OBJECT ORIENTED PROGRAMMING

Inheritance

Initializer list

- Initializer list is used to initialize data member(s) of the class
- The list of members to be initialized is indicated with constructor as a comma-separated list followed by a colon

```
class Point {
private:
    int x;
    int y;
public:
    Point(int i = 0, int j = 0):x(i), y(j) {}
    /* The above use of Initializer list is optional
    as the constructor can also be written as:
        Point(int i = 0, int j = 0) {
            x = i;
            y = j;
        }
    */

    int getX() const {return x;}
    int getY() const {return y;}
};
```

But there are situations where initialization of data members inside constructor doesn't work and Initializer list must be used.

Uses of initializer list

1. For initialization of non-static const data members
2. For initialization of member objects which do not have default constructor

```
class Test {
    const int t;
    int x;
    int y;
public:
    Test(int a, int b, int c):t(a)
    {
      x=b;
      y=c ;

    }
  int getT() { return t; }
};

int main() {
    Test t1(11, 12, 10);
    cout<<t1.getT();
    return 0;
}
```

```
class A {
    int x;
public:
    A(int a )
    {
        x = a;
        cout << "A's Constructor called: Value of : x ";
        cout<< x << endl;
    }
};

// Class B contains object of A
class B {
    A objA;
    int y;
public:
    B(int b, int c): objA(b)
    {
        y=c;
        cout << "B's Constructor called";
    }
};


int main() {
    B obj(10, 11);
    return 0;
}
```

Uses of initializer list (continued)

3. For initialization of reference members
4. When constructor's parameter name is same as data member

```
class Test {
    int &t;
    //other member variables
public:
    Test(int &t):t(t)
    {
        //other members
    }
    int getT()
    {
        return t;
    }
};
int main() {
    int x = 20;
    Test t1(x);
    cout<<t1.getT()<<endl;
    x = 30;
    cout<<t1.getT()<<endl;
    return 0;
}
```

```
class A {
    int i;
    //other member variables
public:
    A::A(int i):i(i)
    {
        //other member variables

    } // Either Initializer list or this pointer must be used
    /* The above constructor can also be written as
    A(int i) {
    this->i = i;
    }
    */
    int getI()
    {
        return i;
    }
};

int main() {
    A objA(10);
    cout<<objA.getI();
    return 0;
}
```

Uses of initializer list (continued)

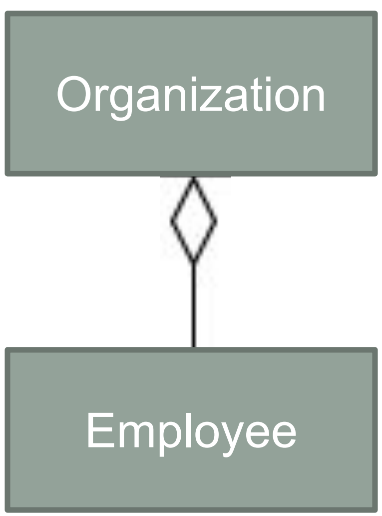
5. For initialization of base class members

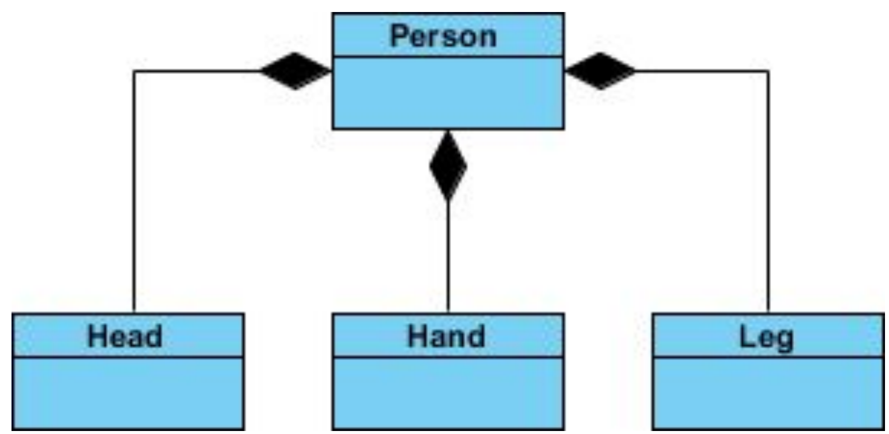
   Parameterized constructor of the base class **can only** be called using Initializer List

```
class A {
    int i;
public:
    A(int );
};

A::A(int arg) {
    i = arg;
    cout << "A's Constructor called: Value of i: " << i << endl;
}

// Class B is derived from A
class B: public A {
public:
    B(int );
};

B::B(int x):A(x) { //Initializer list must be used
    cout << "B's Constructor called";
}

int main() {
    B obj(10);
    return 0;
}
```

