Pointers

CHAPTER

10

IN THIS CHAPTER

Pointers are the hobgoblin of C++ (and C) programming; seldom has such a simple idea inspired so much perplexity for so many. But fear not. In this chapter we will try to demystify pointers and show practical uses for them in C++ programming.

What are pointers for? Here are some common uses:

- Accessing array elements
- Passing arguments to a function when the function needs to modify the original argument
- Passing arrays and strings to functions
- Obtaining memory from the system
- Creating data structures such as linked lists

Pointers are an important feature of C++ (and C), while many other languages, such as Visual Basic and Java, have no pointers at all. (Java has references, which are sort of watered-down pointers.) Is this emphasis on pointers really necessary? You can do a lot without them, as their absence from the preceding chapters demonstrates. Some operations that use pointers in C++ can be carried out in other ways. For example, array elements can be accessed with array notation rather than pointer notation (we'll see the difference soon), and a function can modify arguments passed by reference, as well as those passed by pointers.

However, in some situations pointers provide an essential tool for increasing the power of C++. A notable example is the creation of data structures such as linked lists and binary trees. In fact, several key features of C++, such as virtual functions, the `new` operator, and the `this` pointer (discussed in Chapter 11, "Virtual Functions"), require the use of pointers. So, although you can do a lot of programming in C++ without using pointers, you will find them essential to obtaining the most from the language.

In this chapter we'll introduce pointers gradually, starting with fundamental concepts and working up to complex pointer applications.

If you already know C, you can probably skim over the first half of the chapter. However, you should read the sections in the second half on the `new` and `delete` operators, accessing member functions using pointers, arrays of pointers to objects, and linked-list objects.

Addresses and Pointers

The ideas behind pointers are not complicated. Here's the first key concept: Every byte in the computer's memory has an *address*. Addresses are numbers, just as they are for houses on a street. The numbers start at 0 and go up from there—1, 2, 3, and so on. If you have 1MB of memory, the highest address is 1,048,575. (Of course you have much more.)

Your program, when it is loaded into memory, occupies a certain range of these addresses. That means that every variable and every function in your program starts at a particular address. Figure 10.1 shows how this looks.

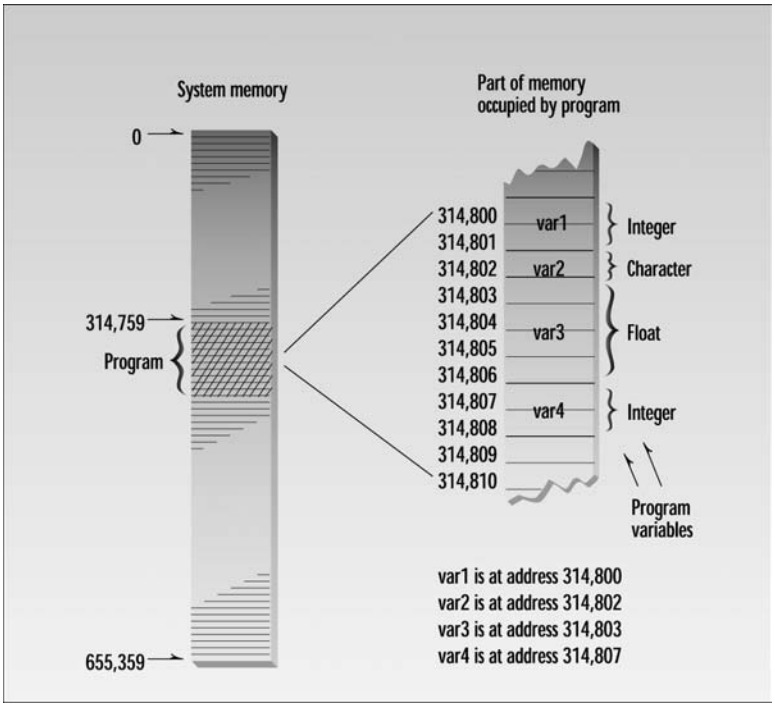

FIGURE 10.1
Memory addresses.

The Address-of Operator &

You can find the address occupied by a variable by using the *address-of* operator &. Here's a short program, VARADDR, that demonstrates how to do this:

```
// varaddr.cpp
// addresses of variables
#include <iostream>
using namespace std;

int main()
   {
   int var1 = 11;           //define and initialize
   int var2 = 22;           //three variables
   int var3 = 33;
```

```
   cout << &var1 << endl    //print the addresses
        << &var2 << endl    //of these variables
        << &var3 << endl;
   return 0;
   }
```

This simple program defines three integer variables and initializes them to the values 11, 22, and 33. It then prints out the addresses of these variables.

The actual addresses occupied by the variables in a program depend on many factors, such as the computer the program is running on, the size of the operating system, and whether any other programs are currently in memory. For these reasons you probably won't get the same addresses we did when you run this program. (You may not even get the same results twice in a row.) Here's the output on our machine:

```
0x8f4ffff4     <------  address of var1
0x8f4ffff2     <------  address of var2
0x8f4ffff0     <------  address of var3
```

Remember that the *address* of a variable is not at all the same as its *contents*. The contents of the three variables are 11, 22, and 33. Figure 10.2 shows the three variables in memory.

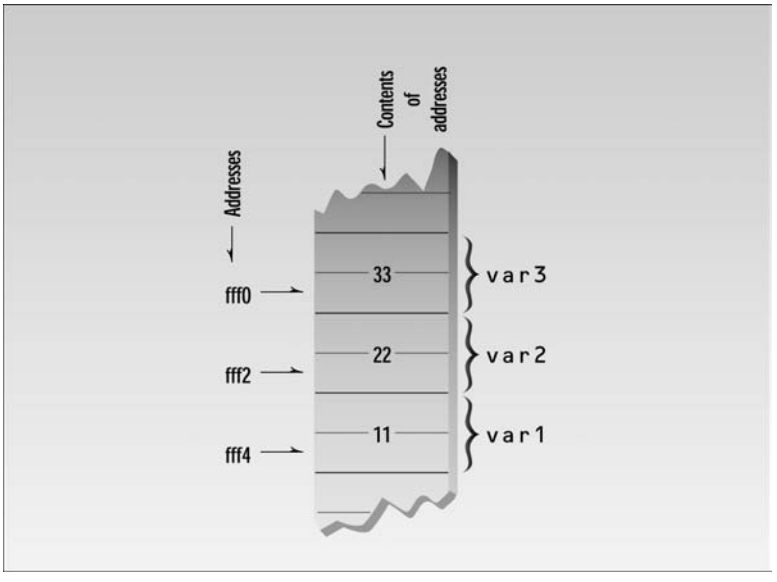

FIGURE 10.2
Addresses and contents of variables.

The << insertion operator interprets the addresses in hexadecimal arithmetic, as indicated by the prefix `0x` before each number. This is the usual way to show memory addresses. If you aren't familiar with the hexadecimal number system, don't worry. All you really need to know is that each variable starts at a unique address. However, you might note in the output that each address differs from the next by exactly 2 bytes. That's because integers occupy 2 bytes of memory (on a 16-bit system). If we had used variables of type `char`, they would have adjacent addresses, since a `char` occupies 1 byte; and if we had used type `double`, the addresses would have differed by 8 bytes.

The addresses appear in descending order because local variables are stored on the stack, which grows downward in memory. If we had used global variables, they would have ascending addresses, since global variables are stored on the heap, which grows upward. Again, you don't need to worry too much about these considerations, since the compiler keeps track of the details for you.

Don't confuse the address-of operator `&`, which precedes a variable name in a variable declaration, with the reference operator `&`, which follows the type name in a function prototype or definition. (References were discussed in Chapter 5, "Functions.")

Pointer Variables

Addresses by themselves are rather limited. It's nice to know that we can find out where things are in memory, as we did in VARADDR, but printing out address values is not all that useful. The potential for increasing our programming power requires an additional idea: *variables that hold address values*. We've seen variable types that store characters, integers, floating-point numbers, and so on. Addresses are stored similarly. A variable that holds an address value is called a *pointer variable*, or simply a *pointer*.

What is the data type of pointer variables? It's not the same as the variable whose address is being stored; a pointer to `int` is not type `int`. You might think a pointer data type would be called something like `pointer` or `ptr`. However, things are slightly more complicated. The next program, PTRVAR, shows the syntax for pointer variables.

```
// ptrvar.cpp
// pointers (address variables)
#include <iostream>
using namespace std;

int main()
   {
   int var1 = 11;             //two integer variables
   int var2 = 22;
```

```
   cout << &var1 << endl      //print addresses of variables
        << &var2 << endl << endl;

   int* ptr;                  //pointer to integers

   ptr = &var1;               //pointer points to var1
   cout << ptr << endl;       //print pointer value

   ptr = &var2;               //pointer points to var2
   cout << ptr << endl;       //print pointer value
   return 0;
   }
```

This program defines two integer variables, `var1` and `var2`, and initializes them to the values 11 and 22. It then prints out their addresses.

The program next defines a *pointer variable* in the line

```
int* ptr;
```

To the uninitiated this is a rather bizarre syntax. The asterisk means *pointer to*. Thus the statement defines the variable `ptr` as a *pointer to* `int`. This is another way of saying that this variable can hold the addresses of integer variables.

What's wrong with the idea of a general-purpose pointer type that holds pointers to any data type? If we called it type `pointer` we could write declarations like

```
pointer ptr;
```

The problem is that the compiler needs to know *what kind of variable the pointer points to*. (We'll see why when we talk about pointers and arrays.) The syntax used in C++ allows pointers to any type to be declared:

```
char* cptr;         // pointer to char
int* iptr;          // pointer to int
float* fptr;        // pointer to float
Distance* distptr;  // pointer to user-defined Distance class
```

and so on.

Syntax Quibbles

We should note that it is common to write pointer definitions with the asterisk closer to the variable name than to the type.

```
char *charptr;
```

It doesn't matter to the compiler, but placing the asterisk next to the type helps emphasize that the asterisk is part of the variable type (pointer to `char`), not part of the name itself.

If you define more than one pointer of the same type on one line, you need only insert the type-pointed-to once, but you need to place an asterisk before each variable name.

```
char* ptr1, * ptr2, * ptr3;  // three variables of type char*
```

Or you can use the asterisk-next-to-the-name approach.

```
char *ptr1, *ptr2, *ptr3;  // three variables of type char*
```

Pointers Must Have a Value

An address like 0x8f4ffff4 can be thought of as a *pointer constant*. A pointer like `ptr` can be thought of as a *pointer variable*. Just as the integer variable `var1` can be assigned the constant value 11, so can the pointer variable `ptr` be assigned the constant value 0x8f4ffff4.

When we first define a variable, it holds no value (unless we initialize it at the same time). It may hold a garbage value, but this has no meaning. In the case of pointers, a garbage value is the address of something in memory, but probably not of something that we want. So before a pointer is used, a specific address must be placed in it. In the PTRVAR program, `ptr` is first assigned the address of `var1` in the line

```
ptr = &var1;      <------  put address of var1 in ptr
```

Following this, the program prints out the value contained in `ptr`, which should be the same address printed for `&var1`. The same pointer variable `ptr` is then assigned the address of `var2`, and this value is printed out. Figure 10.3 shows the operation of the PTRVAR program. Here's the output of PTRVAR:

```
0x8f51fff4     <------  address of var1
0x8f51fff2     <------  address of var2

0x8f51fff4     <------  ptr set to address of var1
0x8f51fff2     <------  ptr set to address of var2
```

To summarize: A pointer can hold the address of any variable of the correct type; it's a receptacle awaiting an address. However, it must be given some value, or it will point to an address we don't want it to point to, such as into our program code or the operating system. Rogue pointer values can result in system crashes and are difficult to debug, since the compiler gives no warning. The moral: Make sure you give every pointer variable a valid address value before using it.

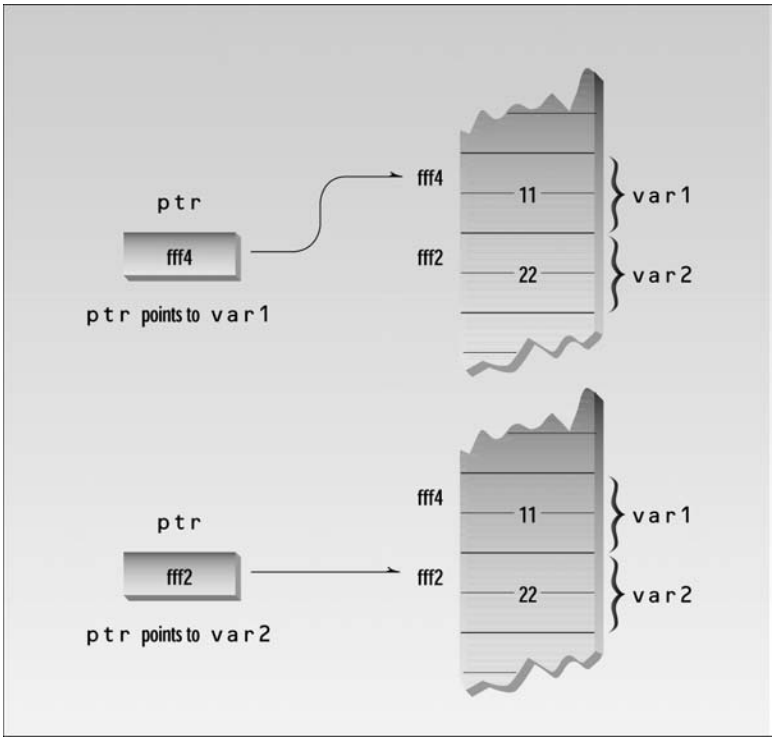

FIGURE 10.3
Changing values in `ptr`.

Accessing the Variable Pointed To

Suppose that we don't know the name of a variable but we do know its address. Can we access the contents of the variable? (It may seem like mismanagement to lose track of variable names, but we'll soon see that there are many variables whose names we don't know.)

There is a special syntax to access the value of a variable using its address instead of its name. Here's an example program, PTRACC, that shows how it's done:

```
// ptracc.cpp
// accessing the variable pointed to
#include <iostream>
using namespace std;

int main()
   {
   int var1 = 11;             //two integer variables
   int var2 = 22;
```

```
   int* ptr;                  //pointer to integers

   ptr = &var1;               //pointer points to var1
   cout << *ptr << endl;      //print contents of pointer (11)

   ptr = &var2;               //pointer points to var2
   cout << *ptr << endl;      //print contents of pointer (22)
   return 0;
   }
```

This program is very similar to PTRVAR, except that instead of printing the address values in `ptr`, we print the integer value stored at the address that's stored in `ptr`. Here's the output:

```
11
22
```

The expression that accesses the variables `var1` and `var2` is `*ptr`, which occurs in each of the two `cout` statements.

When an asterisk is used in front of a variable name, as it is in the `*ptr` expression, it is called the *dereference operator* (or sometimes the *indirection operator)*. It means *the value of the variable pointed to by*. Thus the expression `*ptr` represents the value of the variable pointed to by `ptr`. When `ptr` is set to the address of `var1`, the expression `*ptr` has the value 11, since `var1` is 11. When `ptr` is changed to the address of `var2`, the expression `*ptr` acquires the value 22, since `var2` is 22. Another name for the dereference operator is the *contents of* operator, which is another way to say the same thing. Figure 10.4 shows how this looks.

