ITM 312, Fall 2013

Miscellaneous Topics Lecture Notes

14.1. Instance and Static Members

a. Definitions

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

      1. Default variable modifier for classes

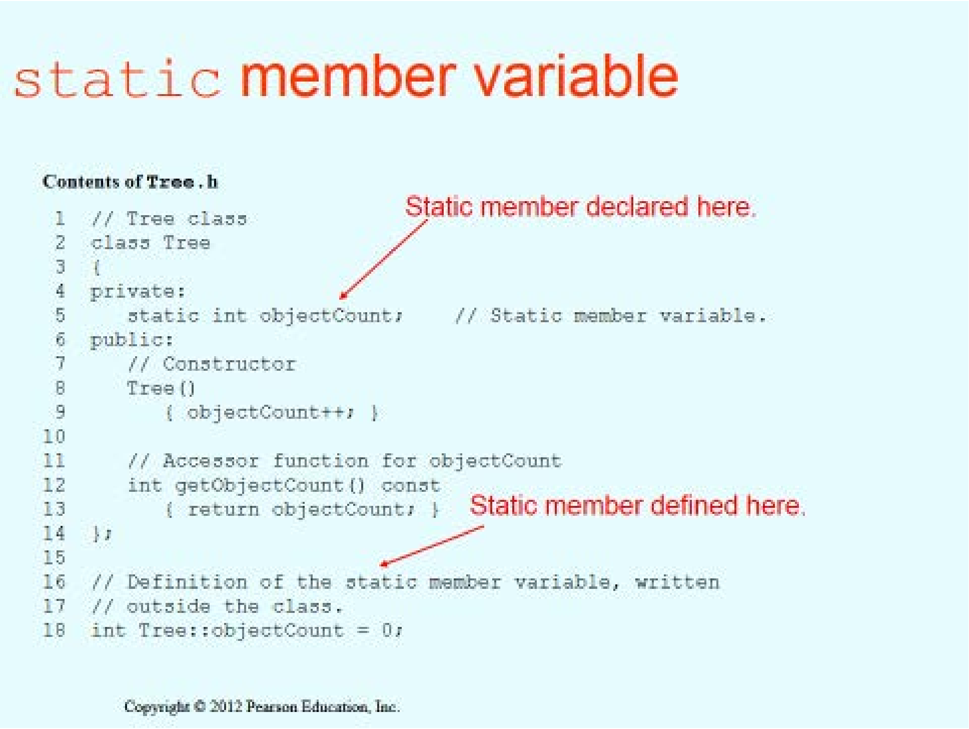
   ii. `static` variable: one variable shared among all objects of a class

      1. if you change a static variable in one object, it gets changed in all objects

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

      1. cannot access any instance variables

b. `static` member variable example:



`static` member variable

Contents of `Tree.h`

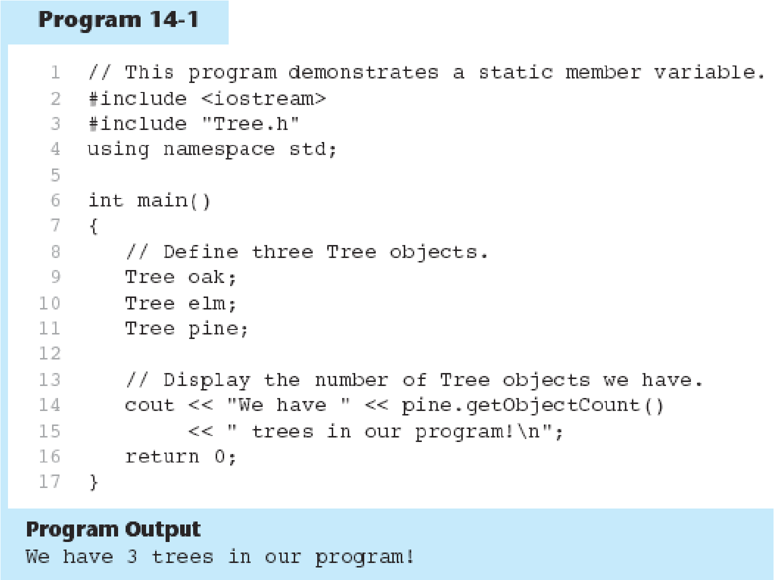
```
// Tree class
class Tree
{
private:
   static int objectCount;     // Static member variable.
public:
   // Constructor
   Tree()
      { objectCount++; }

   // 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 declared here.

Static member defined here.



Program 14-1

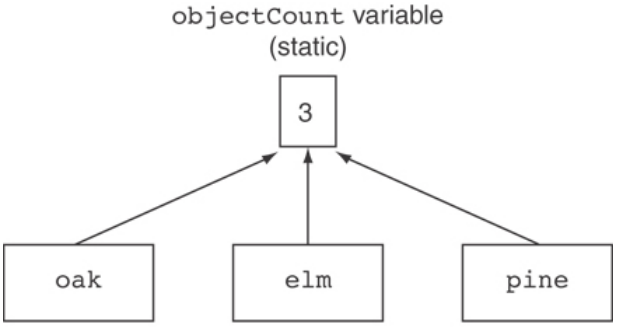
```
// This program demonstrates a static member variable.
#include <iostream>
#include "Tree.h"
using namespace std;

