

Applied Programming

Engr. Abdul-Rahman Mahmood

DPM, MCP, QMR(ISO9001:2000)

 armahmood786@yahoo.com


 alphapeeler.sf.net/pubkeys/pkey.htm

 pk.linkedin.com/in/armahmood

 www.twitter.com/alphapeeler

 www.facebook.com/alphapeeler

 abdulmahmood-sss  alphasecure

 armahmood786@hotmail.com

 <http://alphapeeler.sf.net/me>

 alphasecure@gmail.com

 <http://alphapeeler.sourceforge.net>

 <http://alphapeeler.tumblr.com>

 armahmood786@jabber.org

 alphapeeler@aim.com

 mahmood_cubix  48660186

 alphapeeler@icloud.com

 <http://alphapeeler.sf.net/acms/>

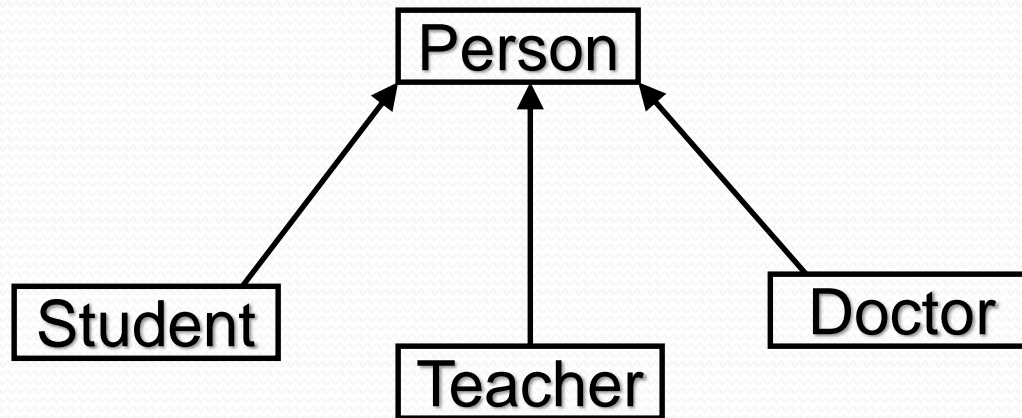
Inheritance

- A child inherits characteristics of its parents
- Besides inherited characteristics, a child may have its own unique characteristics

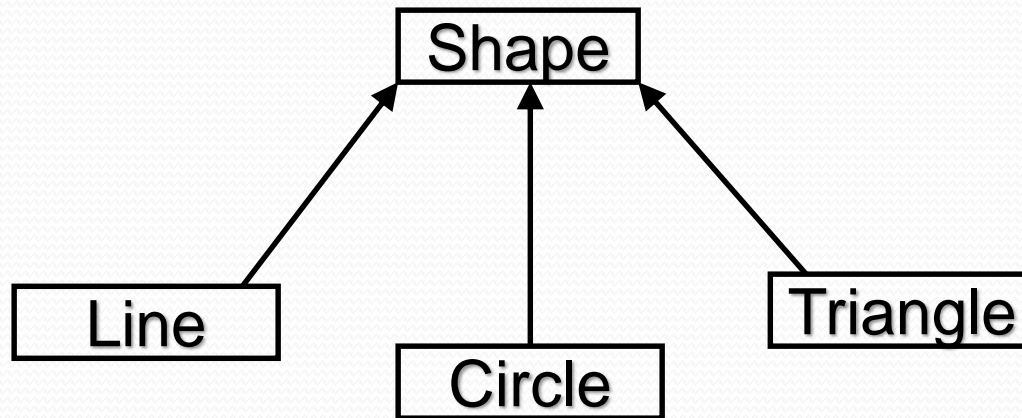
Inheritance in Classes

- If a class B inherits from class A then it contains all the characteristics (information structure and behaviour) of class A
- The parent class is called *base* class and the child class is called *derived* class
- Besides inherited characteristics, derived class may have its own unique characteristics