You can use a pointer not only to display a variable's value, but also to perform any operation you would perform on the variable directly. Here's a program, PTRTO, that uses a pointer to assign a value to a variable, and then to assign that value to another variable:

```
// ptrto.cpp
// other access using pointers
#include <iostream>
using namespace std;

int main()
   {
   int var1, var2;          //two integer variables
   int* ptr;                //pointer to integers

   ptr = &var1;             //set pointer to address of var1
   *ptr = 37;               //same as var1=37
   var2 = *ptr;             //same as var2=var1

   cout << var2 << endl;    //verify var2 is 37
   return 0;
   }
```

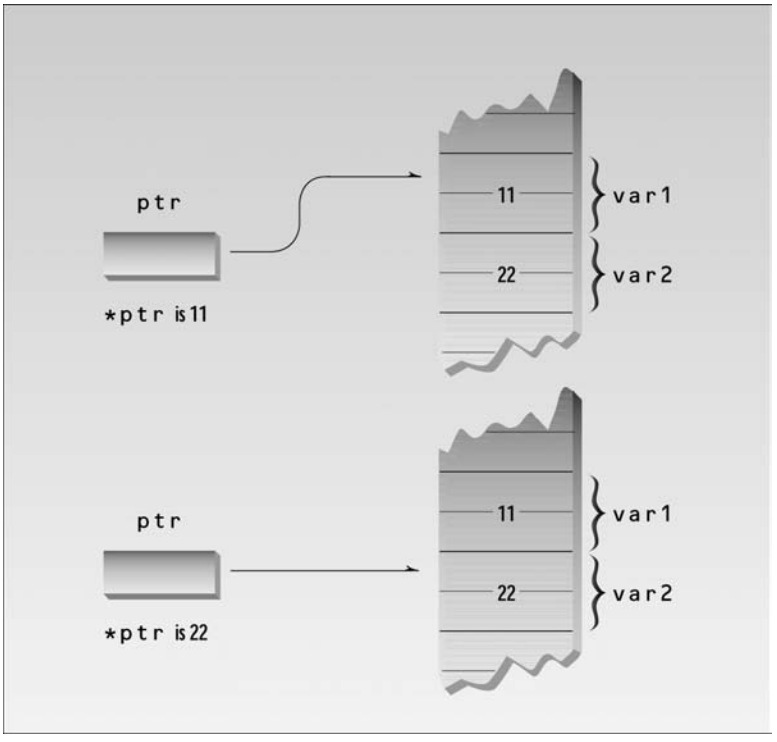

FIGURE 10.4
Access via pointer.

Remember that the asterisk used as the dereference operator has a different meaning than the asterisk used to declare pointer variables. The dereference operator *precedes* the variable and means *value of the variable pointed to by*. The asterisk used in a declaration means *pointer to*.

```
int* ptr;    //declaration: pointer to int
*ptr = 37;   //indirection: value of variable pointed to by ptr
```

Using the dereference operator to access the value stored in an address is called *indirect addressing*, or sometimes *dereferencing*, the pointer.

Here's a capsule summary of what we've learned so far:

```
int v;        //defines variable v of type int
int* p;       //defines p as a pointer to int
p = &v;       //assigns address of variable v to pointer p
v = 3;        //assigns 3 to v
*p = 3;       //also assigns 3 to v
```

The last two statements show the difference between normal or direct addressing, where we refer to a variable by name, and pointer or indirect addressing, where we refer to the same variable using its address.

In the example programs we've shown so far in this chapter, there's really no advantage to using the pointer expression to access variables, since we can access them directly. The value of pointers becomes evident when you can't access a variable directly, as we'll see later.

Pointer to `void`

Before we go on to see pointers at work, we should note one peculiarity of pointer data types. Ordinarily, the address that you put in a pointer must be the same type as the pointer. You can't assign the address of a `float` variable to a pointer to `int`, for example:

```
float flovar = 98.6;
int* ptrint = &flovar;  //ERROR: can't assign float* to int*
```

However, there is an exception to this. There is a sort of general-purpose pointer that can point to any data type. This is called a pointer to `void`, and is defined like this:

```
void* ptr;   //ptr can point to any data type
```

Such pointers have certain specialized uses, such as passing pointers to functions that operate independently of the data type pointed to.

The next example uses a pointer to `void` and also shows that, if you don't use `void`, you must be careful to assign pointers an address of the same type as the pointer. Here's the listing for PTRVOID:

```
// ptrvoid.cpp
// pointers to type void
#include <iostream>
using namespace std;

int main()
   {
   int intvar;               //integer variable
   float flovar;             //float variable

   int* ptrint;              //define pointer to int
   float* ptrflo;            //define pointer to float
   void* ptrvoid;            //define pointer to void

   ptrint = &intvar;         //ok, int* to int*
// ptrint = &flovar;         //error, float* to int*
// ptrflo = &intvar;         //error, int* to float*
   ptrflo = &flovar;         //ok, float* to float*
```

```
   ptrvoid = &intvar;         //ok, int* to void*
   ptrvoid = &flovar;         //ok, float* to void*
   return 0;
   }
```

You can assign the address of `intvar` to `ptrint` because they are both type `int*`, but you can't assign the address of `flovar` to `ptrint` because the first is type `float*` and the second is type `int*`. However, `ptrvoid` can be given any pointer value, such as `int*`, because it is a pointer to `void`.

If for some unusual reason you really need to assign one kind of pointer type to another, you can use the `reinterpret_cast`. For the lines commented out in PTRVOID, that would look like this:

```
ptrint = reinterpret_cast<int*>(flovar);
ptrflo = reinterpret_cast<float*>(intvar);
```

The use of `reinterpret_cast` in this way is not recommended, but occasionally it's the only way out of a difficult situation. Static casts won't work with pointers. Old-style C casts can be used, but are always a bad idea in C++. We'll see examples of `reinterpret_cast` in Chapter 12, "Streams and Files," where it's used to alter the way a data buffer is interpreted.

Pointers and Arrays

There is a close association between pointers and arrays. We saw in Chapter 7, "Arrays and Strings," how array elements are accessed. The following program, ARRNOTE, provides a review.

```
// arrnote.cpp
// array accessed with array notation
#include <iostream>
using namespace std;

int main()
   {                                       //array
   int intarray[5] = { 31, 54, 77, 52, 93 };

   for(int j=0; j<5; j++)                  //for each element,
      cout << intarray[j] << endl;         //print value
   return 0;
   }
```

The `cout` statement prints each array element in turn. For instance, when `j` is 3, the expression `intarray[j]` takes on the value `intarray[3]` and accesses the fourth array element, the integer 52. Here's the output of ARRNOTE:

```
31
54
77
52
93
```

Surprisingly, array elements can be accessed using pointer notation as well as array notation. The next example, PTRNOTE, is similar to ARRNOTE except that it uses pointer notation.

```
// ptrnote.cpp
// array accessed with pointer notation
#include <iostream>
using namespace std;

int main()
   {                                        //array
   int intarray[5] = { 31, 54, 77, 52, 93 };

   for(int j=0; j<5; j++)                   //for each element,
      cout << *(intarray+j) << endl;        //print value
   return 0;
   }
```

The expression `*(intarray+j)` in PTRNOTE has exactly the same effect as `intarray[j]` in ARRNOTE, and the output of the programs is identical. But how do we interpret the expression `*(intarray+j)`? Suppose `j` is 3, so the expression is equivalent to `*(intarray+3)`. We want this to represent the contents of the fourth element of the array (52). Remember that the name of an array is its address. The expression `intarray+j` is thus an address with something added to it. You might expect that `intarray+3` would cause 3 bytes to be added to `intarray`. But that doesn't produce the result we want: `intarray` is an array of integers, and 3 bytes into this array is the middle of the second element, which is not very useful. We want to obtain the fourth *integer* in the array, not the fourth byte, as shown in Figure 10.5. (This figure assumes 2-byte integers.)

The C++ compiler is smart enough to take the size of the data into account when it performs arithmetic on data addresses. It knows that `intarray` is an array of type `int` because it was declared that way. So when it sees the expression `intarray+3`, it interprets it as the address of the fourth *integer* in `intarray`, not the fourth byte.

But we want the *value* of this fourth array element, not the *address*. To take the value, we use the dereference operator (`*`). The resulting expression, when `j` is 3, is `*(intarray+3)`, which is the content of the fourth array element, or 52.

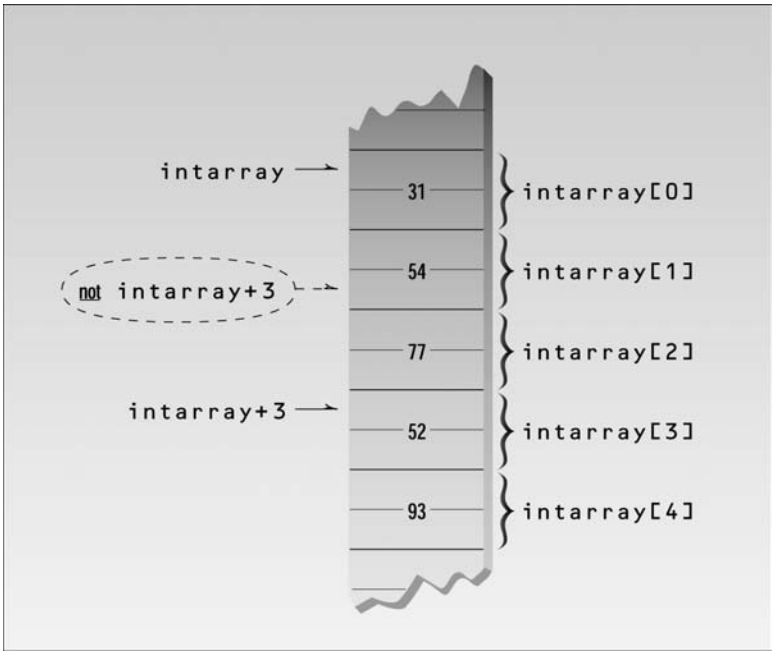

FIGURE 10.5
Counting by integers.

Now we see why a pointer declaration must include the type of the variable pointed to. The compiler needs to know whether a pointer is a pointer to `int` or a pointer to `double` so that it can perform the correct arithmetic to access elements of the array. It multiplies the index value by 2 in the case of type `int`, but by 8 in the case of `double`.

Pointer Constants and Pointer Variables

Suppose that, instead of adding `j` to `intarray` to step through the array addresses, you wanted to use the increment operator. Could you write `*(intarray++)`?

The answer is no, and the reason is that you can't increment a constant (or indeed change it in any way). The expression `intarray` is the address where the system has chosen to place your array, and it will stay at this address until the program terminates. `intarray` is a pointer constant. You can't say `intarray++` any more than you can say `7++`. (In a multitasking system, variable addresses may change during program execution. An active program may be swapped out to disk and then reloaded at a different memory location. However, this process is invisible to your program.)

But while you can't increment an address, you can increment a pointer that holds an address. The next example, PTRINC, shows how:

```
// ptrinc.cpp
// array accessed with pointer
#include <iostream>
using namespace std;

int main()
   {
   int intarray[] = { 31, 54, 77, 52, 93 }; //array
   int* ptrint;                             //pointer to int
   ptrint = intarray;                       //points to intarray

   for(int j=0; j<5; j++)                   //for each element,
      cout << *(ptrint++) << endl;          //print value
   return 0;
   }
```

Here we define a pointer to `int`—`ptrint`—and give it the value `intarray`, the address of the array. Now we can access the contents of the array elements with the expression

```
*(ptrint++)
```

The variable `ptrint` starts off with the same address value as `intarray`, thus allowing the first array element, `intarray[0]`, which has the value 31, to be accessed as before. But, because `ptrint` is a variable and not a constant, it can be incremented. After it is incremented, it points to the second array element, `intarray[1]`. The expression `*(ptrint++)` then represents the contents of the second array element, or 54. The loop causes the expression to access each array element in turn. The output of PTRINC is the same as that for PTRNOTE.

Pointers and Functions

In Chapter 5 we noted that there are three ways to pass arguments to a function: by value, by reference, and by pointer. If the function is intended to modify variables in the calling program, these variables cannot be passed by value, since the function obtains only a copy of the variable. However, either a reference argument or a pointer can be used in this situation.

Passing Simple Variables

We'll first review how arguments are passed by reference, and then compare this to passing pointer arguments. The PASSREF program shows passing by reference.

```
// passref.cpp
// arguments passed by reference
#include <iostream>
using namespace std;

int main()
   {
```

```
   void centimize(double&);    //prototype

   double var = 10.0;          //var has value of 10 inches
   cout << "var = " << var << " inches" << endl;

   centimize(var);             //change var to centimeters
   cout << "var = " << var << " centimeters" << endl;
   return 0;
   }
//-------------------------------------------------------------
void centimize(double& v)
   {
   v *= 2.54;                  //v is the same as var
   }
```

Here we want to convert a variable `var` in `main()` from inches to centimeters. We pass the variable by reference to the function `centimize()`. (Remember that the `&` following the data type `double` in the prototype for this function indicates that the argument is passed by reference.) The `centimize()` function multiplies the original variable by 2.54. Notice how the function refers to the variable. It simply uses the argument name `v`; `v` and `var` are different names for the same thing.

Once it has converted `var` to centimeters, `main()` displays the result. Here's the output of PASSREF:

```
var = 10 inches
var = 25.4 centimeters
```

The next example, PASSPTR, shows an equivalent situation when pointers are used:

```
// passptr.cpp
// arguments passed by pointer
#include <iostream>
using namespace std;

int main()
   {
   void centimize(double*);    //prototype

   double var = 10.0;          //var has value of 10 inches
   cout << "var = " << var << " inches" << endl;

   centimize(&var);            //change var to centimeters
   cout << "var = " << var << " centimeters" << endl;
   return 0;
   }
//-------------------------------------------------------------
void centimize(double* ptrd)
```

```
{
*ptrd *= 2.54;              //*ptrd is the same as var
}
```

The output of PASSPTR is the same as that of PASSREF.

The function `centimize()` is declared as taking an argument that is a pointer to `double`:

```
void centimize(double*)   // argument is pointer to double
```

When `main()` calls the function, it supplies the address of the variable as the argument:

```
centimize(&var);
```

Remember that this is not the variable itself, as it is in passing by reference, but the variable's address.

Because the `centimize()` function is passed an address, it must use the dereference operator, `*ptrd`, to access the value stored at this address:

```
*ptrd *= 2.54;  // multiply the contents of ptrd by 2.54
```

Of course this is the same as

```
*ptrd = *ptrd * 2.54;  // multiply the contents of ptrd by 2.54
```

where the standalone asterisk means multiplication. (This operator really gets around.)

Since `ptrd` contains the address of `var`, anything done to `*ptrd` is actually done to `var`. Figure 10.6 shows how changing `*ptrd` in the function changes `var` in the calling program.

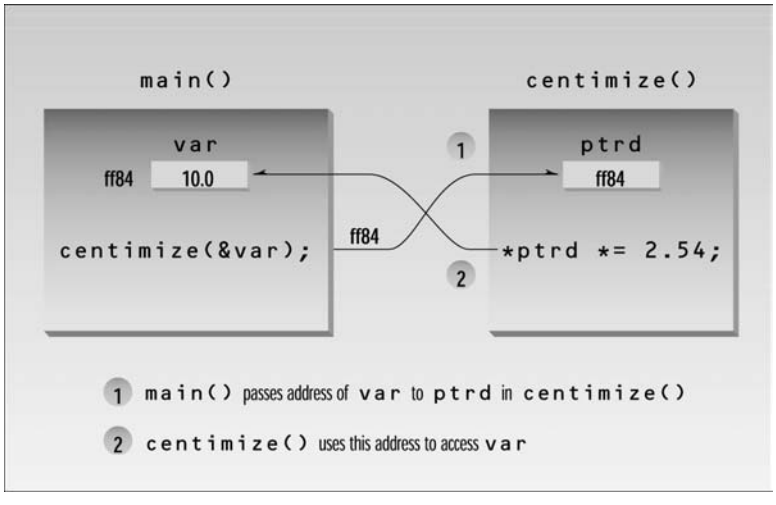

FIGURE 10.6
Pointer passed to function.

Passing a pointer as an argument to a function is in some ways similar to passing a reference. They both permit the variable in the calling program to be modified by the function. However, the mechanism is different. A reference is an alias for the original variable, while a pointer is the address of the variable.

