Overview of Inheritance

Inheritance is the capability of one class to acquire properties and characteristics from another class. The class whose properties are inherited by other class is called the **Parent** or **Base** or **Super** class. And, the class which inherits properties of other class is called **Child** or **Derived** or **Sub** class.

Inheritance makes the code reusable. When we inherit an existing class, all its methods and fields become available in the new class, hence code is reused.

NOTE : All members of a class except Private, are inherited

Purpose of Inheritance

1. Code Reusability
2. Method Overriding (Hence, Runtime Polymorphism.)
3. Use of Virtual Keyword

Basic Syntax of Inheritance

class Subclass_name : access_mode Superclass_name

While defining a subclass like this, the super class must be already defined or at least declared before the subclass declaration.

Access Mode is used to specify, the mode in which the properties of superclass will be inherited into subclass, public, privtate or protected.

Inheritance Visibility Mode

Depending on Access modifier used while inheritance, the availability of class members of Super class in the sub class changes. It can either be private, protected or public.

Public Inheritance

This is the most used inheritance mode. In this the protected member of super class becomes protected members of sub class and public becomes public.

class Subclass : public Superclass

Private Inheritance

In private mode, the protected and public members of super class become private members of derived class.

class Subclass : Superclass   // By default its private inheritance

Table showing all the Visibility Modes

	Derived Class	**Derived Class**	**Derived Class**
Base class	**Public Mode**	**Private Mode**	**Protected Mode**
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

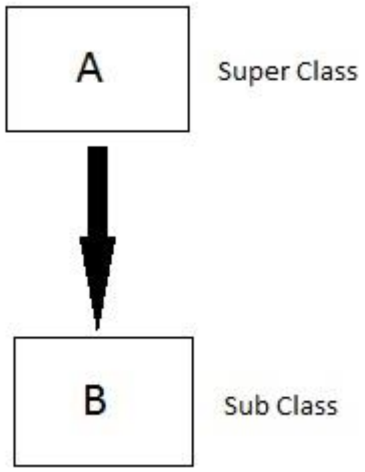
Types of Inheritance

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)

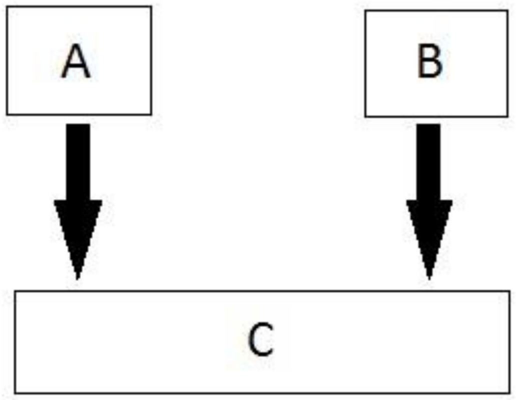
Single Inheritance

In this type of inheritance one derived class inherits from only one base class. It is the simplest form of Inheritance.



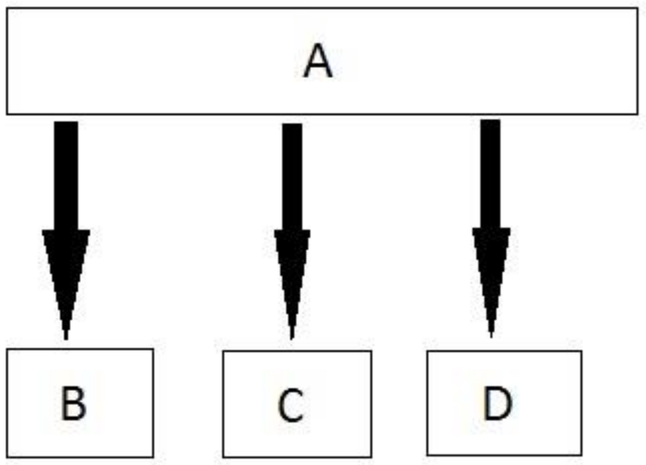
Multiple Inheritance

In this type of inheritance a single derived class may inherit from two or more than two base classes.



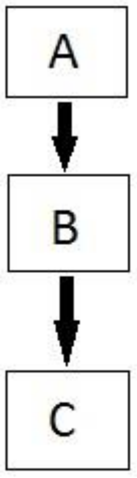
Hierarchical Inheritance

In this type of inheritance, multiple derived classes inherits from a single base class.



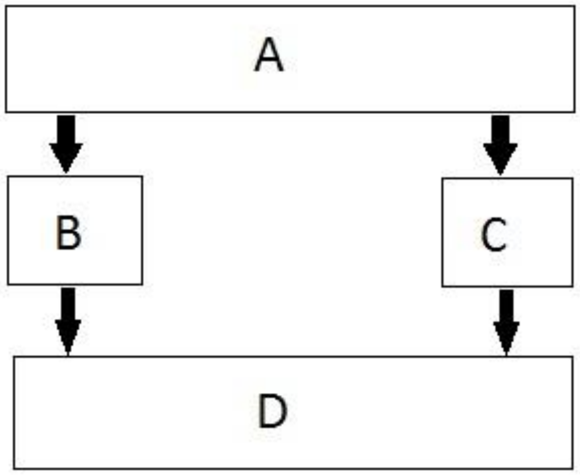
Multilevel Inheritance

In this type of 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.



