting code is harder to understand, so it is not recommended. (See also Quality Tip 10.1.)

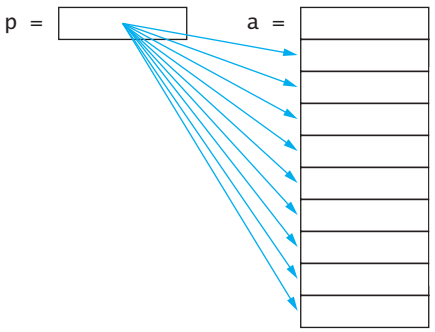



**Figure 6**

A Pointer Variable Traversing the Elements of an Array

## Quality Tip 10.1

### Program Clearly, not Cleverly

Some programmers take great pride in minimizing the number of instructions, even if the resulting code is hard to understand. For example, here is a legal implementation of the `maximum` function:

```
double maximum(const double* a, int a_size)
{
   if (a_size == 0) return 0;
   double highest = *a;
   while (--a_size > 0)
      if (*++a > highest)
         highest = *a;
   return highest;
}
```

This implementation uses two tricks. First, the function parameters `a` and `a_size` are variables, and it is legal to modify them. Moreover, the expressions

```
--a_size
```

and

```
++a
```

mean "decrement or increment the variable and return the new value". Therefore, `*++a` is the location to which `a` points after it has been incremented.

Please do not use this programming style. Your job as a programmer is not to dazzle other programmers with your cleverness, but to write code that is easy to understand and maintain.

## Common Error 10.5

### Confusing Array and Pointer Declarations

It can be confusing to tell whether a particular variable declaration yields a pointer variable or an array variable. There are four cases:

```
int* p; // p is a pointer
int a[10]; // a is an array
int a[] = { 2, 3, 5, 7, 11, 13 }; // a is an array
void f(int a[]); // a is a pointer
```

In the first case, you must initialize `p` to point somewhere before you use it.

## Common Error 10.6

### Returning a Pointer to a Local Array

Consider this function that tries to return a pointer to an array containing two elements, the minimum and the maximum value of an array.

```
double* minmax(const double a[], int a_size)
{
   assert(a_size > 0);
   double result[2];
   result[0] = a[0]; /* result[0] is the minimum */
   result[1] = a[0]; /* result[1] is the maximum */

   for (int i = 0; i < a_size; i++)
   {
      if (a[i] < result[0]) result[0] = a[i];
      if (a[i] > result[1]) result[1] = a[i];
   }
   return result; // ERROR!
}
```

The function returns a pointer to the starting element of the `result` array. However, that array is a local variable of the `minmax` function. The local variable is no longer valid when the function exits, and the values will soon be overwritten by other functions calls.

Unfortunately, it depends on various factors when the values are overwritten. Consider this test of the flawed `minmax` function:

```
double a[] = { 3, 5, 10, 2 };
double* mm = minmax(a, 4);
cout << mm[0] << " " << mm[1] << "\n";
```

One compiler yields the expected result:

```
2 10
```

However, another compiler yields:

```
1.78747e-307 10
```

It just happens that the other compiler chose a different implementation of the `iostream` library that involved more function calls, thereby clobbering the `result[0]` value sooner.

It is possible to work around this limitation, by returning a pointer to an array that is allocated on the heap. But the best solution is to avoid arrays and pointers altogether and to use vectors instead. As you have seen in Chapter 9, a function can easily and safely receive and return `vector<double>` objects:

```
vector<double> minmax(const vector<double>& a)
{
   assert (a.size() > 0);
   vector<double> result(2);
   result[0] = a[0]; /* result[0] is the minimum */
   result[1] = a[0]; /* result[1] is the maximum */

   for (int i = 0; i < a.size(); i++)
   {
      if (a[i] < result[0]) result[0] = a[i];
      if (a[i] > result[1]) result[1] = a[i];
   }
   return result; // OK!
}
```

## Advanced Topic 10.5

### Dynamically Allocated Arrays

You can allocate arrays of values from the heap. For example,

```
int staff_capacity = . . .;
Employee* staff = new Employee[staff_capacity];
```

The `new` operator allocates an array of `n` objects of type `Employee`, each of which is constructed with the default constructor. It returns a pointer to the starting element of the array. Because of array/pointer duality, you can access elements of the array with the `[]` operator: `staff[i]` is the `Employee` element with offset `i`.

