Introduction to Object-Oriented Programming

The most widely used object-oriented programming language today is C++. C++ provides *classes*—which are its objects. Classes really distinguish C++ from C. In fact, before the name C++ was coined, the C++ language was called “C with classes.”

This chapter attempts to expose you to the world of object-oriented programming, often called *OOP*. You will probably not become a master of OOP in these few short pages, however, you are ready to begin expanding your C++ knowledge.

This chapter introduces the following concepts:

- C++ classes
- Member functions
- Constructors
- Destructors

This chapter concludes your introduction to the C++ language. After mastering the techniques taught in this book, you will be ready to modify the mailing list program in Appendix F to suit your own needs.

What Is a Class?

A *class* is a user-defined data type that resembles a structure. A class can have data members, but unlike the structures you have seen thus far, classes can also have *member functions*. The data members can be of any type, whether defined by the language or by you. The member functions can manipulate the data, create and destroy class variables, and even redefine C++'s operators to act on the class objects.

Classes have several types of members, but they all fall into two categories: data members and member functions.

Data Members

Data members can be of any type. Here is a simple class:

```
// A sphere class.
class Sphere
{
public:
   float r;          // Radius of sphere
   float x, y, z;    // Coordinates of sphere
};
```

Notice how this class resembles structures you have already seen, with the exception of the `public` keyword. The `Sphere` class has four data members: `r`, `x`, `y`, and `z`. In this case, the `public` keyword plays an important role; it identifies the class `Sphere` as a structure. As a matter of fact, in C++, a *public class* is physically identical to a structure. For now, ignore the `public` keyword; it is explained later in this chapter.

Member Functions

A class can also have *member functions* (members of a class that manipulate data members). This is one of the primary features that distinguishes a class from a structure. Here is the `Sphere` class again, with member functions added:

```
#include <math.h>

const float PI = 3.14159;
// A sphere class.
class Sphere
{
public:
   float r;        // Radius of sphere
   float x, y, z;  // Coordinates of sphere
   Sphere(float xcoord, float ycoord, float zcoord, float radius)
     { x = xcoord; y = ycoord; z = zcoord; r = radius; }
   ~Sphere() { }
   float volume()
   {
      return (r * r * r * 4 * PI / 3);
   }
   float surface_area()
   {
     return (r * r * 4 * PI);
   }
};
```

This `Sphere` class has four member functions: `Sphere()`, `~Sphere()`, `volume()`, and `surface_area()`. The class is losing its similarity to a structure. These member functions are very short. (The one with the strange name of `~Sphere()` has no code in it.) If the codes of the member functions were much longer, only the prototypes would appear in the class, and the code for the member functions would follow later in the program.

C++ programmers call class data *objects* because classes do more than simply hold data. Classes act on data; in effect, a class is an object that manipulates itself. All the data you have seen so far in this book is *passive* data (data that has been manipulated by code in the program). Classes' member functions actually manipulate class data.

Constructors create and initialize class data.

In this example, the class member `Sphere()` is a special function. It is a *constructor* function, and its name must always be the same as its class. Its primary use is declaring a new instance of the class.

Examples

1. The following program uses the `Sphere()` class to initialize a class variable (called a class *instance*) and print it.

```
// Filename: C32CON.CPP
// Demonstrates use of a class constructor function.

#include <iostream.h>
const float PI = 3.14159;  // Approximate value of pi.

// A sphere class.
class Sphere
{
public:
   float r;          // Radius of sphere
   float x, y, z;  // Coordinates of sphere
   Sphere(float xcoord, float ycoord,
          float zcoord, float radius)
        { x = xcoord; y = ycoord; z = zcoord; r = radius; }
   ~Sphere() { }
   float volume()
   {
     return (r * r * r * 4 * PI / 3);
   }
   float surface_area()
   {
      return (r * r * 4 * PI);
   }
};

void main()
{
  Sphere s(1.0, 2.0, 3.0, 4.0);

   cout << “X = “ << s.x << “, Y = “ << s.y
        << “, Z = “ << s.z << “, R = “ << s.r << “\n”;
   return;
}
```

Note: In OOP, the `main()` function (and all it calls) becomes smaller because member functions contain the code that manipulates all class data.