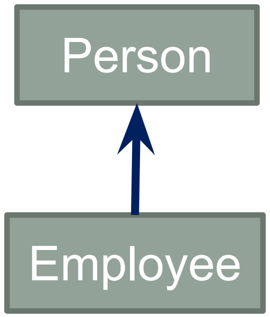
Inheritance and Composition

- The common ways to relate two classes in a meaningful way are:
  - Inheritance (“is-a” relationship)
  - Composition / Aggregation (“has-a” relationship)





Inheritance



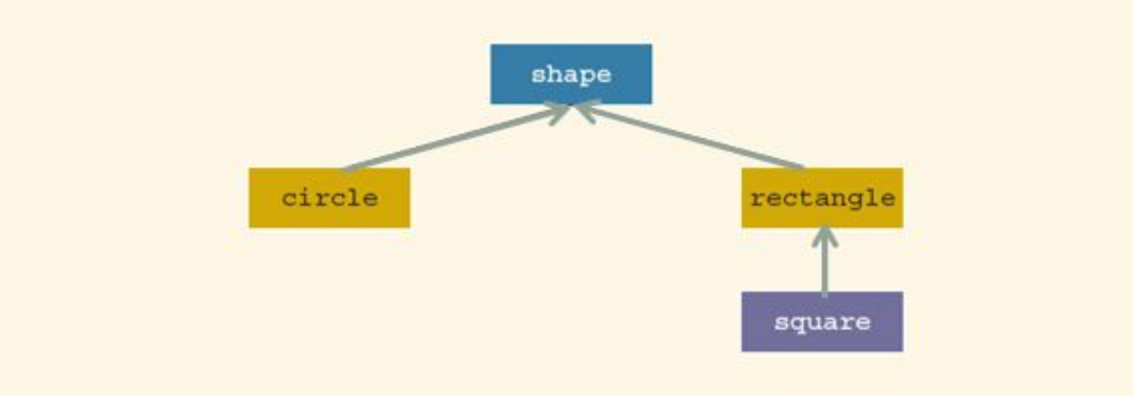
- Inheritance is an "is-a" relationship
  - Example: "every employee is a person"
  - Person's attributes: First Name, Last Name and CNIC No.
  - Employee's attributes: First Name, Last Name, CNIC No., Employee ID, Salary, Joining Date etc.
- Inheritance lets us create new classes from existing classes
  - New classes are called the derived classes/child classes
  - Existing classes are called the base classes / parent classes
- Derived classes inherit the properties of the base classes

Examples

Base class	Derived classes
• Student	• Graduate Student • Undergraduate student
• Shape	• Circle • Rectangle
• Employee	• Faculty Member • Staff member

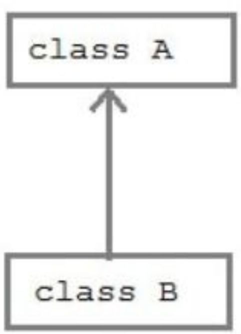
Inheritance (continued)

- Inheritance can be viewed as a tree-like, or hierarchical, structure wherein a base class is shown with its derived classes

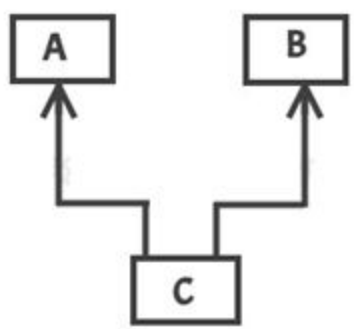


Inheritance (continued)

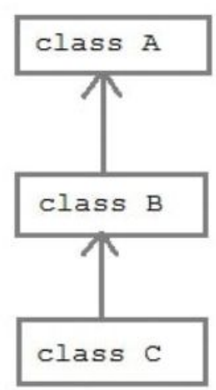
- Single inheritance: derived class has a single base class
  - Ex. Circle class from Shape class
- Mutlilevel Inheritance: where one can inherit from a derived class, thereby making this derived class the base class for the new class.
  - **Ex. Son->Father->Grand Father**
- Multiple inheritance: derived class has more than one base class
  - Ex. Son class from Mother class and Father class



Simple Inheritance



Multiple Inheritance



Multilevel inheritance

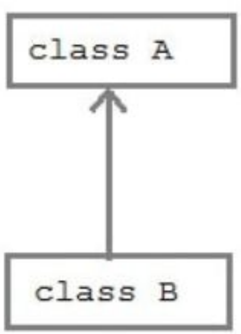
Defining a simple subclass (1)

We can use each class as a base (or a foundation) to define or build another class (**a subclass**). It's also possible to use **more than one class to define a subclass**. You can see both of these cases given below.