Hybrid (Virtual) Inheritance

Hybrid Inheritance is combination of Hierarchical and Multilevel Inheritance.



Single Inheritance

```
// inheritance using English Distances
#include <iostream>
using namespace std;
enum posneg { pos, neg };          //for sign in DistSign
////////////////////////////////////////////////////////////////
class Distance                     //English Distance class
   {
   protected:                      //NOTE: can't be private
      int feet;
      float inches;
   public:                         //no-arg constructor
      Distance() : feet(0), inches(0.0)
         {  }                      //2-arg constructor)
      Distance(int ft, float in) : feet(ft), inches(in)
         {  }
      void getdist()               //get length from user
         {
         cout << "\nEnter feet: ";  cin >> feet;
         cout << "Enter inches: ";  cin >> inches;
         }
      void showdist() const        //display distance
         { cout << feet << "\'-" << inches << '\"'; }
   };
////////////////////////////////////////////////////////////////
class DistSign : public Distance   //adds sign to Distance
   {
   private:
      posneg sign;                 //sign is pos or neg
   public:
                                   //no-arg constructor
      DistSign() : Distance()      //call base constructor
         { sign = pos; }           //set the sign to +

                                   //2- or 3-arg constructor
      DistSign(int ft, float in, posneg sg=pos) :
```

```
               Distance(ft, in)    //call base constructor
          { sign = sg; }           //set the sign

      void getdist()               //get length from user
               {
               Distance::getdist();     //call base getdist()
               char ch;                 //get sign from user
               cout << "Enter sign (+ or -): ";  cin >> ch;
               sign = (ch=='+') ? pos : neg;
               }
      void showdist() const         //display distance
               {
               cout << ( (sign==pos) ? "(+)" : "(-)" );  //show sign
               Distance::showdist();                     //ft and in
               }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   DistSign alpha;                   //no-arg constructor
   alpha.getdist();                  //get alpha from user

   DistSign beta(11, 6.25);          //2-arg constructor

   DistSign gamma(100, 5.5, neg);    //3-arg constructor

                                  //display all distances
   cout << "\nalpha = ";  alpha.showdist();
   cout << "\nbeta = ";  beta.showdist();
   cout << "\ngamma = ";  gamma.showdist();
   cout << endl;
   return 0;
   }
```

Public and private inheritance

```
// tests publicly- and privately-derived classes

#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class A                   //base class
   {
   private:
      int privdataA;      //(functions have the same access
   protected:             //rules as the data shown here)
      int protdataA;
   public:
      int pubdataA;
   };
////////////////////////////////////////////////////////////////
class B : public A        //publicly-derived class
   {
   public:
      void funct()
         {
         int a;
         a = privdataA;  //error: not accessible
         a = protdataA;  //OK
         a = pubdataA;   //OK
         }
   };
////////////////////////////////////////////////////////////////
class C : private A      //privately-derived class
   {
   public:
      void funct()
         {
         int a;
         a = privdataA;  //error: not accessible
         a = protdataA;  //OK
         a = pubdataA;   //OK
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   int a;

   B objB;
   a = objB.privdataA;   //error: not accessible
   a = objB.protdataA;   //error: not accessible
   a = objB.pubdataA;    //OK (A public to B)

   C objC;
   a = objC.privdataA;   //error: not accessible
   a = objC.protdataA;   //error: not accessible
   a = objC.pubdataA;    //error: not accessible (A private to C)
   return 0;
   }
```

Overriding functions in the subclasses

```
// models employee database using inheritance
#include <iostream>
using namespace std;
const int LEN = 80;                 //maximum length of names
////////////////////////////////////////////////////////////////
class employee                      //employee class
   {
   private:
      char name[LEN];               //employee name
      unsigned long number;         //employee number
   public:
      void getdata()
         {
         cout << "\n   Enter last name: "; cin >> name;
         cout << "   Enter number: ";      cin >> number;
         }
      void putdata() const
         {
         cout << "\n   Name: " << name;
         cout << "\n   Number: " << number;
         }
   };
////////////////////////////////////////////////////////////////
class manager : public employee     //management class
   {
   private:
      char title[LEN];              //"vice-president" etc.
      double dues;                  //golf club dues
   public:
```

```cpp
      void getdata()
         {
         employee::getdata();
         cout << "   Enter title: ";          cin >> title;
         cout << "   Enter golf club dues: "; cin >> dues;
         }
      void putdata() const
         {
         employee::putdata();
         cout << "\n   Title: " << title;
         cout << "\n   Golf club dues: " << dues;
         }
   };
////////////////////////////////////////////////////////////////
class scientist : public employee  //scientist class
   {
   private:
      int pubs;                    //number of publications
   public:
      void getdata()
         {
         employee::getdata();
         cout << "   Enter number of pubs: "; cin >> pubs;
         }
      void putdata() const
         {
         employee::putdata();
         cout << "\n   Number of publications: " << pubs;
         }
   };
////////////////////////////////////////////////////////////////
class laborer : public employee    //laborer class
   {
   };
////////////////////////////////////////////////////////////////
int main()
   {
   manager m1, m2;
   scientist s1;
   laborer l1;

   cout << endl;            //get data for several employees
   cout << "\nEnter data for manager 1";
   m1.getdata();

   cout << "\nEnter data for manager 2";
   m2.getdata();

   cout << "\nEnter data for scientist 1";
   s1.getdata();

   cout << "\nEnter data for laborer 1";
   l1.getdata();
                            //display data for several employees
   cout << "\nData on manager 1";
   m1.putdata();

   cout << "\nData on manager 2";
   m2.putdata();

   cout << "\nData on scientist 1";
   s1.putdata();
   cout << "\nData on laborer 1";
   l1.putdata();
   cout << endl;
   return 0;
   }
```