Passing Arrays

We've seen numerous examples, starting in Chapter 7, of arrays passed as arguments to functions, and their elements being accessed by the function. Until this chapter, since we had not yet learned about pointers, this was done using array notation. However, it's more common to use pointer notation instead of array notation when arrays are passed to functions. The PASSARR program shows how this looks:

```
// passarr.cpp
// array passed by pointer
#include <iostream>
using namespace std;
const int MAX = 5;           //number of array elements

int main()
   {
   void centimize(double*);  //prototype

   double varray[MAX] = { 10.0, 43.1, 95.9, 59.7, 87.3 };

   centimize(varray);        //change elements of varray to cm

   for(int j=0; j<MAX; j++)  //display new array values
      cout << "varray[" << j << "]="
           << varray[j] << " centimeters" << endl;
   return 0;
   }
//-------------------------------------------------------------
void centimize(double* ptrd)
   {
   for(int j=0; j<MAX; j++)
      *ptrd++ *= 2.54;       //ptrd points to elements of varray
   }
```

The prototype for the function is the same as in PASSPTR; the function's single argument is a pointer to `double`. In array notation this is written as

```
void centimize(double[]);
```

That is, `double*` is equivalent here to `double[]`, although the pointer syntax is more commonly used.

Since the name of an array is the array's address, there is no need for the address operator `&` when the function is called:

```
centimize(varray);  // pass array address
```

In `centimize()`, this array address is placed in the variable `ptrd`. To point to each element of the array in turn, we need only increment `ptrd`:

```
*ptrd++ *= 2.54;
```

Figure 10.7 shows how the array is accessed. Here's the output of PASSARR:

```
varray[0]=25.4 centimeters
varray[1]=109.474 centimeters
varray[2]=243.586 centimeters
varray[3]=151.638 centimeters
varray[4]=221.742 centimeters
```

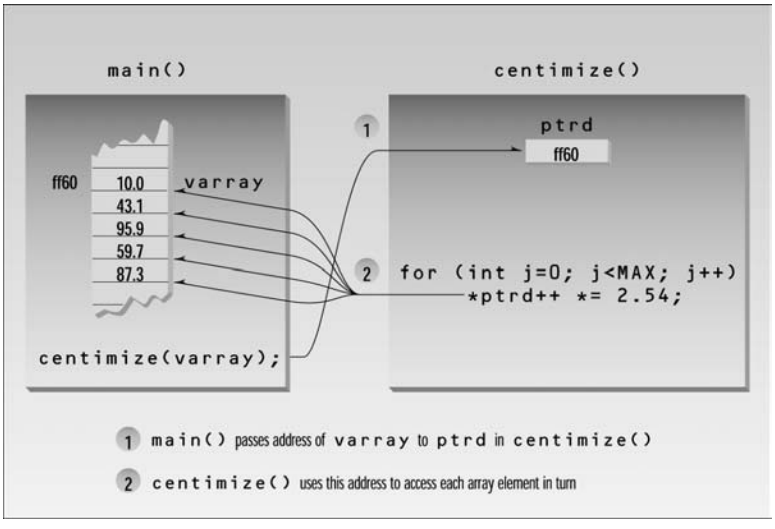

FIGURE 10.7
Accessing an array from a function.

Here's a syntax question: How do we know that the expression `*ptrd++` increments the pointer and not the pointer contents? In other words, does the compiler interpret it as `*(ptrd++)`, which is what we want, or as `(*ptrd)++`? It turns out that `*` (when used as the dereference operator) and `++` have the same precedence. However, operators of the same precedence are distinguished in a second way: by *associativity*. Associativity is concerned with whether the compiler performs operations starting with an operator on the right or an operator on the left.

If a group of operators has right associativity, the compiler performs the operation on the right side of the expression first, then works its way to the left. The unary operators such as * and ++ have right associativity, so the expression is interpreted as *(ptrd++), which increments the pointer, not what it points to. That is, the pointer is incremented first and the dereference operator is applied to the resulting address.

Sorting Array Elements

As a further example of using pointers to access array elements, let's see how to sort the contents of an array. We'll use two program examples—the first to lay the groundwork, and the second, an expansion of the first, to demonstrate the sorting process.

Ordering with Pointers

The first program is similar to the REFORDER program in Chapter 6, "Objects and Classes," except that it uses pointers instead of references. It orders two numbers passed to it as arguments, exchanging them if the second is smaller than the first. Here's the listing for PTRORDER:

```
// ptrorder.cpp
// orders two arguments using pointers
#include <iostream>
using namespace std;

int main()
   {
   void order(int*, int*);         //prototype

   int n1=99, n2=11;               //one pair ordered, one not
   int n3=22, n4=88;

   order(&n1, &n2);                //order each pair of numbers
   order(&n3, &n4);

   cout << "n1=" << n1 << endl;    //print out all numbers
   cout << "n2=" << n2 << endl;
   cout << "n3=" << n3 << endl;
   cout << "n4=" << n4 << endl;
   return 0;
   }
//-------------------------------------------------------------
void order(int* numb1, int* numb2) //orders two numbers
   {
   if(*numb1 > *numb2)             //if 1st larger than 2nd,
      {
      int temp = *numb1;           //swap them
      *numb1 = *numb2;
```

```
      *numb2 = temp;
      }
   }
```

The function `order()` works the same as it did in REFORDER, except that it is passed the addresses of the numbers to be ordered, and it accesses the numbers using pointers. That is, `*numb1` accesses the number in `main()` passed as the first argument, and `*numb2` accesses the second.

Here's the output from PTRORDER:

```
n1=11   <------- this and
n2=99   <------- this are swapped, since they weren't in order
n3=22   <------- this and
n4=88   <------- this are not swapped, since they were in order
```

We'll use the `order()` function from PTRORDER in our next example program, PTRSORT, which sorts an array of integers.

```
// ptrsort.cpp
// sorts an array using pointers
#include <iostream>
using namespace std;

int main()
   {
   void bsort(int*, int);       //prototype
   const int N = 10;            //array size
                                //test array
   int arr[N] = { 37, 84, 62, 91, 11, 65, 57, 28, 19, 49 };

   bsort(arr, N);               //sort the array

   for(int j=0; j<N; j++)       //print out sorted array
      cout << arr[j] << " ";
   cout << endl;
   return 0;
   }
//--------------------------------------------------------------
void bsort(int* ptr, int n)
   {
   void order(int*, int*);      //prototype
   int j, k;                    //indexes to array

   for(j=0; j<n-1; j++)         //outer loop
      for(k=j+1; k<n; k++)      //inner loop starts at outer
         order(ptr+j, ptr+k);   //order the pointer contents
   }
```

```
//--------------------------------------------------------------
void order(int* numb1, int* numb2)  //orders two numbers
   {
   if(*numb1 > *numb2)          //if 1st larger than 2nd,
      {
      int temp = *numb1;        //swap them
      *numb1 = *numb2;
      *numb2 = temp;
      }
   }
```

The array `arr` of integers in `main()` is initialized to unsorted values. The address of the array, and the number of elements, are passed to the `bsort()` function. This sorts the array, and the sorted values are then printed. Here's the output of the PTRSORT:

```
11 19 28 37 49 57 62 65 84 91
```

The Bubble Sort

The `bsort()` function sorts the array using a variation of the bubble sort. This is a simple (although notoriously slow) approach to sorting. Here's how it works, assuming we want to arrange the numbers in the array in ascending order. First the first element of the array (`arr[0]`) is compared in turn with each of the other elements (starting with the second). If it's greater than any of them, the two are swapped. When this is done we know that at least the first element is in order; it's now the smallest element. Next the second element is compared in turn with all the other elements, starting with the third, and again swapped if it's bigger. When we're done we know that the second element has the second-smallest value. This process is continued for all the elements until the next-to-the-last, at which time the array is assumed to be ordered. Figure 10.8 shows the bubble sort in action (with fewer items than in PTRSORT).

In PTRSORT, the number in the first position, 37, is compared with each element in turn, and swapped with 11. The number in the second position, which starts off as 84, is compared with each element. It's swapped with 62; then 62 (which is now in the second position) is swapped with 37, 37 is swapped with 28, and 28 is swapped with 19. The number in the third position, which is 84 again, is swapped with 62, 62 is swapped with 57, 57 with 37, and 37 with 28. The process continues until the array is sorted.

The `bsort()` function in PTRSORT consists of two nested loops, each of which controls a pointer. The outer loop uses the loop variable `j`, and the inner one uses `k`. The expressions `ptr+j` and `ptr+k` point to various elements of the array, as determined by the loop variables. The expression `ptr+j` moves down the array, starting at the first element (the top) and stepping down integer by integer until one short of the last element (the bottom). For each position taken by `ptr+j` in the outer loop, the expression `ptr+k` in the inner loop starts pointing one below `ptr+j` and moves down to the bottom of the array. Each time through the inner loop, the

elements pointed to by `ptr+j` and `ptr+k` are compared, using the `order()` function, and if the first is greater than the second, they're swapped. Figure 10.9 shows this process.

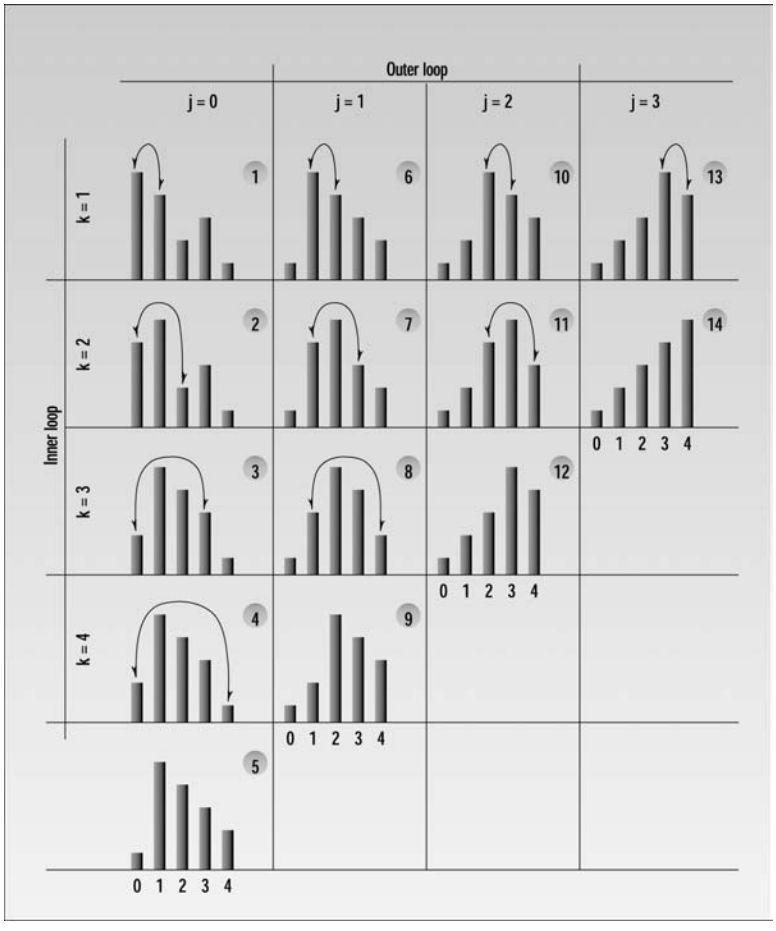

FIGURE 10.8
Operation of the bubble sort.

The PTRSORT example begins to reveal the power of pointers. They provide a consistent and efficient way to operate on array elements and other variables whose names aren't known to a particular function.

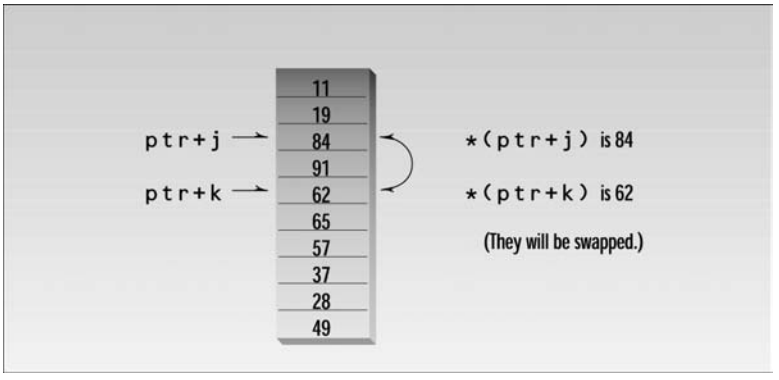

FIGURE 10.9
Operation of PTRSORT.

Pointers and C-Type Strings

As we noted in Chapter 7, C-type strings are simply arrays of type `char`. Thus pointer notation can be applied to the characters in strings, just as it can to the elements of any array.

Pointers to String Constants

Here's an example, TWOSTR, in which two strings are defined, one using array notation as we've seen in previous examples, and one using pointer notation:

```
// twostr.cpp
// strings defined using array and pointer notation
#include <iostream>
using namespace std;

int main()
   {
   char str1[] = "Defined as an array";
   char* str2 = "Defined as a pointer";

   cout << str1 << endl;    // display both strings
   cout << str2 << endl;

// str1++;                  // can't do this; str1 is a constant
   str2++;                  // this is OK, str2 is a pointer

   cout << str2 << endl;    // now str2 starts "efined..."
   return 0;
   }
```

In many ways these two types of definition are equivalent. You can print out both strings as the example shows, use them as function arguments, and so on. But there is a subtle difference: `str1` is an address—that is, a pointer constant—while `str2` is a pointer variable. So `str2` can be changed, while `str1` cannot, as shown in the program. Figure 10.10 shows how these two kinds of strings look in memory.

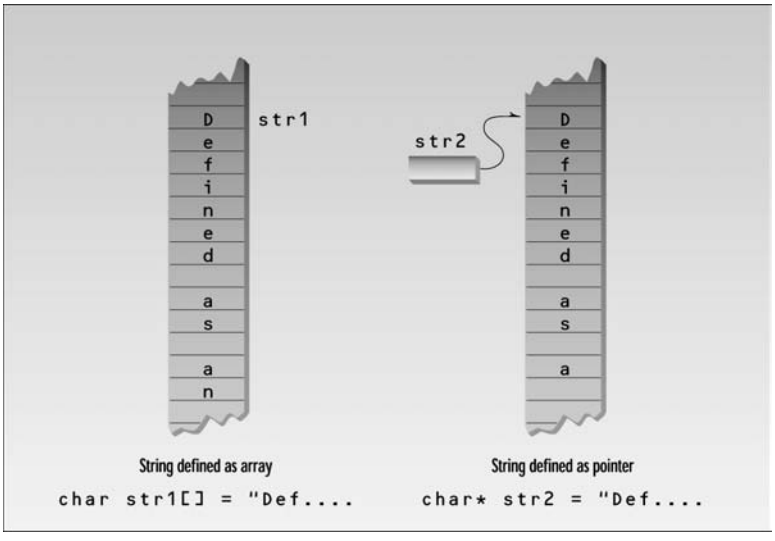

FIGURE 10.10
Strings as arrays and pointers.

We can increment `str2`, since it is a pointer, but once we do, it no longer points to the first character in the string. Here's the output of TWOSTR:

```
Defined as an array
Defined as a pointer
efined as a pointer     <------- following str2++ ('D' is gone)
```

A string defined as a pointer is considerably more flexible than one defined as an array. The following examples will make use of this flexibility.

Strings as Function Arguments

Here's an example that shows a string used as a function argument. The function simply prints the string, by accessing each character in turn. Here's the listing for PTRSTR:

```
// ptrstr.cpp
// displays a string with pointer notation
#include <iostream>
using namespace std;
```

```
int main()
   {
   void dispstr(char*);     //prototype
   char str[] = "Idle people have the least leisure.";

   dispstr(str);            //display the string
   return 0;
   }
//-------------------------------------------------------------
void dispstr(char* ps)
   {
   while( *ps )             //until null character,
      cout << *ps++;        //print characters
   cout << endl;
   }
```

The array address `str` is used as the argument in the call to function `dispstr()`. This address is a constant, but since it is passed by value, a copy of it is created in `dispstr()`. This copy is a pointer, `ps`. A pointer can be changed, so the function increments `ps` to display the string. The expression `*ps++` returns the successive characters of the string. The loop cycles until it finds the null character (`'\0'`) at the end of the string. Since this has the value 0, which represents *false*, the `while` loop terminates at this point.

