Inheritance

Introduction

Inheritance is a way to form new classes using classes that have already been defined

New class specieis which class it's based on and "inherits" all of the base class's functions/data for free, can add its own new functions/data

Idea

- First define the **superclass** and implement all its member functions
- Then define the **subclass** explictly basing it on the **superclass**
- Finally add new variables and member functions

Three Uses of Inheritance

Resuse: write code once in base class, reuse the same code in derived classes

- Every **public** method in the base class is automatically reused in the dirved class
- **Private members** in the base class are hidden
- **Protected members**: derived class able to reuse 'private' member function but not rest of the program
  - Derived class can use as a public member, but...

    ```
    //FAILS
    stan.chargeBattery(); //in main()
    ```

  - NEVER make member variables protected: a class's member variables are for it to access alone

Extension: add new behaviors or data to derived class

- Extensions in derived class is unknown to base class

Specialization: redefine an existing behavior from the base class with a new behavior in derived class

- `virtual`
- Derived class will be default always use the most derived version of a specialized method

  ```
  class NerdyStudent : public Student
  {
  ```

```
    public:
        virtual void cheer()
        {
            cout << "go algorithms!";
        }
        void getExcitedAboutCS()
        {
            cheer(); //calls Nerdy's cheer()
        }
};

int main()
{
    NerdyStudent lily;
    lily.getExcitedAboutCS();
}
```

- If want to use base class's version:

```
void getExcitedAboutCS()
{
    Student::cheer();
}
```

- Derived class method reuse base-class method that itoverrides

```
class NerdyStudent : public Student
{
    public:
        virtual string whatILike()
        {
            string fav = Student::whatILike();
            fav += " bunsen burners";
            return fav;
        }
};
```

Inheritance & Construction

When you define a derived object, it has both superclass and subclass parts

> C++ always constructs the base part first, then the derived part second

Example

```
//base class
class Robot
{
    public:
        Robot()
            Call m_bat's constructor
        {
            m_x = m_y = 0;
        }
    private:
        ...
};

//derived class
class ShieldedRobot : public Robot
{
    public:
        ShieldedRobot()
            Call Robot's constructor
            Call m_sg's constructor
        {
            m_shieldStrength = 1;
        }
    private:
        int m_shieldStrength;
        ShieldGenerator m_sg;
};

int main()
{
    Shielded Robot phyllis;
}
```

- Steps:
    1. First call base class constructor
    2. Then construct derived object's member variables
    3. Then run body of derived constructor

    ```
    Robot's data:
    m_x = 0 m_y = 0
    m_bat FULL

    ShieldedRobot's data:
    m_sg ON
    m_shieldStrength = 1
    ```

Inheritance & Destruction

> C++ destructs the derived part first, then the base part second

Example

```
class Robot
{
    public:
        ~Robot()
        {
            m_bat.discharge();
        }
        Call m_bat's destructor
    ...
};

class ShieldedRobot : public Robot
{
    public:
        ~ShieldedRobot()
        {
            m_sg.turnGeneratorOff();
        }
        Call m_sg's destructor
        Call Robot's destructor
    ...
};
```

- Steps
    1. Run derived object destructor body
    2. Destroy data members
    3. Call base class's destructor

Inheritance & Initializer Lists

> When base class does not have a default constructor

Example

```
class Animal
{
    public:
        Animal(int Ibs)
        { m_Ibs = Ibs; }

        void what_do_i_weigh()
        { cout << m_Ibs << "Ibs!\n"; }
    private:
        int m_Ibs;
```

```
};

class Duck : public Animal
{
    public:
        Duck()
            : Animal(2), m_belly(1)
        { m_feathers = 99; }

        void who_am_i()
        { cout << "A duck!"; }
    private:
        int m_feathers;
        Stomach m_belly;
};
```

- Derived class calls base class's default constructor if there is *no initializer lists*
- The first item in the initializer list MUST BE the name of the base class

```
Duck(int Ibs) : Animal(Ibs)
{
    m_feathers = 99;
}
```

