More about Classes

Tony Gaddis

Ch 14

Instance and Static Members

CONCEPT: Each instance of a class has its own copies of the class’s instance variables. If a member variable is declared static, however, all instances of that class have access to that variable. If a member function is declared static, it may be called without any instances of the class being defined.

Instance and Static Members

instance variable: a member variable in a class. Each object has its own copy.

`static` variable: one variable shared among all objects of a class

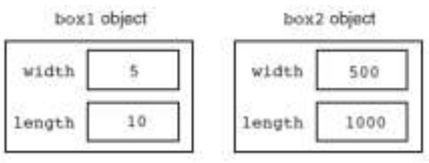
`static` member function: can be used to access `static` member variable; can be called before any objects are defined

Instance Variables

Each class object (an instance of a class) has its own copy of the class's member variables.

An object's member variables are separate and distinct from the member variables of other objects of the same class.

```
Rectangle box1, box2;

// Set the width and length for box1.
box1.setWidth(5);
box1.setLength(10);

// Set the width and length for box2.
box2.setWidth(500);
box2.setLength(1000);
```



Static Members

It is possible to create a member variable or member function that does not belong to any instance of a class. Such members are known as a *static member variables and static member functions.*

Static Member Variables

When a member variable is declared with the key word static, there will be only one copy of the member variable in memory, regardless of the number of instances of the class that might exist. A single copy of a class's static member variable is shared by all instances of the class.

For example, the following Tree class uses a static member variable to keep count of the number of instances of the class that are created.

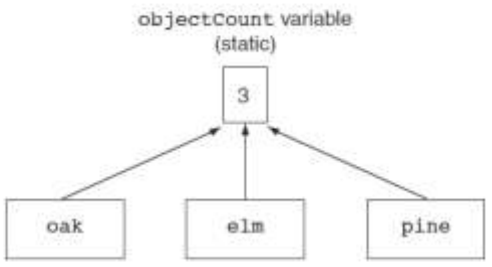
Contents of `Tree.h`

Static member declared here.

```
// Tree class
class Tree
{
private:
   static int objectCount;     // Static member variable.
public:
   // Constructor
   Tree()
      { objectCount++; }

   // Accessor function for objectCount
   int getObjectCount() const
      { return objectCount; }
};

// Definition of the static member variable, written
// outside the class.
int Tree::objectCount = 0;
```

Static member defined here.

Three Instances of the Tree Class, But Only One `objectCount` Variable



Instances of the `Tree` class

Static Member Functions

You declare a static member function by placing the static keyword in the function's prototype.

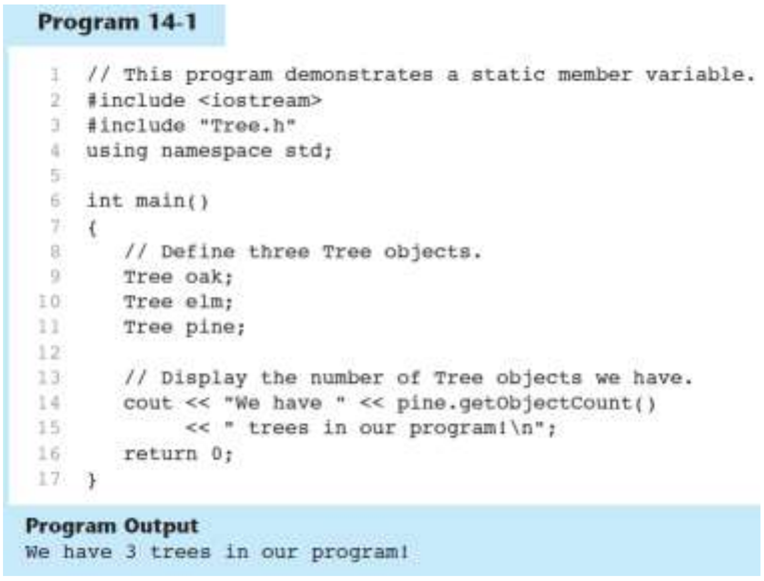
```
static ReturnType FunctionName (ParameterTypeList);
```

- Declared with `static` before return type:
  ```
  static int getObjectCount() const
     { return objectCount; }
  ```

