OOP in C++

Object Oriented Programming

- Classes and Objects
- Four major features:
  - Encapsulation
  - Inheritance
  - Polymorphism
  - Abstraction

Encapsulation

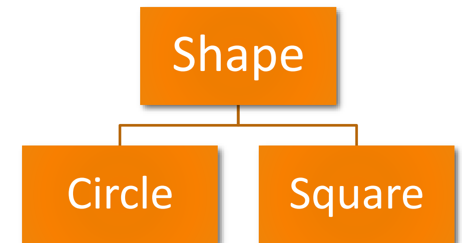
- Binding data and methods together and restricting access.
- Controlling access to the data of an object
  - Sometimes called “Data Hiding”
- Access Specifiers
  - private -> only inside class.
  - public -> accessible everywhere
  - protected- >accessible in derived classes
- Other languages
  - Handled with getters & setters in Java
  - Handled with Properties in C#
- C++ uses getters and setters as well

Encapsulation Example

```
class Car
{
public:
       Car( );

       // Getters
       int   getMiles( );
       char* getColor( );

       // Setters - Can't directly set miles
       void setColor(char* newColor);

private:
       int   miles;
       char* color;
};
```

Inheritance

- Reuse properties of another class.
- Basing a class on another class
  - Parent – Child relationship
  - Promotes code reuse
- C# and Java support single inheritance
  - Can only inherit from a single class at most
- C++ supports multiple inheritance
  - Can inherit from any number of classes
  - Sometimes problematic




```
class Parent {
public:
     void show(){
     cout << "Parent" ;
}
};
class Child: public Parent {
};
```

Types of Inheritance

- Single Inheritance
  - One parent ->one child
- Multiple Inheritance

```
class A{};
class B{};
class C: public A, public B{};
```

- Multilevel Inheritance

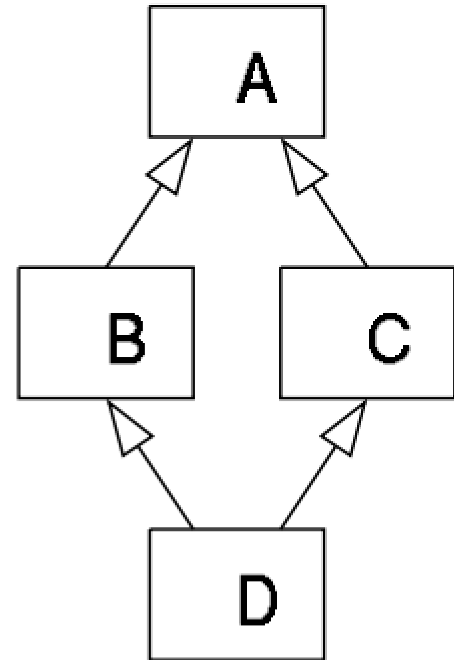
  A->B ->C

- Hierarchical Inheritance
  - One parent -> multiple children

Inheritance Type	Public members	Protected members
public	public	protected
protected	protected	protected
private	private	private