Levels of inheritance

```
// multiple levels of inheritance
#include <iostream>
using namespace std;
const int LEN = 80;                 //maximum length of names
////////////////////////////////////////////////////////////////
class employee
   {
   private:
      char name[LEN];               //employee name
      unsigned long number;         //employee number
   public:
      void getdata()
         {
         cout << "\n   Enter last name: "; cin >> name;
         cout << "   Enter number: ";      cin >> number;
         }
      void putdata() const
         {
         cout << "\n   Name: " << name;
         cout << "\n   Number: " << number;
         }
   };
////////////////////////////////////////////////////////////////
class manager : public employee    //manager class
   {
   private:
      char title[LEN];              //"vice-president" etc.
      double dues;                  //golf club dues
   public:
      void getdata()
         {
         employee::getdata();
         cout << "   Enter title: ";          cin >> title;
         cout << "   Enter golf club dues: "; cin >> dues;
         }
      void putdata() const
         {
         employee::putdata();
         cout << "\n   Title: " << title;
         cout << "\n   Golf club dues: " << dues;
         }
   };
////////////////////////////////////////////////////////////////
class scientist : public employee  //scientist class
   {
   private:
      int pubs;                     //number of publications
   public:
      void getdata()
         {
         employee::getdata();
         cout << "   Enter number of pubs: "; cin >> pubs;
```

```
        }
      void putdata() const
        {
        employee::putdata();
        cout << "\n   Number of publications: " << pubs;
        }
   };
////////////////////////////////////////////////////////////////
class laborer : public employee    //laborer class
   {
   };
////////////////////////////////////////////////////////////////
class foreman : public laborer     //foreman class
   {
   private:
      float quotas;  //percent of quotas met successfully
   public:
      void getdata()
        {
        laborer::getdata();
        cout << "   Enter quotas: "; cin >> quotas;
        }
      void putdata() const
        {
        laborer::putdata();
        cout << "\n   Quotas: " << quotas;
        }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   laborer l1;
   foreman f1;

   cout << endl;
   cout << "\nEnter data for laborer 1";
   l1.getdata();
   cout << "\nEnter data for foreman 1";
   f1.getdata();

   cout << endl;
   cout << "\nData on laborer 1";
   l1.putdata();
   cout << "\nData on foreman 1";
   f1.putdata();
   cout << endl;
   }
```

Member functions in multiple inheritance

```
// englmult.cpp
// multiple inheritance with English Distances
#include <iostream>
#include <string>
using namespace std;
////////////////////////////////////////////////////////////////
class Type                        //type of lumber
   {
   private:
      string dimensions;
      string grade;
   public:                        //no-arg constructor
      Type() : dimensions("N/A"), grade("N/A")
         {  }
                                  //2-arg constructor
      Type(string di, string gr) : dimensions(di), grade(gr)
         {  }
      void gettype()              //get type from user
         {
         cout << "   Enter nominal dimensions (2x4 etc.): ";
         cin >> dimensions;
         cout << "   Enter grade (rough, const, etc.): ";
         cin >> grade;
         }
      void showtype() const       //display type
         {
         cout << "\n   Dimensions: " << dimensions;
         cout << "\n   Grade: " << grade;
         }
   };
////////////////////////////////////////////////////////////////
class Distance                    //English Distance class
   {
   private:
      int feet;
      float inches;
   public:                        //no-arg constructor
      Distance() : feet(0), inches(0.0)
         {  }                     //constructor (two args)
      Distance(int ft, float in) : feet(ft), inches(in)
         {  }
      void getdist()              //get length from user
         {
         cout << "   Enter feet: ";  cin >> feet;
         cout << "   Enter inches: ";  cin >> inches;
         }
      void showdist() const       //display distance
         { cout  << feet << "\'-" << inches << '\"'; }
   };
////////////////////////////////////////////////////////////////
class Lumber : public Type, public Distance
   {
   private:
      int quantity;                         //number of pieces
      double price;                         //price of each piece
   public:                                  //constructor (no args)
      Lumber() : Type(), Distance(), quantity(0), price(0.0)
         {  }
                                    //constructor (6 args)
      Lumber( string di, string gr,         //args for Type
              int ft, float in,             //args for Distance
              int qu, float prc ) :         //args for our data
              Type(di, gr),                 //call Type ctor
              Distance(ft, in),             //call Distance ctor
              quantity(qu), price(prc)      //initialize our data
          {  }
      void getlumber()
         {
         Type::gettype();
         Distance::getdist();
         cout << "   Enter quantity: "; cin >> quantity;
         cout << "   Enter price per piece: "; cin >> price;
         }
      void showlumber() const
         {
         Type::showtype();
         cout << "\n   Length: ";
         Distance::showdist();
         cout << "\n   Price for " << quantity
                << " pieces: $" << price * quantity;
         }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Lumber siding;                   //constructor (no args)

   cout << "\nSiding data:\n";
   siding.getlumber();              //get siding from user

                                    //constructor (6 args)
   Lumber studs( "2x4", "const", 8, 0.0, 200, 4.45F );

                                    //display lumber data
   cout << "\nSiding";  siding.showlumber();
   cout << "\nStuds";     studs.showlumber();
   cout << endl;
   return 0;
   }
```