Indeed, this program looks different from those you have seen so far. This example is your first true exposure to OOP programming. Here is the output of this program:

```
X = 1, Y = 2, Z = 3, R = 4
```

This program illustrates the `Sphere()` constructor function. The constructor function is the only member function called by the program. Notice the `~Sphere()` member function constructed `s`, and initialized its data members as well.

Destructors erase class data.

The other special function is the *destructor* function, `~Sphere()`. Notice that it also has the same name as the class, but with a *tilde* (~) as a prefix. The destructor function never takes arguments, and never returns values. Also notice that this destructor doesn't do anything. Most destructors do very little. If a destructor has no real purpose, you do not have to specify it. When the class variable goes out of scope, the memory allocated for that class variable is returned to the system (in other words, an automatic destruction occurs). Programmers use destructor functions to free memory occupied by class data in advanced C++ applications.

Similarly, if a constructor doesn't serve any specific function, you aren't required to declare one. C++ allocates memory for a class variable when you define the class variable, just as it does for all other variables. As you learn more about C++ programming, especially when you begin using the advanced concept of *dynamic memory allocation*, constructors and destructors become more useful.

2. To illustrate that the `~Sphere()` destructor does get called (it just doesn't do anything), you can put a `cout` statement in the constructor as seen in the next program:

```
// Filename: C32DES.CPP
// Demonstrates use of a class destructor function.
```

```
#include  <iostream.h>
#include  <math.h>
const float PI = 3.14159;   // Approximate value of pi.

// A sphere class
class Sphere
{
public:
   float  r;        // Radius of sphere
   float  x, y, z;  // Coordinates of sphere
   Sphere(float xcoord, float ycoord,
          float zcoord, float radius)
   { x = xcoord; y = ycoord; z = zcoord; r = radius; }
   ~Sphere()
   {
    cout << “Sphere (“ << x << “, “ << y
         << “, “ << z << “, “ << r << “) destroyed\n”;
   }
   float volume()
   {
      return (r * r * r * 4 * PI / 3);
   }
   float surface_area()
   {
     return (r * r * 4 * PI);
   }
};

void main(void)
{
   Sphere s(1.0, 2.0, 3.0, 4.0);
   // Construct a class instance.
   cout << “X = “ << s.x << “, Y = “
        << s.y << “, Z = “ << s.z << “, R = “ << s.r << “\n”;
   return;
}
```

Here is the output of this program:

```
X = 1, Y = 2, Z = 3, R = 4
Sphere (1, 2, 3, 4) destroyed
```

Notice that `main()` did not explicitly call the destructor function, but `~Sphere()` was called automatically when the class instance went out of scope.

3. The other member functions have been waiting to be used. The following program uses the `volume()` and `surface_area()` functions:

```
// Filename: C32MEM.CPP
// Demonstrates use of class member functions.

#include  <iostream.h>
#include  <math.h>
const float PI = 3.14159;  // Approximate value of pi.

// A sphere class.
class Sphere
{
public:
   float r;          // Radius of sphere
   float x, y, z;    // Coordinates of sphere
   Sphere(float xcoord, float ycoord,
          float zcoord, float radius)
   { x = xcoord; y = ycoord; z = zcoord; r = radius; }
   ~Sphere()
   {
      cout << “Sphere (“ << x << “, “ << y
           << “, “ << z << “, “ << r << “) destroyed\n”;
   }
   float volume()
   {
       return (r * r * r * 4 * PI / 3);
   }
   float surface_area()
   {
     return (r * r * 4 * PI);
   }
};  // End of class.

void main()
{
   Sphere s(1.0, 2.0, 3.0, 4.0);
   cout << “X = “ << s.x << “, Y = “ << s.y
        << “, Z = “ << s.z << “, R = “ << s.r << “\n”;
   cout << “The volume is “ << s.volume() << “\n”;
   cout << “The surface area is “
        << s.surface_area() << “\n”;
}
```

The `volume()` and `surface_area()` functions could have been made *in-line*. This means that the compiler embeds the functions in the code, rather than calling them as functions. In C32MEM.CPP, there is essentially a separate function that is called using the data in `Sphere()`. By making it in-line, `Sphere()` essentially becomes a macro and is expanded in the code.

