MORE OPERATOR STUFF

Review - Code!

- Let’s write a simple dynamic array class (not like one you’d ever write)
  - constructor/destructor
  - private pointer variable
  - member get/set functions
  - member length function
  - copy constructor

Question

- Remember how the copy constructor works?

```
// construct an Employee
Employee samuel( “Samuel T. Larson” );

// construct sam as a copy of samuel
Employee sam( samuel );
```

- This works fine when we’re *constructing* an object, but how about later? Can we assign objects to each other?

```
sam = sammy;        // does this work?
```

Yup.

- Turns out that yes, this does work.
- C++ automatically ***overloads the assignment operator*** for you – it defines a function that gets called when code tries to assign something to your class
- This default operator does a piecewise assignment – same as the default copy constructor
- And we can make our own version, too! (Why would we want to?)

Assignment Operator Overloading

- The function to overload the operator looks like this:

```
Employee& operator=( const Employee& rhs )
{
    // do assignment stuff in here…
}
```

- It works almost exactly like the copy constructor…
- …except this one returns Employee&

Assignment Chaining

- Remember: assignments are done right to left
- The result of **b = c** needs to be something that can be assigned to **a**
- **operator=** is the function handling **b = c**
- So operator= needs to return something that can be assigned to **a**: the result of **b = c**

```
Employee a, b, c;
a = b = c;
```

Each time a value gets assigned to an instance of Employee, the operator= function gets called

So:

- What value should the operator= function return?
- It needs to be *the current object* for assignment chaining to work
- We know how to refer to *other* objects (by name, by pointer, etc.)
- But how do we refer to the value of the current instance from *within* that instance?

Introducing **this**

- The C++ keyword **this** solves this problem
- every object gets a pointer called **this** to its own address

```
class cat
{
public:
    cat()
    {
        // same-same
        meow = 189;
        this->meow = 189;
    }
private:
    int meow;
};
```

- **this** is of type `const cat*` in this case, and is not modifiable
- **this** can only be used from inside a class (why?)

So: (again)

- What value should the operator= function return?
- We need to return the current object (so it can be assigned again!)
- **this** gives us a pointer to the current object

```
Employee& operator=( const Employee& rhs )
{
    return (what?)
}
```

anyway, back to…

Operator Overloading

- In that example we overloaded (defined for this class) the assignment operator
- Turns out we can overload all *kinds* of operators: +, *, -, *, <<, >>, and a fair number of others
- This lets us give actions to our class in other ways than calling public member functions

Overloadable Operators

- Here’s the operators you can overload:

```
+   -   *   /   =   <   >   +=   -=   *=   /=   <<   >>
<<=   >>=   ==   !=   <=   >=   ++  --   %   &   ^   !   |
~   &=   ^=   |=   &&   ||   %=   []   ()   ,   ->   *   ->
new   delete   new[]   delete[]
```

- You can do all *kinds* of funky stuff with these. Usually we just stick to the basics.

Operators are functions!

- We overloaded the = operator with this function:

```
Employee& operator=( const Employee& rhs )
{
    return (what?)
}
```

- Overloaded operators are just regular C++ functions! Not much special about them.
- This is one of the rare times that a function *name* can be “non-standard” though!

For example…

- Say we've got a complex number class called Complex
- It's natural for us to want to do things like this:

```
Complex a, b;
Complex c = a + b;
```

- Operator overloading lets us define how the + operator works for our Complex class

Side Note:

- According to some schools of thought, operator overloading is a bit dangerous
- The reason: you can’t see what you’re getting when you read the code:
- In this code, there are no possible side-effects:

```
int a = 10, b = 5;
a += b;
```

- But with our own classes, it’s not easy to tell *what* the overloaded operators actually do.

```
myArray a, b;
a += b;
```

Moral of the Story

- To write good code:
- Overload operators should mimic the functions of their built-in counterparts
- If you want to do anything else, write an appropriately-named member function to do it for you

Implementations

```
class Complex
{
public:
   Complex();

   Complex( const Complex& c )
   {
      // this needs an implementation
   }

   Complex& operator=( const Complex& c )
   {
      // so does this
   }

private:
   float real, imag;
};
```

- How would we implement the copy constructor and operator=?
- Do we really need *both* of them?