Constructors in multiple inheritance

```
// multiple inheritance with English Distances
#include <iostream>
#include <string>
using namespace std;
////////////////////////////////////////////////////////////////
class Type                        //type of lumber
   {
   private:
      string dimensions;
      string grade;
   public:                        //no-arg constructor
      Type() : dimensions("N/A"), grade("N/A")
         {  }
                                  //2-arg constructor
      Type(string di, string gr) : dimensions(di), grade(gr)
         {  }
      void gettype()              //get type from user
         {
         cout << "   Enter nominal dimensions (2x4 etc.): ";
         cin >> dimensions;
         cout << "   Enter grade (rough, const, etc.): ";
         cin >> grade;
         }
      void showtype() const       //display type
         {
         cout << "\n   Dimensions: " << dimensions;
         cout << "\n   Grade: " << grade;
         }
   };
////////////////////////////////////////////////////////////////
class Distance                    //English Distance class
   {
   private:
      int feet;
      float inches;
   public:                        //no-arg constructor
      Distance() : feet(0), inches(0.0)
         {  }                     //constructor (two args)
      Distance(int ft, float in) : feet(ft), inches(in)
         {  }
      void getdist()              //get length from user
         {
         cout << "   Enter feet: ";  cin >> feet;
         cout << "   Enter inches: ";  cin >> inches;
         }
      void showdist() const       //display distance
         { cout  << feet << "\'-" << inches << '\"'; }
   };
////////////////////////////////////////////////////////////////
class Lumber : public Type, public Distance
   {
   private:
      int quantity;                      //number of pieces
      double price;                      //price of each piece
   public:                               //constructor (no args)
      Lumber() : Type(), Distance(), quantity(0), price(0.0)
         {  }
                                  //constructor (6 args)
      Lumber( string di, string gr,      //args for Type
              int ft, float in,          //args for Distance
              int qu, float prc ) :      //args for our data
              Type(di, gr),              //call Type ctor
              Distance(ft, in),          //call Distance ctor
              quantity(qu), price(prc)   //initialize our data
          {  }
      void getlumber()
         {
         Type::gettype();
         Distance::getdist();
         cout << "   Enter quantity: "; cin >> quantity;
         cout << "   Enter price per piece: "; cin >> price;
         }
      void showlumber() const
         {
         Type::showtype();
         cout << "\n   Length: ";
         Distance::showdist();
         cout << "\n   Price for " << quantity
              << " pieces: $" << price * quantity;
         }
   };
////////////////////////////////////////////////////////////////

int main()
   {
   Lumber siding;                     //constructor (no args)

   cout << "\nSiding data:\n";
   siding.getlumber();                //get siding from user

                                      //constructor (6 args)
   Lumber studs( "2x4", "const", 8, 0.0, 200, 4.45F );

                                      //display lumber data
   cout << "\nSiding";  siding.showlumber();
   cout << "\nStuds";     studs.showlumber();
   cout << endl;
   return 0;
   }
```

Ambiguity in multiple inheritance

```
// ambigu.cpp
// demonstrates ambiguity in multiple inheritance
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class A
   {
   public:
      void show()  { cout << "Class A\n"; }
   };
class B
   {
   public:
      void show()  { cout << "Class B\n"; }
   };
class C : public A, public B
   {
   };
////////////////////////////////////////////////////////////////
int main()
   {
   C objC;            //object of class C
// objC.show();       //ambiguous--will not compile
   objC.A::show();    //OK
   objC.B::show();    //OK
   return 0;
   }
```