To deallocate the array, you use the `delete[]` operator.

```
delete[] staff;
```

It is an error to deallocate an array with the `delete` operator (without the `[]`). However, the compiler can't detect this error—it doesn't remember whether a pointer variable points to a single object or to an array of objects. Therefore, you must be careful and remember which pointer variables point to individual objects and which pointer variables point to arrays.

Heap arrays have one big advantage over array variables. If you declare an array variable, you must specify a fixed array size when you compile the program. But when you allocate an array on the heap, you can choose a different size for each program run.

If you later need more elements, you can allocate a bigger heap array, copy the elements from the smaller array into the bigger array, and delete the smaller array:

```
int bigger_capacity = 2 * staff_capacity;
Employee* bigger = new Employee[bigger_capacity];
for (int i = 0; i < staff_capacity; i++)
   bigger[i] = staff[i];
delete[] staff;
staff = bigger;
staff_capacity = bigger_capacity;
```

As you can see, heap arrays are more flexible than array variables. However, you should not actually use them in your programs. Use `vector` objects instead. A `vector` contains a pointer to a dynamic array, and it automatically manages it for you.

## 10.5 Pointers to Character Strings

C++ has two mechanisms for manipulating strings. The `string` class stores an arbitrary sequence of characters and supports many convenient operations such as concatenation and string comparison. However, C++ also inherits a more primitive level of string handling from the C language, in which strings are represented as arrays of `char` values.

While we don't recommend that you use character pointers or arrays in your programs, you occasionally need to interface with functions that receive or return `char*` values. Then you need to know how to convert between `char*` pointers and `string` objects.

In particular, literal strings such as `"Harry"` are actually stored inside `char` arrays, not `string` objects. When you use the literal string `"Harry"` in an expression, the compiler allocates an array of 6 characters (including a `'\0'` terminator—see Section 9.5.3). The value of the string expression is a `char*` pointer to the starting letter. For example, the code

```
string name = "Harry";
```

is equivalent to

```
char* p = "Harry"; // p points to the letter 'H'
name = p;
```

The `string` class has a constructor `string(char*)` that you can use to convert any character pointer or array to a safe and convenient `string` object. That constructor is called whenever you initialize a `string` variable with a `char*` object, as in the preceding example.

Here is another typical scenario. The `tmpnam` function of the standard library yields a unique string that you can use as the name of a temporary file. It returns a `char*` pointer:

```
char* p = tmpnam(NULL);
```

Simply turn the `char*` return value into a `string` object:

```
string name = p;
```

or

```
string name(p);
```

Conversely, some functions require a parameter of type `char*`. Then use the `c_str` member function of the `string` class to obtain a `char*` pointer that points to the first character in the string object.

For example, the `tempnam` function in the standard library, which also yields a name for a temporary file, lets the caller specify a directory. (Note that the `tmpnam` and `tempnam` function names are confusingly similar.) The `tempnam` function expects a `char*` parameter for the directory name. You can therefore call it as follows:

```
string dir = . . .;
char* p = tempnam(dir.c_str(), NULL);
```

As you can see, you don't have to use character arrays to interface with functions that use `char*` pointers. Simply use `string` objects and convert between `string` and `char*` types when necessary.

## Common Error 10.7

### Confusing Character Pointers and Arrays

Consider the pointer declaration

```
char* p = "Harry";
```

Note that this declaration is entirely different from the array declaration

```
char s[] = "Harry";
```

The second declaration is just a shorthand for

```
char s[6] = { 'H', 'a', 'r', 'r', 'y', '\0' };
```

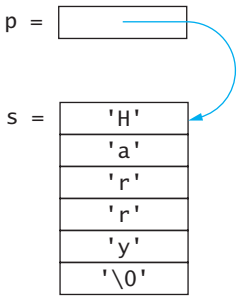



**Figure 7**

Character Pointers and Arrays

The variable p is a pointer that points to the starting character of the string. The characters of the string are stored elsewhere, not in p. In contrast, the variable s is an array of six characters. Perhaps confusingly, when used inside an expression, s denotes a pointer to the starting character in the array. But there is an important difference: p is a pointer *variable* that you can set to another character location. But the value s is *constant*—it always points to the same location. See Figure 7.