Multiple Inheritance Issues

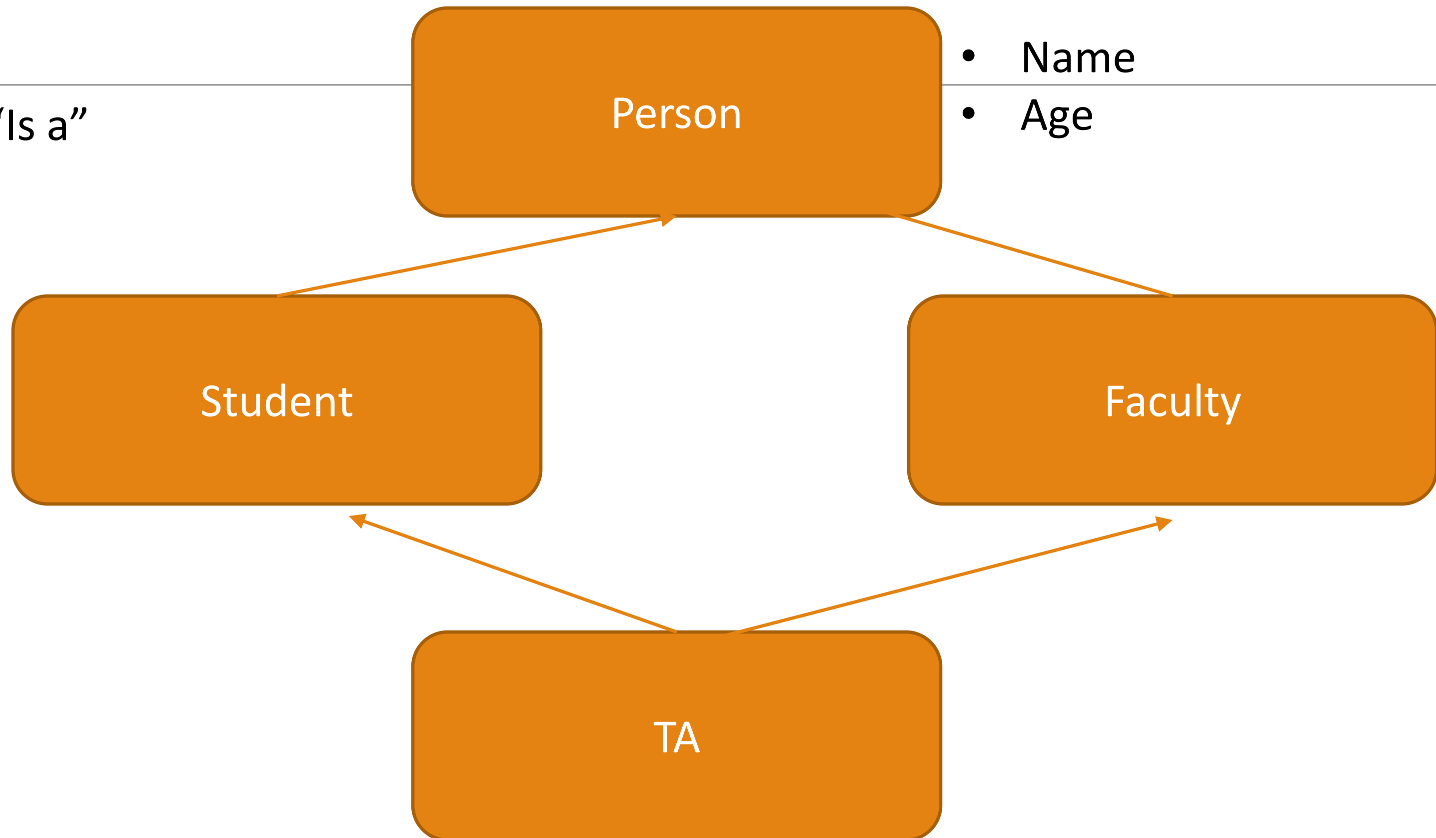
- The “Diamond Problem”
- A is the base (parent) class
  - B and C inherit from A
  - D inherits both B and C
- What if B and C each override a method from A?
  - Which version does D get?




```
class A{
  public :
    int x;
    };
    Class B: public A{};
    Class C : public A{};
    Class D: public B, public C{};// problem D gets two copies of A
```

Example

Use the “Is a” syntax




```
class TA : public Faculty, public Student
 { …
};
```

Access Levels

Same basic access levels as other languages

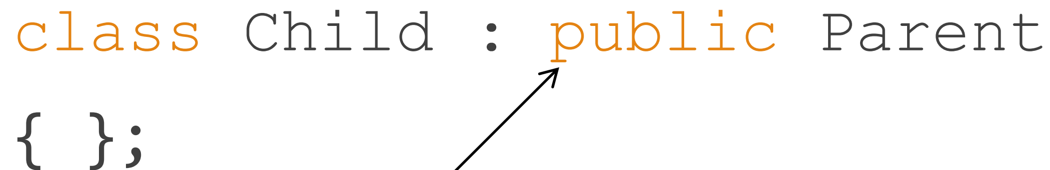
Public – All code can access

Private – Only accessible to code in this class

Protected – Accessible to code in this class
and any child classes

Inheritance Syntax

Syntax should look mostly familiar



Notice the access specifier

- Specifies the access of the inheritance
- Which parts “outside code” has access to