Copying a String Using Pointers

We've seen examples of pointers used to obtain values from an array. Pointers can also be used to insert values into an array. The next example, COPYSTR, demonstrates a function that copies one string to another:

```
// copystr.cpp
// copies one string to another with pointers
#include <iostream>
using namespace std;

int main()
   {
   void copystr(char*, const char*);  //prototype
   char* str1 = "Self-conquest is the greatest victory.";
   char str2[80];               //empty string

   copystr(str2, str1);         //copy str1 to str2
   cout << str2 << endl;        //display str2
   return 0;
   }
```

```
//-------------------------------------------------------------
void copystr(char* dest, const char* src)
   {
   while( *src )                //until null character,
      *dest++ = *src++;         //copy chars from src to dest
   *dest = '\0';                //terminate dest
   }
```

Here the `main()` part of the program calls the function `copystr()` to copy `str1` to `str2`. In this function the expression

```
*dest++ = *src++;
```

takes the value at the address pointed to by `src` and places it in the address pointed to by `dest`. Both pointers are then incremented, so the next time through the loop the next character will be transferred. The loop terminates when a null character is found in `src`; at this point a null is inserted in `dest` and the function returns. Figure 10.11 shows how the pointers move through the strings.

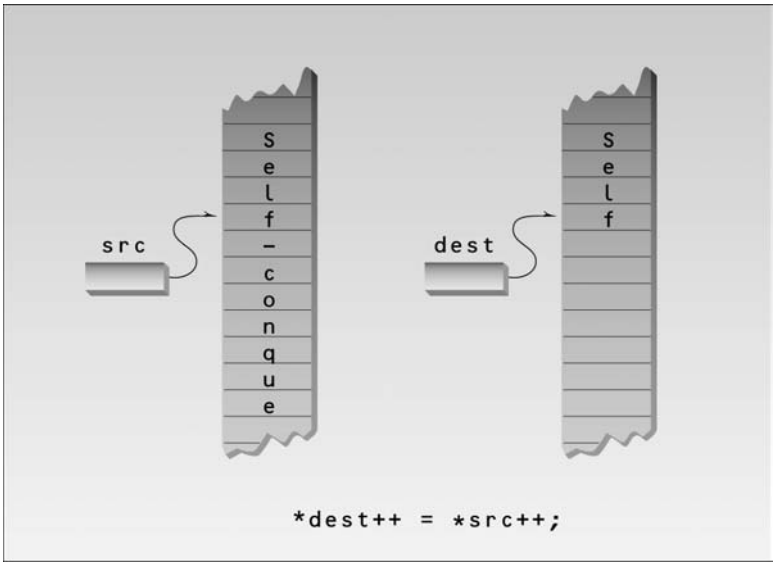

FIGURE 10.11
Operation of COPYSTR.

Library String Functions

Many of the library functions we have already used for strings have string arguments that are specified using pointer notation. As an example you can look at the description of `strcpy()` in your compiler's documentation (or in the STRING.H header file). This function copies one string to another; we can compare it with our homemade `copystr()` function in the COPYSTR example. Here's the syntax for the `strcpy()` library function:

```
char* strcpy(char* dest, const char* src);
```

This function takes two arguments of type `char*`. (The next section, "The `const` Modifier and Pointers," explains the meaning of `const` in this context.) The `strcpy()` function also returns a pointer to `char`; this is the address of the `dest` string. In other respects, this function works very much like our homemade `copystr()` function.

The `const` Modifier and Pointers

The use of the `const` modifier with pointer declarations can be confusing, because it can mean one of two things, depending on where it's placed. The following statements show the two possibilities:

```
const int* cptrInt;  //cptrInt is a pointer to constant int
int* const ptrcInt;  //ptrcInt is a constant pointer to int
```

Following the first declaration, you cannot change the value of whatever `cptrInt` points to, although you can change `cptrInt` itself. Following the second declaration, you can change what `ptrcInt` points to, but you cannot change the value of `ptrcInt` itself. You can remember the difference by reading from right to left, as indicated in the comments. You can use `const` in both positions to make the pointer and what it points to constant.

In the declaration of `strcpy()` just shown, the argument `const char* src` specifies that the characters pointed to by `src` cannot be changed by `strcpy()`. It does not imply that the `src` pointer itself cannot be modified. To do that the argument declaration would need to be `char* const src`.

Arrays of Pointers to Strings

Just as there are arrays of variables of type `int` or type `float`, there can also be arrays of pointers. A common use for this construction is an array of pointers to strings.

In Chapter 7 the STRARAY program demonstrated an array of `char*` strings. As we noted, there is a disadvantage to using an array of strings, in that the subarrays that hold the strings must all be the same length, so space is wasted when strings are shorter than the length of the subarrays (see Figure 7.10 in Chapter 7).

Let's see how to use pointers to solve this problem. We will modify STRARAY to create an array of pointers to strings, rather than an array of strings. Here's the listing for PTRTOSTR:

```
// ptrtostr.cpp
// an array of pointers to strings
#include <iostream>
using namespace std;
const int DAYS = 7;             //number of pointers in array

int main()
   {                            //array of pointers to char
   char* arrptrs[DAYS] = { "Sunday", "Monday", "Tuesday",
               "Wednesday", "Thursday",
               "Friday", "Saturday"  };

   for(int j=0; j<DAYS; j++)    //display every string
      cout << arrptrs[j] << endl;
   return 0;
   }
```

The output of this program is the same as that for STRARAY:

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

When strings are not part of an array, C++ places them contiguously in memory, so there is no wasted space. However, to find the strings, there must be an array that holds pointers to them. A string is itself an array of type `char`, so an array of pointers to strings is an array of pointers to `char`. That is the meaning of the definition of `arrptrs` in PTRTOSTR. Now recall that a string is always represented by a single address: the address of the first character in the string. It is these addresses that are stored in the array. Figure 10.12 shows how this looks.

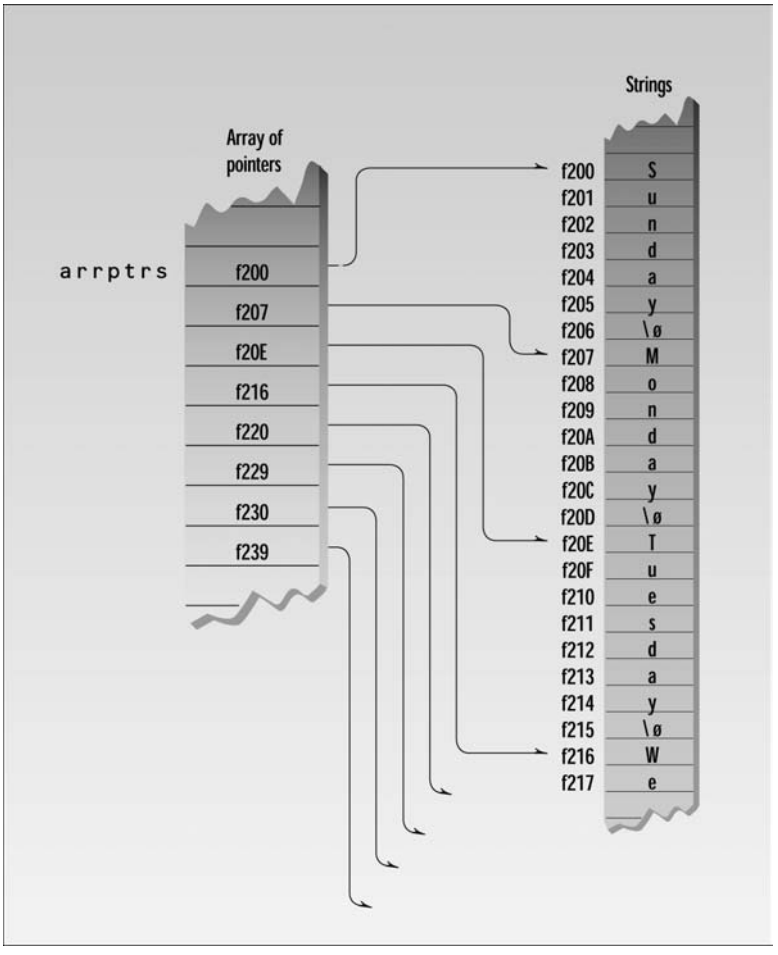

FIGURE 10.12
Array of pointers and strings.

Memory Management: `new` and `delete`

We've seen many examples where arrays are used to set aside memory. The statement

```
int arr1[100];
```

reserves memory for 100 integers. Arrays are a useful approach to data storage, but they have a serious drawback: We must know at the time we write the program how big the array will be. We can't wait until the program is running to specify the array size. The following approach won't work:

```
cin >> size;     // get size from user
int arr[size];   // error; array size must be a constant
```

The compiler requires the array size to be a constant.

But in many situations we don't know how much memory we need until runtime. We might want to store a string that was typed in by the user, for example. In this situation we can define an array sized to hold the largest string we expect, but this wastes memory. (As we'll learn in Chapter 15, "The Standard Template Library," you can also use a vector, which is a sort of expandable array.)

The new Operator

C++ provides a different approach to obtaining blocks of memory: the new operator. This versatile operator obtains memory from the operating system and returns a pointer to its starting point. The NEWINTRO example shows how new is used:

```
// newintro.cpp
// introduces operator new
#include <iostream>
#include <cstring>            //for strlen
using namespace std;

int main()
   {
   char* str = "Idle hands are the devil's workshop.";
   int len = strlen(str);     //get length of str

   char* ptr;                 //make a pointer to char
   ptr = new char[len+1];     //set aside memory: string + '\0'

   strcpy(ptr, str);          //copy str to new memory area ptr

   cout << "ptr=" << ptr << endl;  //show that ptr is now in str

   delete[] ptr;              //release ptr's memory
   return 0;
   }
```

The expression

```
ptr = new char[len+1];
```

returns a pointer to a section of memory just large enough to hold the string str, whose length len we found with the strlen() library function, plus an extra byte for the null character '\0' at the end of the string. Figure 10.13 shows the syntax of a statement using the new operator. Remember to use brackets around the size; the compiler won't object if you mistakenly use parentheses, but the results will be incorrect.

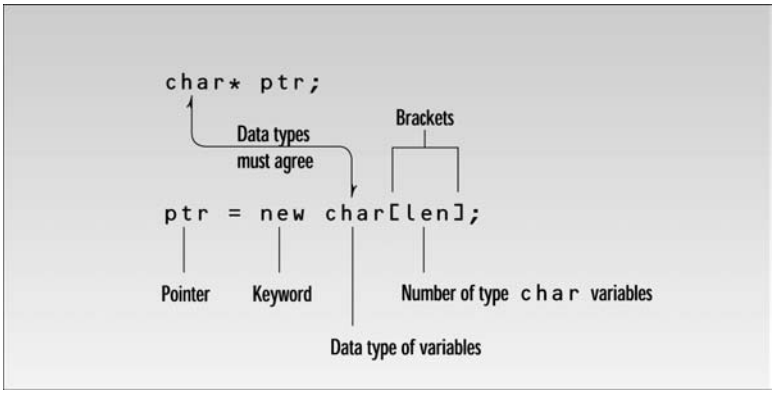

FIGURE 10.13
Syntax of the `new` *operator.*

Figure 10.14 shows the memory obtained by `new` and the pointer to it.

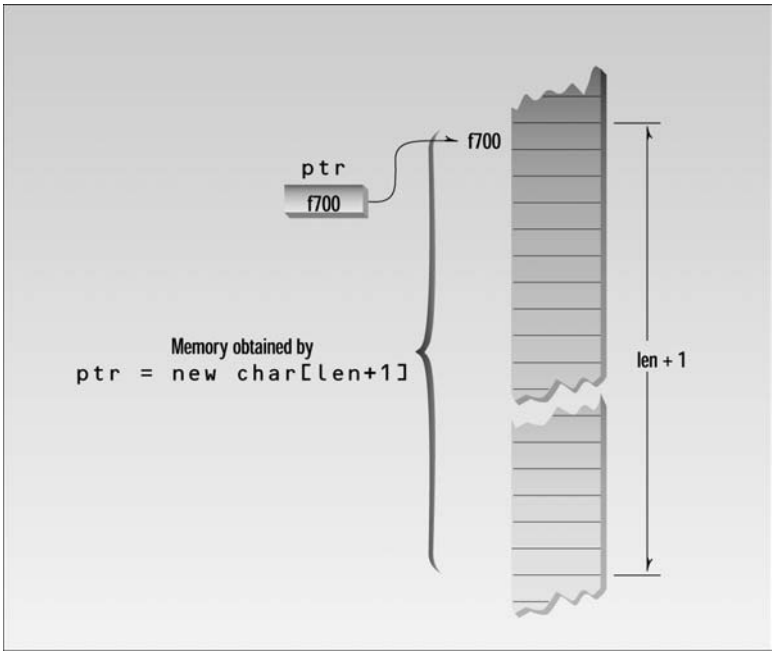

FIGURE 10.14
Memory obtained by the `new` *operator.*

In NEWINTRO we use `strcpy()` to copy string `str` to the newly created memory area pointed to by `ptr`. Since we made this area equal in size to the length of `str`, the string fits exactly. The output of NEWINTRO is

```
ptr=Idle hands are the devil's workshop.
```

C programmers will recognize that `new` plays a role similar to the `malloc()` family of library functions. The `new` approach is superior in that it returns a pointer to the appropriate data type, while `malloc()`'s pointer must be cast to the appropriate type. There are other advantages as well.

C programmers may wonder whether there is a C++ equivalent to `realloc()` for changing the size of memory that has already been reallocated. Sorry, there's no `renew` in C++. You'll need to fall back on the ploy of creating a larger (or smaller) space with `new`, and copying your data from the old area to the new one.

The `delete` Operator

If your program reserves many chunks of memory using `new`, eventually all the available memory will be reserved and the system will crash. To ensure safe and efficient use of memory, the `new` operator is matched by a corresponding `delete` operator that returns memory to the operating system. In NEWINTRO the statement

```
delete[] ptr;
```

returns to the system whatever memory was pointed to by `ptr`.

Actually, there is no need for this operator in NEWINTRO, since memory is automatically returned when the program terminates. However, suppose you use `new` in a function. If the function uses a local variable as a pointer to this memory, the pointer will be destroyed when the function terminates, but the memory will be left as an orphan, taking up space that is inaccessible to the rest of the program. Thus it is always good practice, and often essential, to delete memory when you're through with it.

Deleting the memory doesn't delete the pointer that points to it (`str` in NEWINTRO), and doesn't change the address value in the pointer. However, this address is no longer valid; the memory it points to may be changed to something entirely different. Be careful that you don't use pointers to memory that has been deleted.

The brackets following `delete` indicate that we're deleting an array. If you create a single object with `new`, you don't need the brackets when you delete it.

```
ptr = new SomeClass;  // allocate a single object
. . .
delete ptr;           // no brackets following delete
```

However, don't forget the brackets when deleting arrays of objects. Using them ensures that all the members of the array are deleted, and that the destructor is called for each one.

A String Class Using new

The new operator often appears in constructors. As an example, we'll modify the String class, last seen in examples such as STRPLUS in Chapter 8, "Operator Overloading." You may recall that a potential defect of that class was that all String objects occupied the same fixed amount of memory. A string shorter than this fixed length wasted memory, and a longer string—if one were mistakenly generated—could crash the system by extending beyond the end of the array. Our next example uses new to obtain exactly the right amount of memory. Here's the listing for NEWSTR:

```
// newstr.cpp
// using new to get memory for strings
#include <iostream>
#include <cstring>       //for strcpy(), etc
using namespace std;
////////////////////////////////////////////////////////////
class String             //user-defined string type
   {
   private:
      char* str;                    //pointer to string
   public:
      String(char* s)               //constructor, one arg
         {
         int length = strlen(s);    //length of string argument
         str = new char[length+1];  //get memory
         strcpy(str, s);            //copy argument to it
         }
      ~String()                     //destructor
         {
         cout << "Deleting str.\n";
         delete[] str;              //release memory
         }
      void display()                //display the String
         {
         cout << str << endl;
         }
   };
////////////////////////////////////////////////////////////
int main()
   {                                //uses 1-arg constructor
   String s1 = "Who knows nothing doubts nothing.";
```

```
   cout << "s1=";                  //display string
   s1.display();
   return 0;
   }
```

The output from this program is

```
s1=Who knows nothing doubts nothing.
Deleting str.
```

The `String` class has only one data item: a pointer to `char` called `str`. This pointer will point to the string held by the `String` object. There is no array within the object to hold the string. The string is stored elsewhere; only the pointer to it is a member of `String`.

