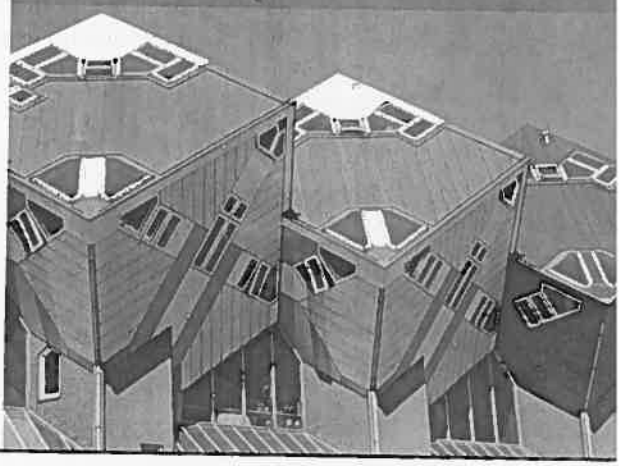
Chapter 8

Arrays and Pointers

8.1 Introduction to Pointers
8.2 Array Names as Pointers
8.3 Pointer Arithmetic
8.4 Passing Addresses
8.5 Common Programming Errors
8.6 Chapter Summary



Programmers often don't consider that memory addresses of variables are used extensively throughout the executable versions of their programs. The computer uses these addresses to keep track of where variables and instructions are physically located in the computer. One of C++'s advantages is that it allows programmers to access these addresses. This access gives programmers a view into a computer's basic storage structure, resulting in capabilities and programming power that aren't available in other high-level languages. This is accomplished by using a feature called pointers. Although other languages provide pointers, C++ extends this feature by providing pointer arithmetic; that is, pointer values can be added, subtracted, and compared.

Fundamentally, pointers are simply variables used to store memory addresses. This chapter discusses the basics of declaring pointers, explains the close relationship of pointers and arrays, and then describes techniques of applying pointer variables in other meaningful ways.

8.1 Introduction to Pointers

In an executable program, every variable has three major items associated with it: the value stored in the variable, the number of bytes reserved for the variable, and where in memory these bytes are located. The memory location of the first byte reserved for a variable is known

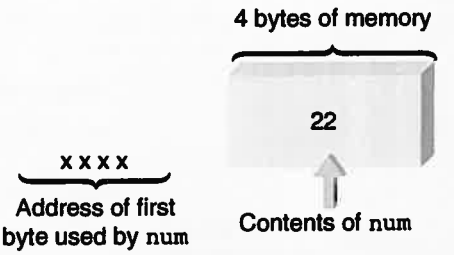


Figure 8.2 The variable num stored somewhere in memory

C++ permits you to go further, however, and display the address corresponding to any variable. The address that's displayed corresponds to the address of the first byte set aside in the computer's memory for the variable.

To determine a variable's address, the address operator, `&`, must be used. You have seen this symbol before in declaring reference variables. When used to display an address, it means "the address of," and when placed in front of a variable name, it's translated as the address of the variable.[2] For example, `&num` means "the address of num," `&total` means "the address of `total`," and `&price` means "the address of `price`." Program 8.2 uses the address operator to display the address of the variable num.

Program 8.2

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

int main()
{
  int num;

  num = 22;
  cout << "The value stored in num is " << num << endl;
  cout << "The address of num = " << &num << endl;

  return 0;
}
```

This is the output of Program 8.2:

```
The value stored in num is 22
The address of num = 0012FED4
```

[2]When used in the declaration of a reference variable (see Section 6.3), the `&` symbol refers to the data type preceding it. For example, the declaration `int &num` is read as "num is the address of an `int`" or, more commonly, "num is a reference to an `int`."

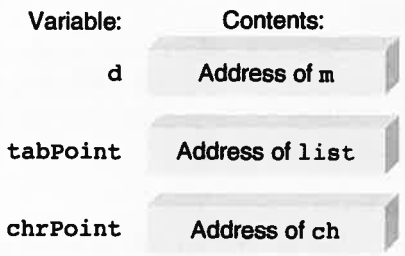


Figure 8.5 Storing more addresses

The variables `numAddr`, `d`, `tabPoint`, and `chrPoint` are formally called **pointer variables** or **pointers**. Pointers are simply variables used to store the addresses of other variables.

Using Addresses

To use a stored address, C++ provides an **indirection operator**, `*`. The `*` symbol, when followed by a pointer (with a space permitted both before and after the `*`), means "the variable whose address is stored in." Therefore, if `numAddr` is a pointer (a variable that stores an address), `*numAddr` means *the variable whose address is stored in* `numAddr`. Similarly, `*tabPoint` means *the variable whose address is stored in* `tabPoint`, and `*chrPoint` means *the variable whose address is stored in* `chrPoint`. Figure 8.6 shows the relationship between the address contained in a pointer variable and the variable.

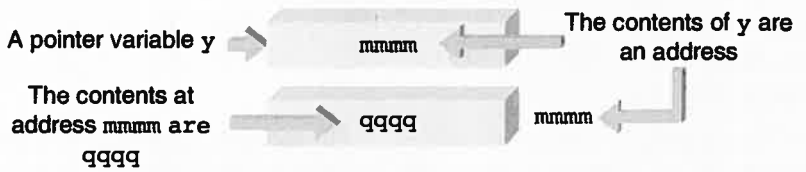


Figure 8.6 Using a pointer variable

Although `*d` means "the variable whose address is stored in `d`," it's commonly shortened to the statement "the variable pointed to by `d`." Similarly, referring to Figure 8.6, `*y` can be read as "the variable pointed to by `y`." The value that's finally obtained, as shown in this figure, is `qqqq`.

When using a pointer variable, the value that's finally obtained is always found by first going to the pointer for an address. The address contained in the pointer is then used to get the variable's contents. Certainly, this procedure is a rather indirect way of getting to the final value, so the term **indirect addressing** is used to describe it.

Because using a pointer requires the computer to do a double lookup (retrieving the address first, and then using the address to retrieve the actual data), you might wonder why you'd want to store an address in the first place. The answer lies in the shared relationship between pointers and arrays and the capability of pointers to create and delete variable storage locations dynamically, as a program is running. Both topics are discussed in the next section. For now, however, given that each variable has a memory address associated with it, the idea of storing an address shouldn't seem unusual.

Program 8.3

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

int main()
{
  int *numAddr;        // declare a pointer to an int
  int miles, dist;     // declare two integer variables

  dist = 158;          // store the number 158 in dist
  miles = 22;          // store the number 22 in miles
  numAddr = &miles;    // store the "address of miles" in numAddr

  cout << "The address stored in numAddr is " << numAddr << endl;
  cout << "The value pointed to by numAddr is " << *numAddr << "\n\n";

  numAddr = &dist;  // now store the address of dist in numAddr
  cout << "The address now stored in numAddr is " << numAddr << endl;
  cout << "The value now pointed to by numAddr is " << *numAddr << endl;

  return 0;
}
```