4. In the following program, `volume()` has been changed to an in-line function, creating a more efficient program:

```
// Filename: C32MEM1.CPP
// Demonstrates use of in-line class member functions.

#include  <iostream.h>
#include  <math.h>
const float PI = 3.14159;  // Approximate value of pi.

// A sphere class.
class Sphere
{
public:
   float r;        // Radius of sphere
   float x, y, z;  // Coordinates of sphere
   Sphere(float xcoord, float ycoord, float zcoord, float radius)
   { x = xcoord; y = ycoord; z = zcoord; r = radius; }
   ~Sphere()
   {
     cout << “Sphere (“ << x << “, “ << y
          << “, “ << z << “, “ << r << “) destroyed\n”;
   }
   inline float volume()
   {
     return (r * r * r * 4 * PI / 3);
   }
   float surface_area()
   {
      return (r * r * 4 * PI);
   }
};

void main()
{
   Sphere s(1.0, 2.0, 3.0, 4.0);
   cout << “X = “ << s.x << “, Y = “ << s.y
        << “, Z = “ << s.z << “, R = “ << s.r << “\n”;
   cout << “The volume is “ << s.volume() << “\n”;
   cout << “The surface area is “ << s.surface_area() << “\n”;
}
```

The inline functions expand to look like this to the compiler:

```
// C32MEM1A.CPP
// Demonstrates use of in-line class member functions.

#include  <iostream.h>
#include  <math.h>
const float PI = 3.14159;  // Approximate value of pi.

// A sphere class
class Sphere
{
public:
   float r;        // Radius of sphere
   float x, y, z;  // Coordinates of sphere
   Sphere(float xcoord, float ycoord, float zcoord, float radius)
   { x = xcoord; y = ycoord; z = zcoord; r = radius; }
   ~Sphere()
   {
      cout << “Sphere (“ << x << “, “ << y
            << “, “ << z << “, “ << r << “) destroyed\n”;
   }
   inline float volume()
   {
      return (r * r * r * 4 * PI / 3);
   }
   float surface_area()
   {
     return (r * r * 4 * PI);
   }
};
```

```
void main()
{
   Sphere s(1.0, 2.0, 3.0, 4.0);
   cout << “X = “ << s.x << “, Y = “ << s.y
        << “, Z = “ << s.z << “, R = “ << s.r << “\n”;
   cout << “The volume is “ << (s.r * s.r * s.r * 4 * PI / 3)
        << “\n”;
   cout << “The surface area is “ << s.surface_area() << “\n”;
}
```

The advantage of using in-line functions is that they execute faster—there’s no function-call overhead involved because no function is actually called. The disadvantage is that if your functions are used frequently, your programs become larger and larger as functions are expanded.

Default Member Arguments

You can also give member functions arguments by default. Assume by default that the *y* coordinate of a sphere will be 2.0, the *z* coordinate will be 2.5, and the radius will be 1.0. Rewriting the previous example’s constructor function to do this results in this code:

```
Sphere(float xcoord, float ycoord = 2.0, float zcoord = 2.5,
       float radius = 1.0)
  { x = xcoord; y = ycoord; z = zcoord; r = radius; }
```

You can create a sphere with the following instructions:

```
Sphere s(1.0);                // Use all default

Sphere t(1.0, 1.1);           // Override y coord

Sphere u(1.0, 1.1, 1.2);      // Override y and z

Sphere v(1.0, 1.1, 1.2, 1.3); // Override all defaults
```

