Friend functions C++	Friend classes	Inheritance	Multiple inheritances	Virtual Base Class	Classes inside classes

Friend functions C++

- private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not affect *friends*.
- If we want an external function will access private or protected member of a class then we need to declare that function as friend of that class. This is done using friend keyword.
- It is done by declaring a prototype of this external function within the class, and preceding it with the keyword **friend** as shown below.

Java does not have the friend keyword

```
// friend functions
#include <iostream>
using namespace std;
class CRectangle {
        int width, height;
        public:
            void set_values (int, int);
            int area () {return (width * height);}
            friend CRectangle duplicate (CRectangle);
};
void CRectangle::set_values (int a, int b) {
      width = a;      height = b;
}
CRectangle duplicate (CRectangle rectparam)
{
      CRectangle rectres;      rectres.width = rectparam.width*2;
      rectres.height = rectparam.height*2;
      return (rectres);
}
```

```
int main () {
   CRectangle rect, rectb;
   rect.set_values (2,3);
   rectb = duplicate (rect);
   cout << rectb.area();
   return 0;
}
24
```

Not a member of class so no object reference.

No class scope operator as it is not a member of class

This function is able to access the members width and height of different objects of type CRectangle, which are private members as it is declared as friend of Crectangle.

Notice that neither in the declaration of duplicate() nor in its later use in main() have we considered duplicate a member of class CRectangle. It simply has access to its private and protected members without being a member.

Generally, the use of friend functions is out of an object-oriented programming methodology, so whenever possible it is better to use members of the same class to perform operations with them.

Friend classes

We can also define a class as friend of another one. Then the class within which friend is declared (CSquare) granting the access to the protected and private members of it to the friend class (CRectangle).

```
// friend class
#include <iostream>
using namespace std;
class CSquare;
class CRectangle {
    int width, height;
    public:
        int area ()   {return (width * height);}
        void convert (CSquare a);
};
class CSquare {
    private:
        int side;
    public:
        void set_side (int a)  {side=a;}
        friend class CRectangle;
};
void CRectangle::convert (CSquare a) {
        width = a.side;     height = a.side;
}
```

```
int main () {
    CSquare sqr;
    CRectangle rect;
    sqr.set_side(4);
    rect.convert(sqr);
    cout << rect.area();
    return 0;
}
```

```
16
```

In this example, we have declared CRectangle as a friend of CSquare so that CRectangle member functions could have access to the protected and private members of CSquare, more concretely to CSquare::side, which describes the side width of the square.

You may also see something new at the beginning of the program: an empty declaration of class CSquare. This is necessary because within the declaration of CRectangle we refer to CSquare (as a parameter in convert()). The definition of CSquare is included later, so if we did not include a previous empty declaration for CSquare this class would not be visible from within the definition of CRectangle.

In our example, CRectangle is considered as a friend class by CSquare, but CRectangle does not consider CSquare to be a friend, so CRectangle can access the protected and private members of CSquare but not the reverse way. Of course, we could have declared also CSquare as friend of CRectangle if we wanted to.

Another property of friendships is that they are *not* *transitive*: The friend of a friend is not considered to be a friend unless explicitly specified.

Inheritance in C++

- Inheritance a key property of OOP helps to achieve reusability.
- It allows creating classes which are derived from other classes, so that they automatically include some of its "parent's" members, plus its own.

  For example, we want to declare a series of classes that describe polygons like our CRectangle, or like CTriangle. They have certain common properties, such as both can be described by means of only two sides: height and base. So we can create a base class CPolygon with common properties and other classes like CRectangle and CTriangle can be created by inheriting CPolygon with specific features that are different from one type of polygon to the other.

- Derived class can have access to all the accessible members of the base class.
- We use a colon (:) in the declaration of the derived class to indicated it is derived from base class:

  ```
  class derived_class_name: public base_class_name
  { /*...*/ };
  ```