The output of Program 8.3 is as follows:

```
The address stored in numAddr is 0012FEC8
The value pointed to by numAddr is 22

The address now stored in numAddr is 0012FEBC
The value now pointed to by numAddr is 158
```

The only use for Program 8.3 is to help you understand what gets stored where, so review the program to see how the output was produced. The declaration statement `int *numAddr;` declares `numAddr` to be a pointer used to store the address of an integer variable. The statement `numAddr = &miles;` stores the address of the variable `miles` in the pointer `numAddr`. The first `cout` statement causes this address to be displayed. The second `cout` statement uses the indirection operator (`*`) to retrieve and display the value pointed to by `numAddr`, which is, of course, the value stored in `miles`.

Because `numAddr` has been declared as a pointer to an integer variable, you can use this pointer to store the address of any integer variable. The statement `numAddr = &dist` illustrates this use by storing the address of the variable `dist` in `numAddr`. The last two `cout` statements verify the change in `numAddr`'s value and confirm that the new stored address points to the variable `dist`. As shown in Program 8.3, only addresses should be stored in pointers.

Reference Variables[4] Although references are used almost exclusively as function parameters and return types, they can also be declared as variables. For completeness, this use of references is explained in this section.

After a variable has been declared, it can be given an additional name by using a **reference declaration**, which has this form:

```
dataType& newName = existingName;
```

For example, the reference declaration

```
double& sum = total;
```

equates the name `sum` to the name `total`. Both now refer to the same variable, as shown in Figure 8.8.

Figure 8.8 `sum` is an alternative name for `total`

After establishing another name for a variable by using a reference declaration, the new name, referred to as an **alias**, can be used in place of the original name. For example, take a look at Program 8.4.

Program 8.4

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

int main()
{
  double total = 20.5;     // declare and initialize total
  double& sum = total;     // declare another name for total

  cout << "sum = " << sum << endl;
  sum = 18.6;              // this changes the value in total
  cout << "total = " << total << endl;

  return 0;
}
```

[4]This section can be omitted with no loss of subject continuity.

The following sequence of instructions makes use of this same correspondence between a and b by using pointers:

```
int b;        // b is an integer variable
int *a = &b; // a is a pointer - store b's address in a
*a = 10;      // this changes b's value to 10 by explicit
              // dereference of the address in a
```

Here, a is defined as a pointer initialized to store the address of b. Therefore, `*a` (which can be read as "the variable whose address is in a" or "the variable pointed to by a") is b, and the expression `*a = 10` changes b's value to 10. Notice that with pointers, the stored address can be altered to point to another variable; with references, the reference variable can't be altered to refer to any variable except the one it's initialized to. Also, notice that to dereference a, you must use the indirection operator, *. As you might expect, the * is also called the **dereferencing operator**.

EXERCISES 8.1

1. (Review) What are the three items associated with the variable named `total`?
2. (Review) If `average` is a variable, what does `&average` mean?
3. (Practice) For the variables and addresses in Figure 8.9, determine the addresses corresponding to the expressions `&temp`, `&dist`, `&date`, and `&miles`.

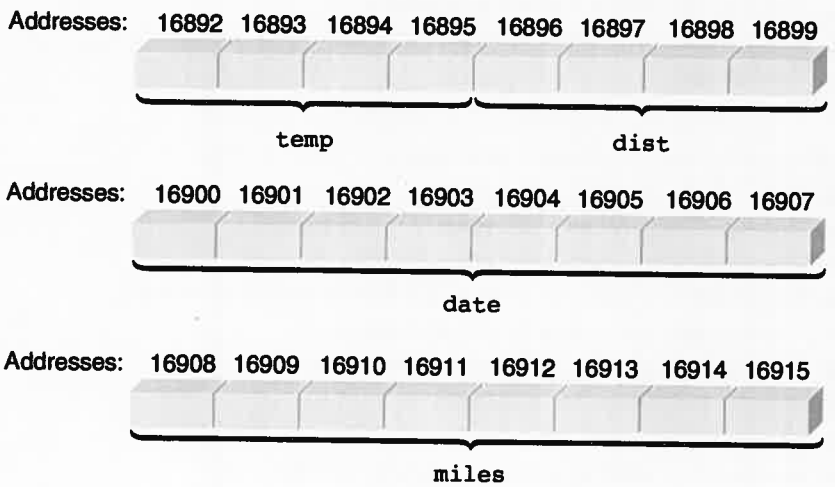


Figure 8.9 Memory bytes for Exercise 3

9. (Practice) Write English sentences that describe what's contained in the following declared variables:
   a. `char *keyAddr;`
   b. `int *m;`
   c. `double *yldAddr;`
   d. `long *yPtr;`
   e. `double *pCou;`
   f. `int *ptDate;`

10. (Practice) Which of the following is a declaration for a pointer?
    a. `long a;`
    b. `char b;`
    c. `char *c;`
    d. `int x;`
    e. `int *p;`
    f. `double w;`
    g. `float *k;`
    h. `float l;`
    i. `double *z;`

11. (Practice) For the following declarations,

