CONSTRUCTORS & DESTRUCTORS & MORE!!!

Class Review

- What is encapsulation?
- In C++, how is a struct different than a class?
- How do you declare member functions in a class? How do you define them?
- What's the syntax for calling those methods?
- What happens when we mark a method as public? A member variable? How about private?

review - Fun With Code!

- Let's write a circle class with:
  - a radius
  - get/set member functions
  - methods to calculate area and circumference

Question

- So if a variable is declared private (like alpha and beta)...
- Then can outside code - like main() - initialize it?
- If not, how does it ever get initialized?

```
class Data
{
public:
    int getAlpha();

private:
    int alpha;
    int beta;
};
```

Constructors

- This kind of initialization happens through a **constructor**
- A constructor is a special class method that is run when the object is first instantiated
- Purpose of a constructor: to initialize the object, setup any dynamic memory, etc.

Constructors

constructors have:

The *same name* as the class (aka Data)

no return type

```
class Data
{
public:
    Data();
    Data( int a, int b );

    int getAlpha();

private:
    int alpha;
    int beta;
};
```

Note that constructors can be overloaded too -

You can have as many constructors as you need, as long as each one has a unique signature

Constructors

- A constructor with no parameters is called a *default constructor*. That lets you do this:

```
Data d;
```

- The other constructor allows you to do this:

```
Data d(4,5);
```

```
class Data
{
public:
    Data();
    Data( int a, int b );

    int getAlpha();

private:
    int alpha;
    int beta;
};
```

Default Constructors

- You aren't required to define *any* constructors (we didn't in the last class!)
- If you don't define any constructors, C++ will define an empty constructor for you - it doesn't actually do anything
- Once you define *any* constructor then C++ stops giving you the empty one for free

```
class BZisaFoo
{
public:
    BZisaFoo( int a );
};
```

```
// this will not compile
BZisaFoo correct;
```

Default Parameters

- Constructors can have default parameters too
- Like any other C++ function, you have to make sure that constructors aren't ambiguous!

```
class Circle
{
public:
    Circle();
    Circle( float radius = 1.0 );
};
```