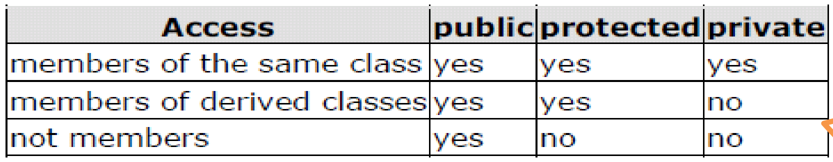
- The public access specifier before base class may be replaced by any one of the other access specifiers protected and private.
- This access specifier before base class limits the most accessible level for the members inherited from the base class:
  - The members with a more accessible level are inherited with this level instead
  - While the members with an equal or more restrictive access level keep their restrictive level in the derived class.

```
// derived classes
#include <iostream>
using namespace std;
class CPolygon {
   protected:
      int width, height;
   public:
      void set_values (int a, int b)
         { width=a; height=b;}
};
```

```
class CRectangle: public CPolygon {
   public:
      int area ()   { return (width * height); }
};
class CTriangle: public CPolygon {
   public:
      int area ()  { return (width * height / 2); }
};
```

```
int main () {
  CRectangle rect;    rect.set_values (4,5);     CTriangle trgl;       trgl.set_values (4,5);
  cout << rect.area() << endl;     cout << trgl.area() << endl;     return 0;   }
20    10
```

Since we wanted width and height to be accessible from members of the derived classes CRectangle and CTriangle and not only by members of CPolygon, we have used protected access instead of private. We can summarize the different access types according to who can access them in the following way:



Access	public	protected	private
members of the same class	yes	yes	yes
members of derived classes	yes	yes	no
not members	yes	no	no

"not members" represent any access from outside the class, such as from main(), from another class or from a function.

In our example, the members inherited by CRectangle and CTriangle have the same access permissions as they had in their base class CPolygon:

CPolygon::width // protected access CRectangle::width // protected access
CPolygon::set_values() // public access CRectangle::set_values() // public access

This is because we have used the public keyword to define the inheritance relationship on each of the derived classes: class CRectangle: public CPolygon { ... }

Since public is the most accessible level, by specifying this keyword the derived class will inherit all the members with the same levels they had in the base class. If we specify a more restrictive access level like **protected**, all public members of the base class are inherited as protected in the derived class. Whereas if we specify the most restricting of all access levels: **private**, all the base class members are inherited as private.

For example, if daughter was a class derived from mother that we defined as:

class daughter: protected mother;

This would set protected as the maximum access level for the members of daughter that it inherited from mother. That is, all members that were public in mother would become protected in daughter. Of course, this would not restrict daughter to declare its own public members. That maximum access level is only set for the members inherited from mother.

What is inherited from the base class?

In principle, a derived class inherits every member of a base class except:

- its constructor and its destructor
- its operator=() members
- its friends

Order of Constructor Call

Although the constructors and destructors of the base class are not inherited themselves, its default constructor (i.e., its constructor with no parameters) and its destructor are **always called when a new object of a derived class is created or destroyed**.

Base class constructors are always called in the derived class constructors. Whenever you create derived class object, first the base class default constructor is executed and then the derived class's constructor finishes execution.

Points to Remember

- Whether derived class's default constructor is called or parameterised is called, base class's default constructor is always called inside them.

- If the base class has no default constructor or you want that an overloaded constructor is called when a new derived object is created, you can specify it in each constructor definition of the derived class:

derived_constructor_name (parameters) : base_constructor_name (parameters) {...}

```
// constructors and derived classes
#include <iostream>
using namespace std;
class mother {
    public:
        mother ()       { cout << "mother: no parameters\n"; }
        mother (int a)  { cout << "mother: int parameter\n"; }
};
class daughter : public mother {
    public:
        daughter (int a)  { cout << "daughter: int parameter\n\n"; }
};
class son : public mother {
    public:
        son (int a) : mother (a)  {
            cout << "son: int parameter\n\n";
        }
};
int main () {
        daughter cynthia (0);
        son daniel(0);    return 0;
}
```

```
Output :

mother: no parameters
daughter: int parameter
mother: int parameter
son: int parameter
```

Notice the difference between which mother's constructor is called when a new daughter object is created and which when it is a son object.

```
daughter (int a)      // nothing specified:
                         call default
son (int a) : mother (a)   // constructor
                              specified: call this
```

Inheritance is always transitive, so that a class can inherit features from superclass many levels away. If class Dog is a subclass of Mammal, and class Mammal is a subclass of class Animal, then Dog will inherit attributes both from Mammal and from Animal.

Subclass can override behavior from parent class. For example, the class Platypus overrides the reproduction behavior inherited from class Mammal, since platypuses lay eggs.

Why is Base class Constructor called inside Derived class ?

