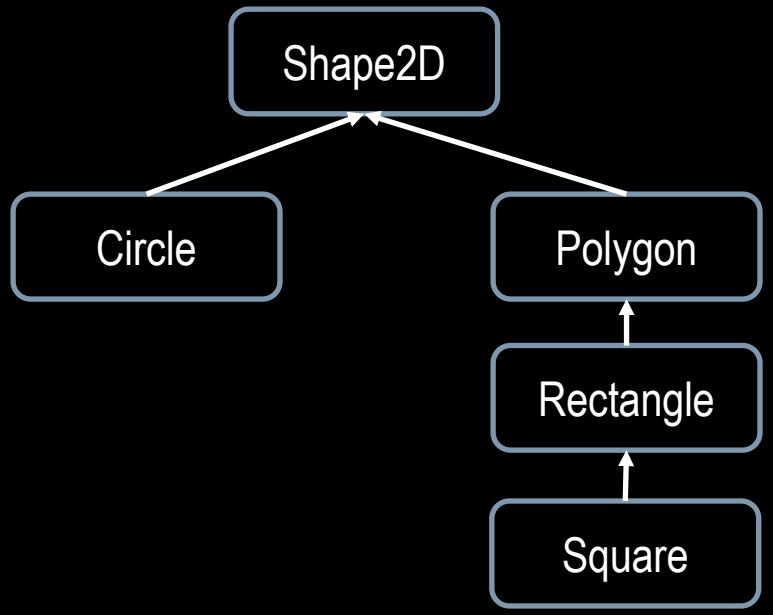
# Polymorphism

# Last time: Inheritance




Base class defined as usual

For this course, always put "public" (it's possible to use "private" here but you shouldn't for what we do)

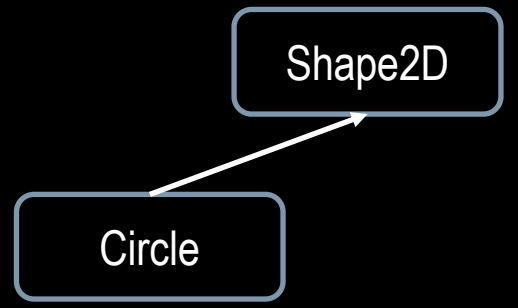
```
class Shape2D{
    ...
};

class Circle : public Shape2D{
    ...
};


class Polygon : public Shape2D{
    ...
};


class Rectangle : public Polygon{
    ...
};


class Square : public Rectangle{
    ...
};
```

The colon in the class definition means inheritance is happening. `Circle` is *derived from* `Shape2D`.

# Last Time: Example of Inheritance

```
class Shape2D{
    public:
    Shape2D(){}
    std::string getType() const {return type;}
    void setType(const std::string &type){this->type = type;}

    private:
    std::string type;
};

class Circle : public Shape2D{
    public:
    Circle(){
        setType("Circle");
    }
};

int main(){

    Shape2D shape;
    cout << "Shape type: " << shape.getType() << endl;

    Circle circle;
    cout << "Circle type: " << circle.getType() << endl;

    return 0;
}
```

We can use `setType` because it's defined as public in `Shape2D`




Blank because we never set the type string in shape

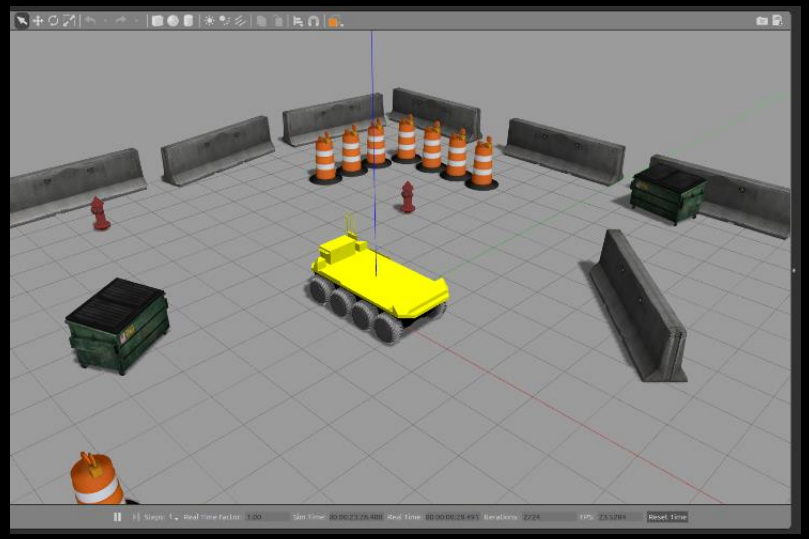
Output is:

```
Shape type:
Circle type: Circle
```

