SESSION

20

# Class Constructors II

## Session Checklist

✔ Creating constructors that have arguments

✔ Passing arguments to the constructors of data members

✔ Determining the order of construction of data members

✔ Examining the special properties of the copy constructor

**30 Min. To Go**

A very simple constructor worked well for the simple `Student` class in Session 19. However, the `Student` class would not become very complex before the limitations of such a constructor became clear. Consider, for example, that a student has a name and a social security number-based ID. The constructor in Session 19 had no arguments, so it had no choice but to initialize the object as "empty." The constructor is supposed to create a valid object — a no-named student with a null ID probably does not represent a valid student.

## Constructors with Arguments

C++ enables the programmer to define a constructor with arguments as shown in Listing 20-1.

**Listing 20-1**
*Defining a Constuctor with Arguments*

```
// NamedStudent - this program demonstrates how adding
//                arguments to the constructor makes
//                for a more realistic class
#include <stdio.h>
#include <iostream.h>
#include <string.h>

// Student
class Student
{
  public:
    Student(char *pszName, int nID)
    {
        // Store a copy of the person’s name in the object:
        // allocate from the heap the same
        // size string as that passed
        int nLength = strlen(pszName) + 1;
        this->pszName = new char[nLength];

        // now copy the name passed into the
        // newly allocated buffer
        strcpy(this->pszName, pszName);

        // finally save off the student id
        this->nID = nID;
    }

    ~Student()
    {
        delete pszName;
        pszName = 0;
    }

  protected:
    char* pszName;
    int   nID;
};

void fn()
{
    // create a local student
    Student  s1("Stephen Davis", 12345);

    // now create a student off of the heap
    Student* pS2 = new Student("Kinsey Davis", 67890);

    // be sure to return any objects allocated from
    // the heap
    delete pS2;
}
```

The `Student(char*, int)` constructor begins by allocating a block of heap memory to hold a copy of the string passed it. The function `strlen()` returns the number of bytes in a character string; however, because `strlen()` does not count the `null`, we have to add 1 to that count to calculate the amount of heap memory that is needed. The `Student` constructor copies the string passed it to this block of memory. Finally, the constructor assigns the student ID in the class.

The function `fn()` creates two `Student` objects, one local to the function and a second off of the heap. In both cases, `fn()` passes the name and ID number of the `Student` object being created.

Note

**In this `NamedStudent` example, the arguments to the `Student` constructor carry the same name as the data members that they initialize. This is not a requirement of C++, just a convention that I prefer. However, a locally defined variable has precedence over a data member of the same name. Thus, an unqualified reference within `Student()` to `pszName` refers to the argument; however, `this->pszName` always refers to the data member irrespective of any locally declared variables. While some find this practice confusing, I think it highlights the connection between the argument and the data member. In any case, you should be acquainted with this common practice.**

Constructors can be overloaded in the same way that functions with arguments can be overloaded.

Note

**Remember that to overload a function means to have two functions with the same name, which are differentiated by their different types of arguments.**

C++ chooses the proper constructor based on the arguments in the declaration. For example, the class Student can simultaneously have the three constructors shown in the following snippet:

```
#include <iostream.h>
#include <string.h>
class Student
{
  public:
    // the available constructors
    Student();
    Student(char* pszName);
    Student(char* pszName, int nID);
    ~Student();

  protected:
    char* pszName;
    int   nID;
};

//the following invokes each constructor in turn
int main(int nArgs, char* pszArgs[])
{
   Student noName;
   Student freshMan("Smel E. Fish");
   Student xfer("Upp R. Classman", 1234);
   return 0;
}
```

Because the object noName appears with no arguments, it is constructed using the constructor Student::Student(). This constructor is called the *default*, or *void*, *constructor*. (I prefer the latter name, but the former is more common so I use it in this book.)

It is often the case that the only difference between one constructor and another is the addition of a default value for the missing argument. Suppose, for example, that a student created without a student ID is given the ID of 0. To help avoid extra work, C++ enables the programmer to designate a default argument. I could have combined the latter two constructors as follows:

```
Student(char* pszName, int nID = 0);

Student s1("Lynn Anderson", 1234);
student s2("Stella Prater");
```

Both s1 and s2 are constructed using the same Student(char*, int) constructor. In Stella's case, a default value of 0 is supplied to the constructor.

