PES UNIVERSITY

Department of Computer Science and Engineering

PES UNIVERSITY

UE19CS208B: Programming with C++

UNIT 4

Ms. Kusuma K V

Inheritance: Programming technique for defining a new class (known as a derived class, super class, parent class) in terms of an existing class (known as the base class, sub class, child class). The derived class inherits the members of the base class.

- We often find that one object is a specialization / generalization of another.
- Object Oriented Analysis and Design (OOAD) models this using **ISA** relationship.
- C++ models **ISA** relationship by **Inheritance of classes.**

Classes related by inheritance form a hierarchy. Typically there is a base class at the root of the hierarchy, from which the other classes inherit, directly or indirectly. These inheriting classes are known as derived classes. The base class defines those members that are common to the types in the hierarchy. Each derived class defines those members that are specific to the derived class itself.

Inheritance in C++: Semantics

- Derived ISA Base
- **Data Members**
  - Derived class inherits all data members of Base class
  - Derived class may add data members 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 function of Base class by redefining it with the same name; but different signature
- **Construction-Destruction**
  - A constructor of the Derived class must first call a constructor of the Base class to construct the 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
- **Access Specification**
  - Derived class cannot access private members of Base class
  - Derived class can access protected, and public members of Base class

The syntax for derivation is as follows:

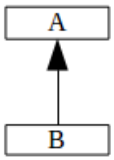
```
class <name of derived class> : <access specifier><name of base class>
{
  /*definition of base class*/
}
```

Eg: Suppose a class **A** already exists. Then, a new class (specialized) **B** can be derived from class (generalized) A as follows:

```
class B:public A
{
  /*new features of class B*/
}
```

A derived class must specify from which class(es) it inherits. It does so in its class derivation list, which is a colon followed by a comma-separated list of names of previously defined classes. Each base class name may be preceded by an optional access specifier, which is one of public, protected, or private. During derivation , if access specifier is not specified in class derivation list, the default derivation is private (for class and public for struct).

Diagrammatic representation of inheritance (B **is a** A) (A is Base class, B is derived class).



Inheritance in C++: Data Members and Object Layout

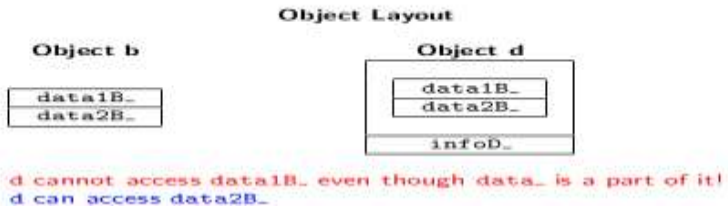
- Derived ISA Base
- Data Members
  - Derived class inherits all data members of Base class
  - Derived class may add data members of its own
- Object Layout
  - Derived class layout contains an instance of the Base class
  - Further, Derived class layout will have data members of its own
  - C++ does not guarantee the relative position of the Baseclass instance and Derived class members

```
class B { // Base Class
    int data1B_;
public:
    int data2B_;
    // ...
};

class D: public B { // Derived Class
    // Inherits B::data1B_
    // Inherits B::data2B_
    int infoD_; // Adds D::infoD_
public:
    // ...
};

B b;

D d;
```



//datamembers.cpp

```
//This program demonstrates the accessibility of base class and derived class data members
#include<iostream>
using namespace std;
class B
{
  int mb1;
public :
  int mb2;
  B(int b1=0,int b2=0)    //ctor of B with default parameters
  {
   mb1=b1;
   mb2=b2;
  }
};

class D: public B  //D derives from B in public mode
{
  int md1;
public:
  int md2;
  //ctor of D with default parameters, explicitly calls base class ctor: B(b1,b2)
  D(int b1=0,int b2=0,int d1=0,int d2=0):B(b1,b2)
  {
   md1=d1;
   md2=d2;
  }
```

```
};
int main()
{
  B b;
  D d;

// cout<<b.mb1<<endl;//B:mb1 is not accessible to Derived class D, as well as main because mb1 is private
  cout<<b.mb2<<endl;
// cout<<d.md1<<endl;  //D:md1 is not accessible to main, because md1 is private
  cout<<d.md2<<endl;
// cout<<d.mb1<<endl;//B:mb1 is not accessible to Derived class D, as well as main because mb1 is private
  cout<<d.mb2<<endl;

  return 0;
}
/* Output:
0
0
0
*/
```