Multiple Layers of Inheritance

Example: Animal -> Duck -> Mallard

```
class Duck : public Animal
{
    public:
        Duck(int Ibs, int numF) :
            Animal(Ibs - 1)
        { m_feathers = numF; }

        void who_am_i()
        {
            cout << "A duck!";
        }
    private:
        int m_feathers;
};

class Mallard : public Duck
{
    public:
        Mallard(string &name) :
            Duck(5, 50)
```

```
        { myName = name; }
    private:
        string myName;
};
```

Inheritance & Assignment Operators

Assigning one instance of a derived class to another

```
ShieldedRobot larry, curly;
larry.setShield(5);
larry.setX(12);
larry.setY(15);

curly.setShield(75);
curly.setX(7);
curly.setY(9);

larry = curly;
```

- C++ first copies the base data from curry to larry, then copies derived data from curly to larry

> Works fine only in cases where all members are **not dynamically allocated & have defined assignment operators**

Defining assignment operator

```
class Person
{
    public:
        Person() { myBook = new Book; }
        Person(const Person &other);
        Person& operator=(const Person &other);
        ...
    private:
        Book *myBook;
};

class Student : public Person
{
    public:
        Student(const Student& other)
            : Person(other)
        {
            ...//make a copy of other's linked list of classes
        }

        Student& operator=(const Student& other)
```

```
        {
            if (this != &other)
            {
                Person::operator=(other);
                //do copy swap
                return *this;
            }
        }
    private:
        LinkedList *myClasses;
}
```

Scenario: Write a system that will let me draw pictures

- Different shapes: circle, line segments, triangles, etc.
- Assume that different shapes will have their separate properties -> will be different classes
- Picture is a collection of all these shapes
- Pseudocode:
    - pic (empty to start out)
    - add a Circle to pic
    - add a Rect to pic
    - add a Cirlce to pic
- Demo Code

```
class Circle
{
    void move(double xnew, doubleynew);
    void draw() const;
    double m_x;
    double m_y;
    double m_r;
};

class Rectangle
{
    void move(double xnew, double ynew);
    void draw() const;
    double m_x;
    double m_y;
    double m_dx;
    double m_dy;
};

Circle* ca[100];
Rectangle* ra[100];
```

```
//Repeat all this code, so many repeated code...
void f(Circle& x)
{
    x.move(10, 20); //we need a different version of move for each shape
    x.draw();
}

void f(Rectangle& x)
{
    x.move(10, 20);
    x.draw();
}

Circle c;
c.move(15, 5);
c.draw();
f(c);

ca[0] = new Circle;
ra[0] = new Rectangle;
ca[1] = new Circle;

for (int k = 0; ...; k++)
{
    ca[k] -> draw();
}
for (int k = 0; ...; k++)
{
    ra[k]->draw();
}
```

- How to get rid of repeated code?
    - Have only one array `??? pic[100];`
    - Have only one `f` function `void f(???& x)`
    - Idea: `pic` is a collection of `shape`
        - A `circle` is a kind of `shape`, and a `rectangle` is a kind of`shape`
- One type and another type are all kinds of some other type...
- Introduce the type `shape`
    - `pic` array will hold shapes
    - Can create any shapes in dynamic allocation

```
class Shape
{
```

```
};

class Circle : public Shape // A Circle is a kind of Shape
{

};

class Rectangle : public Shape // A Rectangle is a kind of Shape

Shape* pic[100];

pic[0] = new Circle;
pic[1] = new Rectangle;
pic[2] = new Circle;
```

- Relationship between `Shape` and the actual shape classes:

  `Shape` : generalization

  `Circle` and `Rectangle`: specialization

  - `Shape`: **base class**
  - `Circle`: **derived class**

Single Inheritance

- **Heterogeneous Collection**: Pointer to a derived object can be automatically converted to a base object

```
Derived* ==> Base*
Derived& ==> Base&
```

