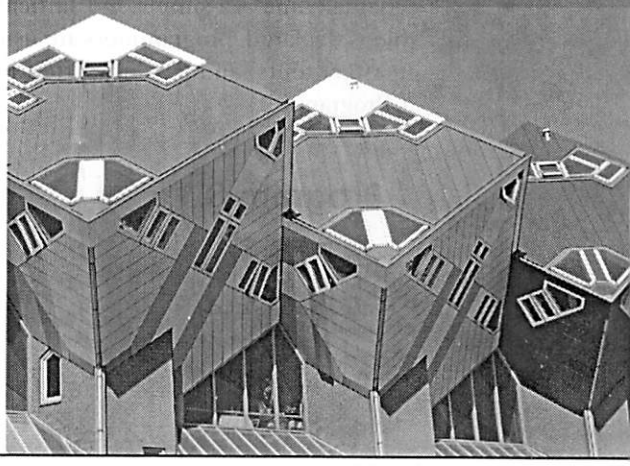
Chapter 8

Arrays and Pointers



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

as the variable's address. Knowing the location of this first byte and how many bytes have been allocated to the variable (which is based on its data type) allows the executable program to access the variable's contents. Figure 8.1 illustrates the relationship between these three items (address, number of bytes, and contents).

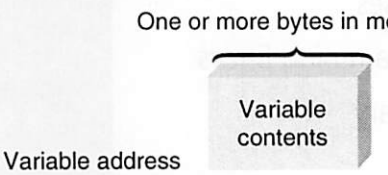


Figure 8.1 A typical variable

For most applications, a variable's internal storage is of little or no concern because the variable name is a simple and sufficient means of locating its contents. Therefore, after a variable is declared, programmers are usually concerned only with the name and value assigned to it (its contents) and pay little attention to where this value is stored. For example, take a look at Program 8.1.

Program 8.1

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

int main()
{

  int num;

  num = 22;
  cout << "The value stored in num is " << num << endl;
  cout <<  sizeof(num) << " bytes are used to store this value" << endl;

  return 0;
}
```

This is the output displayed when Program 8.1 is run:

```
The value stored in num is 22
4 bytes are used to store this value
```

Program 8.1 displays both the number 22, which is the value stored in the integer variable num, and the amount of storage used for this integer variable.[1] Figure 8.2 shows the information that Program 8.1 provides.

[1]The amount of storage allocated for each data type is compiler dependent. Refer to Section 2.1.

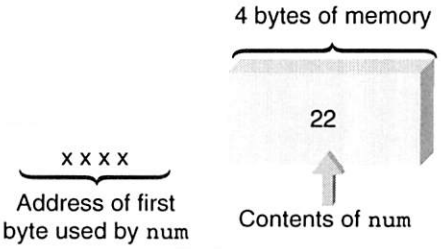


Figure 8.2 The variable num stored somewhere in memory

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 8.3 shows the additional address information provided by Program 8.2's output.

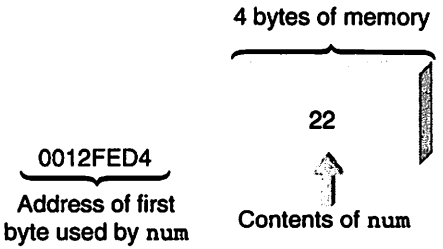


Figure 8.3 A more complete picture of the variable num

Clearly, the address output by Program 8.2 depends on both the computer used to run the program and what other programs or data files are in memory when the program runs. Every time Program 8.2 runs, however, it displays the address of the first memory location used to store num. As Program 8.2's output shows, the address is displayed in hexadecimal notation (see Section 2.6). This display has no effect on how the program uses addresses internally; it merely provides a means of displaying addresses in a more compact representation than the internal binary system used by the computer. As you see in the following sections, however, using addresses (as opposed to just displaying them) gives C++ programmers a powerful programming tool.

Storing Addresses

Besides displaying the address of a variable, as in Program 8.2, you can store addresses in suitably declared variables. For example, the statement

```
numAddr = &num;
```

stores the address corresponding to the variable num in the variable numAddr, as illustrated in Figure 8.4.

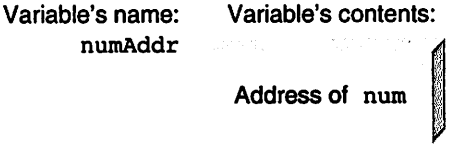


Figure 8.4 Storing num's address in numAddr

Similarly, the statements

```
d = &m;
tabPoint = &list;
chrPoint = &ch;
```

store addresses of the variables m, list, and ch in the variables d, tabPoint, and chrPoint, as shown in Figure 8.5.

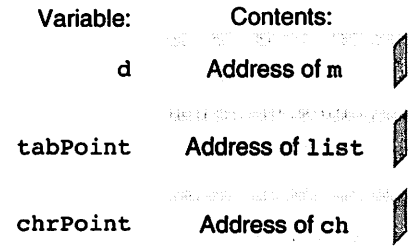


Figure 8.5 Storing more addresses

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.

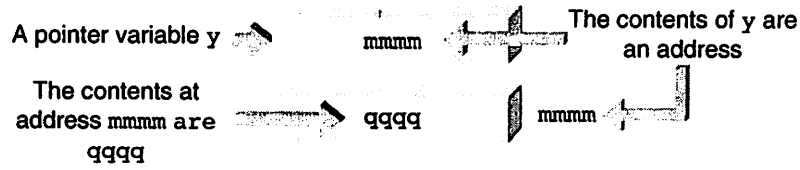


Figure 8.6 Using a pointer variable

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.

Declaring Pointers

Like all variables, pointers must be declared before they can be used to store an address. When you declare a pointer variable, C++ requires also specifying the type of variable that's pointed to. For example, if the address in the pointer `numAddr` is the address of an integer, this is the correct declaration for the pointer:

```
int *numAddr;
```

This declaration is read as "the variable pointed to by `numAddr` (from `*numAddr` in the declaration) is an integer."[3]

Notice that the declaration `int *numAddr;` specifies two things: First, the variable pointed to by `numAddr` is an integer, and second, `numAddr` must be a pointer (because it's declared with an asterisk, `*`). Similarly, if the pointer `tabPoint` points to (contains the address of) a double-precision number and `chrPoint` points to a character variable, the required declarations for these pointers are as follows:

```
double *tabPoint;
char *chrPoint;
```

These two declarations can be read as "the variable pointed to by `tabPoint` is a `double`" and "the variable pointed to by `chrPoint` is a `char`." Because all addresses appear the same, the compiler needs this additional information to know how many storage locations to access when it uses the address stored in the pointer.

Here are other examples of pointer declarations:

```
char *inkey;
int *numPt;
double *nmlPtr
```

To understand pointer declarations, reading them backward is helpful, starting with the asterisk, `*`, and translating it as "the variable whose address is stored in" or "the variable pointed to by." Applying this method to pointer declarations, the declaration `char *inkey;`, for example, can be read as "the variable whose address is stored in `inkey` is a `char`" or "the variable pointed to by `inkey` is a `char`." Both these statements are often shortened to the simpler "`inkey` points to a `char`." All three interpretations of the declaration statement are correct, so you can select and use the description that makes the most sense to you. Program 8.3 puts this information together to construct a program using pointers.

---

[3]Pointer declarations can also be written in the form *`dataType*  pointerName`*, with a space between the indirection operator and the pointer name. This form, however, is error prone when multiple pointers are declared in the same declaration statement and the asterisk is inadvertently omitted after declaring the first pointer name. For example, the declaration `int* num1, num2;` declares num1 as a pointer variable and num2 as an integer variable. To accommodate multiple pointers in the same declaration and clearly mark a variable as a pointer, the examples in this book adhere to the convention of placing an asterisk in front of each pointer name. This potential error rarely occurs with reference declarations because references are used almost exclusively as formal parameters, and single declarations of parameters are mandatory.

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.

It certainly would have been much simpler if the pointer used in Program 8.3 could have been declared as `pointer numAddr;`. This declaration, however, conveys no information about the storage used by the variable whose address is stored in `numAddr`. This information is essential when the pointer is used with the indirection operator, as in the second `cout` statement in Program 8.3. For example, if an integer's address is stored in `numAddr`, typically only 4 bytes of storage are retrieved when the address is used. If a character's address is stored in `numAddr`, only 1 byte of storage is retrieved, and a `double` typically requires retrieving 8 bytes of storage. The declaration of a pointer must, therefore, include the data type of the variable being pointed to, as shown in Figure 8.7.

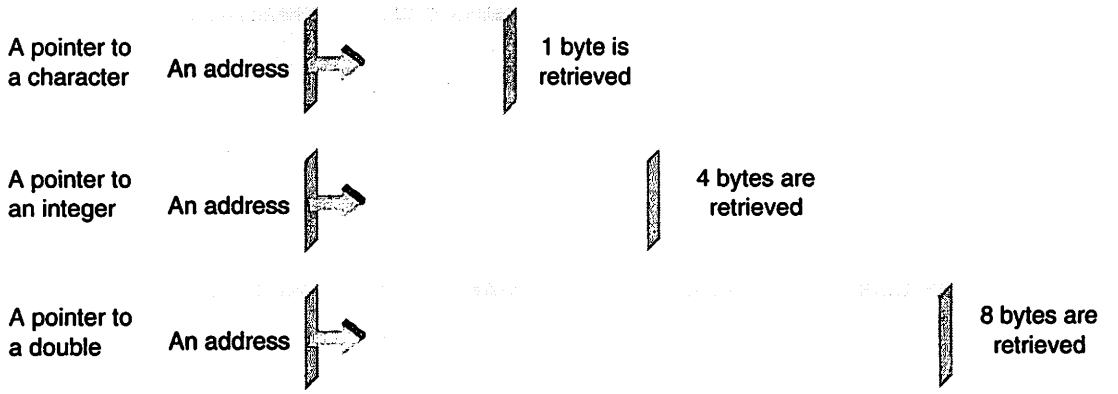


Figure 8.7 Addressing different data types by using pointers

References and Pointers

At this point, you might be asking what the difference is between a pointer and a reference. Essentially, a **reference** is a named constant for an address; therefore, the address named as a reference can't be altered after the address has been assigned. Clearly, for a reference parameter (see Section 6.3), a new reference is created and assigned an address each time the function is called. The address in a pointer, used as a variable or function parameter (discussed in Section 8.4), can be changed after its initial assignment.

In passing an address to a function, beginning programmers tend to prefer using references, as described in Section 6.3. The reason is the simpler notation for reference parameters, which eliminates the address operator (`&`) and indirection operator (`*`) required for pointers. Technically, references are said to be **automatically dereferenced** or **implicitly dereferenced** (the two terms are used synonymously). In contrast, pointers must be explicitly dereferenced by using the indirection operator. In other situations, such as dynamically allocating new sections of memory for additional variables as a program is running and as an alternative to accessing array elements (both discussed in Section 8.2), pointers are required.

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.

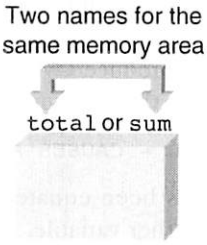


Figure 8.8 `sum` is an alternative name for `total`

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 output is produced by Program 8.4:

```
sum = 20.5
total = 18.6
```

Because the variable sum is simply another reference to the variable total, the first cout statement in Program 8.4 displays the value stored in total. Changing the value in sum then changes the value in total, which the second cout statement in this program displays.

When constructing reference variables, keep two points in mind. First, the reference variable should be of the same data type as the variable it refers to. For example, this sequence of declarations

```
int num = 5;
double& numref = num;  // INVALID - CAUSES A COMPILER ERROR
```

doesn't equate numref to num; rather, it causes a compiler error because the two variables are of different data types.

Second, a compiler error is produced when an attempt is made to equate a reference variable to a constant. For example, the following declaration is invalid:

```
int& val = 5;  // INVALID - CAUSES A COMPILER ERROR
```

After a reference name has been equated to one variable name correctly, the reference *can't* be changed to refer to another variable.

As with all declaration statements, multiple references can be declared in a single statement, as long as each reference name is preceded by an ampersand. Therefore, the following declaration creates two reference variables named sum and average:[5]

```
double& sum = total, & average;
```

Another way of looking at references is to consider them pointers with restricted capabilities that hide a lot of the dereferencing required with pointers. For example, take a look at these statements:

```
int b;      // b is an integer variable
int& a = b; // a is a reference variable that stores b's address
a = 10;     // this changes b's value to 10
```

Here, a is declared as a reference variable that's effectively a named constant for the address of the b variable. Because the compiler knows from the declaration that a is a reference variable, it automatically assigns b's address (rather than b's contents) to a in the declaration statement. Finally, in the statement a = 10;, the compiler uses the address stored in a to change the value stored in b to 10. The advantage of using the reference is that it accesses b's value automatically without having to use the indirection operator, *. As noted previously, this type of access is referred to as an "automatic dereference."

[5]Reference declarations can also be written in the form *dataType &newName = existingName*, with a space between the ampersand and the data type. This form isn't used much, however, because it can be confused easily with the notation used to assign addresses to pointer variables.

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`.

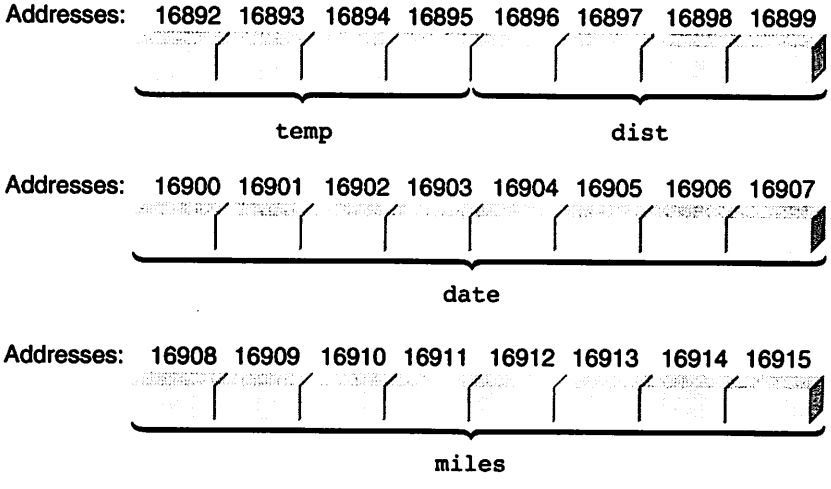


Figure 8.9 Memory bytes for Exercise 3

4. (Practice) a. Write a C++ program that includes the following declaration statements. Have the program use the address operator and a `cout` statement to display the addresses corresponding to each variable.