Examples

1. Default arguments are used in the following code.

```
// Filename: C32DEF.CPP
// Demonstrates use of default arguments in
// class member functions.

#include  <iostream.h>
#include  <math.h>
const float PI = 3.14159;  // Approximate value of pi.

// A sphere class.
class Sphere
{
public:
   float r;        // Radius of sphere
   float x, y, z;  // Coordinates of sphere
   Sphere(float xcoord, float ycoord = 2.0,
          float zcoord = 2.5, float radius = 1.0)
    { x = xcoord; y = ycoord; z = zcoord; r = radius; }
   ~Sphere()
    {
      cout << “Sphere (“ << x << “, “ << y
           << “, “ << z << “, “ << r << “) destroyed\n”;
    }
    inline float volume()
    {
      return (r * r * r * 4 * PI / 3);
    }
    float surface_area()
    {
     return (r * r * 4 * PI);
    }
};

void main()
{
   Sphere s(1.0);                // use all default
   Sphere t(1.0, 1.1);           // override y coord
   Sphere u(1.0, 1.1, 1.2);      // override y and z
   Sphere v(1.0, 1.1, 1.2, 1.3); // override all defaults
   cout << “s: X = “ << s.x << “, Y = “ << s.y
        << “, Z = “ << s.z << “, R = “ << s.r << “\n”;
    cout << “The volume of s is “ << s.volume() << “\n”;
    cout << “The surface area of s is “ << s.surface_area() << “\n”;
    cout << “t: X = “ << t.x << “, Y = “ << t.y
         << “, Z = “ << t.z << “, R = “ << t.r << “\n”;
    cout << “The volume of t is “ << t.volume() << “\n”;
    cout << “The surface area of t is “ << t.surface_area() << “\n”;
    cout << “u: X = “ << u.x << “, Y = “ << u.y
         << “, Z = “ << u.z << “, R = “ << u.r << “\n”;
    cout << “The volume of u is “ << u.volume() << “\n”;
    cout << “The surface area of u is “ << u.surface_area() << “\n”;
    cout << “v: X = “ << v.x << “, Y = “ << v.y
         << “, Z = “ << v.z << “, R = “ << v.r << “\n”;
    cout << “The volume of v is “ << v.volume() << “\n”;
    cout << “The surface area of v is “ << v.surface_area() << “\n”;
    return;

}
```

Here is the output from this program:

```
s: X = 1, Y = 2, Z = 2.5, R = 1
The volume of s is 4.188787
The surface area of s is 12.56636
t: X = 1, Y = 1.1, Z = 2.5, R = 1
The volume of t is 4.188787
The surface area of t is 12.56636
u: X = 1, Y = 1.1, Z = 1.2, R = 1
The volume of u is 4.188787
The surface area of u is 12.56636
v: X = 1, Y = 1.1, Z = 1.2, R = 1.3
The volume of v is 9.202764
The surface area of v is 21.237148
Sphere (1, 1.1, 1.2, 1.3) destroyed
Sphere (1, 1.1, 1.2, 1) destroyed
Sphere (1, 1.1, 2.5, 1) destroyed
Sphere (1, 2, 2.5, 1) destroyed
```

Notice that when you use a default value, you must also use the other default values to its right. Similarly, once you define a function’s parameter as having a default value, every parameter to its right must have a default value as well.

