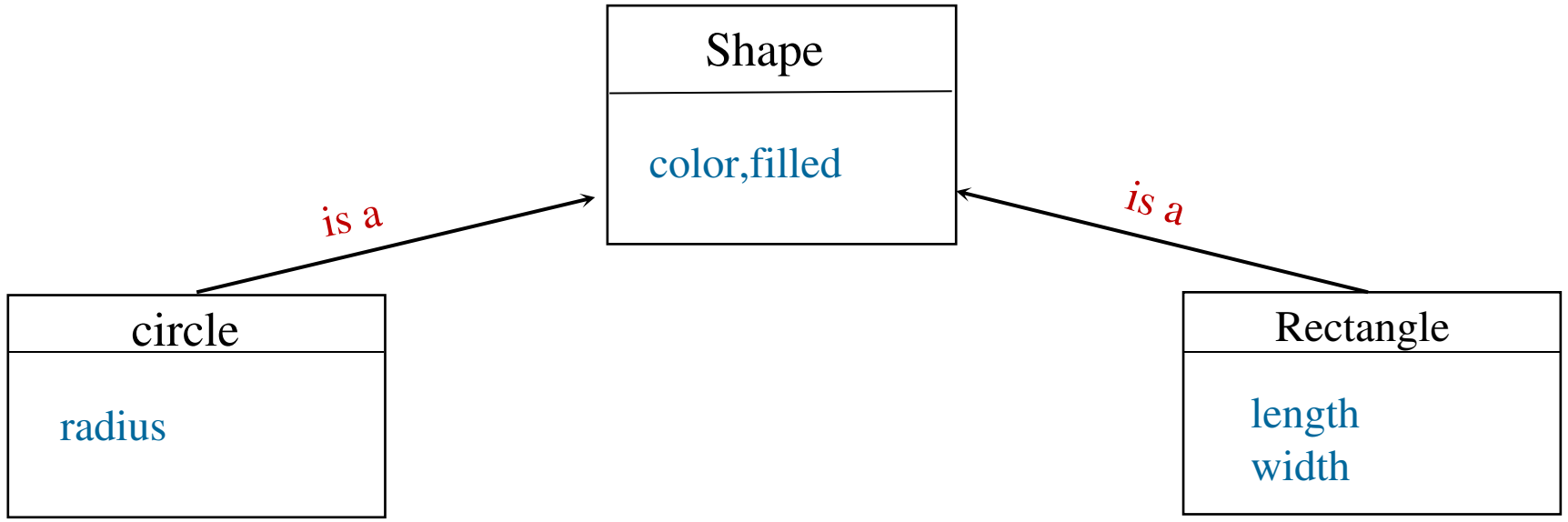
Inheritance and Polymorphism

The four pillars of OOP

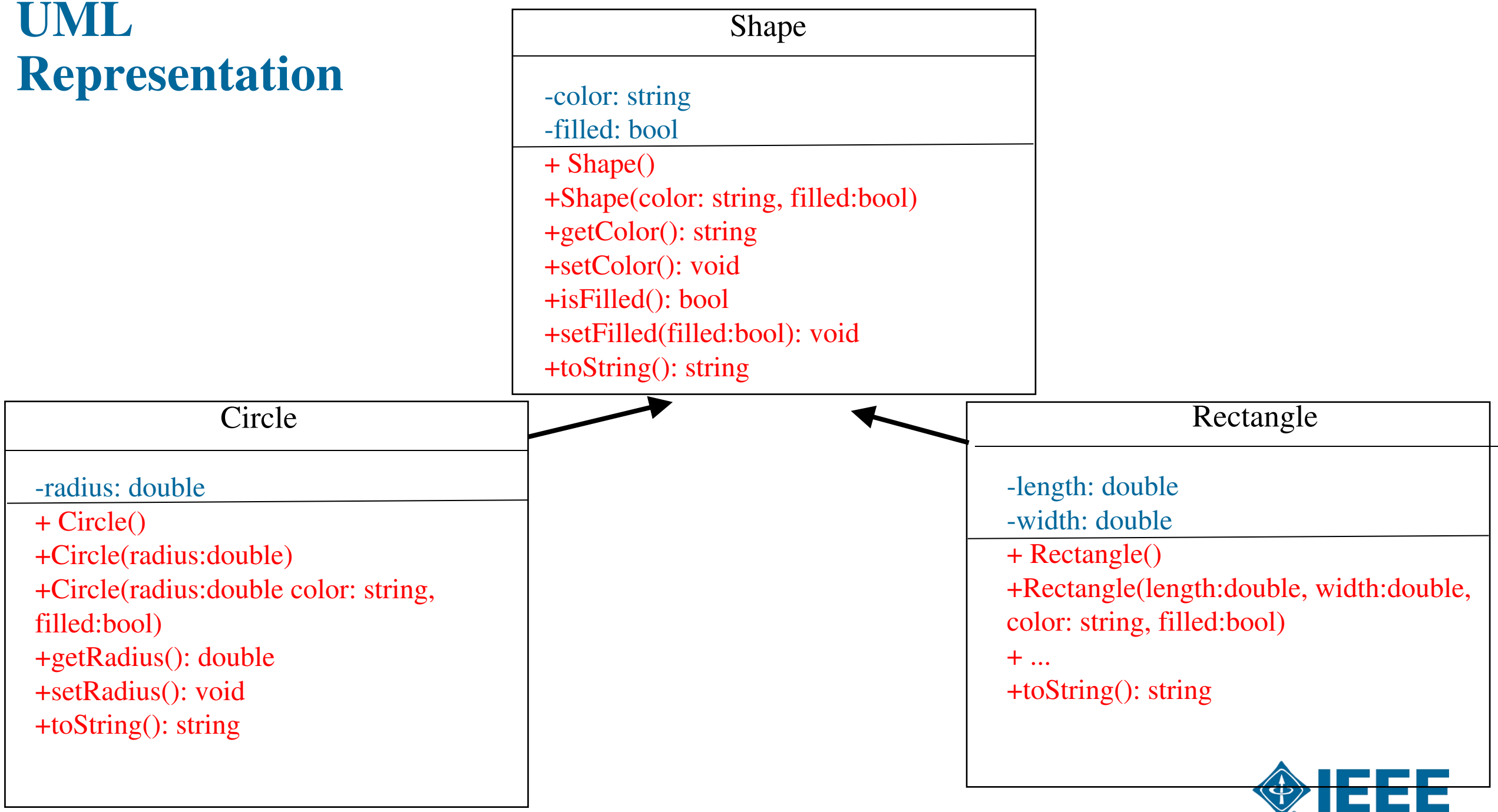
- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

What is inheritance ?

- It's a way of creating new class based on an existing class
- The new class (child class) derives the properties & characteristics from the existing class (parent class)



UML Representation



Shape.h

```
class Shape{
   private:
   string color;
   bool filled;

   public:
   Shape();
   Shape(const string& c, bool fill );
   string getColor() ;
   void setColor(const string& c);
   bool isFilled() ;
   void setFilled(bool fill);
   string toString() ;

};
```

Shape.cpp

```
Shape::Shape(){
      color="Red";
      filled=false;
   }
   Shape::Shape(const string& c, bool fill ){
      color=c;
      filled=fill;
   }
   string Shape::getColor() {return color;}
   void Shape::setColor(const string& c) {color=c;}
   bool Shape::isFilled() {return filled;}
   void Shape::setFilled(bool fill) {filled=fill;}
   string Shape::toString() {return "Shape";}
```

Circle.h

```
class Circle: public Shape{
  private:
  double radius;
  public:
  Circle();
  Circle(double r);
  Circle(double r, const string&c, bool fill);
  double getRadius() ;
  void setRadius(double r);
   string toString();
```

Circle.cpp

```
  Circle::Circle(){ radius=1;}
  Cirlce::Circle(double r){radius=r;}
  Circle::Circle(double r, const string&c, bool fill){
      radius=r;
      setColor(c);  //Can't say color=c, as color is
private in parent class
      setFilled(fill);// Can't say filled =fill for the same
reason
  }
  double Circle::getRadius() { return radius;}
  void Circle::setRadius(double r){radius=r;}
  string Circle::toString(){return "Circle";}
```

Shape.h

```
class Shape{
  private:
  string color;
  bool filled;

  public:
  Shape();
  Shape(const string& c, bool fill );
  string getColor() ;
  void setColor(const string& c);
  bool isFilled() ;
  void setFilled(bool fill);
  string toString() ;

};
```

Circle.h

```
class Circle: public Shape{
  private:
  double radius;
  public:
  Circle();
  Circle(double r);
  Circle(double r, const string&c, bool fill);
  double getRadius() ;
  void setRadius(double r);
  string toString();
```

Q1)How does inheritance deal with these functions?