We can call `getType` because it's defined as public in `Shape2D`

# We want flexibility in how we use Derived classes

- Often we want to refer to objects by their Base class (even if they are Derived objects)
- E.g. let's say I want to define a scene in a simulator, consisting of many circle and rectangles (and potential other shapes, too)
- I would need a separate data structure to keep track of every type of shape:



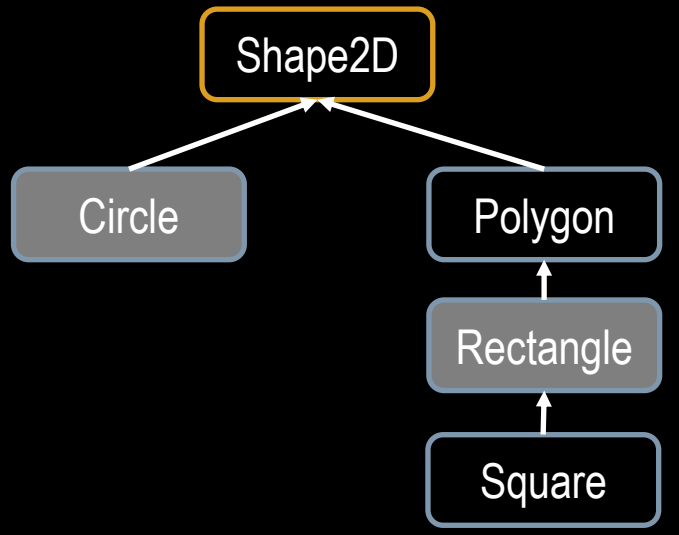
```
Circle circle1(Point(2,3),0.2);
Circle circle2(Point(4,2),0.5);
vector<Circle*> circles = {&circle1, &circle2};

std::vector<Point> verts1 = {Point(1,2), Point(4,2), Point(4,4), Point(1,4)};
Rectangle rect1(verts1);

std::vector<Point> verts2 = {Point(3,2), Point(6,2), Point(6,4), Point(3,4)};
Rectangle rect2(verts2);

vector<Rectangle*> rects = {&rect1, &rect2};
```

- What if someone came along and implemented a new shape type?
  - They would have to edit my code to include a data structure for that shape, too

# Using Pointers to Base classes

- Instead, we can refer to derived objects by what they have in common, i.e. their shared base class

```
Circle circle1(Point(2,3),0.2);
Circle circle2(Point(4,2),0.5);

std::vector<Point> verts1 = {Point(1,2), Point(4,2), Point(4,4), Point(1,4)};
Rectangle rect1(verts1);

std::vector<Point> verts2 = {Point(3,2), Point(6,2), Point(6,4), Point(3,4)};
Rectangle rect2(verts2);

vector<Shape2D*> shapes = {&circle1, &circle2, &rect1, &rect2};
```




- But there's a problem:

```
vector<Shape2D*> shapes = {&circle1, &circle2, &rect1, &rect2};
shapes[0]->
```

getType inline std::string Shape2D::getType() con…
printType
setType
type

C++ now doesn't know about all the derived stuff in `Circle`, it only knows about the `Shape2D` part of the object

# Aside: Pointers and Up/Down Casting

- Why does this work?

```
vector<Shape2D*> shapes = {&circle1, &circle2, &rect1, &rect2};
```

- Converting from a Derived class pointer to a Base class pointer, called upcasting, happens automatically
- The other way, converting from a Base pointer to a Derived pointer (called downcasting), won't happen automatically
  - Need to do this explicitly, but we won't in this course

```
Shape2D* pshape;
Circle* pcircle;
pshape = pcircle; //upcasting works!
pcircle = pshape; //trying to downcast, won't compile!
```

# Using references to Base classes: e.g. streams in C++

- Goal: Define a function that prints to either a file (ofstream) or cout (ostream), depending on what is passed in.