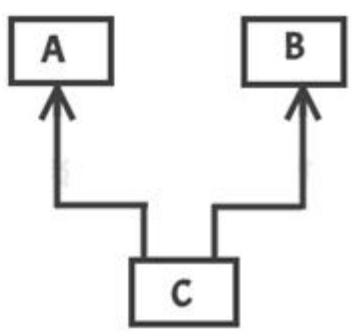
Note that the arrows always point to the superclass(es).

The left diagram illustrates a "**single inheritance**", and the right one a "**multiple inheritance**" or "**multi-inheritance**".

We can also write about super classes as **base** classes, and subclasses as **derived** classes.



Simple Inheritance



Multiple Inheritance

Defining a simple subclass (2)

The class on the right → will serve as **a superclass**. Analyze its structure – it's not difficult, we promise.

The program emits the following text:

101

```
class Super
{
private:
    int storage;
public:
    void put(int val){ storage=val; }
    int get(){return storage;}
};

int main()
{
    Super object;
    object.put(100);
    object.put(object.get()+1);
    cout << object.get() << endl;
    return 0;
}
```

Inheritance (continued)

- General syntax of a derived class:

```
class className: memberAccessSpecifier baseClassName
{
    member list
};
```

  - Where `memberAccessSpecifier` is `public`, `protected`, or `private` (default)
- The `private` members of a base class are inaccessible in the derived class
  - Derived class cannot directly access them
  - Ex. `Class` SonClass: `public` FatherClass

```
{
        };
```

Defining a simple subclass (3)

If we want to **define a class named Y as a subclass of a superclass named X**, we use the following syntax

```
class Y : private or public or protected X
  {
     member list;
  };
```

If there's more than one superclass, we have to enlist them all using commas as separators, like this:

```
class A : X, Y, Z
   {
       member list;
  };
```

Let's start with the simplest possible case.

Defining a simple subclass (4)

Take a look here → We've defined a class named *Sub*, which is a subclass of a class named *Super*. We may also say that the *Sub* class is derived from the *Super* class.

The *Sub* class introduces neither new variables nor new functions. Does this mean that any object of the *Sub*class inherits all the traits of the *Super* class, being in fact a copy of the *Super* class's objects?

No. It doesn't.

```
class Super
{
private:
    int storage;
public:
    void put(int val){ storage=val; }
    int get(){return storage;}
};

class Sub: Super
{
};

int main()
{
    Sub object;
    object.put(100);
    object.put(object.get()+1);
    cout << object.get() << endl;
    return 0;
}
```

6.1.4 Defining a simple subclass (4) cont'd

If we compile the following code, we'll get nothing but compilation errors saying that the *put* and *get* methods are inaccessible. Why?

When we **omit the visibility specifier**, the compiler assumes that we're going to apply a "**private inheritance**". This means that **all public superclass components turn into private access, and private superclass components won't be accessible** at all. It consequently means that you're not allowed to use the latter inside the subclass.

This is exactly what we want now.

Defining a simple subclass (4) cont'd

We have to tell the compiler that **we want to preserve the previously used access policy**. We do this by using a "public" visibility specifier:

class Sub : **public** Super { };

Don't be misled: this doesn't mean that the private components of the *Super*class (like the *storage* variable) will magically turn into public ones. Private components will remain inaccessible, public components will remain public.

Defining a simple subclass (5)

Objects of the *Sub* class may do almost the same things as their older siblings created from the *Super*class. We use the word ‘almost’ because being a subclass also means that **the class has lost access to the private components of the superclass**.

We cannot write a member function of the *Sub* class which would be able to directly manipulate
the *storage* variable.

```
class Super
{
private:
    int storage;
public:
    void put(int val){ storage=val; }
    int get(){return storage;}
};

class Sub: public Super
{
};

int main()
{
    Super object;
    object.put(100);
    object.put(object.get()+1);
    cout << object.get() << endl;
    return 0;
}
```

Defining a simple subclass (5) cont'd

There's the third access level we haven't mentioned yet. It's called "**protected**".

The keyword *protected* means that any component marked with it **behaves like a public component when used by any of the subclasses and looks like a private component to the rest of the world**.

We should add that this is true only for **publicly inherited** classes (like the *Super* class in our example previous example)

Let's make use of the keyword right now.