```
int *xPt, *yAddr;
long *dtAddr, *ptAddr;
double *ptZ;
int a;
long b;
double c;
```

determine which of the following statements is valid:

a. `yAddr = &a;`	b. `yAddr = &b;`	c. `yAddr = &c;`
d. `yAddr = a;`	e. `yAddr = b;`	f. `yAddr = c;`
g. `dtAddr = &a;`	h. `dtAddr = &b;`	i. `dtAddr = &c;`
j. `dtAddr = a;`	k. `dtAddr = b;`	l. `dtAddr = c;`
m. `ptZ = &a;`	n. `ptAddr = &b;`	o. `ptAddr = &c;`
p. `ptAddr = a;`	q. `ptAddr = b;`	r. `ptAddr = c;`
s. `yAddr = xPt;`	t. `yAddr = dtAddr;`	u. `yAddr = ptAddr;`

12. (Practice) For the variables and addresses in Figure 8.10, fill in the data determined by the following statements:
    a. `ptNum = &m;`
    b. `amtAddr = &amt;`
    c. `*zAddr = 25;`
    d. `k = *numAddr;`
    e. `ptDay = zAddr;`
    f. `*ptYr = 2011;`
    g. `*amtAddr = *numAddr;`

both the array's starting address and the amount of storage each element uses. Calling the fourth element `grade[3]` forces the compiler to make this address computation:

```
&grade[3] = &grade[0] + (3 * sizeof(int))
```

Remembering that the address operator (`&`) means "the address of," this statement is read as "the address of `grade[3]` equals the address of `grade[0]` plus 3 times the size of an integer (which is 12 bytes)." Figure 8.12 shows the address computation used to locate `grade[3]`.

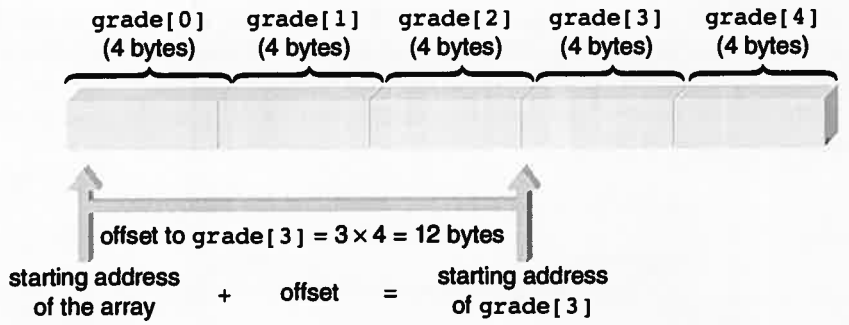


Figure 8.12 Using a subscript to obtain an address

Because a pointer is a variable used to store an address, you can create a pointer to store the address of the first element of an array. Doing so allows you to mimic the computer's operation in accessing array elements. Before you do this, take a look at Program 8.5.

Program 8.5

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

int main()
{
  const int ARRAYSIZE = 5;

  int i, grade[ARRAYSIZE] = {98, 87, 92, 79, 85};

  for (i = 0; i < ARRAYSIZE; i++)
    cout << "\nElement " << i << " is " << grade[i];

  cout << endl;

  return 0;
}
```

Table 8.1 shows the correspondence between elements referenced by subscripts and by pointers and offsets. Figure 8.15 illustrates the relationships listed in this table.

Table 8.1 Array Elements Can Be Referenced in Two Ways

Array Element	Subscript Notation	Pointer Notation
Element 0	`grade[0]`	`*gPtr` or `(gPtr + 0)`
Element 1	`grade[1]`	`*(gPtr + 1)`
Element 2	`grade[2]`	`*(gPtr + 2)`
Element 3	`grade[3]`	`*(gPtr + 3)`
Element 4	`grade[4]`	`*(gPtr + 4)`

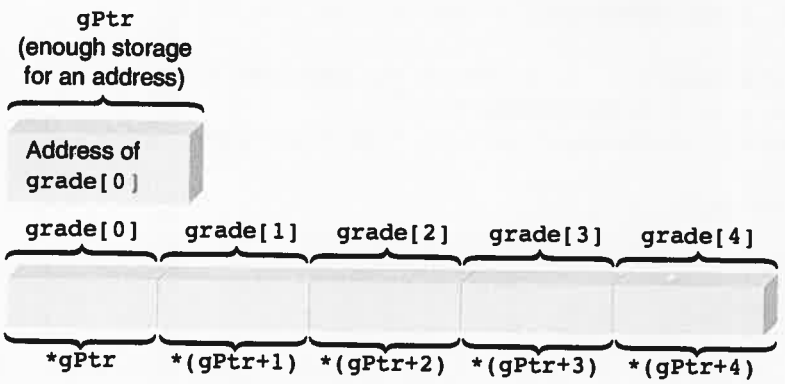


Figure 8.15 The relationship between array elements and pointers

Using the correspondence between pointers and subscripts shown in Figure 8.15, the array elements accessed in Program 8.5 with subscripts can now be accessed with pointers, which is done in Program 8.6.

