1.7 An Initialization Alternative

If a constructor takes only one parameter, then you can use an alternative method toinitialize it. Consider the following program:

```
#include <iostream>
class myclass
{
    int a;
public:
myclass(int x);
intget_a();
};
myclass::myclass(int x)
{
a = x;
}
intmyclass::get_a()
{
return a;
}
int main()
{
    myclassob = 4; // calls myclass(4)
    cout<<ob.get_a();
    return 0;
}
```

Here, the constructor for myclass takes one parameter. Pay special attention to howob is declared in main( ). It uses this declaration:

```
myclassob = 4;
```

In this form of initialization, 4 is automatically passed to the x parameter in the myclass( ) constructor. That is, the declaration statement is handled by the compiler as if it were written like this:

```
myclassob = myclass(4);
```

In general, any time that you have a constructor that requires only one argument, you can use either ob(x) or ob = x to initialize an object. The reason for this is that whenever you create a constructor that takes one argument, you are also implicitly creating a conversion from the type of that argument to the type of the class.Remember that the alternative shown here applies only to constructors that have exactly one parameter.

1.8 Inline Functions

Before we continue exploring the class, a small, but important digression is in order. Although it does not pertain specifically to object-oriented programming, one very useful feature of C++, called an inline function, is frequently used in class definitions. An inline function is a function whose code is expanded in line at the point at which it is invoked, rather than being called. There are two ways to create an inline function. The first is to use the inline modifier. For example, to create an inline function called f that returns an int and takes no parameters, you declare it like this:

```
inline int f()
{
// ...
}
```

The inline modifier precedes all other aspects of a function's declaration.Thereason for inline functions is efficiency. Every time a function is called, a series of instructions must be executed, both to set up the function call, including pushing arguments onto the stack, and to return from the function. In some cases, many CPU cycles are used to perform these actions. However, when a function is expanded in line, no such

overhead exists, and the overall speed of your program will increase. Even so, in cases where the inline function is large, the overall size of your program will also increase. For this reason, the best inline functions are those that are very small. Larger functions are usually left as normal functions.The following program demonstrates inline.

```
#include <iostream.h>
class cl
{
    int i; // private by default
public:
     intget_i();
     void put_i(int j);
} ;

inline int cl::get_i()
{
    return i;
}
inline void cl::put_i(int j)
{
    i = j;
}

int main()
{
cl s;
s.put_i(10);
cout<<s.get_i();
return 0;
}
```

Here, the code for get_i( ) and put_i( ) is expanded in line rather than being called. Thus, in main( ), the line

```
s.put_i(10);
```

is functionally equivalent to this assigment statement:

```
s.i = 10;
```

Of course, because i is private to cl, this line could not actually be used in main( ), but by in-lining put_i( ), the same effect is produced and the function call is avoided.

It is important to understand that technically, inline is a request, not a command, that the compiler generate inline code. There are various situations that might prevent the compiler from complying with the request. Here are some examples:

1.9 Creating Inline Functions Inside a Class

There is another way to create an inline function. This is accomplished by defining the code to a member function inside a class declaration. Any function that is defined inside a class declaration is automatically made into an inline function. It is not necessary to precede its declaration with the keyword inline. For example, the preceding program can be rewritten as shown here:

```
#include <iostream.h>
class cl
{
     int i; // private by default
public:
      // automatic inline functions
      intget_i() { return i; }
   void put_i(int j) { i = j; }
};

int main()
{
     cl s;
     s.put_i(10);
     cout<<s.get_i();

     return 0;
}
```

Here, get_i( ) and put_i( ) are defined inside cl and are automatically inline.Notice the way the function code is arranged inside cl. For very short functions, this arrangement reflects common C++ style. However, there is no reason that you could not format the functions as shown here:

```
    class cl
    {
         int i; // private by default
public:
         // inline functions
         intget_i()
         {
         return i;
         }
         void put_i(int j)
         {
         i = j;
         }
    };
```

Generally, short functions like those illustrated in this example are defined inside the class declaration.

1.10 Arrays of Objects

You can create arrays of objects in the same way that you create arrays of any other data type. For example, the following program establishes a class called display that holds the resolution of a video display mode. Inside main( ), an array of three display objects is created, and the objects that comprise the elements of the array are accessed by using the normal array-indexing procedure.

```
// An example of arrays of objects
#include <iostream.h>
```

```
enum resolution {low, medium, high};
class display
{
   int width;
   int height;
   resolution res;
public:
   void set_dim(int w, int h)
   {width = w; height = h;}
   void get_dim(int&w, int&h)
   {w = width; h = height;}
   void set_res(resolution r) {res = r;}
   resolution get_res() {return res;}
};
char names[3][7] = { "low","medium", "high"};
int main()
{
   display display_mode[3];
   int i, w, h;
   display_mode[0].set_res(low);
   display_mode[0].set_dim(640, 480);

   display_mode[1].set_res(medium);
   display_mode[1].set_dim(800, 600);

   display_mode[2].set_res(high);
   display_mode[2].set_dim(1600, 1200);

   cout<< "Available display modes:\n\n";

   for(i=0; i<3; i++)
{
cout<< names[display_mode[i].get_res()] << ": ";
   display_mode[i].get_dim(w, h);
out << w << " by " << h << "\n";
   }

   return 0;
}
```

This program produces the following output:

```
Available display modes:
low: 640 by 480
medium: 800 by 600
high: 1600 by 1200
```

Notice how the two-dimensional character array names is used to convert between an enumerated value and its equivalent character string. In all enumerations that do not contain explicit initializations, the first constant has the value 0, the second 1, and so on. Therefore, the value returned by get_res( ) can be used to index the names array, causing the appropriate name to be printed.