Q2)What does this word mean?

Constructors

Constructors of a Parent class aren't inherited in the Child class, but they are invoked (explicitly or implicitly) from the constructor of the Child class.

1) implicit calling (no argument Parent class Constructor)

```
Circle::Circle(){
radius=1;
}
```

equivalent to

```
Circle::Circle() : Shape()
{
radius=1;
}
```

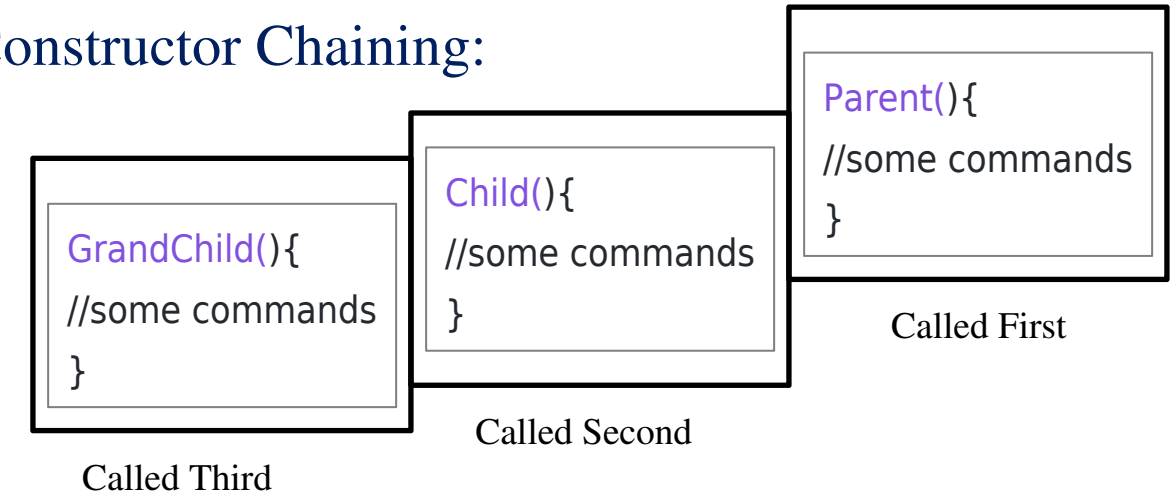
```
Circle::Circle(double r){
radius=r;
}
```

equivalent to

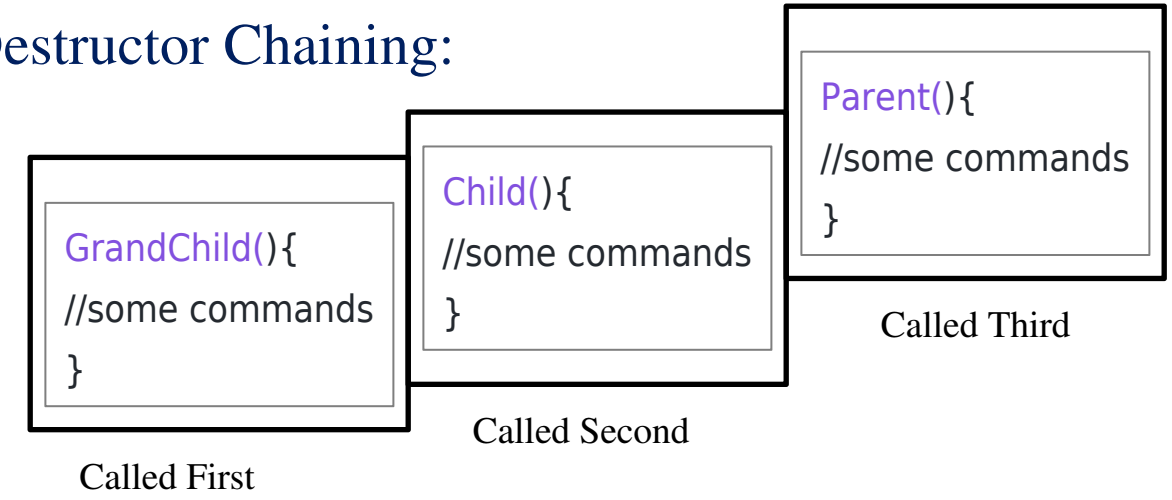
```
Circle::Circle(double r) : Shape()
{
radius=r;
}
```

