Classes

Virtual Methods and Abstract Classes

Automatic Happenings

C++ can provide a default constructor:

- first, it activates the default constructor for the base class
- second, it activates the default constructor for any new member variables that the derived class has but the base class does not

This is only provided if you don’t declare any constructors yourself!

- if you need to implement your own constructor(s), you no longer get the automatic default constructor

Automatic Happenings

C++ still automates some things, even if you do provide constructors

- for example, it will automatically call the default constructor of the base class <u>before</u> executing any constructors of the derived class
- likewise, it will automatically call default constructors for any member variables in the class being initialized <u>before</u> executing the actual constructor itself

We <u>can</u> specify otherwise...

- still, it’s useful to know what is happening when your objects are being created

Automatic Happenings

A simple demonstration:

```
// a simple class
class Character {
    public:
        Character() { }

    private:
        string name;
};
```

Automatic Happenings

A simple demonstration:

```
// create a Character object
Character bob;
```

What happens:

- since this isn't a derived class, no base class constructors are called
- the `name` property (a `string`) gets created via its default constructor
- finally, the constructor for the `Character` class is executed

Automatic Happenings

A demonstration with inheritance:

```
// a child class of Character
class Player : public Character {
    public:
        Player() { }

    private:
        Character side_kick;
};
```

Automatic Happenings

A demonstration with inheritance:

```
// create a Player object
Player josh_wins;
```

What happens:

- this IS a derived class, so C++ first calls the default constructor of the base class (after calling the default constructors of any base-class data members)
- the `side_kick` property (a `Character` object) gets created via its default constructor (after calling the default constructors of any of its data members)
- finally, the constructor for the `Player` class is executed

Automatic Happenings

But what if the base class had <u>no</u> default constructor?

```
// a simple class (without a default constructor)
class Character {
    public:
        Character(const string& n) { name = n; }

    private:
        string name;
};
```

Automatic Happenings

But what if the base class had no default constructor?

```
// a child class of Character
class Player : public Character {
    public:
        Player() { }

    private:
        Character side_kick;
};
```

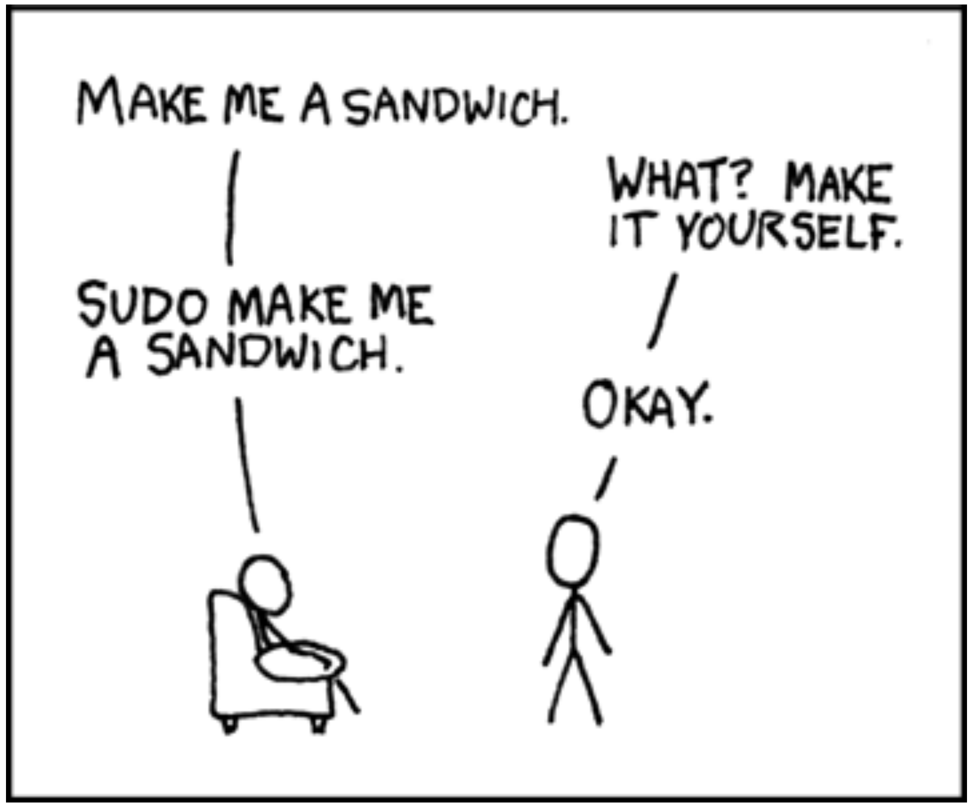
Would this still work?

```
Player fail;
```

I asked, so obviously not...

Why do I even bother?