which constructor would this use?

```
Circle c;
```

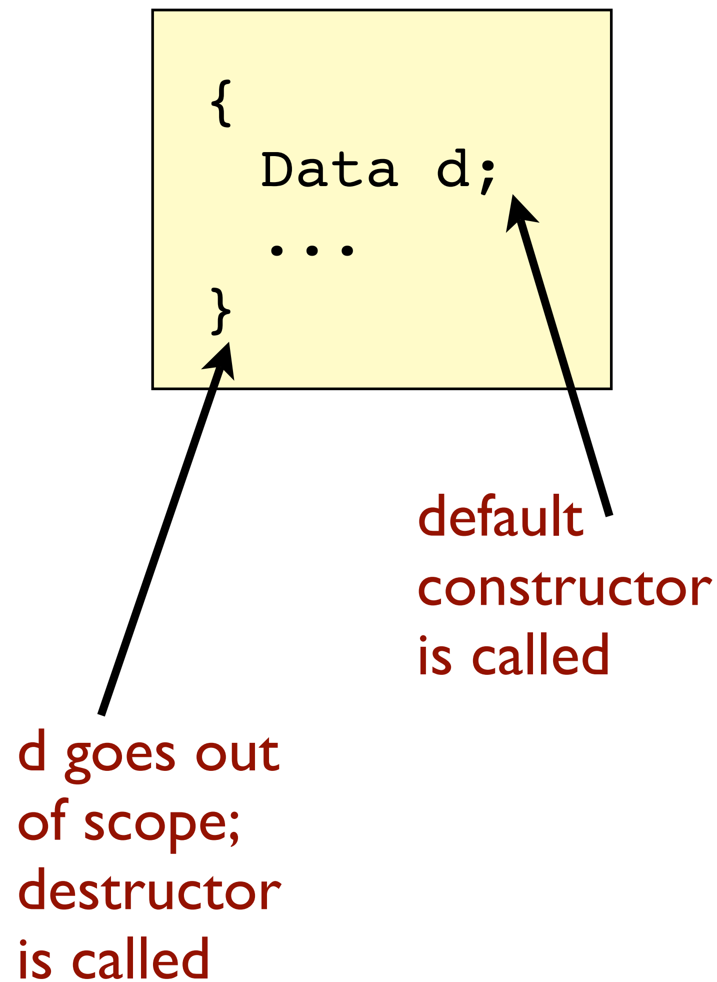
Destructors

- Constructors are called when an object is created...
- A **destructor** is called when the object is deleted.
- A destructor has no return value, and is named after the class, but with a tilde (~) at the beginning.

```
class speaker
{
public:
    speaker();
    ~speaker();
};
```

To Summarize...

- A *constructor* is a special function thatis called when an object is *created*
- A *destructor* is a special function that is called when an object is *destroyed*
  - when the object is manually deleted (via delete)
  - or, when the object goes out of scope



Question

- What's wrong with the following snippet of code:

```
class Circle
{
    int Circle();
    int Circle( float radius );
};
```

```
#include <iostream>
using namespace std;

class printer
{
public:
    printer()
    {
        cout << “CREATE”
             << endl;
    }

    ~printer()
    {
        cout << “DESTROY”
             << endl;
    }
};

int main()
{
    printer a[5];
    return 0;
}
```

Quizlet

- Is this valid code?
- If not, what’s wrong with it?
- What would the output be if it worked properly?

introducing **const**

```
class time
{
public:
     int hour() const;
    void setHour();

private:
    int tHour;
    int tMinute;
    int tSecond;
};

int time::hour() const
{
    // this is an error!
    tSecond = 10;
}
```

- Defining good interfaces also can protect you from your *own* mistakes
- For example... accessor methods that get variables can be marked as read-only, so the compiler will generate an error if that method tries to modify anything in the class
- This is done with C++ keyword **const**, which has been sadly neglected until now

`const` methods

```
class time
{
public:
    int hour() const;
    void setHour();

private:
    int tHour;
    int tMinute;
    int tSecond;
};

int time::hour() const
{
    // this is an error!
    tSecond = 10;
}
```

The keyword **`const`** comes *after* the method name - think of it as part of the function name

It also has to be there in the function definition

Since hour() is marked **`const`**, it can't modify anything in the class without causing a compiler error.

`const` methods

```
int global = 42;

void changeGlobal()
{
    global++;
}

class time
{
public:
     int hour() const;
    void setHour( int h )
    {
        tHour = h;
    }

private:
    int tHour;
    int tMinute;
    int tSecond;
};
```

```
int time::hour() const
{
    return tHour;
}
```

```
int time::hour() const
{
    changeGlobal();
    return tHour;
}
```

```
int time::hour() const
{
    setHour( 11 );
    return tHour;
}
```

Which of these versions of the hour() method will compile?

`const` parameters

- `const` is especially useful for references
- Pass-by-reference is efficient, but leaves parameters open to getting changed in ways you might not expect
- If the function accepts a *const reference*, you have some assurance that parameters will remain unchanged!

```
struct bigData { ... };

int sneakyFunc( bigData& b )
{
    b.count++;
    return b.number*3;
}

int main()
{
    bigData data;
    sneakyFunc( data );
}
```

