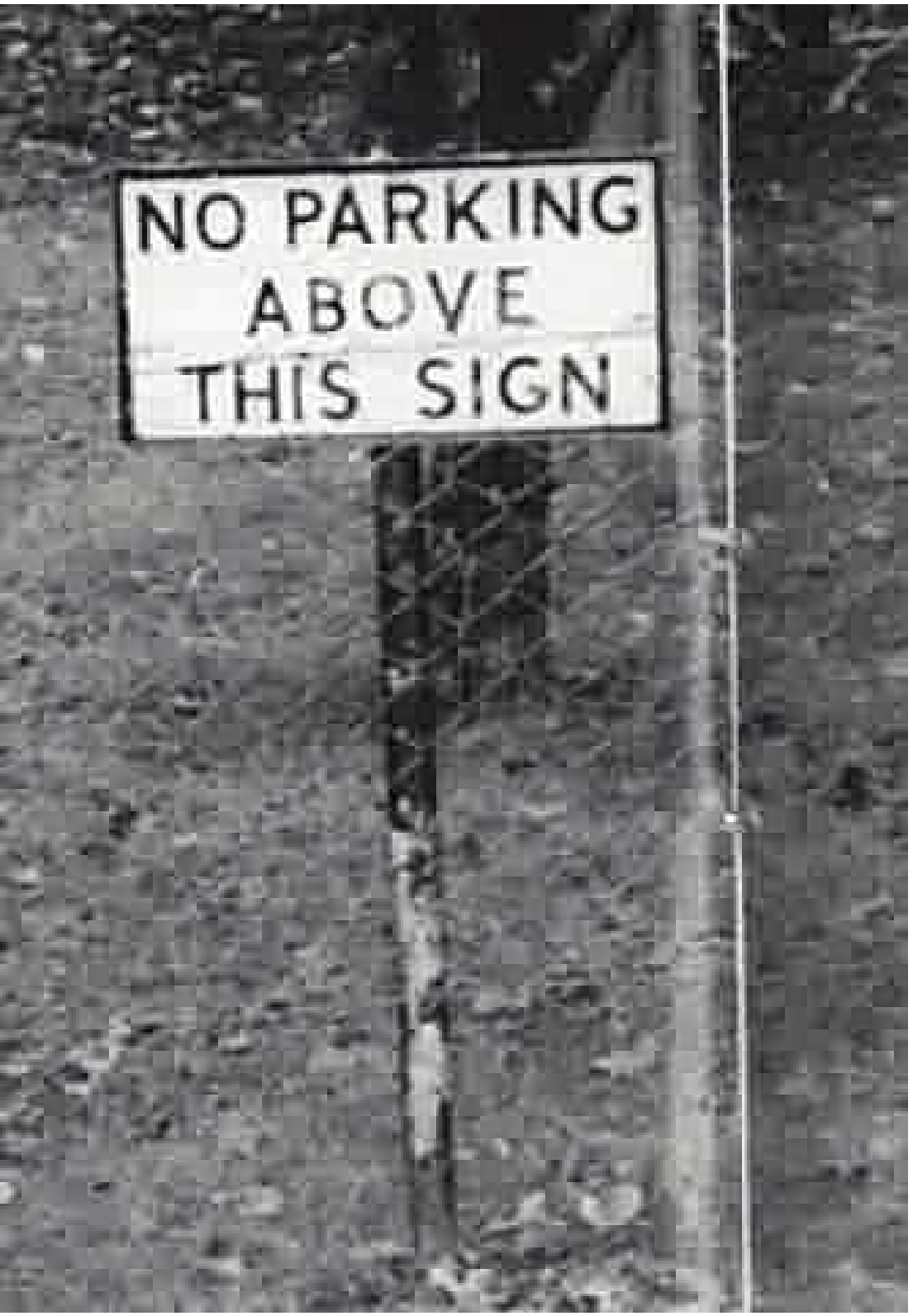


Operator Overloading

Review Questions

- How do we define a constructor?
- What is a copy constructor? How do we define one?
- What sorts of constructors are defined for you automatically (by C++) if you don't define them yourself?
- When is a destructor called?
- What does the keyword "extern" do?

Header Review

- As a review of header files, let's take the myArray class we wrote the last time and stick it into its own set of .cpp and .h files.
- And maybe a few other things…

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&

Side Note: References

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

- A reference is very much like a pointer – it's a reference to a variable, not the variable itself
- Returning an Employee& is like returning a pointer to this object
- This is used for overloaded operators, but usually it's better to use pointers. (Why?)

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[]
```

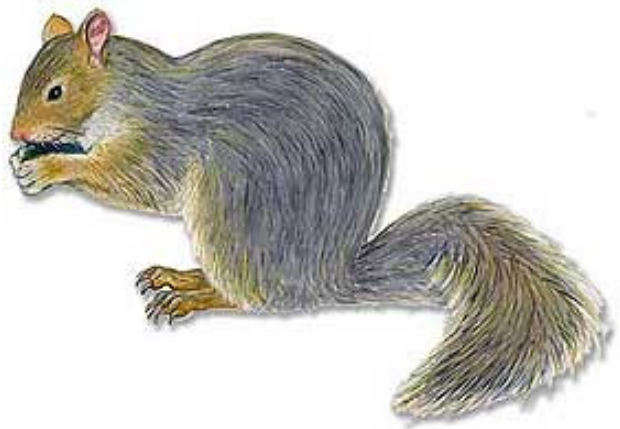
- Aka, if you've overloaded +, you'll probably want to overload += as well.

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 precendence of an operator.

Random Overloading Stuff 2

- You can often figure out the syntax of how to overload an operator just by thinking about it.
- …but otherwise, Google it.
- The way you implement operators sometimes defines the way C++ will let you use them.
- Let's add [] access to myArray!