```
int num, count;
long date;
float slope;
double yield;
```

   b. After running the program written for Exercise 4a, draw a diagram of how your computer has set aside storage for the variables in the program. On your diagram, fill in the addresses the program displays.

   c. Modify the program written in Exercise 4a (using the `sizeof()` operator discussed in Section 2.1) to display the amount of storage your computer reserves for each data type. With this information and the address information provided in Exercise 4b, determine whether your computer set aside storage for the variables in the order in which they were declared.

5. (Review) If a variable is declared as a pointer, what must be stored in the variable?

6. (Practice) Using the indirection operator, write expressions for the following:
   a. The variable pointed to by `xAddr`
   b. The variable whose address is in `yAddr`
   c. The variable pointed to by `ptYld`
   d. The variable pointed to by `ptMiles`
   e. The variable pointed to by `mptr`
   f. The variable whose address is in `pdate`
   g. The variable pointed to by `distPtr`
   h. The variable pointed to by `tabPt`
   i. The variable whose address is in `hoursPt`

7. (Practice) Write declaration statements for the following:
   a. The variable pointed to by `yAddr` is an integer.
   b. The variable pointed to by `chAddr` is a character.
   c. The variable pointed to by `ptYr` is a long integer.
   d. The variable pointed to by `amt` is a double-precision variable.
   e. The variable pointed to by `z` is an integer.
   f. The variable pointed to by `qp` is a single-precision variable.
   g. `datePt` is a pointer to an integer.
   h. `yldAddr` is a pointer to a double-precision variable.
   i. `amtPt` is a pointer to a single-precision variable.
   j. `ptChr` is a pointer to a character variable.

8. (Review) a. What are the variables `yAddr`, `chAddr`, `ptYr`, `amt`, `z`, `qp`, `datePt`, `yldAddr`, `amtPt`, and `ptChr` used in Exercise 7 called?

   b. Why are the variable names `amt`, `z`, and `qp` used in Exercise 7 not good choices for pointer names?

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;`

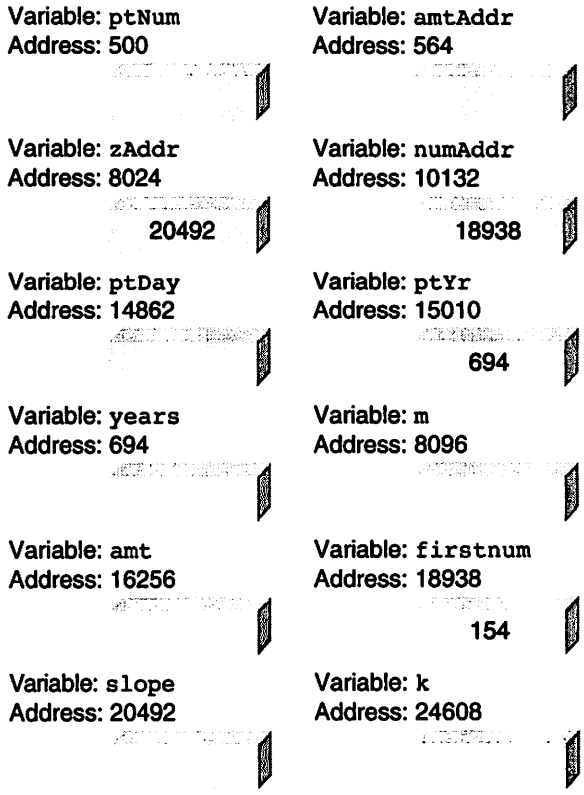


Figure 8.10 Memory locations for Exercise 12

8.2 Array Names as Pointers

Although pointers are simply, by definition, variables used to store addresses, there's also a direct and close relationship between array names and pointers. This section describes this relationship in detail. Figure 8.11 illustrates the storage of a one-dimensional array named `grade`, which contains five integers. Each integer requires 4 bytes of storage.

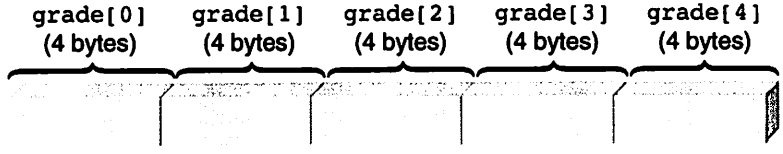


Figure 8.11 The `grade` array in storage

Using subscripts, the fourth element in the `grade` array is referred to as `grade[3]`. The use of a subscript, however, conceals the computer's extensive use of addresses. Internally, the computer immediately uses the subscript to calculate the array element's address, based on

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]`.

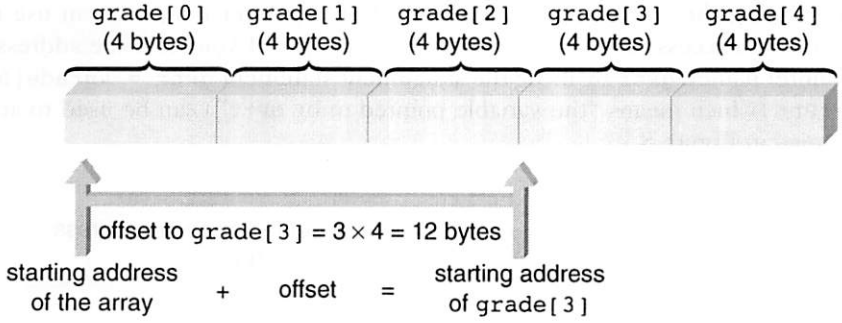


Figure 8.12 Using a subscript to obtain an address

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;
}
```

When Program 8.5 runs, it produces the following display:

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

Program 8.5 displays the values of the `grade` array by using standard subscript notation. By storing the address of array element 0 in a pointer first, you can use the address in the pointer to access each array element. For example, if you store the address of `grade[0]` in a pointer named `gPtr` by using the assignment statement `gPtr = &grade[0];`, the expression `*gPtr` (which means "the variable pointed to by `gPtr`") can be used to access `grade[0]`, as shown in Figure 8.13.

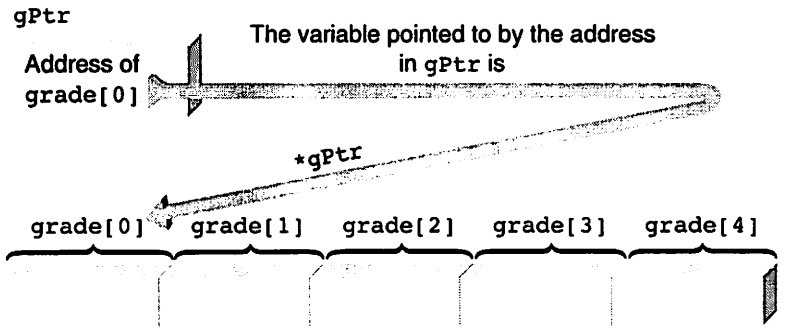


Figure 8.13 The variable pointed to by `*gPtr` is `grade[0]`

One unique feature of pointers is that offsets can be included in expressions using pointers. For example, the 1 in the expression `*(gPtr + 1)` is an **offset**. The complete expression references the integer that's one beyond the variable pointed to by `gPtr`. Similarly, as illustrated in Figure 8.14, the expression `*(gPtr + 3)` references the variable that's three integers beyond the variable pointed to by `gPtr`: the variable `grade[3]`.