A shortcut

- We can often implement one function by using another (we did this with constructors, remember?)
- The copy constructor and operator= are very similar. Rather than implementing both of them, you can just implement the operator=.
- What would the copy constructor look like?

```
Complex( const Complex& c )
{
    // what does this look like?
}
```

Why does this matter?

- Partly because it makes things easier.
- Partly because… let's take a look at the list of overloadable operators again!

```
+   -   *   /   =   <   >   +=   -=   *=   /=   <<   >>
<<=   >>=   ==   !=   <=   >=   ++  --   %   &   ^   !   |
~   &=   ^=   |=   &&   ||   %=   []   ()   ,   ->   *   ->
new   delete   new[]   delete[]
```

- Aka, if you've overloaded +, you'll probably want to overload += as well.

Another example

```
class Complex
{
public:
   Complex();

   bool operator==( const Complex& c )
   {
      if( real == c.real )
          return true;
      else
          return false;
   }

private:
   float real, imag;
};
```

- Here we’re overloading the equality (==) operator
- Will these work?

```
Complex p, q;

if( p == q )
    ;  // do something


if( p != q )
    ;  // do something else
```

Nope.

- Turns out that == and != are *different* operators
- If you want to use !=, you have to define it

```
Complex p, q;

if( p == q )
    ;  // do something


if( p != q )
    ;  // do something else
```

This is the error that Visual C++ generates:

Cpptest.cpp: error C2676: binary '!=' : 'Complex' does not define this operator or a conversion to a type acceptable to the predefined operator

```
bool operator!=( const Complex& c )
{
    // how do we implement this?
}
```

Another operator: multiplication

- Let’s look at the vector3D class again:

```
class Vector3D
{
public:
   Vector3D();

private:
   float x, y, z;
};
```

- Based on what we’ve seen so far, what would the operator* function look like?

Overloading the Overloads

- We defined an operator* function that accepts a Vector3D, but we can make it accept other types too
- We can overload the overloaded operators!

```
class Vector3D
{
public:
   Vector3D();
   Vector3D operator*( Vector3D& rhs );


private:
   float x, y, z;
};
```

- How do define another version of this function that accepts a single float?

Random Overloading Stuff

- Assuming the operators are correctly implemented, can we do this?

```
Vector3D* vec = new Vector3D;
Vec = vec * 4;
```

- Why or why not?

Stuff You Can’t Do:

- Overload these operators: . .* :: ?:
- Overload operators for primitive types (int, float, etc.)
- Create new operators! You’re stuck with the ones that C++ understands.
- Change the arity of an operator (make a binary operator unary, etc)
- Change the precedence of an operator.

Question

```
void doStuff( int x )
{
    cout << x << endl;
}
```

- Say we’ve got a very simple doStuff function…

- Can we do this?

```
float bob = 5.2;
doStuff( bob );
```

- Why does this work?

Question

```
class Complex
{
public:
   Complex();

private:
   float real, imag;
};
```

- Say we’ve got a very simple Complex class…

- Can we do this?

```
Complex number;
char whatever[100];
strcpy( whatever, number );
```

- Why would this **not** work?

Type Conversions

- Remember this stuff?

```
float bob = 5.2;

// implicit type conversion
doStuff( bob );

// explicit type conversion
doStuff( (int)bob );
```

- Whether explicitly or implicitly, C++ will convert types when it can
- We can add this functionality to classes, too!