Tip

**Default values for arguments can be supplied to any function; however, they find the greatest use in reducing the number of constructors.**

## Constructing Class Members

C++ constructs data member objects at the same time that it constructs the object itself. There is no obvious way to pass arguments when constructing member objects.

Note

**In the examples shown in Session 19, there was no need to pass arguments to the data member constructors — that version of Student relied on the application to call an init() function to initialize the object to some reasonable value.**

Consider Listing 20-2 in which the class Student includes an object of class StudentId. I outfitted both with WRITE statements in the constructor in order to see what's going on.

**Listing 20-2**
*Student Constructor with a member object of class StudentID*

```
// DefaultStudentId - create a student object using the
//                    the next available student ID
#include <stdio.h>
#include <iostream.h>
#include <string.h>
```

*Continued*

**Listing 20-2** *Continued*

```
// StudentId
int nNextStudentId = 0;
class StudentId
{
  public:
    StudentId()
    {
        nId = ++nNextStudentId;
        cout << “Assigning student id “ << value << “\n”;
    }

    ~StudentId()
    {
        cout << “Destructing student id “ << value << “\n”;
  protected:
    int nId;
};

// Student
class Student
{
  public:
    // constructor - create a student with the name
    //               supplied (use default of “no name”)
    Student(char *pszName  = “no name”)
    {
       cout << “Constructing student “ << pszName << “\n”;

       // copy the string passed to the local member
       this->pszName = new char[strlen(pszName) + 1];
       strcpy(this->pszName, pszName);
    }

    ~Student()
    {
        cout << “Destructing student “ << pszName << “\n”;
        delete pszName;
        pszName = 0;
    }

  protected:
    char* pszName;
    StudentId id;
};

int main(int nArgs, char* pszArg[])
{
   Student s("Randy");
   return 0;
}
```

A student ID is assigned to each student as the `Student` object is constructed. In this example, IDs are handed out sequentially using the global variable `nextStudentId` to store the next ID to be assigned.

The output from executing this simple program follows:

```
Assigning student id 1
Constructing student Randy
Destructing student Randy
Destructing student id 1
```

Notice that the message from the `StudentId` constructor appears before the output from the `Student` constructor and the message from the `StudentId` destructor appears after the output from the `Student` destructor.

Tip

**With all these constructors performing output, you might think that constructors must output something. Most constructors don't output a darned thing. Book constructors do because readers usually don't take the good advice provided by authors and single-step the programs.**

If the programmer does not provide a constructor, the default constructor provided by C++ automatically invokes the default constructors for any data members. The same is true come harvesting time. The destructor for the class automatically invokes the destructor for any data members that have destructors. The C++ provided destructor does the same.

Okay, this is all great for the default constructor. But what if we wanted to invoke a constructor other than the default? Where do we put the object? To demonstrate, let's assume that instead of calculating the student ID, it is provided to the Student constructor, which passes the ID to the constructor for class StudentId.

First, let's look at what doesn't work. Consider the program shown in Listing 20-3.

**Listing 20-3**
*An incorrect way to construct a data member object*

```
// FalseStudentId - the following attempts to create
//                  the student id data member using
//                  a non-default constructor.

#include <stdio.h>
#include <iostream.h>
#include <string.h>

class StudentId
{
  public:
    StudentId(int nId = 0)
    {
        this->nId = nId;
        cout << "Assigning student id " << nId << "\n";
    }

    ~StudentId()
    {
        cout << "Destructing student id " << nId << "\n";
    }
  protected:
    int nId;
};

class Student
{
  public:
    // constructor - create an object with a specified
    //               name and student id
    Student(char* pszName  = "no name", int ssId = 0)
    {
       cout << “Constructing student “ << pszName << “\n”;
       this->pszName = new char[strlen(pszName) + 1];
       strcpy(this->pszName, pszName);

       //don’t try this at home kids. It doesn’t work
       StudentId id(ssId);    //construct a student id
    }

    ~Student()
    {
        cout << “Destructing student “ << this->pszName << “\n”;
        delete this->pszName;
        pszName = 0;
    }
  protected:
    char* pszName;
    StudentId id;
};

int main()
{
    Student s(“Randy”, 1234);
    cout << “This message from main\n”;
    return 0;
}
```