Inheritance in C++: Member Functions – Overrides and Overloads

- Derived ISA Base
- 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 function of Base class by redefining it with the same name; but different signature
  - Derived class may add new member functions
- Static Member Functions
  - Derived class does not inherit the static member functions of Base class
- Friend Functions
  - Derived class does not inherit the friend functions of Base

//memberFn.cpp

```
//This program demonstrates that all member functions of base class are inherited in derived class
#include<iostream>
using namespace std;

class B
{
  public: void x(){cout<<"I'm x of B"<<endl;}
          void y(){cout<<"I'm y of B"<<endl;}
};

class D: public B
{
  //inherits x() and y() of B
  //adds new z() in D
  public: void z(){cout<<"I'm new to D, not inherited from B"<<endl;}
};

int main()
{
  B b;
  D d;

  b.x();
  b.y();

  d.x();          //calls the inherited x
  d.y();          //calls the inherited y
  d.z();          //calls z

  return 0;
}
/* Output:
I'm x of B
I'm y of B
I'm x of B
I'm y of B
```

```
I'm new to D, not inherited from B
*/
```

Derived classes inherit the members of their base class. However, a derived class needs to be able to provide its own definition for operations. In such cases, the derived class needs to override the definition it inherits from the base class, by providing its own definition. Also it may overload the methods of base class.

//ovrldOverrd.cpp

```
//This program demonstrates overloading and overriding of methods in derived class
#include<iostream>
using namespace std;

class B
{
  public: void x(){cout<<"I'm x of B"<<endl;}
          void y(){cout<<"I'm y of B"<<endl;}

};

class D: public B
{
  public: void x(){cout<<"I'm Overridden x of D"<<endl;}              //Overrides B::x()
          void y(int a){cout<<"I'm Overloaded y of D"<<", a="<<a<<endl;}  //Overloads B::y()
          void z(){cout<<"I'm new to D, not inherited from B"<<endl;}
};

int main()
{
  B b;
  D d;

  b.x();
  b.y();

  d.x();                          //static polymorphism
  d.y(2);                         //static polymorphism
  d.z();

  return 0;
}
/* Output:
```

```
I'm x of B
I'm y of B
I'm Overridden x of D
I'm Overloaded y of D, a=2
I'm new to D, not inherited from B
*/
```

Note: It is a good practice to make the methods to be overridden in derived class as virtual in base class. So that it supports dynamic polymorphism. Discussed in virtual functions further in this notes.

Inheritance in C++: Constructor & Destructor

- Derived ISA Base
- Constructor-Destructor
  - Derived class inherits the Constructors and Destructor of Base class (but in a different semantics)
  - Derived class cannot override or overload a Constructor or the Destructor of Base class
- Construction-Destruction
  - A constructor of the Derived class must first call a constructor of the Base class to construct the 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

//ctorDtor1.cpp

```
//This program demonstrates the order in which ctor and dtor are called
#include<iostream>
using namespace std;

class B
{
  int m_b;
public:
  B(int);
  void show();
  ~B();
};

B::B(int b)
{
  cout<<"Base class ctor"<<this<<endl;
```

```
  m_b=b;
}
void B::show()
{
  cout<<"Base class data="<<m_b<<endl;
}

B::~B()
{
  cout<<"Base class dtor"<<this<<endl;
}

class D:public B
{
  int m_d;
public:
  D(int,int);
  void show();
  ~D();
};

D::D(int b,int d):B(b)     //Note the call to Base class ctor in initialization list of derived class ctor
{
  cout<<"Derived class ctor"<<this<<endl;
  m_d=d;
}

void D::show()
{
  B::show();
  cout<<"Derived class data="<<m_d<<endl;
}

D::~D()
{
  cout<<"Derived class dtor"<<this<<endl;
}

int main()
```

```
{
  B b(10);
  b.show();
  D d(20,30);
  d.show();
  return 0;
}
/*Output:
Base class ctor0x7ffc6379df10
Base class data=10
Base class ctor0x7ffc6379df20
Derived class ctor0x7ffc6379df20
Base class data=20
Derived class data=30
Derived class dtor0x7ffc6379df20
Base class dtor0x7ffc6379df20
Base class dtor0x7ffc6379df10
*/
```

Note: Constructors are invoked in the order of inheritance. i.e., First, base class constructor then member objects constructor, then derived class constructor is invoked. See ctorDtor2.cpp.