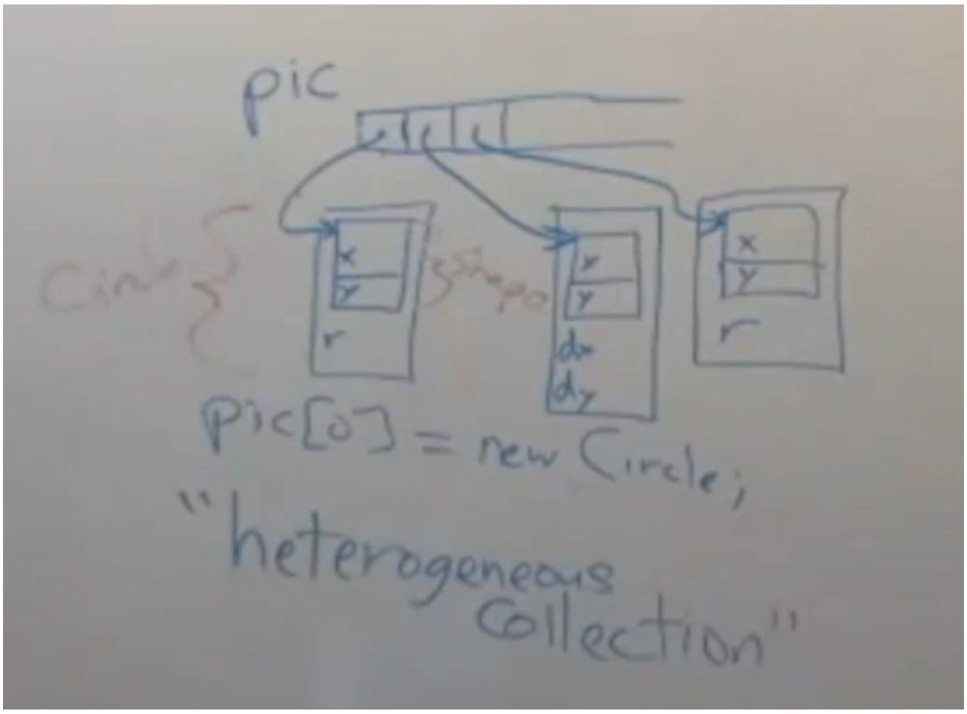
  - An array of the base class can store objects of the derived class
  - If I say you can bring me a mammal, you can bring me a dog
    - If I say you can bring me a Shape, you can bring me a Circle
  - The opposite does NOT work
    - If I say you can bring me a dog, you can't bring me other mammals!
  - Underlying Compiler Logic
    - Actual implementation
      - The memory representation of every object of the derived type has an object of the base type embedded inside it

    - Every `Circle` object will have embedded inside it a `Shape` object

- `pic[0] = new Circle`

    - `new Circle` gives a pointer to the `Circle` object
    - This pointer is converted to a pointer to a `Shape`, then assigned to `pic` array
    - Implementation: Compiler sees this, every Circle object will have a Shape object inside it
        - the Shape part of every Circle could be e.g. always 4 bytes down from the top of the object
        - Compiler generate code:
            - take the address of this circle, add 4 bytes to it, store into pic array
        - Compiler wants this conversion to be CHEAP
            - Add constant to an address is cheap already
            - But if constant = 0 -> cheaper (no instructions)
                - Most compiler will put the base object at the top of the derived object

              The start of the base object inside is the same as the satrt of the derived object
    - The pointer that is assigned to the `pic` array is *actually pointing to the Shape object inside the circle*

- What is common for all derived objects -> base type
    - All `Shapes` can moved, can be drawn, have `x` and `y`

    ```
    class Shape
    {
        void move(double xnew, double ynew);
        void draw() const;
        double m_x;
        double m_y;
    }

    class Circle: public Shape //in addition to all properties of shape,
    has this
    {
        double m_r;
    }

    class Rectangle : public Shape
    {
        double m_dx;
        double m_dy;
    }
    ```