Gives me something to do, I suppose. =)



Automatic Happenings

Remember what happens when we do this:

```
// create a Player object (but not really)

Player fail;
```

What happens:

- C++ first calls the default constructor of the base class—hey, wait a minute!
- There no longer IS a default constructor of the base class, so this doesn't work
- Likewise, the `side_kick` property (a `Character` object) would get created via its default constructor (oh, wait... it has no default constructor anymore!)

Is there no hope for the Player class?

- unfortunately, there still is hope (I prefer despair and suffering, personally)...

Member Initialization Lists

We can specify which constructors get called in an initialization list

- this takes the same form as we saw earlier
- however, we can also use it to specify which base-class constructor gets called

Example:

```
// using an initialization list
Player() : Character("Cale"),
           side_kick(Character("Richard")) { }
```

This lets us avoid using the default constructor

- we specify which version of the base-class constructor to call with `Character("Cale")`
- we specify the version for the member variable with `side_kick(Character("Richard"))`

Member Initialization Lists

Which of the following two classes is more efficient?

```
// version 1
class Character {
    public:
        Character(const string& n) { name = n; }

    private:
        string name;
};
```

Member Initialization Lists

Which of the following two classes is more efficient?

```
// version 2
class Character {
    public:
        Character(const string& n) : name(n) { }

    private:
        string name;
};
```

Member Initialization Lists

The difference is in the constructors, obviously:

```
// version 1
Character(const string& n) { name = n; }


// version 2
Character(const string& n) : name(n) { }
```

What operations have to happen for each version?

- the first version requires the default string constructor AND the string assignment operator
- the second version requires only the string copy constructor

Member Initialization Lists

To make it all official, here are the rules:

- the member initialization list appears in the implementation of a constructor, after the argument list but before the function body (curly braces)
- the list begins with a colon, followed by a comma-separated list of items to initialize
- for a derived class, the list can contain the name of the base class, followed by an argument list (in parentheses) for a base class constructor
- this list can also contain any member variable, followed by the value that should be used to initialize it (in parentheses)
- when the derived class constructor is called, the constructor for the base class is activated first and then member variables are initialized with the specified values
- if a base class constructor is not activated in the member initialization list, then the default constructor for the base class will automatically be activated automatically

The Member Selection Operator (->)

Lets say you have this object declared:

```
// a Warlock object
Warlock warlock("Richard");
```

Also, assume that the `Warlock` class the following method:

```
// makes things die
void fwoom() const;
```

The `warlock` could call the `fwoom` method like this:

```
// something dies
warlock.fwoom();
```

The Member Selection Operator (->)

But what if warlock were declared not as an object, but as a pointer?

```
// a pointer to a Warlock object
Warlock* warlock = new Warlock("Richard");
```

We now have to dereference `warlock` before calling the method:

```
// something dies
(*warlock).fwoom();
```

A much nicer alternative is to use the member selection operator (->)

```
// something dies
warlock->fwoom();
```

The Member Selection Operator (->)

Compare these two versions:

```
// dereference the pointer, then access member
(*warlock).fwoom();


// dereference the pointer, then access member
warlock->fwoom();
```

The member selection operator (->) is a nice shortcut:

- it first dereferences the pointer on the left-hand side
- then it accesses the member on the right-hand side

Overriding Methods

Given these two classes (each implementing a say_hi method):

```
// parent class
class Human {
    public:
        void say_hi() { cout << "Hi!" << endl; }
};

// child class
class Pirate : public Human {
    public:
        void say_hi() { cout << "Argh!" << endl; }
};
```

Overriding Methods

What do we see when we do this?

```
// declare two objects--one a Human, one a Pirate
Human human;
Pirate pirate;

// call their say_hi method
human.say_hi();  // prints: Hi!
pirate.say_hi(); // prints: Argh!
```

Each object uses the method from its own class

- this is pretty obvious...

Overriding Methods

The base class object can only ever be treated as a `Human`...

- it will only ever use its own method

However, a `Pirate` can be treated like a `Pirate` OR like a `Human`:

```
// declare a pirate object
Pirate pirate;

// declare some pointers
Pirate* p1 = &pirate; // treat him like a Pirate
Human*  p2 = &pirate; // treat him like a Human
```

Dynamic vs Static Binding

Both pointers still point to a Pirate object:

```
Pirate* p1 = &pirate; // treat him like a Pirate
Human*  p2 = &pirate; // treat him like a Human
```

The `Pirate` pointer will use the `Pirate` method:

```
p1->say_hi();  // via Pirate pointer
```

However, C++ can handle the base-class pointer two different ways:

```
p2->say_hi();  // via Human pointer
// what gets displayed?
```

Dynamic vs Static Binding

Given a base-class pointer to a derived class object:

```
Pirate pirate;         // declare a Pirate object
Human* ptr = &pirate; // treat him like a Human
```