2. You also can call more than one constructor; this is called *overloading* the constructor. When having more than one constructor, all with the same name of the class, you must give them each a different parameter list so the compiler can determine which one you intend to use. A common use of overloaded constructors is to create an uninitialized object on the receiving end of an assignment, as you see done here:

```
// C32OVCON.CPP
// Demonstrates use of overloaded constructors.

#include  <iostream.h>
#include  <math.h>
const float PI  = 3.14159;  // Approximate value of pi.

// A sphere class.
class Sphere
{
public:
   float r;        // Radius of sphere
   float x, y, z;  // Coordinates of sphere
   Sphere() { /* doesn’t do anything... */ }
   Sphere(float xcoord, float ycoord,
          float zcoord, float radius)
   { x = xcoord; y = ycoord; z = zcoord; r = radius; }
   ~Sphere()
   {
      cout << “Sphere (“ << x << “, “ << y
           << “, “ << z << “, “ << r << “) destroyed\n”;
   }
   inline float volume()
   {
     return (r * r * r * 4 * PI / 3);
   }
   float surface_area()
   {
     return (r * r * 4 * PI);
   }
};

void main()
{
   Sphere s(1.0, 2.0, 3.0, 4.0);
```

```
    Sphere t;    // No parameters (an uninitialized sphere).

    cout << “X = “ << s.x << “, Y = “ << s.y
         << “, Z = “ << s.z << “, R = “ << s.r << “\n”;
    t = s;
    cout << “The volume of t is “ << t.volume() << “\n”;
    cout << “The surface area of t is “ << t.surface_area()
         << “\n”;
    return;
}
```

Class Member Visibility

Recall that the `Sphere()` class had the label `public`. Declaring the `public` label is necessary because, by default, all members of a class are `private`. Private members cannot be accessed by anything but a member function. In order for data or member functions to be used by other programs, they must be explicitly declared `public`. In the case of the `Sphere()` class, you probably want to hide the actual data from other classes. This protects the data’s integrity. The next program adds a `cube()` and `square()` function to do some of the work of the `volume()` and `surface_area()` functions. There is no need for other functions to use those member functions.

```
// Filename: C32VISIB.CPP
// Demonstrates use of class visibility labels.

#include <iostream.h>
#include <math.h>
const float PI = 3.14159;  // Approximate value of pi.

// A sphere class.
class Sphere
{
private:
   float r;        // Radius of sphere
   float x, y, z;  // Coordinates of sphere
   float cube() { return (r * r * r); }
   float square() { return (r * r); }
public:
   Sphere(float xcoord, float ycoord, float zcoord, float radius)
   { x = xcoord; y = ycoord; z = zcoord; r = radius; }
   ~Sphere()
   {
     cout << “Sphere (“ << x << “, “ << y
          << “, “ << z << “, “ << r << “) destroyed\n”;
   }
   float volume()
   {
     return (cube() * 4 * PI / 3);
   }
   float surface_area()
   {
      return (square() * 4 * PI);
   }
};

void main()
{
   Sphere s(1.0, 2.0, 3.0, 4.0);
   cout << “The volume is “ << s.volume() << “\n”;
   cout << “The surface area is “ << s.surface_area() << “\n”;
   return;
}
```

Notice that the line showing the data members had to be removed from `main()`. The data members are no longer directly accessible except by a member function of class `Sphere`. In other words, `main()` can never directly manipulate data members such as `r` and `z`, even though it calls the constructor function that created them. The private member data is only visible in the member functions. This is the true power of *data hiding;* even your own code cannot get to the data! The advantage is that you define specific data-retrieval, data-display, and data-changing member functions that `main()` must call to manipulate member data. Through these member functions, you set up a buffer between your program and the program’s data structures. If you change the way the data is stored, you do not have to change `main()` or any functions that `main()` calls. You only have to change the member functions of that class.

Review Questions

The answers to the review questions are in Appendix B.

1. What are the two types of class members called?
2. Is a constructor always necessary?
3. Is a destructor always necessary?
4. What is the default visibility of a class data member?
5. How do you make a class member visible outside its class?

Review Exercise

Construct a class to hold personnel records. Use the following data members, and keep them private:

```
char      name[25];
float     salary;
char      date_of_birth[9];
```

Create two constructors, one to initialize the record with its necessary values and another to create an uninitialized record. Create member functions to alter the individual's name, salary, and date of birth.

Summary

You have now been introduced to classes, the data type that distinguishes C++ from its predecessor, C. This was only a cursory glimpse of object-oriented programming. However, you saw that OOP offers an advanced view of your data, combining the data with the member functions that manipulate that data. If you desire to learn more about C++ and become a "guru" of sorts, try *Using Microsoft C/C++ 7* (Que, 0-88022-809-1).