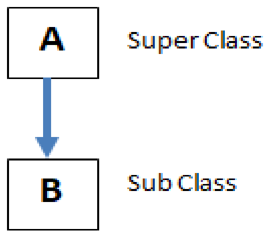
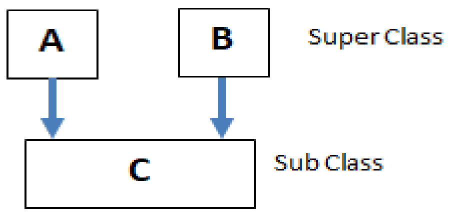
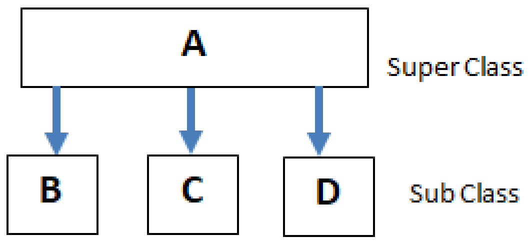
Constructors have a special job of initializing the object properly. A Derived class constructor has access only to its own class members, but a Derived class object also have inherited property of Base class, and only base class constructor can properly initialize base class members. Hence all the constructors are called, else object wouldn't be constructed properly.

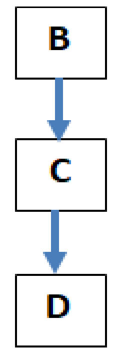
Constructor call in Multiple Inheritance

Its almost the same, all the Base class's constructors are called inside derived class's constructor, in the same order in which they are inherited.

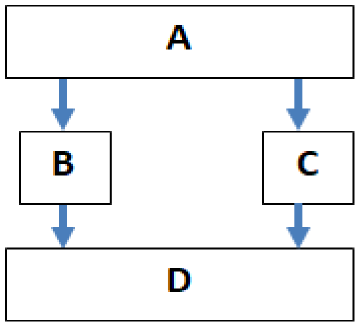
class A : public B, public C ; In this case, first class B constructor will be executed, then class C constructor and then class A constructor.

Types of Inheritance

In C++, we have 5 different types of Inheritance. Namely, 1. Single Inheritance 2. Multiple Inheritance 3. Hierarchical Inheritance 4. Multilevel Inheritance 5. Hybrid Inheritance (also known as Virtual Inheritance)	**Single Inheritance** One derived class inherits from only one base class. 
Multiple Inheritance A single derived class may inherit from two or more than two base classes. 	**Hierarchical Inheritance** Multiple derived classes inherit from a single base class. 
Multilevel Inheritance	**Hybrid (Virtual) Inheritance** Hybrid Inheritance is combination of Hierarchical and Mutilevel Inheritance.



The derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.

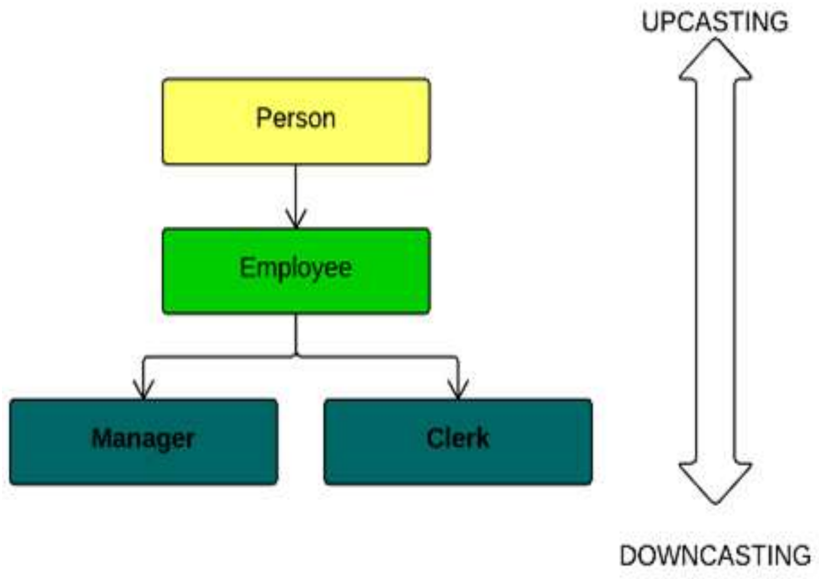


Upcasting and Downcasting in C++

Upcasting and downcasting are an important part of C++. Upcasting and downcasting gives a possibility to build complicated programs with a simple syntax. It can be achieved by using Polymorphism.

