(15)

namespace

- is a declarative region (scope) that provides a scope to the identifiers (the names of types, functions, variables etc) inside it.
- Use used to organize code into logical groups and to prevent name collisions that can occur especially when our code base includes multiple libraries
- It provides a class like modularization without class like semantics
- Obliviates the use of file level scoping of C (file) static

Ex.

```
namespace myNamespace {
    int myData;
    void myFunc()
    { cout << "myFunc()" << endl; }
}
class myClass
{   int data;
public:
    myClass(int d) { data = d; }
    void disp()
    { cout << "disp()" << endl;
    }
};

int main()
{
```

```
myNamespace::myvar = 6;
```

```
cout << myNamespace::myvar << endl;
myNamespace::myfun();
myNamespace::myClass Obj(25);
Obj.Display();
}
```

Name hiding abs()

```
#include <cstdlib>
int abs(int)
{
    if(n<-128) return 0
    if(n>127) return 0
    if(n<0) return -n
    return n
}
int main()
{
    cout<<Obj(-203)<<endl;
    cout<<abs(-6)<<endl;
}
```

→ this abs will hide the abs defined in cstdlib header file.

Again using namespace

```
#include <cstdlib>
namespace myNs
{
    int abs(int n)
    {
        if(n<-128) return 0;
```

```
if(n>127) return 0
if(n<0) return -n
return n
}
}
int main()
{
cout<<myNs::abs(-20)<<endl;
cout<<myNs::abs(1000)<<endl;
cout<<abs(20)<<endl;
```

using both abs

namespace is used to resolve integration name collision.

Nested namespace

- A namespace may be nested

```
using namespace std;
int data=0;
namespace name1{
    int data=1;
    namespace name2{
        int data=2;
    }
}
```

```
cout<<data<<endl;
cout<<name1::data<<endl;
cout<<name1::name2::data<<endl;
```

(154) using keyword and its shortcut

using namespace std;

```
namespace name1{
    int v11 = 1;
    int v12 = 2;
}
namespace name2
{   int v21 = 3;
    int v22 = 4;
}
using namespace name1;
using namespace
using name2::v21;
int main()
{ cout<<v11<<endl;
  cout<<name1::v12<<endl;
  cout<<v21<<endl;
  cout<<v22<<endl;
}
```

using namespace name1; → all symbols of name1 will be available

using name2::v21; → only v21 symbol of name2 will be available

If symbol that is subsequently used it will try to check it that name space has that symbol.

compiler error

* using or using namespace hides some of the names.

Ex:

(1ST)

```
int data=0;
namespace name1{
    int data=1
}
int main()
{   using name1::data;
    cout<<data<<endl;
    cout<<name1::data<<endl;
    cout<<::data<<endl;
}
```

// hides global data

unknown namespace it refers global namespace.

* entire C++ standard lib are present in std.

namespaces are open.
new declaration can be added

```
namespace open
{ int x=30;
}
namespace open
{   int y=40; }

int main()
{   using namespace open;
    x=y=20;
    cout<<x<<y<<endl;
}
```

O/P: 20 20

* A namespace can be ~~unnamed~~ unnamed but we cannot access it from outside. we use it to segregate symbol.

namespace vs class

namespace	class
every namespace is not a class	every class in turns defines a namespace.
can be re-opened and new declaration can be added	~~has multiple instance~~ Can not be re-opened
No instance of namespace can be created	Has multiple instance
Using declaration allowed	No using declaration.
~~Anonymous~~ It can be un named.	~~Unnamed~~ Unnamed class not allowed

* exp. scope
  - Block scope
  - Function scope
  - class scope
  - Namespace scope
  - file scope
  - global scope

(152)

Inheritance

ISA relationship

* We often find one Object is a specialization/generalization of another.
* OOAD (Object oriented analysis and Design) models this using ISA relationship.
* C++ model ISA relationship by inheritance of classes.