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!
```



Instances of the `Tree` class

   i. In the `Tree` class, every time a new instance is created, it increments the `objectCount` variable

   ii. `objectCount` is the same across all `Tree` objects

c. `static` member function

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

   ii. Static member functions can only access static member data

   iii. Can be called independent of objects:
   `int num = Tree::getObjectCount(); // no object needed to invoke the method, as getObjectCount is static`

d. Why use static variables?

   i. Unique ID's for every instance of a class (based off of number of objects)

   ii. Internal lookup tables – making the var static allows memory savings by keeping a single version for all instances of the class

e. The `this` pointer

   i. `this`: predefined pointer available to a class's member functions

   ii. Always points to the instance (object) of the class whose function is being called [to override local scope]

   iii. Is passed as a hidden argument to all non-static member functions

   iv. Can be used to access members that may be hidden by parameters with same name

   v. Example:

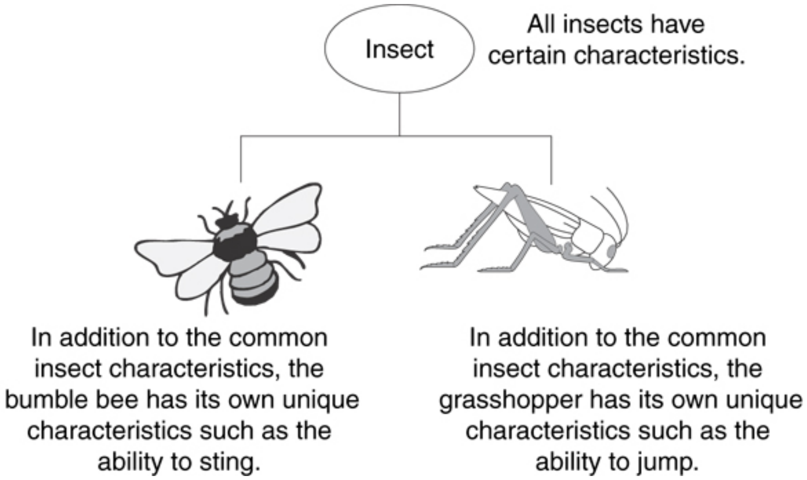
```
class SomeClass
{
        private:
                    int num;
        public:
                    void setNum(int num)
                    { this->num = num; }
                    ...
};
```

15.1. What is Inheritance?

a. Provides a way to create a new class from an existing class

b. The new class is a specialized version of the existing class

c. Example:


Insect
All insects have certain characteristics.
In addition to the common insect characteristics, the bumble bee has its own unique characteristics such as the ability to sting.
In addition to the common insect characteristics, the grasshopper has its own unique characteristics such as the ability to jump.

d. Inheritance establishes an "is a" relationship between classes.

   i. A *poodle* is a *dog*

   ii. A *car* is a *vehicle*

   iii. A *flower* is a *plant*

iv. A *football player* is an *athlete*

e. Terminology:

i. Base class (or parent) – inherited from

ii. Derived class (or child) – inherits from the base class

iii. Notation:

```
class Student            // base class
{
      . . .
};
class UnderGrad : public student // derived class
{
      . . .
};
```

f. An object of a derived class 'is a(n)' object of the base class

g. Example:

i. an `UnderGrad` is a `Student`

ii. a `Mammal` is an `Animal`

h. A derived object has all of the characteristics of the base class

i. An object of the derived class has:

i. all members defined in child class

ii. all members declared in parent class

j. An object of the derived class can use:

i. all `public` members defined in child class

ii. all `public` members defined in parent class

k. Use inheritance for these properties (to save time/reduce work)

15.2. Protected Members and Class Access

a. `protected` member access specification: like `private`, but accessible by:

i. Member functions of the class that originally declared the member.

ii. Friends of the class that originally declared the member.

iii. Classes derived with public or protected access from the class that originally declared the member.

iv. Direct privately derived classes that also have private access to protected members.

b. Class access specification: determines how `private`, `protected`, and `public` members of base class are inherited by the derived class

c. Access Modifiers

   i. `public` – object of derived class can be treated as object of base class (not vice-versa)

   ii. `protected` – more restrictive than `public`, but allows derived classes to know details of parents

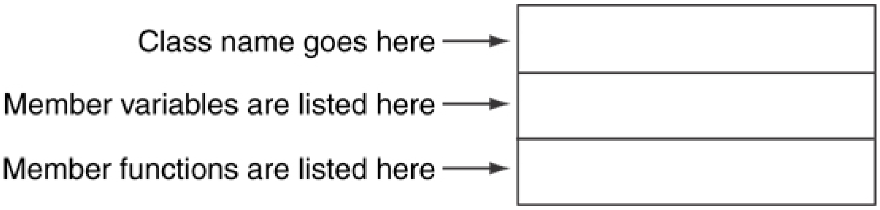
   iii. `private` – prevents objects of derived class from being treated as objects of base class.

13.15. The Unified Modeling Language

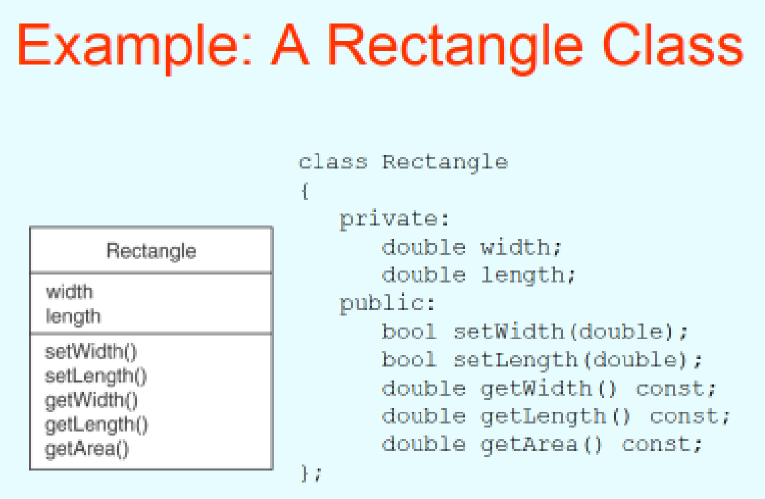
a. *UML* stands for *Unified Modeling Language*.

b. The UML provides a set of standard diagrams for graphically depicting object-oriented systems

c. A UML diagram for a class has three main sections:


Class name goes here
Member variables are listed here
Member functions are listed here

d. Step 1:

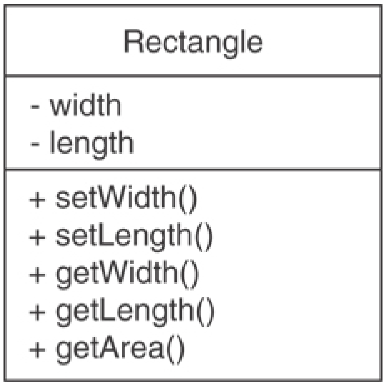


Example: A Rectangle Class

```
class Rectangle
{
   private:
      double width;
      double length;
   public:
      bool setWidth(double);
      bool setLength(double);
      double getWidth() const;
      double getLength() const;
      double getArea() const;
};
```

e. Step 2: Access Specification Notation

In UML you indicate a private member with a minus (-) and a public member with a plus(+).



