AMBITION GURU

Object-Oriented programming in C++ (CSIT 202)

II semester, BScCSIT

Compiled by Ankit Bhattarai
Ambition Guru College

Compiled by ab

Syllabus

AMBITION GURU

Unit	Contents	Hours	Remarks
1.	Introduction to C++ and OOP	3	
2.	Classes and objects	7	
3.	Operator Overloading & Type Conversion	7	
4.	Inheritance	7	
5.	**Polymorphism and Virtual Functions**	**7**	
6.	Templates and Exception Handling	7	
7.	I/O Stream	8	

Practical Works

Credit hours : 3

AMBITION GURU

Unit 5
Polymorphism and Virtual Functions
(7 hrs.)

Function overloading, Function overriding, Pointers to objects, Virtual functions and late binding, Pure virtual functions and abstract classes.

AMBITION GURU

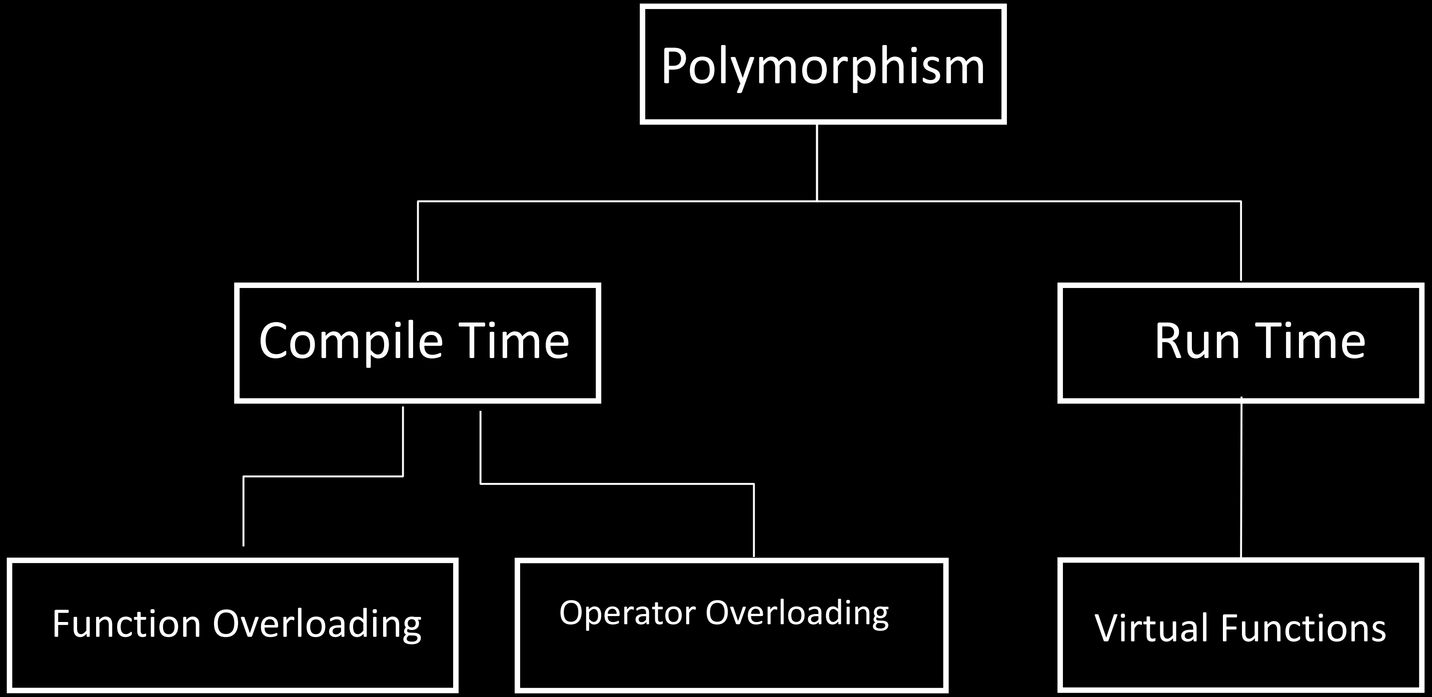


Fig: Types of polymorphism in C++

Compiled by ab

AMBITION GURU

Polymorphism

- **Polymorphism** means ability to take more than one form. There are two types of polymorphism, compile-time polymorphism and run-time polymorphism.
- Function overloading and operator overloading shows compile time polymorphism whereas run-time polymorphism is achieved through the use of virtual function.