Constructors

Constructor Chaining:



Destructor Chaining:



```
class Parent{
   public:
   Parent(){cout<<"Parent class called"<<endl;}
   ~Parent(){cout<<"Parent class destructed"<<endl;}
};

class child : public Parent
{
   Public:
   Child(){ cout<<"Child class called"<<endl;}
   ~child(){cout<<"Child class destructed"<<endl;}
}
```

main

```
int main(){
   child y;
   return 0;
}
```

RUN

Circle.cpp

```
Circle::Circle(){ radius=1;}
Cirlce::Circle(double r){radius=r;}
Circle::Circle(double r, const string&c, bool fill){
    radius=r;
    setColor(c);  //Can't say color=c, as color is private in parent class
    setFilled(fill);// Can't say filled =fill for the same reason
}
```

Circle.cpp

```
Circle::Circle(){ radius=1;}
Cirlce::Circle(double r){radius=r;}
Circle::Circle(double r, const string&c, bool fill) : Shape(){
    radius=r;
    setColor(c);  //Can't say color=c, as color is private in parent class
    setFilled(fill);// Can't say filled =fill for the same reason
}
```

Circle.cpp

```
Circle::Circle(){ radius=1;}
Cirlce::Circle(double r){radius=r;}
Circle::Circle(double r, const string&c, bool fill) : Shape( c , fill){
   radius=r;
}
```

And that's explicit calling

Shape.h

```
class Shape{
   private:
   string color;
   bool filled;

   public:
   Shape();
   Shape(const string& c, bool fill );
   string getColor() ;
   void setColor(const string& c);
   bool isFilled() ;
   void setFilled(bool fill);
   string toString();

};
```

Circle.h

```
class Circle: public Shape{
   private:
   double radius;
   public:
   Circle();
   Circle(double r);
   Circle(double r, const string&c, bool fill);
   double getRadius() ;
   void setRadius(double r);
   string toString();
```

Q1)How does inheritance deal with these functions?

Redefining Parent Class Function

-Redefined Function: it's a function in the child class that has the same name and parameters as the function in Parent class

-It differs from Overloading [in overloading, parameters must be different]

-The parent class uses the parent class version, while the child class uses the child class version

Redefining Parent Class Function

```
int main(){
  Shape s;
  Circle c;
   cout<<s.toString()<<endl;
   cout<<c.toString()<<endl;
   return 0;
}
```

RUN

Redefining Parent Class Function

What if you want to call the function defind in the parent class from the child class?

```
int main(){
  Circle c;
   cout<<c.Shape::toString()<<endl;
   return 0;
}
```

RUN

Generic Programming

Generic Programming: An object of a derived class can be used whenever an object of its base class is required

```
void displayColor(const Shape& x){
  cout<< x.getColor();
}
```

```
int main(){
Circle c1;
displayColor(c1);
return 0;
}
```

RUN

Generic Programming

```
void ShapeName(Shape& x){
   cout<< x.toString()<<endl;
}
```

```
int main(){
   Shape s;
   ShapeName(s);
   Circle c;
   ShapeName(c);
   return 0;
}
```

RUN

Shape.h

```
class Shape{
   private:
   string color;
   bool filled;

   public:
   Shape();
   Shape(const string& c, bool fill );
   string getColor() ;
   void setColor(const string& c);
   bool isFilled() ;
   void setFilled(bool fill);
   string toString() ;

};
```

Circle.h

```
class Circle: public Shape{
   private:
   double radius;
   public:
   Circle();
   Circle(double r);
   Circle(double r, const string&c, bool fill);
   double getRadius() ;
   void setRadius(double r);
   string toString();
```

Q2)What does this word mean?

Access modifiers

-Data Hiding.