Constructor in NEWSTR

The constructor in this example takes a normal `char*` string as its argument. It obtains space in memory for this string with `new`; `str` points to the newly obtained memory. The constructor then uses `strcpy()` to copy the string into this new space.

Destructor in NEWSTR

We haven't seen many destructors in our examples so far, but now that we're allocating memory with `new`, destructors become important. If we allocate memory when we create an object, it's reasonable to deallocate the memory when the object is no longer needed. As you may recall from Chapter 6, a destructor is a routine that is called automatically when an object is destroyed. The destructor in NEWSTR looks like this:

```
~String()
   {
   cout << "Deleting str.";
   delete[] str;
   }
```

This destructor gives back to the system the memory obtained when the object was created. You can tell from the program's output that the destructor executed at the end of the program. Objects (like other variables) are typically destroyed when the function in which they were defined terminates. This destructor ensures that memory obtained by the `String` object will be returned to the system, and not left in limbo, when the object is destroyed.

We should note a potential glitch in using destructors as shown in NEWSTR. If you copy one `String` object to another, say with a statement like `s2 = s1`, you're really only copying the pointer to the actual (`char*`) string. Both objects now point to the same string in memory. But if you now delete one string, the destructor will delete the `char*` string, leaving the other object with an invalid pointer. This can be subtle, because objects can be deleted in non-obvious ways, such as when a function in which a local object has been created returns. In Chapter 11 we'll see how to make a smarter destructor that counts how many `String` objects are pointing to a string.

Pointers to Objects

Pointers can point to objects as well as to simple data types and arrays. We've seen many examples of objects defined and given a name, in statements like

```
Distance dist;
```

where an object called `dist` is defined to be of the `Distance` class.

Sometimes, however, we don't know, at the time that we write the program, how many objects we want to create. When this is the case we can use `new` to create objects while the program is running. As we've seen, `new` returns a pointer to an unnamed object. Let's look at a short example program, ENGLPTR, that compares the two approaches to creating objects.

```
// englptr.cpp
// accessing member functions by pointer
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance              //English Distance class
   {
   private:
      int feet;
      float inches;
   public:
      void getdist()        //get length from user
         {
         cout << "\nEnter feet: ";  cin >> feet;
         cout << "Enter inches: ";  cin >> inches;
         }
      void showdist()       //display distance
         { cout << feet << "\'-" << inches << '\"'; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Distance dist;           //define a named Distance object
   dist.getdist();          //access object members
   dist.showdist();         //   with dot operator

   Distance* distptr;       //pointer to Distance
   distptr = new Distance;  //points to new Distance object
   distptr->getdist();      //access object members
   distptr->showdist();     //   with -> operator
   cout << endl;
   return 0;
   }
```

This program uses a variation of the English `Distance` class seen in previous chapters. The `main()` function defines `dist`, uses the `Distance` member function `getdist()` to get a distance from the user, and then uses `showdist()` to display it.

Referring to Members

ENGLPTR then creates another object of type `Distance` using the `new` operator, and returns a pointer to it called `distptr`.

The question is, how do we refer to the member functions in the object pointed to by `distptr`? You might guess that we would use the dot (`.`) membership-access operator, as in

```
distptr.getdist();   // won't work; distptr is not a variable
```

but this won't work. The dot operator requires the identifier on its left to be a variable. Since `distptr` is a pointer to a variable, we need another syntax. One approach is to *dereference* (get the contents of the variable pointed to by) the pointer:

```
(*distptr).getdist();  // ok but inelegant
```

However, this is slightly cumbersome because of the parentheses. (The parentheses are necessary because the dot operator (`.`) has higher precedence than the dereference operator (`*`). An equivalent but more concise approach is furnished by the membership-access operator, which consists of a hyphen and a greater-than sign:

```
distptr->getdist();   // better approach
```

As you can see in ENGLPTR, the `->` operator works with pointers to objects in just the same way that the `.` operator works with objects. Here's the output of the program:

```
Enter feet: 10 <------- this object uses the dot operator
Enter inches: 6.25
10'-6.25"

Enter feet: 6    <------- this object uses the -> operator
Enter inches: 4.75
6'-4.75"
```

Another Approach to `new`

You may come across another—less common—approach to using `new` to obtain memory for objects.

Since `new` can return a pointer to an area of memory that holds an object, we should be able to refer to the original object by dereferencing the pointer. The ENGLREF example shows how this is done.

```
// englref.cpp
// dereferencing the pointer returned by new
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Distance                    // English Distance class
   {
   private:
      int feet;
      float inches;
   public:
      void getdist()              // get length from user
         {
         cout << "\nEnter feet: ";  cin >> feet;
         cout << "Enter inches: ";  cin >> inches;
         }
      void showdist()             // display distance
         { cout << feet << "\'-" << inches << '\"'; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Distance& dist = *(new Distance);  // create Distance object
                                      // alias is "dist"
   dist.getdist();                    // access object members
   dist.showdist();                   //    with dot operator
   cout << endl;
   return 0;
   }
```

The expression

```
new Distance
```

returns a pointer to a memory area large enough for a `Distance` object, so we can refer to the original object as

```
*(new Distance)
```

This is the object pointed to by the pointer. Using a reference, we define `dist` to be an object of type `Distance`, and we set it equal to `*(new Distance)`. Now we can refer to members of `dist` using the dot membership operator, rather than `->`.

This approach is less common than using pointers to objects obtained with `new`, or simply declaring an object, but it works in a similar way.

An Array of Pointers to Objects

A common programming construction is an array of pointers to objects. This arrangement allows easy access to a group of objects, and is more flexible than placing the objects themselves in an array. (For instance, in the PERSORT example in this chapter we'll see how a group of objects can be sorted by sorting an array of pointers to them, rather than sorting the objects themselves.)

Our next example, PTROBJS, creates an array of pointers to the `person` class. Here's the listing:

```
// ptrobjs.cpp
// array of pointers to objects
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class person                        //class of persons
   {
   protected:
      char name[40];                //person's name
   public:
      void setName()                //set the name
         {
         cout << "Enter name: ";
         cin >> name;
         }
      void printName()              //get the name
         {
         cout << "\n   Name is: " << name;
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   person* persPtr[100];     //array of pointers to persons
   int n = 0;                //number of persons in array
   char choice;

   do                                   //put persons in array
      {
      persPtr[n] = new person;          //make new object
      persPtr[n]->setName();            //set person's name
      n++;                              //count new person
      cout << "Enter another (y/n)? "; //enter another
      cin >> choice;                    //person?
      }
   while( choice=='y' );                //quit on 'n'
```

```
   for(int j=0; j<n; j++)              //print names of
      {                                //all persons
      cout << "\nPerson number " << j+1;
      persPtr[j]->printName();
      }
   cout << endl;
   return 0;
   }  //end main()
```

The class `person` has a single data item, `name`, which holds a string representing a person's name. Two member functions, `setName()` and `printName()`, allow the name to be set and displayed.

Program Operation

The `main()` function defines an array, `persPtr`, of 100 pointers to type `person`. In a `do` loop it then asks the user to enter a name. With this name it creates a `person` object using `new`, and stores a pointer to this object in the array `persPtr`. To demonstrate how easy it is to access the objects using the pointers, it then prints out the `name` data for each `person` object.

Here's a sample interaction with the program:

```
Enter name: Stroustrup     <------- user enters names
Enter another (y/n)? y
Enter name: Ritchie
Enter another (y/n)? y
Enter name: Kernighan
Enter another (y/n)? n
Person number 1     <------- program displays all names stored
   Name is: Stroustrup
Person number 2
   Name is: Ritchie
Person number 3
   Name is: Kernighan
```

Accessing Member Functions

We need to access the member functions `setName()` and `printName()` in the `person` objects pointed to by the pointers in the array `persPtr`. Each of the elements of the array `persPtr` is specified in array notation to be `persPtr[j]` (or equivalently by pointer notation to be `*(persPtr+j)`). The elements are pointers to objects of type `person`. To access a member of an object using a pointer, we use the `->` operator. Putting this all together, we have the following syntax for `getname()`:

```
persPtr[j]->getName()
```

This executes the `getname()` function in the `person` object pointed to by element `j` of the `persPtr` array. (It's a good thing we don't have to program using English syntax.)

A Linked List Example

Our next example shows a simple linked list. What is a linked list? It's another way to store data. You've seen numerous examples of data stored in arrays. Another data structure is an array of pointers to data members, as in the PTRTOSTRS and PTROBJS examples. Both the array and the array of pointers suffer from the necessity to declare a fixed-size array before running the program.

A Chain of Pointers

The linked list provides a more flexible storage system in that it doesn't use arrays at all. Instead, space for each data item is obtained as needed with `new`, and each item is connected, or *linked*, to the next data item using a pointer. The individual items don't need to be located contiguously in memory the way array elements are; they can be scattered anywhere.

In our example the entire linked list is an object of class `linklist`. The individual data items, or links, are represented by structures of type `link`. Each such structure contains an integer—representing the object's single data item—and a pointer to the next link. The list itself stores a pointer to the link at the head of the list. This arrangement is shown in Figure 10.15.

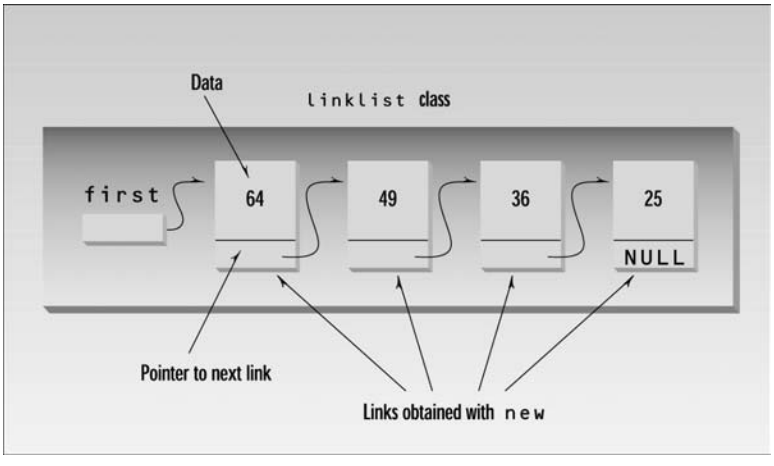

FIGURE 10.15
A linked list.

Here's the listing for LINKLIST:

```
// linklist.cpp
// linked list
#include <iostream>
using namespace std;
```

```
////////////////////////////////////////////////////////////////
struct link                             //one element of list
   {
   int data;                            //data item
   link* next;                          //pointer to next link
   };
////////////////////////////////////////////////////////////////
class linklist                          //a list of links
   {
   private:
      link* first;                      //pointer to first link
   public:
      linklist()                        //no-argument constructor
         { first = NULL; }              //no first link
      void additem(int d);              //add data item (one link)
      void display();                   //display all links
   };
//-------------------------------------------------------------
void linklist::additem(int d)           //add data item
   {
   link* newlink = new link;            //make a new link
   newlink->data = d;                   //give it data
   newlink->next = first;               //it points to next link
   first = newlink;                     //now first points to this
   }
//-------------------------------------------------------------
void linklist::display()                //display all links
   {
   link* current = first;               //set ptr to first link
   while( current != NULL )             //quit on last link
      {
      cout << current->data << endl;    //print data
      current = current->next;          //move to next link
      }
   }
////////////////////////////////////////////////////////////////
int main()
   {
   linklist li;       //make linked list

   li.additem(25);    //add four items to list
   li.additem(36);
   li.additem(49);
   li.additem(64);
```

```
li.display();      //display entire list
return 0;
}
```

The `linklist` class has only one member data item: the pointer to the start of the list. When the list is first created, the constructor initializes this pointer, which is called `first`, to `NULL`. The `NULL` constant is defined to be 0. This value serves as a signal that a pointer does not hold a valid address. In our program a link whose next member has a value of `NULL` is assumed to be at the end of the list.

Adding an Item to the List

The `additem()` member function adds an item to the linked list. A new link is inserted at the beginning of the list. (We could write the `additem()` function to insert items at the end of the list, but that is a little more complex to program.) Let's look at the steps involved in inserting a new link.

First, a new structure of type `link` is created by the line

```
link* newlink = new link;
```

This creates memory for the new `link` structure with `new` and saves the pointer to it in the `newlink` variable.

Next we want to set the members of the newly created structure to appropriate values. A structure is similar to a class in that, when it is referred to by pointer rather than by name, its members are accessed using the `->` member-access operator. The following two lines set the `data` variable to the value passed as an argument to `additem()`, and the `next` pointer to point to whatever address was in `first`, which holds the pointer to the start of the list.

```
newlink->data = d;
newlink->next = first;
```

Finally, we want the `first` variable to point to the new link:

```
first = newlink;
```

The effect is to uncouple the connection between `first` and the old first link, insert the new link, and move the old first link into the second position. Figure 10.16 shows this process.

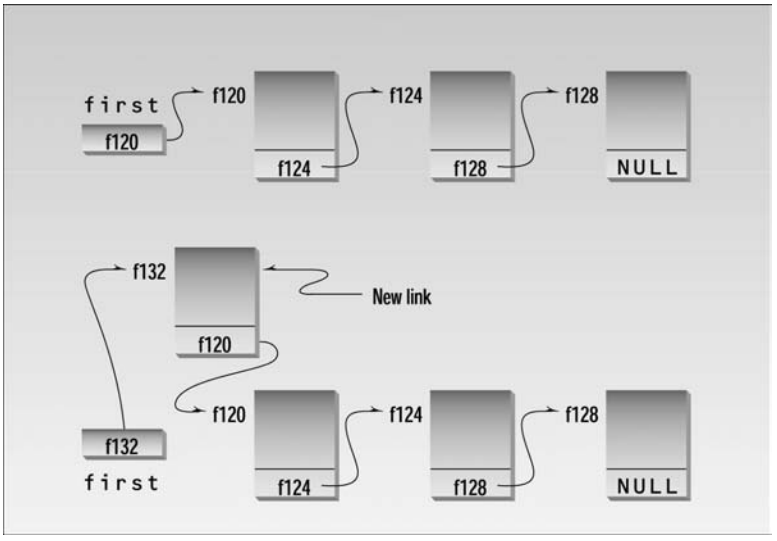

FIGURE 10.16
Adding to a linked list.

Displaying the List Contents

Once the list is created it's easy to step through all the members, displaying them (or performing other operations). All we need to do is follow from one `next` pointer to another until we find a `next` that is `NULL`, signaling the end of the list. In the function `display()`, the line

```
cout << endl << current->data;
```

prints the value of the data, and

```
current = current->next;
```

moves us along from one link to another, until

```
current != NULL
```

in the `while` expression becomes false. Here's the output of LINKLIST:

```
64
49
36
25
```

Linked lists are perhaps the most commonly used data storage arrangements after arrays. As we noted, they avoid the wasting of memory space engendered by arrays. The disadvantage is that finding a particular item on a linked list requires following the chain of links from the head of the list until the desired link is reached. This can be time-consuming. An array element, on the other hand, can be accessed quickly, provided its index is known in advance. We'll have more to say about linked lists and other data-storage techniques in Chapter 15, "The Standard Template Library."

Self-Containing Classes

We should note a possible pitfall in the use of self-referential classes and structures. The `link` structure in LINKLIST contained a pointer to the same kind of structure. You can do the same with classes:

```
class sampleclass
   {
   sampleclass* ptr;  // this is fine
   };
```

However, while a class can contain a pointer to an object of its own type, it cannot contain an *object* of its own type:

```
class sampleclass
   {
   sampleclass obj;  // can't do this
   };
```

This is true of structures as well as classes.

Augmenting LINKLIST

The general organization of LINKLIST can serve for a more complex situation than that shown. There could be more data in each link. Instead of an integer, a link could hold a number of data items or it could hold a pointer to a structure or object.