- Inheritance: The derived class inherits all properties of the base class
    - Functions and data members

Implementations of Base and Derived Functions

Consider this code... `move()`

```
void Shape::move(double xnew, double ynew)
{
    m_x = xnew;
    m_y = ynew;
}

Circle c;
c.move(15, 5);
f(c);
```

- Circle object `c` is declared
- `c.move` will change the x and y coordinates of `c`
- `f(c)` takes reference to a Shape object -> automatic conversion
    - In the function, `x` will be a reference to the Shape part of the ciccle

        - `x.move()` changes the x and y

- This works perfectly!
    - All it takes to move all shapes is moving the reference points, so there is no difference between moving any shapes, this implementation works
- Same `move()` function for every type of Shape

Consider this... `draw()`

- Every `draw()` function is different

```
class Circle : public Shape
{
    void draw() const;
    double m_r;
};

class Rectangle : public Shape
{
    void draw() const;
    double m_dx;
    double m_dy;
};

void Circle::draw() const
{
    ...draw a circle of radius m_r...
}

void Shape::draw() const
{
    ...for now, just draw a vague cloud centered at m_x, m_y...
}

c.move(15, 5);
```

- We have to implement a `Shape::draw()` function because we declared it
    - Even though we don't know how to draw just a shape
    - If we don't need to draw a shape, can we just not declare it in Shape? **NO**
        - `pic` holds a collection of Shapes, compiler does not know whether every Shape object can be drawn or not
            - Due to separate compilation
        - Code will NOT compile
- `Circle::draw()` needs to override the `Shape` implemention

Implementing Base Class Function: `Shape::draw()`

Static Binding Versus Dynamic Binding

- When the compiler sees that there is a base reference or pointer that calls a member function that each derived type has defined differently, how does it know which one to call?
    - We only know which version to call at runtime since the array can have different types stored
    - Decision:

      Should the compiler compile code that will at compile-time already has built-into it exactly which function to call or should it compile code that will make a decision at runtime to decide which function to call?
- Static binding
    - Compile-time, bind which function body each call will execute
    - `move()`: the same across all
    - `draw()` when something like `c.draw()` since caller is a Circle object, obvious to bind Circle's draw function
- Dynamic binding
    - Decide during runtime, compiler generate code that decide during execution which function body to call
    - `draw()` when called through a pointer/reference
        - when we don't know what kind of shape the caller is referring to
- Most programming languages do not have this issue
    - Types are known at runtime
    - Or does dynamic binding built-in
        - In cases where it is not needed -> costly
    - In C++ static binding is the default
        - For the `draw()` function: whenever any reference calls draw, Shape's draw() will be called -> PROBLEM!

Ensuring Dynamic Binding... The `virtual` keyword

```
class Shape
{
    void move(double xnew, double ynew);
    virtual void draw() const;
    double m_x;
```

```
    double m_y;
}
```

- Keyword `virtual` tells compiler that this function will be dynamically bound
  - Required only in the base class, but convention to write it in ALL classes

    ```
    class Cirlce : public Shape
    {
        virtual void draw() const;
        double m_r;
    }
    ```

Virtual Functions

Calling Base Class Function from Derived Class Function

Consider this code...

I create a new type `warningSymbol` that needs a new `move()` function

```
void warningSymbol::move(double xnew, double ynew)
{
    Shape::move(xnew, ynew);
    //or this->Shape::move(xynew, ynew);
    ...flash three times...
}

warningSymbol ws;
ws.move(15, 5); //static binding
```

However...

```
void f(Shape& x)
{
    x.move(10, 20);
}

f(ws);
```

- Warning symbol will not move correctly
  - Will not flash when call through reference/pointer of base class
  - Inconsistency!

- This is because we overrided a non-virtual function: we have to change the `move()` function to `virtual`

  ```
  virtual void move(double xnew, double ynew);
  ```

> **Never override a non-virtual function**

New Functions of Derived Classes

Consider this code...