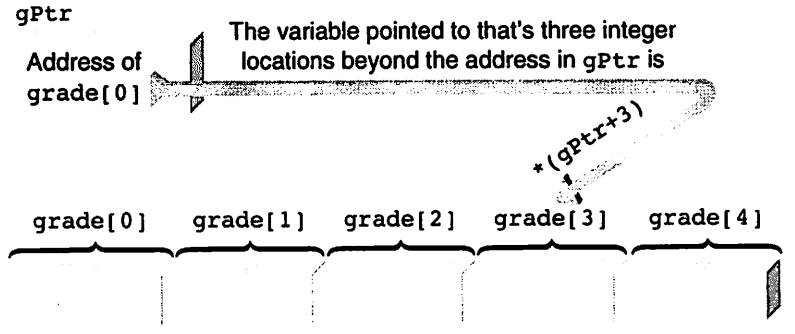


Figure 8.14 An offset of 3 from the address in `gPtr`

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)`

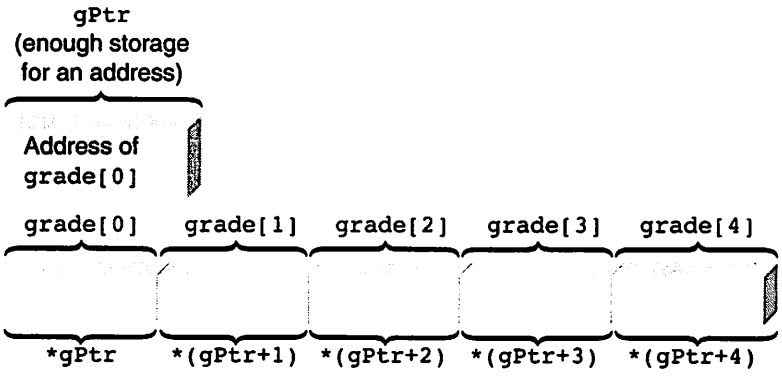


Figure 8.15 The relationship between array elements and pointers

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
```

Program 8.6

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

int main()
{

  const int ARRAYSIZE = 5;

  int *gPtr;              // declare a pointer to an int
  int i, grade[ARRAYSIZE] = {98, 87, 92, 79, 85};

  gPtr = &grade[0];       // store the starting array address
  for (i = 0; i < ARRAYSIZE; i++)
    cout << "\nElement " << i << " is " << *(gPtr + i);

  cout << endl;

  return 0;
}
```

Notice that this display is the same as Program 8.5's display. The method used in Program 8.6 to access array elements simulates how the compiler references array elements internally. The compiler automatically converts any subscript a programmer uses to an equivalent pointer expression. In this case, because the declaration of `gPtr` includes the information that integers are pointed to, any offset added to the address in `gPtr` is scaled automatically by the size of an integer. Therefore, `*(gPtr + 3)`, for example, refers to the address of `grade[0]` plus an offset of 12 bytes (`3 * 4`), assuming `sizeof(int) = 4`. This result is the address of `grade[3]` shown in Figure 8.15.

The parentheses in the expression `*(gPtr + 3)` are necessary to reference an array element correctly. Omitting the parentheses results in the expression `*gPtr + 3`. Because of operator precedence, this expression adds 3 to "the variable pointed to by `gPtr`." Because `gPtr` points to `grade[0]`, this expression adds the value of `grade[0]` and 3 together. Note also that the expression `*(gPtr + 3)` doesn't change the address stored in `gPtr`. After the computer uses the offset to locate the correct variable from the starting address in `gPtr`, the offset is discarded and the address in `gPtr` remains unchanged.

Although the pointer `gPtr` used in Program 8.6 was created specifically to store the `grade` array's starting address, doing so is unnecessary. When an array is created, the compiler creates an internal pointer constant for it automatically and stores the array's starting address in this pointer. In almost all respects, a pointer constant is identical to a programmer-created pointer variable, but as you'll see, there are some differences.

For each array created, the array name becomes the name of the pointer constant the compiler creates for the array, and the starting address of the first location reserved for the array

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.

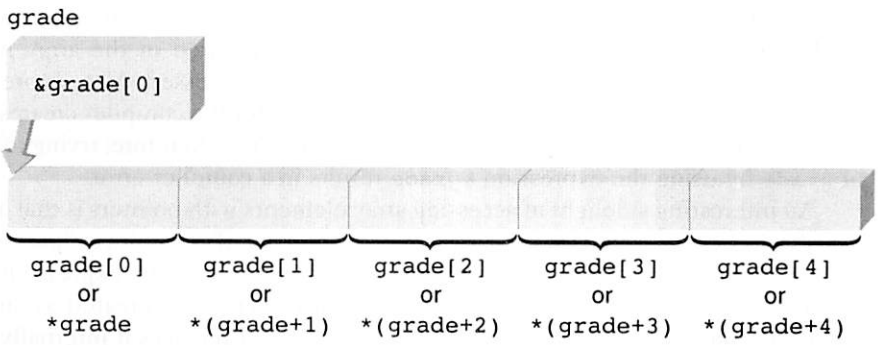


Figure 8.16 Creating an array also creates a pointer

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;
}
```

In most respects, an array name and a pointer can be used interchangeably. A true pointer, however, is a variable, and the address stored in it *can* be changed. An *array name is a pointer constant*, and the address stored in the pointer *can't* be changed by an assignment statement. Therefore, a statement such as `grade = &grade[2];` is invalid. This should come as no surprise. Because the purpose of an array name is to locate the beginning of the array correctly, allowing a programmer to change the address stored in the array name defeats this purpose and leads to havoc when array elements are accessed. Also, expressions taking the address of an array name are invalid because the pointer the compiler creates is internal to the computer, not stored in memory, as pointer variables are. Therefore, trying to store the address of `grade` by using the expression `&grade` results in a compiler error.

An interesting sidelight of accessing array elements with pointers is that any pointer access can always be replaced with a subscript reference, even if the pointer "points to" a scalar variable. For example, if `numPtr` is declared as a pointer variable, the expression `*(numPtr + i)` can also be written as `numPtr[i]`, even though `numPtr` isn't created as an array. As before, when the compiler encounters the subscript notation, it replaces it internally with the equivalent pointer notation.

Dynamic Array Allocation[6]

As each variable is defined in a program, it's assigned sufficient storage from a pool of computer memory locations made available to the compiler. After memory locations have been reserved for a variable, these locations are fixed for the life of that variable, whether they're used or not. For example, if a function requests storage for an array of 500 integers, the storage is allocated and fixed from the point of the array's definition. If the application requires fewer than 500 integers, the unused allocated storage isn't released back to the system until the array goes out of existence. If, on the other hand, the application requires more than 500 integers, the integer array's size must be increased and the function defining the array must be recompiled.

An alternative to this fixed or static allocation of memory storage locations is **dynamic allocation** of memory. Under a dynamic allocation scheme, the amount of storage to be allocated is determined and adjusted at runtime rather than compile time. Dynamic allocation of memory is useful when dealing with lists because it allows expanding the list as new items are added and contracting the list as items are deleted. For example, in constructing a list of grades, you don't need to know the exact number of grades. Instead of creating a fixed array to store grades, having a mechanism for enlarging and shrinking the array as needed is useful. Table 8.2 describes two C++ operators, `new` and `delete`, that provide this capability. (These operators require the `new` header file.)

[6]This topic can be omitted on first reading with no loss of subject continuity.

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.

Program 8.8

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

int main()
{
  int numgrades, i;

  cout << "Enter the number of grades to be processed: ";
  cin  >> numgrades;

  int *grades = new int[numgrades];  // create the array

  for(i = 0; i < numgrades; i++)
  {
    cout << "  Enter a grade: ";
    cin  >> grades[i];
  }
  cout << "\nAn array was created for " << numgrades << " integers\n";
  cout << " The values stored in the array are:";
  for (i = 0; i < numgrades; i++)
    cout << "\n   " << grades[i];
  cout << endl;

  delete[] grades;   // return the storage to the heap
                     // the [] is required for array deletions
  return 0;
}
```

Notice in Program 8.8 that the `delete` operator is used with braces where the `new` operator was used previously to create an array. The `delete[]` statement restores the allocated block of storage back to the free storage area (the heap) while the program is running.[9] The only address `delete` requires is the starting address of the dynamically allocated storage block. Therefore, any address returned by `new` can be used subsequently by `delete` to restore reserved memory back to the computer. The `delete` operator doesn't alter the address passed

[9]The operating system should return allocated storage to the heap automatically when the program has finished running. Because this return doesn't always happen, however, it's crucial to restore dynamically allocated memory explicitly to the heap when the storage is no longer needed. The term **memory leak** is used to describe the condition that occurs when dynamically allocated memory isn't returned explicitly by using the `delete` operator and the operating system doesn't reclaim previously allocated memory.