Example – Inheritance



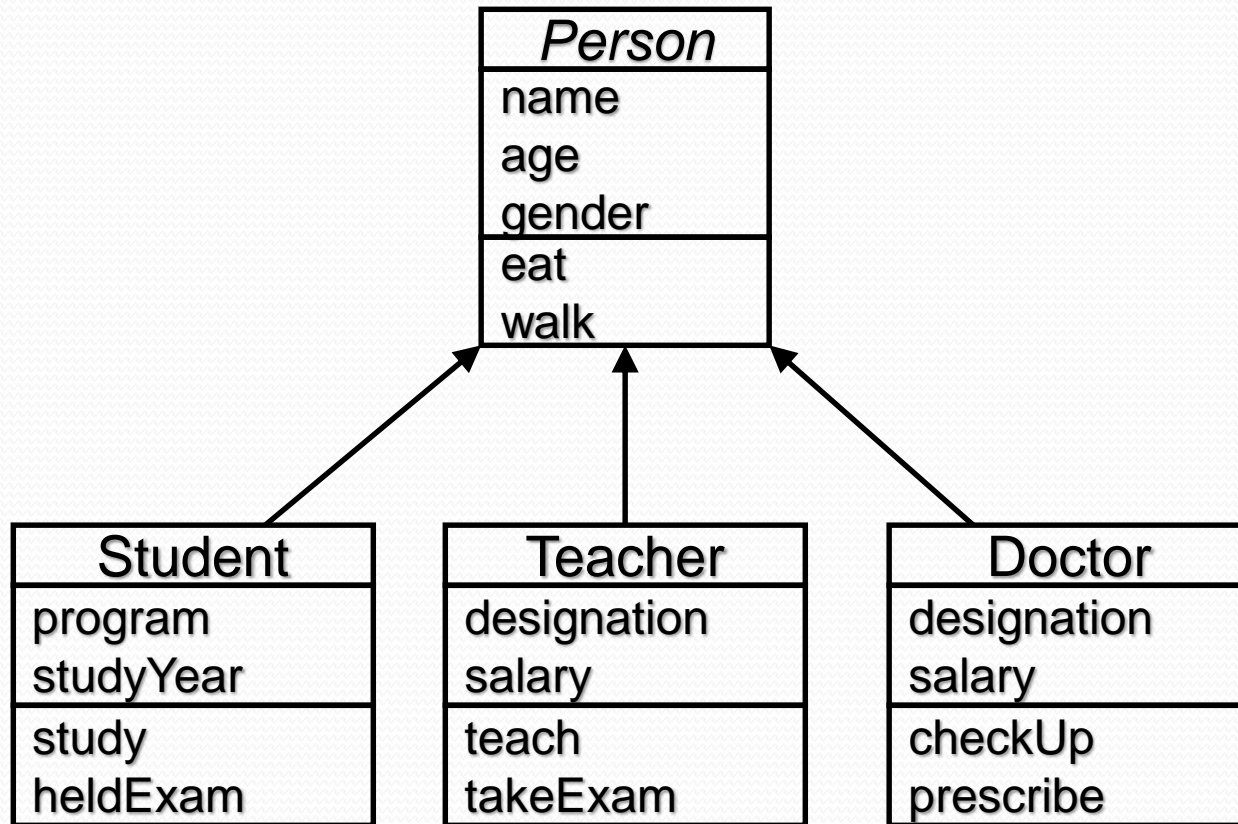
Example – Inheritance



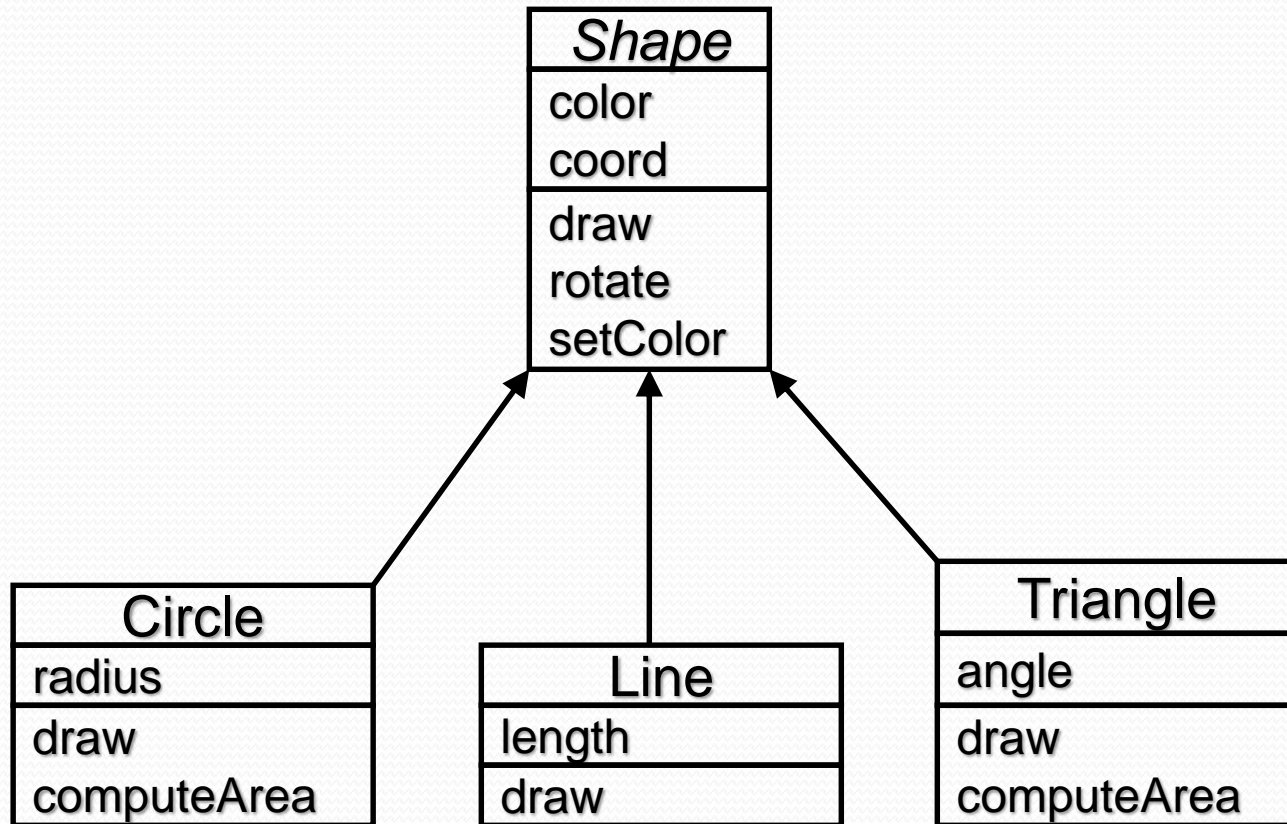
Inheritance – “IS A” or “IS A KIND OF” Relationship