AMBITION GURU

Compile time Polymorphism

- In **compile time polymorphism**, the compiler at the compile time know all the matching arguments, therefore the compiler is able to select the appropriate function for a particular function call at the compile time itself.
- Sometimes compile time polymorphism is also called as ***early binding*** *or* ***static binding*** *or* ***static linking.***
- Early binding simply means that an object is bound to its function call at compile time.

Function Overloading

- More than one functions having same name but differs either in number of arguments or type of arguments or both is said to be function overloading.
- Function overloading shows the compile time polymorphism because the particular function call and its associated function definition is known by the compiler before the running of the program i.e.at the compile time only.

Advantages of function overloading:

- Remembering names is easier
- Lesser complexity
- Increases readability

WAP to find the volume of a cube, cuboid and cylinder using the concept of functionoverloading.

AMBITION GURU

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

void volume(float l)
{
    cout<<"The volume of the cube is:"<<l*l*l<<endl;
}

void volume(float l,float b,float h)
{
    cout<<"The volume of the cuboid is:"<<l*b*h<<endl;
}

void volume(float r, float h)
{
    cout<<"The volume of the cylinder is:"<<3.14*r*r*h;
}
```

AMBITION GURU

```
int main( )
{
    volume(3.5);
    volume(3.5,6.5,9.5);
    volume(3.5,5.5);
    return 0;
}
```

Output:

```
The volume of the cube is:42.875
The volume of the cuboid is:216.125
The volume of the cylinder is:211.558
```

AMBITION GURU

WAP to find the cube of a integer , float and double number using the concept of function overloading(passing single argument to the function)

WAP to find the cube of a integer , float and double number using the concept of function overloading(passing single argument to the function)

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

void findcube(int a){
   cout<<"The cube of the integer number is:"<<a*a*a<<endl;
}

void findcube(float b)
{
   cout<<"The cube of the float number is:"<<b*b*b<<endl;
}

void findcube(double c)
{
   cout<<"The cube of the double number is:"<<c*c*c;
}
```

```
int main( )
{
   int x;
   float y;
   double z;
   cout<<"Enter values for int, float double type variables respectively:";
   cin>>x>>y>>z;
   findcube(x);
   findcube(y);
   findcube(z);
   return 0;
}
```

Output:

```
Enter values for int, float double type variables respectively:2
3.5
4.5e+7
The cube of the integer number is:8
The cube of the float number is:42.875
The cube of the double number is:9.1125e+22
```

AMBITION GURU

Operator Overloading
(Read from previous unit notes)

Run time Polymorphism

- **Run time polymorphism** means, the code associated with particular function call is known by the compiler only during the run of program.
- Sometimes run time polymorphism is also known as ***late binding*** *or* ***dynamic binding***. Run-time polymorphism indicates the form of a member function that can be changed at runtime.
- In C++ run time polymorphism is achieved with the help of **virtual function, class hierarchies(function overriding) and base class pointer pointing to derived class.**

Compiled by ab

AMBITION GURU

Function Overriding

(Read from previous unit Inheritance notes)

Question:
Define function overriding. What are the rules for function overriding. Explain with an example.

Compiled by ab

AMBITION GURU

Pointer to base class and derived class

Question:
Define pointer in C++. How do you initialize a pointer and access the value stored in it? WAP to illustrate your answer.

Compiled by ab

Pointer to base class and derived class:

- We can create both the pointer objects of base class as well as the pointer object of a derived class.
- As the properties of base class are inherited in a derived class, we can use the *pointer object of base class to point the object of a derived class.*
- However, the reverse is not true because the inheritance do not work in reverse order.
- The base class pointer object which is pointing to derived class object can access only the members that are inherited from base class.
- The derived class pointer object which is pointing to the derived class object can access all the features of derived class.

