CIT237
Chapter 14:
More About Classes

October 30, 2019

Reminder

- Quiz 5 will be held at the start of class on

  Wednesday, November 6.

- The material covered on Quiz 5 will be:
  - Lectures of October 21 through October 30.
  - Chapters 13 and 14.
  - Further updates will be provided as the date approaches.

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

Declaring a `static` Member Variable

- The static Variable is *declared* inside the class declaration:

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

    static int getObjectCount() const
       { return objectCount; }
};
```

Defining a `static` Member Variable

- The *definition* of the Static Member Variable is outside the class declaration:

```
int Tree::objectCount = 0;
```

- This definition statement causes the variable to be created in memory.
- Unlike an instance variable, a *static* variable exists before any objects of the class are created.

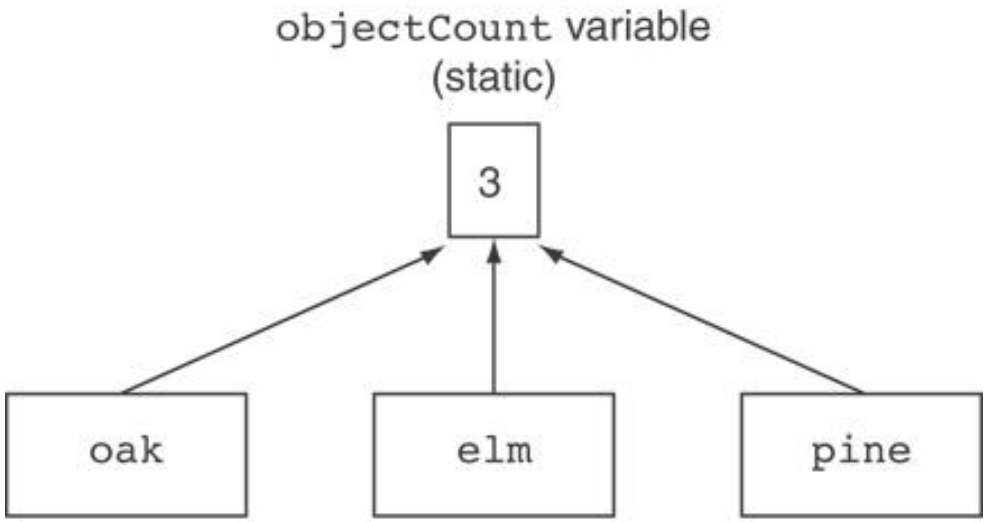
Using the `Tree` Class

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

   // Display number of Tree objects:
   cout << "We have "
        << pine.getObjectCount()
        << " Tree objects!\n";
   return 0;
}
```

- Notice that we called `getObjectCount()` with an *existing* object.

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() const
{ return objectCount; }
```

- Static member functions can only access static member data. Why?
- Can be called independent of objects:

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

The `this` Pointer

`this`:

- a predefined pointer available to a class's member functions
- It always points to the instance (object) of the class whose function is being called.
- It is passed as a hidden argument to all non-static member functions.
- It can be used to access members that may be "hidden" by function parameters with same name.

`this` Pointer Example

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

A perfectly reasonable question: Why wouldn't the programmer just use some *other* name for the function parameter, so the class variable would not be "hidden"?

Answer: Very often programmers do exactly that, but in case that is inconvenient, the language does provide this mechanism to resolve the conflict.

Memberwise Assignment

- In general, we can use the **=** operator to assign one object to another, or to initialize an object with another object's data.
- Copies member-to-member. for example

  `instance2 = instance1;` means:

  copy all member values from `instance1` and assign the values to the corresponding member variables of `instance2`.
- If we use this syntax to initialize an object, then it is actually calling a special constructor function called the "copy constructor":