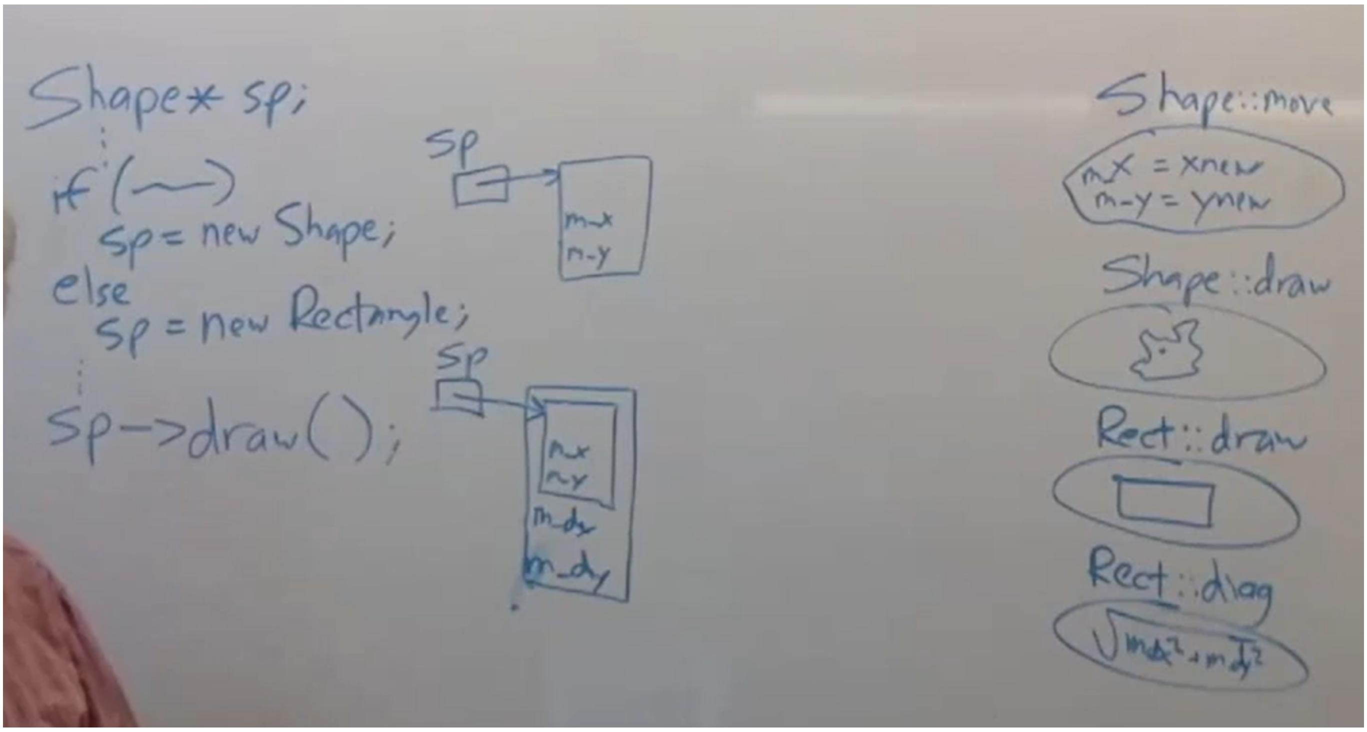
```
class Rectangle : public Shape
{
    virtual void draw() const;
    virtual double diag() const;
    double m_dx;
    double m_dy;
}

double Rectangle::diag() const
{
    return sqrt(m_dx * m_dx + m_dy * m_dy);
}
```

- Why is `diag()` virtual?
  - We might want derived class of Rectangle (e.g. Square) that wants a diagonal function
    - We can figure out a diagonal of a square more efficiently (just a side * square root of 2)

Compiler Implementation of Virtual Functions

Consider this scenario...