Static binding uses the parent class (`Human`) method:

```
ptr->say_hi();  // prints: Hi!
```

Dynamic binding uses the derived class (`Pirate`) method:

```
ptr->say_hi();  // prints: Argh!
// ... even though ptr is a pointer to a Human!
```

Dynamic vs Static Binding

Static binding occurs at compile-time

- the compiler can only use the declared type of the variable to make its decision
- it will always use the method that belongs to the class declared for the variable
- static binding for a pointer to a `Human` will always use the method provided in the `Human` class, even if the pointer actually points to a derived-class object (a `Pirate`)

Dynamic binding defers the decision until run-time

- while your program is running, it can check whether a pointer points to a base-class object or to a derived-class object
- if it points to a base-class object, it uses the method in the base class
- if it points to a derived class object, it uses the method in the derived class
- this means that a `Human` pointer that points to a `Human` object will use the `Human` method, while a `Human` pointer that points to a `Pirate` object will use the `Pirate` method

How do you tell C++ which version you want?

Static Binding

The say_hi method (using static binding):

```
// parent class
class Human {
    public:
        void say_hi() {cout << "Hi!" << endl;}
};

// child class
class Pirate : public Human {
    public:
        void say_hi() {cout << "Argh!" << endl;}
};
```

Dynamic Binding

The say_hi method (using dynamic binding):

```
// parent class
class Human {
    public:
        virtual void say_hi() {cout << "Hi!" << endl;}
};               virtual keyword

// child class
class Pirate : public Human {
    public:
        virtual void say_hi() {cout << "Argh!" << endl;}
};               virtual keyword
```

Virtual Methods

To use dynamic binding, make a method virtual

- this tells C++ that it must evaluate which method to use at RUNTIME, rather than when it compiles your code
- the parent must declare its method as virtual for a child class to override it in this way
- if a derived class reimplements a virtual method declared by a parent class, its implementation automatically becomes virtual, too—regardless of whether you declare it virtual or not

Dynamic binding applies only to pointers and reference variables

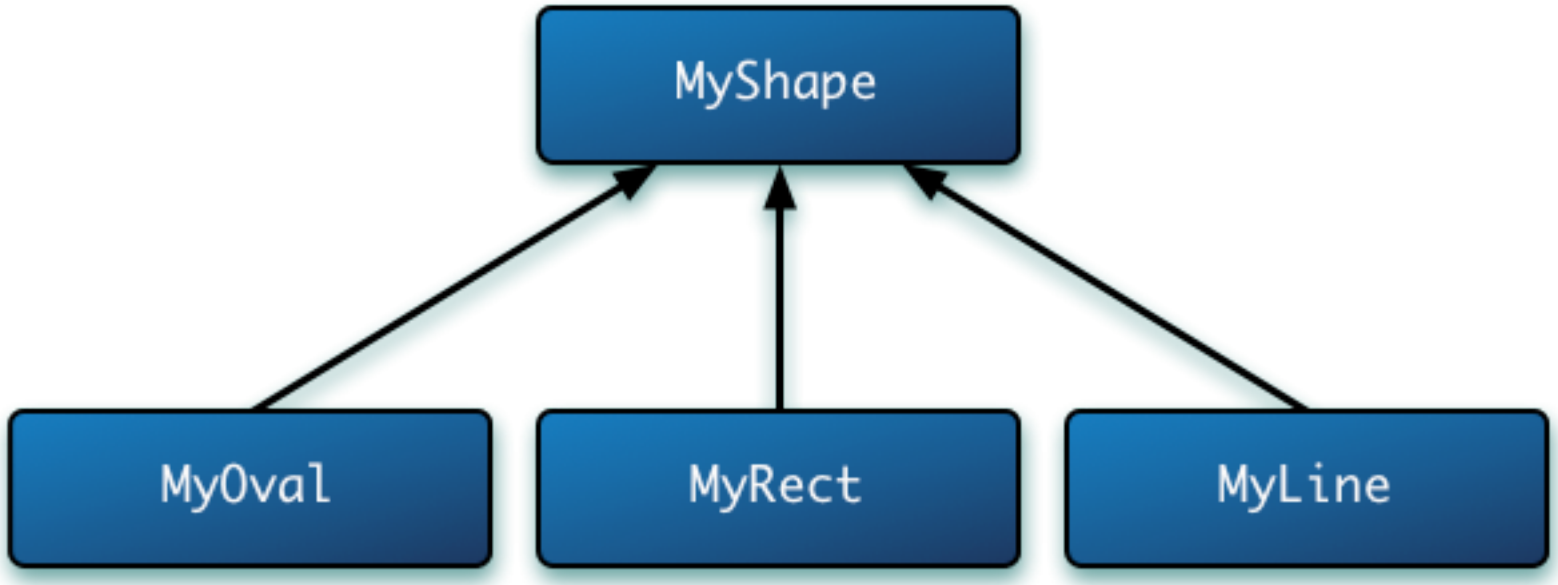
- a pointer is just a memory address (where it points could potentially be anything)
- a reference is just an automatically dereferenced pointer (nice feature of C++)
- non-pointer, non-reference variables can <u>only</u> store the data type they declare, so no dynamic binding is possible

Okay, that’s great and all...