```
Rectangle r2 = r1;
```

Copy Constructors

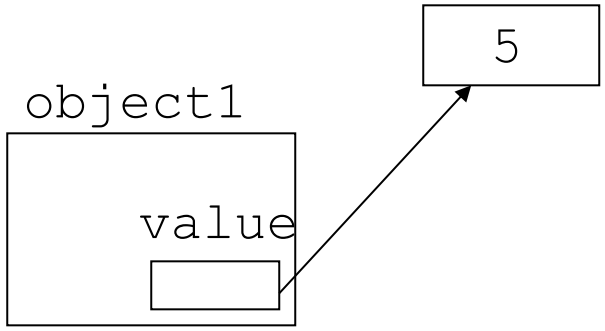
- Special constructor used when a newly created object is initialized to the data of another object of same class.

- The “default” copy constructor:
  - Copies field-to-field: “memberwise assignment”.
  - The default copy constructor works fine in many cases, but *sometimes* a programmer needs to write their own copy constructor.

Dynamically Allocated Member Data

Problem: what if object contains a pointer?

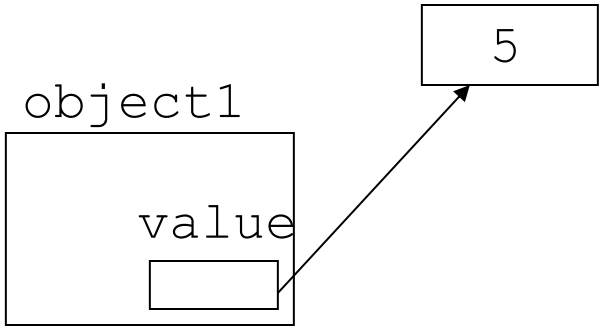
```
class SomeClass
{ public:
      SomeClass(int val = 0)
         {value=new int; *value = val;}
      int getVal();
      void setVal(int);
    private:
      int *value;
}
```



Step 1: First Instance Looks OK

If the class uses the "default" copy constructor, then the behavior is like the following examples:

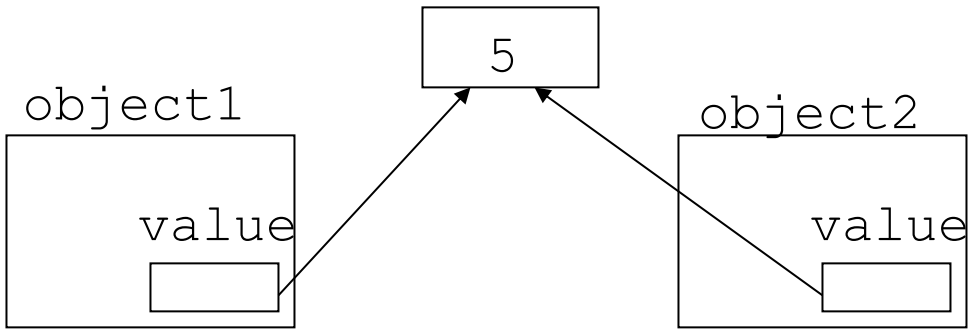
```
SomeClass object1(5);
```



Step 2: Second object is a "Memberwise" copy of the first

What we get using memberwise copy with object containing dynamic memory:

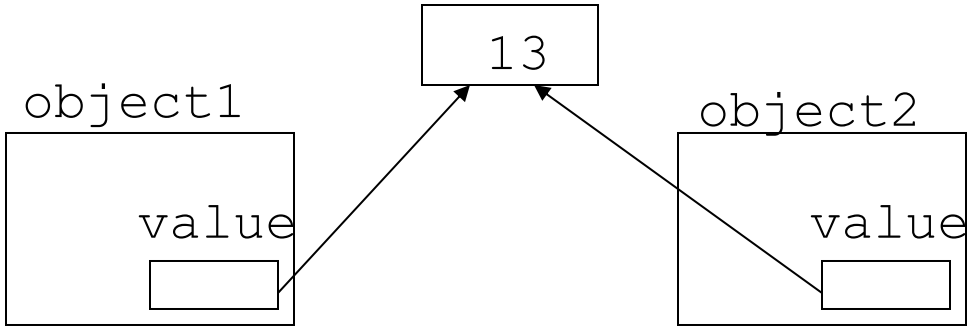
```
SomeClass object1(5);
SomeClass object2 = object1;
```



Step 3: Object1 Data Can Be Lost

Modifying object2 causes the dynamically allocated data in object1 to be overwritten:

```
SomeClass object1(5);
SomeClass object2 = object1;
object2.setVal(13);
cout << object1.getVal(); // also 13
```



Programmer-Defined Copy Constructor

- The programmer can address this issue by writing another copy constructor for the class.
- This works better with objects containing pointers:

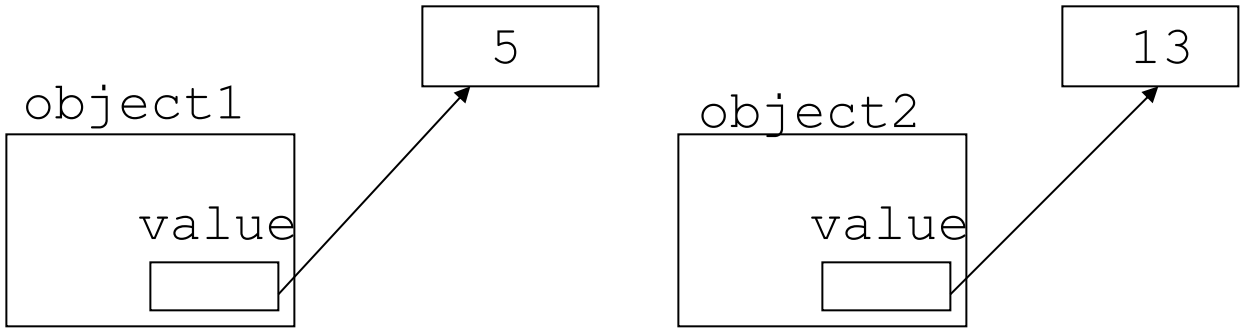
```
SomeClass::SomeClass(const SomeClass &obj)
{
   value = new int;
   *value = obj.value;
}
```

- A copy constructor takes a reference parameter to an object of the same class.

Avoids Multiple Ownership Conflict

- The same scenario results in each object now pointing to separate dynamic memory:

```
SomeClass object1(5);
SomeClass object2 = object1;
object2.setVal(13);
cout << object1.getVal(); // still 5
```



Safeguard the Original Object

- Since copy constructor has a reference to the object it is copying from,

  ```
  SomeClass::SomeClass(SomeClass &obj)
  ```

  it *could* modify that object.

- To prevent this from happening, make the object parameter `const`:

```
SomeClass::SomeClass
                    (const SomeClass &obj){
      . . .
}
```

Pause to Reflect on OO Syntax

- When we first learned about functions, we would pass input arguments, and possibly get a value returned by the function:

```
largest = max(a, b);
```

- Now that we are using objects, our function invocations look a little different:

```
rectangle1.overlap(rectangle2);
```

- The object-oriented way of thinking involves one *object* calling the function, and passing a reference to the "other" object.

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.*,

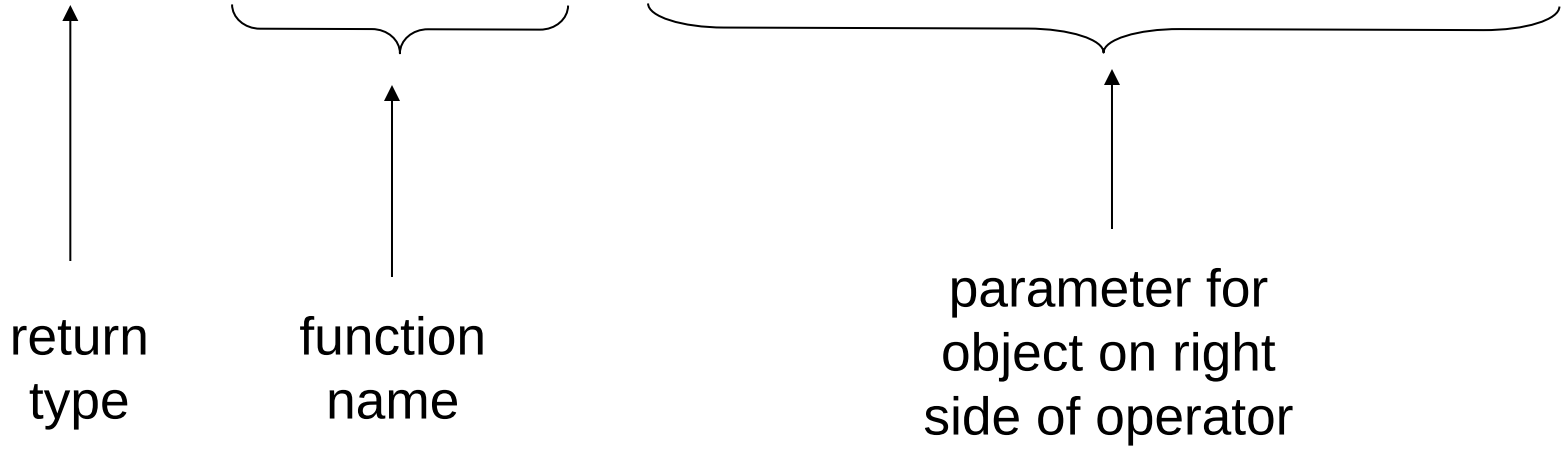
  `operator+` to overload the **+** operator, and

  `operator=` to overload the **=** operator
- Prototype for the overloaded operator should be in the declaration of the class that is overloading it.
- Overloaded operator function definition should be with other member functions.

Operator Overloading

- Prototype:

```
void operator=(const SomeClass &rval)
```



- Operator is called via object on left side

Invoking an Overloaded Operator

- General OO function invocation:

```
object1.someFunction(object2);
```

- Operator can be invoked as a member function:

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

- It can also be used in more intuitive manner:

```
object1 = object2;
```

Overloaded operator can return a Value

Consider a class representing two-dimensional Points:

```
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);
// display distance between 2 points.
cout << point2 - point1 << endl; //
  displays 2.82843
```

Using Operator Overloading

- Operator Overloading provides concise notation, which *can* be convenient for the user of a class, BUT…
  - If the wrong operators are used, or the resulting class interface is not what people usually expect to see (by normal mathematical conventions), then overloaded operators can make it hard to write code using your class.
  - In general, if a well named function is more intuitive than the overloaded operator, then use the named function.

Returning an Object

- Return type the same as the left operand supports notation like:

```
object1 = object2 = object3;
```

- Function declared as follows:

```
const SomeClass operator=(const SomeClass &rval)
```

- Inside the function, include as last statement:

```
return *this;
```

which returns a copy of the “current object”.

Notes on Overloaded Operators

- You can change the "meaning" of an operator.
- You cannot change the number of operands that the operator takes
- Only certain operators can be overloaded. Cannot overload the following operators:

```
?:
.
.*
::
sizeof
```

Overloading Types of Operators

- `++`, `--` operators overloaded differently for prefix vs. postfix notation
- Overloaded relational operators should return a `bool` value
- Overloaded stream operators `>>`, `<<` must return reference to `istream`, `ostream` objects and take `istream`, `ostream` objects as parameters

Overloaded [ ] Operator

- Can create classes that behave like arrays, provide bounds-checking on subscripts
- Overloaded [ ] returns a reference to object, not an object itself

Aggregation

- Aggregation: an object of one class is a member of another class
- Supports the modeling of '**has a**' relationship between classes: the object of the enclosing class '**has a**' object of the enclosed class.

Aggregation Example

```
class StudentInfo
{
   private:
          string firstName, LastName;
          string address, city, state, zip;
  ...
};

class Student
{
   private:
          StudentInfo personalData;
  ...
};
```

Aggregation in UML Diagrams