The following display is produced when Program 8.6 runs:

```
Element 0 is 98
Element 1 is 87
Element 2 is 92
Element 3 is 79
Element 4 is 85
```

is stored in this pointer. Therefore, declaring the grade array in Programs 8.4 and 8.5 actually reserves enough storage for five integers, creates an internal pointer named grade, and stores the address of grade[0] in the pointer, as shown in Figure 8.16.

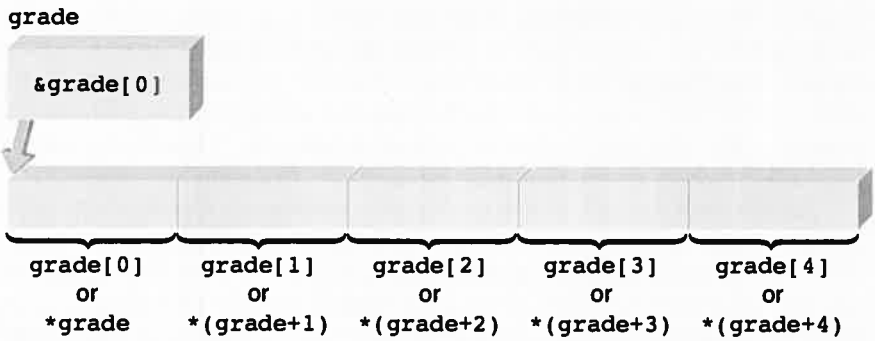


Figure 8.16 Creating an array also creates a pointer

The implication is that every access to grade made with a subscript can be replaced by an access using the array name, grade, as a pointer. Therefore, wherever the expression grade[i] is used, the expression *(grade + i) can also be used. This equivalence is shown in Program 8.7, where grade is used as a pointer to access all its elements. It produces the same output as Programs 8.5 and 8.6. However, using grade as a pointer makes it unnecessary to declare and initialize the pointer gPtr used in Program 8.6.

Program 8.7

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

int main()
{
  const int ARRAYSIZE = 5;

  int i, grade[ARRAYSIZE] = {98, 87, 92, 79, 85};

  for (i = 0; i < ARRAYSIZE; i++)
    cout << "\nElement " << i << " is " << *(grade + i);
  cout << endl;

  return 0;
}
```

Table 8.2 The new and delete Operators (Require the new Header File)

Operator Name	Description
new	Reserves the number of bytes requested by the declaration. Returns the address of the first reserved location or NULL if not enough memory is available.
delete	Releases a block of bytes reserved previously. The address of the first reserved location must be passed as an argument to the operator.

Dynamic storage requests for scalar variables or arrays are made as part of a declaration or an assignment statement.[7] For example, the declaration statement `int *num = new int;` reserves an area large enough to hold one integer and places this storage area's address in the pointer num. This same dynamic allocation can be made by first declaring the pointer with the declaration statement `int *num;` and then assigning the pointer an address with the assignment statement `num = new int;`. In either case, the allocated storage comes from the computer's free storage area.[8]

Dynamic allocation of arrays is similar but more useful. For example, the declaration

```
int *grades = new int[200];
```

reserves an area large enough to store 200 integers and places the first integer's address in the pointer grades. Although the constant 200 has been used in this declaration, a variable dimension can be used. For example, take a look at this sequence of instructions:

```
cout << "Enter the number of grades to be processed: ";
cin  >> numgrades;
int *grades = new int[numgrades];
```

In this sequence, the actual size of the array that's created depends on the number the user inputs. Because pointer and array names are related, each value in the newly created storage area can be accessed by using standard array notation, such as `grades[i]`, instead of the pointer notation `*(grades + i)`. Program 8.8 shows this sequence of code in the context of a complete program.

[7]Note that the compiler provides dynamic allocation and deallocation from the stack for all auto variables automatically.
[8]A computer's free storage area is formally called the heap. It consists of unallocated memory that can be allocated to a program, as requested, while the program is running.

to it, but simply removes the storage the address references. Following is a sample run of Program 8.8:

```
Enter the number of grades to be processed: 4
  Enter a grade: 85
  Enter a grade: 96
  Enter a grade: 77
  Enter a grade: 92

