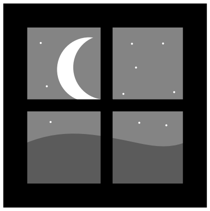

SESSION 19

Maintaining Class Integrity

Session Checklist

- ✔ Writing and using a constructor
- ✔ Constructing data members
- ✔ Writing and using a destructor
- ✔ Controlling access to data members

30 Min. To Go

An object cannot be made responsible for it's own well-being if it has no control over how it is created and how it is accessed. This Session examines the facilities C++ provides for maintaining object integrity.

Creating and Destroying Objects

C++ can initialize an object as part of the declaration. For example:

```
class Student
{
  public:
   int   semesterHours;
```

```
   float gpa;
};

void fn()
{
   Student s = {0, 0};
   //...function continues...
}
```

Here `fn()` has total control of the `Student` object.

We could outfit the class with an initialization function that the application calls as soon as the object is created. This gives the class control over how its data members are initialized. This solution appears as follows:

```
class Student
{
  public:
    // data members
    int   semesterHours;
    float gpa;

    // member functions
    // init - initialize an object to a valid state
    void init()
    {
       semesterHours = 0;
       gpa = 0.0;
    }
};

void fn()
{
    // create a valid Student object
    Student s;     //create the object...
    s.init();      //...then initialize it in valid state

    //...function continues...
}
```

The problem with this "init" solution is that the class must rely on the application to call the `init()` function. This is still not the solution we seek. What we really want is a mechanism that automatically initializes an object when it is created.

The constructor

C++ enables a class to assume responsibility for initializing its objects via a special function called the constructor.

Note

A *constructor* is a member function that is called automatically when an object is created. Similarly, a *destructor* is called when an object expires.

C++ embeds a call to the constructor whenever an object is created. The constructor carries the same name as the class. That way, the compiler knows which member function is the constructor.

Note

The designers of C++ could have made up a different rule, such as: "The constructor must be called `init()`."The Java language uses just such a rule. A different rule wouldn't make any difference, as long as the compiler could recognize the constructor from among the other member functions.

With a constructor, the class `Student` appears as follows:

```
class Student
{
  public:
    // data members
    int    semesterHours;
    float gpa;

    // member functions
    Student()
    {
        semesterHours = 0;
        gpa = 0.0;
    }
};
```

```
void fn()
{
    Student s;       //create an object and initialize it

    //...function continues...
}
```

At the point of the declaration of `s`, the compiler inserts a call to the constructor `Student::Student()`.

This simple constructor was written as an inline member function. Constructors can be written also as outline functions. For example:

```
class Student
{
  public:
    // data members
    int    semesterHours;
    float gpa;

    // member functions
    Student();
};

Student::Student()
{
    semesterHours = 0;
    gpa = 0.0;
}
int main(int nArgc, char* pszArgs)
{
    Student s;       //create the object and initialize it
    return 0;
}
```

I added a small `main()` function here so that you can execute this program. You really should single-step this simple program in your debugger before going any further.

As you single-step through this example, control eventually comes to rest at the `Student s;` declaration. Press Step In one more time and control magically jumps to `Student::Student()`. Continue single-stepping through the constructor. When the function has finished, control returns to the statement after the declaration.

Multiple objects can be declared on a single line. Rerun the single-step process with `fn()` declared as follows:

```
int main(int nArgc, char* pszArgs)
{
    Student s[5];    //create an array of objects
}
```

The constructor is invoked five times, once for each element in the array.

Tip

If you can't get the debugger to work (or you just don't want to bother), add an output statement to the constructor so that you can see output to the screen whenever the constructor is invoked. The effect is not as dramatic, but it is convincing.

20 Min. To Go

Limitations on the constructor

The constructor can only be invoked automatically. It cannot be called like a normal member function. That is, you cannot use something similar to the following to reinitialize a `Student` object:

```
void fn()
{
    Student s;       //create and initialize the object

   //...other stuff...
    s.Student();     //reinitilize it; this doesn't work
}
```

The constructor has no return type, not even `void`. The constructors you see here also have `void` arguments.