- C++ allows that a derived class pointer (or reference) to be treated as base class pointer. This is upcasting.
- Downcasting is an opposite process, which consists in converting base class pointer (or reference) to derived class pointer.
- Upcasting and downcasting should not be understood as a simple casting of different data types. It can lead to a great confusion.
- Both upcasting and downcasting do not change object by itself. When you use upcasting or downcasting you just "label" an object in different ways.

To understand the concepts we use following hierarchy of classes:



- As you can see, Manager and Clerk are both Employee. They are both Person too
- Both Manager and Clerk are identified by First and Last name, have salary; you can show information about them and add a bonus to their salaries. We have to specify these properties only once in the Employee class:
- In the same time, Manager and Clerk classes are different. Manager takes a commission fee for every contract, and Clerk has information about his Manager.

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

class Person
{
            //content of Person
};
class Employee: public Person
{
public:
      Employee(string fName, string lName, double sal)
      {
               FirstName = fName;       LastName = lName;      salary = sal;
      }
      string FirstName;        string LastName;              double salary;
      void show()
      {
            cout << "First Name: " << FirstName << " Last Name: " << LastName << " Salary: "
                              << salary<< endl;
      }
      void addBonus(double bonus)    {   salary += bonus;   }
};

class Manager :public Employee
{
public:
     Manager(string fName, string lName, double sal, double comm) : Employee(fName, lName, sal)
     {
            Commision = comm;
     }
     double Commision;
     double getComm()  {     return Commision;    }
};

class Clerk :public Employee
{
public:
     Clerk(string fName, string lName, double sal, Manager* man) :Employee(fName, lName, sal)
     {
            manager = man;
     }
     Manager* manager;
     Manager* getManager() {           return manager;    }
};
```

```
void congratulate(Employee* emp)
{
        cout << "Happy Birthday!!!" << endl;    emp->addBonus(200);    emp->show();
};

int main()
{
  //pointer to base class object
  Employee* emp;

  //object of derived class
  Manager m1("Steve", "Kent", 3000, 0.2);
  Clerk c1("Kevin","Jones", 1000, &m1);

  //implicit upcasting
  emp = &m1;

  //It's ok
  cout<<emp->FirstName<<endl;
  cout<<emp->salary<<endl;

  //Fails because upcasting is used
  //cout<<emp->getComm();

  congratulate(&c1);
  congratulate(&m1);

  cout<<"Manager of "<<c1.FirstName<<" is "<<c1.getManager()->FirstName;
}
```

Manager and Clerk are always Employees. Moreover, Employee is a Person. Therefore, Manager and Clerk are Persons too. You have to understand it before we start learning upcasting and downcasting.

UPCASTING

- Upcasting is a process of treating a pointer or a reference of derived class object as a base class pointer.
- You do not need to upcast manually. You just need to assign derived class pointer (or reference) to base class pointer:

```
Employee* emp;                          //pointer to base class object
Manager m1("Steve", "Kent", 3000, 0.2);   //object of derived class
emp = &m1;    //implicit upcasting
```

- When you use upcasting, the object is not changing. Nevertheless, when you upcast an object, you will be able to access only member functions and data members that are defined in the base class:

```
//It's ok
emp->FirstName;        emp->salary;
//Fails because upcasting is used
emp->getComm();
```

Example of upcasting usage

One of the biggest advantage of upcasting is the capability of writing generic functions for all the classes that are derived from the same base class. Look on example:

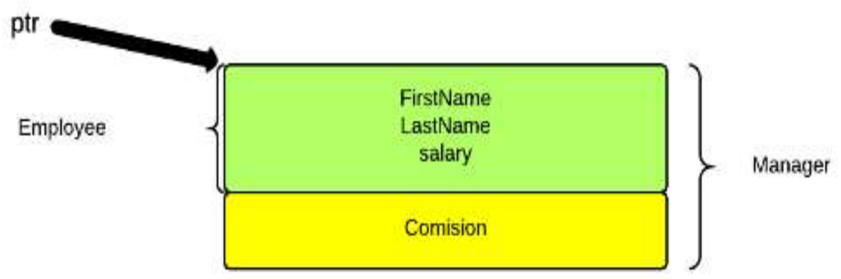
```
void congratulate(Employee* emp)
{
        cout << "Happy Birthday!!!" << endl;
        emp->show();
        emp->addBonus(200);
};
```

This function will work with all the classes that are derived from the Employee class. When you call it with objects of type Manager and Person, they will be automatically upcasted to Employee class:

```
//automatic upcasting
congratulate(&c1);
congratulate(&m1);
```

Memory layout

Look on the memory layout of the Employee and Manager classes:



Of course, this model is simplified view of memory layout for objects. However, it represents the fact that when you use base class pointer to point up an object of the derived class, you can access only elements that are defined in the base class (green area). Elements of the derived class (yellow area) are not accessible when you use base class pointer.

DOWNCASTING

- Downcasting is an opposite process for upcasting. It converts base class pointer to derived class pointer.
- Downcasting must be done manually. It means that you have to specify explicit type cast.
- Downcasting is not safe as upcasting. You know that a derived class object can be always treated as base class object. However, the opposite is not right. For example, a Manager is always a Person; But a Person is not always a Manager. It could be a Clerk too.
- You have to use an explicit cast for downcasting:

```
//pointer to base class object
Employee* emp;
//object of derived class
Manager m1("Steve", "Kent", 3000, 0.2);
//implicit upcasting
emp = &m1;
//explicit downcasting from Employee to Manager
Manager* m2 = (Manager*)(emp);
```

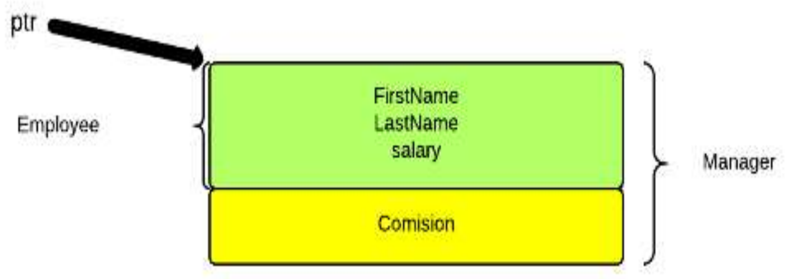
This code compiles and runs without any problem, because emp points to an object of Manager class.

- What will happen, if we try to downcast a base class pointer that is pointing to an object of base class and not to an object of derived class? Try to compile and run this code:

```
Employee e1("Peter", "Green", 1400);
//try to cast an employee to Manager
Manager* m3 = (Manager*)(&e1);
cout << m3->getComm() << endl;
```

e1 object is not an object of Manager class. It does not contain any information about commission. That is why such an operation can produce unexpected results.

Look on the memory layout again:



When you try to downcast base class pointer (Employee) that is not actually pointing up an object of derived class (Manager), you will get access to the memory that does not have any information about derived class object (yellow area). **This is the main danger of downcasting.**

You can use a safe cast that can help you to know, if one type can be converted correctly to another type. For this purpose, use dynamic cast.

Dynamic Cast

- dynamic_cast is an operator that converts safely one type to another type. In the case, the conversation is possible and safe, it returns the address of the object that is converted. Otherwise, it returns nullptr.
- dynamic_cast has the syntax : **dynamic_cast**<new_type> (**object**)
- If you want to use dynamic cast for downcasting, base class should be polymorphic - it must have at least one virtual function. Modify base class Person by adding a virtual function:

```
        virtual void foo() {}
```

Now you can use downcasting for converting Employee class pointers to derived classes pointers.

```
Employee e1("Peter", "Green", 1400);
Manager* m3 = dynamic_cast<Manager*>(&e1);
if (m3)
        cout << m3->getComm() << endl;
else
        cout << "Can't  cast from Employee to Manager" << endl;
```

In this case, dynamic cast returns nullptr. Therefore, you will see a warning message

Inheritance and Static Functions

- They are inherited into the derived class.
- If you redefine a static member function in derived class, all the other overloaded functions in base class are hidden.
- Static Member functions can never be virtual. We will study about Virtual in polymorphism.

Hybrid Inheritance and Virtual Class

In Multiple Inheritance, the derived class inherits from more than one base class. Hence, in Multiple Inheritance there are a lot chances of ambiguity.

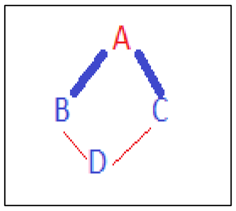
```
class A    { void show(); };