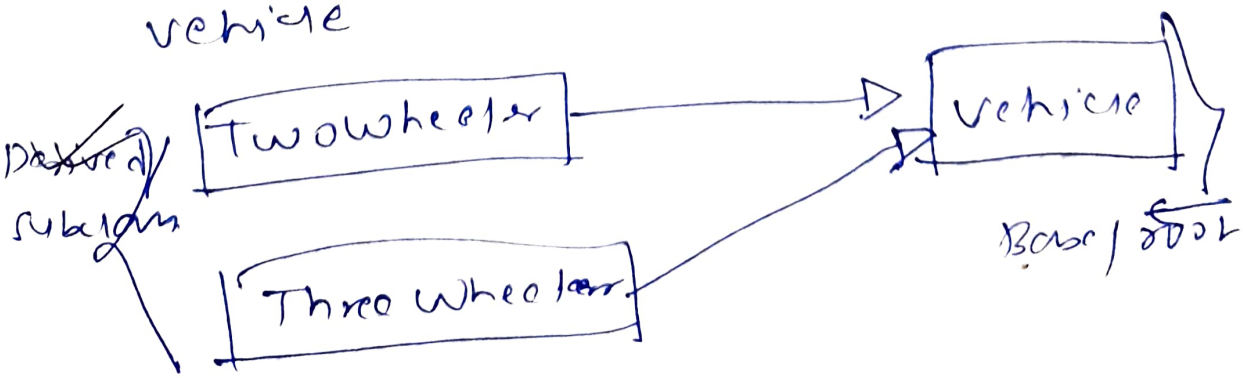
ex.

Rose is A flower

(i) Rose has properties of flowers
(ii) Rose has some additional properties
(iii) Rose is a specialization of flower
(iv) flower is a generalization of Rose

RedRose ISA a Rose

TwoWheeler ISA vehicle. Three wheeler is a vehicle



Manager is a Employee

158

* One generalization concept and multiple specialized concept

* manager can perform all of Employee + some additional which Employee can not do

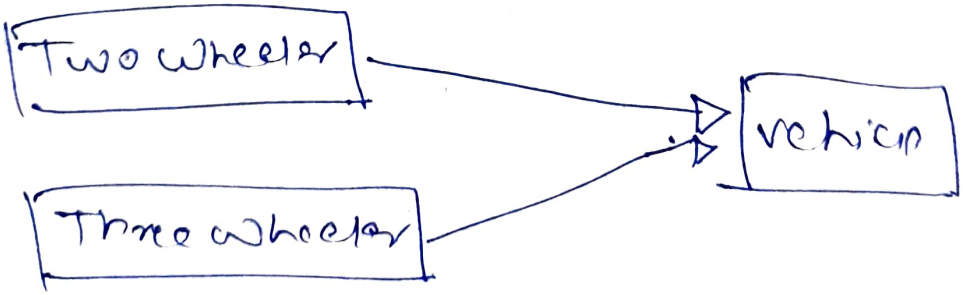
Single inheritance

① manager IS A Employee.

class Employee: // Base class.

class manager: public Employee // Derived class = manager.

② Hybrid inheritance: ⊙ concept where one Base class has more than one specialize concept



class vehicle;

Class TwoWheeler: public vehicle;

Class ThreeWheeler: public vehicle;

(159) multilevel inheritance

RedRose ISA a Rose. Rose ISA Flower

class Flower;

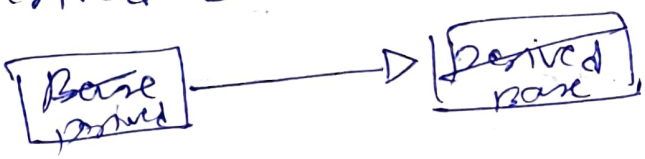
class Rose : public Flower;

class RedRose : public Rose;

* Landline phone smartphone.

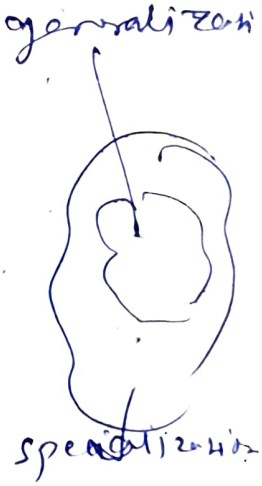
Inheritance in C++: Semantics

* Derived ISA Base



class Base;

class Derived : public Base.



* Derived ISA a Base.