Aggregation: Classes within classes

```
// containership with employees and degrees
#include <iostream>
#include <string>
using namespace std;
////////////////////////////////////////////////////////////////
class student                  //educational background
   {
   private:
      string school;           //name of school or university
      string degree;           //highest degree earned
   public:
      void getedu()
         {
         cout << "   Enter name of school or university: ";
         cin >> school;
         cout << "   Enter highest degree earned \n";
         cout << "   (Highschool, Bachelor's, Master's, PhD): ";
         cin >> degree;
         }
      void putedu() const
         {
         cout << "\n   School or university: " << school;
         cout << "\n   Highest degree earned: " << degree;
         }
   };
////////////////////////////////////////////////////////////////
class employee
   {
   private:
      string name;             //employee name
      unsigned long number;    //employee number
   public:
      void getdata()
         {
         cout << "\n   Enter last name: "; cin >> name;
         cout << "   Enter number: ";      cin >> number;
         }
      void putdata() const
         {
         cout << "\n   Name: " << name;
         cout << "\n   Number: " << number;
         }
   };
////////////////////////////////////////////////////////////////
class manager                  //management
   {
   private:
      string title;            //"vice-president" etc.
      double dues;             //golf club dues
      employee emp;            //object of class employee
      student stu;             //object of class student
   public:
      void getdata()
         {
         emp.getdata();
         cout << "   Enter title: ";          cin >> title;
         cout << "   Enter golf club dues: "; cin >> dues;
         stu.getedu();
         }
      void putdata() const
         {
         emp.putdata();
         cout << "\n   Title: " << title;
         cout << "\n   Golf club dues: " << dues;
         stu.putedu();
         }
   };
////////////////////////////////////////////////////////////////
class scientist                //scientist
   {
   private:
      int pubs;                //number of publications
      employee emp;            //object of class employee
      student stu;             //object of class student
   public:
      void getdata()
         {
         emp.getdata();
         cout << "   Enter number of pubs: "; cin >> pubs;
         stu.getedu();
         }
      void putdata() const
         {
         emp.putdata();
         cout << "\n   Number of publications: " << pubs;
         stu.putedu();
         }
   };
////////////////////////////////////////////////////////////////
class laborer                  //laborer
   {
   private:
      employee emp;            //object of class employee
   public:
      void getdata()
         { emp.getdata(); }
      void putdata() const
         { emp.putdata(); }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   manager m1;
   scientist s1, s2;
   laborer l1;

   cout << endl;
   cout << "\nEnter data for manager 1";    //get data for
   m1.getdata();                            //several employees

   cout << "\nEnter data for scientist 1";
   s1.getdata();

   cout << "\nEnter data for scientist 2";
   s2.getdata();

   cout << "\nEnter data for laborer 1";
   l1.getdata();

   cout << "\nData on manager 1";           //display data for
   m1.putdata();                            //several employees

   cout << "\nData on scientist 1";
   s1.putdata();

   cout << "\nData on scientist 2";
   s2.putdata();

   cout << "\nData on laborer 1";
   l1.putdata();
   cout << endl;
   return 0;
   }
```

Virtual functions

Normal functions access with pointers (early binding)

```
// notvirt.cpp
// normal functions accessed from pointer
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Base                        //base class
   {
   public:
      void show()                 //normal function
         { cout << "Base\n"; }
   };
////////////////////////////////////////////////////////////////
class Derv1 : public Base        //derived class 1
   {
   public:
      void show()
         { cout << "Derv1\n"; }
   };
////////////////////////////////////////////////////////////////
class Derv2 : public Base        //derived class 2
   {
   public:
      void show()
         { cout << "Derv2\n"; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Derv1 dv1;           //object of derived class 1
   Derv2 dv2;           //object of derived class 2
   Base* ptr;           //pointer to base class

   ptr = &dv1;          //put address of dv1 in pointer
   ptr->show();         //execute show()

   ptr = &dv2;          //put address of dv2 in pointer
   ptr->show();         //execute show()
   return 0;
   }
```

<u>Output</u>

```
Base
Base
```

<u>Virtual functions accessed with pointers (late binding)</u>

```
// virt.cpp
// virtual functions accessed from pointer
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Base                         //base class
   {
   public:
      virtual void show()          //virtual function
         { cout << "Base\n"; }
   };
////////////////////////////////////////////////////////////////
class Derv1 : public Base          //derived class 1
   {
   public:
      void show()
         { cout << "Derv1\n"; }
   };
////////////////////////////////////////////////////////////////
class Derv2 : public Base          //derived class 2
   {
   public:
      void show()
         { cout << "Derv2\n"; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   Derv1 dv1;             //object of derived class 1
   Derv2 dv2;             //object of derived class 2
   Base* ptr;             //pointer to base class

   ptr = &dv1;            //put address of dv1 in pointer
   ptr->show();           //execute show()

   ptr = &dv2;            //put address of dv2 in pointer
   ptr->show();           //execute show()
   return 0;
   }
```

