Ch 14: More About Classes

CS 2308
Fall 2011

Jill Seaman

Lecture 14

Using content from textbook slides: Starting Out with C++, Gaddis, Pearson/Addison-Wesley

Instance and Static Members

- instance variable: a member variable in a class. Each object (instance) 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;
  - cannot access instance variables

Tree class declaration

```
// Tree class
class Tree  {
  private:
    static int objectCount;
  public:
    Tree();
    int getObjectCount();
};

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

// Member functions defined
Tree::Tree() {
   objectCount++;
}
int Tree::getObjectCount() {
   return objectCount;
}
```

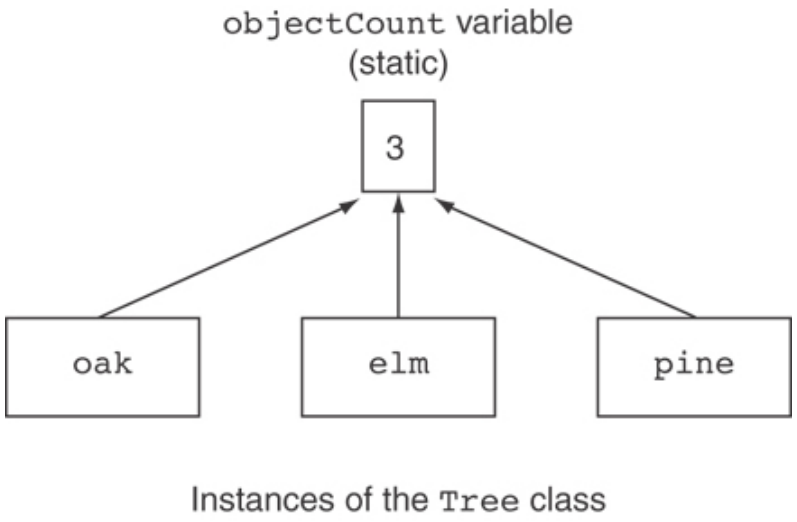

Program demo of static variable

```
#include <iostream>
using namespace std;
#include "Tree.h"

int main() {
   Tree oak;
   Tree elm;
   Tree pine;

   cout << "We have " << pine.getObjectCount()
        << "Trees in our program.\n";
   return 0;
}
```

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



Instances of the Tree class

static member function

- Declared with static before return type:

```
static int getObjectCount();
```

- Static member functions can access static member data only

```
int Tree::getObjectCount() {
   return objectCount;
}
```

- Can be called independently of objects (use class name):

```
cout << "We have " << Tree::getObjectCount()
     << "Trees in our program.\n";
```

Memberwise Assignment

- Can use = to
  - assign one object to another, or
  - initialize an object with another object's data
- Copies member to member. e.g.,

```
instance2 = instance1;
```

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

- Used at initialization: `Time t2 = t1;`
- Used to pass function parameters by value

Memberwise assignment: demo

```
Time t1 = Time(10, 20);
Time t2 = Time(12, 40);

cout << "t1: " << t1.display() << endl;
cout << "t2: " << t2.display() << endl;

t2 = t1;

cout << "t1: " << t1.display() << endl;
cout << "t2: " << t2.display() << endl;
```

```
t2 = t1; //equivalent to:
t2.hour = t1.hour;
t2.minute = t1.minute;
```

```
Output:
t1: 10:20
t2: 12:40
t1: 10:20
t2: 10:20
```

Copy Constructors

- Special constructor used when a newly created object is initialized using another object of same class.
  - This includes passing arguments by value
- Default copy constructor copies field-to-field (memberwise assignment)
- Default copy constructor works fine in many cases

SomeClass declaration

- Problem: what if object contains a pointer?

```
class SomeClass
{
   private:
      int *value;       //ptr to int
   public:
      SomeClass(int val);
      ~SomeClass();
      int getVal();
      void setVal(int);
};
```

SomeClass Implementation

- Implentation of SomeClass

```
#include "SomeClass.h"

SomeClass::SomeClass(int val) {
   value = new int;
   *value = val;
}

SomeClass::~SomeClass() {
   delete value;
}

void SomeClass::setVal(int val) {
   *value = val;
}

int SomeClass::getVal() {
   return *value;
}
```