- Static member functions can only access static member data

Can be called independent of objects:

```
int num = Tree::getObjectCount();
```



Program 14-1

```
// This program demonstrates a static member variable.
#include <iostream>
#include "Tree.h"
using namespace std;

int main()
{
   // Define three Tree objects.
   Tree oak;
   Tree elm;
   Tree pine;

   // Display the number of Tree objects we have.
   cout << "We have " << pine.getObjectCount()
        << " trees in our program!\n";
   return 0;
}
```

Program Output

```
We have 3 trees in our program!
```

Friends of Classes

SKIP

Friend: a function or class that is not a member of a class, but has access to private members of the class

A friend function can be a stand-alone function or a member function of another class

It is declared a friend of a class with `friend` keyword in the function prototype

```
friend ReturnType FunctionName (ParameterTypeList)
```

Checkpoint

14.1 What is the difference between an instance member variable and a static member variable?

14.2 Static member variables are declared inside the class declaration. Where are static member variables defined?

14.3 Does a static member variable come into existence in memory before, at the same time as, or after any instances of its class?

14.4 What limitation does a static member function have?

14.5 What action is possible with a static member function that isn't possible with an instance member function?

Memberwise Assignment

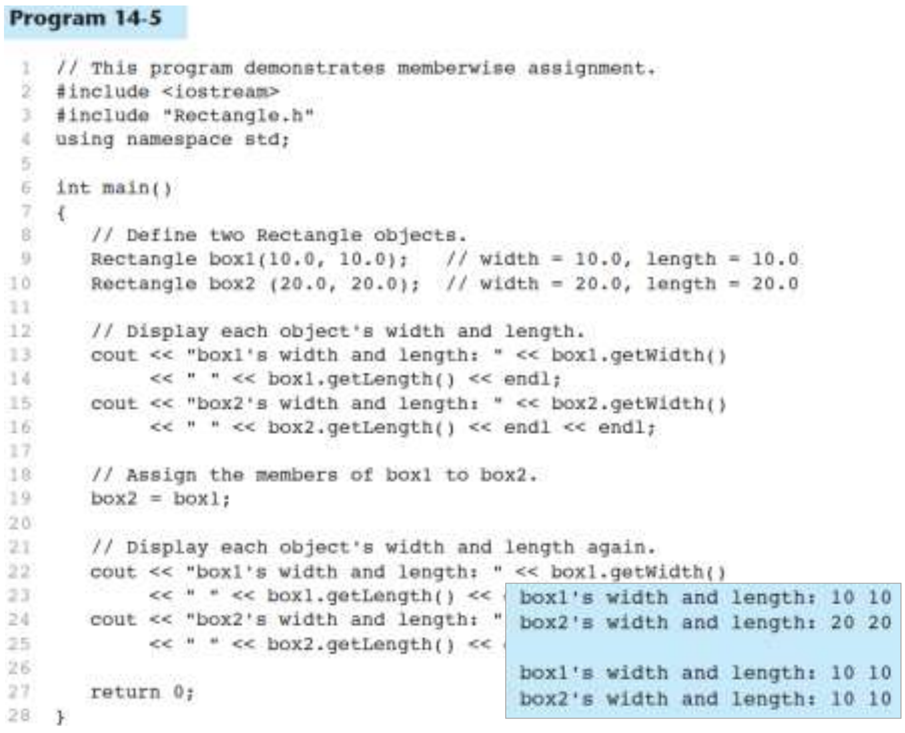
CONCEPT: The = operator may be used to assign one object's data to another object, or to initialize one object with another object's data. By default, each member of one object is copied to its counterpart in the other object.

instance2 = instance1;

means: copy all member values from instance1 and assign to the corresponding member variables of instance2

Can be used at initialization:

Rectangle r2 = r1;