* Data members

* Derived class inherits all Data members of Base class

* Derived class may add derived member of its own

* Member functions.

* Derived class inherits all member functions of Base class.

* Derived class may override a member

function of Base class by redefining it with the same signature.

* Derived class may overload a member fn of Base class by redefining it with the same name but diff signature

Access specification

* Derived class cannot access private member of Base class
* Derived class can access protected member of Base class

Construction Destruction

* A construction of the derived class must first call a constructor of the Base class to construct Base class instance of the Derived class
* The destructor of the derived class must call the destructor of the Base class to destruct the Base class instance of the Derived class.

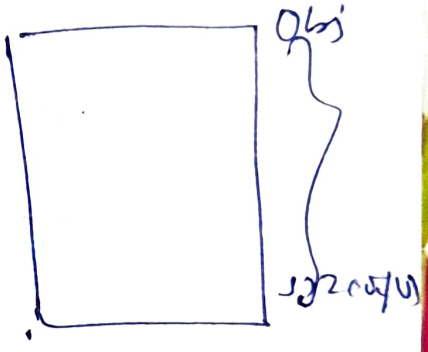
(161) Data members and Object layout

Object layout: Basically organization of data members within an object or class in the way that they get organized in memory.

Two Basically properties of layout

(i) All data members will be allocated space in the object in memory in a contiguous manner.

```
class B
{   int dataB
    public:
        int data2B
};
```

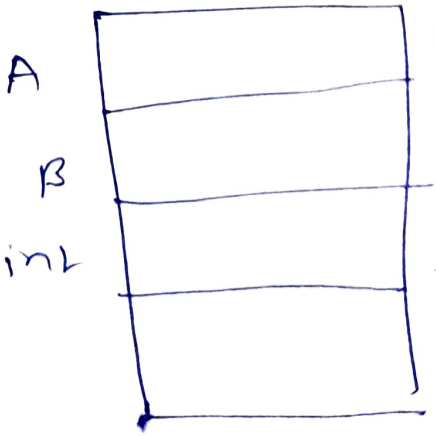


(ii) Order: It will come in the order in which they have been put down in the declaration

class

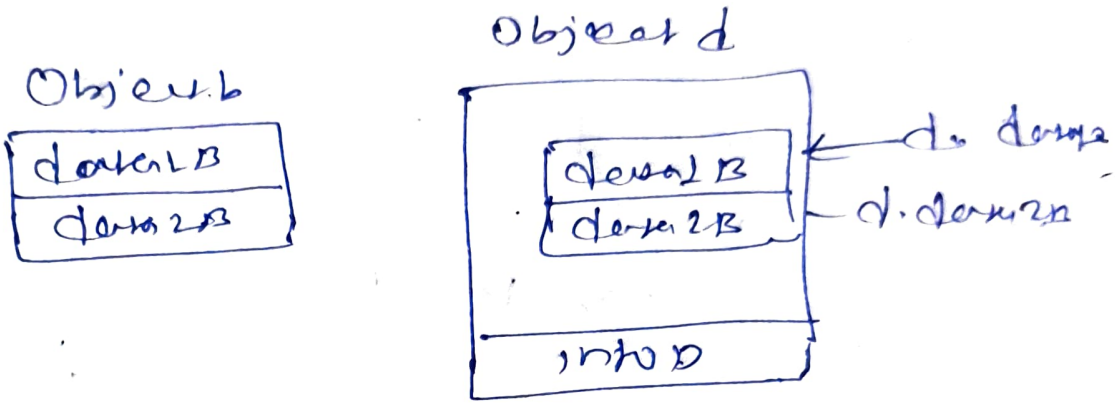
```
class A;
class B;
class C
{   A a;
    B b;
    int i;
};
```

Composed Concept



Here this object of this classes could be part of layout of another class

```
class D: public B
    // inherits B::: data1B
    // inherits B::: data2B
    int intoD // Add D::: intoD
public:
};
B b
D d
```



Since D is a B. Hence B D also. Gets a component of B But that component is unique. When I inherits we have only unique component coming in derived class Object from Base class Object

Since D is a B it will have an explicit instance of B Object as a part of D Object

(163) Data members and object layout

* Derived ISA Base (public inheritance)

* Data members

  * ~~Data member~~

    Derived class inherits all data member of Base class (true in every case of inheritance)

  * Derived class may add data members of its own.