-For a Parent class to control inheritance

```
private:
  string color;
  bool filled;
```

Types of Access modifiers:

public:

private:

-Not accessible by object of child classes

protected:

-Like private, but accessible by objects of child classes

Access Specifiers

-For a Child class to control inheritance

```
class Circle: public Shape
```

public: Object of child class can be treated as an object of parent class(not vice versa) [public members of parent class are public in child]

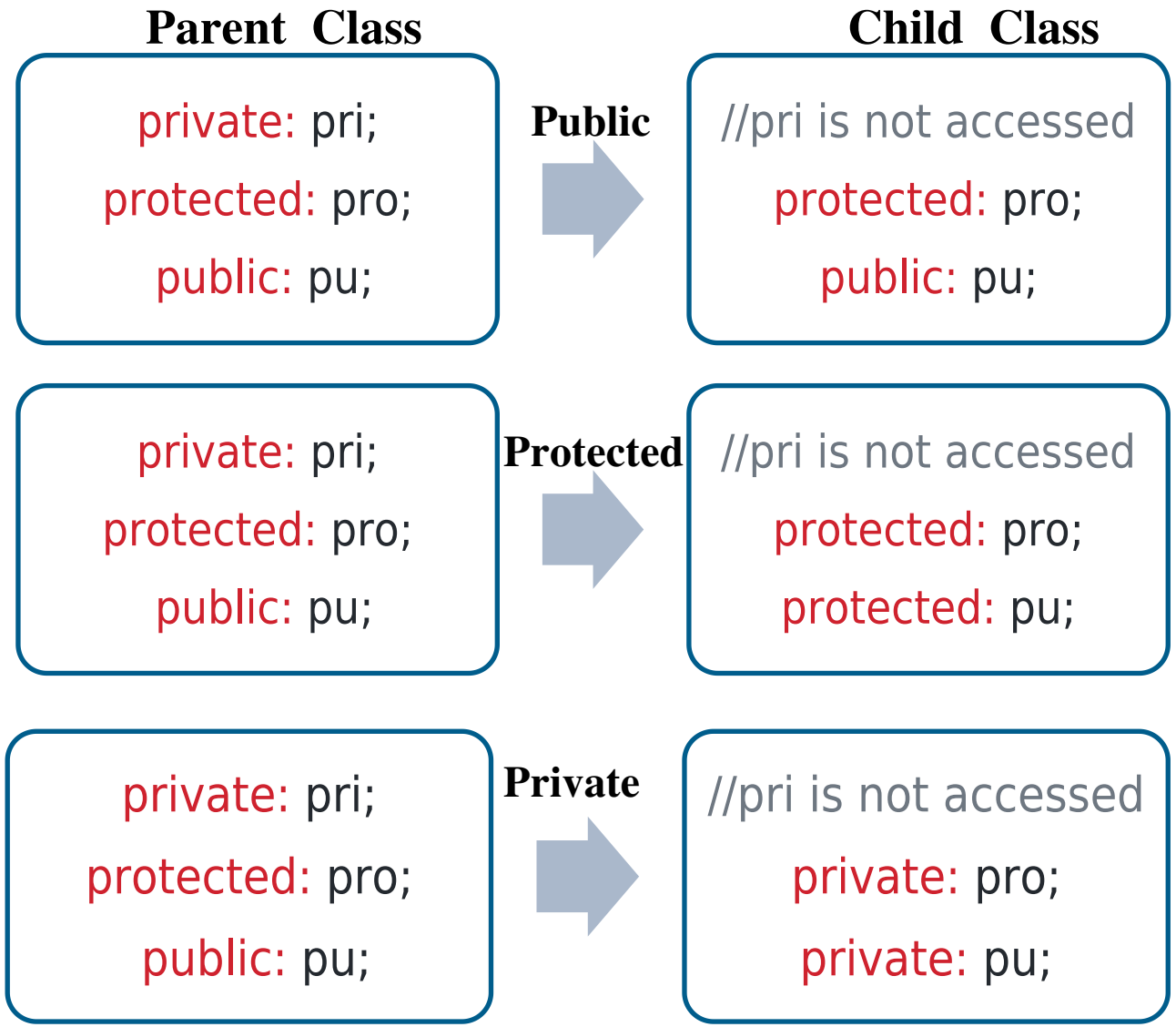
private : [Default]

-Object of child class can't be treated as an object of parent class[public members of parent class are private in child (so they can't be inherited to grand children)

protected:

-Like private, but[public members of parent class are protected in child (so they can be inherited to grand children)

Access Specifiers



Access Specifiers

-Using protected or private as an access specifire makes the inheritance **not an is-a Relationship**

Part 2: Polymorphism

Static binding

Earlier we saw this example, and the output wasn't as we expected, so why is that?

```
void ShapeName(Shape& x){
   cout<< x.toString()<<endl;
}
```

Functions calls are bound at compile time.
This is called staic binding

```
int main(){
   Shape s;
   ShapeName(s);
   Circle c;
   ShapeName(c);
   return 0;
}
```

RUN

```
string Shape::toString() const {
return "Shape";}
```

```
string Circle::toString()const{
return "Circle";}
```

How to overcome this problem?