we can change sneakyfunc to:

```
int sneakyFunc( const bigData& b )
```

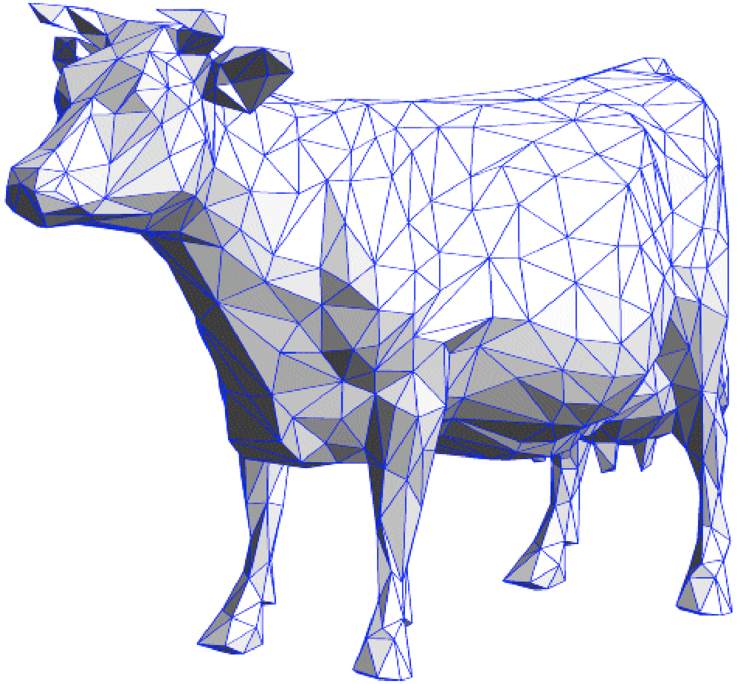
Then it can't change any values inside b, because the parameter is marked const. By adding const we ensure b remains unchanged.

... and finally ...

- There's the const variable. Useful for things like mathematical constants:

```
const float pi = 3.14159265;
```

- Any variable can be declared const; once it is initialized, it can't be changed.



Problems with Pointers

- Problems with pointers:
  - What are they pointing to? Can you be sure it's anything useful?
  - Dereferencing a NULL pointer causes problems
  - Dereferencing a "wrong" pointer also causes problems
  - What happens with uninitialized pointers?
  - All that funny syntax to deal with!

Introducing References!

- **References** are a C++ feature to deal with some of those issues
- A reference variable links to another variable:

```
int normalVariable = 42;
int& reference = normalVariable;

cout << normalVariable << endl;
cout << reference << endl;
```

Here, reference is linked to **`normalVariable`**! **`reference`** doesn't have its own memory location – it just uses **`normalVariable`**'s

Declaring Reference Variables

- A reference variable is declared by sticking an ampersand (&) after the type:

```
int normalVariable = 42;
int& reference = normalVariable;
```

- The same rules apply as for pointers: the “&” only applies to the *first* name to follow it

```
int bob = 42;
int& a = bob, b = bob;
```

- In this example, “a” is a reference – b is *not* a reference, but instead a *copy* of bob

More Reference Declaration Stuff

- Also like pointers, the spacing around the & doesn’t matter:

```
int &a = bob;     // same-same
int& b = bob;
```

- There can be an unlimited number of references to a “normal” variable:

```
int normalVariable = 42;
int& a = normalVariable;
int& b = normalVariable;
int& c = normalVariable;
```

Using Reference Variables

- There is **no difference** between reference variables and regular variables when it comes to usage!

```
int normalVariable = 42;
int& reference = normalVariable;

reference++;
normalVariable++;
```

Reference Rules

- A reference variable:
  - *must* be initialized to another variable
  - *can't* be changed after initialization
    - there's no syntax for doing this!
  - can never be NULL
    - … so no need to worry about dereferencing a NULL pointer
    - Why can a reference never be NULL?

Things to Remember…

- Remember: pointers contain an address!
  - There's a difference between changing the pointer's address and changing the value of what it points to
- Reference variables hide this from you
  - With a reference, you *can't* change the address or what it points to (no pointer arithmetic)
  - You can think of a reference variable as another way to access whatever variable it links to

Not Exactly New

- We’ve seen reference variables before:

```
void swap( int& a, int& b )
{
    int temp = a;
    a = b;
    b = temp;
}
```

- So now that you know what a reference is, what’s *actually* going on here?