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).

5. (Practice) Write a declaration to store the following values in an array named `rates`: 12.9, 18.6, 11.4, 13.7, 9.5, 15.2, and 17.6. Include the declaration in a program that displays the values in the array by using pointer notation.

6. (Modify) a. Repeat Exercise 6a in Section 7.1, but use pointer references to access all array elements.
   b. Repeat Exercise 6b in Section 7.1, but use pointer references to access all array elements.

7. (Modify) Repeat Exercise 7 in Section 7.1, but use pointer references to access all array elements.

8. (Modify) As described in Table 8.2, the new operator returns the address of the first new storage area allocated or returns `NULL` if there's insufficient storage. Modify Program 8.8 to check that a valid address has been returned before attempting to place values in the `grades` array. Display an appropriate message if not enough storage is available.

8.3 Pointer Arithmetic

Pointer variables, like all variables, contain values. The value stored in a pointer is, of course, an address. Therefore, by adding and subtracting numbers to pointers, you can obtain different addresses. Additionally, the addresses in pointers can be compared by using any of the relational operators (==, !=, <, >, and so forth) that are valid for comparing other variables. When performing arithmetic on pointers, you must be careful to produce addresses that point to something meaningful. In comparing pointers, you must also make comparisons that make sense. Take a look at these declarations:

```
int nums[100];
int *nPt;
```

To set the address of `nums[0]` in `nPt`, either of these assignment statements can be used:

```
nPt = &nums[0];
nPt = nums;
```

Both assignment statements produce the same result because `nums` is a pointer constant containing the address of the first location in the array: the address of `nums[0]`. Figure 8.17 illustrates the memory allocation resulting from the previous declaration and assignment statements, assuming each integer requires 4 bytes of memory, and the location of the beginning of the `nums` array is address 18934.

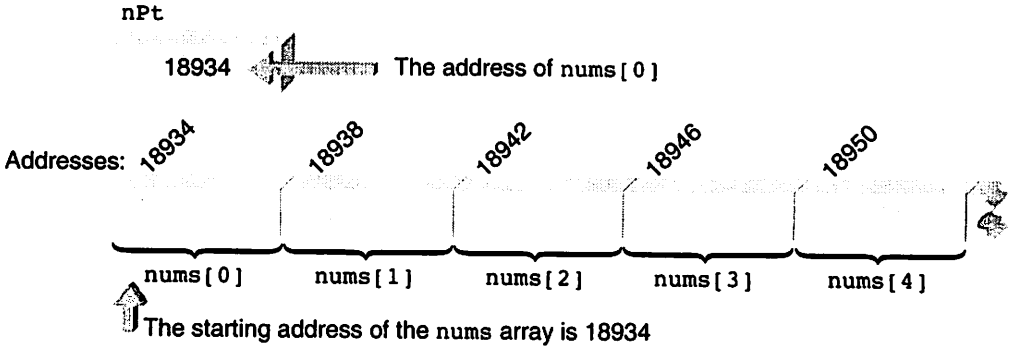


Figure 8.17 The nums array in memory

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.

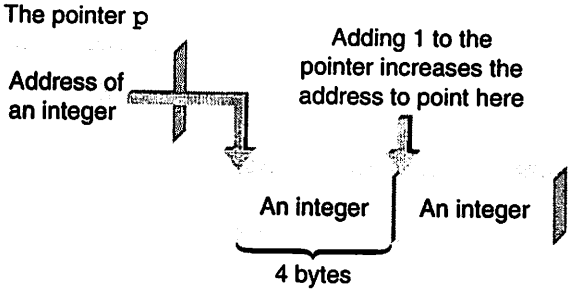


Figure 8.18 Increments are scaled when used with pointers

In reviewing Figure 8.18, notice that the increment added to the pointer is scaled to account for the fact that the pointer is used to point to integers. It is, of course, up to the programmer to make sure the correct type of data is stored in the new address contained in the pointer.

The increment and decrement operators can be applied as both prefix and postfix pointer operators. All the following combinations using pointers are valid:

```
*ptNum++     // use the pointer and then increment it
*++ptNum     // increment the pointer before using it
*ptNum--     // use the pointer and then decrement it
*--ptNum     // decrement the pointer before using it
```

Of these four possible forms, the most commonly used is `*ptNum++` because it allows accessing each array element as the address is "marched along" from the array's starting address to the address of the last array element. Program 8.9 shows this use of the increment operator. In this program, each element in the `nums` array is retrieved by successively incrementing the address in `nPt`.

Program 8.9

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

int main()
{
  const int VALUES = 5;

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

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

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

  return 0;
}
```

Program 8.9 produces the following output:

```
The total of the array elements is 115
```

The expression `total = total + *nPt++` in Program 8.9 accumulates the values pointed to by the `nPt` pointer. In this expression, the `*nPt` part causes the computer to retrieve the integer pointed to by `nPt`. Next, the postfix increment, `++`, adds 1 to the address in `nPt` so

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.

Pointer Initialization

Like all variables, pointers can be initialized when they're declared. When initializing pointers, however, you must be careful to set an address in the pointer. For example, an initialization such as

```
int *ptNum = &miles;
```

is valid only if `miles` is declared as an integer variable before `ptNum` is. This statement creates a pointer to an integer and sets the address in the pointer to the address of an integer variable. If the variable `miles` is declared after `ptNum` is declared, as follows, an error occurs:

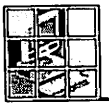
```
int *ptNum = &miles;
int miles;
```

The error occurs because the address of `miles` is used before `miles` has even been defined. Because the storage area reserved for `miles` hasn't been allocated when `ptNum` is declared, the address of `miles` doesn't exist yet.

Pointers to arrays can also be initialized in their declaration statements. For example, if `prices` has been declared as an array of double-precision numbers, either of the following declarations can be used to initialize the pointer `zing` to the address of the first element in `prices`:

```
double *zing = &prices[0];
double *zing = prices;
```

The last initialization is correct because `prices` is a pointer constant containing an address of the correct type. (The variable name `zing` was selected in this example to reinforce the idea that any variable name can be selected for a pointer.)



EXERCISES 8.3

1. (Modify) Replace the `while` statement in Program 8.10 with a `for` statement.
2. (Program) a. Write a program that stores the following numbers in an array named `rates`: 6.25, 6.50, 6.8, 7.2, 7.35, 7.5, 7.65, 7.8, 8.2, 8.4, 8.6, 8.8, and 9.0. Display the values in the array by changing the address in a pointer called `dispPt`. Use a `for` statement in your program.
   b. Modify the program written in Exercise 2a to use a `while` statement.
3. (Program) a. Write a program that stores the string `Hooray for All of Us` in an array named `strng`. Use the declaration `strng[] = "Hooray for All of Us";`, which ensures that the end-of-string escape sequence `\0` is included in the array. Display the characters in the array by changing the address in a pointer called `messPt`. Use a `for` statement in your program.
   b. Modify the program written in Exercise 3a to use the `while` statement `while (*messPt++ != '\0')`.
   c. Modify the program written in Exercise 3a to start the display with the word `All`.

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.

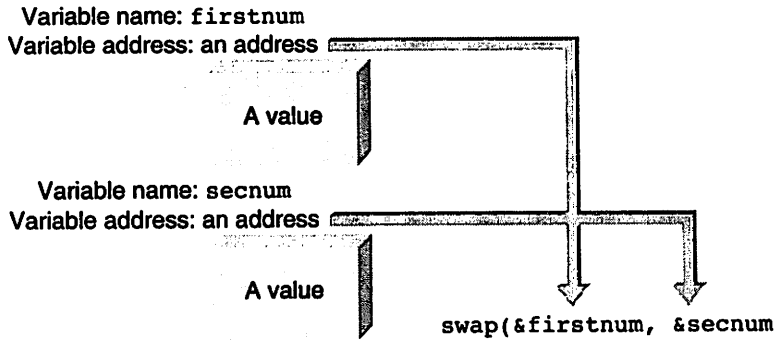


Figure 8.19 Explicitly passing addresses to `swap()`

Passing an address with a reference parameter or the address operator is referred to as a **pass by reference** because the called function can reference, or access, variables in the calling function by using the passed addresses. As you saw in Section 6.3, pass by references can be made with reference parameters. In this section, you see how addresses passed with the address operator are used. Specifically, you use the addresses of the variables `firstnum` and `secnum` passed to `swap()` to exchange their values—a procedure done previously in Program 6.8 with reference parameters.

One of the first requirements in writing `swap()` is to construct a function header that receives and stores the passed values, which in this case are two addresses. As you saw in Section 8.1, addresses are stored in pointers, which means the parameters of `swap()` must be declared as pointers.

Assuming `firstnum` and `secnum` are double-precision variables and `swap()` returns no value, a suitable function header for `swap()` is as follows:

```
void swap(double *nm1Addr, double *nm2Addr);
```

The choice of the parameter names `nm1Addr` and `nm2Addr` is, as with all parameter names, up to the programmer. The declaration `double *nm1Addr`, however, states that the parameter named `nm1Addr` is used to store the address of a double-precision value. Similarly, the declaration `double *nm2Addr` specifies that `nm2Addr` also stores the address of a double-precision value.

Before writing the body of `swap()` to exchange the values in `firstnum` and `secnum`, it's useful to verify that the values accessed by using the addresses in `nm1Addr` and `nm2Addr` are correct. Program 8.11 performs this check.

The output displayed when Program 8.11 runs is as follows:

```
The number whose address is in nm1Addr is 20.5
The number whose address is in nm2Addr is 6.25
```

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.

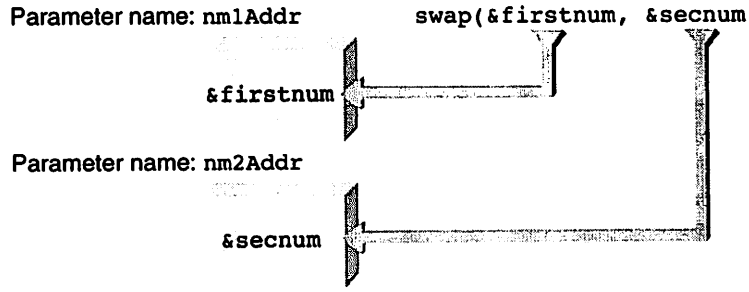


Figure 8.20 Storing addresses in parameters

Having verified that `swap()` can access `main()`'s local variables `firstnum` and `secnum`, you can now expand `swap()` to exchange the values in these variables. The values in `main()`'s variables `firstnum` and `secnum` can be interchanged from within `swap()` by using the three-step interchange algorithm described in Section 6.3:

1. Store `firstnum`'s value in a temporary location.
2. Store secnum's value in `firstnum`.
3. Store the temporary value in secnum.

Using pointers in `swap()`, this algorithm takes the following form:

1. Store the value of the variable that `nm1Addr` points to in a temporary location by using the statement `temp = *nm1Addr;` (see Figure 8.21).

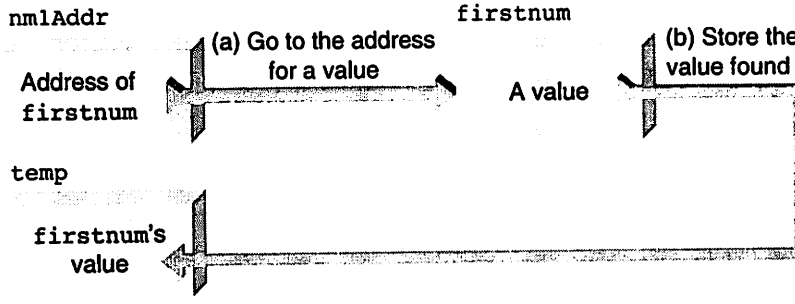


Figure 8.21 Indirectly storing `firstnum`'s value

2. Store the value of the variable whose address is in `nm2Addr` in the variable whose address is in `nm1Addr` with the statement `*nm1Addr = *nm2Addr;` (see Figure 8.22).

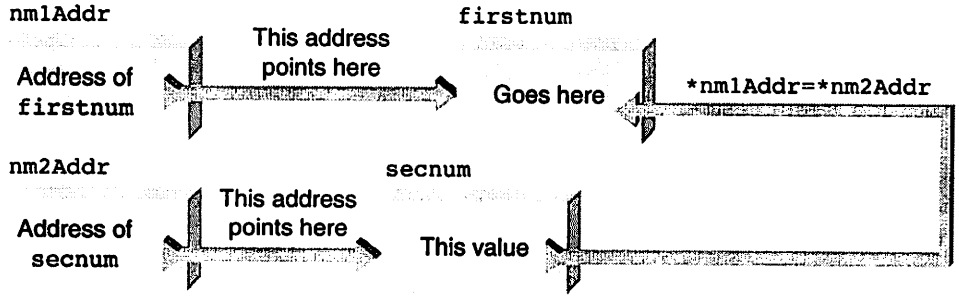


Figure 8.22 Indirectly changing `firstnum`'s value

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).

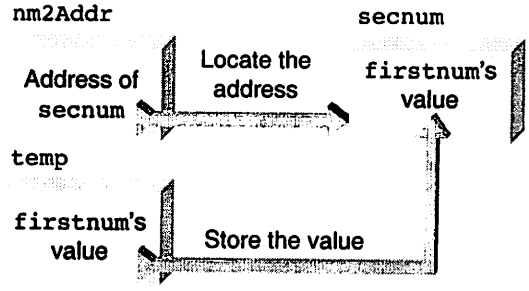


Figure 8.23 Indirectly changing `secnum`'s value

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
```

Program 8.12

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

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

  cout << "The value stored in firstnum is: " << firstnum << endl;
  cout << "The value stored in secnum is: " << secnum << "\n\n";

  swap(&firstnum, &secnum);          // call swap

  cout << "The value stored in firstnum is now: "
       << firstnum <<  endl;
  cout << "The value stored in secnum is now: "
       << secnum << endl;
  return 0;
}