- That's what polymorphism is for.

- Polymorphism: The ability of a method to behave differently in different scenarios by using the parent class object to refer to a child class object

Virtual functions and dynamic binding

Functions that are redefined in the child class should be virtual in the parent class

-Defined in parent class using the keyword virtual:

-A virtual function is dynamically bond to calls at runtime.

```
class Shape{
.....
virtual string toString()
{return "Shape";}
};
```

-At runtime: C++ determines the type of object making the call and bind the function to the apropriate version of the function

Shape.h

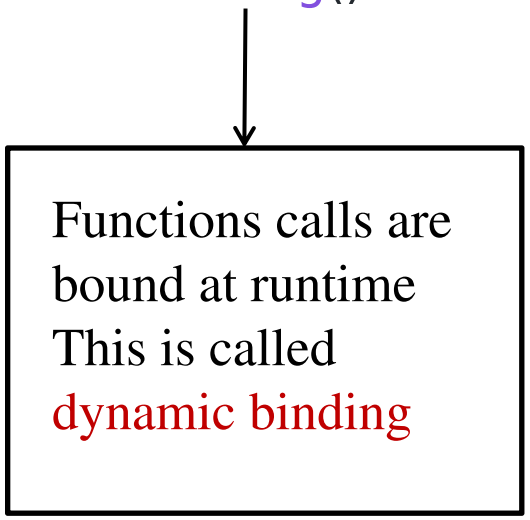
```
class Shape{
   private:
   string color;
   bool filled;

   public:
   Shape();
   Shape(const string& c, bool fill );
   string getColor() ;
   void setColor(const string& c);
   bool isFilled();
   void setFilled(bool fill);
   virtual string toString() ;

};
```

Shape.cpp

```
Shape::Shape(){
      color="Red";
      filled=false;
   }
   Shape::Shape(const string& c, bool fill ){
      color=c;
      filled=fill;
   }
   string Shape::getColor() {return color;}
   void Shape::setColor(const string& c) {color=c;}
   bool Shape::isFilled() {return filled;}
   void Shape::setFilled(bool fill) {filled=fill;}
   string Shape::toString() {return "Shape";}
```

virtual functions and dynamic binding

```
void ShapeName(Shape& x){
   cout<< x.toString()<<endl;
}
```




```
int main(){
   Shape s;
   ShapeName(s);
   Circle c;
   ShapeName(c);
   return 0;
}
```

RUN

```
class Shape{
.....
virtual string toString()
{return "Shape";}
};

string Circle::toString()const{
return "Circle";}   //Overridding
```

Virtual functions and dynamic binding

For the polymorphismic behaviour to occur, then the object must be referenced by a pointer or a refrence

```
void ShapeName(Shape& x)

void ShapeName(Shape* x)
```

virtual functions and dynamic binding

```
void ShapeName(Shape* x){
   cout<< x -> toString()<<endl;
}
```

```
int main(){
   Shape s;
   ShapeName( &s);
   Circle c;
   ShapeName( &c);
   return 0;
}
```

RUN

```
class Shape{
.....
virtual string toString()
{return "Shape";}
};

string Circle::toString()const{
return "Circle";}
```

virtual functions and dynamic binding

```
void ShapeName(Shape* x){
   cout<< x -> toString()<<endl;
}
```

```
int main(){
   Shape* ps=new Shape;
   ShapeName( ps);
   delete ps;
   Circle* pc=new Circle;
   ShapeName( pc);
   delete pc;
   return 0;
}
```

RUN

```
class Shape{
.....
virtual string toString()
{return "Shape";}
};

string Circle::toString()const{
return "Circle";}
```

When to use virtual functions?

It’s not a good idea to make all parent class functions virtual.

-Why?