Additional member functions could perform such activities as adding and removing links from an arbitrary part of the chain. Another important member function is a destructor. As we mentioned, it's important to delete blocks of memory that are no longer in use. A destructor that performs this task would be a highly desirable addition to the `linklist` class. It could go through the list using `delete` to free the memory occupied by each link.

Pointers to Pointers

Our next example demonstrates an array of pointers to objects, and shows how to sort these pointers based on data in the object. This involves the idea of pointers to pointers, and may help demonstrate why people lose sleep over pointers.

The idea in the next program is to create an array of pointers to objects of the `person` class. This is similar to the PTROBJS example, but we go further and add variations of the `order()` and `bsort()` functions from the PTRSORT example so that we can sort a group of `person` objects based on the alphabetical order of their names. Here's the listing for PERSORT:

```
// persort.cpp
// sorts person objects using array of pointers
#include <iostream>
#include <string>                 //for string class
using namespace std;
////////////////////////////////////////////////////////////////
class person                      //class of persons
   {
   protected:
      string name;                //person's name
   public:
      void setName()              //set the name
         { cout << "Enter name: "; cin >> name; }
      void printName()            //display the name
         { cout << endl << name; }
      string getName()            //return the name
         { return name; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   void bsort(person**, int);     //prototype
   person* persPtr[100];          //array of pointers to persons
   int n = 0;                     //number of persons in array
   char choice;                   //input char

   do {                           //put persons in array
      persPtr[n] = new person;    //make new object
      persPtr[n]->setName();      //set person's name
      n++;                        //count new person
      cout << "Enter another (y/n)? "; //enter another
      cin >> choice;              //   person?
      }
```

```
   while( choice=='y' );          //quit on 'n'

   cout << "\nUnsorted list:";
   for(int j=0; j<n; j++)         //print unsorted list
      { persPtr[j]->printName(); }

   bsort(persPtr, n);             //sort pointers

   cout << "\nSorted list:";
   for(j=0; j<n; j++)             //print sorted list
      { persPtr[j]->printName(); }
   cout << endl;
   return 0;
   }  //end main()
//--------------------------------------------------------------
void bsort(person** pp, int n)    //sort pointers to persons
   {
   void order(person**, person**);  //prototype
   int j, k;                      //indexes to array

   for(j=0; j<n-1; j++)           //outer loop
      for(k=j+1; k<n; k++)        //inner loop starts at outer
         order(pp+j, pp+k);       //order the pointer contents
   }
//--------------------------------------------------------------
void order(person** pp1, person** pp2)  //orders two pointers
   {                              //if 1st larger than 2nd,
   if( (*pp1)->getName() > (*pp2)->getName() )
      {
      person* tempptr = *pp1;     //swap the pointers
      *pp1 = *pp2;
      *pp2 = tempptr;
      }
   }
```

When the program is first executed it asks for a name. When the user gives it one, it creates an object of type `person` and sets the `name` data in this object to the name entered by the user. The program also stores a pointer to the object in the `persPtr` array.

When the user types `n` to indicate that no more names will be entered, the program calls the `bsort()` function to sort the `person` objects based on their `name` member variables. Here's some sample interaction with the program:

```
Enter name: Washington
Enter another (y/n)? y
Enter name: Adams
```

```
Enter another (y/n)? y
Enter name: Jefferson
Enter another (y/n)? y
Enter name: Madison
Enter another (y/n)? n
Unsorted list:
Washington
Adams
Jefferson
Madison

Sorted list:
Adams
Jefferson
Madison
Washington
```

Sorting Pointers

Actually, when we sort `person` objects, we don't move the objects themselves; we move the pointers to the objects. This eliminates the need to shuffle the objects around in memory, which can be very time-consuming if the objects are large. It could also, if we wanted, allow us to keep multiple sorts—one by name and another by phone number, for example—in memory at the same time without storing the objects multiple times. The process is shown in Figure 10.17.

To facilitate the sorting activity, we've added a `getName()` member function to the `person` class so we can access the names from `order()` to decide when to swap pointers.

The `person**` Data Type

You will notice that the first argument to the `bsort()` function, and both arguments to `order()`, have the type `person**`. What do the two asterisks mean? These arguments are used to pass the address of the array `persPtr`, or—in the case of `order()`—the addresses of elements of the array. If this were an array of type `person`, the address of the array would be type `person*`. However, the array is of type *pointers* to `person`, or `person*`, so its address is type `person**`. The address of a pointer is a pointer to a pointer. Figure 10.18 shows how this looks.

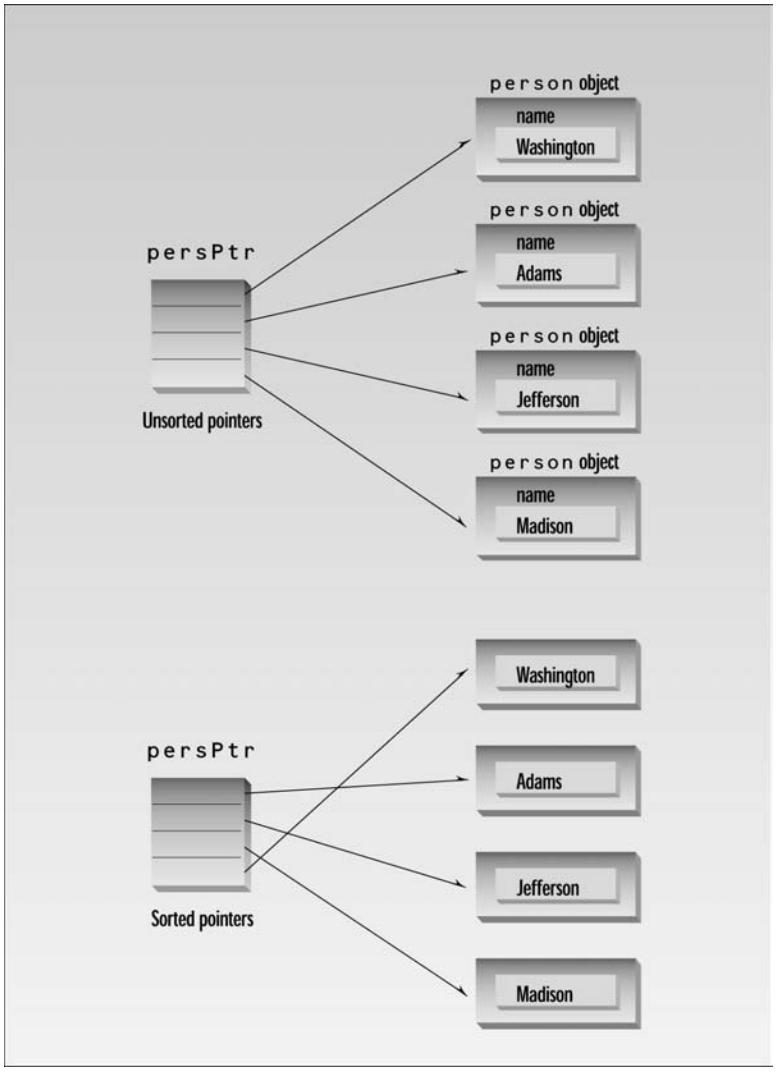

FIGURE 10.17
Sorting an array of pointers.

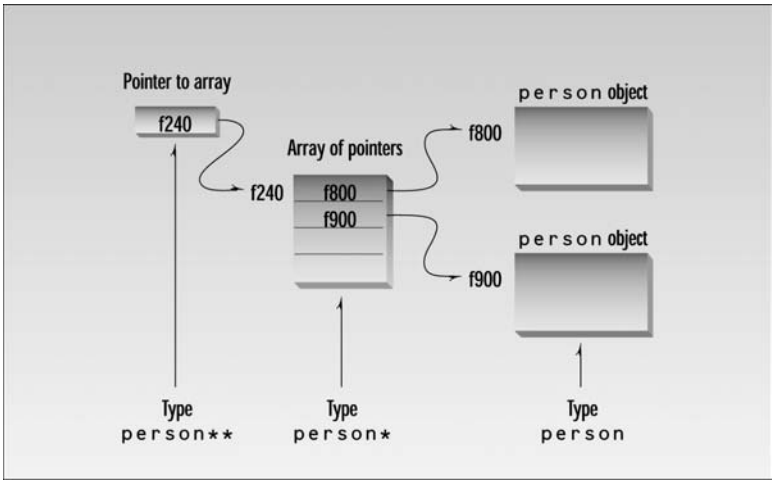

FIGURE 10.18
Pointer to an array of pointers.

Compare this program with PTRSORT, which sorted an array of type `int`. You'll find that the data types passed to functions in PERSORT all have one more asterisk than they did in PTRSORT, because the array is an array of pointers.

Since the `persPtr` array contains pointers, the construction

```
persPtr[j]->printName()
```

executes the `printName()` function in the object pointed to by element `j` of `persPtr`.

Comparing Strings

The `order()` function in PERSORT has been modified to order two strings lexigraphically—that is, by putting them in alphabetical order. To do this it compares the strings using the C++ library function `strcmp()`. This function takes the two strings `s1` and `s2` as arguments, as in `strcmp(s1, s2)`, and returns one of the following values.

Value	*Condition*
<0	`s1` comes before `s2`
0	`s1` is the same as `s2`
>0	`s1` comes after `s2`

The strings are accessed using the syntax

```
(*pp1)->getname()
```

The argument pp1 is a pointer to a pointer, and we want the name pointed to by the pointer it points to. The member-access operator -> dereferences one level, but we need to dereference another level, hence the asterisk preceding pp1.

Just as there can be pointers to pointers, there can be pointers to pointers to pointers, and so on. Fortunately such complexities are seldom encountered.

A Parsing Example

Programmers are frequently faced with the problem of unravelling or *parsing* a string of symbols. Examples are commands typed by a user at the keyboard, sentences in natural languages (such as English), statements in a programming language, and algebraic expressions. Now that we've learned about pointers and strings, we can handle this sort of problem.

Our next (somewhat longer) example will show how to parse arithmetic expressions like

```
6/3+2*3-1
```

The user enters the expression, the program works its way through it, character by character, figures out what it means in arithmetic terms, and displays the resulting value (7 in the example). Our expressions will use the four arithmetic operators: +, -, *, and /. We'll simplify the numbers we use to make the programming easier by restricting them to a single digit. Also, we won't allow parentheses.

This program makes use of our old friend the Stack class (see the STAKARAY program in Chapter 7). We've modified this class so that it stores data of type char. We use the stack to store both numbers and operators (both as characters). The stack is a useful storage mechanism because, when parsing expressions, we frequently need to access the last item stored, and a stack is a last-in-first-out (LIFO) container.

Besides the Stack class, we'll use a class called express (short for *expression*), representing an entire arithmetic expression. Member functions for this class allow us to initialize an object with an expression in the form of a string (entered by the user), parse the expression, and return the resulting arithmetic value.

Parsing Arithmetic Expressions

Here's how we parse an arithmetic expression. We start at the left, and look at each character in turn. It can be either a *number* (always a single digit—a character between 0 and 9), or an *operator* (the characters +, -, *, and /).

If the character is a number, we always push it onto the stack. We also push the first operator we encounter. The trick is how we handle subsequent operators. Note that we can't execute the current operator, because we haven't yet read the number that follows it. Finding an operator is

merely the signal that we can execute the previous operator, which is stored on the stack. That is, if the sequence `2+3` is on the stack, we wait until we find another operator before carrying out the addition.

Thus whenever we find that the current character is an operator (except the first), we pop the previous number (`3` in the preceding example) and the previous operator (`+`) off the stack, placing them in the variables `lastval` and `lastop`. Finally we pop the first number (`2`) and carry out the arithmetic operation on the two numbers (obtaining `5`). Can we always execute the previous operator? No. Remember that `*` and `/` have a higher precedence than `+` and `-`. In the expression `3+4/2`, we can't execute the `+` until we've done the division. So when we get to the `/` in this expression, we must put the `2` and the `+` back on the stack until we've carried out the division.

On the other hand, if the current operator is a `+` or `-`, we know we can always execute the previous operator. That is, when we see the `+` in the expression `4-5+6`, we know it's all right to execute the `-`, and when we see the `-` in `6/2-3` , we know it's okay to do the division. Table 10.1 shows the four possibilities.

TABLE 10.1 Operators and Parsing Actions

Previous Operator	*Current Operator*	*Example*	*Action*
`+` or `-`	`*` or `/`	`3+4/`	Push previous operator and previous number (`+`, `4`)
`*` or `/`	`*` or `/`	`9/3*`	Execute previous operator, push result (`3`)
`+` or `-`	`+` or `-`	`6+3+`	Execute previous operator, push result (`9`)
`*` or `/`	`+` or `-`	`8/2-`	Execute previous operator, push result (`4`)

The `parse()` member function carries out this process of going through the input expression and performing those operations it can. However, there is more work to do. The stack still contains either a single number or several sequences of number-operator-number. Working down through the stack, we can execute these sequences. Finally, a single number is left on the stack; this is the value of the original expression. The `solve()` member function carries out this task, working its way down through the stack until only a single number is left. In general, `parse()` puts things on the stack, and `solve()` takes them off.

The PARSE Program

Some typical interaction with PARSE might look like this:

```
Enter an arithmetic expression
of the form 2+3*4/3-2.
No number may have more than one digit.
Don't use any spaces or parentheses.
Expression: 9+6/3

The numerical value is: 11
Do another (Enter y or n)?
```

Note that it's all right if the *results* of arithmetic operations contain more than one digit. They are limited only by the numerical size of type `char`, from –128 to +127. Only the input string is limited to numbers from 0 to 9.

Here's the listing for the program:

```
// parse.cpp
// evaluates arithmetic expressions composed of 1-digit numbers
#include <iostream>
#include <cstring>                  //for strlen(), etc
using namespace std;
const int LEN = 80;    //length of expressions, in characters
const int MAX = 40;    //size of stack
////////////////////////////////////////////////////////////////
class Stack
   {
   private:
      char st[MAX];                 //stack: array of chars
      int top;                      //number of top of stack
   public:
      Stack()                       //constructor
         { top = 0; }
      void push(char var)           //put char on stack
         { st[++top] = var; }
      char pop()                    //take char off stack
         { return st[top--]; }
      int gettop()                  //get top of stack
         { return top; }
   };
////////////////////////////////////////////////////////////////
class express                       //expression class
   {
   private:
      Stack s;                      //stack for analysis
      char* pStr;                   //pointer to input string
      int len;                      //length of input string
```

```
   public:
      express(char* ptr)            //constructor
         {
         pStr = ptr;                //set pointer to string
         len = strlen(pStr);        //set length
         }
      void parse();                 //parse the input string
      int solve();                  //evaluate the stack
   };
//--------------------------------------------------------------
void express::parse()               //add items to stack
   {
   char ch;                         //char from input string
   char lastval;                    //last value
   char lastop;                     //last operator

   for(int j=0; j<len; j++)         //for each input character
      {
      ch = pStr[j];                 //get next character

      if(ch>='0' && ch<='9')        //if it's a digit,
         s.push(ch-'0');            //save numerical value
                                    //if it's operator
      else if(ch=='+' || ch=='-' || ch=='*' || ch=='/')
         {
         if(s.gettop()==1)          //if it's first operator
            s.push(ch);             //put on stack
         else                       //not first operator
            {
            lastval = s.pop();      //get previous digit
            lastop = s.pop();       //get previous operator
            //if this is * or / AND last operator was + or -
            if( (ch=='*' || ch=='/') &&
                (lastop=='+' || lastop=='-') )
               {
               s.push(lastop);      //restore last two pops
               s.push(lastval);
               }
            else                    //in all other cases
               {
               switch(lastop)       //do last operation
                  {                 //push result on stack
                  case '+': s.push(s.pop() + lastval); break;
                  case '-': s.push(s.pop() - lastval); break;
                  case '*': s.push(s.pop() * lastval); break;
                  case '/': s.push(s.pop() / lastval); break;
                  default:  cout << "\nUnknown oper"; exit(1);
```

```
                }  //end switch
             }  //end else, in all other cases
          s.push(ch);              //put current op on stack
          }  //end else, not first operator
       }  //end else if, it's an operator
    else                           //not a known character
       { cout << "\nUnknown input character"; exit(1); }
    }  //end for
 }  //end parse()
//--------------------------------------------------------------
int express::solve()               //remove items from stack
 {
 char lastval;                     //previous value

 while(s.gettop() > 1)
    {
    lastval = s.pop();             //get previous value
    switch( s.pop() )              //get previous operator
       {                           //do operation, push answer
       case '+': s.push(s.pop() + lastval); break;
       case '-': s.push(s.pop() - lastval); break;
       case '*': s.push(s.pop() * lastval); break;
       case '/': s.push(s.pop() / lastval); break;
       default:  cout << "\nUnknown operator"; exit(1);
       }  //end switch
    }  //end while
 return int( s.pop() );            //last item on stack is ans
 }  //end solve()
////////////////////////////////////////////////////////////////
int main()
 {
 char ans;                         //'y' or 'n'
 char string[LEN];                 //input string from user

 cout << "\nEnter an arithmetic expression"
         "\nof the form 2+3*4/3-2."
         "\nNo number may have more than one digit."
         "\nDon't use any spaces or parentheses.";
 do {
    cout << "\nEnter expresssion: ";
    cin >> string;                          //input from user
    express* eptr = new express(string);    //make expression
    eptr->parse();                          //parse it
    cout << "\nThe numerical value is: "
         << eptr->solve();                  //solve it
    delete eptr;                            //delete expression
```

```
    cout << "\nDo another (Enter y or n)? ";
    cin >> ans;
    } while(ans == 'y');
return 0;
}
```

This is a longish program, but it shows how a previously designed class, `Stack`, can come in handy in a new situation; it demonstrates the use of pointers in a variety of ways; and it shows how useful it can be to treat a string as an array of characters.