Conversion Operators

```
class Complex
{
public:
   Complex();
   operator int();

private:
   float real, imag;
};

Complex::operator int()
{
   return (int)real;
}
```

- The **operator int()** function automatically gets called when you try to convert the code to an integer
- This means you can use Complex anywhere you'd use an integer – Complex gets automatically converted to an int

Anatomy of a Conversion Operator

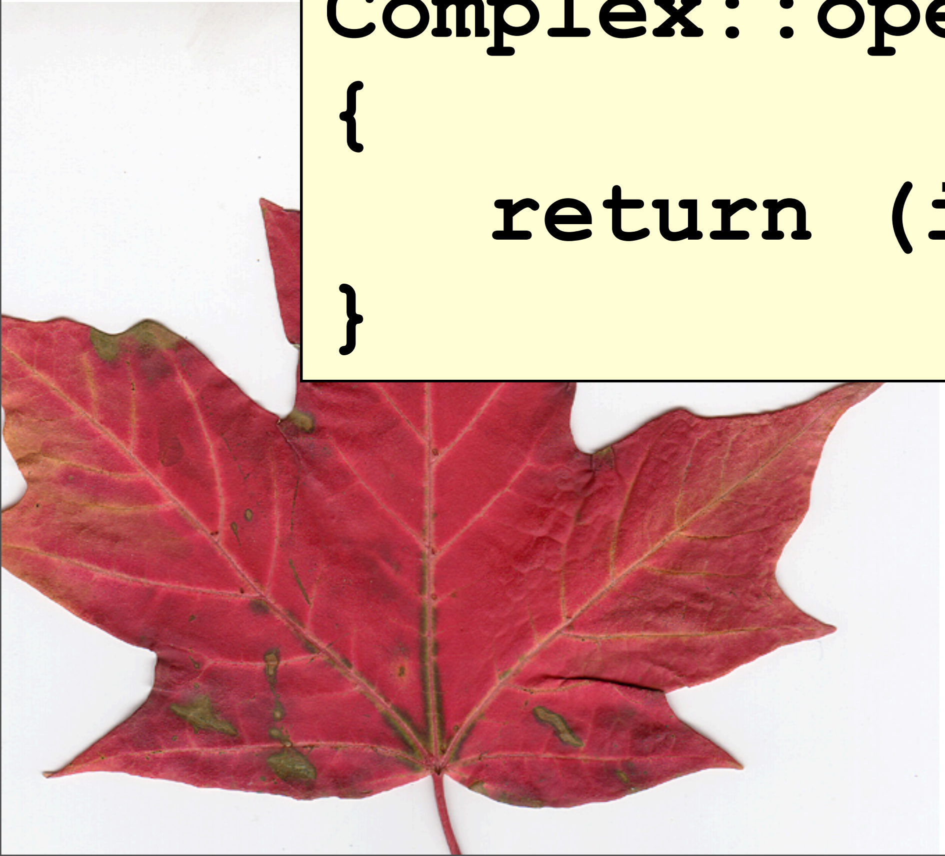
no return type (why?)

operator keyword

type this function converts to

```
Complex::operator int()
{
    return (int)real;
}
```

parenthesis close out the function signature



Finish the Example

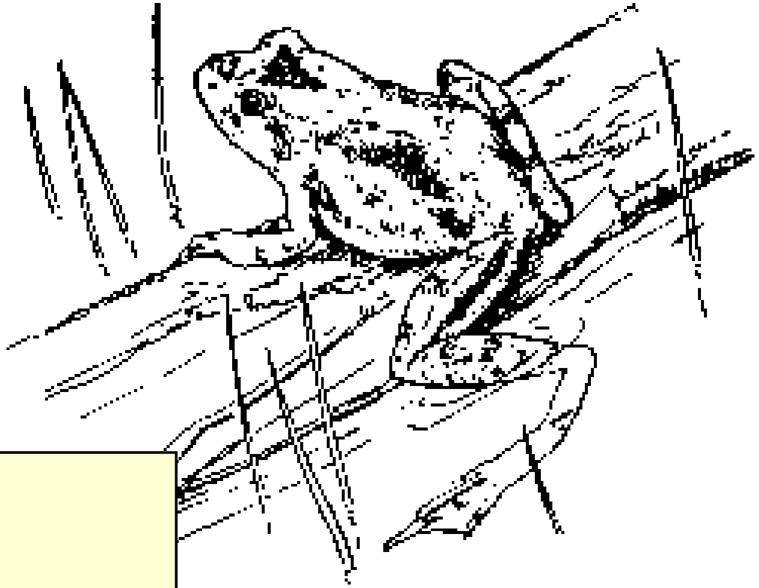
```
class Complex
{
public:
   Complex();
   operator ???();

private:
   float real, imag;
};

Complex::operator ??()
{
   return ??;
}
```

- Let’s say we wanted to do a string conversion operator:

```
Complex number;
cout << number << endl;

// this should print out
// the word “hello”
```

- How would we do that?