//ctorDtor2.cpp

```
//This program demonstrates the order in which ctor and dtor are called
//This program also demonstrates multi level inheritance
#include<iostream>
using namespace std;
class A
{
public:
  A(){ cout<<"A class ctor"<<endl; }
  ~A(){ cout<<"A class dtor"<<endl; }
};
class B:public A
{
public:
  B(){ cout<<"B class ctor"<<endl; }
  ~B(){    cout<<"B class dtor"<<endl; }
};
```

```
class C:public B
{
  B bObj;    //bObj is an object of class B, so first A class ctor is invoked then B class ctor

public:
  C(){ cout<<"C class ctor"<<endl; }
  ~C(){ cout<<"C class dtor"<<endl; }
};
int main()
{
  C c;
  return 0;
}
/*Output:
A class ctor
B class ctor
A class ctor
B class ctor
C class ctor
C class dtor
B class dtor
A class dtor
B class dtor
A class dtor
*/
```

Copy Constructors and Assignment Operator in Derived Classes

When defining a derived class, you need to be careful about copy constructors and operator= . We saw that the initialization phase of a derived-class constructor initializes the base-class part(s) of a derived object as well as initializes its own members. As a result, the copy and move constructors for a derived class must copy/move the members of its base part as well as the members in the derived. Similarly, a derived-class assignment operator must assign the members in the base part of the derived object. i.e., When a derived class defines a copy or move operation, that operation is responsible for copying or moving the entire object, including base-class members.

If your derived class does not have any special data (pointers, usually) that require a nondefault copy constructor or operator= , you don't need to have one, regardless of whether or not the base class has one. If your derived class omits the copy constructor or

operator= , a default copy constructor or operator= will be provided for the data members specified in the derived class and the base class copy constructor or operator= will be used for the data members specified in the base class.

On the other hand, if you do specify a copy constructor in the derived class, you need to explicitly chain to the parent copy constructor, as shown in the following code. If you do not do this, the **default constructor** (**not the copy constructor!**) will be used for the parent portion of the object.

Defining a derived copy constructor

When we define a copy or move constructor for a derived class, we ordinarily use the corresponding base-class constructor to initialize the base part of the object. The initializer **Base(d)** passes a Derived object to a base-class constructor. **Base(d)** will (ordinarily) match the Base copy constructor. The Derived object, d, will be bound to the Base& parameter in that constructor. The Base copy constructor will copy the base part of d into the object that is being created.

```
class Base
{
public:
        Base();                         //Base Class ctor prototype
        Base(const Base& b);            //Base class copy ctor prototype
};
class Derived : public Base
{
public:
        Derived();                      //Derived Class ctor prototype
        Derived(const Derived& d);      //Derived class copy ctor prototype
};
Derived::Derived(const Derived& d) : Base(d)//Copy the base members
  /*Initializers for members of Derived*/  {/*.....*/}
```

Had the initializer for the base class been omitted, the Base default constructor would be used to initialize the base part of a D object. Assuming D's constructor copies the derived members from d, this newly constructed object would be oddly configured: Its Base members would hold default values, while its D members would be copies of the data from another object.

```
// probably incorrect definition of the D copy constructor
// base-class part is default initialized, not copied
D(const D& d) /* member initializers, but no base-class initializer  */
{ /* ...  */ }
```

Defining a derived assignment operator

Like the copy and move constructors, a derived-class assignment operator , must assign its base part explicitly. The following code shows how to call the parent's assignment operator from the derived class:

```
Derived& Derived::operator=(const Derived& rhs)
{
        if (&rhs == this)                       //handle self assignment
        {
                return *this;
        }

        Base::operator=(rhs);                   // Calls parent's operator=
        // Do necessary assignments for derived class
        return *this;
}
```

This operator starts by checking for self assignment. If self assignment then returns the current object itself. If not self assignment then explicitly calls the base-class assignment operator to assign the members of the base part of the derived object. The base-class operator will (presumably) correctly handle self-assignment and, if appropriate, will free the old value in the base part of the left-hand operand and assign the new values from rhs. Once that operator finishes, we continue doing whatever is needed to assign the members in the derived class.

Access Specifiers

A derived class inherits the members defined in its base class. However, the member functions in a derived class may not necessarily access the members that are inherited from the base class. Like any other code that uses the base class, a derived class may access the public members of its base class but may not access the private members. However, sometimes a base class has members that it wants to let its derived classes use while still prohibiting access to those same members by other users. We specify such members after a protected access specifier.