The constructor for `StudentId` was changed to accept a value externally (the default value is necessary to get the example to compile, for reasons that become clear shortly). Within the constructor for `Student`, the programmer (that’s me) has (cleverly) attempted to construct a `StudentId` object named `id`.

The output from this program shows a problem:

```
Assigning student id 0
Constructing student Randy
Assigning student id 1234
Destructing student id 1234
This message from main
Destructing student Randy
Destructing student id 0
```

Part IV–Saturday Evening
Session 20

First, the constructor appears to be invoked twice, the first time with zero before the Student constructor begins and a second time with the expected 1234 from within the Student constructor. Apparently, a second StudentId object has been created within the constructor for Student different from the StudentId data member

The explanation for this rather bizarre behavior is that the data member id already exists by the time the body of the constructor is entered. Rather than constructing an existing data member id, the declaration provided in the constructor creates a local object of the same name. This local object is destructed upon returning from the constructor.

Somehow we need a different mechanism to indicate “construct the existing member; don’t create a new one.” C++ defines the new construct as follows:

```
class Student
{
  public:
    Student(char* pszName  = “no name”, int ssId = 0)
        : id(ssId) // construct the data member id to the
                   // specified value
    {
       cout << “Constructing student “ << pszName << “\n”;
       this->pszName = new char[strlen(pszName) + 1];
       strcpy(this->pszName, pszName);
    }
    // ...
};
```

Take particular notice of the first line of the constructor. We haven’t seen this before. The : means that what follows are calls to the constructors of data members of the current class. To the C++ compiler, this line reads: “Construct the member id using the argument ssId of the Student constructor. Whatever data members are not called out in this fashion are constructed using the default constructor.”

**The entire program appears on the accompanying CD-ROM with the name StudentId.**

This new program generates the expected result:

```
Assigning student id 1234
Constructing student Randy
This message from main
Destructing student Randy
Destructing student id 1234
```

The object `s` is created in `main()` with a student name of “Randy” and a student id of 1234. This object is automatically destructed upon reaching the end of `main()`.

The : syntax must also be used to assign values to `const` or reference type members. Consider the following silly class:

```
class SillyClass
{
  public:
    SillyClass(int& i) : nTen(10), refI(i)
    {
    }
  protected:
    const int nTen;
    int& refI;
};
```

By the time that the program begins executing the code contained in the body of the constructor, the data members `nTen` and `refI` are already created. These data member types must be initialized before control enters the constructor body.

Tip

**Any data member can be declared using the preceding syntax.**

## Order of Construction

When there are multiple objects, all with constructors, the programmer usually doesn’t care about the order in which things are built. If one or more of the constructors have side effects, however, the order can make a difference.

Note

**A *side effect* is some change in state caused by a function but not part of the arguments or the returned object. For example, if a function assigns a value to a global variable, that would be a side effect.**

The rules for the order of construction are:

- Locals and static objects are constructed in the order in which their declarations are invoked.
- Static objects are constructed only once.
- All global objects are constructed before `main()`.
- Global objects are constructed in no particular order.
- Members are constructed in the order in which they are declared in the class.

Let's consider each rule in turn.

## *Local objects are constructed in order*

Local objects are constructed in the order in which the program encounters their declaration. Normally, this is the same as the order in which the objects appear in the function, unless your function jumps around particular declarations. (By the way, jumping around declarations is a bad thing to do. It confuses the reader and the compiler.)

## *Static objects are constructed only once*

Static objects are similar to other local variables, except they are constructed only once. This is to be expected because they retain their value from one invocation of the function to the next. However, unlike C, which is free to initialize statics when the program begins, C++ must wait until the first time control passes through the static's declaration to perform the construction. Consider the following trivial program:

```
class DoNothing
{
  public:
   DoNothing(int initial)
   {
   }
};
```

```
void fn(int i)
{
   static DoNothing staticDN(1);
    DoNothing localDN(2);
}
```

The variable staticDN is constructed the first time that fn() is called. The second time that fn() is invoked, the program only constructs localDN.