```
Rectangle
- width
- length
+ setWidth()
+ setLength()
+ getWidth()
+ getLength()
+ getArea()
```

f. Step 3: Data Type Notation

To indicate the data type of a member variable, place a colon followed by the name of the data type after the name of the variable (this is UML notation):

```
- width : double
- length : double
```

g. Step 4: Parameter Type Notation

To indicate the data type of a function's parameter variable, place a colon followed by the name of the data type after the name of the variable.

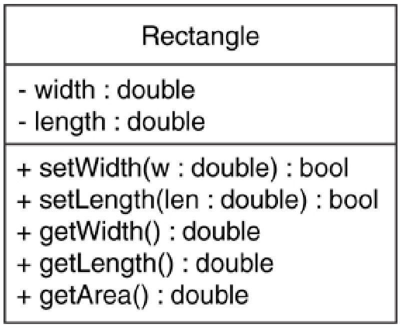
```
+ setWidth(w : double)
```

h. Step 5: Function Return Type Notation

To indicate the data type of a function's return value, place a colon followed by the name of the data type after the function's parameter list.

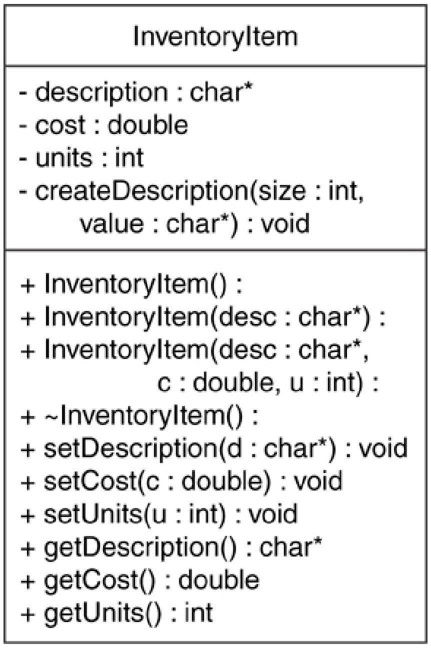
```
+ setWidth(w : double) : void
```

i. Finished product:



```
Rectangle
- width : double
- length : double
+ setWidth(w : double) : bool
+ setLength(len : double) : bool
+ getWidth() : double
+ getLength() : double
+ getArea() : double
```

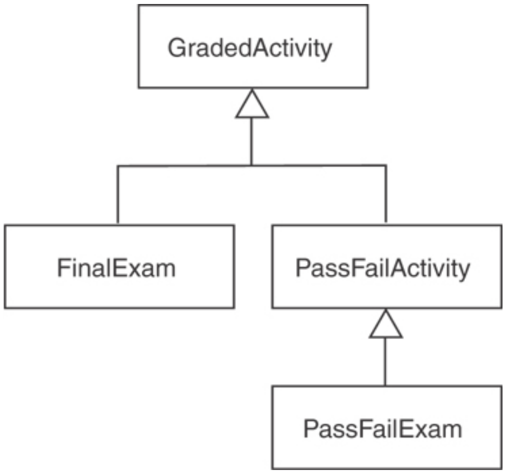
j. With Constructors and Destructors:



```
InventoryItem
- description : char*
- cost : double
- units : int
- createDescription(size : int,
      value : char*) : void
+ InventoryItem() :
+ InventoryItem(desc : char*) :
+ InventoryItem(desc : char*,
            c : double, u : int) :
+ ~InventoryItem() :
+ setDescription(d : char*) : void
+ setCost(c : double) : void
+ setUnits(u : int) : void
+ getDescription() : char*
+ getCost() : double
+ getUnits() : int
```

k. Class diagram for inheritance:

Consider the `GradedActivity`, `FinalExam`, `PassFailActivity`, `PassFailExam` hierarchy in Chapter 15.


GradedActivity
FinalExam
PassFailActivity
PassFailExam