- Each derived class is a special kind of its base class

Example – “IS A” Relationship



Example – “IS A” Relationship



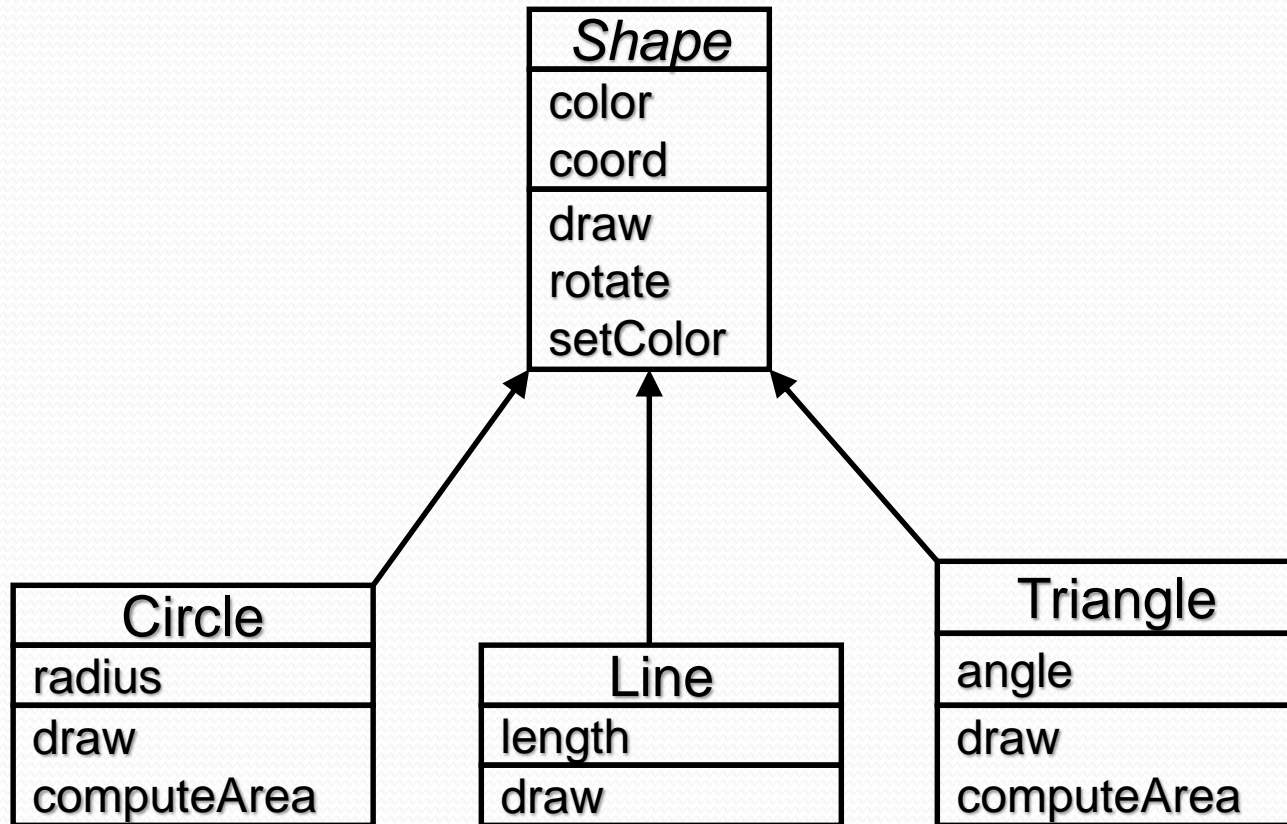
Inheritance – Advantages

- Reuse
- Less redundancy
- Increased maintainability

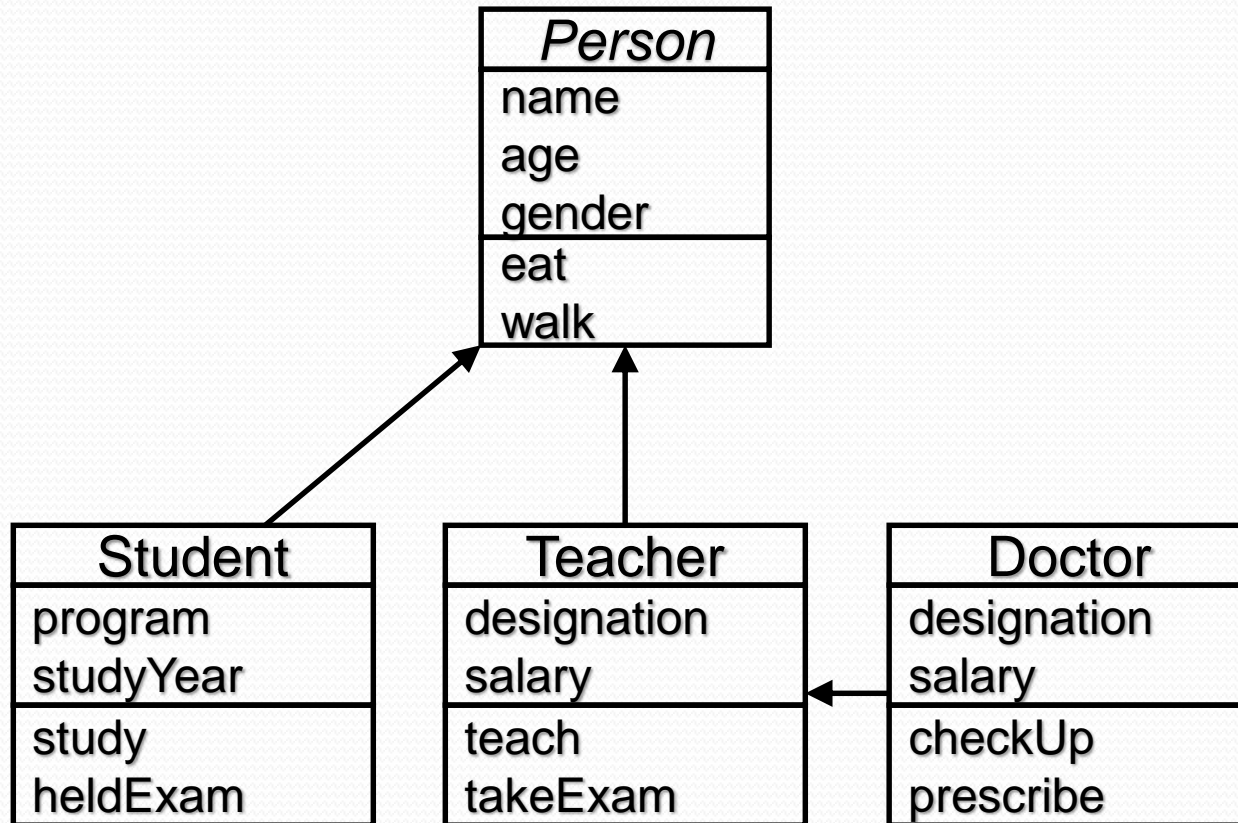
Reuse with Inheritance

- Main purpose of inheritance is reuse
- We can easily add new classes by inheriting from existing classes
 - Select an existing class closer to the desired functionality
 - Create a new class and inherit it from the selected class
 - Add to and/or modify the inherited functionality

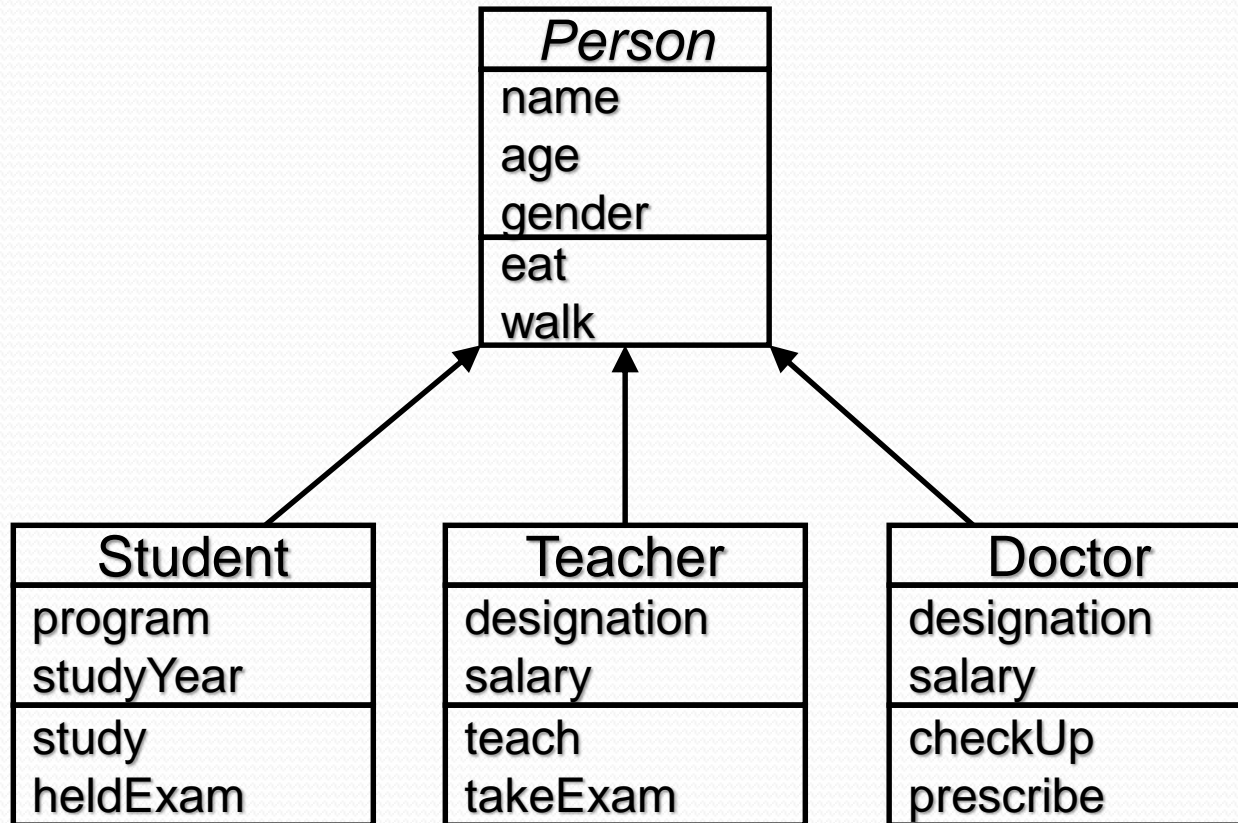
Example Reuse



Example Reuse



Example Reuse



C++ Inheritance

Reminder about public, private and protected access specifiers:

- 1 If a member or variables defined in a class is private, then they are accessible by members of the same class only and cannot be accessed from outside the class.
- 2 Public members and variables are accessible from outside the class.
- 3 Protected access specifier is a stage between private and public. If a member functions or variables defined in a class are protected, then they cannot be accessed from outside the class but can be accessed from the derived class.

