MORE ABOUT CLASSES AND OOP

STATIC MEMBER

- If a member **variable** is declared static , all objects of that class have access to that variable.
- If a member **function** is declared static, it may be called before any instances of the class are defined.

STATIC MEMBER VARIABLE

- Member variables have two categories
  - Instance variable - must be associated with a particular instance of a class
  - Static variable - is not associated with any specific instance of a class but all classes

THREE THINGS TO REMEMBER ABOUT STATIC MEMBER VARIABLES

Must be **declared** in class with keyword **static**:

```
class IntVal
{
public:
    IntVal(int val)
    { value = val;
         valCount++
      }
    int getVal();
    void setVal(int);
    private:
    int value;
    static int valCount;
   };
```

THREE THINGS TO REMEMBER ABOUT STATIC MEMBER VARIABLES

Must be **defined** outside of the class:

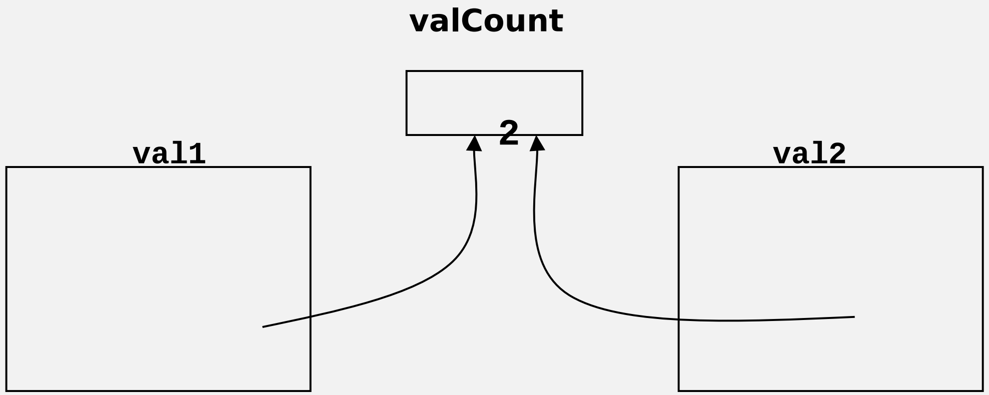
```
    //Definition outside of class
   int IntVal::valCount = 0;
```

THREE THINGS TO REMEMBER ABOUT STATIC MEMBER VARIABLES

Can be **accessed or modified** by any object of the class: Modifications by one object are visible to all objects of the class:

prog11_2.cpp budget.h

IntVal val1, val2;



STATIC MEMBER FUNCTIONS

- Unlike a public member function, a static member function does not need to be associated with any instance of an object.
- Syntax :
  - Static <return type> <function name> (<parameter litst>)
- Work with static member variables of the class

STATIC MEMBER FUNCTION

- Can be called independently of class objects, through the class name
  - cout << SomeClass::getSomeValue();
- Remember when an object is created the ***this*** pointer created with the object - when you call a static function without an object you do not get a ***this*** pointer
- Can be called before any instance object of the class has been created
- Used often to manipulate static member variables of the class
- prog11_3.cpp, budget2.h, budget2.cpp

FRIENDS OF CLASSES

- Not a member of class
- Remember private member variables are hidden from all programs outside the class
- A friend function is not a member of a class, but has access to the class private members
- Is a stand-alone function or a member function of another class
- Declare a friend of a class with the friend keyword in the function prototype

FRIEND FUNCTION DECLARATIONS

- Stand-Alone Function

```
class aClass
 {
   private:
     int x;
     friend void fSet(aClass &c, int a);
 };

 void fSet(aClass &c, int a)
 {
    c.x = a;
 }
```

FRIEND FUNCTION DECLARATIONS

- As member of another class:

Notice the scope

```
 class aClass
{ private:
   int x;
   friend void OtherClass::fSet(aClass &c, int a);
};
class OtherClass
{ public:
    void fSet(aClass &c, int a)
    { c.x = a; }
};
```

FRIEND CLASS DECLARATION

- An entire class can be declared a friend of a class:

```
 class aClass
{private:
  int x;
  friend class frClass;
};

class frClass
{public:
  void fSet(aClass &c,int a){c.x = a;}
  int fGet(aClass c){return c.x;}
};
```

FRIEND CLASS DECLARATION

- If **`frClass`** is a friend of **`aClass`**, then all member functions of **`frClass`** have unrestricted access to all members of **`aClass`**, including the private members.
- In general, restrict the property of Friendship to only those functions that must have access to the private members of a class.