## All global objects are constructed before main()

All global variables go into scope as soon as the program starts. Thus, all global objects are constructed before control is passed to main().

## Global objects are constructed in no particular order

Figuring out the order of construction of local objects is easy. An order is implied by the flow of control. With globals, there is no such flow to give order. All globals go into scope simultaneously, remember? Okay, you argue, why can't the compiler just start at the top of the file and work its way down the list of global objects? That would work fine for a single file (and I presume that's what most compilers do).

Unfortunately, most programs in the real world consist of several files that are compiled separately and then built together. Because the compiler has no control over the order in which these files are linked, it cannot affect the order in which global objects are constructed from file to file.

Most of the time this is ho-hum stuff. Occasionally, though, it can generate bugs that are extremely difficult to track down. (It happens just often enough to make it worth mentioning in a book.)

Consider this example:

```
class Student
{
  public:
    Student (unsigned id) : studentId(id)
    {
    }
    const unsigned studentId;
};

class Tutor
```

```
{
  public:
    // constructor - assign the tutor to a student
    //               by recording the student's id
    Tutor(Student &s)
    {
        tutoredId = s.studentId;
    }
  protected:
    unsigned tutoredId;
};

// create a student globally
Student randy(1234);

//assign that student a tutor
Tutor   jenny(randy);
```

Here the constructor for `Student` assigns a student ID. The constructor for `Tutor` records the ID of the student to help. The program declares a student `randy` and then assigns that student a tutor `jenny`.

The problem is that we are making the implicit assumption that `randy` gets constructed before `jenny`. Suppose that it was the other way around. Then `jenny` would get constructed with a block of memory that had not yet been turned into a `Student` object and, therefore, had garbage for a student ID.

Tip

**This example is not too difficult to figure out and more than a little contrived. Nevertheless, problems deriving from global objects being constructed in no particular order can appear in subtle ways. To avoid this problem, don't allow the constructor for one global object to refer to the contents of another global object.**

## *Members are constructed in the order in which they are declared*

Members of a class are constructed according to the order in which they are declared within the class. This is not quite as obvious as it might sound. Consider this example:

```
class Student
{
  public:
   Student (unsigned id, unsigned age) : sAge(age), sId(id)
   {
   }
   const unsigned sId;
   const unsigned sAge;
};
```

In this example, sId is constructed before sAge even though it appears second in the constructor's initialization list. The only time you could probably detect any difference in the construction order is if both of these were members of classes that had constructors and these constructors had some mutual side effect.

### Destructors are invoked in the reverse order of the constructors

Finally, no matter in what order the constructors kick off, you can be assured that the destructors are invoked in the reverse order. (It's nice to know that there's at least one rule in C++ that has no if's, and's, or but's.)

**10 Min.
To Go**

## The Copy Constructor

A copy constructor is a constructor that has the name X::X(X&), where X is any class name. That is, it is the constructor of class X which takes as its argument a reference to an object of class X. Now, I know that this sounds really useless, but think for a moment about what happens when you call a function similar to the following:

```
void fn1(Student fs)
{
   //...
}
int fn2()
{
   Student ms;
   fn1(ms);
   return 0;
}
```

As you know, a copy of the object ms — and not the object itself — is passed to the function fn1(). C++ could just make an exact copy of the object and pass that to fn1().

This is not acceptable in C++. First, as I have pointed out, it takes a constructor to create an object, even a copy of an existing object. Second, what if we don't want a simple copy of the object? (Let's ignore the "why?" of this for a little while.) We need to be able to specify how the copy should be constructed.

The CopyStudent program shown in Listing 20-4 demonstrates the point.