An array was created for 4 integers
 The values stored in the array are:
   85
   96
   77
   92
```

EXERCISES 8.2

1. (Practice) Replace each of the following references to a subscripted variable with a pointer reference:

a. `prices[5]`	b. `grades[2]`	c. `yield[10]`
d. `dist[9]`	e. `mile[0]`	f. `temp[20]`
g. `celsius[16]`	h. `num[50]`	i. `time[12]`

2. (Practice) Replace each of the following pointer references with a subscript reference:

a. `*(message + 6)`	b. `*amount`	c. `*(yrs + 10)`
d. `*(stocks + 2)`	e. `*(rates + 15)`	f. `*(codes + 19)`

3. (Practice) a. List three things the declaration statement `double prices[5];` causes the compiler to do.
   b. If each double-precision number uses 8 bytes of storage, how much storage is set aside for the `prices` array?
   c. Draw a diagram similar to Figure 8.16 for the `prices` array.
   d. Determine the byte offset in relation to the start of the `prices` array, corresponding to the offset in the expression `*(prices + 3)`.

4. (Practice) a. Write a declaration to store the string `"This is a sample"` in an array named `samtest`. Include the declaration in a program that displays the values in `samtest` by using a `for` loop that uses a pointer access to each element in the array.
   b. Modify the program written in Exercise 4a to display only array elements 10 through 15 (the letters s, a, m, p, l, and e).

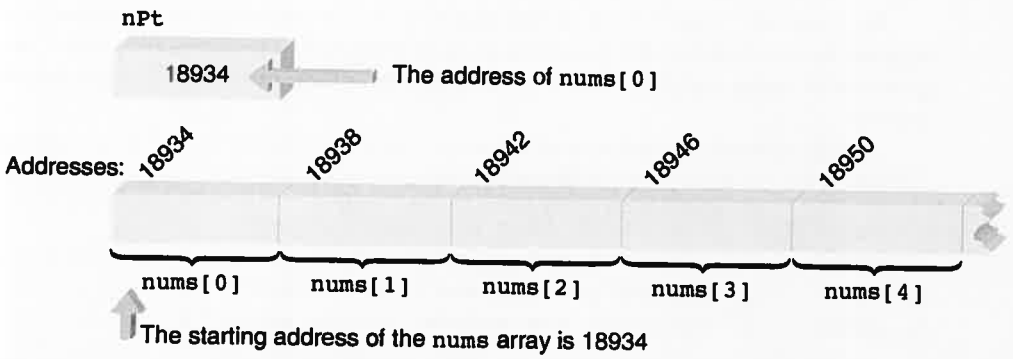


Figure 8.17 The nums array in memory

After `nPt` contains a valid address, values can be added and subtracted from the address to produce new addresses. When adding or subtracting numbers to pointers, the computer adjusts the number automatically to ensure that the result still "points to" a value of the correct type. For example, the statement `nPt = nPt + 4;` forces the computer to scale the 4 by the correct number to make sure the resulting address is the address of an integer. Assuming each integer requires 4 bytes of storage, as shown in Figure 8.17, the computer multiplies the 4 by 4 and adds 16 to the address in `nPt`. The resulting address is 18950, which is the correct address of `nums[4]`.

The computer's automatic scaling ensures that the expression `nPt + i`, where `i` is any positive integer, points to the `i`th element beyond the one currently pointed to by `nPt`. Therefore, if `nPt` initially contains the address of `nums[0]`, `nPt + 4` is the address of `nums[4]`, `nPt + 50` is the address of `nums[50]`, and `nPt + i` is the address of `nums[i]`. Although actual addresses are used in Figure 8.17 to illustrate the scaling process, programmers don't need to be concerned with the actual addresses the computer uses. Manipulating addresses with pointers generally doesn't require knowledge of the actual addresses.

Addresses can also be incremented or decremented with the prefix and postfix increment and decrement operators. Adding 1 to a pointer causes the pointer to point to the next element of the type being pointed to. Decrementing a pointer causes the pointer to point to the previous element. For example, if the pointer variable `p` is a pointer to an integer, the expression `p++` increments the address in the pointer to point to the next integer, as shown in Figure 8.18.

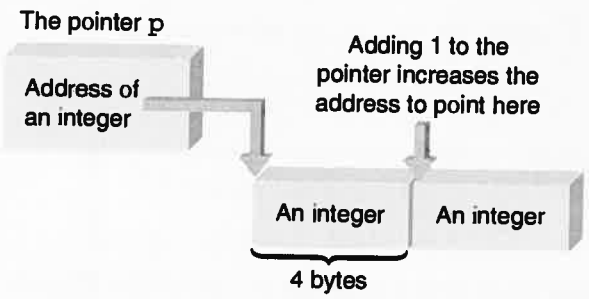


Figure 8.18 Increments are scaled when used with pointers

that nPt then contains the address of the next array element. The computer, of course, scales the increment so that the actual address in nPt is the correct address of the next element.

Pointers can also be compared, which is particularly useful when dealing with pointers that point to elements in the same array. For example, instead of using a counter in a for loop to access each array element, the address in a pointer can be compared to the array's starting and ending addresses. The expression

```
nPt <= &nums[4]
```

is true (non-zero) as long as the address in nPt is less than or equal to the address of nums[4]. Because nums is a pointer constant containing the address of nums[0], the term &nums[4] can be replaced by the equivalent term nums + 4. Using either form, Program 8.9 can be rewritten in Program 8.10 to continue adding array elements while the address in nPt is less than or equal to the address of the last array element.

Program 8.10

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

int main()
{
  const int VALUES = 5;

  int nums[VALUES] = {16, 54, 7, 43, -5};
  int total = 0, *nPt;

  nPt = nums;     // store address of nums[0] in nPt
  while (nPt < nums + VALUES)
    total += *nPt++;

  cout << "The total of the array elements is " << total << endl;

  return 0;
}
```

In Program 8.10, the compact form of the accumulating expression total += *nPt++ was used in place of the longer form, total = total + *nPt++. Also, the expression nums + 4 doesn't change the address in nums. Because nums is an array name, not a pointer variable, its value can't be changed. The expression nums + 4 first retrieves the address in nums, adds 4 to this address (scaled appropriately), and uses the result for comparison purposes. Expressions such as *nums++, which attempt to change the address, are invalid. Expressions such as *nums or *(nums + i), which use the address without attempting to alter it, are valid.

4. (Program) Write a program that stores the following numbers in the array named `miles`: 15, 22, 16, 18, 27, 23, and 20. Have your program copy the data stored in `miles` to another array named `dist`, and then display the values in the `dist` array. Your program should use pointer notation when copying and displaying array elements.