Public vs. Private Inheritance

The access specifier used during inheritance affects how base class members are treated in the derived class.

```
class Child : public Parent { };
```

From outside the Child class, the Parent's public inherited members can be accessed

This is how inheritance works in C# & Java

Public vs. Private Inheritance

```
class Child : private Parent { };
class Child : Parent { };
// private by default!
```

From outside the Child class, the Parent’s public inherited members are inaccessible!

- Child can still access them, but no one else can

As if they were actually private members to begin with

Inheritance Example

```
class base

{
        public:
                int x;
        protected:
                int y;
        private:
                int z;
};
```

```
class publicDerived: public base
{
        // x is public
        // y is protected
        // z is not accessible from publicDerived
};

class protectedDerived: protected base
{
        // x is protected
        // y is protected
        // z is not accessible from protectedDerived
};

class privateDerived: private base
{
        // x is private
        // y is private
        // z is not accessible from privateDerived
}
```

Accessing Base Class Members

Access base class members by preceding the member name with the class name

```
void Child::printChild( )
{
      Parent::printParent( );
      cout << "Also a child!" << endl;
}
```

There's no "base" keyword

Accessing Base Class Members-in Derived Class

```
#include <iostream>

using namespace std;

class Base{

public :
  int x;
  void display(){
  cout << "Base display :x = "<<x<<
  endl;
  }

protected:
   int y;

private:
   int z;
  };
```

```
class Derived: public Base {

public:

   void setValues(){
     x=10;y=20;
     //z= 30;// ERROR (private in base)
   }
    void show(){
     cout<<"Derived show :x = "<<x<<",y = "<<y <<endl;
     }
   };

   int main()
   {
     Derived obj;
     obj.x=5;
     obj.display();
     obj.setValues();
     obj.show();
     return 0;
   }
```

Accessing Base Class Members- outside the class

PUBLIC INHERITANCE

```
obj.x = 10; // allowed

obj.display() // allowed
```

PRIVATE INHERITANCE

```
class Derived: private Base{};


Derived obj;

obj.x =10;// error became
               private
```

Order of Constructor Call

When an object of derived class is created: - Base class constructor is called first, then the derived class constructor

```
#include <iostream>
using namespace std;
class Base{
public:
    Base(){
       cout <<"Base constructor "<< endl;}
};
class Derived: public Base {
public:
   Derived(){
            cout <<"Derived constructor "<<endl;
          }
};
int main(){
           Derived d;
}
```

Base constructor
Derived Constructor

Example- Base Class Constructors

We have the following class hierarchy:

```
class Child : public Parent
{ /* code omitted */ };
```

The Child constructor will automatically call the Parent's default constructor

```
Child::Child(int a, int b)
{
    // Parent's default constructor automatically called
    // Constructor code here
}
```

Order of Constructor Call- parameterized

If the base class has only parameterized constructor, the derived class must explicitly call it.

```
#include <iostream>
using namespace std;
class Base{
public:
    Base(int x){
       cout <<"Base constructor "<< endl;}
};
class Derived: public Base {
public:
   Derived(int y):Base(y){ // calling base constructor
            cout <<"Derived constructor "<<endl;
          }
};
int main(){
           Derived d= Derived(5);
}
```

Base constructor is called using initializer list. If you don't call it -> COMPILATION ERROR

Base Class Constructors

Every class in the inheritance hierarchy must have one of its constructors called

- Either implicitly - automatically
- Or explicitly – as show below

Calling Parent’s constructor from Child:

```
Child::Child(int a, int b) : Parent(a, b)
{
    // Constructor code here
}
```

Order of Constructor Call-

multiple constructors

Derived class can choose which base constructor to call.

```
#include <iostream>
using namespace std;
class Base{
public:
    Base(){ cout <<"Base default "<< endl;}
    Base(int x){ cout <<"Base para "<< endl;}
};
class Derived: public Base {
public:
   Derived():Base(10){ // calling base constructor
            cout <<"Derived constructor "<<endl;
          }
};
int main(){
           Derived d;
}
```