Compiled by ab

- Thus, we can conclude that the accessibility of the pointer depends upon the type of the pointer(base or derived) rather than the object to which it is pointing.

```
#include<iostream>
using namespace std;
class Base
{
    public:
        int m;
    void display( )
    {
        cout<<"The value of m="<<m<<endl;
    }
};
```

```
class Derived:public Base
{
    public:
        int d;
    void display( )
    {
        cout<<"The value of m and d="<<m<<","<<d<<endl;
    }
};
```

```
int main( )
{
    Base *bptr;
    Derived D;
    bptr=&D;
    bptr->m=10;
    //bptr->d=20; error
    bptr->display( );
    Derived *dptr;
    dptr=&D;
    dptr->d=20;
    dptr->display( );
    return 0;
}
```

AMBITION GURU

Output:

```
The value of m=10
The value of m and d=10,20
```

Note:

1. Base class pointer can point to a derived class object (can access only members of Base class)
2. Derived class pointer cannot point to a base class object directly

AMBITION GURU

Need of virtual function

Need of virtual function:

- We have seen earlier that the accessibility of the pointer depends upon the type of the pointer rather than the object to which it is pointing.
- In order to access the function depending upon the object to which it is being pointed to by the pointer object virtual function is needed.
- Virtual function is needed in a class to achieve the run-time polymorphism i.e. if virtual function is used then the compiler knows the appropriate version of the same functions looking to the object being pointed to by base class pointer object during the run of the program.

AMBITION GURU

Compiled by ab

Need of virtual function:

- When we use the same function name in both the base and the derived classes, the function in base class is declared as virtual using the keyword virtual preceding its normal declaration.
- When the function is made virtual , C++ determines which function to use at run time based on the type of object pointed by the base pointer, rather than the type of the pointer. Thus, by making the base pointer to point to different objects we can execute different versions of the virtual function.

AMBITION GURU

Compiled by ab

- Virtual function is initially defined in the base class and is redefined in each derived classes which are derived from base class. The keyword virtual is optional in each derived class.

Syntax:

```
class class_name
{
// protected data_members of a class;
        public:
        virtual return_type function_name( ) //virtual function
        {
                        //body of the virtual function;
        }
};
```

Properties/ rules for virtual function:

AMBITION GURU

- The virtual functions must be members of some class.
- They cannot be static members.
- They are accessed by using object pointers.
- A virtual function can be a friend of another class.
- A virtual function in a base class must be defined, even though it may not be used.
- The prototypes of the base class version of a virtual function and all the derived class versions must be identical. If two functions with the same name have different prototypes, C++ considers them as overloaded functions, and the virtual function mechanism is ignored.
- We cannot have virtual constructors, but we can have virtual destructors.

AMBITION GURU

Write a program that shows the runtime polymorphism in C++

//Write a program that shows the runtime polymorphism in C++:

AMBITION GURU

```
#include<iostream>
using namespace std;
class Animal
{
    public:
    virtual void display( )   //Virtual function
    {
        cout<<"Animal class:"<<endl;
    }
};
```

```
class Cow:public Animal
{
public:
     void display( )     //virtual function redefined in derived
     {
          cout<<"Cow class:"<<endl;
     }
};
```

```
class Dog: public Animal
{
public:
void display( )
{
cout<<"Dog class:"<<endl;
}
};
```

Output:

```
Animal class:
Cow class:
Dog class:
```

```
int main( )
{
Animal *Anm;
Animal a1;
Cow c1;
Dog d1;
Anm=&a1;
Anm->display( );
Anm=&c1;
Anm->display( );
Anm=&d1;
Anm->display( );
return 0;
}
```

AMBITION GURU

Array of pointer object of Base class

Array of pointer object of Base class:

- We can also make array of pointer object of base class and access the different version of same function according to the object being pointed to by the base class pointer object.

```
#include<iostream>
using namespace std;
class Animal
{
public:
virtual void display( ) //Virtual function
{
cout<<"Animal class:"<<endl;
}
};
```

