POLYMORPHISM

The word "polymorphism" means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

Types of Polymorphism

- **Compile-time Polymorphism.**
- **Runtime Polymorphism.**

Runtime Polymorphism

This type of polymorphism is achieved by **Function Overriding**. Late binding and dynamic polymorphism are other names for runtime polymorphism. The function call is resolved at runtime in runtime polymorphism. In contrast, with compile time polymorphism, the compiler determines which function call to bind to the object after deducing it at runtime.

Runtime Polymorphism

This type of polymorphism is achieved by **Function Overriding**. Late binding and dynamic polymorphism are other names for runtime polymorphism. The function call is resolved at runtime in runtime polymorphism. In contrast, with compile time polymorphism, the compiler determines which function call to bind to the object after deducing it at runtime.

A. Function Overriding

Function Overriding occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

```
class Parent
{
public:
        void GeeksforGeeks()
        {
        statements;
        }
};

class Child: public Parent
{
public:
        void GeeksforGeeks()
        {
        Statements;
        }
};

int main()
{
Child Child_Derived;
Child_Derived.GeeksforGeeks();
return 0;
}
```

Virtual Function

A virtual function (also known as virtual methods) is a member function that is declared within a base class and is re-defined (overridden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the method.

Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for the function call.

They are mainly used to achieve Runtime polymorphism.

Functions are declared with a virtual keyword in a base class.

Rules for Virtual Functions

The rules for the virtual functions in C++ are as follows:

1. Virtual functions cannot be static.
2. A virtual function can be a friend function of another class.
3. Virtual functions should be accessed using a pointer or reference of base class type to achieve runtime polymorphism.
4. The prototype of virtual functions should be the same in the base as well as the derived class.
5. They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.
6. A class may have a virtual destructor but it cannot have a virtual constructor.

The resolving of a function call is done at runtime.A virtual function is a member function that is declared in the base class using the keyword virtual and is re-defined (Overridden) in the derived class.

Some Key Points About Virtual Functions:

- Virtual functions are Dynamic in nature.
- They are defined by inserting the keyword "**virtual**" inside a base class and are always declared with a base class and overridden in a child class
- A virtual function is called during Runtime

```
// C++ program to illustrate
// concept of Virtual Functions

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

```
class base {
public:
        virtual void print() { cout << "print base class\n"; }

        void show() { cout << "show base class\n"; }
};

class derived : public base {
public:
        void print() { cout << "print derived class\n"; }

        void show() { cout << "show derived class\n"; }
};

int main()
{
        base* bptr;
        derived d;
        bptr = &d;

        // Virtual function, binded at runtime
        bptr->print();

        // Non-virtual function, binded at compile time
        bptr->show();

        return 0;
}
```

Output

```
print derived class

show base class
```

Limitations of Virtual Functions

- **Slower:** The function call takes slightly longer due to the virtual mechanism and makes it more difficult for the compiler to optimize because it does not know exactly which function is going to be called at compile time.
- **Difficult to Debug:** In a complex system, virtual functions can make it a little more difficult to figure out where a function is being called from.

Friend Class

A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class. For example, a LinkedList class may be allowed to access private members of Node.

A friend class can access both private and protected members of the class in which it has been declared as friend.

```
class Node {
private:
    int key;
    Node* next;
    /* Other members of Node Class */

    // Now class  LinkedList can
    // access private members of Node
    friend class LinkedList;
};
```

```
#include<iostream>
using namespace std;
class A
{
    int x;
          public:
    A()
    {
        x=10; }
    friend class B;    //friend class
};
class B
{  public:
        void display(A &t)
        {
            cout<<endl<<"The value of x="<<t.x;
        }
};

main()
{
    A _a;
    B _b;
    _b.display(_a);
    return 0;
}
```

Output

```
The value of x=10
```

In this example, class B is declared as a friend inside the class A. Therefore, B is a friend of class A.

Class B can access the private members of class A.

```
/ Example: Find the largest of two numbers using Friend Function
#include<iostream>
using namespace std;
class Largest
{
    int a,b,m;
    public:
        void set_data();
        friend void find_max(Largest);
};
void Largest::set_data()
{
    cout<<"Enter the First No:";
    cin>>a;
    cout<<"Enter the Second No:";
    cin>>b;
}
void find_max(Largest t)
{
    if(t.a>t.b)
        t.m=t.a;
    else
        t.m=t.b;

        cout<<"Maximum Number is\t"<<t.m;
}
main()
{
    Largest l;
    l.set_data();
    find_max(l);
    return 0;}
```

Output

```
Enter the First No:Enter the Second No:Maximum Number is    0
```

Output:-

```
Enter the First No: 21

Enter the Second No:32

Maximum Number is    32
```

C++ Friend function

If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.

By using the keyword friend compiler knows the given function is a friend function.

For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.

Declaration of friend function in C++

```
1.  class class_name
2.  {
3.      friend data_type function_name(argument/s);      // syntax of friend function.
4.  };
```

Characteristics of a Friend function:

- The function is not in the scope of the class to which it has been declared as a friend.
- It cannot be called using the object as it is not in the scope of that class.
- It can be invoked like a normal function without using the object.
- It cannot access the member names directly and has to use an object name and dot membership operator with the member name.
- It can be declared either in the private or the public part.

C++ friend function Example

Let's see the simple example of C++ friend function used to print the length of a box.

```
1.  #include <iostream>
2.  using namespace std;
3.  class Box
4.  {
5.     private:
6.        int length;
7.     public:
8.        Box(): length(0) { }
9.        friend int printLength(Box); //friend function
10. };
```

```
11.  int printLength(Box b)
12.  {
13.     b.length += 10;
14.     return b.length;
15.  }
16.  int main()
17.  {
18.     Box b;
19.     cout<<"Length of box: "<< printLength(b)<<endl;
20.     return 0;
21.  }
```

Output:

```
Length of box: 10
```