- Derived Is A Base
- Access Specification
  - Derived class cannot access private members of Base class
  - Derived class can access public members of Base class
- Protected Access Specification
  - A new protected access specification is introduced for Base class
  - Derived class can access protected members of Base class
  - No other class or global function can access protected members of Base class

//protected.cpp

```
//Demonstrates the accessibility of different data members in derived class when publicly derived
//Protected members are accessible only in its own class and its child class
#include<iostream>
using namespace std;
class B
{
  int b1;
  public : int b2;
   B(int=0,int=0,int=0);
   void showB(){cout<<"B::"<<b1<<""<<b2<<""<<b3<<endl;}
  protected: int b3;
};
B::B(int p,int q,int r)
{
  b1=p;
  b2=q;
  b3=r;
}
class D: public B
{
  //b1(inherited) is not accessible directly in D. Because it is private in B.
  //b2(inherited) accessible in D. Because it is public in B.
  //b3(inherited) accessible in D. Because it is protected in B.
  int d1;
  public: int d2;
   D(int p,int q,int r,int s,int t):B(p,q,r){d1=s; d2=t;}     //ctor
   void showD()  {
            cout<<"Base class instance of D is ";
            B::showB();
            cout<<"Specific members of D are ";
            cout<<"D::"<<d1<<""<<d2<<endl;
   }
   void showProtectedB() {
   cout<<"To show, protected members of base class are accessible in derived class"<<endl;
   cout<<"I'm protected member of base class B::b3, accessed in derived class D:"<<b3<<endl;
   }
};
```

```
int main()
{
  B b(1,2,3);
  D d(10,20,30,40,50);
//b1 and d1 are not accessible because they are private members
//b3 is not accessible because it is protected member
// cout<<b.b1<<endl;
// cout<<b.b2<<endl;   //accessible, because public in B
// cout<<b.b3<<endl;
// cout<<d.d1<<endl;
// cout<<d.d2<<endl;   //accessible, because public in D
// cout<<d.b1<<endl;
// cout<<d.b2<<endl;   //accessible, because public in D
// cout<<d.b3<<endl;
  cout<<"I'm object B"<<endl;
  b.showB();

  cout<<endl<<"I'm object D"<<endl;
  d.showD();

  cout<<endl;
  d.showProtectedB();

  return 0;
}
/* Output:
I'm object B
B::1 2 3

I'm object D
Base class instance of D is B::10 20 30
Specific members of D are D::40 50

To show, protected members of base class are accessible in derived class
I'm protected member of base class B::b3, accessed in derived class D:30
*/
```

Visibility across Access and Inheritance

Visibility Matrix

Visibility \ Inheritance	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	private	private	private

Virtual Functions and Polymorphism, Function Overriding

Basics on upcasting and downcasting.

Upcast: Base class pointer holding derived class object (allowed).

Downcast: Derived class pointer holding base class object (Error. If done forecfully via cast then no compilation error, but risky)

Note: sliced down: Slicing happens when an object of derived type is used to initialize or assign an object of the base type(by valuei.e., not by pointer or reference). The derived portion of the object is "sliced down," leaving only the base portion, which is assigned to the base. Eg: A a=b; (where a is base class object and b is derived class object).

