2.3 Overloading Constructors

Although they perform a unique service, constructors are not much different from other types of functions, and they too can be overloaded. To overload a class's constructor, simply declare the various forms it will take and define each action relative to these forms. For example, the following program declares a class called timer, when an object of type timer is created, it is given an initial time value. When the run( ) function is called, the timer print the number of seconds. In this example, the constructor has been overloaded to allow the time to be specified in seconds as either an integer or a string, or in minutes and seconds by specifying two integers.

```
#include <iostream.h>
#include <stdlib.h>
class timer
{int seconds;
public:
    // seconds specified as a string
    timer(char *t) { seconds = atoi(t); }
    // seconds specified as integer
    timer(int t) { seconds = t; }
    // time specified in minutes and seconds
    timer(int min, int sec) { seconds = min*60 +
  sec; }
    void run();
} ;
void timer::run()
{
cout<<seconds<< "\n"; // print the seconds
}
int main()
{
  timer a(10), b("20"), c(1, 10);
  a.run(); // count 10 seconds
  b.run(); // count 20 seconds
  c.run(); // count 1 minute, 10 seconds
  return 0;
}
```

When a, b, and c are created inside main( ), they are given initial values using the three different methods supported by the overloaded constructor functions. Each approach causes the appropriate constructor to be utilized, thus properly initializing all three variables.

In the preceding program, you may see little value in overloading a constructor function, because it is not difficult to simply decide on a single way of specifying the time. However, if you were creating a library of classes for someone else to use, then you might want to supply constructors for the most common forms of initialization, thereby allowing the programmer to utilize the most appropriate form for his or her program. Also, as you will see shortly, there is one C++ attribute that makes overloaded constructors quite valuable.

2.4 Dynamic Initialization

In C++, both local and global variables can be initialized at run time. This process is sometimes referred to as dynamic initialization. So far, most initializations that you have seen in this book have used constants. However, under dynamic initialization, a variable can be initialized at run time using any C++ expression valid at the time the variable is declared. This means that you can initialize a variable by using other variables and/or function calls, so long as the overall expression has meaning when the declaration is encountered. For example, the following are all perfectly valid variable initializations in C++:

```
int n = strlen(str);
double arc = sin(theta);
float d = 1.02 * count / deltax;
```

2.5Applying Dynamic Initialization to Constructors

Like simple variables, objects can be initialized dynamically when they are created. This feature allows you to create exactly the type of object you need, using information that is known only at run time. To illustrate how dynamic initialization works,

let’s rework the timer program from the previous section.

Recall that in the first example of the timer program, there is little to be gained by overloading the timer() constructor, because all objects of its type are initialized using constants provided at compile time. However, in cases where an object will be initialized at run time, there may be significant advantages to providing a variety of initialization formats.

This allows you, the programmer, the flexibility of using the constructor that most closely matches the format of the data available at the moment.For example, in the following version of the timer program, dynamic initialization is used to construct two objects, b and c, at run time:

```
// Demonstrate dynamic initialization.
#include <iostream.h>
#include <stdlib.h>
class timer
{
    int seconds;
public:
// seconds specified as a string
timer(char *t) { seconds = atoi(t); }
// seconds specified as integer
timer(int t) { seconds = t; }
// time specified in minutes and seconds
timer(int min, int sec) { seconds = min*60 +
sec; }
void run();
} ;
void timer::run()
{
cout<<seconds<< "\n"; // print the seconds
}
int main()
{
  timer a(10);
  a.run();
  cout<< "Enter number of seconds: ";
```

```
    char str[80];
    cin>>str;
    timer b(str); // initialize at run time
  b.run();
    cout<< "Enter minutes and seconds: ";
    int min, sec;
    cin>> min >> sec;
    timer c(min, sec); // initialize at run
  time
    c.run();
    return 0;
}
```

As you can see, object a is constructed using an integer constant. However, objects b and c are constructed using information entered by the user. For b, since the user enters a string, it makes sense that timer( ) is overloaded to accept it. In similar fashion, object c is also constructed at run time from user input. In this case, since the time is entered as minutes and seconds, it is logical to use this format for constructing object c. As the example shows, having a variety of initialization formats keeps you from having to perform conversions when initializing an object.

The point of overloading constructors is to help programmers handle greater complexity by allowing objects to be constructed in the most natural manner relative to their specific use. Since there are three common methods of passing timing values to an object, it makes sense that timer( ) be overloaded to accept each method. However, overloading timer( ) to accept days or nanoseconds is probably not a good idea. Littering your code with constructors to handle seldom-used contingencies has a destabilizing influence on your program.

2.6 Assigning Objects

If both objects are of the same type (that is, both are objects of the same class), then one object may be assigned to another. It is not sufficient for the two classes to simply be physically

similar—their type names must be the same. By default, when one objectis assigned to another, a bitwise copy of the firstobject's data is copied to the second. The following program demonstrates object assignment:

```
    // Demonstrate object assignment.
    #include <iostream.h>
    class myclass
    {
int a, b;
    public:
      void setab(int i, int j) { a = i, b = j; }
      void showab();
    };
    void myclass::showab()
    {
        cout<< "a is " << a << '\n';
        cout<< "b is " << b << '\n';
    }

    int main()
    {
      myclass ob1, ob2;
      ob1.setab(10, 20);
      ob2.setab(0, 0);
      cout<< "ob1 before assignment: \n";
      ob1.showab();
      cout<< "ob2 before assignment: \n";
      ob2.showab();
      cout<< '\n';

      ob2 = ob1; // assign ob1 to ob2

      cout<< "ob1 after assignment: \n";
      ob1.showab();
      cout<< "ob2 after assignment: \n";
      ob2.showab();

      return 0;
    }
```

This program displays the following output:

```
ob1 before assignment:
a is 10 b is 20
ob2 before assignment:
a is 0 b is 0
ob1 after assignment:
a is 10 b is 20
ob2 after assignment:
a is 10 b is 20
```

By default, all data from one object is assigned to the other using a bit-by-bit copy. (That is, an exact duplicate is created.) However, as you will see later, it is possible to overload the assignment operator so that customized assignment operations can be defined.

Remember: Assignment of one object to another simply makes the data in those objects identical. The two objects are still completely separate. Thus, a subsequent modification of one object's data has no effect on that of the other.