Program 14-5

```
// This program demonstrates memberwise assignment.
#include <iostream>
#include "Rectangle.h"
using namespace std;

int main()
{
   // Define two Rectangle objects.
   Rectangle box1(10.0, 10.0);    // width = 10.0, length = 10.0
   Rectangle box2 (20.0, 20.0);   // width = 20.0, length = 20.0

   // Display each object's width and length.
   cout << "box1's width and length: " << box1.getWidth()
        << " " << box1.getLength() << endl;
   cout << "box2's width and length: " << box2.getWidth()
        << " " << box2.getLength() << endl << endl;

   // Assign the members of box1 to box2.
   box2 = box1;

   // Display each object's width and length again.
   cout << "box1's width and length: " << box1.getWidth()
        << " " << box1.getLength() <<
   cout << "box2's width and length: "
        << " " << box2.getLength() <<

   return 0;
}
```

```
box1's width and length: 10 10
box2's width and length: 20 20

box1's width and length: 10 10
box2's width and length: 10 10
```

Copy Constructors

CONCEPT: A copy constructor is a special constructor that is called whenever a new object is created and initialized with another object’s data.

Most of the time, the default memberwise assignment behavior in C++ is perfectly acceptable.

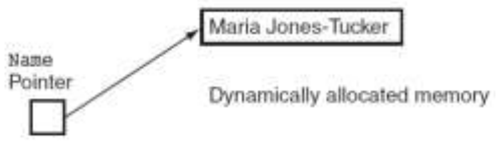
There are instances, however, where memberwise assignment cannot be used. For example, consider the following class.

Contents of `PersonInfo.h` (Version 1)

```cpp
#include <cstring>

class PersonInfo
{
private:
   char *name;
   int age;

public:
   PersonInfo(char *n, int a)
      { name = new char[strlen(n) + 1];
        strcpy(name, n);
        age = a; }

   ~PersonInfo()
      { delete [] name; }

   const char *getName()
      { return name; }

   int getAge()
      { return age; }
};
```

A potential problem with this class lies in the fact that one of its members, name, is a pointer. The constructor performs a critical operation with the pointer: it dynamically allocates a section of memory and copies a string to it. For instance, the following statement creates a personInfo object named person1, whose name member references dynamically allocated memory holding the string "Maria Jones-Tucker":

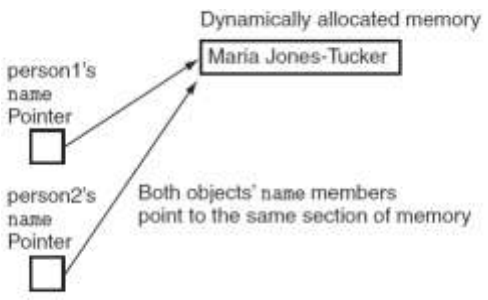
```
PersonInfo person1("Maria Jones-Tucker", 25);
```



Consider what happens when another PersonInfo object is created and initialized with the person1 object, as in the following statement:

```
PersonInfo person2 = person1;
```

In the statement above, person2's constructor isn't called. Instead, memberwise assignment takes place, copying each of person1's member variables into person2. This means that a separate section of memory is not allocated for person2's name member. It simply gets a copy of the address stored in person1's name member. Both pointers will point to the same address,


Dynamically allocated memory
Maria Jones-Tucker
person1's
name
Pointer
person2's
name
Pointer
Both objects' name members
point to the same section of memory

The solution to this problem is to create a *copy constructor for the object. A copy constructor* is a special constructor that's called when an object is initialized with another object's data. It has the same form as other constructors, except it has a reference parameter of the same class type as the object itself. For example, here is a copy constructor for the PersonInfo class:

```
PersonInfo(PersonInfo &obj)
{
    name = new char[strlen(obj.name) + 1];
    strcpy(name, obj.name);
    age = obj.age;
}
```

NOTE: C++ requires that a copy constructor's parameter be a reference object.

Using const Parameters in Copy Constructors

Because copy constructors are required to use reference parameters, they have access to their argument's data. Since the purpose of a copy constructor is to make a copy of the argument, there is no reason the constructor should modify the argument's data. With this in mind, it's a good idea to make copy constructors' parameters constant by specifying the const key word in the parameter list.

```
PersonInfo(const PersonInfo &obj)
{
      name = new char[strlen(obj.name) + 1];
      strcpy(name, obj.name);
      age = obj.age;
}
```

The Default Copy Constructor

If a class doesn't have a copy constructor, C++ creates a *default copy constructor for it. The default* copy constructor performs the memberwise assignment discussed in the previous section.