## Common Error 10.8

### Copying Character Pointers

There is an important difference between copying `string` objects and pointers of type `char*`. Consider this example:

```
string s = "Harry";
string t = s;
t[0] = 'L'; // now s is "Harry" and t is "Larry"
```

After copying `s` into `t`, the `string` object `t` contains a copy of the characters of `s`. Modifying `t` has no effect on `s`. However, copying character pointers has a completely different effect:

```
char* p = "Harry";
char* q = p;
q[0] = 'L'; // now both p and q point to "Larry"
```

After copying `p` into `q`, the pointer variable `q` contains the same memory address as `p`. The assignment to `q[0]` overwrites the starting letter in the string to which both `p` and `q` point (see Figure 8).

Note that you cannot assign one character array to another. The following assignment is illegal:

```
char a[] = "Harry";
char b[6];
b = a; // ERROR
```

The standard library provides the `strcpy` function to copy a character array to a new location:

```
strcpy(b, a);
```

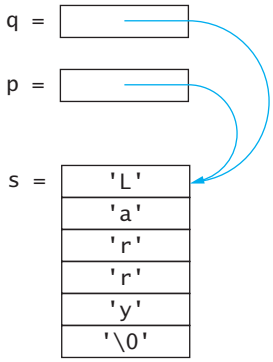



**Figure 8**

Two Character Pointers into the Same Character Array

The target pointer `b` must point to an array with sufficient space in it. It is a common beginner's error to try to copy a string into a character array with insufficient space. There is a safer function, `strncpy`, with a third parameter that specifies the maximum number of characters to copy:

```
strncpy(b, a, 5);
```

An even worse error is to use an uninitialized pointer variable for the target of the copy:

```
char* p;
strcpy(p, "Harry");
```

This is not a syntax error. The `strcpy` function expects two character pointers. However, where is the string copied to? The target address `p` is an uninitialized pointer, pointing to a random location. The characters in the string `"Harry"` are now copied into that random location, either overwriting whatever was there before or triggering a processor exception that terminates the program.

There is an easy way to avoid this bug. Ask yourself, "Where does the storage for the target string come from?" Character arrays don't magically appear; you have to allocate them. The target of the string copy must be a character array of sufficient size to accommodate the characters.

```
char buffer[100];
strcpy(buffer, "Harry"); // OK
```

As you can see, string handling with character arrays and pointers is tedious and error-prone. The `string` class was designed to be a safe and convenient alternative. For that reason, we strongly recommend that you use the `string` class in your own code.

## Chapter Summary

1. A pointer denotes the location of a value in memory.
2. The `*` operator locates the value to which a pointer points.
3. Finding the value to which a pointer points is called dereferencing.
4. Use the `->` operator to access a data member or a member function through an object pointer.
5. The `NULL` pointer does not point to any object.
6. It is an error to dereference an uninitialized pointer or the `NULL` pointer.
7. You can obtain values of any type from the heap with the `new` operator. You must recycle them with the `delete` operator.
8. Pointers can be used to model optional values (by using a `NULL` pointer when the value is not present).
9. Pointers can be used to provide shared access to a common value.
10. The value of an array variable is a pointer to the starting element of the array.

11. Pointer arithmetic means to add an integer offset to an array pointer. The result is a pointer that skips past the given number of elements.

12. The array-pointer duality law states that `a[n]` is identical to `*(a + n)`, where `a` is a pointer into an array and `n` is an integer offset.

13. When passing an array to a function, only the starting address is passed. The parameter declaration *type_name* `a[]` is equivalent to *type_name*`* a`.

14. Low-level string manipulation functions use pointers of type `char*`. You can construct `string` variables from `char*` pointers, and you can use the `c_str` member function to obtain a `char*` pointer from a `string` object.