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

class D: public B, public C {};

int main() {
   D obj;   obj.show();
}
```

In this case both class B and C inherits function show() from class A. Hence class D has two inherited copies of function show(). In main() function when we call function show(), then ambiguity arises, because compiler doesn't know which show() function to call. Hence we use Virtual keyword while inheriting class.

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

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

We tell compiler to call any one out of the two show() funtions.

Hybrid Inheritance and Constructor call



As we all know that whenever a derived class object is instantiated, the base class constructor is always called. But in case of Hybrid Inheritance, as discussed in above example, if we create an instance of class D, then following constructors will be called :

- before class D's constructor, constructors of its super classes will be called, hence constructors of class B, class C and class A will be called.
- when constructors of class B and class C are called, they will again make a call to their super class's constructor.
- This will result in multiple calls to the constructor of class A, which is undesirable.

When we use virtual base class as shown above, there is a single instance of virtual base class which is shared by multiple classes that inherit from it. Hence the constructor of the base class is only called once by the constructor of concrete class, which in our case is class D.

If there is any call for initializing the constructor of class A in class B or class C, while creating object of class D, all such calls will be skipped.

Multiple inheritances

Multiple inheritance : a class inherits members from more than one class (not possible in Java).
For example, if we had a specific class to print on screen (COutput) and we wanted our classes CRectangle and CTriangle to also inherit its members in addition to those of CPolygon we could write:

```
class CRectangle: public CPolygon, public COutput;
class CTriangle: public CPolygon, public COutput;
```

Here is the complete example:

```
// multiple inheritance
#include <iostream>
using namespace std;
class CPolygon {
    protected:
        int width, height;
    public:
        void set_values (int a, int b)
            { width=a; height=b;}
};
class COutput {
    public:
```

```
class CRectangle: public CPolygon,
                        public COutput {
   public:
      int area ()   { return (width * height); }
};
class CTriangle: public CPolygon,
                       public COutput {
   public:
      int area ()  { return (width * height / 2); }
};
```

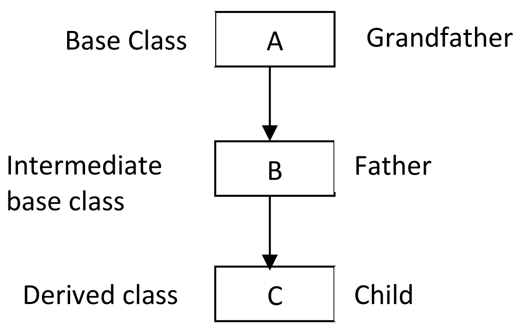
```
int main () {
    CRectangle rect;          CTriangle trgl;
    rect.set_values (4,5);     trgl.set_values (4,5);
    rect.output (rect.area());
    trgl.output (trgl.area());
    return 0;
}
```

```
        void output (int i);
};
void COutput::output (int i) {      cout << i << endl;   }
```

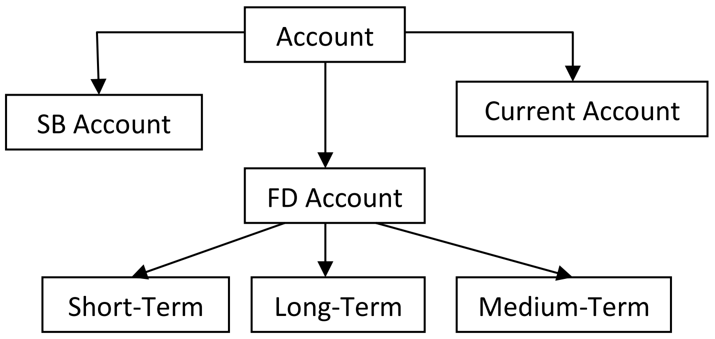
Virtual Base Class

There may be different types of inheritance. Some of the important types are Multilevel, Hierarchical, Multiple and Multipath.

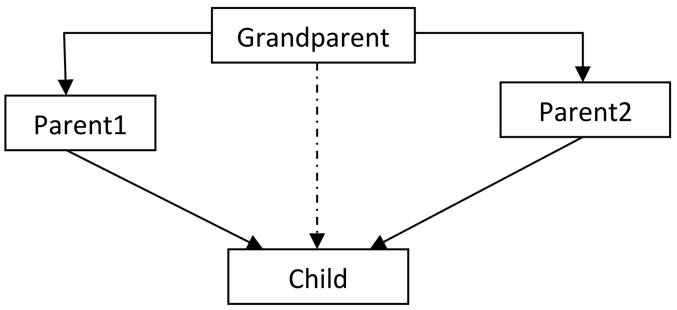
Multilevel Inheritance



Hierarchical Inheritance



Multipath Inheritance



In some situation we may need to use of both multiple and multilevel inheritance as shown in multipath The **Child** has two direct base class **Parent1** and **Parent2** which themselves have a common base class **Grandparent**.

So Child inherits the behavior of Grandparent via two paths. It can also inherit directly as show in broken line.

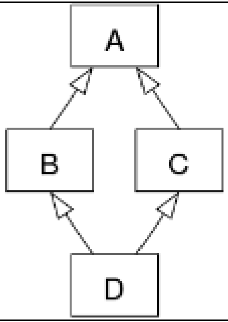
- The **grandparent** is sometimes referred to as **indirect base class.**
- This type of inheritance might pose some problems. All the public and protected members of grandparent are inherited into child **twice** via parent1 and parent2. This will cause **ambiguity**.
- The duplication of inherited members due to three multiple paths can be avoided by making the common base class as **virtual base class** while declaring the direct or intermediate base classes.

Class Grandparent
{
}
	Class Parent1 : virtual public Grandparent
	{
	}
		Class Parent2 : virtual public Grandparent
		{
		}
			Class Child : public Parent1, public Parent2
			{
			}

- When a class is made a virtual base class, C++ takes care to see that only one copy of that is inherited, regardless of how many inheritance paths exist between the virtual base class and derived class.

Solving the Diamond Problem with Virtual Inheritance

- Multiple inheritance in C++ is a powerful, but tricky tool, that often leads to problems if not used carefully.
- A complicated situation that arises when using multiple inheritance is the **diamond problem**, which occurs when two classes are derived from a base class and another class is obtained by inheritance from those derived classes.
- An ambiguity will rise
  - If the derived classes override the same method from the base class and merged class try to call that method.
  - Joining class also try to override that method.
- A second problem that can occur with the diamond pattern is that if the two classes derived from the same base class, and that base class has one or more members, then those members will be duplicated in the joining class.



- Suppose we have 2 classes B and C that derive from the ***same*** class A
- We also have class D that derives from ***both*** B and C by using multiple inheritance.
- From figure it is clear it forms the shape of a diamond – which is why this problem is called the **diamond problem**.

Let's translate it into actual code:

```
// The Animal class is a base class
class Animal { /* ... */ }; // base class
{
    int weight;
    public:
    int getWeight() { return weight;};
};
int main( )
{
    Liger lg ;
    /*COMPILE ERROR, the code below will not
       get past any C++ compiler */
    int weight = lg.getWeight();
}
```

```
class Tiger : public Animal { /* ... */ };
class Lion : public Animal { /* ... */ }
class Liger : public Tiger, public Lion { /* ... */ };
```

Problem : As Liger derives from both the Tiger and Lion classes – the Liger object "lg" will contain *two* subobjects of the *Animal* base class.

The call "lg.getWeight()" is ambiguous because the compiler **does not know**

- if the call to getWeight refers to the copy of getWeight that the Liger object lg inherited through the Lion class or
- the copy that lg inherited through the Tiger class.

Virtual inheritance only solves the duplication of the grandfather! BUT you still need to specify the methods to be virtual in order to get the methods correctly overrided...

Solution to the Diamond Problem

If the inheritance from the Animal class to both the Lion class and the Tiger class is marked as virtual, then C++ will ensure that only one subobject of the Animal class will be created for every Liger object. This is what the code for that would look like:

```
class Tiger : virtual public Animal { /* ... */ };

class Lion : virtual public Animal { /* ... */ }

int main( )
{
      Liger lg ;
      /*THIS CODE WILL NOW COMPILE OK NOW THAT WE'VE USED THE VIRTUAL KEYWORD IN THE
         TIGER AND LION CLASS DECLARATIONS */
      int weight = lg.getWeight();
}
```

You can see that the only change we've made is to add the "virtual" keyword to the Tiger and Lion class declarations. Now the Liger class object will have only one Animal subobject, and the code below will compile just fine.

Other ambiguity arise from multiple inheritance and its solution

Example of ambiguities and its solution :

f() function is available in class A and B and it is also overridden in subclass C. So call of f() will cause ambiguity.