Order of Constructor Call-
multiple inheritance

In multiple inheritance, Constructors are called in order of inheritance(left -> right).

```
#include <iostream>
using namespace std;
class A{
    public:
      A(){ cout <<"A"<< endl;}
};
class B {
    public:
     B(){  cout <<"B "<<endl;}
};
class C:public A, public B {
    public:
     C(){  cout <<"C "<<endl;}
};
int main(){
           C obj;
}
```

A
B
C

Order of Constructor Call-
multilevel inheritance

```
#include <iostream>
using namespace std;
class A{
    public:
      A(){ cout <<"A"<< endl;}
};
class B :public A {
    public:
     B(){  cout <<"B "<<endl;}
};
class C:public B {
    public:
     C(){  cout <<"C "<<endl;}
};
int main(){
           C obj;
}
```

A
B
C

Order of Constructor Call-
Diamond Problem

```
class A{
    public:
      A(){ cout <<"A"<< endl;}
};
class B :public A {
    public:
     B(){  cout <<"B "<<endl;}
};
class C:public A {
    public:
     C(){  cout <<"C "<<endl;}
};
class D:public B, public C {
    public:
     D(){  cout <<"D "<<endl;}
};
int main(){
           D obj;
}
```

A
B
A
C
D

```
class A{
    public:
      A(){ cout <<"A"<< endl;}
};
class B :virtual public A {
    public:
     B(){  cout <<"B "<<endl;}
};
class C: virtual public A {
    public:
     C(){  cout <<"C "<<endl;}
};
class D:public B, public C {
    public:
     D(){  cout <<"D "<<endl;}
};
int main(){
           D obj;
}
```

A
B
C
D

Virtual and Parameterized Constructor

```
class A{
    public:
      A(int x){ cout <<"A: "<<x<< endl;}
};
class B :virtual public A {
    public:
     B():A)10){}
};
class C:virtual public A {
    public:
     C():A(20){}
};
class D:public B, public C {
    public:
     D(): A(100){}
};
int main(){
          D obj;
}
```

A: 100

```
D() {}  // no A()
```

If the most derived doesn't initialize virtual base -> Compilation error(if no default constructor in A)

Let's Practice

You are building a simple system to represent vehicles.

- Create a base class `Vehicle` with following attributes.
  - Protected - > `brand` (string) and `year`(int)
  - Public -> parameterized constructor
- Create a derived class `Car` that inherits from Vehicle. Add protected `numDoors`(int)
- Create a constructor for `Car` that initializes `brand` and `year` using base class constructor. Also, initialize `numDoors`. Add a method `displayCar`() that prints details.
- Create another derived class `Motorcycle` from Vehicle that adds private `hasSideCar(bool)`
- Use constructor initializer list and print `brand`, `year` and whether it has a `sidecar`.

Let’s Practice

- Modify `Vehicle`:
  - Add default constructor
  - Print “`vehicle created`” when it runs
  - Which constructor runs first : base or derived ?
- Create a class `ElectricCar` that inherits from `Car` .
- Add `batterycapacity`(int) Constructor must initialize : `Vehile -> Car-> ElectricCar` chain properly
- Add a method to display all details .
- Add a constructor print messages:

  “`Vehicle constructor`”

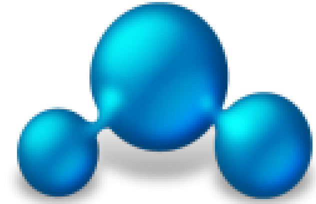
  “`Car constructor`”

  “`Electric constructor`”
- Modify all classes and add destructors with print messages. What's the order of destructor calls?