## Review Exercises

**Exercise R10.1.** Find the mistakes in the following code. Not all lines contain mistakes. Each line depends on the lines preceding it. Watch out for uninitialized pointers, null pointers, pointers to deleted objects, and confusing pointers with objects.

```
1  int* p = new int;
2  p = 5;
3  *p = *p + 5;
4  Employee e1 = new Employee("Hacker, Harry", 34000);
5  Employee e2;
6  e2->set_salary(38000);
7  delete e2;
8  Time* pnow = new Time();
9  Time* t1 = new Time(2, 0, 0);
10 cout << t1->seconds_from(pnow);
11 delete *t1;
12 cout << t1->get_seconds();
13 Employee* e3 = new Employee("Lin, Lisa", 68000);
14 cout << c.get_salary();
15 Time* t2 = new Time(1, 25, 0);
16 cout << *t2.get_minutes();
17 delete t2;
```

**Exercise R10.2.** A pointer variable can contain a pointer to a valid object, a pointer to a deleted object, `NULL`, or a random value. Write code that creates and sets four pointer variables `a`, `b`, `c`, and `d` to show each of these possibilities.

**Exercise R10.3.** What happens when you dereference each of the four pointers that you created in the preceding assignment? Write a test program if you are not sure.

**Exercise R10.4.** What happens if you forget to delete an object that you obtained from the heap? What happens if you delete it twice?

**Exercise R10.5.** What does the following code print?

```
Employee harry = Employee("Hacker, Harry", 35000);
Employee boss = harry;
Employee* pharry = new Employee("Hacker, Harry", 35000);
Employee* pboss = pharry;
boss.set_salary(45000);
(*pboss).set_salary(45000);
cout << harry.get_salary() << "\n";
cout << boss.get_salary() << "\n";
cout << pharry->get_salary() << "\n";
cout << pboss->get_salary() << "\n";
```

**Exercise R10.6.** Pointers are addresses and have a numerical value. You can print out the value of a pointer as `cout << (unsigned long)(p)`. Write a program to compare `p`, `p + 1`, `q`, and `q + 1`, where `p` is an `int*` and `q` is a `double*`. Explain the results.

**Exercise R10.7.** In Chapter 2, you saw that you can use a cast `(int)` to convert a `double` value to an integer. Explain why casting a `double*` pointer to an `int*` pointer doesn't make sense. For example,

```
double values[] = { 2, 3, 5, 7, 11, 13 };
int* p = (int*)values; // why won't this work?
```

**Exercise R10.8.** Which of the following assignments are legal in C++?

```
void f(int p[])
{
   int* q;
   const int* r;
   int s[10];
   p = q; // 1
   p = r; // 2
   p = s; // 3
   q = p; // 4
   q = r; // 5
   q = s; // 6
   r = p; // 7
   r = q; // 8
   r = s; // 9
   s = p; // 10
   s = q; // 11
   s = r; // 12
}
```

**Exercise R10.9.** Given the definitions

```
double values[] = { 2, 3, 5, 7, 11, 13 };
double* p = values + 3;
```

explain the meanings of the following expressions:

(a) `values[1]`

(b) `values + 1`

(c) `*(values + 1)`

(d) `p[1]`
(e) `p + 1`
(f) `p - values`

Exercise R10.10. Explain the meanings of the following expressions:

(a) `"Harry" + 1`
(b) `*("Harry" + 2)`
(c) `"Harry"[3]`
(d) `[4]"Harry"`

Exercise R10.11. How can you implement a function `minmax` that computes both the minimum and the maximum of the values in an array of integers and stores the result in an `int[2]` array?

Exercise R10.12. What is the difference between the following two variable definitions?

(a) `char a[] = "Hello";`
(b) `char* b = "Hello";`

Exercise R10.13. What is the difference between the following three variable definitions?

(a) `char* p = NULL;`
(b) `char* q = "";`
(c) `char r[] = { '\0' };`

Exercise R10.14. Consider this program segment:

```
char a[] = "Mary had a little lamb";
char* p = a;
int count = 0;
while (*p != '\0')
{
   count++;
   while (*p != ' ' && *p != '\0') p++;
   while (*p == ' ') p++;
}
```

What is the value of `count` at the end of the outer `while` loop?

Exercise R10.15. What are the limitations of the `strcat` and `strncat` functions when compared to the `+` operator for concatenating `string` objects?

## Programming Exercises

Exercise P10.1. Implement a class `Person` with the following fields:

- the name
- a pointer to the person's best friend (a `Person*`)
- a popularity counter that indicates how many other people have this person as their best friend

Write a program that reads in a list of names, allocates a `new Person` for each of them, and stores them in a `vector<Person*>`. Then ask the name of the best friend for each of the `Person` objects. Locate the object matching the friend's name and call a `set_best_friend` method to update the pointer and counter. Finally, print out all `Person` objects, listing the name, best friend, and popularity counter for each.

