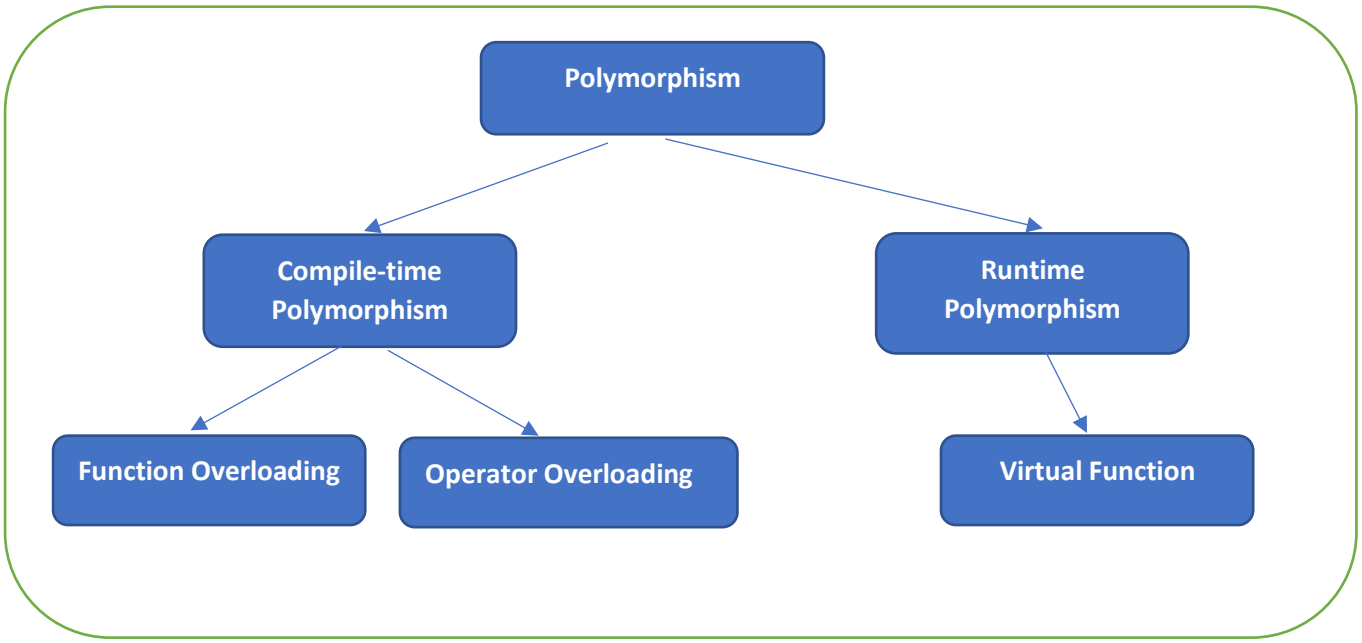
➢ POLYMORPHISM

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. A real-life example of polymorphism is a person who at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee. So, the same person exhibits different behaviour in different situations. This is called polymorphism. Polymorphism is considered one of the important features of Object-Oriented Programming.

Types of Polymorphism:

- Compile-time Polymorphism
- Runtime Polymorphism



Compile-Time Polymorphism:

This type of polymorphism is achieved by function overloading or operator overloading.

A. Function Overloading

When there are multiple functions with the same name but different parameters, then the functions are said to be **overloaded,** hence this is known as Function Overloading. Functions can be overloaded by **changing the number of arguments** or/and **changing the type of arguments**. In simple terms, it is a feature of object-oriented programming providing many functions that have the same name but distinct parameters when numerous tasks are listed

under one function name. There are certain Rules of Function Overloading that should be followed while overloading a function.