Returning References

- A reference is a type (just like any other variable) and can be returned from a function:

```
int& exampleFunction()
{
    int variable = 10;
    return variable;
}
```

- This is valid syntax, but it has a problem – what do we need to be careful of when returning a reference?

So…

- What's *good* about references?
- What's *not so good* about references?
- When would you use a reference ?
- When would you use a pointer?
- How does **const** come in handy when we're dealing with references?

Copying Classes

```
class Square
{
public:
    Square();
    Square( int, int, int, int );
    int area();
private:
    int x, y, w, h;
};
```

Let's say we have an instance of the Square class:

```
Square ted;
```

And we want to copy all its data into a new Square instance that we're creating. Can we do this?

```
Square bill( ted );
```

Sure we can!

- C++ automatically defines a **copy constructor** for each class.
- That copy constructor copies each element of the class individually, by *value*, into the new class

```
class String
{
public:
   String( char* s );
private:

   // dynamically allocated
   char* str;
};
```

- This is often fine, but not always
- Why would we not want to do this with this String class?

Copy Constructors

- We can also define our own copy constructor. It looks like this:

```
class String
{
public:
   String( char* s );
   String( const String& s );
private:

   // dynamically allocated
   char* str;
};
```

The copy constructor accepts a *const reference*.

In this class the copy constructor would allocate memory before copying.

- This copy constructor replaces the default C++ one.

Using Multiple Files

- Most programs have too much code to fit in a single source file
- So how do we separate code into multiple source files?
- We can do this because there's usually a difference between *declaration* and *definition*

Header Files

- We use header files to contain *declarations* of stuff: classes and functions, mainly
- *Definitions* can go in a separate source file
- Any source file that includes the header file can use anything declared in that header
- Each source file is compiled into a separate binary "object file"; they all get linked together in a final linking stage

Example! Example!

Note that when we're #including header files we've made, instead of "standard" ones, we use quotes in our include statement instead of <> brackets

func.h

```
void func();
```

main.cpp

```
#include "func.h"

int main()
{
    func();
    return 0;
}
```

func.cpp

```
#include <stdio.h>
#include "func.h"

void func()
{
    printf( "hi!\n" );
}
```

Other Header Stuff

structs.h

```
#ifndef _STRUCTS_H_
#define _STRUCTS_H_

struct foo
{
};

#endif
```

- or -

structs.h

```
#pragma once

struct foo
{
};
```

- In the "C++ is dumb" category...
- Sometimes you'll see stuff like this in a header file to make sure that the header only gets included once
- If a header is included more than once, the compiler will complain that "foo" is defined more than once

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…
}
```

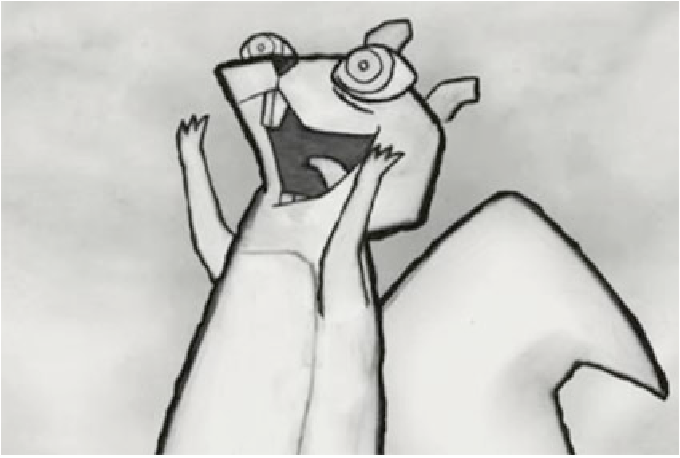
- It works almost exactly like the copy constructor…
- …except this one returns Employee&

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

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

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

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

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

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

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.