Note

The constructor with no arguments is known as the default or void constructor.

The constructor can call other functions. Thus, if you want to be able to reinitialize an object at will, write the following:

```
class Student
{
  public:
```

```
    // data members
    int    semesterHours;
    float gpa;

    // member functions
    // constructor - initialize the object automatically
                     when it is created
    Student()
    {
        init();
    }

    // init - initialize the object
    void init()
    {
        semesterHours = 0;
        gpa = 0.0;
    }
};

void fn()
{
    Student s;       //create and initialize the object

   //...other stuff...
    s.init();        //reinitilize it
}
```

Here the constructor calls a universally available `init()` function, which performs the actual initialization.

Constructing data members

The data members of a class are created at the same time as the object itself. The object data members are actually constructed in the order in which they appear and immediately before the rest of the class. Consider the ConstructMembers program in Listing 19-1. Write statements were added to the constructors of the individual class so that you can see the order in which the objects are created.

Listing 19-1
The ConstructMembers Program

```
// ConstructMembers - create an object with data members
//                    that are also objects of a class
#include <stdio.h>
#include <iostream.h>
class Student
{
  public:
    Student()
    {
       cout << "Constructing student\n";
    }
};

class Teacher
{
  public:
    Teacher()
    {
       cout << "Constructing teacher\n";
    }
};

class TutorPair
{
  public:
    Student student;
    Teacher teacher;
    int     noMeetings;

    TutorPair()
    {
       cout << "Constructing tutor pair\n";
       noMeetings = 0;
    }
};
```

Continued

Listing 19-1 *Continued*

```
int main(int nArgc, char* pArgs[])
{
    cout << "Creating a tutor pair\n";

    TutorPair tp;

    cout << "Back in main\n";
    return 0;
}
```

Executing this program generates this output:

```
Creating a tutor pair
Constructing student
Constructing teacher
Constructing tutor pair
Back in main
```

Creating the object `tp` in `main` invokes the constructor for `TutorPair` automatically. Before control passes to the body of the `TutorPair` constructor, however, the constructors for the two member objects — `student` and `teacher` — are invoked.

The constructor for `Student` is called first because it is declared first. Then the constructor for `Teacher` is called. After these objects are constructed, control returns to the open brace and the constructor for `TutorPair` is allowed to initialize the remainder of the object.

Note

It would not do for `TutorPair` to be responsible for initializing `student` and `teacher`. Each class is responsible for initializing its own objects.

The destructor

Just as objects are created, so are they destroyed. If a class can have a constructor to set things up, it should also have a special member function that's called to destruct, or take apart, the object.

The *destructor* is a special member function that is called when an object is destroyed or, to use C++ parlance, is destructed.

A class may allocate resources in the constructor; these resources need to be deallocated before the object ceases to exist. For example, if the constructor opens a file, the file needs to be closed. Or, if the constructor allocates memory from the heap, this memory must be freed before the object goes away. The destructor allows the class to do these clean-up tasks automatically without relying on the application to call the proper member functions.

The destructor member has the same name as the class, but a tilde (~) precedes it. Like a constructor, the destructor has no return type. For example, the class `Student` with a destructor added appears as follows:

```
class Student
{
  public:
    // data members
    // the roll up figures
    int   semesterHours;
    float gpa;

    // an array to hold each individual grade
    int*  pnGrades;

    // member functions
    // constructor - called when object created;
    //               initializes data members including
    //               allocating an array off of the heap
    Student()
    {
        semesterHours = 0;
        gpa = 0.0;

        // allocate room for 50 grades
        pnGrades = new int[50];
    }

    // destructor - called when object destroyed to put the
    //              heap memory back
   ~Student()
    {
        //return memory to the heap
        delete pnGrades;
```

```
        pnGrades = 0;
    }
};
```

If more than one object is being destructed, then the destructors are invoked in the reverse order from the order in which the constructors were called. This is also true when destructing objects that have class objects as data members. Listing 19-2 shows the output from the program shown in Listing 19-1 with the addition of destructors to all three classes.

