



C++ CLASSES

# struct review

- What is the C++ syntax for declaring a structure?
- How do you access the contents of a structure...
  - declared statically?
  - declared dynamically?
- Can structs be passed to a function by value? By reference?
- What are some problems with each method of passing a struct to a function?





# Structures, cont.

- Structures are essentially a concept from C
- They have several limitations:
  - The assignment operator doesn't work - hence, copying them is a pain
  - You can easily have uninitialized data (everybody forgets sometime!)
  - The program using the struct has full access to everything in it

# Full Access

- The program using the struct has full access to everything in it
- Why can this be a problem?
- Sometimes you want to put restrictions on what data a variable can contain:

If you were designing a clock, you'd probably want to ensure that  
 $0 \leq \text{minute} \leq 59$  ...

... but nothing would stop you from writing code like this:

```
Time curTime;  
curTime.minute = 85;
```

# **OBJECT ORIENTED PROGRAMMING**



- Object Oriented Programming (OOP) is a methodology that addresses some of these limitations.
- Structures are intended to hold data, but most problems require both data *and* the logic that operates on that data
- OOP gives us that abstraction, by letting us couple the data together with functions that do stuff with that data.

# OOP Basics

- Let's say you want to do a lot of work with 3D vectors. For a 3D vector, you have:
  - **Data:**  $x, y, z$  (all floating point numbers)
  - **Operations:** addition, normalization, etc.
- We can bring these things together by defining an **abstract data type**.
- But first... how would we use a struct to represent the data parts?

# vector3D as a structure

```
struct vector3D
{
    float x;
    float y;
    float z;
};
```

This is how we would define a struct to handle the data side of things.

- Remember: this is a *definition* of a structure. Here we *define* what data is going to be in the structure - the data itself doesn't exist until we make an *instance* of this structure.
- So how do we add the other part of an ADT - the operations that use this data?

# introducing: class

We define an ADT in C++ using the `class` keyword. Here's the `class` version of the `vector3D` structure:

```
class vector3D
{
public:
    float x;
    float y;
    float z;
};
```

We're now using `class` here instead of `struct`.

We have this new `public` keyword sitting there. We'll get to this in a bit.

Everything else is *exactly the same!* Right now this behaves *exactly* like our `struct` version.



# introducing: class

doesn't work!

```
class vector3D
{
public:
    float x = 3.0;
    float y = -1.5;
    float z = 42.0;
};
```

works fine

```
class vector3D
{
public:
    float x;
    float y;
    float z;
};
```

- Again, like a struct, this isn't an actual usable object yet - it's a *definition* of what an object will look like when we get around to making one of this type.
- So we can't initialize variables here (does it make sense why not?)

# Adding methods

- We've got the data part defined: now we need to need to define the operations part.
- You do that by adding functions that belong to the class (these are often called **methods**, or sometimes **member functions**).
- The point of these methods is to operate on the data within the class.
- Maybe we often need to calculate the length of a 3D vector. How do we add a method to do that?

# Adding methods 2

```
class vector3D
{
public:
    float length();

    float x;
    float y;
    float z;
};
```

- Now we've added `length()`, a method to calculate the Euclidean length of the vector (just an example - the math isn't important).
- Note that this is just a declaration (or prototype) of the function - we still need to define the body of the method!



# Adding methods 3

```
class vector3D
{
public:
    float length();

    float x;
    float y;
    float z;
};

float vector3D::length()
{
    float dist;
    dist = x*x + y*y + z*z;
    return sqrt(dist);
}
```

The body of a function is often defined outside of a class declaration.

We tell the compiler that this function belongs to the class `vector3D` using the *scope resolution operator* (`::`)

```
float vector3D::length()
{
    // body goes here
}
```

# Adding methods #4



```
class vector3D
{
public:
    float length();

    bool isOnFire()
    {
        return false;
    }

    float x;
    float y;
    float z;
};

float vector3D::length()
{
    float dist;
    dist = x*x + y*y +z*z;
    return sqrt(dist);
}
```

We can also define methods within the body of the class itself.

isOnFire() is completely defined within the function declaration; no external body is required (or allowed) for this function.

# Adding methods 5

```
class vector3D
{
public:
    float length();

    float x;
    float y;
    float z;
};

float vector3D::length()
{
    float dist;
    dist = x*x + y*y + z*z;
    return sqrt(dist);
}
```

- *Every* method that belongs to a class *must* be declared in the class declaration!
- This isn't like regular functions, where you can just define a function without giving it a prototype first
- The prototype goes in the class declaration

# Classes and Scope

```
class vector3D
{
public:
    float length();

    float x;
    float y;
    float z;
};

float vector3D::length()
{
    float dist;
    dist = x*x + y*y + z*z;
    return sqrt(dist);
}
```

- Every class defines its own scope:  $x$ ,  $y$ , and  $z$  are all part of `vector3D`'s scope
- Every method in a class has access to that scope
- So, `length()` can access  $x$ ,  $y$ , and  $z$  as if they were local variables
- Can two classes have member variables with the name "distance"?



# Access Controls

- Remember this example from earlier?
- We wanted to avoid letting code set the minute variable to something invalid
- If Time is a struct, nothing prevents us from setting minute to something weird.

If you were designing a clock, you'd probably want to ensure that  
 $0 \leq \text{minute} \leq 59$  ...