Type of Inheritance

When deriving a class from a base class, the base class may be inherited through **public**, **protected** or **private** inheritance. The type of inheritance is specified by the access- specifier.

We hardly use **protected** or **private** inheritance, but **public** inheritance is commonly used. While using different type of inheritance, following rules are applied:

Type of Inheritance

✓ **Public Inheritance:** When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private** members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.

Type of Inheritance

✓ **Protected Inheritance:** When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived class.

✓ **Private Inheritance:** When deriving from a **private** base class, **public** and **protected** members of the base class become **private** members of the derived class

C++ Inheritance

Inheritance Example:

```
class MyClass
```

```
{    public:  
    MyClass(void) { x=0; }  
    void f(int n1)  
    { x= n1*5;}  
    void output(void) { cout<<x; }  
    private:  
    int x;  
};
```

C++ Inheritance

Inheritance Example:

```
class sample: public MyClass
{ public:
  sample(void) { s1=0; }
  void f1(int n1)
      { s1=n1*10;}
  void output(void)
  { MyClass::output();  cout << s1; }
private:
  int s1;
};
```

C++ Inheritance

Inheritance Example:

```
int main(void)
{
    sample s;
    s.f(10);
    s.output();
    s.f1(20);
    s.output();
}
```

The output of the above program is

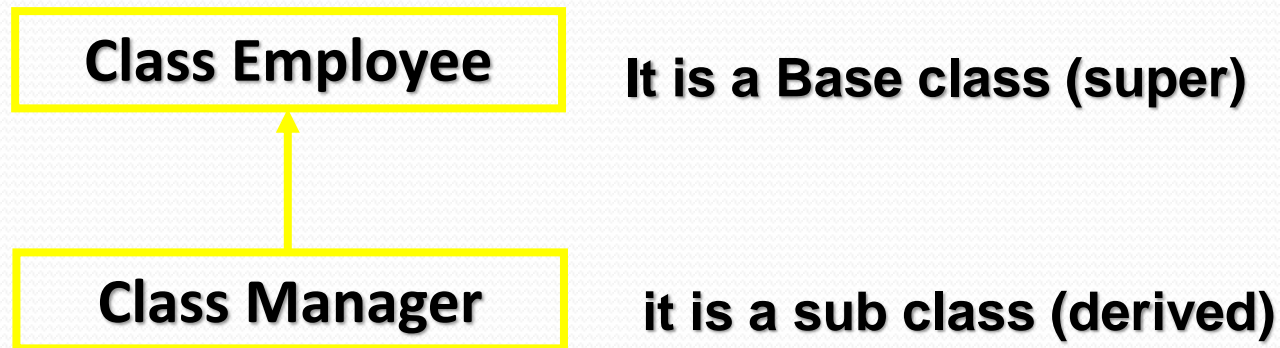
50

200

Types of Inheritance

1. Single class Inheritance:

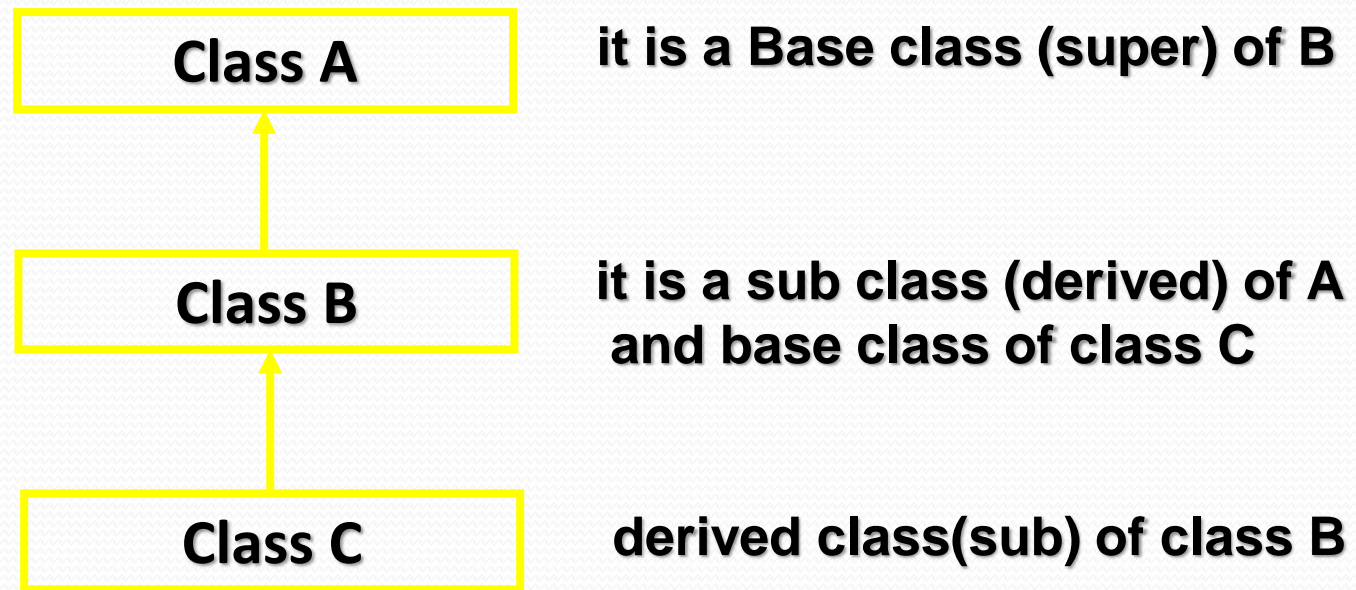
Single inheritance is the one where you have a single base class and a single derived class.



Types of Inheritance

2. Multilevel Inheritance:

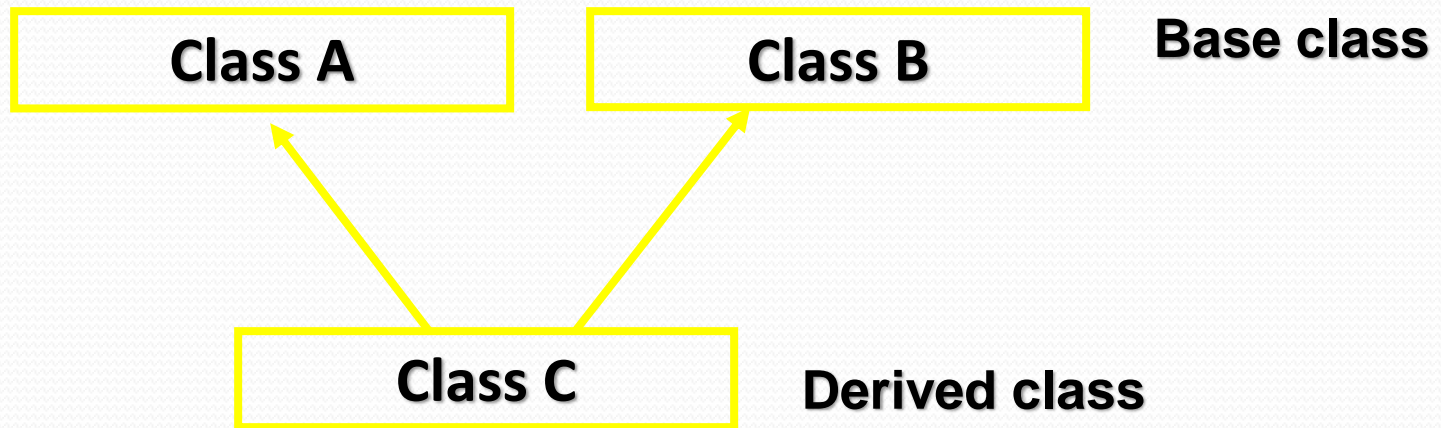
In Multi level inheritance, a class inherits its properties from another derived class.