//upCast_StaticPoly.cpp

```
//pgm demonstrates the different ways to access data members and member functions of a class
//1)via object 2)via pointer 3)via reference
//The pgm also demonstrates static polymorphism (none of the member functions are virtual)
#include<iostream>
using namespace std;

class A
{
  int ma1;
  int ma2;
public:
  A(int a1,int a2){ma1=a1;ma2=a2;}
  void display(){cout<<ma1<<""<<ma2<<endl;}
};
```

```
class B:public A
{
  int mb1;
public: int mb2;

  B(int a1,int a2,int b1,int b2):A(a1,a2){mb1=b1;mb2=b2;}
  void display(){A::display();cout<<mb1<<""<<mb2<<endl;}
};

int main()
{
  A a(2,3);
  cout<<"Object a"<<endl;
  a.display();

  B b(4,5,6,7);
  cout<<"Object b"<<endl;
  b.display();

  A* pA=&a;
  B* pB=&b;

  cout<<endl;
  cout<<"Object a via ptr"<<endl;
  pA->display();

  cout<<"Object b via ptr"<<endl;
  pB->display();

  A& rA=a;
  B& rB=b;
  cout<<endl;
  cout<<"Object a via reference"<<endl;
  rA.display();

  cout<<"Object b via reference"<<endl;
  rB.display();
```

```
//Below line is an error, because we are downcasting i.e., storing a base class object in a derived class pointer
//          pB=&a;

//Below line is OK, because we are upcasting i.e., storing a derived class object in a base class pointer
  pA=&b;

  cout<<endl;
  cout<<"Static Polymorphism, even though pA holds derived class object, it calls A::display() ";
  cout<<"because display() is not virtual in base class. ";
  cout<<"Binding happens at compile time, function is called based on pointer type"<<endl;
  pA->display();          //calls A::display(), because static binding happens

//Below line is an error, because we are downcasting i.e., storing a base class object in a derived class reference
//          rB=a;

//Below line is OK, because we are upcasting i.e., storing a derived class object in a base class pointer
  A& refA=b;

  cout<<endl;
  cout<<"Static Polymorphism, even though refA holds derived class object, it calls A::display() ";
  cout<<"because display() is not virtual in base class. ";
  cout<<"Binding happens at compile time, function is called based on reference type"<<endl;
  refA.display();          //calls A::display(), because static binding happens

  return 0;
}
/*Output:
Object a
2 3
Object b
4 5
6 7
Object a via ptr
```

2 3
Object b via ptr
4 5
6 7

Object a via reference
2 3
Object b via reference
4 5
6 7

Static Polymorphism, even though pA holds derived class object, it calls A::display() because display() is not virtual in base class. Binding happens at compile time, function is called based on pointer type
4 5

Static Polymorphism, even though refA holds derived class object, it calls A::display() because display() is not virtual in base class. Binding happens at compile time, function is called based on reference type
4 5
*/

Virtual functionis a member function that defines type-specific behavior. Calls to a virtual made through a reference or pointer are resolved at run time, based on the type of the object to which the reference or pointer is bound (Polymorphism).

- A base class specifies that a member function should be dynamically bound by preceding its declaration with the keyword **virtual**.
- Any nonstatic member function, other than a constructor, may be virtual.
- The virtual keyword appears only on the declaration inside the class and may not be used on a function definition that appears outside the class body.
- A function that is declared as virtual in the base class is implicitly virtual in the derived classes as well.

Because the decision as to which version to run depends on the type of the argument, that decision can't be made until run time. Therefore, dynamic binding is sometimes known as run-time binding.

Override: Virtual function defined in a derived class that has the same parameter list as a virtual in a base class overrides the base-class definition.

- A derived-class function that overrides an inherited virtual function must have exactly the same parameter type(s) as the base-class function that it overrides.

- With one exception, the return type of a virtual in the derived class also must match the return type of the function from the base class. The exception applies to virtuals that return a reference (or pointer) to types that are themselves related by inheritance. That is, if D is derived from B, then a base class virtual can return a B* and the version in the derived can return a D*. However, such return types require that the derived-to-base conversion from D to B is accessible.

Note: In C++, dynamic binding applies only to functions declared as virtual and called through a reference or pointer. Member functions that are not declared as virtual are resolved at compile time, not run time.

Note: A derived-class object contains a subobject corresponding to each of its base classes. Because every derived object contains a base part, we can convert a reference or pointer to a derived-class type to a reference or pointer to an accessible base class.

Note: calls to any function (virtual or not) on an object are also bound at compile time.