<u>Output</u>

```
Derv1
Derv2
```

Abstract class and pure virtual functions

```
// virtpure.cpp
// pure virtual function
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class Base                          //base class
   {
   public:
      virtual void show() = 0;     //pure virtual function
   };
////////////////////////////////////////////////////////////////
class Derv1 : public Base           //derived class 1
   {
   public:
      void show()
         { cout << "Derv1\n"; }
   };
////////////////////////////////////////////////////////////////
class Derv2 : public Base           //derived class 2
   {
   public:
      void show()
         { cout << "Derv2\n"; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
// Base bad;                  //can't make object from abstract class
   Base* arr[2];              //array of pointers to base class
   Derv1 dv1;                 //object of derived class 1
   Derv2 dv2;                 //object of derived class 2

   arr[0] = &dv1;             //put address of dv1 in array
   arr[1] = &dv2;             //put address of dv2 in array

   arr[0]->show();            //execute show() in both objects
   arr[1]->show();
   return 0;
   }
```

Virtual functions and Polymorphism

```
// virtpers.cpp
// virtual functions with person class
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class person                           //person class
   {
   protected:
      char name[40];
   public:
      void getName()
         { cout << "   Enter name: "; cin >> name; }
      void putName()
         { cout << "Name is: " << name << endl; }
      virtual void getData() = 0;         //pure virtual func
      virtual bool isOutstanding() = 0;   //pure virtual func
   };
////////////////////////////////////////////////////////////////
class student : public person        //student class
   {
   private:
      float gpa;                //grade point average
   public:
      void getData()            //get student data from user
         {
         person::getName();
         cout << "   Enter student's GPA: "; cin >> gpa;
         }
      bool isOutstanding()
         { return (gpa > 3.5) ? true : false; }
   };
////////////////////////////////////////////////////////////////
class professor : public person      //professor class
   {
   private:
      int numPubs;               //number of papers published
   public:
      void getData()             //get professor data from user
         {
         person::getName();
         cout << "   Enter number of professor's publications: ";
         cin >> numPubs;
         }
      bool isOutstanding()
         { return (numPubs > 100) ? true : false; }
   };
////////////////////////////////////////////////////////////////
int main()
   {
   person* persPtr[100];      //array of pointers to persons
   int n = 0;                 //number of persons on list
   char choice;

   do {
      cout << "Enter student or professor (s/p): ";
      cin >> choice;
      if(choice=='s')                  //put new student
         persPtr[n] = new student;     //   in array
      else                             //put new professor
         persPtr[n] = new professor;   //   in array
      persPtr[n++]->getData();         //get data for person
      cout << "   Enter another (y/n)? ";  //do another person?
      cin >> choice;
      } while( choice=='y' );          //cycle until not 'y'

   for(int j=0; j<n; j++)              //print names of all
      {                                //persons, and
      persPtr[j]->putName();           //say if outstanding
      if( persPtr[j]->isOutstanding() )
         cout << "   This person is outstanding\n";
      }
   return 0;
   }  //end main()
```

Hybrid Inheritance and virtual base 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.

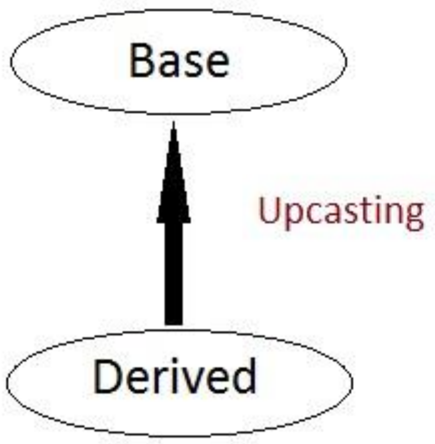
```
class B : virtual public A {};

class C : virtual public A {};

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

Now by adding virtual keyword, we tell compiler to call any one out of the two show() functions.

```
// normbase.cpp
// ambiguous reference to base class

class Parent
   {
   protected:
      int basedata;
   };
class Child1 : public Parent
   { };
class Child2 : public Parent
   { };
class Grandchild : public Child1, public Child2
   {
   public:
      int getdata()
         { return basedata; }   // ERROR: ambiguous
   };