Types of Inheritance

3. Multiple Inheritances:

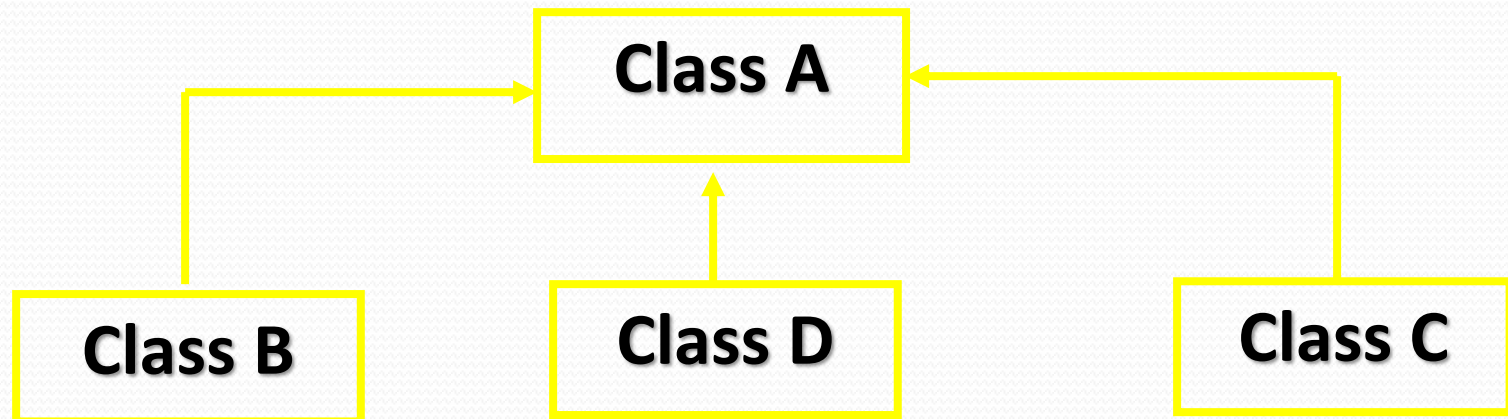
In Multiple inheritances, a derived class inherits from multiple base classes. It has properties of both the base classes.



Types of Inheritance

4. Hierarchical Inheritance:

In hierarchical Inheritance, it's like an inverted tree. So multiple classes inherit from a single base class. It's quite analogous to the File system in a unix based system.



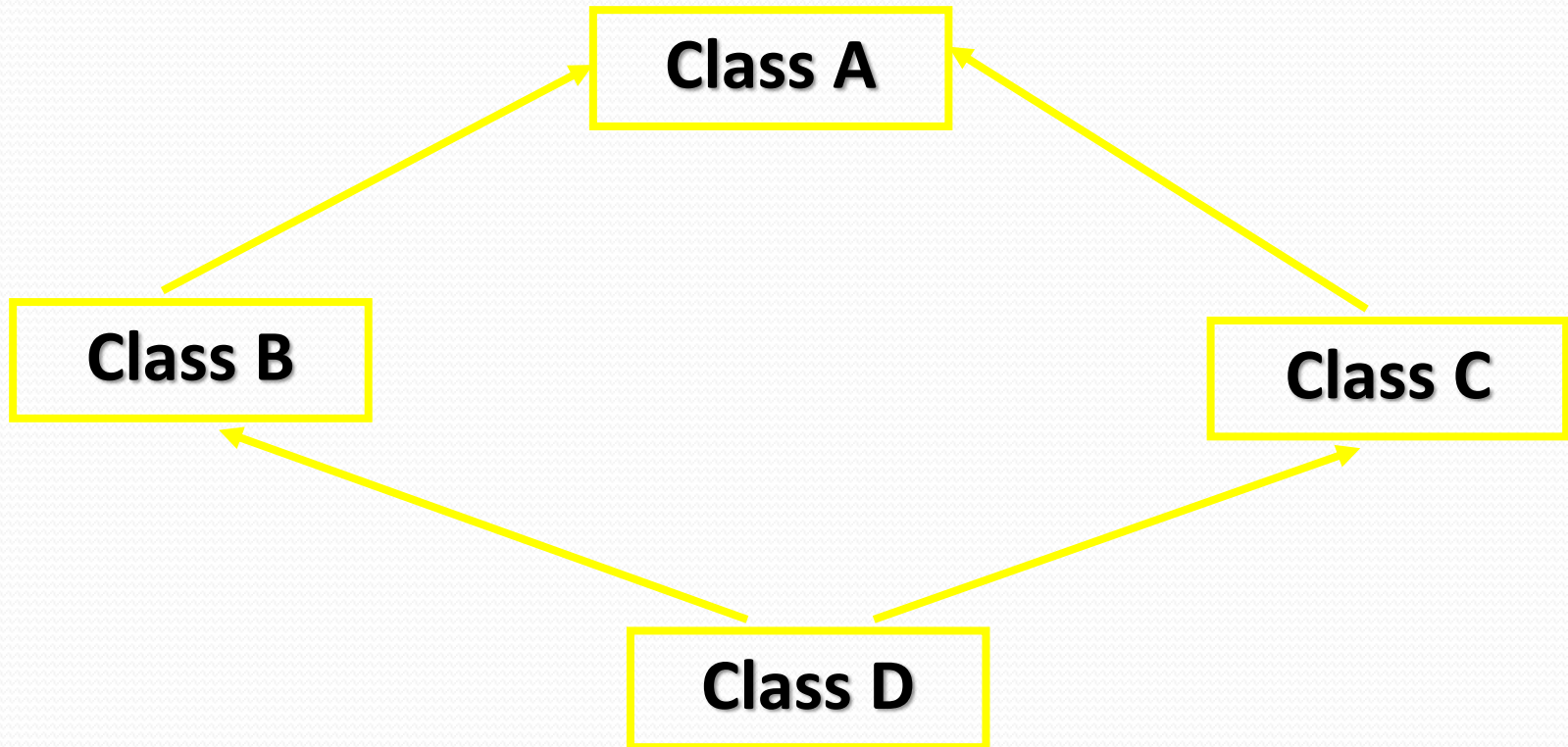
Types of Inheritance

5. Hybrid Inheritance:

- ✓ In this type of inheritance, we can have mixture of number of inheritances but this may generate an error of using same name function from no of classes, which will bother the compiler to how to use the functions.
- ✓ Therefore, it will generate errors in the program. This has known as ambiguity or duplicity.
- ✓ Ambiguity problem can be solved by using **virtual base classes**