These programs demonstrate Static variabls and functions as well as Friend functions.

- prog11_4.cpp, auxil.h, budget3.h, budget3.cpp, auxil.cpp
- Unless it is an operator overloading and/or I tell you to use one, you may **not** use friends in any of the assignments.

DEFAULT COPY CONSTRUCTOR AND ASSIGNMENT OPERATOR

- A special constructor provided by C++
- Used to copy objects
  - Uses member wise copying technique
  - Member wise copy examples:
    - Assignment:
    - prog11_5.cpp
    - Default Copy Constructor
    - prog11_6.cpp

ASSIGNMENT VS COPY CONSTRUCTOR

- It is importance that you know the difference between when an assignment operator is called and when a copy constructor is called.
- Assignment
  - When you have two fully constructed object and one is “set equal to another”
    - Box b1(5,10);
    - Box b2(8, 15);
      - b2 = b1; //**This is an assignment.**

ASSIGNMENT VS COPY CONSTRUCTOR

- It is importance that you know the difference between when an assignment operator is called and when a copy constructor is called.
- Copy Constructor
  - Since this is a constructor and constructors are only called one time for an object, a copy constructor is called when one unconstructed object is set equal to a fully constructed object.
    - Box b1(5,10); //this is a fully constructed object
    - Box b2 = b1; //no constructor has been called on b2 and we are setting it = to a fully constructed object (b1) so the copy constructor is called here.

PROBLEMS WITH DEFAULT COPY CONSTRUCTOR

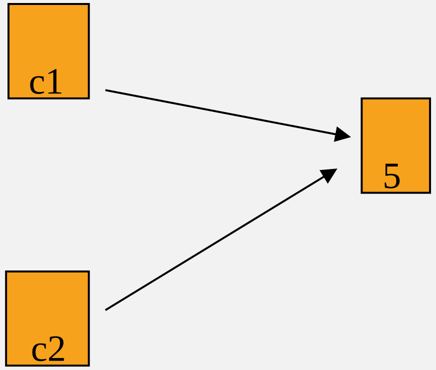
- Problems occur when objects contain pointers to dynamically allocated memory

```
class CpClass{
private:
int *p;
public:
CpClass(int v) {
   p = new int;
   *p = v;
}
     ~CpClass() {
   delete p;
}
};
```

Suppose we have the following

```
CpClass c1(5);
if (something)
{
  CpClass c2=c1;

}
```



When c2 goes out of scope the destructor is called and bad things happen (This is a shallow Copy)

PROBLEMS CAUSED WHEN OBJECTS SHARE DYNAMIC ALLOCATED MEMORY

- The destruction of one object deletes memory still in use by other objects
- Modifying memory by one object affects other objects sharing that memory

NumberArray.h, NumberArray.cpp prog11_7.cpp

HOW TO FIX THIS PROBLEM – DEFINE YOUR OWN COPY CONSTRUCTOR

- A copy constructor has a parameter of the same type as the class and the parameter is a reference
- It is also a good idea to make this a ***const*** to keep you from accidently changing the data of the object being passed in
  - CpClass (const CpClass &obj)

HOW TO FIX THIS PROBLEM – DEFINE YOUR OWN COPY CONSTRUCTOR - CONTINUED

- Uses the data in the object passed as parameter to initialize the object being created
- Allocate separate memory to hold new object’s dynamic member data
- Copies the data, not the pointer, from the original object to the new object

```
CpClass (const CpClass &obj)
    {
      p = new int;
      *p = *obj.p;
    }
```

EXAMPLE COPY CONSTRUCTOR

```
class CpClass
{   private:
   int *p;
     public:
   CpClass (const CpClass &obj)
   { p = new int;   *p = *obj.p; }
   CpClass(int v){ p = new int; *p = v;}
   ~CpClass( ) {delete p;}
};
```

EXAMPLES

- We saw in prog11_7.cpp and NumberArray.cpp NumberArray.h the problem that not defining our own copy constructor can cause.