Operator Overloading

Assume that a class named Date exists, and objects of the Date class hold the month, day, and year in member variables. Suppose the Date class has a member function named add. The add member function adds a number of days to the date, and adjusts the member variables if the date goes to another month or year. For example, the following statement adds five days to the date stored in the today object:

```
today.add(5);
```

Although it might be obvious that the statement is adding five days to the date stored in today, the use of an operator might be more intuitive. For example, look at the following statement:

```
today += 5;
```

This statement uses the standard += operator to add 5 to today. This behavior does not happen automatically, however. The += operator must be *overloaded for this action to* occur.

Operator Overloading

Operators such as =, +, and others can be redefined when used with objects of a class

The name of the function for the overloaded operator is `operator` followed by the operator symbol, *e.g.*,

`operator+` to overload the + operator, and

`operator=` to overload the = operator

Prototype for the overloaded operator goes in the declaration of the class that is overloading it

Overloaded operator function definition goes with other member functions

Overloading the = Operator

Prototype:

$$\underset{\substack{\uparrow\\ \text{return type}}}{\texttt{void}}\ \underbrace{\texttt{operator=}}_{\substack{\uparrow\\ \text{function name}}}\underbrace{\texttt{(const SomeClass \&rval)}}_{\substack{\uparrow\\ \text{parameter for object on right side of operator}}}$$

Operator is called via object on left side

Invoking an Overloaded Operator

Operator can be invoked as a member function:

```
object1.operator=(object2);
```

It can also be used in more conventional manner:

```
object1 = object2;
```

Returning a Value

Overloaded operator can return a value

```
class Point2d
{
  public:
    double operator-(const point2d &right)
    { return sqrt(pow((x-right.x),2)
                + pow((y-right.y),2)); }
...
  private:
    int x, y;
};

Point2d point1(2,2), point2(4,4);
// Compute and display distance between 2 points.
cout << point2 - point1 << endl; // displays
  2.82843
```

The `this` Pointer

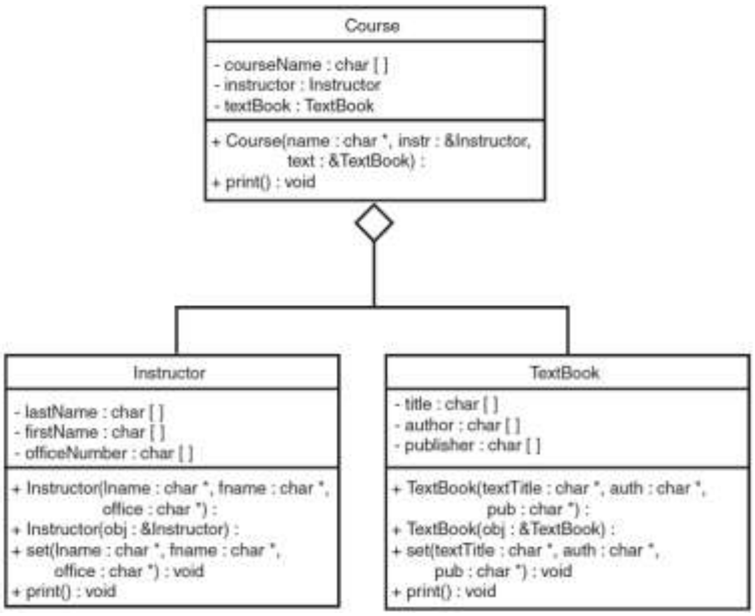
`this`: predefined pointer available to a class's member functions
Always points to the instance (object) of the class whose function is being called
Is passed as a hidden argument to all non-static member functions
Can be used to access members that may be hidden by parameters with same name

```
class SomeClass
{
   private:
          int num;
   public:
          void setNum(int num)
          { this->num = num; }
          ...
};
```

Aggregation

CONCEPT: Aggregation occurs when a class contains an instance of another class.