Below is the C++ program to show function overloading or compile-time polymorphism:

```
#include <iostream>
using namespace std;
class Xyz {
public:
    // Function with 1 int parameter
    void func(int x)
    {
        cout << "value of x is " << x << endl;
    }
    // Function with same name but 1 double parameter
    void func(double x)
    {
        cout << "value of x is " << x << endl;
    }
    // Function with same name and 2 int parameters
    void func(int x, int y)
    {
       cout << "value of x and y is " << x << ", " << y<< endl;
    }
};
// Driver code
int main()
{
    Xyz obj1;
    // Function being called depends on the parameters passed
    // func() is called with int value
    obj1.func(7);
```

```
    // func() is called with double value
    obj1.func(9.132);
    // func() is called with 2 int values
    obj1.func(85, 64);
    return 0;
}
```

Output:

value of x is 7

value of x is 9.132

value of x and y is 85, 64

Explanation: In the above example, a single function named function **func()** acts differently in three different situations, which is a property of polymorphism.

B. Operator Overloading

C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example, we can make use of the addition operator (+) for string class to concatenate two strings. We know that the task of this operator is to add two operands. So, a single operator '+', when placed between integer operands, adds them and when placed between string operands, concatenates them.

Below is the C++ program to demonstrate operator overloading:

```
#include <iostream>
using namespace std;
class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i = 0)
    {
        real = r;
        imag = i;
    }
```

```
    // This is automatically called
    // when '+' is used with between
    // two Complex objects
    Complex operator+(Complex const& obj)
    {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }
    void print() { cout << real << " + i" << imag << endl; }
};
// Driver code
int main()
{
    Complex c1(10, 5), c2(2, 4);
    // An example call to "operator+"
    Complex c3 = c1 + c2;
    c3.print();
}
```

Output:

12 + i9

Explanation: In the above example, the operator '+' is overloaded. Usually, this operator is used to add two numbers (integers or floating-point numbers), but here the operator is made to perform the addition of two imaginary or complex numbers.

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.

```
#include <iostream>
using namespace std;
class Parent
{
	public:
		void helloWorld(){
			cout<<"Hello World from Parent"<<endl;
		}
};
class Child:public Parent
{
	public:
		// Overriding function
		void helloWorld(){
			cout<<"Hello World from Child"<<endl;
		}
};
int main()
{
	Child child;
	child.helloWorld();
	return 0;
}
```

Output:

Hello World from Child

Runtime Polymorphism cannot be achieved by data members in C++. Let's see an example where we are accessing the field by reference variable of parent class which refers to the instance of the derived class.

```
#include <iostream>
using namespace std;
// Base class declaration.
class Animal {
public:
    string color = "Black";
};
// inheriting Animal class.
class Dog : public Animal {
public:
    string color = "Grey";
};

// Driver code
int main(void)
{
    Animal d = Dog();
    // accessing the field by reference variable which refers to derived
    cout << d.color;
}
```

Output:

Black

We can see that the parent class reference will always refer to the data member of the parent class.

B. Virtual Function

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

Below is the C++ program to demonstrate virtual function:

Example1

// the Virtual Function

```
#include <iostream>

using namespace std;

// Declaring a Base class

class GFG_Base {

public:

   // virtual function

   virtual void display()

   {

      cout << "Called virtual Base Class function"<< "\n\n";


   }

   void print()

   {
```

```
        cout << "Called GFG_Base print function"<< "\n\n";
    }
};

// Declaring a Child Class
class GFG_Child : public GFG_Base {
public:
    void display()
    {
        cout << "Called GFG_Child Display Function"<< "\n\n";
    }

    void print()
    {
        cout << "Called GFG_Child print Function"<<"\n\n";
    }
};

// Driver code
int main()
{
    // Create a reference of class GFG_Base
    GFG_Base* base;
    GFG_Child child;
    base = &child;
```

```
    // This will call the virtual function
    base->GFG_Base::display();
    // this will call the non-virtual function
    base->print();
}
```

Output:

Called virtual Base Class function

Called GFG_Base print function

Example2

//C++ program for virtual function overriding

```
#include <iostream>
using namespace std;
class base {
public:
    virtual void print()
    {
        cout << "print base class" << endl;
    }
    void show() { cout << "show base class" << endl; }
};

class derived : public base {
public:
    // print () is already virtual function in derived class, we could also declare as
    // virtual void print () explicitly
    void print() { cout << "print derived class" << endl; }
    void show() { cout << "show derived class" << endl; }
};
```

```
// Driver code
int main()
{
    base* bptr;
    derived d;
    bptr = &d;
    // Virtual function, binded at runtime (Runtime polymorphism)
    bptr->print();
    // Non-virtual function, binded at compile time
    bptr->show();
    return 0;
}
```

Output:

print derived class

show base class

- **MORE TYPES OF POLYMORPHISM**

When we talk about Polymorphism in C++, we come to hear the following four types:

- Adhoc
- Inclusion
- Coercive
- Parametric

1. **Ad-hoc Polymorphism**, also called as **Overloading Ad-hoc Polymorphism** allows functions having same name to act differently for different types. For example: The + operator adds two integers and concatenates two strings.

Above example could be better illustrated by invoking the function "sum()" in under-mentioned code:

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

int sum(int x, int y)
{
    int c = x + y;
```

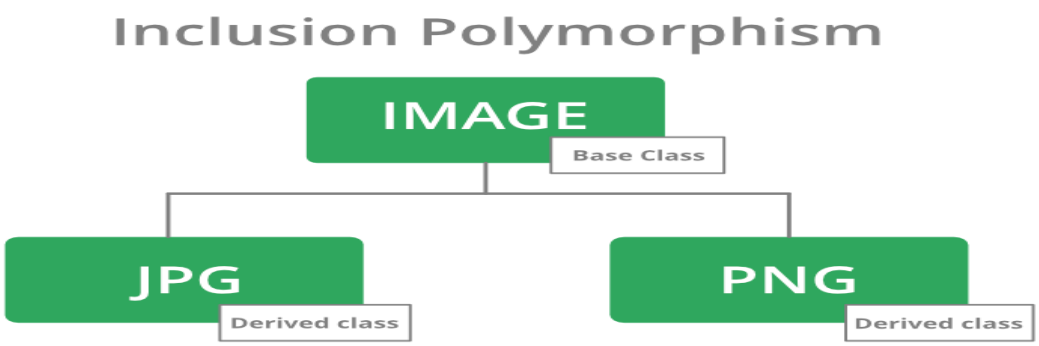
```
    return c;
}

string sum(const char* x, const char* y)
{
    string summation(x);
    summation += y;
    return summation;
}

int main()
{
    cout << sum(50, 20)
      << " :- Integer addition Output\n";
    cout << sum("Polymorphism", " achieved")
      << " :- String Concatenation Output\n";
}
```

Output:

70 :- Integer addition Output
Polymorphism achieved :- String Concatenation Output

Hence, by calling two different functions(which differ in the type of arguments) having the same names, to execute multiple operations, we have successfully achieved Ad-hoc Polymorphism.

2. **Inclusion Polymorphism**, also called as **Subtyping Inclusion Polymorphism** is the ability to use derived classes through base class pointers and references. It is also known as Run-time polymorphism because the address of the function is not located by the Compiler at compile-time, rather, the right pointer from the virtual table is dereferenced to invoke the function at run-time. The concept of Virtual Function, also known as Dynamic Linkage, is employed to achieve Inclusion Polymorphism. The usage of Virtual Function allows the selection of that function which is to be invoked based on the kind of object for which it is called. **For example:** To implement such a Polymorphism technique, let us take different files under consideration such as .jpg, .gif, .png files. All these files fall under the category of Image Files.



So, they can be represented as Classes Derived from Image Base Class and overriding the display() pure virtual function. Above example could be better understood by the following illustration:

```
#include <iostream>

using namespace std;


class Image {


public:
        Image()
        {
        }


        virtual void display() = 0;
};
class Jpg : public Image {
public:
        Jpg()
        {
        }


        void display()
        {
                cout << "JPG Image File" << endl;
        }
};


class Png : public Image {
public:
        Png()
        {
```

```
	}

	void display()
	{
		cout << "PNG Image File" << endl;
	}
};