Listing 19-2
Output of ConstructMembers After Destructors Are Added

```
Creating a tutor pair
Constructing student
Constructing teacher
Constructing tutor pair
Back in main
Destructing tutor pair
Destructing teacher
Destructing student
```

CD-ROM

The entire program is contained on the accompanying CD-ROM.

The constructor for `TutorPair` is invoked at the declaration of `tp`. The `Student` and `Teacher` data objects are created in the order that they are contained in `TutorPair` before the body of `TutorPair()` is given control. Upon reaching the close brace of `main()`, `tp` goes out of scope. C++ calls `~TutorPair` to destruct `tp`. After the destructor has finished disassembling the `TutorPair` object, `~Student` and `~Teacher` destruct the data member objects.

10 Min. To Go

Access Control

Initializing an object into a known state is only half the battle. The other half is to make sure that external functions cannot “reach into” an object and diddle with its data members.

Note

Allowing external functions access to the data members of a class is akin to allowing me access to the internals of my microwave. If I reach into the microwave and change the wiring, I can hardly blame the designer if the oven catches fire.

The protected keyword

C++ also enables a class to declare members to be off limits to nonmember functions. C++ uses the keyword `protected` to flag a set of class members as not being accessible from functions external to the class.

Note

A class member is *protected* if it can only be accessed from other members of the class.

Tip

The opposite of *protected* is *public*. A *public* member can be accessed from both member and nonmember functions.

For example, in the following version of `Student`, only the functions `grade(double, int)` and `grade()` are accessible to external functions.

```
// ProtectedMembers - demonstrate the use of
//                    protected members
#include <stdio.h>
#include <iostream.h>

// Student
class Student
{
  protected:
    double dCombinedScore;
    int    nSemesterHours;

  public:
    Student()
    {
       dCombinedScore = 0;
       nSemesterHours = 0;
```

```
    }

    // grade - add in the effect of another course grade
    double grade(double dNewGrade, int nHours)
    {
        // if the arguments represent legal values...
        if (dNewGrade >= 0 && dNewGrade <= 4.0)
        {
            if (nHours >0 && nHours <= 5)
            {
                // ...update the GPA information
                dCombinedScore += dNewGrade * nHours;
                nSemesterHours += nHours;
            }
        }
        return grade();
    }

    // grade - return the current GPA
    double grade()
    {
        return dCombinedScore / nSemesterHours;
    }

    // semesterHours - return the number of semester
    //                 hours the student has attended
    //                 school
    int semesterHours()
    {
        return nSemesterHours;
    }
};

int main(int nArgc, char* pszArgs[])
{
    // create a student object from the heap
    Student* pS = new Student;

    // add in a few grades
    pS->grade(2.5, 3);
```

```
    pS->grade(4.0, 3);
    pS->grade(3, 3);

    // now retrieve the current GPA
    cout << "Resulting GPA is " << pS->grade() << "\n";

    return 0;
}
```

This version of Student maintains two data members. dCombinedScore reflects the sum of the weighted grades, while nSemesterHours reflects the total number of semester hours completed. The function grade(double, int) updates both the sum of the weighted grades and the number of semester hours. Its namesake function, grade(), returns the current GPA, which it calculates as the ratio of the weighted grades and the total number of semester hours.

Tip

grade(double, int) adds the effect of a new course to the overall GPA whereas grade(void) returns the current GPA. This dichotomy of one function updating a value while the other simply returns it is very common.

A grade() function which returns the value of some data member is called an access function because it provides access to the data member.

While certainly not foolproof, the grade(double, int) function demonstrates a little of how a class can protect itself. The function runs a few rudimentary checks to make sure that the data being passed it is reasonable. The Student class knows that valid grades stretch from 0 to 4. Further, the class knows that the number of semester hours for one course lies between 0 and 5 (the upper range is my own invention).