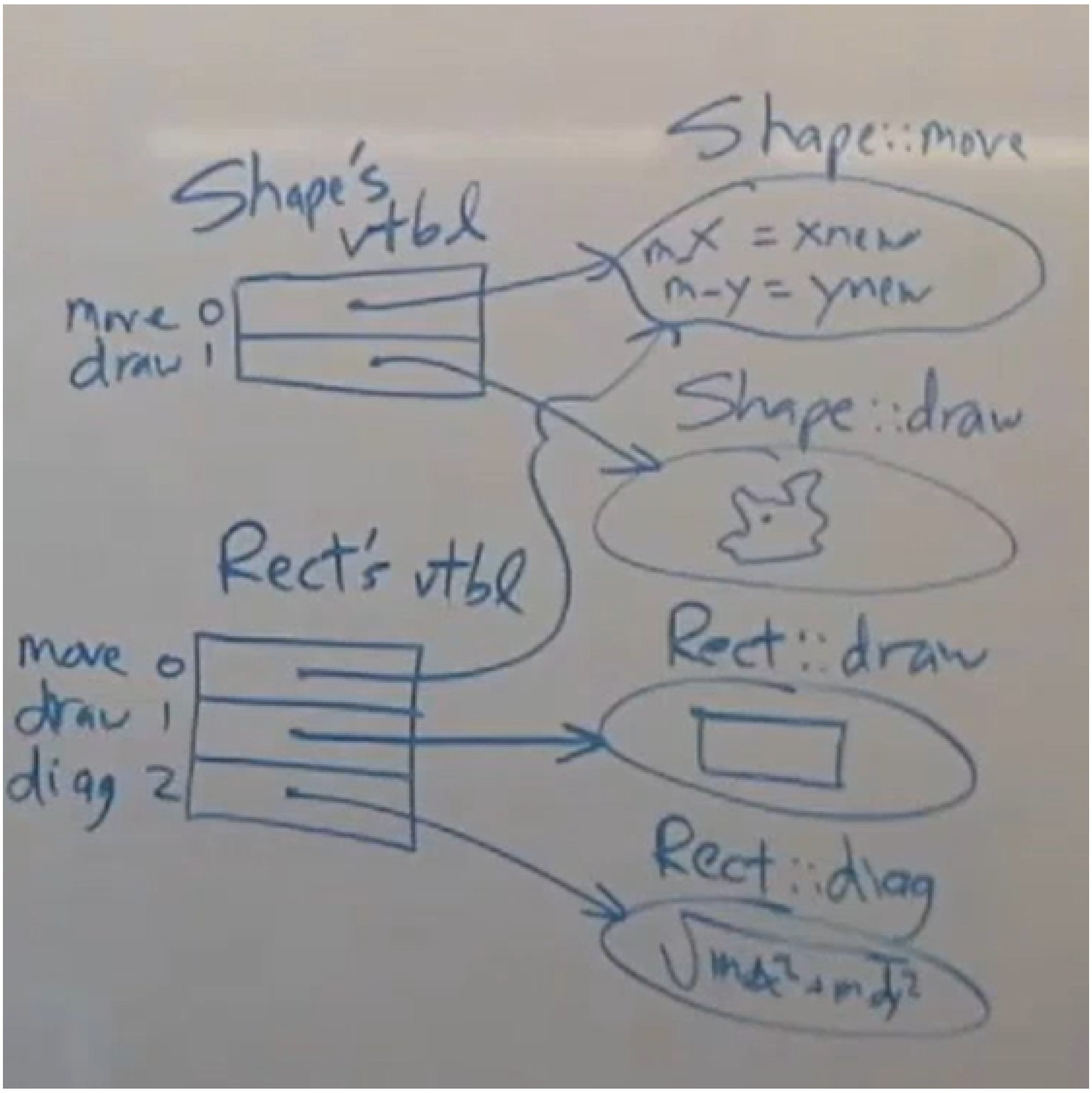


Since `sp` points to a `Shape` object, it doesn't know by default that this `Shape` object is in a `Rectangle`