Object layout

* Derived class layout contains or its - form of the Base class

* Derived class layout will have data members of its own.

* C++ does not guarantee the relative position of the Base class inside and derived class members

Member functions overrides and overloads

* Derived ISA Base. associate with the member function (only those fn of Base class which have this pointer)

  * Derived class inherits all member functions of Base

  * Derived class may override a member fn of base class by redefining it with the same signature

  * Derived class may overload a

member fn of Base class by redefining it with the same ~~signature~~ name but with diff signature

* Derived class may add any new member fn.

Static member fn

* Derived class doesnot inherits the static member fn of Base class

* Friend Function

* Derived class doesnot inherit the friend fn of Base class

Overrides and Overloads

I

we are trying to attach ~~to~~ two ~~different~~ diff fn to ~~same~~ to fn by the same name ordinarily by the same signature. and fn must belong to two diff classes which are related by generalization/specialization inheritance.

Overload it necessarily require that

(i) Both fn (overloaded) must belong to the same class either directly or by inheritance

(ii) They must differ in the signature they have.

ex.

```
class B {
public:
    void f(int i);
    void g(int i);
};

class D : public B
{ public:
    // inherits B::f(int)
    // inherits B::g(int)
};

B b
D d
b.f(1)  // calls B::f(int)
b.g(1)  // calls B::g(int)
d.f(3)  // calls B::f(int)
d.g(4)  // calls B::g(int)
```

* Derived class automatically inherits from of Base class

ex. 2

```
class D: public B
{ public:
    // inherits B::f(int)
    void f(int) // Overrides B::f(int)
    void f(string &) Overloads B::f(int)
    // inherits B::g(int)
    void h(int) // Adds D::h(int)
};
```

B b

D d

b.f()

b.g()

d.f(3) // calls D::f(int)

d.g(y) // calls B::g(int)

~~d.rd(Hr~~

d.f("red") // calls D::f(string)

d.h(5)

Four possible combinations

(I) ~~Base~~ Derived class simply inherits

(II) Overrides

(III) Overloads

(IV) Add a new fn

Access member of Base class: protected Access

* Derived IS A Base

* Access specification

~~*~~ Derived class cannot access private member of Base class

* Derived class can access public member of Base class.

protected access specification:

* A new protected access specifier is introduced in Base class to allow

(162) member (data or fn) to be accessed only in derived class but not to any other global or outside of class.

* Derived class can access protected members of Base class

* No other class or global fn can access protected members of Base class

* A protected member in Base class is like public in derived class

* A protected member in Base class is like private in other class or global fn.

* protected member is for Base class and also is for the class with whom Base class has very close relationship of generalization / specialization.

e.g.

```
class B
{ private:  // Inaccessible to child
    int data; //          to others

  public:
    //...
    void print()
    { cout << "data= " << data <<
                          endl;
    }
};
```

```
class D:: public B
{   int into;
public:
    //...
    void print()
    {//cout<<"data "<< data <<endl;
     cout<<"into: "<<into<<endl;
    }
};
```

Inaccessible → solution make data protected in B.

```
B b(0)
D d(1,2)
    b.data = 5 //inaccessible here
    b.print
    d.print
}
```

solution

```
class B
{   protected:          //Accessible to child
    int data;           //Inaccessible to others
public:
    void print()
    {  cout<<"data="<<data<<
       endl;
    }
};
```

```
class D : public B
{
    int info;
public:
    void print()
    {
        cout << "data = " << data <<
                                endl;
        cout << "info = " << info <<
                                endl;
    }
};
```

Accessible since data is protected in B.

```
B b[0];
b[0].data
b[0].info
```

```
b.data = 5 //
```

Inaccessible to others

```
b.print();
d.print();
```

Ex. 3 Friending in B.