// virtbase.cpp
// virtual base classes

class Parent
   {
   protected:
      int basedata;
   };
class Child1 : virtual public Parent   // shares copy of Parent
   { };
class Child2 : virtual public Parent   // shares copy of Parent
   { };
class Grandchild : public Child1, public Child2
   {
   public:
      int getdata()
         { return basedata; }    // OK: only one copy of Parent
   };
```

Upcasting in C++

Upcasting is using the Super class's reference or pointer to refer to a Sub class's object. Or we can say that, the act of converting a Sub class's reference or pointer into its Super class's reference or pointer is called upcasting.



The opposite of upcasting is d**owncasting**, in which we convert Super class's reference or pointer into derived class's reference or pointer.

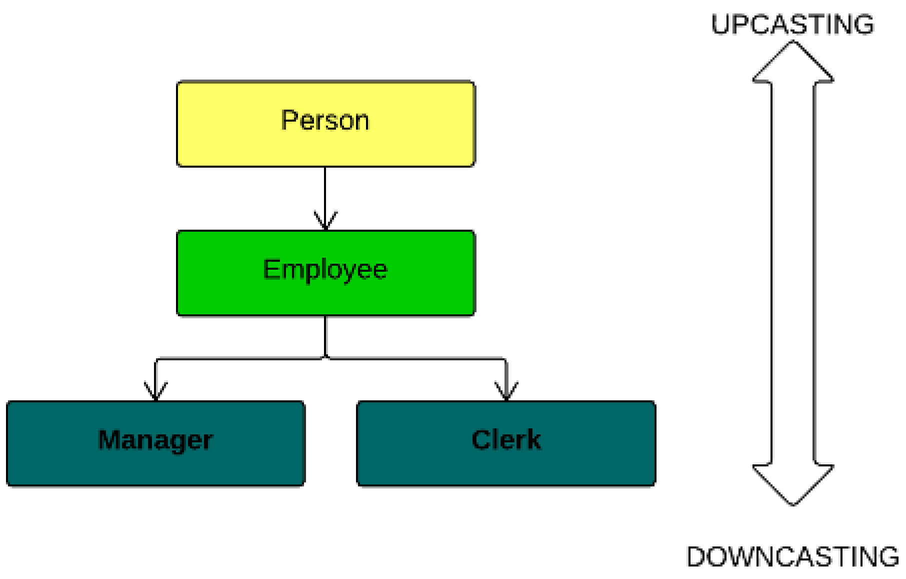
Upcasting and downcasting are an important part of C++. Upcasting and downcasting gives a possibility to build complicated programs with a simple syntax. It can be achieved by using Polymorphism.

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

We will use the following hierarchy of classes:



As you can see, Manager and Clerk are both Employee. They are both Person too. What does it mean? It means that Manager and Clerk classes inherit properties of Employee class, which inherits properties of Person class.

For example, we don't need to specify that 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();
   //correction: cout<<((Manager* )emp)->getComm();
   congratulate(&c1);
   congratulate(&m1);

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

Output

```
Steve
3000
Happy Birthday!!!
First Name: Kevin Last Name: Jones Salary: 1200
Happy Birthday!!!
First Name: Steve Last Name: Kent Salary: 3200
Manager of Kevin is Steve
```

Both upcasting and downcasting do not change object by itself. When you use upcasting or downcasting you just "label" an object in different ways.

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:

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

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:

```
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();
  //correction: cout<<((Manager* )emp)->getComm();

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

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

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);
};
```

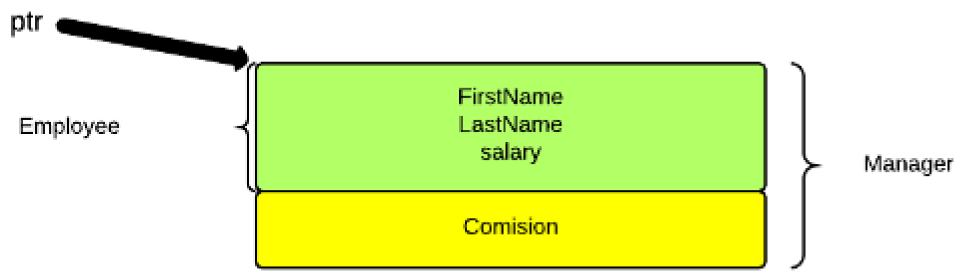
This function will work with all the classes that are derived from the Employee class. When you call it with objects of type Manager and Person, they will be automatically upcasted to Employee class:

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

Memory layout

As you know, derived class extends properties of the base class. It means that derived class has properties (data members and member functions) of the base class and defines new data members and member functions.

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);
```

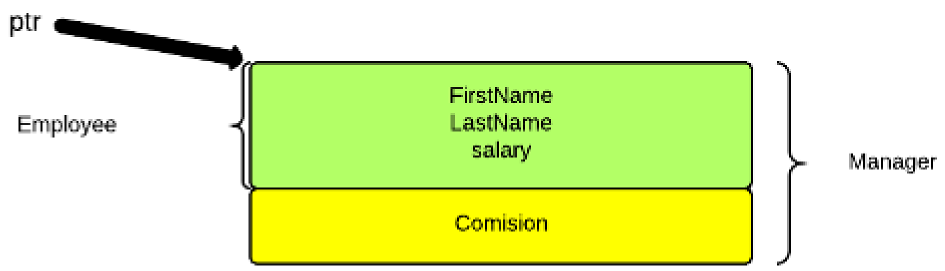
This code compiles and runs without any problem, because emp points to an object of Manager class.

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?

```
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 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 following 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 test() {}
```

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.

Runtime Type Information (RTTI)

Runtime Type Information (RTTI) is the concept of determining the type of any variable during execution (runtime.)

The RTTI mechanism contains:

- The operator dynamic_cast
- The operator typeid
- The struct type_info

RTTI can only be used with polymorphic types. This means that with each class you make, you must have at least one virtual function (either directly or through inheritance.)

Compatibility note: On some compilers you have to enable support of RTTI to keep track of dynamic types. So to make use of dynamic_cast (see next section) you have to enable this feature. See you compiler documentation for more detail.