... but nothing would stop you from writing code like this:

```
Time curTime;  
curTime.minute = 85;
```





# Public Access

```
class vector3D
{
public:
    float length();

    float x;
    float y;
    float z;
};

int main()
{
    vector3D v;
    v.x = 1.5;
}
```

- This brings us back to the mysterious `public` keyword.
- Any variables declared in the public section can be accessed by *any* part of the program
- Any function in the public section can be called by any part of the program
- `main()` can modify `v.x`, because `x` is public

# Private Access

```
class vector3D
{
public:
    float length();

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

float vector3D::length()
{
    float dist;
    dist = x*x + y*y + z*z;
    return sqrt(dist);
}

int main()
{
    vector3D v;
    v.x = 1.5;    // bad!
}
```

- **private** is another access specifier that we use to “hide” member variables
- The only functions that can access private variables are methods in *that class*
- Likewise, private functions (methods) can only be called by other methods in the same class
- Now, main() can't access x!

# Access Specifiers

```
class vector3D
{
public:
    float length();
    void flip();

private:
    void doPrivateStuff();
    void doubleUp();
    float calcTangent();

    float x;
    float y;
    float z;

public:
    void normalize();
};
```

- **public** and **private** start their own sections: everything in that section has that access attribute (until a new section starts)
- If you don't specify, the default access specifier for a class is *private*
- There's also a third access specifier: **protected**
- We'll talk about that one later on in the course.

# Creating Instances of an Object



- So far, remember, we've just defined the class
- The process of using that class definition to make a real, usable object is called **instantiation** - this allocates memory for the object and lets you do stuff with it (how *much* memory do we need?)
- You make instances of the class the same way you do for any other type:

```
vector3D vec;  
vector3D* ptr = new vector3D;
```

# Using Objects

- Once you've instantiated an object, you access its members the same way you access a structure
- We access class methods the same way: using the dot operator (.) or the arrow operator (->)

```
// declared statically  
vector3D vec;  
vec.x = 42;  
cout << vec.length() << endl;  
  
// declared dynamically  
vector3D* ptr = new vector3D();  
ptr->x = 42;  
cout << ptr->length() << endl;
```

What has to be true in order for this code snippet to compile?



# Accessor Methods

```
class time
{
private:
    int tHour;
    int tMinute;
    int tSecond;
};

int main()
{
    time t;

    // bad!
    t.tHour = 9;

    return 0;
}
```

- Once we've declared variables private, how does outside code (e.g. main) get access to them?
- They can't access them directly...
- The only code that *can* access private variables are member functions.
- So we need member functions to access these variables for us.

# Accessor Methods

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

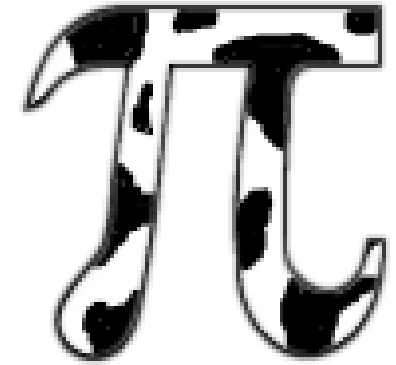
private:
    int tHour;
    int tMinute;
    int tSecond;
};

int main()
{
    time t;
    t.setHour( 9 );
    cout << t.hour();
    return 0;
}
```

- The accessor methods themselves are public, so they can be called by anything
- The set accessor method can also make sure that the data is valid:

```
void time::setHour( int h )
{
    if( h > 12 )
        h = 12;
    if( h < 1 )
        h = 1;
    hHour = h;
}
```

# encapsulation



- A class's *interface* is made up of the public methods you use to interact with the class.
- Accessor methods are used to separate *interface* from *implementation*
- This process is called **encapsulation**.
- The idea: hide all the class implementation details from the code that is using the class - no outside code should *have* to know the details!
- You don't need to know how a car works in order to drive one - just how to use the car's "interface"!



# benefits of encapsulation

- One major reason for encapsulation:
- As long as a class's interface stays consistent, this lets you completely change the way a class works internally and not “break” any code that relies on the class
  - Say we wanted to change our time class to store an *epoch* instead: the number of seconds since midnight.
  - As long as the time interface stays consistent, we just have to write the accessor functions and no other code has to change.

# 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 )
```

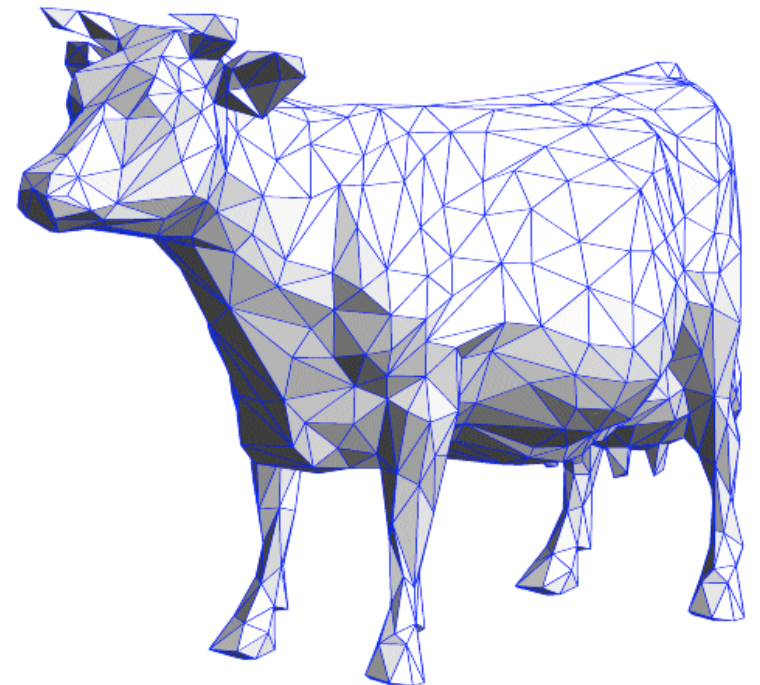
Then it can't change any values inside `b`, because the parameter is marked `const`. By adding `const` we ensure `b` remains unchanged.

# ... 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.





# Fun With Code!

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