Chapter 6:
Polymorphism and virtual functions

BIBHA STHAPIT
ASST. PROFESSOR
IOE, PULCHOWK CAMPUS

Polymorphism

- Having same name but different functionality
- Two categories:
  - **1.Compile time polymorphism**
    - Early or static binding
    - Static polymorphism refers to the binding of functions on the basis of their **signature** (number, type and sequence of parameters).
    - It is also called **early binding** because the calls are already bound to the proper type of functions during the compilation of the program depending upon type and sequence of parameters.
    - E.g. Function overloading, operator overloading

Polymorphism

- **2. Run time polymorphism**
  - Late or dynamic binding
  - A function is said to exhibit dynamic polymorphism if it exists in various forms, and the resolution to different function calls are made dynamically during execution time.
  - This feature makes the program more **flexible** as a function can be called, depending on the context.
  - E.g. Function overriding using Virtual Functions

Pointers to the derived class

- We can create pointer type object of both base class as well as derived class pointer.
- The pointer object of the derived class will always be type compatible with base class pointer.
- Base class pointer can invoke member function that is defined in base class only.
- If member functions are defined in derived class only, then these functions cannot be accessed from base pointer.

```
base *b1;
base b2;
derived d;
b1=&b2;
b1->display( );
b1=&d;
b1->display( );
```

```
class B
{
public:
void show()
{  cout<<"I am in base
show"<<endl;   }
};
class D:public B
{
public:
void show()
{   cout<<"I am in derived
show"<<endl;  }
void display()
{    cout<<"I am in derived only";  }
};
```

```
main()
{
B b, *bp[2];
D d;
bp[0]=&b;
bp[0] ->show();
bp[1] =&d;

bp[1] ->show();  // invokes overridden function of base class only
// bp[1] ->display(); cannot be invoked
D *dp;
dp=&d;
dp -> display();
}
```

Output:

I am in base show

I am in base show

I am in derived only

Need of virtual function

- If we use base class pointer to access the derived class member, then the function overriding cannot be done.
- That is, if the base class and derived class have same function , then only base class member function can be accessed through that pointer even if we assign the address of derived class to the base class pointer.
- Here, the function is associated only to the type of pointer but not the content of the pointer.
- If we want to invoke the function depending on object that is being pointed by the pointer type of object, we need to use **virtual function.**

Virtual function

- Virtual function is a non-static member function that is declared within a base class and redefined by a derived class
- Determines which function to execute during runtime based on type of object pointed by base pointer rather than type of pointer.
- To create a virtual function, precede the function's declaration in base class by keyword “virtual”
- For every base class that has one or more virtual functions, a table of function addresses is created during run time.
- This table of function addresses is called the **virtual table** that contains the address of each and every virtual function that has been defined in the corresponding class.

```
class B
{
public:
virtual void show()
{ cout<<"I am in base show"<<endl; }
void display()
{ cout<<"I am in base display"<<endl; }
};
class D:public B
{
public:
void show()
{ cout<<"I am in derived show"<<endl; }
void display()
{ cout<<"I am in derived
display"<<endl; }
};
```

```
main()
{
B b, *bp[2];
D d;
bp[0]=&b;
bp[1] =&d;

bp[0] ->display();
bp[1] ->display();

bp[0] ->show();
bp[1] ->show();
}
```

Output:

```
I am in base display
I am in base display
I am in base show
I am in derived show
```

"Virtual"ness is inherited

```
class B
{       public:
virtual void show()
{ cout<<"I am in base show"<<endl; }
};
class D1:public B
{       public:
void show()
{ cout<<"I am in derived1 show"<<endl; }
};
class D2:public D1
{       public:
void show()
{ cout<<"I am in derived2 show"<<endl; }
};
```

```
main()
{
B b, *bp[3];
D1 d;
D2 e;

bp[0]=&b;
bp[1] =&d;
bp[2] =&e;

bp[0] ->show();
bp[1] ->show();
bp[2] ->show();
}
```

Output:

I am in base show
I am in derived1 show
I am in derived2 show

Pure Virtual function & abstract class/abstract base class

- Virtual function without its definition in base class is known as “Pure Virtual Function”.
    - virtual void show() = 0;
- Do-nothing function
- The base class containing pure virtual function cannot be used to create objects, hence it is called abstract class or abstract base class.
- Abstract class is used to create base class pointers only for achieving runtime polymorphism.

```
class dimension
{   protected:
        int  l,b;
public:
    dimension(int x, int y): l(x),b(y){ }
    virtual void area()=0;
};

class square:public dimension
{
public:
square(int x):dimension(x,x){ }
void area()
{           cout<<"Area of square is "<<l*l<<endl;   }
};
```

```
class rectangle:public dimension
{
public:
rectangle(int x , int y):dimension(x,y){  }
void area()
{ cout<<"Area of rectangle is "<<l*b<<endl;  }
};

main()
{
    square s(5);
    rectangle r(10,2);
    dimension  *bp[2]={&s, &r};
    bp[0] ->area();
    bp[1] ->area();
}
```

Virtual Destructor

- Destructors are invoked automatically to free memory space
- But in derived classes, it is not invoked automatically because destructors that are non-virtual will not get message under late binding
- So, the destructors are made virtual to free space under late binding method.
- But the constructors cannot be virtual because virtual table would not have been created during object creation so that it would not have anywhere to look up to.

```
class base
{  public:
virtual void show()
{  cout<<"I am in base show"<<endl;  }
virtual ~base()
{   cout<<"I am in base destructor"<<endl;  }
};

class derived : public base
{  public:
void show()
{  cout<<"I am in derived show"<<endl; }
~derived()
{    cout<<"I am in derived destructor"<<endl; }
};
```

```
main()
{
base *bp = new base;
bp ->show();

bp=new derived;
bp ->show();
delete bp;
}
```

Output:

```
I am in base show
I am in derived show
I am in derived destructor
I am in base destructor
```

Run Time Type Information(RTTI)

- Provides information of object during runtime
- Available only for polymorphic class (class with virtual function).
- The “dynamic_cast” and “typeid” operators are used for these purpose
- Must include <typeinfo>header file for typeid operator

dynamic_cast operator

- The dynamic_cast operator is intended to be the most heavily used RTTI component.
- It doesn't answer the question of what type of object a pointer points to.
- Instead, it answers the question of whether you can safely assign the address of the object to a pointer of a particular type.
  - dynamic_cast<target_type> (expr)
- Two types of casting:
  - Upcasting – casting from derived to base
  - Downcasting – casting from base to derived (note: base pointer must hold the address of derived for successful casting)

```
class base
{
public:
    virtual void display(){ };
    void show()
   {       cout<<"Base Class"<<endl;   }
};
class derived : public base
{
public:
   void show()
   {      cout<<"Derived Class"<<endl;   }
};
```

```
main()
{   //UPCASTING
    base *bp;
    derived *dp;
    bp=dynamic_cast<base*>(dp);
    if(bp)
        cout<<"Upcasting successful"<<ends;
            bp->show();

    //DOWNCASTING
    base *bp1=new derived;
    derived *dp1;
    dp1=dynamic_cast<derived*>(bp1);
    if(dp1)
        cout<<"Downcasting successful"<<ends;
            dp1->show();
}
```

Output :

Upcasting successful Base Class

Downcasting successful Derived Class

typeid operator

- typeid is an operator which allows you to access the type of an object at runtime
- Can also be implemented for non-polymorphic class
- This is useful for pointers to derived classes

```
class Animal
{
public:
virtual void show()
{
cout<<"Animal Class"<<endl;
}
};
class Cat:public Animal
{
public:
void show()
{
cout<<"Cat Class"<<endl;
}
};
main()
{
Animal *ap=new Cat;
Cat c;
int roll;
float marks;
cout<<"Type of ap="<<typeid(ap).name()<<endl;
cout<<"Type of ap="<<typeid(*ap).name()<<endl;
cout<<"Type of c="<<typeid(c).name()<<endl;
cout<<"Type of roll="<<typeid(roll).name()<<endl;
cout<<"Type of marks="<<typeid(marks).name()<<endl;
}
```

Output:

```
Type of ap=P6Animal
Type of ap=3Cat
Type of c=3Cat
Type of roll=i
Type of marks=f
```