Now let’s see how to fix the problem

- prog11_8.cpp (NumberArray2.cpp, NumberArray2.h)

WHEN IS A COPY CONSTRUCTOR USED

- When an object is initialized from an object already created of the same class
- When an object is passed by value to a function
- When an object is returned by value using a return statement from a function

OVERLOADING = OPERATOR

- For the same reason as using user defined copy constructor you need to overload the = Operator
- C++ provided = Operator uses member wise assignment. If you have dynamically allocated member (pointers) variables you will have problems
- On the next slide we will discuss several examples of the = operator

WHEN IS THE COPY CONSTRUCTOR AND “=” GETTING CALLED

Demonstrates the problem with not having an overloaded = operator. We will run this and then fix the problem and rerun.

prog11_9.cpp
overload.h, overload.cpp

DESTRUCTOR

- You need to remember to write the destructor as well
  - delete the allocated memory

RULE OF THREE

- In general if a class dynamically allocates memory, in a constructor, you should define:
  - A copy constructor
  - A Destructor
  - An “ = “   equal operator

RULE OF ZERO

- If you don’t have a pointer as a data member you do not need nor should you provide a:
- Copy/move constructor
- Assignment/move operator =
- Destructor
- Why do you think this is true?

OTHER OPERATOR OVERLOADING

- We just talked about overloading the = operator so that we can set one object = to another.
- We can also overload other operators such as:

+	*****	**/**	**!**	**<**	**>**	**+=**
-=	/=	-	<<	>>	!=	<=
>=	&&	||	++	--	[ ]	( )

OPERATOR OVERLOADING

- A couple things you need to know about operator overloading
  - You get to control how the operator behaves with respect to your class as an example:

operatorWeirdness.cpp

  - Something else you need to know is you can not change the number of operands for the operator. For example:
    - +, = , - , etc. are all binary operators when we say a = b that is 2 operands.
    - This is the equivalent of calling a.operator=(b); these do the same thing.
    - Lets add this to our operatorWeirdness.cpp program and see what happens.

OPERATOR OVERLOADING

- There are a couple ways to approach overloading an operator.
  - *Make the overloaded operator a member function of the class.* This allows the operator function access to private members of the class. It also allows the function to use the implicit *this* pointer parameter to access the calling object.
  - *Make the overloaded member function a separate, stand-alone function.* When overloaded in this manner, the operator function must be declared a ***friend*** of the class to have access to the private member of the class.
  - There are some operators such as << or >> that **must** be overloaded as stand-alone friend functions.

OPERATOR OVERLOADING

- You can also overload [ ] operator. Which gives you the ability to write classes that have array-like behaviors. You have already seen this in the string class. This is why string name = “William” can access individual characters.
- intarray.h, intarray.cpp, intarrayDriver1.cpp
- More Examples:
  - Length1.h, Length1.cpp, prog11_11.cpp
  - Box.cpp

REVIEW LVALUE AND RVALUE

- Consider the following:
  - int x = 555;
  - int y = x + 5;
  - string s1 = “Hello ”;
  - string s2 = “world!”;
  - string s3 = s1 + s2;

- What is a lvalue and rvalue?

REVIEW C++ REFERENCE

- How do we create a reference variable in c++?
- Can we do the following? Why or Why not?
  - int& x = 666;
  - String s1 = “hello ”;
  - String s2 = “world”;
  - String& s3 = s1 + s2;

  - 666 is a literal constant and you can not bind it to a reference
  - The result of s1 + s2 is stored in a temporary variable and again you can not bind a reference to a temporary variable

MAGIC OF RVALUE REFERENCE

- C++ has introduced a new type called rvalue reference
- Syntax for a rvalue reference is <type>&& variableName = something;
  - Lets look at: refReview.cpp

- Rvalue reference may appear useless. However, they make Move semantic possible

MOVE SEMANTICS

- Move semantics is a new way of moving resources around in an optimal way by avoiding unnecessary copies of temporary objects.
- We are going to see this through an example. We will use the numberArray class we have worked with several times.
    - overload2.cpp, overload2.h, program11_14.cpp
- Now we will discuss, how the move semantic is going to help make this the above more efficient.
    - Overload3.cpp, overload3.h, program11_15.cpp

MOVE SEMANTICS

- When does the compiler use Move operations
  - A function returns a result by value
  - An object is being assigned to and the right-hand side is a temporary object
  - An object is being initialized from a temporary object

RULE OF FIVE

- There is a new constructor called move and a move assignment operator. We will not talk about these in this class.
- However, if your class desires to use the move semantic, you will need to implement the rule of five
  - Copy constructor
  - Assignment operator =
  - Destructor
  - Move constructor
  - Move assignment operator =

MOVE SEMANTICS

- Move semantics is a new way of moving resources around in an optimal way by avoiding unnecessary copies of temporary objects.
- We are going to see this through an example. We will use the numberArray class we have worked with several times.
  - overload2.cpp, overload2.h, program11_14.cpp
- Now we will discuss, how the move semantic is going to help make this the above more efficient.
  - Overload3.cpp, overload3.h, program11_15.cpp