- What should the function signature be?

```
void printStuff(string& to_print, ostream& stream){
     stream << to_print;
}
```

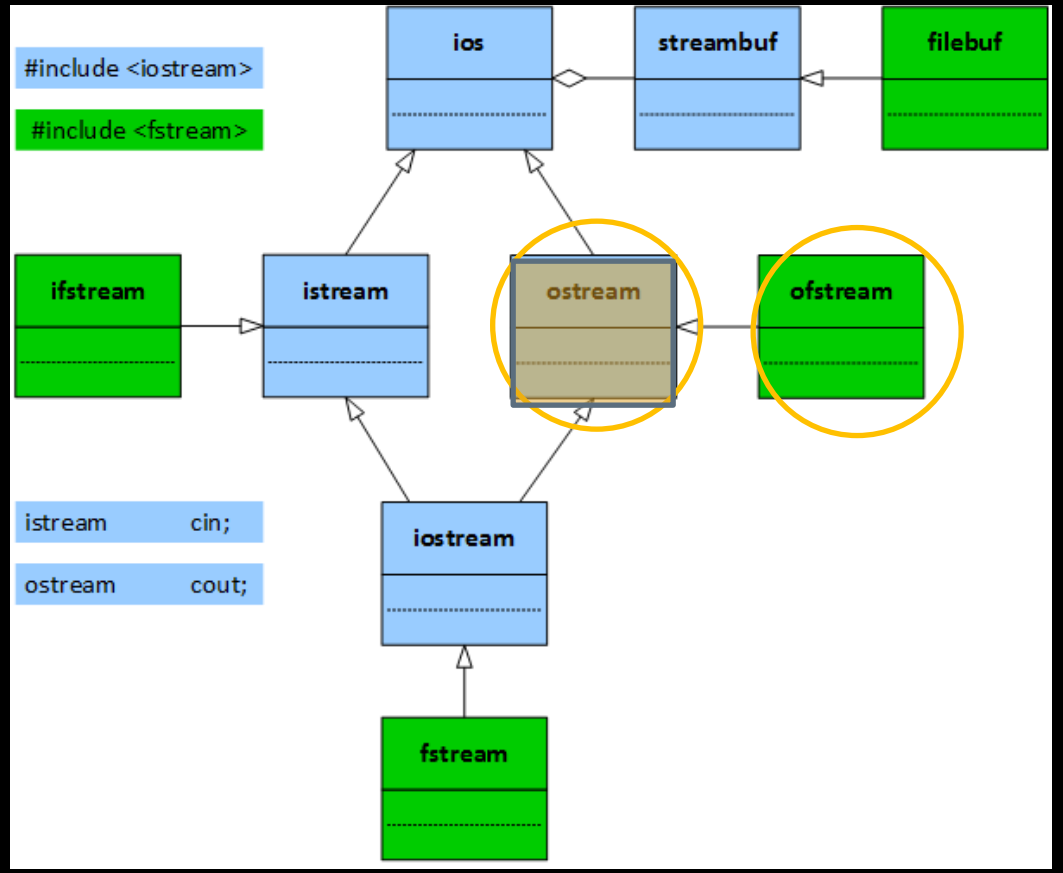



Figure Source

# Why not always use pointers/references to the Base class?

- Problem: This way we can only access the Base version of functions
- So *even if we overrode* printType(), C++ would use the Base version of printType() when calling shapes[0]->printType()

```
class Shape2D{
    public:
    Shape2D(){}
    ...
    void printType() const {std::cout << type << std::endl;}

    private:
    std::string type;
};

class Circle : public Shape2D{
    public:
    Circle(Point center, double radius){setType("Circle");...}
    void printType() const {std::cout << getType() << " R: "<<
                            radius << std::endl;}
    private:
    double radius;
    Point center;
};
```

```
int main(){
    Circle circle1(Point(2,3),0.2);
    Circle circle2(Point(4,2),0.5);
    vector<Shape2D*> shapes = {&circle1, &circle2};
    shapes[0]->printType();
    return 0;
}
```