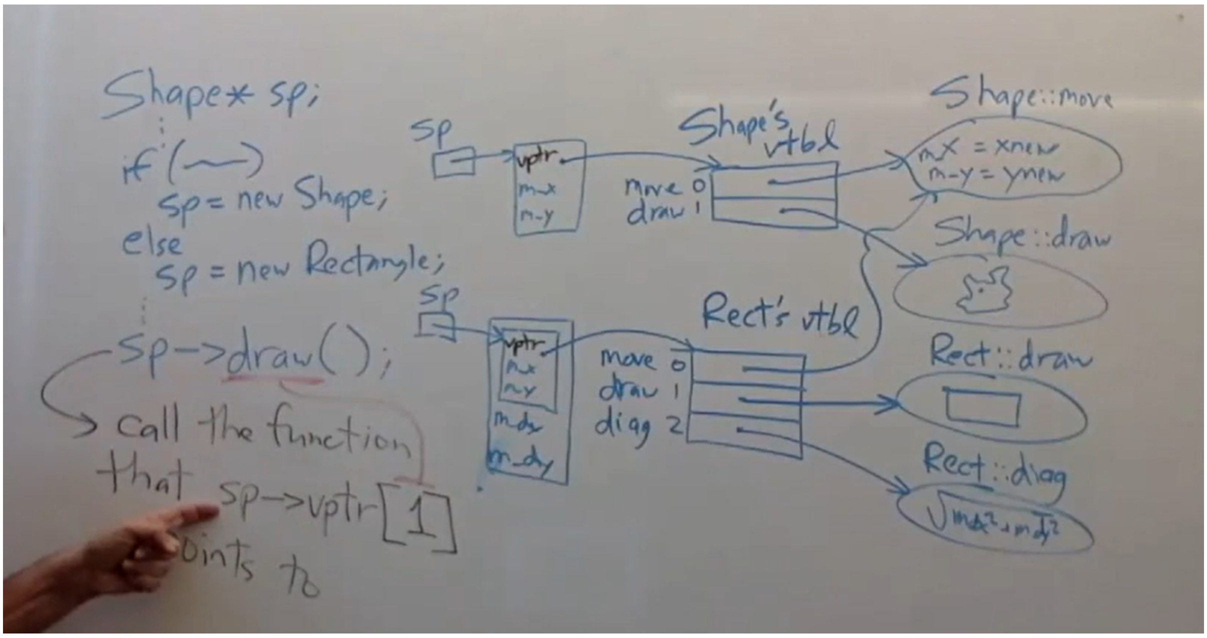
So how does the compiler figure out this is a rectangle and call the corresponding function?

- Method 1: (The C Way)
  - Add an extra member to `Shape` that is an integer code etc. that encodes what kind of `Shape` it is
    - Just a Shape: type 0, Circle: 1, Rect: 2, etc...
  - Compiler:
    - Follow the pointer to the `Shape` object
    - Look into code number
      - Some switch statement -> based on the code number call a function
  - Disadvantage:
    - Everytime you invent a new type, we have to add a new code number for this shape
    - If some old code already compiled (.o file) -> this file will be **incorrect**
    - Recompile every piece of code that uses `Shape` -> COSTLY!!!!!
- Method 2: **The Virtual Table (VTBL)**
  - Compiler set up table for a class with virtual functions
    - One entry for each virtual function (arbitrarily chosen at first)
    - For the `Shape` class: two virtual functions
      - In the slots: pointer to the functions (how to execute the functions)

    - For the `Rectangle` class: three virtual functions (2 inherited)
        - Entry that is associated with the function *must be identical* from the base
            - `Rectangle` associated `move()` with 0, `draw()` with 1
            - Additional functions are arbitrarily associated
            - Every derived class will have the same entry for inherited virtual functions
        - What's the right way to draw? `Rect::draw`
        - What's the right way to move? Rectangle no declare -> base class



  - Compiler now knows that the `draw()` function is associated with entry 1
    - Call the function that slot 1 points to... but WHICH??
  - Virtual pointers that are stored in the `Shape` class under the hood
    - Virtual pointer points to the corresponding virtual table



  - They still work with new additional stuff

`Shape`'s `draw()` function: Is the implementation necessary?