**Listing 20-4**
*Copy Constructor in Action*

```
// CopyStudent - demonstrate a copy constructor by
//               passing an object by value
#include <stdio.h>
#include <iostream.h>
#include <string.h>

class Student
{
  public:
    // create an initialization function
    void init(char* pszName, int nId, char* pszPreamble = "\0")
    {
        int nLength = strlen(pszName)
                    + strlen(pszPreamble)
                    + 1;
        this->pszName = new char[nLength];
        strcpy(this->pszName, pszPreamble);
        strcat(this->pszName, pszName);
        this->nId = nId;
    }

    //conventional constructor
    Student(char* pszName  = "no name", int nId = 0)
    {
       cout << "Constructing new student " << pszName << "\n";
       init(pszName, nId);
    }
    //copy constructor
    Student(Student &s)
    {
       cout << “Constructing Copy of “
            << s.pszName
            << “\n”;
       init(s.pszName, s.nId, “Copy of “);
    }

    ~Student()
    {
       cout << “Destructing “ << pszName << “\n”;
       delete pszName;
    }
  protected:
    char* pszName;
    int   nId;
};

//fn - receives its argument by value
void fn(Student s)
{
    cout << “In function fn()\n”;
}

int main(int nArgs, char* pszArgs[])
{
    // create a student object
    Student randy(“Randy”, 1234);

    // now pass it by value to fn()
    cout << “Calling fn()\n”;
    fn(randy);
    cout << “Returned from fn()\n”;

    // done
    return 0;
}
```

The output from executing this program is:

```
Constructing new student Randy
Calling fn()
Constructing Copy of Randy
In function fn()
Destructing Copy of Randy
Returned from fn()
Destructing Randy
```

Starting with main(), we can see how this program works. The normal constructor generates the first message. main() generates the "Calling . . ." message. C++ calls the copy constructor to make a copy of randy to pass to fn(), which generates the next line of output. The copy is destructed at the return from fn(). The original object, randy, is destructed at the end of main().

(This copy constructor does a little bit more than just make a copy of the object; it tacks the phrase Copy of to the front of the name. That was for your benefit. Normally, copy constructors should restrict themselves to just making copies. But, if the truth be known, they can do anything they want.)

## *Shallow copies versus deep copies*

C++ considers the copy constructor to be so important that if you don't provide one with your class, C++ creates one for you. The default copy constructor that C++ provides performs a member-by-member copy.

Performing a member-by-member copy seems the obvious thing to do in a copy constructor. Other than adding the capability to tack silly things such as Copy of to the front of students' names, when would we ever want to do anything but a member-by-member copy?

Consider what happens if the constructor allocates an asset such as memory off the heap. If the copy constructor simply makes a copy of that asset without allocating its own, we end up with a troublesome situation: two objects thinking they have exclusive access to the same asset. This becomes nastier when the destructor is invoked for both objects and they both try to put the same asset back. To make this more concrete, consider our Student class again:

```
class Student
{
  public:
    Person(char *pszName)
    {
        pszName = new char[strlen(pszName) + 1];
        strcpy(pName, pN);
    }
    ~Person()
    {
      delete pszName;
    }
  protected:
   char* pszName;
};
```

Here, the constructor allocates memory off the heap to store the person's name — the destructor dutifully puts this heap memory back as it should. Were this object to be passed to a function by value, C++ would make a copy of the `Student` object, including the pointer `pszName`. The program now has two `Student` objects both pointing to the same block of memory containing the student's name. When the first object is destructed, the block is returned to memory. Destructing the second object results in the same memory being returned to the stack twice — a fatal act for a program to perform.

Tip

**Heap memory is not the only asset that requires a deep copy constructor, but it is the most common. Open files, ports, and allocated hardware (such as printers) also require deep copies. These are the same types of assets that destructors must return. Thus, a good rule of thumb is that if your class requires a destructor to deallocate assets, it also requires a copy constructor.**

### *A "fall back" copy constructor*

There are times when you don't want copies of your object being created. This could be because you cannot or don't want to write the code to perform a deep copy. This is also the case when the object is very large and making a copy, whether shallow or deep, could take a considerable amount of time.

A simple fall back position to avoid the problem of shallow copies being created without your knowledge is to create a protected copy constructor.