5. (Program) Write a C++ program that stores the following letters in the array named `message`: `This is a test`. Have your program copy the data stored in `message` to another array named `mess2` and then display the letters in the `mess2` array.

6. (Program) Write a program that declares three one-dimensional arrays named `miles`, `gallons`, and `mpg`. Each array should be capable of holding 10 elements. In the `miles` array, store the numbers 240.5, 300.0, 189.6, 310.6, 280.7, 216.9, 199.4, 160.3, 177.4, and 192.3. In the `gallons` array, store the numbers 10.3, 15.6, 8.7, 14, 16.3, 15.7, 14.9, 10.7, 8.3, and 8.4. Each element of the `mpg` array should be calculated as the corresponding element of the `miles` array divided by the equivalent element of the `gallons` array: for example, `mpg[0] = miles[0] / gallons[0]`. Use pointers when calculating and displaying the elements of the `mpg` array.

8.4 Passing Addresses

In Section 6.3, you saw one method of passing addresses to a function: using reference parameters. Passing a reference to a function is an implied use of an address because the reference does provides the function with an address. Unfortunately, the actual call statement doesn't reveal what's being passed—it could be an address or a value. For example, the function call `swap(num1,num2);` doesn't reveal whether `num1` or `num2` is a reference (an address) or a value. Only by looking at the declarations for the variables `num1` and `num2`, or by examining the function header for `swap()`, can you determine the data types of `num1` and `num2`. If they have been defined as reference variables, an address is passed; otherwise, the value stored in the variables is passed.

In contrast to passing addresses implicitly with references, addresses can be passed explicitly with pointers. To pass an address to a function explicitly, all you need to do is place the address operator, `&`, in front of the variable being passed. For example, this function call

```
swap(&firstnum, &secnum);
```

passes the addresses of the variables `firstnum` and `secnum` to `swap()`, as shown in Figure 8.19. This function call also clearly indicates that addresses are being passed to the function.

Program 8.11

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

void swap(double *, double *);   // function prototype
int main()
{
  double firstnum = 20.5, secnum = 6.25;

  swap(&firstnum, &secnum);     // call swap
  return 0;
}

// this function illustrates passing pointer arguments
void swap(double *nm1Addr, double *nm2Addr)
{

  cout << "The number whose address is in nm1Addr is "
       << *nm1Addr << endl;
  cout << "The number whose address is in nm2Addr is "
       << *nm2Addr << endl;
  return;
}
```

In reviewing Program 8.11, note two things. First, the function prototype for `swap()`

```
void swap(double *, double *)
```

declares that `swap()` returns no value directly, and its parameters are two pointers that "point to" double-precision values. When the function is called, it requires that two addresses be passed, and each address is the address of a double-precision value.

Second, the indirection operator is used in `swap()` to access the values stored in `firstnum` and `secnum`. The `swap()` function has no knowledge of these variable names, but it does have the address of `firstnum` stored in `nm1Addr` and the address of `secnum` stored in `nm2Addr`. The expression `*nm1Addr` in the first `cout` statement means "the variable whose address is in `nm1Addr`." It is, of course, the `firstnum` variable. Similarly, the second `cout` statement obtains the value stored in `secnum` as "the variable whose address is in `nm2Addr`." As the output shows, pointers have been used successfully to allow `swap()` to access variables in `main()`. Figure 8.20 illustrates storing addresses in parameters.

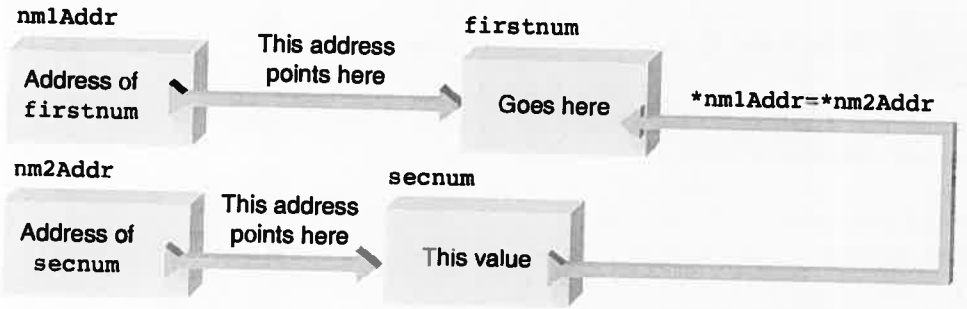


Figure 8.22 Indirectly changing `firstnum`'s value

3. Move the value in the temporary location into the variable whose address is in `nm2Addr` by using the statement `*nm2Addr = temp;` (see Figure 8.23).

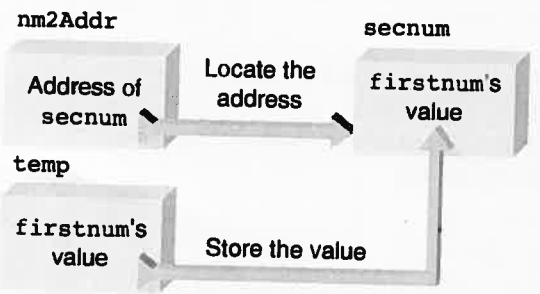


Figure 8.23 Indirectly changing `secnum`'s value

Program 8.12 contains the final form of `swap()`, written according to this description. A sample run of Program 8.12 produced this output:

```
The value stored in firstnum is: 20.5
The value stored in secnum is: 6.25

The value stored in firstnum is now: 6.25
The value stored in secnum is now: 20.5
```

Passing Arrays

When an array is passed to a function, its address is the only item actually passed. "Address" means the address of the first location used to store the array, as shown in Figure 8.24. Because the first location reserved for an array corresponds to element 0 of the array, the "address of the array" is also the address of element 0.

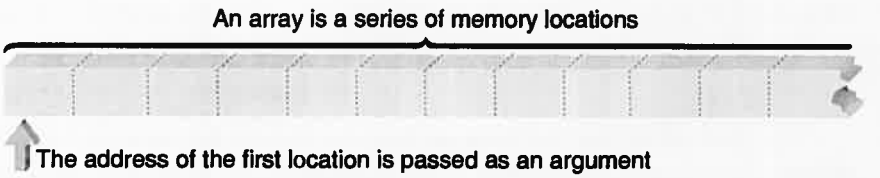


Figure 8.24 An array's address is the address of the first location reserved for the array

For a specific example of passing an array to a function, examine Program 8.13. In this program, the nums array is passed to the `findMax()` function, using conventional array notation.

Program 8.13

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

int findMax(int [], int);     // function prototype
int main()
{
  const int NUMPTS = 5;

  int nums[NUMPTS] = {2, 18, 1, 27, 16};

  cout << "\nThe maximum value is "
       << findMax(nums,NUMPTS) << endl;
  return 0;
}

// this function returns the maximum value in an array of ints
int findMax(int vals[], int numels)
{
  int i, max = vals[0];

  for (i = 1; i < numels; i++)
   if (max < vals[i])
     max = vals[i];
  return max;
}
```

In the first modification to `findMax()`, you make use of this correspondence by simply replacing all references to `vals[i]` with the expression `*(vals + i)`:

```
int findMax(int *vals, int numels)   // find the maximum value
{
  int i, max = *vals;

  for (i = 1; i < numels; i++)
   if (max < *(vals + i) )
     max = *(vals + i);
  return max;
}
```

The second modification of `findMax()` makes use of being able to change the address stored in `vals`. After each array element is retrieved by using the address in `vals`, the address is incremented by 1 in the altering list of the `for` statement. The expression `max = *vals` previously used to set `max` to the value of `vals[0]` is replaced by the expression `max = *vals++`, which adjusts the address in `vals` to point to the second array element. The element this expression assigns to `max` is the array element `vals` points to before it's incremented. The postfix increment, `++`, doesn't change the address in `vals` until after the address has been used to retrieve the first array element.

```
int findMax(int *vals, int numels)   // find the maximum value
{
  int i, max = *vals++;   // get the first element and increment it
  for (i = 1; i < numels; i++, vals++)
  {
    if (max < *vals)
      max = *vals;
  }
  return max;
}
```

Review this version of `findMax()`. Initially, the maximum value is set to "the thing pointed to by `vals`." Because `vals` initially contains the address of the first array element passed to `findMax()`, the value of this first element is stored in `max`. The address in `vals` is then incremented by 1. The 1 added to `vals` is scaled automatically by the number of bytes used to store integers. Therefore, after the increment, the address stored in `vals` is the address of the next array element, as shown in Figure 8.25. The value of this next element is compared with the maximum, and the address is again incremented, this time in the altering list of the `for` statement. This process continues until all array elements have been examined.

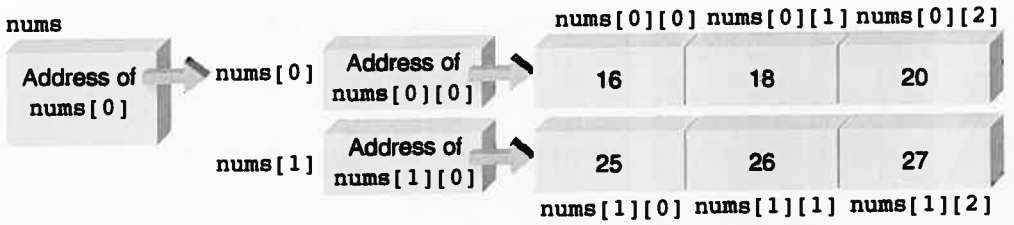


Figure 8.26 Storage of the nums array and associated pointer constants

The availability of the pointer constants associated with a two-dimensional array enables you to access array elements in a variety of ways. One way is to view a two-dimensional array as an array of rows, with each row as an array of three elements. From this viewpoint, the address of the first element in the first row is provided by `nums[0]`, and the address of the first element in the second row is provided by `nums[1]`. Therefore, the variable pointed to by `nums[0]` is `nums[0][0]`, and the variable pointed to by `nums[1]` is `nums[1][0]`. Each element in the array can be accessed by applying an offset to the correct pointer. Therefore, the following notations are equivalent:

Pointer Notation	Subscript Notation	Value
`*nums[0]`	`nums[0][0]`	16
`*(nums[0] + 1)`	`nums[0][1]`	18
`*(nums[0] + 2)`	`nums[0][2]`	20
`*nums[1]`	`nums[1][0]`	25
`*(nums[1] + 1)`	`nums[1][1]`	26
`*(nums[1] + 2)`	`nums[1][2]`	27

You can now go further and replace `nums[0]` and `nums[1]` with their pointer notations, using the address of nums. As shown in Figure 8.26, the variable pointed to by nums is `nums[0]`. That is, `*nums` is `nums[0]`. Similarly, `*(nums + 1)` is `nums[1]`. Using these relationships leads to the following equivalences:

Pointer Notation	Subscript Notation	Value
`*(*nums)`	`nums[0][0]`	16
`*(*nums + 1)`	`nums[0][1]`	18
`*(*nums + 2)`	`nums[0][2]`	20
`*(*(nums + 1))`	`nums[1][0]`	25
`*(*(nums + 1) + 1)`	`nums[1][1]`	26
`*(*(nums + 1) + 2)`	`nums[1][2]`	27

The same notation applies when a two-dimensional array is passed to a function. For example, the two-dimensional array nums is passed to the `calc()` function by using the call

to functions are possible because function names, like array names, are pointer constants. For example, the declaration

```
int (*calc)()
```

declares `calc` to be a pointer to a function that returns an integer. This means `calc` contains the address of a function, and the function whose address is in the variable `calc` returns an integer value. If, for example, the function `sum()` returns an integer, the assignment `calc = sum;` is valid.