Simulation: A Horse Race

As our final example in this chapter we'll show a horse-racing game. In this game a number of horses appear on the screen, and, starting from the left, race to a finish line on the right. This program will demonstrate pointers in a new situation, and also a little bit about object-oriented design.

Each horse's speed is determined randomly, so there is no way to figure out in advance which one will win. The program uses console graphics, so the horses are easily, although somewhat crudely, displayed. You'll need to compile the program with the MSOFTCON.H or BORLACON.H header file (depending on your compiler), and the MSOFTCON.CPP or BORLACON.CPP source file. (See Appendix E, "Console Graphics Lite," for more information.)

When our program, HORSE, is started, it asks the user to supply the race's distance and the number of horses that will run in it. The classic unit of distance for horse racing (at least in English-speaking countries) is the *furlong*, which is 1/8 of a mile. Typical races are 6, 8, 10, or 12 furlongs. You can enter from 1 to 7 horses. The program draws vertical lines corresponding to each furlong, along with start and finish lines. Each horse is represented by a rectangle with a number in the middle. Figure 10.19 shows the screen with a race in progress.

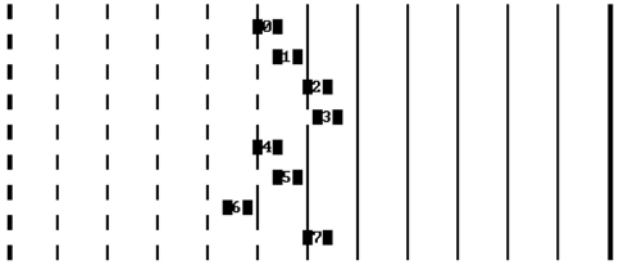

FIGURE 10.19
Output of the HORSE program.

Designing the Horse Race

How do we approach an OOP design for our horse race? Our first question might be, is there a group of similar entities that we're trying to model? The answer is yes, the horses. So it seems reasonable to make each horse an object. There will be a class called `horse`, which will contain data specific to each horse, such as its number and the distance it has run so far (which is used to display the horse in the correct screen position).

However, there is also data that applies to the entire race track, rather than to individual horses. This includes the track length, the elapsed time in minutes and seconds (0:00 at the start of the race), and the total number of horses. It makes sense then to have a `track` object, which will be a single member of the `track` class. You can think of other real-world objects associated with horse racing, such as riders and saddles, but they aren't relevant to this program.

Are there other ways to design the program? For example, what about using inheritance to make the horses descendants of the track? This doesn't make much sense, because the horses aren't a "kind of" race track; they're a completely different thing. Another option is to make the track data into static data of the `horse` class. However, it's generally better to make each different kind of thing in the problem domain (the real world) a separate class in the program. One advantage of this is that it's easier to use the classes in other contexts, such as using the track to race cars instead of horses.

How will the `horse` objects and the `track` object communicate? (Or in UML terms, what will their association consist of?) An array of pointers to `horse` objects can be a member of the `track` class, so the track can access the horses through these pointers. The track will create the horses when it's created. As it does so, it will pass a pointer to itself to each horse, so the horse can access the track.

Here's the listing for HORSE:

```
 // horse.cpp
// models a horse race
#include "msoftcon.h"               //for console graphics
#include <iostream>                 //for I/O
#include <cstdlib>                  //for random()
#include <ctime>                    //for time()
using namespace std;
const int CPF = 5;                  //columns per furlong
const int maxHorses = 7;            //maximum number of horses
class track;                        //for forward references
////////////////////////////////////////////////////////////////
class horse
   {
   private:
      const track* ptrTrack;        //pointer to track
      const int horse_number;       //this horse's number
```

```
      float finish_time;            //this horse's finish time
      float distance_run;           //distance run so far
   public:                          //create the horse
      horse(const int n, const track* ptrT) :
               horse_number(n), ptrTrack(ptrT),
               distance_run(0.0)    //haven't moved yet
          {  }
      ~horse()                      //destroy the horse
         { /*empty*/ }              //display the horse
      void display_horse(const float elapsed_time);
   };  //end class horse
////////////////////////////////////////////////////////////////
class track
   {
   private:
      horse* hArray[maxHorses];     //array of ptrs-to-horses
      int total_horses;             //total number of horses
      int horse_count;              //horses created so far
      const float track_length;     //track length in furlongs
      float elapsed_time;           //time since start of race

   public:
      track(float lenT, int nH);    //2-arg constructor
      ~track();                     //destructor
      void display_track();         //display track
      void run();                   //run the race
      float get_track_len() const;  //return total track length
   };  //end class track
//----------------------------------------------------------------
void horse::display_horse(float elapsed_time) //for each horse
   {                                //display horse & number
   set_cursor_pos( 1 + int(distance_run * CPF),
           2 + horse_number*2 );
                                    //horse 0 is blue
   set_color(static_cast<color>(cBLUE+horse_number));
                                    //draw horse
   char horse_char = '0' + static_cast<char>(horse_number);
   putch(' '); putch('\xDB'); putch(horse_char); putch('\xDB');
                                    //until finish,
   if( distance_run < ptrTrack->get_track_len() + 1.0 / CPF )
      {
      if( rand() % 3 )              //skip about 1 of 3 ticks
         distance_run += 0.2F;      //advance 0.2 furlongs
      finish_time = elapsed_time;   //update finish time
      }
```

```
   else
      {                            //display finish time
      int mins = int(finish_time)/60;
      int secs = int(finish_time) - mins*60;
      cout << " Time=" << mins << ":" << secs;
      }
   }  //end display_horse()
//--------------------------------------------------------------
track::track(float lenT, int nH) :  //track constructor
                     track_length(lenT), total_horses(nH),
                     horse_count(0), elapsed_time(0.0)
        {
        init_graphics();           //start graphics
        total_horses =             //not more than 7 horses
         (total_horses > maxHorses) ? maxHorses : total_horses;
        for(int j=0; j<total_horses; j++)   //make each horse
           hArray[j] = new horse(horse_count++, this);

        time_t aTime;              //initialize random numbers
        srand( static_cast<unsigned>(time(&aTime)) );
        display_track();
        }  //end track constructor
//--------------------------------------------------------------
track::~track()                    //track destructor
   {
   for(int j=0; j<total_horses; j++) //delete each horse
      delete hArray[j];
   }
//--------------------------------------------------------------
void track::display_track()
   {
   clear_screen();                 //clear screen
                                   //display track
   for(int f=0; f<=track_length; f++)    //for each furlong
      for(int r=1; r<=total_horses*2 + 1; r++) //and screen row
         {
         set_cursor_pos(f*CPF + 5, r);
         if(f==0 || f==track_length)
            cout << '\xDE';        //draw start or finish line
         else
            cout << '\xB3';        //draw furlong marker
         }
   }  //end display_track()
//--------------------------------------------------------------
void track::run()
```

```
   {
   while( !kbhit() )
      {
      elapsed_time += 1.75;         //update time
                                    //update each horse
      for(int j=0; j<total_horses; j++)
         hArray[j]->display_horse(elapsed_time);
      wait(500);
      }
   getch();                         //eat the keystroke
   cout << endl;
   }
//--------------------------------------------------------------
float track::get_track_len() const
   { return track_length; }
////////////////////////////////////////////////////////////////
int main()
   {
   float length;
   int total;
                                    //get data from user
   cout << "\nEnter track length (furlongs; 1 to 12): ";
   cin >> length;
   cout << "\nEnter number of horses (1 to 7): ";
   cin >> total;
   track theTrack(length, total);   //create the track
   theTrack.run();                  //run the race
   return 0;
   }  //end main()
```

Keeping Time

Simulation programs usually involve an activity taking place over a period of time. To model the passage of time, such programs typically energize themselves at fixed intervals. In the HORSE program, the `main()` program calls the track's `run()` function. This function makes a series of calls within a `while` loop, one for each horse, to a function `display_horse()`. This function redraws each horse in its new position. The `while` loop then pauses 500 milliseconds, using the console graphics `wait()` function. Then it does the same thing again, until the race is over or the user presses a key.

Deleting an Array of Pointers to Objects

At the end of the program the destructor for the track must delete the horse objects, which it obtained with `new` in its constructor. Notice that we can't just say

```
delete[] hArray;  //deletes pointers, but not horses
```

This deletes the array of pointers, but not what the pointers point to. Instead we must go through the array element by element, and delete each horse individually:

```
for(int j=0; j<total_horses; j++)  //deletes horses
   delete hArray[j];
```

The `putch()` Function

We want each horse to be a different color, but not all compilers allow `cout` to generate colors. This is true of the current version of Borland C++Builder. However, some old C functions will generate colors. For this reason we use `putch()` when displaying the horses, in the line

```
putch(' '); putch('\xDB'); putch(horse_char); putch('\xDB');
```

This function requires the CONIO.H include file (furnished with the compiler). We don't need to include this file explicitly in HORSE.CPP because it is already included in MSOFTCON.H or BORLACON.H.

Multiplicity in the UML

Let's look at a UML class diagram of the HORSE program, shown in Figure 10.20. This diagram will introduce a UML concept called *multiplicity*.

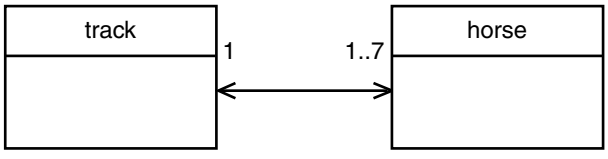

FIGURE 10.20
UML class diagram of the HORSE program.

Sometimes exactly one object of class A relates to exactly one object of class B. In other situations, many objects of a class, or a specific number, may be involved in an association. The number of objects involved in an association is called the *multiplicity* of the association. In class diagrams, numbers or symbols are used at both ends of the association line to indicate multiplicity. Table 10.2 shows the UML multiplicity symbols.

TABLE 10.2 The UML Multiplicity Symbols

Symbol	*Meaning*
1	One
*	Some (0 to infinity)
0..1	None or one

TABLE 10.2 Continued

Symbol	*Meaning*
1..*	One or more
2..4	Two, three, or four
7,11	Seven or eleven

If an association line had a 1 at the Class A end and a * at the class B end, that would mean that one object of class A interacted with an unspecified number of class B objects.

In the HORSE program there is one track but there can be up to 7 horses. This is indicated by the 1 at the `track` end of the association line and the 1..7 at the `horse` end. We assume that one horse is enough for a race, as might happen in time trials.

UML State Diagrams

In this section we'll introduce a new kind of UML diagram: the *state diagram* (also called the statechart diagram).

The UML class diagrams we examined in earlier chapters show relationships between classes. Class diagrams reflect the organization of the program's code. They are *static* diagrams, in that these relationships (such as association and generalization) do not change as the program runs.

However, it's sometimes useful to examine the *dynamic* behavior of particular class objects over time. An object is created, it is affected by events or messages from other parts of the program, it perhaps makes decisions, it does various things, and it is eventually deleted. That is, its situation changes over time. State diagrams show this graphically.

Everyone is familiar with the concept of *state* when applied to devices in our everyday lives. A radio has an `On` state and an `Off` state. A washing machine might have `Washing`, `Rinsing`, `Spinning`, and `Stopped` states. A television set has a state for each channel it is currently receiving (the `Channel 7 Active` state, and so on).

Between the states are *transitions*. As a result of a timer having reached (say) the 20-minute point, the washing machine makes a transition from the `Rinse` state to the `Spin` state. As a result of a message from the remote-control unit, the TV makes a transition from the `Channel 7 Active` state to the `Channel 2 Active` state.

Figure 10.21 shows a state diagram based on the HORSE program seen earlier in this chapter. It shows the different states a `horse` object can find itself in as the program runs.

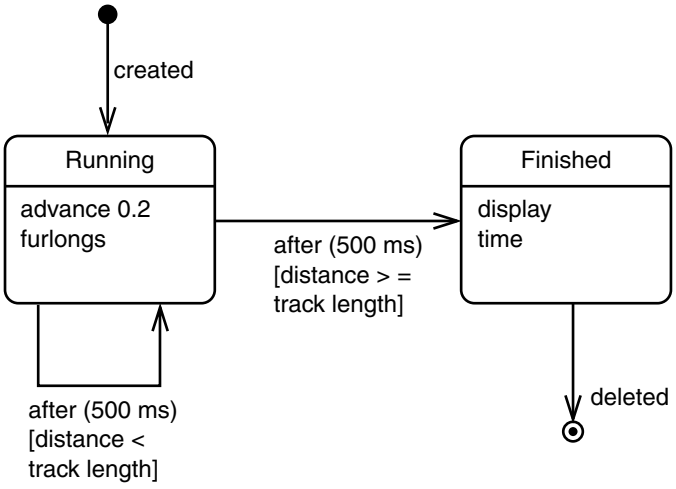

FIGURE 10.21
State diagram of a `horse` *object.*

States

In UML state diagrams, a state is represented by a rectangle with rounded corners. The state is named at the top of the rectangle. State names usually begin with a capital letter. Below the name are any *activities* the object performs when it enters the state.

State diagrams can include two special states: a black disk represents the *initial state*, and a black disk surrounded by a circle represents the *final state*. These are shown in the figure.

After it is created, a `horse` object can be in only two major states: before it reaches the finish line it's in the `Running` state, and afterwards it's in the `Finished` state.

Unlike classes in a class diagram, there's nothing in a program's code that corresponds exactly to states in a state diagram. To know what states to include, you must have an idea what circumstances an object will find itself in, and what it will do as a result. You then make up appropriate names for the states.

Transitions

Transitions between states are represented by directed arrows from one rectangle to another. If the transition is triggered by an event, it can be labeled with the event name, as are the `created` and `deleted` transitions in the figure. Transition names are not capitalized. The names can be closer to English than to C++ usage.

The event that triggers the other two transitions is the timing out of a 500 millisecond timer. The keyword `after` is used to name these transitions, with the time as a parameter.

Transitions can also be labeled with what the UML calls a *guard*: a condition that must be satisfied if the transition is to occur. Guards are written in brackets. The two `after()` transitions have guards as well as event names. Because the events are the same, the guards determine which transition will occur.

Note that one of these transitions is a *self transition*: it returns to the same state where it began.

Racing from State to State

Each time it enters the `Running` state, the `horse` object carries out an activity that consists of increasing the distance it has run by 0.2 furlongs. As long as it has not yet reached the finish line, the `[distance < track length]` guard is true and the `Running` state transitions back to itself. When the horse reaches the finish line, `[distance >= track length]` becomes true, and the `horse` transitions to the `Finished` state, where it displays its total time for the race. It then waits to be deleted.