Note

**If you do not create a copy constructor, C++ creates one for you.**

In the following example, C++ cannot make a copy of the BankAccount class, thereby ensuring that the program does not mistakenly make a deposit to or withdrawal from a copy of the account and not the account itself.

```
class BankAccount
{
   protected:
    int nBalance;
    BankAccount(BankAccount& ba)
    {
    }

  public:
    // ...remainder of class...
};
```

***Done!***

## Review

The constructor is designed to create an object in an initial, valid state; however, many objects do not have a valid default state. For example, it is not possible to create a valid Student object without a name and a student ID. A constructor with arguments enables the parent program to pass initial values for the newly created object. These arguments may also be passed along to any constructors for data members of the class. The order of construction of these members is well defined in the C++ standard (one of the few things that is well defined).

- Arguments to constructors enable the initial state of the object to be determined by the parent program; however, in the event that these initial arguments are illegal, the class must have a “fall back” state.
- Care must be taken with constructors that have side effects, such as changing the value of global variables, because one or more constructors can clash when constructing an object with data members which are objects in their own right.
- A special constructor, the copy constructor, has a prototype of X::X(X&) where X is the name of some class. This class is used to create a copy of an existing object. Copy constructors are extremely important if the object contains a resource that is returned to the resource pool in the destructor.

Although the principles of object-based programming have been explored, we still aren’t where we want to be. We’ve built a microwave oven, we’ve built a box around the working parts, and we've defined a user interface; but we haven't drawn a relationship between microwave ovens and other types of ovens. This is the essence of object-oriented programming, which is discussed in Part 5.

## QUIZ YOURSELF

1. Why would a class such as `Student` have a constructor in the form `Student(char* pszName, int nId)`? (See "Constructors with Arguments.")
2. How can you tell what order global objects are constructed? (See "Global Objects Are Constructed in No Particular Order.")
3. Why would a class need a copy constructor? (See "The Copy Constructor.")
4. What is the difference between a shallow copy and a deep copy? (See "Shallow Copies versus Deep Copies.")

## Exercises

### Problems

Write a copy constructor and a destructor for the following `LinkedList` class:

```
#include <stdio.h>
#include <iostream.h>
#include <string.h>

class LinkedList
{
  protected:
    LinkedList* pNext;
    static LinkedList* pHead;

    char* pszName;

  public:
    // constructor - copy the name provided
    LinkedList(char* pszName)
```

```
        {
            int nLength = strlen(pszName) + 1;
            this->pszName = new char[nLength];
            strcpy(this->pszName, pszName);
            pNext = 0;
        }

        // assume that these two member functions exist
        void addToList();
        void removeFromeList();

        // destructor -
        ~LinkedList()
        {
            // ...what goes here?
        }
        LinkedList(LinkedList& l)
        {
            // ...and here?
        }
    };
```

## Hint

1. Consider what happens if the object is still in the linked list when it is destructed.
2. Remember that the memory allocated off the heap should be returned before the pointer is lost.

## My solution

This is my version of the destructor and copy constructor:

```
        // destructor - remove the object and
        //              remove the entry
        ~LinkedList()
        {
            // if the current object is in a list...
            if (pNext)
            {
                // ...then remove it
                removeFromList();
            }

            // if the object has a block of memory...
            if (pszName)
            {
                // ...return it to the heap
                delete pszName;
            }
            pszName = 0;
        }

        // copy constructor - called to create a copy of
        //                      an existing object
        LinkedList(LinkedList& l)
        {
            // allocate a block of the same size
            // off of the heap
            int nLength = strlen(l.pszName) + 1;
            this->pszName = new char[nLength];

            // now copy the name into this new block
            strcpy(this->pszName, l.pszName);

            // zero out the length as long as the object
            // is not in a linked list
            pNext = 0;
        }
```

If the object is a member of a linked list, then the destructor must remove it before the object is reused and the pNext pointer is lost. In addition, if the object "owns" a block of heap memory, it must also be returned. Similarly, the copy constructor performs a deep copy by allocating a new block of memory off the heap to hold the object name. The copy constructor does not add the object to the existing linked list (although it could).