Defining a simple subclass (6)

As you can see in the example code →
we've added some new functionality to
the *Sub* class.

We've added the *print* function. It isn't especially sophisticated, but it does one important thing: it accesses the *storage* variable from the Superclass. This wouldn't be possible if the variable was declared as **private**.

In the *main* function scope, the variable remains hidden anyway. You mustn't write anything like this:

object.storage = 0;

The compiler will be very stubborn about this.

We almost forgot to mention that our new program will produce the following output:

storage = 101

```
#include <iostream>

using namespace std;

class Super {
protected:
	int storage;
public:
	void put(int val) { storage = val; }
	int get(void) { return storage; }
};

class Sub : public Super {
public:
	void print(void) { cout << "storage = " << storage << endl; }
};

int main(void) {
	Sub object;

	object.put(100);
	object.put(object.get() + 1);
	object.print();
	return 0;
}
```

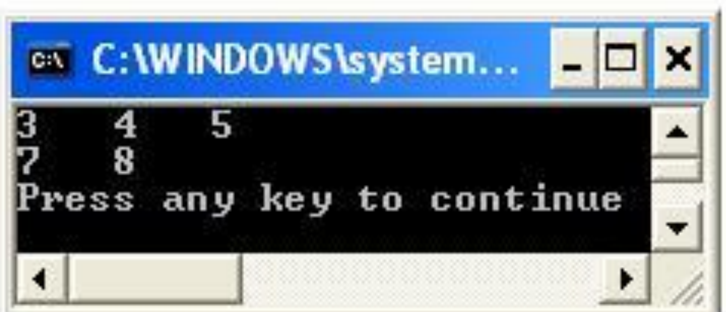
```
#include <iostream>
using namespace std;;

class A          Base
{
     int x;
     int y;
public:
     int z;
     void printA(){cout<<x<<"  "<<y<<"   "<<endl;}
     void setA(int a, int b, int c){x=a; y=b; z=c;}
};

class B: public A  derived
{
     int var1;
public:
     int var2;
     void setB(int a, int b) {var1=a; var2=b;}
     void printB(){cout<<var1<<"  "<<var2<<endl;}
};

void main()
{
     A objA;
     B objB;
}
```

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

class A
{
     int x;
     int y;
public:
     int z;
     void printA(){cout<<x<<"  "<<y<<"   "<<endl;}
     void setA(int a, int b, int c){x=a; y=b; z=c;}
};

class B: public A
{
     int var1;
public:
     int var2;
     void setB(int a, int b) {var1=a; var2=b;}
     void printB(){cout<<var1<<"  "<<var2<<endl;}
};

void main()
{
     A objA;
     B objB;
}
```

Can not access x, y

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

class A
{
     int x;
     int y;
public:
     int z;
     void printA(){cout<<x<<"  "<<y<<"   "<<endl;}
     void setA(int a, int b, int c){x=a; y=b; z=c;}
};

class B: private A
{
     int var1;
public:
     int var2;
     void setB(int a, int b) {var1=a; var2=b;}
     void printB(){cout<<var1<<"  "<<var2<<endl;}
};

void main()
{
     A objA;
     B objB;
}
```

```
#include <iostream>
using namespace std;;
class A
{
     int x;
     int y;
public:
     int z;
     void printA(){cout<<x<<"    "<<y<<"    "<<z<<endl;}
     void setA(int a, int b, int c){x=a; y=b; z=c;}
};
class B:public A
{
     int var1;
public:
     int var2;
     void setB(int a, int b) {var1=a; var2=b;}
     void printB(){cout<<var1<<"    "<<var2<<endl;}
};

void main()
{
     B objB;
     objB.setA(3,4,5);
     objB.setB(7,8);
     objB.printA();
     objB.printB();
}
```



Defining a simple subclass (7)

Now's a good opportunity to do a little summarizing here. We know that any component of the class may be declared as:

- public
- private
- protected

These three keywords may also be used in a completely different context to specify the visibility inheritance model. So far, we've talked about public and private keywords used in such a case. It should be no surprise to you that the protected keyword can be employed in this role, too.

Accessibility

- Accessibility:
  - Private < Protected < Public

When the component is declared as:	When the class is inherited as:	The resulting access inside the subclass is:
public	public	Public
protected		protected
private		none
public	protected	protected
protected		protected
private		none
public	private	private
protected		private
private		none

Inheritance (continued)

- `public` members of base class can be inherited as `public` or `private` members
- The derived class can include additional members--data and/or functions
- The derived class can redefine the `public` member functions of the base class
- All members of the base class are also member variables of the derived class

End of the Lecture