EXERCISES 8.4

1. (Practice) The following declaration was used to create the `prices` array:
   ```
   double prices[500];
   ```
   Write three different headers for a function named `sortArray()` that accepts the `prices` array as a parameter named `inArray` and returns no value.

2. (Practice) The following declaration was used to create the `keys` array:
   ```
   char keys[256];
   ```
   Write three different headers for a function named `findKey()` that accepts the `keys` array as a parameter named `select` and returns no value.

3. (Practice) The following declaration was used to create the `rates` array:
   ```
   double rates[256];
   ```
   Write three different headers for a function named `maximum()` that accepts the `rates` array as a parameter named `speed` and returns a double-precision value.

4. (Modify) Modify the `findMax()` function to locate the minimum value of the passed array. Write the function using only pointers.

5. (Debug) In the second version of `findMax()`, `vals` was incremented in the altering list of the `for` statement. Instead, you do the incrementing in the condition expression of the `if` statement, as follows:
   ```
   int findMax(int *vals, int numels)     // incorrect version
   {
     int i, max = *vals++;    // get the first element and increment

     for (i = 1; i < numels; i++)
       if (max < *vals++)
         max = *vals;
     return (max);
   }
   ```
   Determine why this version produces an incorrect result.

```
  int nums[ROWS][COLS] = { {33,16,29},
                           {54,67,99}};
  arr(nums);
  return 0;
}

void arr(int (*val) [3])
{
  cout << endl << *(*val);
  cout << endl << *(*val + 1);
  cout << endl << *(*(val + 1) + 2);
  cout << endl << *(*val) + 1;
  return;
}
```

b. Given the declaration for `val` in the `arr()` function, is the notation `val[1][2]` valid in the function?

8.5 Common Programming Errors

In using the material in this chapter, be aware of the following possible errors:

1. Attempting to store an address in a variable that hasn't been declared as a pointer.
2. Using a pointer to access nonexistent array elements. For example, if `nums` is an array of 10 integers, the expression `*(nums + 15)` points to a location six integer locations beyond the last array element. Because C++ doesn't do bounds checking on array accesses, the compiler doesn't catch this type of error. It's the same error, disguised in pointer notation form, that occurs when using a subscript to access an out-of-bounds array element.
3. Forgetting to use the brackets, `[]`, after the `delete` operator when dynamically deallocating memory that was allocated dynamically as a array.
4. Incorrectly applying address and indirection operators. For example, if `pt` is a pointer variable, both expressions

   ```
   pt = &45
   pt = &(miles + 10)
   ```

   are invalid because they attempt to take the address of a value. Notice that the expression `pt = &miles + 10`, however, is valid. This expression adds 10 to the address of `miles`. It's the programmer's responsibility to ensure that the final address points to a valid data element.

Some confusion surrounding pointers is caused by careless use of the word *pointer*. For example, the phrase "a function requires a pointer argument" is more clearly understood when you realize it actually means "a function requires an address as an argument." Similarly, the phrase "a function returns a pointer" actually means "a function returns an address."

If you're ever in doubt as to what's contained in a variable or how it should be treated, use a `cout` statement to display the variable's contents, the "thing pointed to," or "the address of the variable." Seeing what's actually displayed often helps sort out what the variable contains.

8.6 Chapter Summary

1. Every variable has an address. In C++, you can obtain the address of a variable by using the address operator, `&`.
2. A pointer is a variable used to store the address of another variable. Pointers, like all C++ variables, must be declared. An asterisk, `*`, is used both to declare a pointer variable and to access the variable whose address is stored in a pointer.
3. An array name is a pointer constant. The value of the pointer constant is the address of the first element in the array. Therefore, if `val` is the name of an array, `val` and `&val[0]` can be used interchangeably.
4. Any access to an array element with subscript notation can always be replaced with pointer notation. That is, the notation `a[i]` can always be replaced by the notation `*(a + i)`. This is true whether `a` was initially declared as an array or a pointer.
5. Arrays can be created dynamically as a program is running. For example, the following sequence of statements creates an array named `grades` of size `num`:

   ```
   cout  << "Enter the array size: ";
   cin   >> num;
   int  *grades = new int[num];
   ```

   The area allocated for the array can be destroyed dynamically by using the `delete[]` operator. For example, the statement `delete[] grades;` returns the allocated area for the `grades` array back to the computer.
6. Arrays are passed to functions as addresses. The called function always receives direct access to the originally declared array elements.
7. When a one-dimensional array is passed to a function, the function's parameter declaration can be an array declaration or a pointer declaration. Therefore, the following parameter declarations are equivalent:

   ```
   double a[];
   double *a;
   ```

8. Pointers can be incremented, decremented, compared, and assigned. Numbers added to or subtracted from a pointer are scaled automatically. The scale factor used is the number of bytes required to store the data type originally pointed to.