Compiled by ab

```
class Cow: public Animal
{
public:
void display( )//virtual function redefined in derived
{
cout<<"Cow class:"<<endl;
}
};

class Dog: public Animal
{
public:
void display( )
{
     cout<<"Dog class:"<<endl;
}
};
```

```
class Rabbit:public Animal
{
public:
void display( )
{
cout<<"Rabbit class:"<<endl;
}
};
```

Output:

```
Animal class:
Cow class:
Dog class:
Rabbit class:
```

```
int main( )
{
Animal a1;
Cow c1;
Dog d1;
Rabbit r1;
Animal *Anm[ ]={&a1,&c1,&d1,&r1};
for(int i=0;i<4;i++)
Anm[i]->display( );
return 0;
}
```

AMBITION GURU

Compiled by ab

Pure Virtual Function

Pure Virtual Function

AMBITION GURU

- Virtual function of the form , ***virtual return_type function_name( ) = 0***; is said to be pure virtual function.
- Pure virtual function is also called as do-nothing function.
- A pure virtual function is a function declared in a base class that has no body(definition).
- The difference between a virtual and pure virtual function is that a virtual function has an implementation with minimal functionality for the objects and gives the derived class the option of overriding the function however a *pure virtual function does not provide an implementation and requires the derived class to override the function*.

```
class Polygon
{
    public:
        virtual void area()=0;   //pure virtual function
    .
    .
    .
};
```

A pure virtual function has the following properties.

1. A pure virtual function has no implementation in the base class.
2. It acts as an empty bucket(virtual function is partially filled bucket) that the derived class are supposed to fill it.

AMBITION GURU

Abstract Class

Abstract class:

- The class containing *at least one pure virtual function* is called an abstract base class or simply abstract class .
- The object of the abstract class cannot be created, only the pointer object can be created. When we will never want to instantiate(create) objects of a base class we call it an abstract class.
- In above, Polygon become abstract since there is presence of pure virtual function area( ).
- The expression = 0 has no any other meaning, the equal sign = 0 does not assign 0 to function area( ). It is only method to tell the compiler that area( ) is pure virtual function hence class polygon is abstract class.

- Contrary to this all classes derived from abstract class and implement pure virtual function of abstract class are called **concrete class.**
- The main objective of an abstract base class is to provide some traits to the derived classes and to create a base pointer required for achieving run time polymorphism(Such a class exists only to act as a parent of derived class). When a virtual function is made pure every derived class must provide its own definition.

```
//Example program:
#include<iostream>
using namespace std;
class polygon //Abstract class
{
protected:
float length,height;
```

```
public:
void input( )
{
cout<<"Enter the values to the length and height:"<<endl;
cin>>length>>height;
}
virtual void area( )=0; //pure virtual function
};

class rectangle: public polygon{
public:
void display( )
{
     cout<<"The length of the rectangle="<<length<<endl;
     cout<<"The height of the rectangle="<<height<<endl;
}
```

```
void area( )
{
cout<<"The area of the rectangle="<<length*height<<endl;
}
};

class triangle: public polygon
{
public:
void display( ){
cout<<"The length of the triangle="<<length<<endl;
cout<<"The height of the triangle="<<height<<endl;
}
void area( )
{
cout<<"The area of the triangle="<<0.5*length*height<<endl;
}
};
```

```
int main( )
{
polygon *pg;
rectangle rec;
rec.input( );
rec.display( );
pg=&rec;
pg->area( );
triangle tg;
tg.input( );
tg.display( );
pg=&tg;
pg->area( );
return 0;
}
```

Output:

```
Enter the values to the length and height:
4
5
The length of the rectangle=4
The height of the rectangle=5
The area of the rectangle=20
Enter the values to the length and height:
5
6
The length of the triangle=5
The height of the triangle=6
The area of the triangle=15
```

Virtual destructor:

- There may arise a situation that the destructor need to be made as virtual as if the dynamic memory allocation is performed in base and derived class then the allocated memory need to be freed usingdelete operation in destructor.
- If the base class pointer object is pointing to derived class object and if the base class destructor is made as non-virtual and if we delete base class pointer object by using the delete operator, then only the destructor from the base class is called. So, to call destructors from the base class as well as from the derived class thebase class destructor should be made as virtual.