Dynamic_cast

The dynamic_cast can only be used with pointers and references to objects. It makes sure that the result of the type conversion is valid and complete object of the requested class. This is way a dynamic_cast will always be successful if we use it to cast a class to one of its base classes. Take a look at the example:

```
        class Base_Class { };
        class Derived_Class: public Base_Class { };

        Base_Class a; Base_Class * ptr_a;
        Derived_Class b; Derived_Class * ptr_b;

        ptr_a = dynamic_cast<Base_Class *>(&b);
        ptr_b = dynamic_cast<Derived_Class *>(&a);
```

The first dynamic_cast statement will work because we cast from derived to base. The second dynamic_cast statement will produce a compilation error because base to derived conversion is not allowed with dynamic_cast unless the base class is polymorphic.

If a class is polymorphic then dynamic_cast will perform a special check during execution. This check ensures that the expression is a valid and complete object of the requested class.
Take a look at the example:

```
// dynamic_cast
#include <iostream>
#include <exception>
using namespace std;

class Base_Class { virtual void dummy() {} };
class Derived_Class: public Base_Class { int a; };

int main () {
  try {
        Base_Class * ptr_a = new Derived_Class;
        Base_Class * ptr_b = new Base_Class;
        Derived_Class * ptr_c;

        ptr_c = dynamic_cast< Derived_Class *>(ptr_a);
        if (ptr_c ==0) cout << "Null pointer on first type-cast" << endl;

        ptr_c = dynamic_cast< Derived_Class *>(ptr_b);
        if (ptr_c ==0) cout << "Null pointer on second type-cast" << endl;

        } catch (exception& my_ex) {cout << "Exception: " << my_ex.what();}
  return 0;
}
```

In the example we perform two dynamic_casts from pointer objects of type Base_Class* (namely ptr_a and ptr_b) to a pointer object of type Derived_Class*.

If everything goes well then the first one should be successful and the second one will fail. The pointers ptr_a and ptr_b are both of the type Base_Class. The pointer ptr_a points to an object of the type Derived_Class. The pointer ptr_b points to an object of the type Base_Class. So when the dynamic type cast is performed then ptr_a is pointing to a full object of class Derived_Class, but the pointer ptr_b points to an object of class Base_Class. This object is an incomplete object of class Derived_Class; thus this cast will fail!

Because this dynamic_cast fails a null pointer is returned to indicate a failure. When a reference type is converted with dynamic_cast and the conversion fails then there will be an exception thrown out instead of the null pointer. The exception will be of the type bad_cast.

With dynamic_cast it is also possible to cast null pointers even between the pointers of unrelated classes. Dynamic_cast can cast pointers of any type to void pointer(void*).

Typeid and typ_info

If a class hierarchy is used then the programmer doesn’t have to worry (in most cases) about the data-type of a pointer or reference, because the polymorphic mechanism takes care of it. In some cases the programmer wants to know if an object of a derived class is used. Then the programmer can make use of dynamic_cast. (If the dynamic cast is successful, then the pointer will point to an object of a derived class or to a class that is derived from that derived class.) But there are circumstances that the programmer (not often) wants to know the prizes data-type. Then the programmer can use the typeid operator.

The typeid operator can be used with:

- Variables
- Expressions
- Data-types

Take a look at the typeid example:

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

int main ()
{
        int * a;
        int b;

        a=0; b=0;
        if (typeid(a) != typeid(b))
        {
                cout << "a and b are of different types:\n";
                cout << "a is: " << typeid(a).name() << '\n';
```

```
                    cout << "b is: " << typeid(b).name() << '\n';
               }
               return 0;
          }
```

Note: the extra header file typeinfo.

The result of a typeid is a const type_info&. The class type_info is part of the standard C++ library and contains information about data-types. (This information can be different. It all depends on how it is implemented.)

A bad_typeid exception is thrown by typeid, if the type that is evaluated by typeid is a pointer that is preceded by a dereference operator and that pointer has a null value.

Virtual Destructors

Destructors in the Base class can be Virtual. Whenever upcasting is done, Destructors of the Base class must be made virtual for proper destruction of the object when the program exits.

NOTE : Constructors are never Virtual, only Destructors can be Virtual.

Upcasting without Virtual Destructor

Let's first see what happens when we do not have a virtual Base class destructor.

```
class Base
{
 public:
 ~Base() {cout << "Base Destructor\t"; }
};

class Derived:public Base
{
 public:
 ~Derived() { cout<< "Derived Destructor"; }
};


int main()
{
 Base* b = new Derived;     //upcasting
 delete b;
}
```

Output : Base Destructor

In the above example, **delete b** will only call the Base class destructor, which is undesirable because, then the object of Derived class still remains because its destructor is never called. Which results in memory leak.

Upcasting with Virtual Destructor

Now let’s see. what happens when we have Virtual destructor in the base class.

```
class Base
{
 public:
 virtual ~Base() {cout << "Base Destructor\t"; }
};

class Derived:public Base
{
 public:
 ~Derived() { cout<< "Derived Destructor"; }
};

int main()
{
 Base* b = new Derived;     //upcasting
 delete b;
}
```

Output :

```
Derived Destructor
Base Destructor
```

When we have Virtual destructor inside the base class, then first Derived class's destructor is called and then Base class's destructor is called.