- We HAVE to define it because compiler needs to know that this function exists when there is a pointer to `Shape`
  - `Shape` has a designated slot in virtual table for this function and that entry will be the same for all derived classes
- Declare this function, but NOT implement it
  - Do we have a `nullptr` for `draw()` in `Shape`'s virtual table?

Pure Virtual

```
class Shape
{
    virtual void draw() const = 0;
};
```

- `draw()` part of `Shape` interface, but no implementation
- Borrows C syntax
- Under the hood implementation
  - Compile-time error when `draw()` is called on `Shape` (which is pure virtual, not defined)
    - When call -> if arrive at `Shape`'s virtual table -> TROUBLE
    - Objects that are just plain Shapes and nothing more will result in this
      - YOU CANNOT CREATE AN OBJECT OF `SHAPE` TYPE -> WILL NOT COMPILE

```
sp = new Shape; //CANNOT EXIST
```

- Cannot:
    1. Declare a variable of `Shape`
    2. Dynamically allocate a `Shape`
- Can:
    1. Pointers to `Shape`

Abstract Base Class

We cannot create an object that is an ABC

> I want a mammal, not a cat, a dog, something that is JUST a mammal does not exist

- Abstraction -> generalization of somethings
- You never want to create an object of an abstract type because it just does not make sense

> An abstract class cannot be instantiated

Not defining a virtual function that was pure in a derived class

Suppose I forgot to define `draw()` for a `Rectangle`

- `Shape` never defined it, it is pure virtual
- `Rectangle` will also inherite the **pure virtual function** -> **Rectangle is an abstract class too**
    - THAT'S A PROBLEM!!!!

Polymorphism

"many forms, many shapes"

- A particular object can have more than one type
    - A `Dog` is also a `Mammal` and an `Animal` and maybe a `LivingThing`
    - Inheritance hierarchy
- The same function name can have different implementations
    - All called the same way, but call different versions of them
    - The function is *polymorphic*

Virtual Destructors

Example

I have a `Polygon` that is a type of `Shape`, which contains a linked list that needs dynamically allocation

```
class Polygon : public Shape
{
    ~Polygon();
    Node* head;
}
```

Hence we need a destructor for `Polygon`

The destructor for `Shape` is just default, because we don't need to define one

Problem walkthrough

```
Shape* sp;
sp = new Polygon;
sp = new Somethingelse;

delete sp;
```

- Compiler needs to find the correct destructor for the dynamically allocated objects
- To destroy `Polygon`, needs to call `Polygon` destructor for the linked list to be well deleted
- But the compiler at compile-time DOES NOT KNOW which object `sp` points to
    - So the choice of which destructor is called made at compile-time is statically bound -> will just call `Shape`'s destructor
    - Then our linked list nodes never destroyed -> MEMORY LEAK
- We need compile to decide at runtime which destructor is called -> **virtual destructor**
- We want compiler-generated destructor for `Shape` to be virtual -> then we need to declare it in `Shape`!!!
    - Because the default is that the `Shape` destructor is not virtual
        - C++ for efficiency -> cost of virtual functions involved when even no inheritance

```
class Shape
{
    virtual ~Shape();
};
```

- Slot in the virtual table for destructor -> designated
    - Then the destructor of any derived class will have the same slot number

> If a class is designed to be a base class, declare a destructor for it, and make it virtual

What about ABC?

- If I never create object of just `Shape`, do I have to implement destructor for `Shape`?
  - **NO YOU CAN'T**

> Have to implement a destructor even for an ABC

- Steps of Destruction
  1. Execute the body of the destructor
  2. Destroy the data members
     - If built-in, nothing
     - If class type, call that class's destructor
  3. Destroy the base part of the derived object
     - If a function is called, we have to implement it