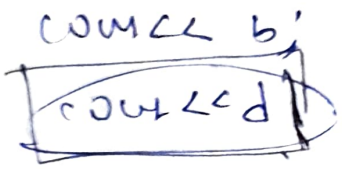
```
class B
{
    protected: int data;
public:
    friend ostream & operator << (ostream &os,
                                  const B &b)
    {
        os << b.data << endl;
        return os;
    }
};
```

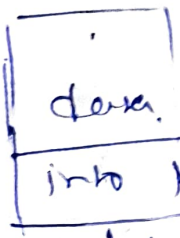
class D: public B
{ int into;

public:
friend fn is not inherited in child class.
};

B b(0);
D d(1,2);

O/P: 0
1





B Object memory
X ignored by compiler

* Here implicit casting is ~~having~~ happening
system will ignore into

Ex-4 class B
{ protected: int data; and pass a B object
public:
friend ostream & operator<<
(ostream &os, const B &b)
{ os << b.data << endl;
return os
}
};
class D : public B
{ friend ostream & operator <<
(ostream &os, const D &d)
{ ~~cout~~
os << d.data << endl;
os << d.into << endl;
} }

Here friend fn is overloaded.

```
B b(0)
D d(0,2)
cout<<b;
cout<<d
```

O/P: 0
1, 2

Constructor and Destructor in inheritance.

* Derived IS A Base

* constructor and destructor
  * Derived class inherits the constructors and destructors of Base class (but in a diff sequence)

A constructor of a class is fixed name fn. if we inherits Base class constructor in derived class then certainly it cannot be a constructor. Base class constructor becomes kind of member fn in derived class. similarly a Base class destructor becomes kind of member fn in b derived class.

Derived class cannot Override or overload a constructor or destructor of Base class

* Construction and destruction

A constructor of the derived class must first call a constructor of Base class to construct the Base class instance of the derived class.

(74) * The destructor of the derived class must call the destructor of Base class to destroy the base class instance of the derived class.

* Any derived class object has Base class instance.

~~Class~~ ex.

```
Class B
{   protected:
        int data;
    public:
        B(int d=0) : data(d)
        { cout << "Base constr" << data << endl;
        ~B()
        { cout << "Base destr" << data << endl;
};
```

```
Class D: public B
{   int info;
    public:
        D(int d, int i): B(d), info(i)
        { cout << "Derived parameterized constr" << data << info << endl;
        }
```

explicit call to Base class constructor.

~~D(int i): info(i)~~

~~{ cout << "Derived~~

(173)

* we cannot proceed with the constructor of derived class object until we have finished with the construction of Base class Object.

→ Default constructor of Base class

```
D(int i): info(i)
{
    cout << " Default constr: " << data <<
     info << endl;
}
~D()
{ cout << " Derived destr: " <<
    data << info << endl;
};
```

→ Here destructor of B will be called.

* If we do not call Base class constructor compiler will provide a free invocation of Base class constructor

```
B b(5);
D d1(1,2); // explicit constr of Base
D d2(3); // Default constructor Base
```

O/p:

Base constr data: 5

" " data: 1

Derived parameterized constr: data=1, info:2

Base constr: data: 0

Derived Default constr: data=0, info=3

(174) Derived destr: data = 0, info = 3

Base destr data = 0

Derived destr: data = 1, info = 2

Base destr: data = 1

Base destr: 0

* Lifetime of Base Object starts before lifetime of derived starts.

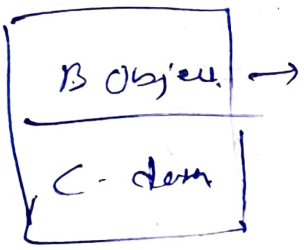
* First constructing Base class instance from derived member of derived class and on wrapping up first destructing the non Base Data member of derived class and then destructing Base class part.

Ex:

```
class B
{  public:
     B() {cout<<"B"<<endl;}
    ~B() {cout<<"~B"<<endl;}
};
class C
{  public:
     C() {cout<<"C"<<endl;}
    ~C() {cout<<"~C"<<endl;}
};
class D : public B
{  : pub C data;
   public:
     D() {cout<<"D"<<endl;}
    ~D() {cout<<"~D"<<endl;}
};
```

D d

o/p:



o/p:

B
C
D
~D
~C
~B

private inheritance:

- It does not mean generalization/specialization
- private inheritance means nothing during software design, only during S/W implementation.