// this function swaps the values in its two arguments
void swap(double *nm1Addr, double *nm2Addr)
{
  double temp;

  temp = *nm1Addr;       // save firstnum's value
  *nm1Addr = *nm2Addr;   // move secnum's value into firstnum
  *nm2Addr = temp;       // change secnum's value
  return;
}
```

As the program output shows, the values stored in `main()`'s variables have been modified in `swap()`, which was made possible by using pointers. To make sure you understand, you could compare this version of `swap()` with the version using references in Program 6.10. The advantage of using pointers rather than references is that the function call specifies that addresses are being used, which is an alert that the function will most likely alter variables of the calling function. The advantage of using references is that the notation is much simpler. Generally, for functions such as `swap()`, ease of notation wins out, and references are used. In passing arrays to functions, however, which is the next topic, the compiler passes an address automatically, which dictates using pointers to store the address.

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.

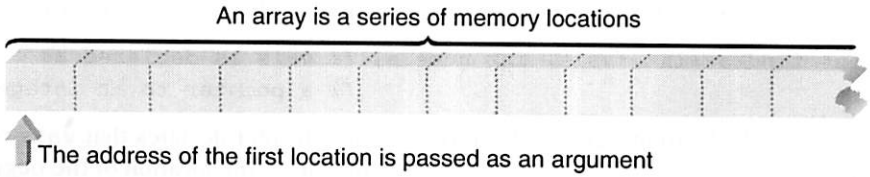


Figure 8.24 An array's address is the address of the first location reserved for the array

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;
}
```

The following output is displayed when Program 8.13 runs:

```
The maximum value is 27
```

The parameter named `vals` in the function header declaration for `findMax()` actually receives the address of the `nums` array. Therefore, `vals` is really a pointer because pointers are variables (or parameters) used to store addresses. Because the address passed to `findMax()` is the address of an integer, the following function header for `findMax()` is also suitable:

```
int findMax(int *vals, int numels) // vals is declared as
                                   // a pointer to an integer
```

The declaration `int *vals` in the function header declares that `vals` is used to store an address of an integer. The address stored is, of course, the location of the beginning of an array. The following is a rewritten version of the `findMax()` function that uses the new pointer declaration for `vals` but retains the use of subscripts to refer to array elements:

```
int findMax(int *vals, int numels)   // find the maximum value
{
  int i, max = vals[0];

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

Regardless of how `vals` is declared in the function header or how it's used in the function body, it's truly a pointer variable. Therefore, the address in `vals` can be modified. This isn't true for the name `nums`, however. Because `nums` is the name of the originally created array, it's a pointer constant. As described in Section 8.2, this means the address in `nums` can't be changed, and the address of `nums` can't be taken. No such restrictions, however, apply to the pointer variable `vals`. Therefore, all the pointer arithmetic you learned in Section 8.3 can be applied to `vals`.

Following are two more versions of `findMax()`, both using pointers instead of subscripts. In the first version, you simply substitute pointer notation for subscript notation. In the second version, you use pointer arithmetic to change the address in the pointer. As stated, access to an array element with the subscript notation *`arrayName`*`[i]` can always be replaced by the pointer notation `*(`*`arrayName`*` + i)`.

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.

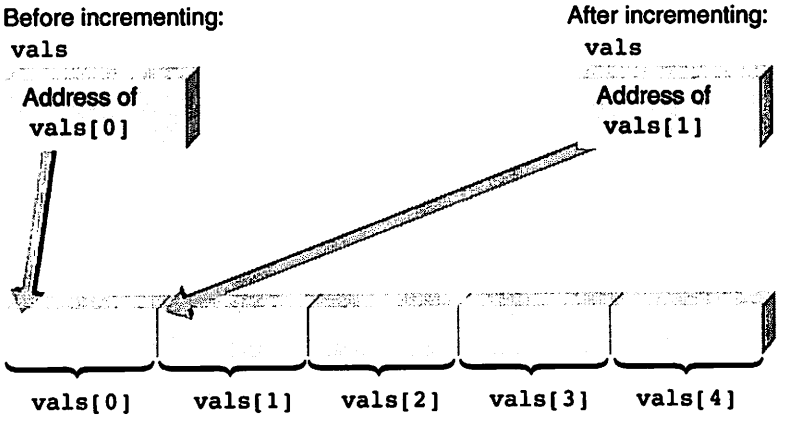


Figure 8.25 Pointing to different elements

The version of `findMax()` you choose is a matter of personal style. Generally, beginning programmers feel more at ease using subscripts rather than pointers. Also, if the program uses an array as the natural storage structure for the application and data, an array access using subscripts is more appropriate to indicate the program's intent clearly. However, as you learn more about data structures, pointers become an increasingly useful and powerful tool. In more complex data structures, there's no simple or easy equivalence for subscripts.

There's one more neat trick you can glean from this discussion. Because passing an array to a function actually involves passing an address, you can pass any valid address. For example, the function call `findMax(&nums[2],3)` passes the address of `nums[2]` to `findMax()`. In `findMax()`, the pointer `vals` stores the address, and the function starts the search for a maximum at the element corresponding to this address. Therefore, from `findMax()`'s perspective, it has received an address and proceeds appropriately.

Advanced Pointer Notation[10]

You can also access multidimensional arrays by using pointer notation, although the notation becomes more cryptic as the array dimensions increase. Pointer notation is especially useful with two-dimensional character arrays, and this section discusses pointer notation for two-dimensional numeric arrays. For example, examine this declaration:

```
int nums[2][3] = { {16,18,20},
                   {25,26,27} };
```

This declaration creates an array of elements and a set of pointer constants named nums, `nums[0]`, and `nums[1]`. Figure 8.26 shows the relationship between these pointer constants and the elements of the nums array.

[10]This topic can be omitted without loss of subject continuity.

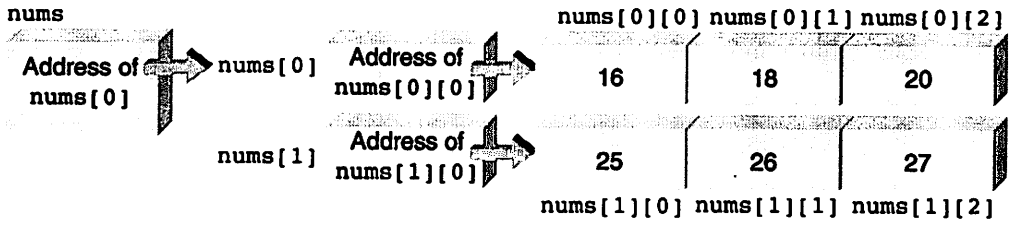


Figure 8.26 Storage of the nums array and associated pointer constants

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

`calc(nums);`. As with all array passes, an address is passed. A suitable function header for the `calc()` function is as follows:

```
void calc(int pt[2][3])
```

As you have seen, the parameter declaration for `pt` can also be the following:

```
void calc(int pt[][3])
```

Using pointer notation, the following is another suitable declaration:

```
void calc(int (*pt)[3])
```

In this declaration, the inner parentheses are required to create a single pointer to arrays of three integers. Each array is, of course, equivalent to a single row of the `nums` array. By offsetting the pointer, each element in the array can be accessed. Notice that without the parentheses, the declaration becomes

```
int *pt[3]
```

which creates an array of three pointers, each one pointing to a single integer. After the correct declaration for `pt` is made (any of the three valid declarations can be used), all the following notations in the `calc()` function are equivalent:

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

The last two notations using pointers are seen in more advanced C++ programs. The first occurs because functions can return any valid C++ scalar data type, including pointers to any of these data types. If a function returns a pointer, the data type being pointed to must be declared in the function's declaration. For example, the declaration

```
int *calc()
```

declares that `calc()` returns a pointer to an integer value, which means the address of an integer variable is returned. Similarly, the declaration