Virtual functions take more time and consume more resources in order to be dynamically bound at run time

Pointers to parent class

Pointers to a parent class can be assigned child classes objects

```
Shape* ps=new Circle;
```

However it can only access methods of the parent class.

```
ps->setColor("Blue");
ps->setRadius(5);
```

Pointers to parent class

Why is that useful?

```
int main(){
   Shape* pShapeArr[3];
    pShapeArr[0]=new Circle;
    pShapeArr[1]=new Rectangle;
    pShapeArr[2]=new Triangle;

//do some operations on the Array
   return 0
}
```

Pointers to parent class

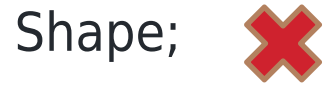
Why is that useful?

```
int main(){
   Shape* pShapeArr[]={new Circle,
    new Rectangle,
   new Triangle};

//do some operations on the Array
   return 0
}
```

Pointers to parent class

Note: Pointers to child classes can't be assigned to parent class objects.

```
Circle* ps=new Shape;
```



Why?
=Because Circles has some functions and properties that shape doesn't have
[All circles are shapes but not all shapes are circles]

Virtual Destructors

It's recommended to make destructors virtual.

```
class Parent{
  public:
  Parent(){cout<<"Parent class called"<<endl;}
  ~Parent(){cout<<"Parent class destructed"<<endl;}
};

class child :: public Parent
{
  Child(){ cout<<"Child class called"<<endl;}
  ~child(){cout<<"Child class destructed"<<endl;}
}
```

```
int main(){
  Parent* ptr=new child;
  delete ptr;
  return 0
}
```

RUN

Virtual Destructors

It's recommended to make destructors virtual.

```
class Parent{
   public:
   Parent(){cout<<"Parent class called"<<endl;}
   virtual~Parent(){cout<<"Parent class destructed"<<endl;}
};

class child :: public Parent
{
   Child(){ cout<<"Child class called"<<endl;}
   ~child(){cout<<"Child class destructed"<<endl;}
}
```

```
int main(){
   Parent* ptr=new child;
   delete ptr;
   return 0
}
```

RUN

Abstract Class and pure virtual function

Abstract Class:

-It has no objects

-acts as the basis of child classes that will/may have objects

Abstract Class and pure virtual function

Creating Abstract Class: One or more of its member functions must be a pure virtual function

Abstract Class and pure virtual function

Pure virtual function: It's a virtual function that must be overridden in the child class that has objects

virtual void fun()=0;

the "=0" indicates a pure virtual function

It must have no function definition in the parent class

Shape.h

```
class Shape{
   private:
   string color;
   bool filled;

   public:
   Shape();
   Shape(const string& c, bool fill );
   string getColor() ;
   void setColor(const string& c);
   bool isFilled() ;
   void setFilled(bool fill);
   virtual string toString() ;
   virtual double getArea()=0;
};
```

Shape.cpp

```
Shape::Shape(){
      color="Red";
      filled=false;
   }
   Shape::Shape(const string& c, bool fill ){
      color=c;
      filled=fill;
   }
   string Shape::getColor() {return color;}
   void Shape::setColor(const string& c) {color=c;}
   bool Shape::isFilled() {return filled;}
   void Shape::setFilled(bool fill) {filled=fill;}
   string Shape::toString() {return "Shape";}
```

Circle.h

```
class Circle: public Shape{
  private:
  double radius;
  public:
  Circle();
  Circle(double r);
  Circle(double r, const string&c, bool fill);
  double getRadius() ;
  void setRadius(double r);
  double string toString();
  double getArea();
```

Circle.cpp

```
  Circle::Circle(){ radius=1;}
  Cirlce::Circle(double r){radius=r;}
  Circle::Circle(double r, const string&c, bool fill){
     radius=r;
     setColor(c);  //Can't say color=c, as color is private
in parent class
     setFilled(fill);// Can't say filled =fill for the same
reason
  }
  double Circle::getRadius() { return radius;}
  void Circle::setRadius(double r){radius=r;}
  string Circle::toString(){return "Circle";}
  double Circle::getArea(){return 3.14 * radius* radius;}
```

Abstract class and pure virtual function

```
void LargerArea( const Shape& x, const Shape& y){
   cout<< max(x.getArea(),y.getArea())<<endl;
}
```

```
int main(){
   Circle c(7);
   Rectangle r(5,4);
   LargerArea(c,r);
   return 0;
}
```