//dynamicPoly_Virtual_Override.cpp

```
//The pgm demonstrates dynamic polymorphism
//In C++, dynamic binding happens when a virtual function is called through a reference (or a pointer) to a base class.
//Virtual function defined in a derived class that has the same parameter list as a virtual in a base class overrides the base-class definition.
#include<iostream>
using namespace std;

class A
{
  int ma1;
  int ma2;
public:

  A(int a1,int a2){ma1=a1;ma2=a2;}
  virtual void display(){cout<<ma1<<""<<ma2<<endl;}                    //virtual function
};

class B:public A
{
  int mb1;
public: int mb2;
  B(int b1,int b2,A a):A(a){mb1=b1;mb2=b2;}
  void display(){A::display();cout<<mb1<<""<<mb2<<endl;}  //A::display() is overridden
};
```

```
int main()
{
  A a(2,3);
  cout<<"Object a"<<endl;
  a.display();

  B b(4,5,a);
  cout<<"Object b"<<endl;
  b.display();

  A* pA=&a;
  B* pB=&b;

  cout<<endl;
  cout<<"Object a via ptr"<<endl;
  pA->display();

  cout<<"Object b via ptr"<<endl;
  pB->display();

  A& rA=a;
  B& rB=b;

  cout<<endl;
  cout<<"Object a via reference"<<endl;
  rA.display();

  cout<<"Object b via reference"<<endl;
  rB.display();

//Below line is an error, because we are downcasting i.e., storing a base class object in a derived class pointer
//        pB=&a;

//Below line is OK, because we are upcasting i.e., storing a derived class object in a base class pointer
  pA=&b;

  cout<<endl;
  cout<<"Dynamic Polymorphism, pA holds derived class object, it calls B::display() ";
```

```
  cout<<"because display() is virtual in base class, binding happens at run time. ";
  cout<<"Function is called depending on the object held by pointer at run time"<<endl;
  pA->display();          //calls B::display(), because dynamic binding happens

//Below line is an error, because we are downcasting i.e., storing a base class object in a derived class reference
//          rB=a;


//Below line is OK, because we are upcasting i.e., storing a derived class object in a base class pointer
  A& refA=b;

  cout<<endl;
  cout<<"Dynamic Polymorphism, refA holds derived class object, it calls B::display() ";
  cout<<"because display() is virtual in base class, binding happens at run time. ";
  cout<<"Function is called depending on the object held by reference at runtime"<<endl;
  refA.display();          //calls B::display(), because dynamic binding happens

  return 0;
}

/*
Output:
Object a
2 3
Object b
2 3
4 5

Object a via ptr
2 3
Object b via ptr
2 3
4 5

Object a via reference
2 3
Object b via reference
2 3
```

4 5

Dynamic Polymorphism, pA holds derived class object, it calls B::display() because display() is virtual in base class, binding happens at run time. Function is called depending on the object held by pointer at run time

2 3

4 5

Dynamic Polymorphism, refA holds derived class object, it calls B::display() because display() is virtual in base class, binding happens at run time. Function is called depending on the object held by reference at runtime

2 3

4 5

*/

final and override specifiers (C++ 11 features)

It is legal for a derived class to define a function with the same name as a virtual in its base class but with a different parameter list(function overload). The compiler considers such a function to be independent from the base-class function. In such cases, the derived version does not override the version in the base class. In practice, such declarations often are a mistake—the class author intended to override a virtual from the base class but made a mistake in specifying the parameter list.

Finding such bugs can be surprisingly hard. Under the new standard we can specify override on a virtual function in a derived class. Doing so makes our intention clear and (more importantly) enlists the compiler in finding such problems for us. The compiler will reject a program if a function marked override does not override an existing virtual function:

```
class B
{
virtual void f1(int) const;
virtual void f2();
void f3();
};
class D1 : public B
{
void f1(int) const override;          // ok: f1 matches f1 in the base
void f2(int) override;          // error: B has no f2(int) function
void f3() override;                   // error: f3 not virtual
void f4() override;                   // error: B doesn't have a function named f4
};
```

In D1, the override specifier on f1 is fine; both the base and derived versions of f1 are const members that take an int and return void. The version of f1 in D1 properly overrides the virtual that it inherits from B.

The declaration of f2 in D1 does not match the declaration of f2 in B—the version defined in B takes no arguments and the one defined in D1 takes an int. Because the declarations don't match, f2 in D1 doesn't override f2 from B; it is a new function that happens to have the same name. Because we said we intended this declaration to be an override and it isn't, the compiler will generate an error.

Because only a virtual function can be overridden, the compiler will also reject f3 in D1. That function is not virtual in B, so there is no function to override.

Similarly f4 is in error because B doesn't even have a function named f4.

We can also designate a function as **final**. Any attempt to override a function that has been defined as final will be flagged as an error:

```
class D2 : public B
{
// inherits f2() and f3() from B and overrides f1(int)
void f1(int) const final;           // subsequent classes can't override f1(int)
};
class D3 : public D2 {
void f2();                          // ok: overrides f2 inherited from the indirect base, B
void f1(int) const;                         // error: D2 declared f2 as final
};
```

Note: final and override specifiers appear after the parameter list (including any const or reference qualifiers) and after a trailing return.