Example Main (for testing)

```
int main()
{
 Car c(“Toyota”,2026,4);
 c.displayCar();

Motorcycle m(“XYZ”,2014,true);
m.displayInfo();

ElectricCar e(“Tesla”,2023,4,75);
e.displayCar();
return 0;
}
```

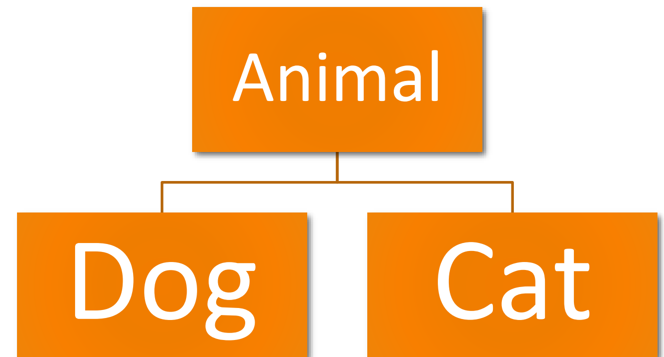
Polymorphism



- Same function name, different behavior
- Treating a child class object as an object of its parent class

- Literally means “many forms”

- Works with variables and pointers!



Polymorphism with Variables

```
// Create some objects (on the stack)
Dog dog = Dog( );
Cat cat = Cat( );
Animal animal = Animal( );

// Attempt to store a child in a parent variable
Animal a2 = dog;                           // Implicit cast
Animal a3 = (Animal)cat;           // Explicit cast

// Cats and dogs are animals!
```

Polymorphism with Pointers

```
// Create some objects (on the heap)
Dog* dog = new Dog( );
Cat* cat = new Cat( );
Animal* animal = new Animal( );


// Attempt to store a child in a parent variable
Animal* a2 = dog;                // Implicit cast
Animal* a3 = (Animal*)cat;       // Explicit cast


// Cat pointers and dog pointers are
// animal pointers
```

Virtual methods

Virtual methods are intended to be over-ridden by derived (child) classes

E.g.

```
class Animal
{
    public:
    Animal(); //Default ctor
    virtual void Speak();//Will over-ride
…
    private:
    …
};
```

ss

```
class Horse
{
    public:
    …
    void Speak(); //Make it neigh!!
…
    private:
    …
};
```

Upcast and downcast

Downcast:

- Convert parent class to child

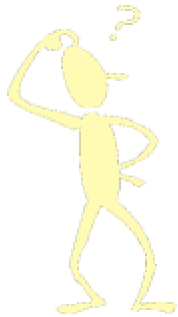
Upcast:

- Convert child class to parent

```
Parent parent;
Child child;

// upcast - implicit type cast allowed
Parent *pParent = &child;

// downcast - explicit type case required
Child *pChild =  (Child *) &parent;

pParent -> sleep();
pChild -> gotoSchool();
```

Abstraction

- Hide internal implementation, show only functionality.
- Abstraction means removing details of features, properties, or functions and emphasizing the more important/ relevant ones ...



"Relevant" to what?

... relevant to the given project (with an eye to future reuse in similar projects)

Abstraction => managing complexity

More Abstraction

- Abstraction is something we do every day
  - Looking at an object, we see those things about it that have meaning to us
  - We abstract the properties of the object, and keep only what we need
  - e.g. students get "name" but not "color of eyes"
- Allows us to represent a complex reality in terms of a simplified model
- Abstraction highlights the properties of an entity that we need and hides the others

Example-1

```
class ATM{
public:
        void withdraw(int amount) {
        authenticate();
        processTransaction(amount);
     }
 Private:
        void authenticate(){
        cout <<"Authentication done \n";
 }
 Void processTransaction(int amount){
        cout<< "Withdrawn : " << amount;
 }
```

Example -2

Abstraction is accomplished by hiding details of implementation

```
#include <iostream>
using namespace std;

class sample {
public:
    int g1, g2;

public:
    void val() {
        cout << "Enter Two values : "; cin >> g1 >> g2;
    }
    void display() {
        cout << g1 << " " << g2;
        cout << endl;
    }
};
int main(){
    sample S;
    S.val();
    S.display();
}
```

The details of how data is input, stored – and later displayed, is all hidden from the user. i.e. all the implementation is abstracted into the class 'sample'