The basic checks made by the grade() method, when added to the fact that the data members are not accessible by outside functions, guarantees a certain amount of data integrity.

Note

There is another access control level called *private*. The distinction between *private* and *protected* will become clearer when we discuss inheritance in Session 21.

The member function semesterHours() does nothing more than return the value of nSemesterHours.

A function that does nothing more than give external functions access to the value of a data member is called an *access function*. An access function enables nonmember functions to read the value of a data member without the capability to change it.

A function that can access the protected members of a class is called a *trusted function*. All member functions are trusted. Nonmember functions can also be designated as trusted using the `friendly` keyword. A function that is friendly to a class is trusted. All of the member functions of a friendly class are friendly. The proper use of `friendly` is beyond the scope of this book.

Static data members

No matter how many members we had protected, our `LinkList` class in Session 15 would still have been vulnerable to outside functions through the global head pointer. What we really want is to draw that pointer back into the protection of the class where we could make it protected. However, we cannot use a normal data member because these are created separately for each instance of `LinkList` — there can be only one head pointer for the entire linked list. C++ provides a solution in the format of static data member.

A *static data member* is one that is not instanced separately for each object. All objects of a given class share the same static data member.

The syntax for declaring a static data member is a bit tortured:

```
class LinkedList
{
  protected:
    // declare pHead to be a member of the class
    // but common to all objects
    static LinkedList* pHead;

    // the standard pNext pointer is instanced separately
    // for each object
    LinkedList* pNext;

    // addHead - add a data member to the beginning
    //           of the list
    void addHead()
    {
```

```
        // make the current entry point to the
        // current beginning of the list
        pNext = pHead;

        // make the current head pointer point to
        // the current object (this)
        pHead = this;
    }

    // ...whatever else...
};

// now allocate a memory location to house the static
// data memory; be sure to initialize the static here
// because the object constructor will not handle it
LinkedList* LinkedList::pHead = 0;
```

The static declaration in the class makes `pHead` a member of the class but does not allocate memory for it. That must be done outside of the class as shown.

The same function `addHead()` accesses `pHead` just as it would access any other data member. First, it points the current object's next pointer to the beginning of the list — the entry pointed at by `pHead`. Second, it changes the head pointer to point to the current entry.

Done!

Remember that the address of the "current entry" is referenced by the keyword `this`.

Tip

As simple as `addHead()` is, examine it very carefully: all objects of class `LinkedList` refer to the same `pHead` member, whereas each object has its own `pNext` pointer.

Note

It is also possible to declare a member function static; this book, however, does not cover such functions.

Review

The constructor is a special member function that C++ calls automatically when an object is created, whether it's because a local variable goes into scope or when an object is allocated off of the heap. It is the responsibility of the constructor to initialize the data members to a legal state. The data members of a class are constructed automatically before the class constructor is called. By comparison, C++ calls a special function known as the destructor when the object is to be destroyed.

- The class constructor gives the class control over how the object is to be created. This keeps the class object from starting life in an illegal state. Constructors are declared the same as other member functions except that they carry the same name as the class and have no return type (not even `void`).
- The class destructor gives the class a chance to return any resources allocated by the constructor. The most common such resource is memory.
- Declaring a member protected makes it inaccessible to untrusted member functions. Member functions are automatically considered trusted.

Quiz Yourself

1. What is a constructor? (See "The Constructor.")
2. What is wrong with calling a function `init()` to initialize an object when it is created? (See "The Constructor.")
3. What is the full name of a constructor for the class `Teacher`? What is its return type? (See "The Constructor.")
4. What is the order of construction for object data members? (See "Constructing Data Members.")
5. What is the full name of the destructor for the class `Teacher`? (See "The Destructor.")
6. What is the significance of the keyword `static` when it is used in connection with data members? (See "Static Data Members.")