Types of Inheritance

5. Hybrid Inheritance:



Concept of Inheritance

- Conceptual examples

- example 1:

base class: circle

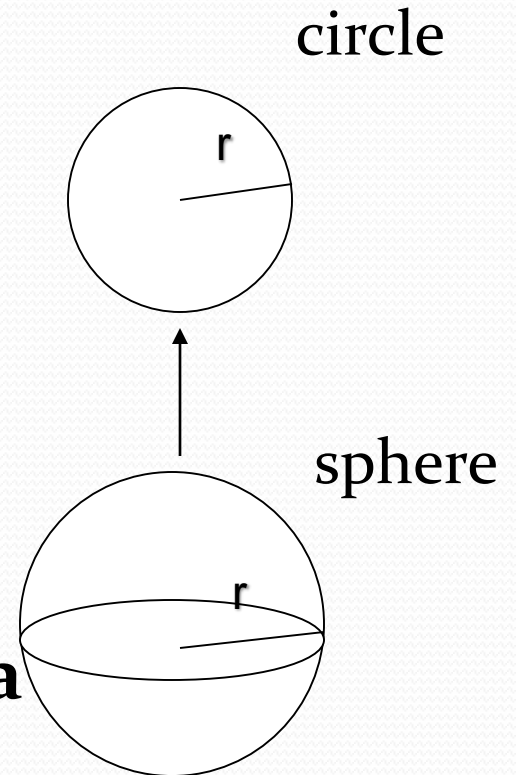
$$\text{area} = 3.1415 * r * r$$

derived class: sphere

$$\text{area} = 4 * \text{circle}::\text{area}$$

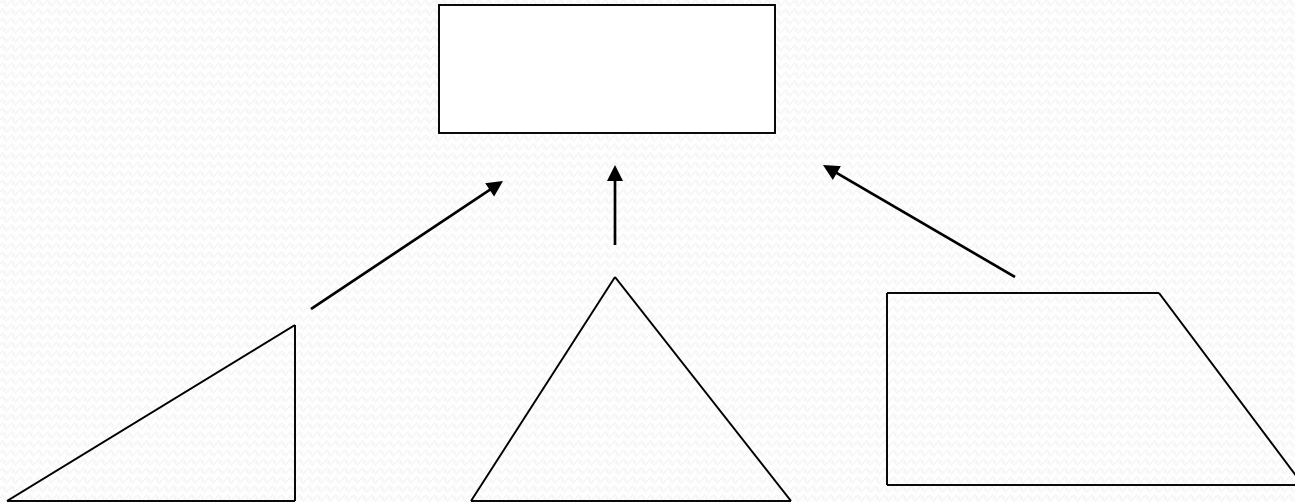
$$\text{volume} = 4/3 * \text{circle}::\text{area}$$

Sphere is kind of circular shape



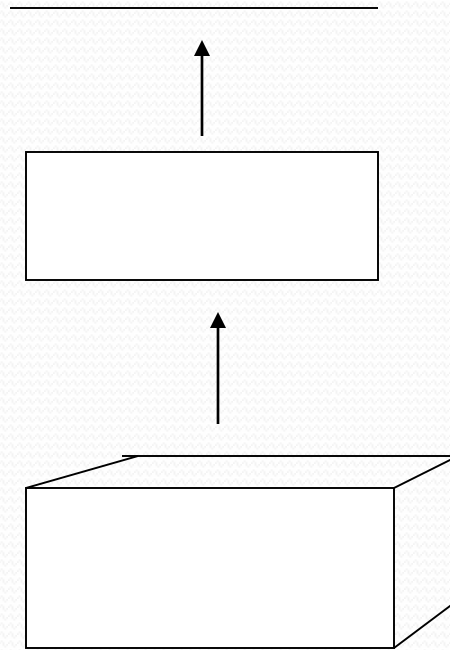
Types of Inheritance (continue)

- Example of simple inheritance



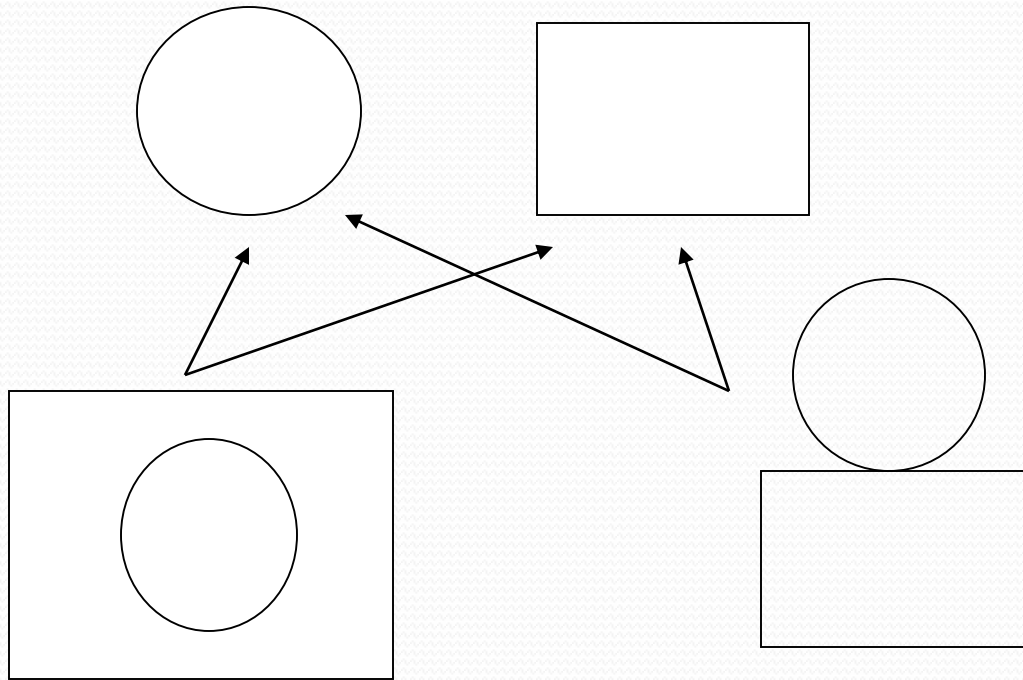
Types of Inheritance (continue)