Aggregation in UML Diagrams


Course
- courseName : char [ ]
- instructor : Instructor
- textBook : TextBook
+ Course(name : char *, instr : &Instructor, text : &TextBook) :
+ print() : void
Instructor
- lastName : char [ ]
- firstName : char [ ]
- officeNumber : char [ ]
+ Instructor(lname : char *, fname : char *, office : char *) :
+ Instructor(obj : &Instructor) :
+ set(lname : char *, fname : char *, office : char *) : void
+ print() : void
TextBook
- title : char [ ]
- author : char [ ]
- publisher : char [ ]
+ TextBook(textTitle : char *, auth : char *, pub : char *) :
+ TextBook(obj : &TextBook) :
+ set(textTitle : char *, auth : char *, pub : char *) : void
+ print() : void

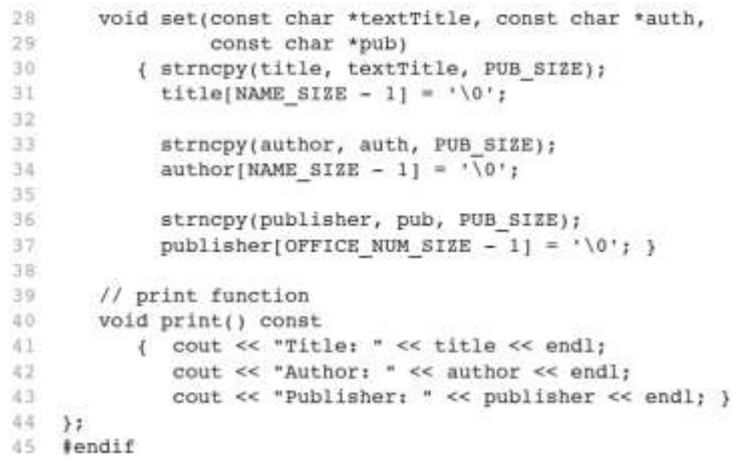
Contents of `Instructor.h`

```
#ifndef INSTRUCTOR
#define INSTRUCTOR
#include <iostream>
#include <cstring>
using namespace std;

// Constants for array sizes
const int NAME_SIZE = 51;
const int OFFICE_NUM_SIZE = 21;

// Instructor class
class Instructor
{
private:
   char lastName[NAME_SIZE];           // Last name
   char firstName[NAME_SIZE];          // First name
   char officeNumber[OFFICE_NUM_SIZE]; // Office number
public:
   // The default constructor stores empty strings
   // in the char arrays.
   Instructor()
      { set("", "", ""); }

   // Constructor
   Instructor(char *lname, char *fname, char *office)
```

```
      { set(lname, fname, office); }

   // set function
   void set(const char *lname, const char *fname,
            const char *office)
      { strncpy(lastName, lname, NAME_SIZE);
        lastName[NAME_SIZE - 1] = '\0';

        strncpy(firstName, fname, NAME_SIZE);
        firstName[NAME_SIZE - 1] = '\0';

        strncpy(officeNumber, office, OFFICE_NUM_SIZE);
        officeNumber[OFFICE_NUM_SIZE - 1] = '\0'; }

   // print function
   void print() const
      {  cout << "Last name: " << lastName << endl;
         cout << "First name: " << firstName << endl;
         cout << "Office number: " << officeNumber << endl; }
};
#endif
```

Contents of `TextBook.h`

```
#ifndef TEXTBOOK
#define TEXTBOOK
#include <iostream>
#include <cstring>
using namespace std;

// Constant for array sizes
const int PUB_SIZE = 51;

// TextBook class
class TextBook
{
private:
   char title[PUB_SIZE];     // Book title
   char author[PUB_SIZE];    // Author name
   char publisher[PUB_SIZE]; // Publisher name
public:
   // The default constructor stores empty strings
   // in the char arrays.
   TextBook()
      { set("", "", ""); }

   // Constructor
   TextBook(char *textTitle, char *auth, char *pub)
      { set(textTitle, auth, pub); }

   // set function
```



```
    void set(const char *textTitle, const char *auth,
             const char *pub)
       { strncpy(title, textTitle, PUB_SIZE);
         title[NAME_SIZE - 1] = '\0';

         strncpy(author, auth, PUB_SIZE);
         author[NAME_SIZE - 1] = '\0';

         strncpy(publisher, pub, PUB_SIZE);
         publisher[OFFICE_NUM_SIZE - 1] = '\0'; }

    // print function
    void print() const
       {  cout << "Title: " << title << endl;
          cout << "Author: " << author << endl;
          cout << "Publisher: " << publisher << endl; }
};
#endif
```