```
class A { virtual void f(); };
class B { virtual void f(); };
class C : public A ,public B { void f(); };
```

This issue can be solved by using explicit qualification. We explicitly say to the compiler where to get the function that we need to call:

```
C* pc = new C;
pc->f();
pc->A::f();      //this calls f() from class A
pc->B::f();      //this calls f() from class B
```

Each base class can be uniquely identified by using the scope resolution operator :: .

- If the C class didn't have the method f() (not overridden) the problem can be solved using implicit conversion :
  ```
  A* pa = pc;
  pa->f();
  ```

or we would have to make a cast in order to call the method from the parent class A.

Another Example

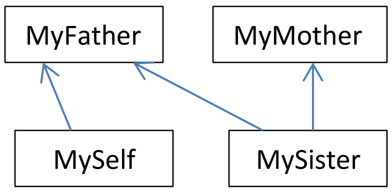
```
#include <iostream>
using namespace std;

// The base class #1
```

```
// Another base class  #2
class MyMother
{
  // notice the same member variables names
```

```
class MyFather
{
  protected:
      char* EyeColor;
      char* HairType;
      double FamSaving;
      int FamCar;

  public:
      MyFather(){}
      ~MyFather(){}
      char* ShowEyeColor()   {
           return EyeColor = "Brown";
      };

      char* ShowHairType() {
           return HairType = "Bald";
      };
      long double FamilySaving() {
           return FamSaving = 100000L;
      };
      int FamilyCar()  {
          return  FamCar = 4;
      };
};
```

```
// as in MyFather class...
protected:
     char* EyeColor;
     char* HairType;
     int FamHouse;

public:
     MyMother(){}
     ~MyMother(){}
     char* ShowMotherEye()  {
            return EyeColor = "Blue";
     };
     char* ShowMotherHair()  {
             return HairType = "Curly Blonde";
     };

     int FamilyHouse() {
          return FamHouse = 3;
     };
};
```

No diamond formation



// Single inheritance derived class ...

```
class MySelf : public MyFather
{
          char* HairType;
          char* Education;
     public:
          MySelf(){ }
          ~MySelf(){ }
        char* ShowMyHair() {   return HairType = "Straight Black";      };
        char* ShowMyEducation()  {    return Education = "Post Graduate";       };
};
// Multiple inheritance derived class...notice the keyword public must follow every
// parent class list as needed...
class MySister: public MyFather, public MyMother
{
        char* SisEye;
```

HairType is already declared in MyFather. Here same name again appear.

```
        float MonAllowance;

        public:
            MySister(){}
            ~MySister(){}
            char* ShowSisEye() {  return SisEye = "Black";     };
            float ShowSisAllownace() {  return MonAllowance = 1000.00;  };
};
```

- char* EyeColor and char* HairType fields are common in MyFather and MyMother. As MySister is inherited from both these can cause **ambiguity**.
- **Solution of ambiguity in multiple inheritance**: The ambiguity can be resolved by using the scope resolution operator :: to specify the class in which the member function lies as given below:

ObjSis.MyMother :: ShowMotherHair ();

```
// The main function
int main()
{
    // instantiate the objects...
    MyFather   ObjFat;      MyMother ObjMot;
    MySelf ObjSelf;          MySister ObjSis;

    cout<<"--My father's data--"<<endl;
    cout<<"His eye:          "<< ObjFat.ShowEyeColor() <<"\n"
       << "His hair:         " <<  ObjFat.ShowHairType() <<"\n"
       << "Family Saving: USD"<< ObjFat.FamilySaving() <<"\n"
       << "Family Car:    "<< ObjFat.FamilyCar() <<" cars.\n";

    cout<<"\n--My mother's data--"<<endl;
    cout<<"Her eye: " <<  ObjMot.ShowMotherEye() <<endl;
    cout<<"Her hair: " << ObjMot.ShowMotherHair()<<endl;
    cout<<"Our family house: "<< ObjMot.FamilyHouse() <<" houses."<<endl;

    // Notice how to access the base/parent class member functions through the child or
       derived objects...
    cout<<"\n--My data--"<<endl;
    cout<<"My Hair: "<< ObjSelf. ShowMyHair() <<endl;
    cout<<"My family saving: USD"<< ObjSelf. MyFather::FamilySaving() <<endl;
    cout<<"My family car: "<< ObjSelf. MyFather::FamilyCar() <<" cars." <<endl;
    cout<<"My education: "<< ObjSelf.ShowMyEducation()<<endl;

    cout<<"\n--My sister's data--"<<endl;
    cout<<"Her eye: "<<ObjSis. ShowSisEye()<<endl;
    cout<<"Our family saving: USD"<< ObjSis.MyFather::FamilySaving()<<endl;
    cout<<"Our family car: "<<ObjSis.MyFather::FamilyCar()<<" cars."<<endl;
    cout<<"Our family house: "<<ObjSis.MyMother::FamilyHouse()<<" houses."<<endl;
    cout<<"Her monthly allowances: USD"<<ObjSis.ShowSisAllownace()<<endl;

    return 0;
}
```