RUN

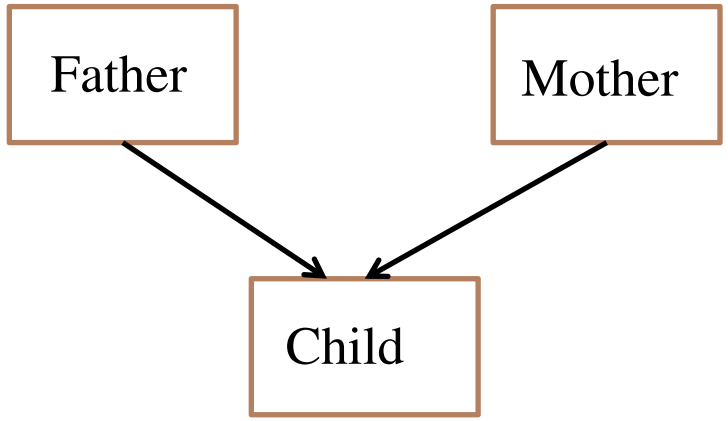
Multiple inheritance

Multiple inheritance.

-C++ allows a child class to have more than one parent class.

-Each of them has its own access specifier in child class defination

```
class Child: public Father, public Mother
```



Multiple inheritance.

Parent class constructors are always called in order given in child class declaration

```
Child::Child(){
//some commands
}
```

equivalent to

```
Child::Child() : Father(), Mother()
{
//some commands
}
```

```
Child::Child() : Mother(),Father()
{
//some commands
}
```

equivalent to

```
Child::Child() : Father(), Mother
{
//some commands
}
```

```
class Child: public Father,
public Mother
```

```
class Father{
   public:
   Father(){cout<<"Father class
called"<<endl;}
   ~Father(){cout<<"Father class
destructed"<<endl;}
};
```

```
class child : public Father,public Mother
{public:
   Child(){ cout<<"Child class called"<<endl;}
   ~child(){cout<<"Child class destructed"<<endl;}
}
```

```
class Mother{
   public:
   Mother(){cout<<"Mother class
called"<<endl;}
   ~Mother(){cout<<"Mother class
destructed"<<endl;}
};
```

```
int main(){
   child y;
   return 0;
}
```

RUN

```
class Father{
   public:
   Father(){cout<<"Father class
called"<<endl;}
   ~Father(){cout<<"Father class
destructed"<<endl;}
};
```

```
class child : public Father,public Mother
{public:
   Child():Mother(),Father()
{ cout<<"Child class called"<<endl;}
   ~child(){cout<<"Child class
destructed"<<endl;}
}
```

```
class Mother{
   public:
   Mother(){cout<<"Mother class
called"<<endl;}
   ~Mother(){cout<<"Mother class
destructed"<<endl;}
};
```

```
int main(){
   child y;
   return 0;
}
```

RUN

```
class student{
public: string Name="Mohamed";
string Major="Engineering";
};
```

```
class ArtistStudent:public student, public
artist { };
```

```
class artist{
public: string Name="Ahmed";
int numberOfPaints = 100;
};
```

```
int main(){
ArtistStudent obj;
cout<<obj.Major<<endl;
cout<<obj.numberOfPaints<<endl;
cout<<obj.Name<<endl;
return 0;
}
```

RUN

```
class student{
public: string Name="Mohamed";
string Major="Engineering";
};
```

```
class ArtistStudent:public student, public
artist { };
```

```
class artist{
public: string Name="Ahmed";
int numberOfPaints = 100;
};
```

```
int main(){
ArtistStudent obj;
cout<<obj.Major<<endl;
cout<<obj.numberOfPaints<<endl;
cout<<student::obj.Name<<endl;
return 0;
}
```

RUN

```
class student{
public: string Name="Mohamed";
string Major="Engineering";
};
```

```
class ArtistStudent:public student, public
artist {
  public: string Name="Omar";
 };
```

```
class artist{
public: string Name="Ahmed";
int numberOfPaints = 100;
};
```

```
int main(){
ArtistStudent obj;
cout<<obj.Major<<endl;
cout<<obj.numberOfPaints<<endl;
cout<<obj.Name<<endl;
return 0;
}
```

RUN

```
class student{
public: string Name="Mohamed";
string Major="Engineering";
};
```

```
class ArtistStudent:public student, public
artist {
  public: string Name="Omar";
 };
```

```
class artist{
public: string Name="Ahmed";
int numberOfPaints = 100;
};
```

```
int main(){
student* obj= new ArtistStudent;
cout<<obj->Major<<endl;
cout<<obj->numberOfPaints<<endl;
cout<<obj->Name<<endl;
delete obj;
return 0;
}
```

RUN

Thanks

presented by Omar Khaled