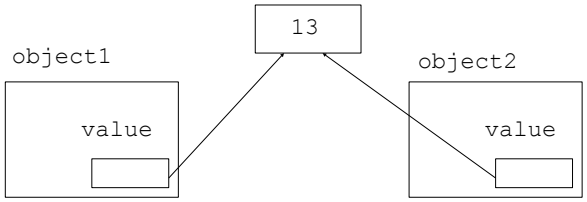

Problem with memberwise-assignment for initialization

What we get from memberwise assignment in objects containing dynamic memory (ptrs):

```
SomeClass object1(5);
SomeClass object2 = object1; //object2.value=object1.value

object2.setVal(13);
cout << object1.getVal();
```

Output: 13



Programmer-Defined Copy Constructor

- Prototype and definition of copy constructor:

```
SomeClass(SomeClass &obj);
```

Add to class declaration

```
SomeClass::SomeClass(SomeClass &obj)
{
   value = new int;
   *value = obj.getValue();  //or *(obj.value)
}
```

- Copy constructor takes a **reference** parameter to an object of the class
  - otherwise it would use initialization to create the obj parameter, which would call the copy constructor for SomeClass: this is an infinite loop

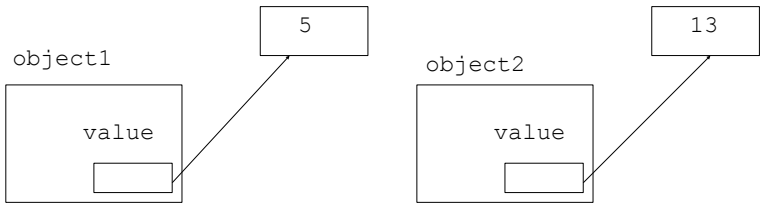
Programmer-Defined Copy Constructor

Each object now points to separate dynamic memory:

```
SomeClass object1(5);
SomeClass object2 = object1;  //now calls copy constr

object2.setVal(13);
cout << object1.getVal();
```

Output: 5



Copy Constructor: limitations

- Copy constructor is called ONLY during initialization of an object, NOT during assignment.
- If you use assignment with SomeClass, you will **still** end up with memberwise-assignment and a shared value:

```
SomeClass object1(5);
SomeClass object2(0);
object2 = object1;      //object2.value=object1.value

object2.setVal(13);
cout << object1.getVal();
```

Output: 13

Operator Overloading

- Operators such as =, +, and others can be redefined to work over objects of a class
- The name of the function defining the over-loaded operator is `operator` followed by the operator symbol:
  `operator+` to overload the + operator, and
  `operator=` to overload the = operator
- Just like a regular member function:
  - Prototype goes in the class declaration
  - Function definition goes in implementation file

Operator Overloading

- Prototype in class declaration:

```
void operator= (SomeClass &rhs);
```

- `operator=` is the function name
- `SomeClass &rhs` is the parameter for the right hand side of operator.
- The operator function 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 invoked using the more conventional syntax:

```
object1 = object2;
```

Overload = for SomeClass

```
class SomeClass
{
   private:
      int *value;
   public:
      SomeClass(SomeClass &obj);
      SomeClass(int);
      ~SomeClass();
      int getVal();
      void setVal(int);
      void operator= (SomeClass &rhs);
};

void SomeClass::operator= (SomeClass &rhs) {
   setVal(rhs.getVal());
}

SomeClass object1(5), object2(0);
object2 = object1;
object2.setVal(13);
cout << object1.getVal() << endl;
```

Output: 5

Returning a Value

- An overloaded operator can return a value

```
class Time
{
   private:
     int hour, minute;
   public:
     int operator- (Time &right);
};

int Time::operator- (Time &right) {
   return (hour%12)*60+minute -
          ((right.hour%12)*60+right.minute);
}

Time time1(12,20), time2(4,40);
int minutesDiff = time2 - time1;
cout << minutesDiff << endl;
```

Output: 260

Overloaded Operators

- Cannot change the number of operands of the operator or the return type
- Overloaded relational operators should return a bool value
- I recommend avoiding using overloaded operators in expressions with <<
  (assign the result of the operation to a variable, then output the variable).