- Since destructors are member functions, they can be made virtual with placing keyword virtual before it. The syntax is

```
virtual~classname( ); //virtual destructor.

  #include<iostream>
  using namespace std;
  class Base
  {
  public:
  ~Base( ) //non-virtual destructor
  {
      cout<<"Base destroyed"<<endl;
  }
  };
```

```
class Derv:public Base
{
public:
~Derv( )
{
    cout<<"Derv destroyed"<<endl;
}
};
int main( )
{
Base *pBase=new Derv; //Dynamic memory allocation
delete pBase;
return 0;
}
```

Output:

Base destroyed

```
#include<iostream>
using namespace std;
class Base
{
     public:
      virtual~Base( ) //Virtual destructor
      {
            cout<<"Base destroyed"<<endl;
      }
};
```

```
class Derv:public Base
{
   public:
~Derv( )
{
    cout<<"Derv destroyed"<<endl;
}
};
int main( )
{
Base *pBase=new Derv; //Dynamic memory allocation
delete pBase;
return 0;
}
```

Output:

```
Derv destroyed
Base destroyed
```

AMBITION GURU

Explanations

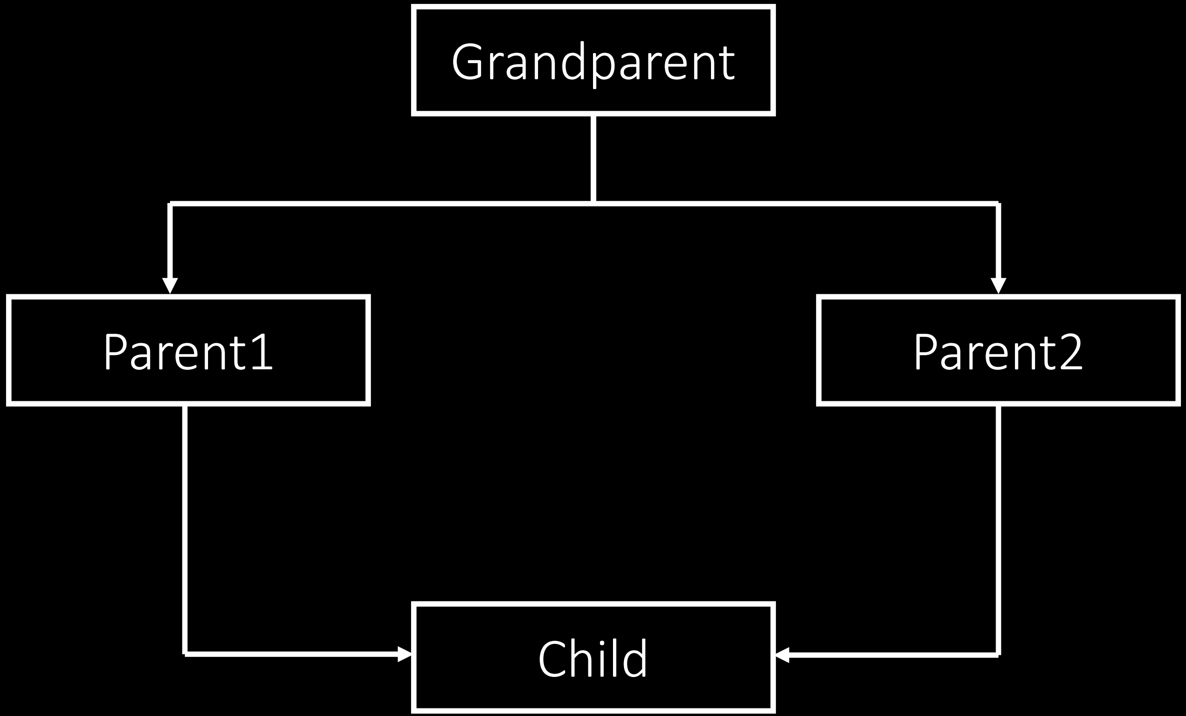
- This shows that the destructor for the Derv part of the object isn’t called, because the base class destructor is not virtual, but if we make it virtual then both derived and base class destructor is called respectively.
- To ensure that derived –class objects are destroyed properly, you should make destructors in all base classes as virtual.

