BAKI ALİ NEFT MƏKTƏBİ
BAKU HIGHER OIL SCHOOL

Programming and Computer Applications-2

Object-Oriented Programming: Polymorphism

Instructor : PhD, Associate Professor Leyla Muradkhanli

Polymorphism

- The process of representing one Form in multiple forms is known as **Polymorphism**. Here one form represent original form or original method always resides in base class and multiple forms represents overridden method which resides in derived classes.
- Polymorphism is derived from 2 greek words: **poly** and morphs. The word "poly" means many and **morphs** means forms. So polymorphism means many forms.

Type of Polymorphism

- Compile time polymorphism
- Run time polymorphism

Compile time Polymorphism

In C++ programming you can achieve compile time polymorphism in two way, which is given below;

- Method overloading
- Method overriding

Method Overloading in C++

Whenever same method name is exiting multiple times in the same class with different number of parameter or different order of parameters or different types of parameters is known as **method overloading**.

In below example method "sum()" is present in Addition class with same name but with different signature or arguments.

Example of Method Overloading in C++

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

class Addition
{
public:
void sum(int a, int b)
{
cout<<a+b;
}
```

Example of Method Overloading in C++

```
void sum(int a, int b, int c)
{
cout<<a+b+c;
}
};
int main()
{
Addition obj;
obj.sum(10, 20);
cout<<endl;
obj.sum(10, 20, 30);
cout<<endl;
}
```

Output

30

60

Method Overriding in C++

Define any method in both base class and derived class with same name, same parameters or signature, this concept is known as **method overriding**.

In below example same method "show()" is present in both base and derived class with same name and signature.

Example of Method Overriding in C++

```
#include <iostream>
#include <iomanip>
using namespace std;
class Base
{
 public:
 void show()
 {
  cout<<"Base class";
 }
};
```

Example of Method Overriding in C++

```
class Derived:public Base
{
 public:
 void show()
 {
  cout<<"Derived Class";
 }
};
```

Example of Method Overriding in C++

```
int main()
{
 Base b;          //Base class object
 Derived d;  //Derived class object
 b.show();
 cout<<endl;
 d.show();
 cout<<endl;
}
```

Output

Base class

Derived Class

Run time Polymorphism

In C++ Run time polymorphism can be achieve by using virtual function.

Virtual Function in C++

A **virtual function** is a member function of class that is declared within a base class and re-defined in derived class.

When you want to use same function name in both the base and derived class, then the function in base class is declared as virtual by using the **virtual** keyword and again re-defined this function in derived class without using virtual keyword.

Virtual Function in C++

Syntax

```
virtual return_type function_name()
{
 .......
 .......
}
```

Virtual Function Example

```
#include <iostream>
using namespace std;
class A
{
public:
 virtual void show()
 {
  cout<<"Base class \n";
 }
};
```

Virtual Function Example

```
class B : public A
{
 public:
  void show()
  {
   cout<<"Derive class \n";
 }
};
```

Virtual Function Example

```
int main()
{
A aobj;
B bobj;
A *ptr;
ptr=&aobj;
ptr->show();  // call base class function
ptr=&bobj;
ptr->show();  // call derive class function
}
```

Output

Base class
Derive class

Pointers to base class

```
// pointers to base class
#include <iostream>
using namespace std;
class Polygon {
  protected:
    int width, length;
  public:
    void set_values (int a, int b)
      { width=a; length=b; }
};
```

Pointers to base class

```
class Rectangle: public Polygon {
  public:
    int area()
      { return width*length; }
};
class Triangle: public Polygon {
  public:
    int area()
      { return width*length/2; }
};
```

Pointers to base class

```
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * poly1 = &rect;
  Polygon * poly2 = &trgl;
  poly1->set_values (4,5);
  poly2->set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0;
}
```

Output

20

10

Virtual members

A **virtual member** is a member function that can be redefined in a derived class, while preserving its calling properties through references. The syntax for a function to become virtual is to precede its declaration with the virtual keyword.

Virtual members

```
// virtual members
#include <iostream>
using namespace std;
class Polygon {
  protected:
    int width, length;
  public:
    void set_values (int a, int b)
      { width=a; length=b; }
    virtual int area ()
      { return 0; }
};
```

Virtual members

```
class Rectangle: public Polygon {
  public:
    int area ()
      { return width * length; }
};

class Triangle: public Polygon {
  public:
    int area ()
      { return (width * length / 2); }
};
```

```
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon poly;
  Polygon * poly1 = &rect;
  Polygon * poly2 = &trgl;
  Polygon * poly3 = &poly;
  poly1->set_values (4,5);
  poly2->set_values (4,5);
  poly3->set_values (4,5);
  cout << poly1->area() <<endl;
  cout << poly2->area() <<endl;
  cout << poly3->area() <<endl;
  return 0;
}
```

Output

20

10

0

Virtual members

In this example, all three classes (Polygon, Rectangle and Triangle) have the same members: width, height, and functions set_values and area.

The member function area has been declared as virtual in the base class because it is later redefined in each of the derived classes. Non-virtual members can also be redefined in derived classes, but non-virtual members of derived classes cannot be accessed through a reference of the base class: i.e., if virtual is removed from the declaration of area in the example above, all three calls to area would return zero, because in all cases, the version of the base class would have been called instead.