1.11 Initializing Object Arrays

If the class includes a parameterized constructor, an array of objects can be initialized. For example, in the following program, samp is a parameterized class and sampArray is an initialized array of objects of that class:

```
    // Initialize an array of objects.
    #include <iostream.h>
    class samp
    {
int a;
    public:
       samp(int n) { a = n; }
       intget_a() { return a; }
    };

    int main()
    {
    sampsampArray[4] = { -1, -2, -3, -4 };
    int i;

    for(i=0; i<4; i++)
    cout<<sampArray[i].get_a() << ' ';
    cout<< "\n";

    return 0;
    }
```

This program displays the following:

```
-1 -2 -3 -4
```

As the output confirms, the values –1 through –4 are passed to the samp constructor. Actually, the syntax shown in the initialization list is shorthand forthis longer form:

```
sampsampArray[4] = {samp(-1), samp(-2),
                    samp(-3), samp(-4) };
```

However, the form used in the program is more common, although this form will work only with arrays whose constructors take only one argument. When initializing an array of objects whose constructor takes more than one argument, you must use the longer form of initialization. For example:

```
#include <iostream.h>
class samp
{
    int a, b;
public:
    samp(int n, int m) { a = n; b = m; }
    intget_a() { return a; }
    intget_b() { return b; }
};

int main()
{
sampsampArray[4][2] =
{ samp(1, 2),samp(3, 4),
samp(5, 6),samp(7, 8),
samp(9, 10),samp(11,12),
samp(13, 14), samp(15, 16)};
   int i;
   for(i=0; i<4; i++)
   {
   cout<<sampArray[i][0].get_a() << ' ';
   cout<<sampArray[i][0].get_b() << "\n";
   cout<<sampArray[i][1].get_a() << ' ';
   cout<<sampArray[i][1].get_b() << "\n";
   }
   cout<< "\n";
   return 0;
}
```

In this example, samp's constructor takes two arguments. Here, the array sampArrayis declared and initialized in main( )

by using direct calls to samp's constructor. Wheninitalizing arrays, you can always use the long form of initialization, even if the object takes only one argument. It's just that the short form is more convenient when only one argument is required. The program displays

1.12 Pointers to Objects

To access an element of an object when using the actual object itself, use the dot operator. To access a specific element of an object when using a pointer to the object, you must use the arrow operator. (The use of the dot and arrow operators for objects parallels their use for structures and unions.)

To declare an object pointer, you use the same declaration syntax that you would use for any other pointer type. The next program creates a simple class called P_ex, defines an object of that class, called ob, and defines a pointer to an object of type P_ex, called p. It then illustrates how to access ob directly, and how to use a pointer to access it indirectly.

```
// A simple example using an object pointer.
#include <iostream.h>
class P_ex
{
     intnum;
public:
      void set_num(intval) {num = val;}
      void show_num();
};

void P_ex::show_num()
{
cout<<num<< "\n";
}

int main()
{
P_exob, *p; //declare an object and pointer to it
ob.set_num(1);
```

```
  // access ob directly
ob.show_num();

   p = &ob; // assign p the address of ob
   p->show_num(); // access ob using pointer

return 0;
}
```

Notice that the address of ob is obtained by using the & (address of) operator in the same way that the address is obtained for any type of variable.

As you know, when a pointer is incremented or decremented, it is increased or decreased in such a way that it will always point to the next element of its base type. The same thing occurs when a pointer to an object is incremented or decremented: the next object is pointed to. To illustrate this, the preceding program has been modified here so that ob is a two-element array of type P_ex. Notice how p is incremented and decremented to access the two elements in the array.

```
// Incrementing and decrementing an object pointer.
 #include <iostream.h>
 class P_ex
 {
    intnum;
 public:
    void set_num(intval) {num = val;}
    void show_num();
 };

 void P_ex::show_num()
 {
 cout<<num<< "\n";
 }
 int main()
 {
  P_exob[2], *p;

  ob[0].set_num(10); // access objects directly
```

```
  ob[1].set_num(20);

  p = &ob[0];// obtain pointer to first element
  p->show_num();// show value of ob[0] using pointer

  p++;  // advance to next object
  p->show_num(); // show value of ob[1] using pointer

  p--;  // retreat to previous object
  p->show_num(); // again show value of ob[0]

  return 0;
}
```

The output from this program is

```
10
20
10
```

As you will see later in this book, object pointers play an important role in one ofC++'s most important concepts: polymorphism.