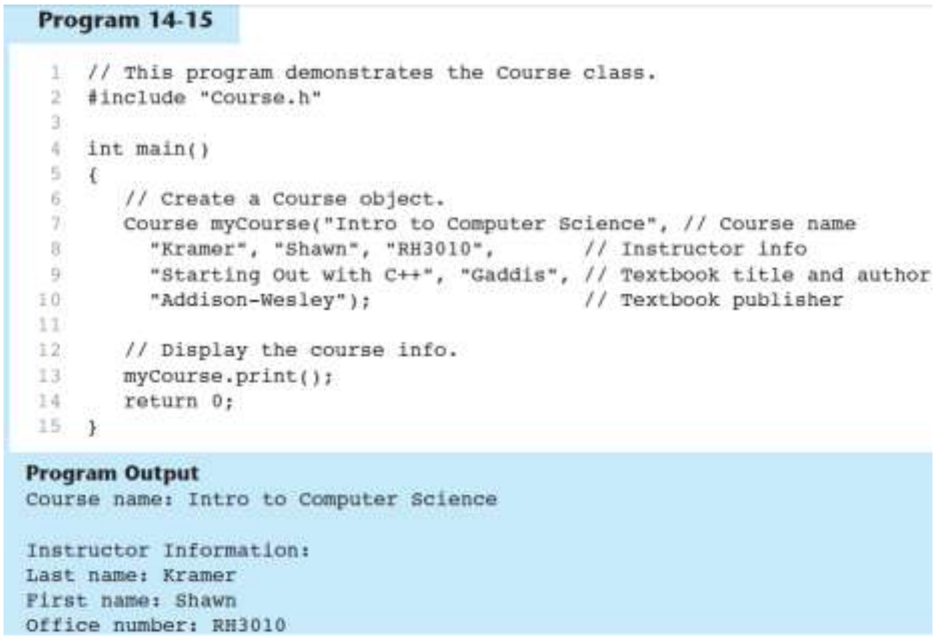
When an instance of one class is a member of another class, it is said that there is a “has a” relationship between the classes. For example, the relationships that exist among the Course, Instructor, and TextBook classes can be described as follows:

- The course *has an instructor.*
- The course *has a textbook.*

Contents of `Course.h`

```
#ifndef COURSE
#define COURSE
#include <iostream>
#include <cstring>
#include "Instructor.h"
#include "TextBook.h"
using namespace std;

// Constant for course name
const int COURSE_SIZE = 51;

class Course
{
private:
   char courseName[COURSE_SIZE];  // Course name
   Instructor instructor;         // Instructor
   TextBook textbook;             // Textbook
```

```
public:
   // Constructor
   Course(const char *course, const char *instrLastName,
          const char *instrFirstName, const char *instrOffice,
          const char *textTitle, const char *author,
          const char *publisher)
      { // Assign the course name.
        strncpy(courseName, course, COURSE_SIZE);
        courseName[COURSE_SIZE - 1] = '\0';

        // Assign the instructor.
        instructor.set(instrLastName, instrFirstName, instrOffice);

        // Assign the textbook.
        textbook.set(textTitle, author, publisher); }

   // print function
   void print() const
      {  cout << "Course name: " << courseName << endl << endl;
         cout << "Instructor Information:\n";
         instructor.print();
         cout << "\nTextbook Information:\n";
         textbook.print();
         cout << endl; }
};
```



Program 14-15

```
// This program demonstrates the Course class.
#include "Course.h"

int main()
{
   // Create a Course object.
   Course myCourse("Intro to Computer Science", // Course name
     "Kramer", "Shawn", "RH3010",        // Instructor info
     "Starting Out with C++", "Gaddis", // Textbook title and author
     "Addison-Wesley");                  // Textbook publisher

   // Display the course info.
   myCourse.print();
   return 0;
}
```

Program Output

```
Course name: Intro to Computer Science

Instructor Information:
Last name: Kramer
First name: Shawn
Office number: RH3010
```