We've shown enough to give you an idea what state diagrams do. There is of course much more to learn about them. We'll see an example of a more complex state diagram that describes an `elevator` object in Chapter 13, "Multifile Programs."

Debugging Pointers

Pointers can be the source of mysterious and catastrophic program bugs. The most common problem is that the programmer has failed to place a valid address in a pointer variable. When this happens the pointer can end up pointing anywhere in memory. It could be pointing to the program code, or to the operating system. If the programmer then inserts a value into memory using the pointer, the value will write over the program or operating instructions, and the computer will crash or evince other uncharming behavior.

A particular version of this scenario takes place when the pointer points to address 0, which is called `NULL`. This happens, for example, if the pointer variable is defined as a *global variable*, since global variables are automatically initialized to 0. Here's a miniprogram that demonstrates the situation:

```
int* intptr;          //global variable, initialized to 0
void main()
   {                  //failure to put valid address in intptr
   *intptr = 37;      //attempts to put 37 in address at 0
   }                  //result is error
```

When `intptr` is defined, it is given the value 0, since it is global. The single program statement will attempt to insert the value 37 into the address at 0.

Fortunately, however, the runtime error-checking unit built into the program by the compiler is waiting for attempts to access address 0, and will display an error message (perhaps an *access violation*, *null pointer assignment*, or *page fault*) and terminate the program. If you see such a message, one possibility is that you have failed to properly initialize a pointer.

Summary

This has been a whirlwind tour through the land of pointers. There is far more to learn, but the topics we've covered here will provide a basis for the examples in the balance of the book and for further study of pointers.

We've learned that everything in the computer's memory has an address, and that addresses are *pointer constants*. We can find the addresses of variables using the address-of operator `&`.

Pointers are variables that hold address values. Pointers are defined using an asterisk (`*`) to mean *pointer to*. A data type is always included in pointer definitions (except `void*`), since the compiler must know what is being pointed to, so that it can perform arithmetic correctly on the pointer. We access the thing pointed to using the asterisk in a different way, as the *dereference operator*, meaning *contents of the variable pointed to by*.

The special type `void*` means a pointer to *any* type. It's used in certain difficult situations where the same pointer must hold addresses of different types.

Array elements can be accessed using array notation with brackets or pointer notation with an asterisk. Like other addresses, the address of an array is a constant, but it can be assigned to a variable, which can be incremented and changed in other ways.

When the address of a variable is passed to a function, the function can work with the original variable. (This is not true when arguments are passed by value.) In this respect passing by pointer offers the same benefits as passing by reference, although pointer arguments must be *dereferenced* or accessed using the dereference operator. However, pointers offer more flexibility in some cases.

A string constant can be defined as an array or as a pointer. The pointer approach may be more flexible, but there is a danger that the pointer value will be corrupted. Strings, being arrays of type `char`, are commonly passed to functions and accessed using pointers.

The `new` operator obtains a specified amount of memory from the system and returns a pointer to the memory. This operator is used to create variables and data structures during program execution. The `delete` operator releases memory obtained with `new`.

When a pointer points to an object, members of the object's class can be accessed using the access operator `->`. The same syntax is used to access structure members.

Classes and structures may contain data members that are pointers to their own type. This permits the creation of complex data structures such as linked lists.

There can be pointers to pointers. These variables are defined using the double asterisk; for example, `int** pptr`.

Multiplicity in UML class diagrams shows the number of objects involved in an association.

UML state diagrams show how a particular object's situation changes over time. States are represented by rectangles with rounded corners, and transitions between states are represented by directed lines.

Questions

Answers to these questions can be found in Appendix G.

1. Write a statement that displays the address of the variable `testvar`.
2. The contents of two pointers that point to adjacent variables of type `float` differ by _____.
3. A pointer is
   a. the address of a variable.
   b. an indication of the variable to be accessed next.
   c. a variable for storing addresses.
   d. the data type of an address variable.
4. Write expressions for the following:
   a. The address of `var`
   b. The contents of the variable pointed to by `var`
   c. The variable `var` used as a reference argument
   d. The data type pointer-to-`char`
5. An address is a _____, while a pointer is a _____.
6. Write a definition for a variable of type pointer-to-`float`.
7. Pointers are useful for referring to a memory address that has no _____.
8. If a pointer `testptr` points to a variable `testvar`, write a statement that represents the contents of `testvar` but does not use its name.

9. An asterisk placed after a data type means _____. An asterisk placed in front of a variable name means _____.
10. The expression `*test` can be said to

    a. be a pointer to `test`.

    b. refer to the contents of `test`.

    c. dereference `test`.

    d. refer to the value of the variable pointed to by `test`.
11. Is the following code correct?

```
int intvar = 333;
int* intptr;
cout << *intptr;
```

12. A pointer to `void` can hold pointers to _____.
13. What is the difference between `intarr[3]` and `*(intarr+3)`?
14. Write some code that uses pointer notation to display every value in the array `intarr`, which has 77 elements.
15. If `intarr` is an array of integers, why is the expression `intarr++` not legal?
16. Of the three ways to pass arguments to functions, only passing by _____ and passing by _____ allow the function to modify the argument in the calling program.
17. The type of variable a pointer points to must be part of the pointer's definition so that

    a. data types don't get mixed up when arithmetic is performed on them.

    b. pointers can be added to one another to access structure members.

    c. no one's religious conviction will be attacked.

    d. the compiler can perform arithmetic correctly to access array elements.
18. Using pointer notation, write a prototype (declaration) for a function called `func()` that returns type `void` and takes a single argument that is an array of type `char`.
19. Using pointer notation, write some code that will transfer 80 characters from the string `s1` to the string `s2`.
20. The first element in a string is

    a. the name of the string.

    b. the first character in the string.

    c. the length of the string.

    d. the name of the array holding the string.
21. Using pointer notation, write the prototype for a function called `revstr()` that returns a string value and takes one argument that represents a string.

22. Write a definition for an array `numptrs` of pointers to the strings `One`, `Two`, and `Three`.
23. The `new` operator
    a. returns a pointer to a variable.
    b. creates a variable called `new`.
    c. obtains memory for a new variable.
    d. tells how much memory is available.
24. Using `new` may result in less _____ memory than using an array.
25. The `delete` operator returns _____ to the operating system.
26. Given a pointer `p` that points to an object of type `upperclass`, write an expression that executes the `exclu()` member function in this object.
27. Given an object with index number 7 in array `objarr`, write an expression that executes the `exclu()` member function in this object.
28. In a linked list
    a. each link contains a pointer to the next link.
    b. an array of pointers points to the links.
    c. each link contains data or a pointer to data.
    d. the links are stored in an array.
29. Write a definition for an array `arr` of 8 pointers that point to variables of type `float`.
30. If you wanted to sort many large objects or structures, it would be most efficient to
    a. place them in an array and sort the array.
    b. place pointers to them in an array and sort the array.
    c. place them in a linked list and sort the linked list.
    d. place references to them in an array and sort the array.
31. Express the multiplicities of an association that has fewer than 10 objects at one end and more than 2 objects at the other.
32. The states in a state diagram correspond to
    a. messages between objects.
    b. circumstances in which an object finds itself.
    c. objects in the program.
    d. changes in an object's situation.

33. True or false: a transition between states exists for the duration of the program.
34. A guard in a state diagram is

    a. a constraint on when a transition can occur.

    b. a name for certain kinds of transitions.

    c. a name for certain kinds of states.

    d. a restriction on the creation of certain states.

Exercises

Answers to starred exercises can be found in Appendix G.

*1. Write a program that reads a group of numbers from the user and places them in an array of type `float`. Once the numbers are stored in the array, the program should average them and print the result. Use pointer notation wherever possible.

*2. Start with the `String` class from the NEWSTR example in this chapter. Add a member function called `upit()` that converts the string to all uppercase. You can use the `toupper()` library function, which takes a single character as an argument and returns a character that has been converted (if necessary) to uppercase. This function uses the CCTYPE header file. Write some code in `main()` to test `upit()`.

*3. Start with an array of pointers to strings representing the days of the week, as found in the PTRTOSTR program in this chapter. Provide functions to sort the strings into alphabetical order, using variations of the `bsort()` and `order()` functions from the PTRSORT program in this chapter. Sort the pointers to the strings, not the actual strings.

*4. Add a destructor to the LINKLIST program. It should delete all the links when a `linklist` object is destroyed. It can do this by following along the chain, deleting each link as it goes. You can test the destructor by having it display a message each time it deletes a link; it should delete the same number of links that were added to the list. (A destructor is called automatically by the system for any existing objects when the program exits.)

5. Suppose you have a `main()` with three local arrays, all the same size and type (say `float`). The first two are already initialized to values. Write a function called `addarrays()` that accepts the addresses of the three arrays as arguments; adds the contents of the first two arrays together, element by element; and places the results in the third array before returning. A fourth argument to this function can carry the size of the arrays. Use pointer notation throughout; the only place you need brackets is in defining the arrays.

6. Make your own version of the library function `strcmp(s1, s2)`, which compares two strings and returns –1 if `s1` comes first alphabetically, 0 if `s1` and `s2` are the same, and 1 if `s2` comes first alphabetically. Call your function `compstr()`. It should take two `char*` strings as arguments, compare them character by character, and return an `int`. Write a `main()` program to test the function with different combinations of strings. Use pointer notation throughout.
7. Modify the `person` class in the PERSORT program in this chapter so that it includes not only a name, but also a `salary` item of type `float` representing the person's salary. You'll need to change the `setName()` and `printName()` member functions to `setData()` and `printData()`, and include in them the ability to set and display the salary as well as the name. You'll also need a `getSalary()` function. Using pointer notation, write a `salsort()` function that sorts the pointers in the `persPtr` array by salary rather than by name. Try doing all the sorting in `salsort()`, rather than calling another function as PERSORT does. If you do this, don't forget that `->` takes precedence over `*`, so you'll need to say

   ```
   if( (*(pp+j))->getSalary() > (*(pp+k))->getSalary() )
       { /* swap the pointers */ }
   ```

8. Revise the `additem()` member function from the LINKLIST program so that it adds the item at the end of the list, rather than the beginning. This will cause the first item inserted to be the first item displayed, so the output of the program will be

   ```
   25
   36
   49
   64
   ```

   To add the item, you'll need to follow the chain of pointers to the end of the list, then change the last link to point to the new link.
9. Let's say that you need to store 100 integers so that they're easily accessible. However, let's further assume that there's a problem: The memory in your computer is so fragmented that the largest array that you can use holds only 10 integers. (Such problems actually arise, although usually with larger memory objects.) You can solve this problem by defining 10 separate `int` arrays of 10 integers each, and an array of 10 pointers to these arrays. The `int` arrays can have names like `a0`, `a1`, `a2`, and so on. The address of each of these arrays can be stored in the pointer array of type `int*`, which can have a name like `ap` (for array of pointers). You can then access individual integers using expressions like `ap[j][k]`, where `j` steps through the pointers in `ap` and `k` steps through individual integers in each array. This looks as if you're accessing a two-dimensional array, but it's really a group of one-dimensional arrays.

   Fill such a group of arrays with test data (say the numbers 0, 10, 20, and so on up to 990). Then display the data to make sure it's correct.

10. As presented, Exercise 9 is rather inelegant because each of the 10 `int` arrays is declared in a different program statement, using a different name. Each of their addresses must also be obtained using a separate statement. You can simplify things by using `new`, which allows you to allocate the arrays in a loop and assign pointers to them at the same time:

```
for(j=0; j<NUMARRAYS; j++)          // allocate NUMARRAYS arrays
   *(ap+j) = new int[MAXSIZE];      // each MAXSIZE ints long
```

    Rewrite the program in Exercise 9 to use this approach. You can access the elements of the individual arrays using the same expression mentioned in Exercise 9, or you can use pointer notation: `*(*(ap+j)+k)`. The two notations are equivalent.

11. Create a class that allows you to treat the 10 separate arrays in Exercise 10 as a single one-dimensional array, using array notation with a single index. That is, statements in `main()` can access their elements using expressions like `a[j]`, even though the class member functions must access the data using the two-step approach. Overload the subscript operator `[]` (see Chapter 9, "Inheritance") to achieve this result. Fill the arrays with test data and then display it. Although array notation is used in the class interface in `main()` to access "array" elements, you should use only pointer notation for all the operations in the implementation (within the class member functions).

12. Pointers are complicated, so let's see whether we can make their operation more understandable (or possibly more impenetrable) by simulating their operation with a class.

    To clarify the operation of our homemade pointers, we'll model the computer's memory using arrays. This way, since array access is well understood, you can see what's really going on when we access memory with pointers.

    We'd like to use a single array of type `char` to store all types of variables. This is what a computer memory really is: an array of bytes (which are the same size as type `char`), each of which has an address (or, in array-talk, an index). However, C++ won't ordinarily let us store a `float` or an `int` in an array of type `char`. (We could use unions, but that's another story.) So we'll simulate memory by using a separate array for each data type we want to store. In this exercise we'll confine ourselves to one numerical type, `float`, so we'll need an array of this type; call it `fmemory`. However, pointer values (addresses) are also stored in memory, so we'll need another array to store them. Since we're using array indexes to model addresses, and indexes for all but the largest arrays can be stored in type `int`, we'll create an array of this type (call it `pmemory`) to hold these "pointers."

    An index to `fmemory` (call it `fmem_top`) points to the next available place where a `float` value can be stored. There's a similar index to `pmemory` (call it `pmem_top`). Don't worry about running out of "memory." We'll assume these arrays are big enough so that each time we store something we can simply insert it at the next index number in the array. Other than this, we won't worry about memory management.

Create a class called `Float`. We'll use it to model numbers of type `float` that are stored in `fmemory` instead of real memory. The only instance data in `Float` is its own "address"; that is, the index where its `float` value is stored in `fmemory`. Call this instance variable `addr`. Class `Float` also needs two member functions. The first is a one-argument constructor to initialize the `Float` with a `float` value. This constructor stores the `float` value in the element of `fmemory` pointed to by `fmem_top`, and stores the value of `fmem_top` in `addr`. This is similar to how the compiler and linker arrange to store an ordinary variable in real memory. The second member function is the overloaded `&` operator. It simply returns the pointer (really the index, type `int`) value in `addr`.

Create a second class called `ptrFloat`. The instance data in this class holds the address (index) in `pmemory` where some other address (index) is stored. A member function initializes this "pointer" with an `int` index value. The second member function is the overloaded `*` (dereference, or "contents of") operator. Its operation is a tad more complicated. It obtains the address from `pmemory`, where its data, which is also an address, is stored. It then uses this new address as an index into `fmemory` to obtain the float value pointed to by its address data.

```
float& ptrFloat::operator*()
   {
   return fmemory[ pmemory[addr] ];
   }
```

In this way it models the operation of the dereference operator (`*`). Notice that you need to return by reference from this function so that you can use `*` on the left side of the equal sign.

The two classes `Float` and `ptrFloat` are similar, but `Float` stores `floats` in an array representing memory, and `ptrFloat` stores `ints` (representing memory pointers, but really array index values) in a different array that also represents memory.

Here's a typical use of these classes, from a sample `main()`:

```
Float var1 = 1.234;          // define and initialize two Floats
Float var2 = 5.678;

ptrFloat ptr1 = &var1;       // define two pointers-to-Floats,
ptrFloat ptr2 = &var2;       // initialize to addresses of Floats

cout << " *ptr1=" << *ptr1;  // get values of Floats indirectly
cout << " *ptr2=" << *ptr2;  // and display them

*ptr1 = 7.123;               // assign new values to variables
*ptr2 = 8.456;               // pointed to by ptr1 and ptr2
```

```
cout << " *ptr1=" << *ptr1;  // get new values indirectly
cout << " *ptr2=" << *ptr2;  // and display them
```

Notice that, aside from the different names for the variable types, this looks just the same as operations on real variables. Here's the output from the program:

```
*ptr1=1.234
*ptr2=2.678

*ptr1=7.123
*ptr2=8.456
```

This may seem like a roundabout way to implement pointers, but by revealing the inner workings of the pointer and address operator, we have provided a different perspective on their true nature.