//multiLvlInh.cpp

```
//Pgm demonstrates multi level inheritance and dynamic polymorphism
//Once a function is defined virtual, it remains virtual down the inheritance hierarchy
#include<iostream>
using namespace std;

class A
{
   int m_a1,m_a2;
public:
A(int a1,int a2)
  {
          m_a1=a1;
```

```
        m_a2=a2;
  }
  virtual void display(){cout<<"Base"<<endl;cout<<m_a1<<""<<m_a2<<endl;}
};

class B:public A
{
  int m_b1;

public:
  B(int a1,int a2,int b1):A(a1,a2)
  {
        m_b1=b1;
  }
  void display(){                              //display() is overridden
        A::display();
        cout<<"Derived"<<endl;cout<<m_b1<<endl;
  }
};

class C:public B
{
  public:C(int a1,int a2,int a3):B(a1,a2,a3){}
  //display is inherited, and not overridden here
};

int main()
{
  A a(2,3);
  B b(4,5,6);
  C c(7,8,9);


  B* pB=&c;
  pB->display();      //Base class pointer holding derived class object c. Calls inherited display() in class C

  cout<<endl;
  A* pA=&b;
```

```
  pA->display(); //Base class pointer holding derived class object b. Calls the overridden display() in class B
  return 0;
}
/* Output:
Base
7 8
Derived
9

Base
4 5
Derived
6
*/
```

VTBL and VPTR

To understand how method hiding is avoided, you need to know a bit more about what the virtual keyword actually does. When a class is compiled in C++, a binary object is created that contains all methods for the class. In the non- virtual case, the code to transfer control to theappropriate method is hard-coded directly where the method is called based on the compile-time type.

If the method is declared virtual , the correct implementation is called through the use of a special area of memory called the vtable, for “virtual table.” For each class that has one or more virtual methods there is a vtable, and every object of such a class contains a pointer to said vtable. This vtable contains pointers to the implementations of the virtual methods. In this way, when a method is called on an object, the pointer is followed into the vtable and the appropriate version of the method is executed based on the actual type of the object.

To better understand how vtables make overriding of methods possible, take the following Super and Sub classes as an example.

```
class Super
{
  public:
        virtual void func1() {}
        virtual void func2() {}
        void nonVirtualFunc() {}
};
class Sub : public Super
{
  public:
```

```
        virtual void func2() override {}
        void nonVirtualFunc() {}
};
```

For this example, assume that you have the following two instances:

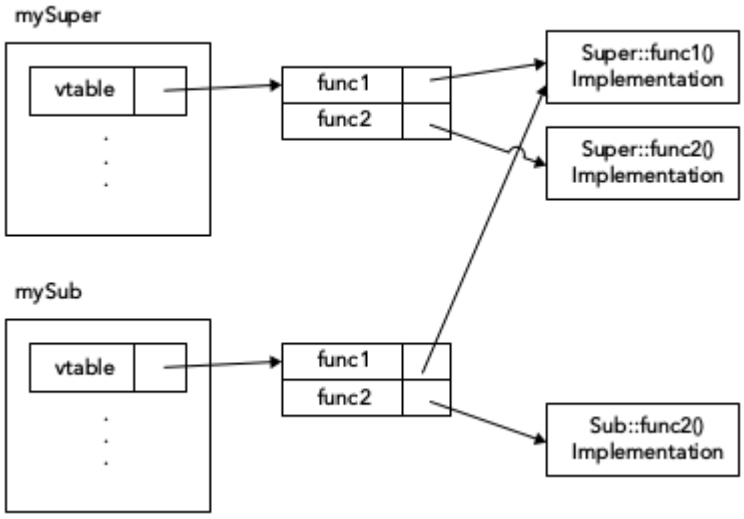
Super mySuper;

Sub mySub;

Below figure shows a high-level view of how the vtables for both instances look. The mySuper object contains a pointer (VPTR i.e., Virtual Pointer) to its vtable. This vtable has two entries, one for func1() and one for func2() . Those entries point to the implementations of Super::func1() and Super::func2() .

mySub also contains a pointer (VPTR) to its vtable which also has two entries, one for func1() and one for func2() . The func1() entry of the mySub vtable points to Super::func1() because Sub does not override func1() . On the other hand, the func2() entry of the mySub vtable points to Sub::func2() .

Note that both vtables do not contain any entry for the nonVirtualFunc() method because that method is not virtual.



Whenever a call is dispatched to a virtual function through a reference to an object or a pointer to an object, then the value of the VPTR of the object is obtained. Then the address of the called function from the corresponding VTBL is obtained. Finally, the function is called through the address thus obtained.