Output is:
`Circle`

We want it to be
`Circle R: 0.2`

# Virtual functions

- Overriding functions that have already been defined (as before) is not very useful here
- Instead, C++ gives us a better way to override functions for derived classes:

```
class Shape2D{
    public:
    Shape2D(){}
    ...
    virtual void printType() const {std::cout << type << std::endl;}

    private:
    std::string type;
};

class Circle : public Shape2D{
    public:
    Circle(Point center, double radius){setType("Circle");...}
    virtual void printType() const {std::cout << getType() <<
" R: "<< radius << std::endl;}
    private:
    double radius;
    Point center;
};
```

```
int main(){
    Circle circle1(Point(2,3),0.2);
    Circle circle2(Point(4,2),0.5);
    vector<Shape2D*> shapes = {&circle1, &circle2};
    shapes[0]->printType();
    return 0;
}
```

Output is:

```
Circle R: 0.2
```

# Virtual functions

- A virtual function is a special type of function that, when called, resolves to the most-derived version of the function that exists between the base and derived class.
- This capability is known as polymorphism
- A derived function is considered a match if it
  1. has the same signature (name, parameter types, and whether it is const) as the base version
  2. has the same return type as the base version

- Using virtual functions is a little bit slower than a regular function since the program has too look through all derived functions to see if there is a match

- Tip: Never call virtual functions from constructors or destructors.
  - Why? It will call the base version of the function from the Base's constructor because the Derived version hasn't been created yet

# Pure Virtual Functions

- It's kind of silly to implement a function in a Base class that you know will get overridden
- More importantly, sometimes you will know that there should be a function but you will need info from derived classes to implement it
  - E.g. we may want a computeArea() function for every shape, but there's no way to compute the area of a general 2D shape
  - We need to know what kind of shape we have to compute the area (i.e. this function should be implemented in a Derived class)
- We also want to call that function with a pointer or reference to the Base class
- To make a pure virtual function, replace the implementation with "=0"

```
class Shape2D{
    public:
    Shape2D(){}
    ...
    virtual void printType() const {std::cout << type << std::endl;}

    private:
    std::string type;
};
```

Pretty useless

```
class Shape2D{
    public:
    Shape2D(){}
    ...
    virtual void printType() const = 0;

    private:
    std::string type;
};
```

A "pure" virtual function

# Abstract Classes

- An abstract class is a class that contains a pure virtual function
- An abstract class **cannot be instantiated**!

```
class Shape2D{
    public:
    Shape2D(){}
    ...
    virtual void printType() const = 0;

    private:
    std::string type;
};

int main(){
    Shape2D s; //won't compile!
    return 0;
}
```

- Why?
  - If you tried to call s.printType(), there would be no function to call!
  - So the object is not considered valid

# Abstract Classes

- However, pointers to abstract classes are no problem!

```
class Shape2D{
    public:
    Shape2D(){}
    ...
    virtual void printType() const = 0;

    private:
    std::string type;
};

int main(){
    Shape2D s; //won't compile!
    Shape2D* ps; //no problem!
    return 0;
}
```

- Why?
  - Creating a pointer doesn't actually create the object, it's just a memory address.

# Using Pointers to Abstract Classes

- Same usage as before!
- But now we don't need to provide a useless implementation in the Base class
- And we **force all derived classes** to implement printType() or else they can't be instantiated

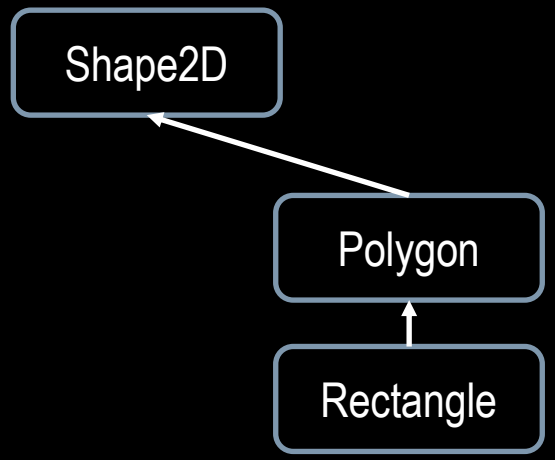
```
class Shape2D{
    public:
    Shape2D(){}
    ...
    virtual void printType() const = 0;
    private:
    std::string type;
};

class Circle : public Shape2D{
    public:
    Circle(Point center, double radius){setType("Circle");...}
    virtual void printType() const {std::cout << getType() <<
" R: "<< radius << std::endl;}
    private:
    double radius;
    Point center;
};
```

```
int main(){
    Circle circle1(Point(2,3),0.2);
    Circle circle2(Point(4,2),0.5);
    vector<Shape2D*> shapes = {&circle1, &circle2};
    shapes[0]->printType();
    return 0;
}
```