AMBITION GURU

Virtual Base Class

Virtual Base Classes

- When two or more classes are derived from a common base class, we can prevent multiple copies of the base class being present in a class derived from those classes by declaring the base class as virtual when it is being inherited. Such a base class is known as virtual base class.
- Consider the situation shown below, with a base class, Grandparent; two derived classes,Parent1 and Parent2; and a fourth class, child, derived from both Parent1 and Parent2.
- Due to this the properties of Grandparent class will be the duplicate copies to a derived class child via Parent1 and Parent2.
- In this arrangement a problem can arise if a member function in the child class wants to access data or functions in the Grandparent class.


Grandparent
Parent1
Parent2
Child

- In order to solve this problem, the concept of virtual base class is used i.e. we make ancestor base class (grandparent class) as virtual while deriving Parent1 and Parent2.
- After making ancestor class as virtual base class we can be sure that only the single copy of the ancestor base class will be inherited to the finally derived class child.

General syntax:

```
class base

{ …. };

class derived1:virtual visibility_mode base

{ …. }:

class derived2:virtual visibility_mode base

{ …. };

class derived: visibility_mode derived1,visibility_mode derived2

{ …. };
```

Example:

```
class Grandparent

{ …. };

class Parent1:virtual public Grandparent

{ …. }:

class Parent2:virtual public Grandparent

{ …. };

class Child: public Parent1,public Parent2

{ …. };
```

AMBITION GURU

```
//Example Program:
#include<iostream>
using namespace std;
class Base
{
    protected:
        int w;
    public:
        void get_w( )
        {
            cout<<"Enter the value to w of a Base class:"<<endl;
            cin>>w;
        }
```

```
          void display_w( )
          {
               cout<<"The value to w of a Base class="<<w<<endl;
          }
};

class derived1: virtual public Base
{
     protected:
          int x;
     public:
          void get_x( )
          {
               cout<<"Enter the value to the x of derived1:"<<endl;
               cin>>x;
          }
```

```
void display_x( )
{
cout<<"The value to the x of a derived1="<<x<<endl;
}
};

class derived2:public virtual Base
{
protected:
int y; public:
void get_y( )
{
cout<<"Enter the value to the y of a derived2:"<<endl;
cin>>y;
}
```

```
void display_y( )
{
cout<<"The value to the y of a derived2="<<y<<endl;
}
};
class derived: public derived1,public derived2
{
protected:
int z;
public:
void get_z( )
{
cout<<"Enter the value to the z of a derived class:"<<endl;
cin>>z;
}
```

```
void display_z( )
{
cout<<"The value to the z of a derived class="<<z<<endl;
}
void display( )
{
cout<<"The sum is:"<<w+x+y+z<<endl
}
};

int main( )
{
derived d;
```

```
d.get_w( );
/*if no virtual keyword was used then this statement will cause error and it should be
called as either d.derived1::get_w( ) or d.derived2::get_w( );*/
d.get_x( );
d.get_y( );
d.get_z( );
d.display_w( );
/* if no virtual keyword was used then this statement will cause error and it should be
called as either d.derived1::display_w( ) or d.derived2::display_w( );*/
d.display_x( );
d.display_y( );
d.display_z( );
d.display( );
return 0;
}
```

Output:

```
Enter the value to w of a Base class:
3
Enter the value to the x of derived1:
4
Enter the value to the y of a derived2:
5
Enter the value to the z of a derived class:
8
The value to w of a Base class=3
The value to the x of a derived1=4
The value to the y of a derived2=5
The value to the z of a derived class=8
The sum is:20
```

Questions

AMBITION GURU

1. Difference between compile time and run time polymorphism.
2. Differentiate between function overloading and function overriding. Give example of each.
3. WAP to find the cube of an integer , float and double number using the concept of function overloading(passing single argument to the function)
4. Differentiate between concrete class and abstract class.
5. Why do we need virtual function? Explain the reason for member function overriding when using virtual function.
6. Write short notes on: Early binding and late binding, Virtual function, Abstract class.
7. Define pointer in C++. How do you initialize a pointer and access the value stored in it? WAP to illustrate your answer.

AMBITION GURU

THANK YOU

Any Queries ?