Question



- Say we had a Node class:

```
class Node
{
   private:
      int data;
      Node* next;
};
```

- …but all the node manipulation was done in a separate class called **LinkedList**.
- Would **LinkedList** be able to access the next variable in an instance of **Node**?

Sometimes you just need a `friend`

- Sometimes you want external code (code that’s not in the class) to be able to access private class variables!
- … but not anyone else.
- This can be done using the C++ **friend** keyword.

How It All Happens

- This is done by adding the “friend class ClassName” within the class:

```
class Node
{
    friend class LinkedList;

    private:
        int data;
        Node* next;
};
```

```
void LinkedList::doStuff()
{
    Node* ptr = head;

    // used to be illegal
    // now it’s legal!
    ptr = ptr->next;
}
```

- Now LinkedList can access all variables in an instance of Node as if they were public.

Friend Declarations

```
class Node
{
   friend class LinkedList;

   private:  // ... etc
};
```

- Friend declarations can be put anywhere in the class – public section, private section, top, bottom, whatever
- The classes that you declare to be friends don’t actually have to exist…
  - so watch for typos!
- A class can have lots of friends!

Friend Functions

```
class Node
{
   friend void breakStuff
();

   private:
      int data;
      Node* next;
};
```

```
void breakStuff()
{
    Node* ptr = head;

    // mwahahahahaha!!!
    delete ptr;
    ptr = NULL;
}
```

- Another option is to declare a single function to be a friend
- Here, the function **`void breakStuff()`** is allowed private access to Node
- How could we make a function in *another class* be a friend?

Friend Functions

```
class Node
{
   friend void breakStuff( int x, float q );

   private:
      int data;
      Node* next;
};
```

to make this work we have to put the entire function signature here!

- How would we make a member function of another class a friend? (Not the *entire* class – just a single member function)

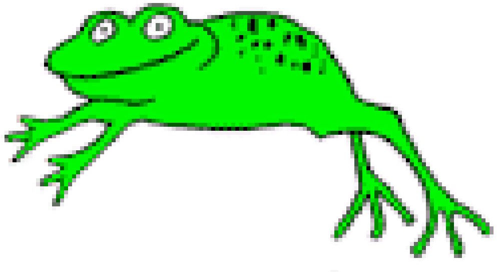
Friendliness

- Is **friend** a good idea or a bad idea?
- Does **friend** break the idea of encapsulation?
- When and why might you want to do this?

How to Remember This:

In C++, all your `friend`s can see your `private`s.

Static: Background

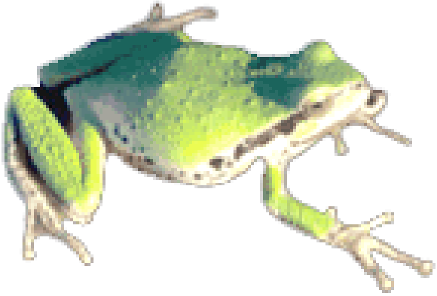


```
class Dog
{
   private:
      char name[50];
      int age;
};
```

```
int main()
{
   Dog gus;
   Dog pepper;
   Dog bitsy;
   Dog charlie;
   Dog toby;
   Dog checkers;
}
```

- Here we have 6 different instances of the class `Dog`.
- Each instance has its own set of member variables.
- So there are 6 different age variables – one per instance.

The Problem



- What if we wanted to keep a count of the number of instances of `Dog` in the entire program?
- Where would it make most sense to keep that counter?
  - A member variable in `Dog`?
  - A global variable?
- The ideal would be a counter that belongs to the *entire class* – not just a single instance of it.