```
double *taxes()
```

declares that `taxes()` returns a pointer to a double-precision value, which means the address of a double-precision variable is returned.

In addition to declaring pointers to integers, double-precision numbers, and C++'s other data types, you can declare pointers that point to (contain the address of) a function. Pointers

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.

6. (Program) a. Write a program that has a declaration in `main()` to store the following numbers in an array named `rates`: 6.5, 7.2, 7.5, 8.3, 8.6, 9.4, 9.6, 9.8, and 10.0. Include a function call to `show()` that accepts `rates` in a parameter named `rates` and then displays the numbers by using the pointer notation `*(rates + i)`.
   b. Modify the `show()` function written in Exercise 6a to alter the address in `rates`. Always use the expression `*rates` rather than `*(rates + i)` to retrieve the correct element.

7. (Program) a. Write a program that has a declaration in `main()` to store the string `Vacation is near` in an array named `message`. Include a function call to `display()` that accepts `message` in an argument named `strng` and then displays the contents of `message` by using the pointer notation `*(strng + i)`.
   b. Modify the `display()` function written in Exercise 7a to use the expression `*strng` rather than `*(strng + i)` to retrieve the correct element.

8. (Program) Write a program that declares three one-dimensional arrays named `price`, `quantity`, and `amount`. Each array should be declared in `main()` and be capable of holding 10 double-precision numbers. The numbers to be stored in `price` are 10.62, 14.89, 13.21, 16.55, 18.62, 9.47, 6.58, 18.32, 12.15, and 3.98. The numbers to be stored in `quantity` are 4, 8.5, 6, 7.35, 9, 15.3, 3, 5.4, 2.9, and 4.8. Have your program pass these three arrays to a function called `extend()`, which calculates the elements in the `amount` array as the product of the equivalent elements in the `price` and `quantity` arrays: for example, `amount[1] = price[1] * quantity[1]`.

   After `extend()` has put values in the `amount` array, display the values in the array from within `main()`. Write the `extend()` function by using pointers.

9. (Program) Write a function named `trimfrnt()` that deletes all leading blanks from a string. Write the function using pointers with the return type `void`.

10. (Program) Write a function named `trimrear()` that deletes all trailing blanks from a string. Write the function using pointers with the return type `void`.

11. (Program) Write a C++ program that asks for two lowercase characters. Pass the two entered characters, using pointers, to a function named `capit()`. The `capit()` function should capitalize the two letters and return the capitalized values to the calling function through its pointer arguments. The calling function should then display all four letters.

12. (Desk check) a. Determine the output of the following program:

```
#include <iostream>
using namespace std;
void arr(int [] [3]); // equivalent to void arr(int (*) [3]);

int main()
{
  const int ROWS = 2;
  const int COLS = 3;
```

```
  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.

5. Taking addresses of pointer constants. For example, given the declarations

```
int nums[25];
int *pt;
```

the assignment

```
pt = &nums;
```

is invalid. The constant nums is a pointer constant that's equivalent to an address. The correct assignment is `pt = nums`.

6. Taking addresses of a reference argument, reference variable, or register variable. The reason is that reference arguments and variables are essentially the same as pointer constants, in that they're named address values. Similarly, the address of a register variable can't be taken. Therefore, for the declarations

```
register int total;
int *ptTot;
```

the assignment

```
ptTot = &total;  // INVALID
```

is invalid. The reason is that register variables are stored in a computer's internal registers, and these storage areas don't have standard memory addresses.

7. Initializing pointer variables incorrectly. For example, the following initialization is invalid:

```
int *pt = 5;
```

Because `pt` is a pointer to an integer, it must be initialized with a valid address.

8. Becoming confused about whether a variable *contains* an address or *is* an address. Pointer variables and pointer arguments contain addresses. Although a pointer constant is synonymous with an address, it's useful to treat pointer constants as pointer variables with two restrictions:

   - The address of a pointer constant can't be taken.
   - The address "contained in" the pointer constant can't be altered.

   Except for these two restrictions, pointer constants and pointer variables can be used almost interchangeably. Therefore, when an address is required, any of the following can be used:

   - A pointer variable name
   - A pointer argument name
   - A pointer constant name
   - A non-pointer variable name preceded by the address operator (for example, `&variable`)
   - A non-pointer argument name preceded by the address operator (for example, `&argument`)

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];
   ```

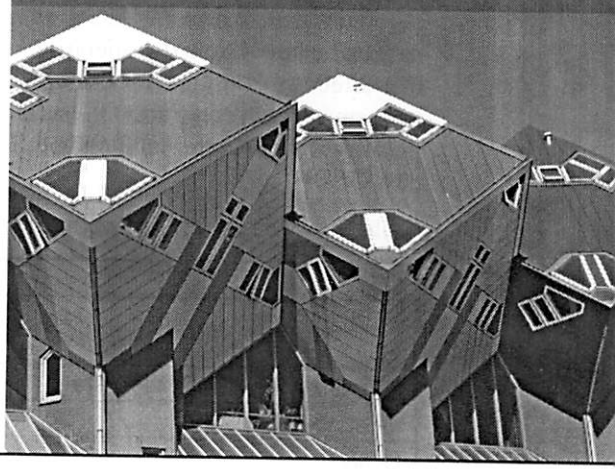
   The area allocated for the array can be destroyed dynamically by using the `delete[]` operator. For example, the statement `delete[] grades;` returns the allocated area for the `grades` array back to the computer.
6. Arrays are passed to functions as addresses. The called function always receives direct 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.

Chapter 9

I/O Streams and Data Files



The data for the programs you have used so far has been assigned internally in the programs or entered by the user during program execution. Therefore, the data used in these programs is stored in the computer's main memory and ceases to exist after the program using it has finished executing. This type of data entry is fine for small amounts of data. However, imagine a company having to pay someone to type in the names and addresses of hundreds or thousands of customers every month when bills are prepared and sent.

As you learn in this chapter, storing large amounts of data outside a program on a convenient storage medium is more sensible. Data stored together under a common name on a storage medium other than the computer's main memory is called a data file. Typically, data files are stored on disks, USB drives, or CD/DVDs. Besides providing permanent storage for data, data files can be shared between programs, so the data one program outputs can be input in another program. In this chapter, you learn how data files are created and maintained in C++.

9.1 I/O File Stream Objects and Methods

To store and retrieve data outside a C++ program, you need two things:

- A file
- A file stream object

Files

A **file** is a collection of data stored together under a common name, usually on a disk, USB drive, or CD/DVD. For example, the C++ programs you store on disk are examples of files. The stored data in a program file is the code that becomes input data to the C++ compiler. In the context of data processing, however, stored programs aren't usually considered data files; the term "data file" typically refers only to files containing the data used in a C++ program.

Each stored data file has a unique filename, referred to as the file's **external name**. The external name is how the operating system (OS) knows the file. When you review the contents of a directory or folder (for example, in Windows Explorer), you see files listed by their external names. Each computer OS has its own specification for the maximum number of characters permitted for an external filename. Table 9.1 lists these specifications for common current and past OSs.

Table 9.1 Maximum Allowable Filename Characters

OS	Maximum Filename Length
DOS	8 characters plus an optional period and 3-character extension
Windows 98, 2000, XP, Vista	255 characters
Windows 7	255 characters
UNIX Early versions Current versions	 14 characters 255 characters

For current OSs, you should take advantage of the increased length specification to create descriptive filenames, but avoid using extremely long filenames because they take more time to type and can result in typing errors. A manageable length for a filename is 12 to 14 characters, with a maximum of 25 characters.

For all the OSs listed in Table 9.1, the following are valid data filenames:

```
prices.dat      records      info.txt
exper1.dat      mvRecord     math.mem
```

Choose filenames that indicate the type of data in the file and the application for which it's used. Typically, the first 8 to 10 characters describe the data, and an optional extension (a period and three or four characters) describes the application used to create the file. For example,