Pure Virtual Functions and Abstract Base Class (ABC)

Pure virtual functionis a virtual function declared in the class using **= 0** just before the semicolon. A pure virtual function need not be (but may be) defined. Classes with pure virtuals are **abstract classes**.

The = 0 may appear only on the declaration of a virtual function in the class. It is worth noting that we can provide a definition for a pure virtual. However, the function body must be defined outside the class. That is, we cannot provide a function body inside the class for a function that is = 0.

A class containing (or inheriting without overridding) a pure virtual function is an **abstract base class**. An abstract base class defines an interface for subsequent classes to override. If a derived class does not define its own version of an inherited pure virtual, then the derived class is abstract as well. We cannot create objects of a type that is an abstract class.

Note: A derived class constructor initializes its direct base class only.

//pureVirtual_ABC.cpp

```
//Program demonstrates pure virtual function (Virtual function equated to zero) and
//Abstract Base Class(Class which has a pure virtual function).
//Abstract Base Class is not instantiable.
//Pgm also demonstrates hierarchial inheritance (2 or more derived classes deriving from the same base class)
#include<iostream>
using namespace std;
class Shape
{
  public:virtual void area()=0;
};
//To show that pure virtual function may have a body. But it has to be defined outside the class.
//void Shape::area(){cout<<"Body"<<endl;}
class Rectangle:public Shape
{
  double ml,mb;
  public: Rectangle(double l=0,double b=0){ml=l;mb=b;}
          void area(){cout<<"Rectangle Area="<<ml*mb<<endl;}

};
class Circle:public Shape
{
  double pi;
  double mr;
  public: Circle(double r=0){mr=r; pi=3.142;}
          void area(){cout<<"Circle Area="<<pi*mr*mr<<endl;}
};
```

```
int main()
{
  //Below line is an error. BecauseShape is an Abstract Base Class(ABC) and hence not instantiable.
// Shape s;

  Rectangle r(2,3);
  r.area();

  Circle c(2);
  c.area();

//Calling Base class area by an object of derived class. Uncomment the body of Shape::area to not get linker error
//        r.Shape::area();          //uncomment this line to call base class area i.e., Shape::area

  return 0;
}
/* Output:
Rectangle Area=6
Circle Area=12.568
*/
```

Virtual Destructors

Non-virtual destructors may result in situations in which memory is not freed by object destruction. If a class is marked as final then destructor may be non- virtual.

For example, if a derived class uses memory that is dynamically allocated in the constructor and deleted in the destructor, it will never be freed if the destructor is never called.

As the following code shows, it is easy to “trick” the compiler into skipping the call to the destructor if it is non- virtual :

```
class Base
{
  public:
        Base() {}
        ~Base() {}
};
class Derived : public Base
{
  public:
        Derived() { mString = new char[30]; }
```

```
        ~Derived() { delete [] mString; }
  private:
        char* mString;
};
int main()
{
  Base* ptr = new Derived();          // mString is allocated here.
  delete ptr;          // ~Base is called, but not ~ Derived because the destructor is not virtual!
  return 0;
}
```

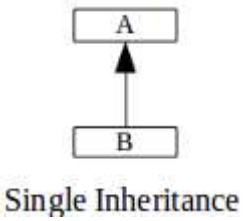
Different types of inheritance:

Single level inheritance (One derived class inherits from one base class)

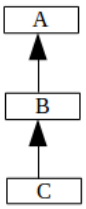
Multi level inheritance (A class inherits from a derived class) Eg: **multiLvlInh.cpp**

Hierarchial inheritance (A single class serves as a base class for more than one derived class) Eg: **pureVirtual_ABC.cpp**

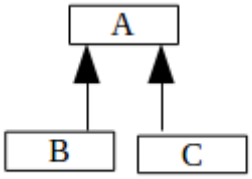
Multiple inheritance (A class derives from more than one base class) (Unit 5)



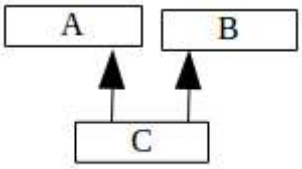
Single Inheritance



Multi level Inheritance



HierarchialInheritance



Multiple Inheritance

References:

- "C++ Primer", Stanley Lippman, Josee Lajoie, Barbara E Moo, Addison-Wesley Professional, 5th Edition
- https://nptel.ac.in/courses/106/105/106105151/