Output is:

```
Circle R: 0.2
```

# Another example of abstract classes:




- `Shape2D` is an abstract class because it has `virtual void printType() const = 0;`

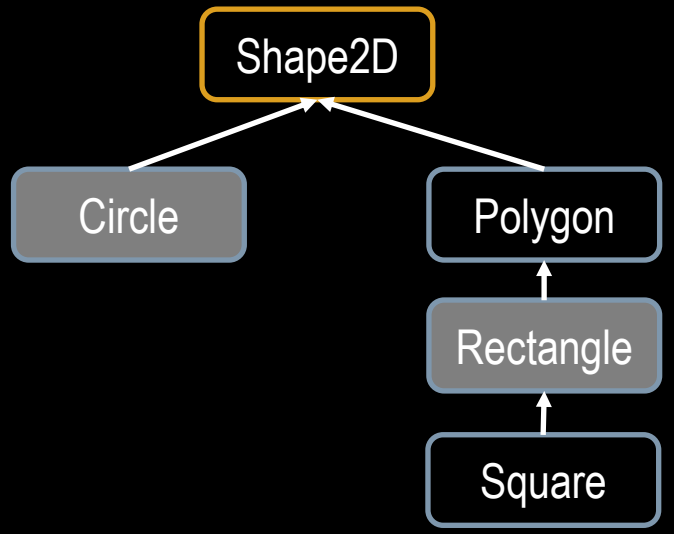
- `Polygon` is **also an abstract class** because it does not implement `printType()`
- Thus `Polygon` cannot be instantiated!

- `Rectangle` is NOT an abstract class because it implements `printType()`

```
class Polygon : public Shape2D{
    public:
    Polygon(const std::vector<point> &vertices){
        this->vertices = vertices;
        setType("Polygon"); //setType defined in Shape2D

    }
    std::vector<Point> getVertices(){return vertices;}

    private:
    std::vector<Point> vertices;
};

class Rectangle : public Polygon{
    public:
    Rectangle(const std::vector<Point> &vertices) : Polygon(vertices){
        setType("Rectangle"); //setType defined in Shape2D
    }
    virtual void printType() const {std::cout << getType() << std::endl;}
};
```

# Putting the two examples together

```
Circle circle1(Point(2,3),0.2);
Circle circle2(Point(4,2),0.5);

std::vector<Point> verts1 = {Point(1,2), Point(4,2), Point(4,4), Point(1,4)};
Rectangle rect1(verts1);

std::vector<Point> verts2 = {Point(3,2), Point(6,2), Point(6,4), Point(3,4)};
Rectangle rect2(verts2);

vector<Shape2D*> shapes = {&circle1, &circle2, &rect1, &rect2};
shapes[0]->printType();
shapes[2]->printType();
```




Output is:

```
Circle R: 0.2
Rectangle
```

# Polymorphism Summary

- We now have a way to access functions in Derived classes while storing pointers to different derived classes in the same data structure

```
    vector<Shape2D*> shapes = {&circle1, &circle2, &rect1, &rect2};
    shapes[0]->printType();
    shapes[2]->printType();
```

Output is:
```
Circle R: 0.2
Rectangle
```

- To do this, we defined a virtual function in the common Shape2D Base class
- Then we made this a pure virtual function by using “=0” instead of the implementation

```
class Shape2D{
    public:
    Shape2D(){}
    ...
    virtual void printType() const = 0;
    private:
    std::string type;
};
```

- This made Shape2D an abstract class and forced Derived classes to implement the function (or to also be abstract, like Polygon)

# Homework

- Homework 3
- Read LaValle Chapter 3.2