- Example of multi-level inheritance



Types of Inheritance (continue)

- Example of multiple inheritance



Defining a Derived class with visibility mode

- Visibility Mode
- 1- Public
- 2- Private
- 3- Protected

Base Class Visibility	Derived class visibility		
	Public derivation	Private derivation	Protected derivation
Private	Not inherited	Not inherited	Not inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

Ex 1: Inheritance

```
1 //=====
2 // Name      : MSAPW03Ex01.cpp
3 // Author    : Engr. Abdul Rahman
4 //=====
5 #include <iostream>
6 using namespace std;
7
8 //Base class
9 class Parent
10 {
11     public:
12     int id_p;
13 };
14
15 // Sub class inheriting from Base Class(Parent)
16 class Child : public Parent
17 {
18     public:
19     int id_c;
20 };
21
22 //main function
23 int main() {
24     Child obj1;
25     // An object of class child has all data members
26     // and member functions of class parent
27     obj1.id_c = 7;
28     obj1.id_p = 91;
29     cout << "Child id is " << obj1.id_c << endl;
30     cout << "Parent id is " << obj1.id_p << endl;
31     return 0;
32 }
```

Output:
Child id is 7
Parent id is 91

Ex 2: Single Inheritance

```
1 //=====  
2 // Name      : MSAPW03Ex02.cpp  
3 // Author     : Engr. Abdul Rahman  
4 // Version    :  
5 // Copyright  : Your copyright notice  
6 // Description: Hello World in C++, Ansi-style  
7 //=====
```

```
8 // C++ program to explain Single inheritance  
9 #include <iostream>  
10 using namespace std;  
11  
12 // base class  
13 class Vehicle {  
14     public:  
15     Vehicle()  
16     {  
17         cout << "This is a Vehicle" << endl;  
18     }  
19 };  
20  
21 // sub class derived from two base classes  
22 class Car: public Vehicle{  
23  
24 };  
25  
26 // main function  
27 int main()  
28 {  
29     // creating object of sub class will  
30     // invoke the constructor of base classes  
31     Car obj;  
32     return 0;  
33 }
```

Output:
This is a Vehicle

Ex 3: Multiple Inheritance

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

class Vehicle { // first base class
public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};

class FourWheeler { // second base class
public:
    FourWheeler()
    {
        cout << "This is a 4 wheeler Vehicle" << endl;
    }
};

// sub class derived from two base classes
class Car: public Vehicle, public FourWheeler {

};

int main()
{
    // creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Output:

This is a Vehicle

This is a 4 wheeler Vehicle

Ex 4: Multilevel Inheritance

```
//=====
// Name      : MSAPW03Ex04.cpp
// Author    : Engr. Abdul Rahman
//=====
// C++ program to implement Multilevel Inheritance
#include <iostream>
using namespace std;
class Vehicle { // base class
public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};
class fourWheeler: public Vehicle {
public:
    fourWheeler() {
        cout<<"Objects with 4 wheels are vehicles"<<endl;
    }
};
// sub class derived from two base classes
class Car: public fourWheeler{
public:
    Car() {
        cout<<"Car has 4 Wheels"<<endl;
    }
};
int main() {
    //creating object of sub class will
    //invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Output:

This is a Vehicle

Objects with 4 wheels are vehicles

Car has 4 Wheels

Ex 5: Hierarchical Inheritance

```
//=====
// Name      : MSAPW03Ex05.cpp
// Author     : Engr. Abdul Rahman
//=====
// C++ program to implement
// Hierarchical Inheritance
#include <iostream>
using namespace std;

class Vehicle { // base class
public:
    Vehicle()
    {
        cout << "This is a Vehicle" << endl;
    }
};

class Car: public Vehicle { // first sub class
};

class Bus: public Vehicle { // second sub class
};

int main()
{
    // creating object of sub class will
    // invoke the constructor of base class
    Car obj1;
    Bus obj2;
    return 0;
}
```

Output:
This is a Vehicle
This is a Vehicle

Ex 6: Hybrid Inheritance

```
//=====
// Name      : MSAPW03Ex06.cpp
// Author    : Engr. Abdul Rahman
//=====
// C++ program for Hybrid Inheritance
#include <iostream>
using namespace std;
class Vehicle { // base class
public:
    Vehicle() {
        cout << "This is a Vehicle" << endl;
    }
};
class Fare { //base class
public:
    Fare() {
        cout<<"Fare of Vehicle\n";
    }
};
class Car: public Vehicle { // first sub class
};
class Bus: public Vehicle, public Fare { // second sub class
};
int main()
{
    // creating object of sub class will
    // invoke the constructor of base class
    Bus obj2;
    return 0;
}
```

Output:
This is a Vehicle
Fare of Vehicle

Polymorphism in C++

- The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.
- Real life example of polymorphism, a person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee.
- So the same person posses different behavior in different situations. This is called polymorphism.

Types of polymorphism

- **In C++ polymorphism is mainly divided into two types:**
 - **Compile time Polymorphism**
 - This type of polymorphism is achieved by function overloading or operator overloading.
 - **Runtime Polymorphism**
 - This type of polymorphism is achieved by Function Overriding.

Function Overloading

- When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by **change in number of arguments** or/and **change in type of arguments**.

Ex 7: Function Overloading

```
1 //=====
2 // Name      : MSAPW03Ex07.cpp
3 // Author    : Engr. Abdul Rahman
4 //=====
5 // C++ program for function overloading
6 #include <iostream>
7 using namespace std;
8 class AlphaPeeler {
9     public:
10    // function with 1 int parameter
11    void func(int x) {
12        cout << "value of x is " << x << endl;
13    }
14    // function with same name but 1 double parameter
15    void func(double x)
16    {
17        cout << "value of x is " << x << endl;
18    }
19    // function with same name and 2 int parameters
20    void func(int x, int y)
21    {
22        cout << "value of x and y is " << x << ", " << y << endl;
23    }
24 };
25 int main() {
26     AlphaPeeler obj1;
27     // Which function is called will depend on the parameters passed
28     obj1.func(7);      // The first 'func' is called
29     obj1.func(9.132);  // The second 'func' is called
30     obj1.func(85,64);  // The third 'func' is called
31     return 0;
32 }
```

Output:

value of x is 7

value of x is 9.132

value of x and y is 85, 64

- **Operator Overloading:** C++ also provide option to overload operators. For example, we can make the operator ('+') for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So a single operator '+' when placed between integer operands , adds them and when placed between string operands, concatenates them.

Ex 8: Operator Overloading

```
1 //=====
2 // Name      : MSAPW03Ex08.cpp
3 // Author    : Engr. Abdul Rahman
4 //=====
5 // CPP program to illustrate Operator Overloading
6 #include<iostream>
7 using namespace std;
8
9 class Complex {
10 private:
11     int real, imag;
12 public:
13     Complex(int r = 0, int i = 0) {real = r;  imag = i;}
14
15     // This is automatically called when '+' is used with
16     // between two Complex objects
17     Complex operator + (Complex const &obj) {
18         Complex res;
19         res.real = real + obj.real;
20         res.imag = imag + obj.imag;
21         return res;
22     }
23     void print() { cout << real << " + i" << imag << endl; }
24 };
25
26 int main()
27 {
28     Complex c1(10, 5), c2(2, 4);
29     Complex c3 = c1 + c2; // An example call to "operator+"
30     c3.print();
31 }
```

Output:
12 + i9

Runtime polymorphism

- This type of polymorphism is achieved by Function Overriding. **Function overriding** on the other hand occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

Ex 9: Function overriding