But why would this ever be useful?

Polymorphism

Let's say you wanted to have an array containing various shapes...



Given the class hierarchy above, you could do this:

```
// array that can contain pointers to MyShape objects
MyShape** shapes = new MyShape*[3];
```

Each derived class IS A MyShape, so the array could point to them, too

Polymorphism

Let's say you wanted to have an array containing various shapes...

```
// array that can contain pointers to MyShape objects
MyShape** shapes = new MyShape*[3];

// insert various shapes into the array
shapes[0] = new MyRect;
shapes[1] = new MyOval;
shapes[2] = new MyLine;
```

Each derived object must be drawn differently, however...

- each derived class will implement its own unique version of `draw`
- dynamic binding must be used to call these methods, since the array contains pointers to base-class objects... but does an individual pointer point to a rectangle, an oval, or a line?

Polymorphism

Corresponding class declarations:

```
// parent class
class MyShape {
    public:
        virtual void draw(Graphics^ g) const;
};

// MyRect class
class MyRect : public MyShape {
    public:
        void draw(Graphics^ g) const; // draw a rectangle
};
```

Polymorphism

Corresponding class declarations:

```
// parent class
class MyShape {
    public:
        virtual void draw(Graphics^ g) const;
};

// MyOval class
class MyOval : public MyShape {
    public:
        void draw(Graphics^ g) const; // draw an oval
};
```

Polymorphism

Corresponding class declarations:

```
    // parent class
    class MyShape {
        public:
            virtual void draw(Graphics^ g) const;
    };

    // MyLine class
    class MyLine : public MyShape {
        public:
            void draw(Graphics^ g) const; // draw a line
    };
```

Polymorphism

Given these virtual methods, this should do what we want:

```
// array that can contain pointers to MyShape objects
MyShape** shapes = new MyShape*[3];

shapes[0] = new MyRect;
shapes[1] = new MyOval;
shapes[2] = new MyLine;

// have each shape draw itself (magically works!)
for (int i = 0; i < 3; i++) {
    shapes[i]->draw();
}
```

Polymorphism:

Multiple meanings associated with a single function

Abstract Classes

Does it make sense to do this?

```
// create a MyShape object
MyShape* shape = new MyShape;


// draw the MyShape
shape->draw();
```

How should a `MyShape` draw itself?

- the `MyRect`, `MyOval`, and `MyLine` classes represent concrete ideas; `MyShape` does not
- when a base class shouldn't be instantiated because it <u>doesn't fully describe something</u>, it is considered an *abstract* class

Pure Virtual Methods

We can enforce the idea of an abstract class with pure virtual methods

- pure virtual methods MUST be implemented by derived classes
- the base class (abstract class) only specifies that derived classes must implement a pure virtual method; it does not provide its own implementation

The syntax for a pure virtual method:

```
// abstract parent class

class MyShape {
                                                        pure virtual
    public:                                                 |
                                                            v
        virtual void draw(Graphics^ g) const = 0;
          ^
};        |_____ virtual keyword
```

Pure Virtual Methods

We can enforce the idea of an abstract class with <u>pure virtual</u> methods

- pure virtual methods MUST be implemented by derived classes
- the base class (abstract class) only specifies that derived classes must implement a pure virtual method; it does not provide its own implementation

A class with a pure-virtual method CANNOT be instantiated

```
// try to create an instance of an abstract class...
MyShape shape; // compiler error!
```

```
// error: cannot declare variable 'shape' to be of abstract type 'MyShape'
//        because the following virtual functions are pure within 'MyShape':
//        virtual void MyShape::draw(Graphics^) const
```

Virtual Methods

Virtual methods:

```
virtual void update();          // non-const method
virtual void print() const;     // const method
```

Pure virtual methods:

```
virtual void update() = 0;        // non-const method
virtual void print() const = 0; // const method
```

Virtual methods allow dynamic binding (decision at runtime)

- if a base class can exist on its own, but derived classes should be able to specify their own behavior, make the appropriate methods virtual
- if a base class is abstract and *should not* be instantiated, make the appropriate methods pure virtual

Multiple Inheritance

Classes in C++ can derive from multiple base classes:

```
// base classes
class Ninja { ... };
class Pirate { ... };

// a NinjaPirate class!
class NinjaPirate : public Ninja, public Pirate {
    // be afraid!
};
```

Simply use a comma-separated list of base classes

- notice that each base class has its own visibility specifier