Introducing static

- This can be done using the C++ keyword **static**.
- static variables are shared amongst all instances of the class – no one instance gets to “own” a static variable!
- Best way to think of a static: the lifetime of a global variable, but the access/scope of a class member variable   (what’s the difference?)

Declaring Static Variables

```
class Dog
{
   private:
      char name[50];
      int age;

      // counter declaration
      static int counter;
};

// actual counter definition
int Dog::counter = 0;
```

- static variables are weird – we *declare* them inside the class, we *define* them outside the class
- Think of them like a function – the declaration is only a prototype!
- It still needs to be defined in the global scope (why?)

So…

```
class Dog
{
   private:
      char name[50];
      int age;

      // counter declaration
      static int counter;
};

// actual counter definition
int Dog::counter = 0;
```

```
int main()
{
   Dog gus;
   Dog pepper;
   Dog bitsy;
   Dog charlie;
   Dog toby;
   Dog checkers;
}
```

- There is a single `counter` variable in this program…
- To which of the 6 Dogs does `counter` “belong”?
- Which of them can access it?
- What do you think is the syntax for doing so? (inside the class)

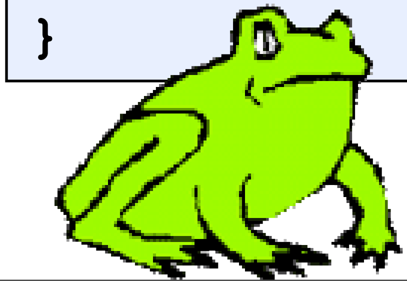
Accessing Static Variables

- Static variables can be accessed exactly like regular variables!
- In addition, access specifiers (public, private, etc.) work the same way they do with non-static variables

```
class Dog
{
   public:
      Dog()
      {
         counter++;
      }

      ~Dog()
      {
         counter--;
      }

   private:
      char name[50];
      int age;

      // counter declaration
      static int counter;
};

// actual counter definition
int Dog::counter = 0;
```

Meanwhile, Outside the Class…

```
int main()
{
   Dog gus;
   Dog pepper;
   Dog bitsy;
   cout << bitsy.counter;

   Dog charlie;
   Dog toby;
   {
      Dog checkers;
      cout << gus.counter;
   }

   cout << toby.counter;
}
```



- As long as `counter` is public, it can be accessed outside the class as if it were a non-static variable

- What happens if there are no instances of `Dog` we can use to access `counter`?

Viva la variables estáticas!!

- Static member variables *always* exist – whether the class has ever been instantiated or not!
- We can access them using the scope resolution operator:

```
int main()
{
    cout << Dog::counter << endl;
}
```

- This works because `counter` belongs to the *class* `Dog` – not any one *instance* of `Dog`!

Static Methods

- Methods (member functions) can also be static!
- Static methods:
  - don’t belong to any particular instance of the class
  - can be called even if there are *no* instances of the class!
  - can *not* access any non-static data in the class

Broke

- In this example program, getCount() tries to access both age and counter
- If we were to do this:

```
cout << Dog::getCount();
```

- … which instance of age would the function access?
- (This doesn’t compile, by the way!)

```
class Dog
{
   public:
      static int getCount()

   private:
      char name[50];
      int age;

      // counter declaration
      static int counter;
};

int Dog::getCount()
{
   age++;
   return counter;
}

// actual counter definition
int Dog::counter = 0;
```

The Rules:

- Can static methods access:
  - Static member variables? - *Yes!*
  - Non-static member variables? - *No!*

- Can regular methods access:
  - Static member variables? - *Yes!*
    - but there's only one copy to be shared amongst all instances of the class
  - Non-static member variables? - *Yes!*
    - This is the normal case

Non-Class Static Variables

- Regular variables can be static too - not just class member variables
- Just like class static variables, regular static variables have:
  - **global** lifetime
  - **local** scope
- Static variables are only initialized *once*

```
void test()
{
   static int bob = 1;
   cout << bob++ << endl;
}

int main()
{
   test();
   test();
   test();
   test();
   test();
}
```