```
3 // Author      : Engr. Abdul Rahman
4 //=====
5 // C++ program for function overriding
6 #include <iostream>
7 using namespace std;
8 class base {
9 public:
10     virtual void print ()
11     { cout<< "print base class" <<endl; }
12     void show ()
13     { cout<< "show base class" <<endl; }
14 };
15
16 class derived:public base {
17 public:
18     void print () //print () is already virtual function in derived class,
19     //we could also declared as virtual void print () explicitly
20     { cout<< "print derived class" <<endl; }
21     void show ()
22     { cout<< "show derived class" <<endl; }
23 };
24
25 int main() {
26     base *bptr;
27     derived d;
28     bptr = &d;
29     //virtual function, binded at runtime (Runtime polymorphism)
30     bptr->print();
31     // Non-virtual function, binded at compile time
32     bptr->show();
33     return 0;
34 }
```

Output:
print derived class
show base class

Virtual Function in C++

- A virtual function is a member function which is declared within a base class and is re-defined (Overridden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.
 - Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
 - They are mainly used to achieve Runtime polymorphism
 - Functions are declared with a **virtual** keyword in base class.
 - The resolving of function call is done at Run-time.

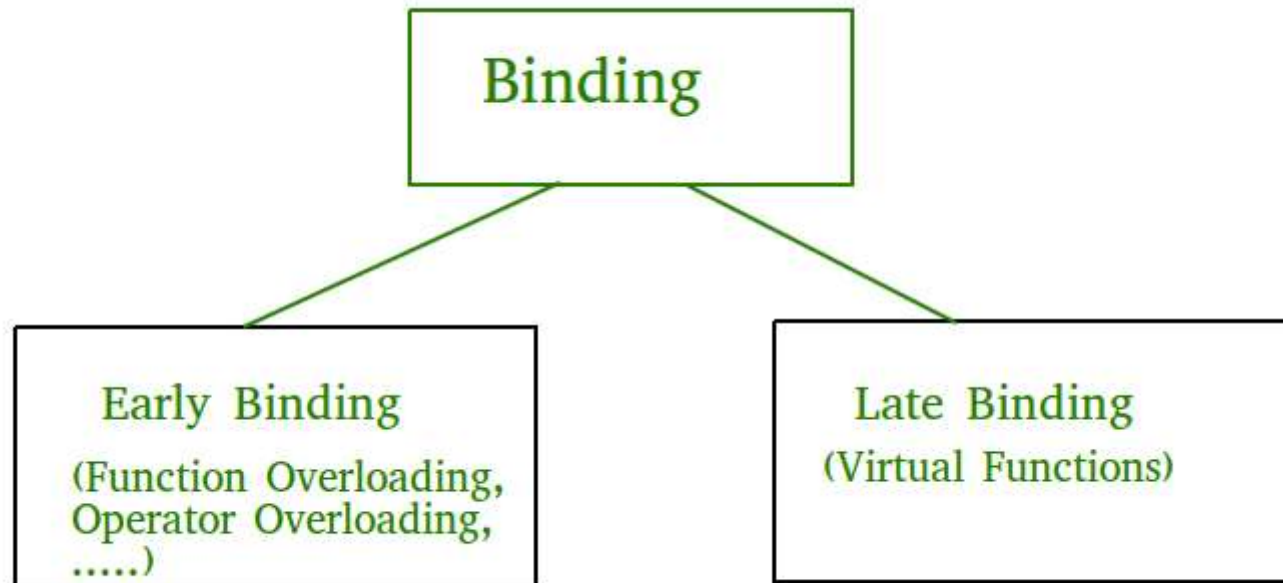
Ex 10: Virtual Function in C++

```
// Author : Engr. Abdul Rahman
//=====
// CPP program to illustrate working of Virtual Functions
#include<iostream>
using namespace std;
class base {
public:
    void fun_1() { cout << "base-1\n"; }
    virtual void fun_2() { cout << "base-2\n"; }
    virtual void fun_3() { cout << "base-3\n"; }
    virtual void fun_4() { cout << "base-4\n"; }
};
class derived : public base {
public:
    void fun_1() { cout << "derived-1\n"; }
    void fun_2() { cout << "derived-2\n"; }
    void fun_4(int x) { cout << "derived-4\n"; }
};
int main() {
    base *p;
    derived obj1;
    p = &obj1;
    p->fun_1(); // Early binding because fun1() is non-virtual in base
    p->fun_2(); // Late binding
    p->fun_3(); // Late binding
    p->fun_4(); // Late binding
    // Early binding but this function call is illegal (produces error)
    // because pointer is of base type and function is of derived class
    //p->fun_4(5);
}
```

Output:
base-1
derived-2
base-3
base-4

Early /Late binding in C++

- Binding refers to the process of converting identifiers (such as variable and performance names) into addresses. Binding is done for each variable and functions. For functions, it means that matching the call with the right function definition by the compiler. It takes place either at compile time or at runtime.



Early /Late binding in C++

- **Early Binding (compile-time polymorphism)** As the name indicates, compiler (or linker) directly associate an address to the function call. It replaces the call with a machine language instruction that tells the mainframe to leap to the address of the function.

Pure Virtual Functions and Abstract Classes in C++

- Sometimes implementation of all function cannot be provided in a base class. Such a class is called abstract class. For example, let Shape be a base class. We cannot provide implementation of function draw() in Shape, but we know every derived class must have implementation of draw(). Similarly an Animal class doesn't have implementation of move() (assuming that all animals move), but all animals must know how to move. We cannot create objects of abstract classes.
- A pure virtual function (or abstract function) in C++ is a virtual function for which we don't have implementation, we only declare it. A pure virtual function is declared by assigning 0 in declaration.

Ex 11: Abstract Classes / Interfaces

```
// Author      : Engr. Abdul Rahman
//=====
// An abstract class
#include<iostream>
using namespace std;

class Base {
    int x;
public:
    virtual void fun() = 0;
    int getX() { return x; }
};

// This class inherits from Base and implements fun()
class Derived: public Base {
    int y;
public:
    void fun() { cout << "fun() called"; }
};

int main(void)
{
    Derived d;
    d.fun();
    // Base b;
    // error: cannot declare variable 'b' to be of abstract type 'Base'
    cout << "\n";
    Base *bp = new Derived();
    bp->fun(); // We can have pointers & references of abstract class type.
    return 0;
}
```

Output:
fun() called
fun() called

Rules of Abstract Class

- *A class is abstract if it has at least one pure virtual function.*
- *We cannot create the instance of abstract class.*
- *We can have pointers and references of abstract class type.*
- *If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.*
- *An abstract class can have constructors.*
- **Interface vs Abstract Classes:**
An interface does not have implementation of any of its methods, it can be considered as a collection of method declarations. In C++, an interface can be simulated by making all methods as pure virtual. In Java, there is a separate keyword for interface.