**Exercise P10.2.** Implement a class `Person` with two fields `name` and `age`, and a class `Car` with three fields:

- the model
- a pointer to the owner (a `Person*`)
- a pointer to the driver (also a `Person*`)

Write a program that prompts the user to specify people and cars. Store them in a `vector<Person*>` and a `vector<Car*>`. Traverse the vector of `Person` objects and increment their ages by one year. Finally, traverse the vector of cars and print out the car model, owner's name and age, and driver's name and age.

**Exercise P10.3.** Enhance the `Employee` class to include a pointer to a `BankAccount`. Read in employees and their salaries. Store them in a `vector<Employee>`. For each employee, allocate a new bank account on the heap, except that two consecutive employees with the same last name should share the same account. Then traverse the vector of employees and, for each employee, deposit 1/12th of their annual salary into their bank account. Afterwards, print all employee names and account balances.

**Exercise P10.4.** Enhance the preceding exercise to delete all bank account objects. Make sure that no object gets deleted twice.

**Exercise P10.5.** Write a function that computes the average value of an array of floating-point data:

```
double average(double* a, int a_size)
```

In the function, use a pointer variable, and not an integer index, to traverse the array elements.

**Exercise P10.6.** Write a function that returns a pointer to the maximum value of an array of floating-point data:

```
double* maximum(double a[], int a_size)
```

If `a_size` is 0, return `NULL`.

**Exercise P10.7.** Write a function that reverses the values of an array of floating-point data:

```
void reverse(double a[], int a_size)
```

In the function, use two pointer variables, and not integer indexes, to traverse the array elements.

**Exercise P10.8.** Implement the `strncpy` function of the standard library.

**Exercise P10.9.** Implement the standard library function

```
int strspn(const char s[], const char t[])
```

that returns the length of the prefix of `s` consisting of characters in `t` (in any order).

**Exercise P10.10.** Write a function

```
void reverse(char s[])
```

that reverses a character string. For example, `"Harry"` becomes `"yrraH"`.

**Exercise P10.11.** Using the `strncpy` and `strncat` functions, implement a function

```
void concat(const char a[], const char b[], char result[],
   int result_maxlength)
```

that concatenates the strings `a` and `b` to the buffer `result`. Be sure not to overrun the result. It can hold `result_maxlength` characters, not counting the `'\0'` terminator. (That is, the buffer has `result_maxlength + 1` bytes available.) Be sure to provide a `'\0'` terminator.

**Exercise P10.12.** Add a method

```
void Employee::format(char buffer[], int buffer_maxlength)
```

to the `Employee` class. The method should fill the `buffer` with the name and salary of the employee. Be sure not to overrun the buffer. It can hold `buffer_maxlength` characters, not counting the `'\0'` terminator. (That is, the buffer has `buffer_maxlength + 1` bytes available.) Be sure to provide a `'\0'` terminator.

**Exercise P10.13.** Write a program that reads lines of text and appends them to a `char buffer[1000]`. Stop after reading 1,000 characters. As you read in the text, replace all newline characters `'\n'` with `'\0'` terminators. Establish an array `char* lines[100]`, so that the pointers in that array point to the beginnings of the lines in the text. Only consider 100 input lines if the input has more lines. Then display the lines in reverse order, starting with the last input line.

**Exercise P10.14.** The preceding program is limited by the fact that it can only handle inputs of 1,000 characters or 100 lines. Remove this limitation as follows. Concatenate the input in one long `string` object. Use the `c_str` method to obtain a `char*` into the string's character buffer. Establish the pointers to the beginnings of the lines as a `vector<char*>`.

**Exercise P10.15.** The preceding problem demonstrated how to use the `string` and `vector` classes to implement resizable arrays. In this exercise, you should implement that capability manually. Allocate a buffer of 1,000 characters from the heap (`new char[1000]`). Whenever the buffer fills up, allocate a buffer of twice the size, copy the buffer contents, and delete the old buffer. Do the same for the array of `char*` pointers—start with a `new char*[100]` and keep doubling the size.