// Main function
int main()
{
	Image* img;
	Jpg jg;
	Png pg;

	// stores the address of Jpg
	img = &jg;

	// invoking display() func of Jpg
	img->display();

	// stores the address of Png
	img = &pg;

	// invoking display() func of Png
	img->display();
	return 0;
}
```

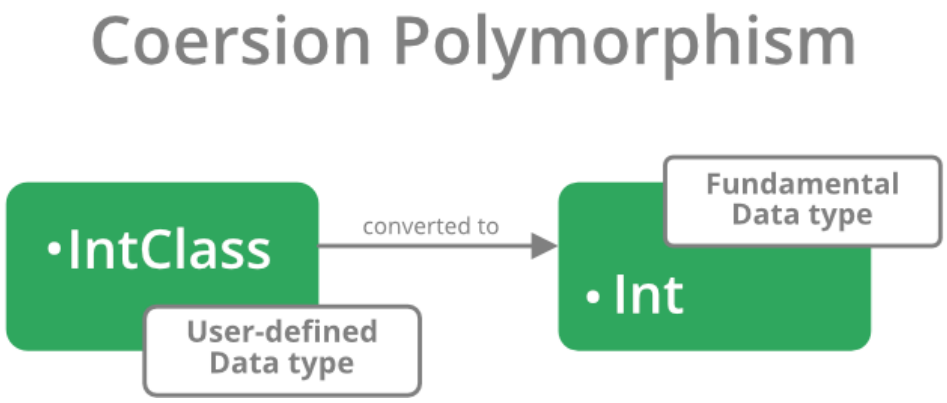
Output:

JPG Image File

PNG Image File

Hence, in the above code, we have two different Classes with a function having the same name and not differing by Parameters, but with different implementations.

3. **Coercion Polymorphism**, also called as **Casting Coersion Polymorphism** occurs when an object or primitive is cast into some other type. It could be either Implicit or Explicit. Implicit casting happens as a responsibility of Compiler itself. For example: float f=100 (integer implicitly gets promoted to float) Explicit casting makes use of some type-casting expressions such as const_cast, dynamic_cast, etc. For example: When a class defines conversion operator for some type, say “int”, then, it could be employed anywhere in the program where integer type data is expected. Illustration Below could make it more easier to understand:




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

class IntClass {
   int num;

public:
   IntClass(int a)
            : num(a)
   {
   }

   operator int() const
   {
            return num;
   } // conversion from User-defined type to Basic type
};

void show(int x)
{
   cout << x << endl;
}
```

```
int main()
{
   IntClass i = 100;
   show(746); // outputs 746
   show(i); // outputs 100
}
```

Output:
746
100

The IntClass reference is used in place of integer type argument, and hence, the concept of Casting is well understood.

4. **Parametric Polymorphism**, also called as **Early Binding Parametric Polymorphism** opens a way to use the same piece of code for different types. It is implemented by the use of Templates. For example: To develop an understanding of this sort of polymorphism, let us execute a program for finding greater of two Integers or two Strings,

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

template <class temp>
temp greater(temp a, temp b)
{
   if (a > b)
            return a;
   else
            return b;
}

int main()
{
   cout << ::greater(55, 11) << endl;

   string str1("Early"), str2("Binding");
   cout << ::greater(str1, str2) << endl;
}
```

Output:
55
Early

Using Templates, the same function can be parameterized with different types of data, but this needs to be decided at compile-time itself, and hence, this polymorphism is

named so. If we wish to achieve such polymorphism for pointers, it turns into Ad-hoc Polymorphism.

- **VIRTUAL FUNCTIONS**

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. The resolving of a function call is done at runtime.

Rules for Virtual Functions

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

- Virtual functions cannot be static.
- A virtual function can be a friend function of another class.
- Virtual functions should be accessed using a pointer or reference of base class type to achieve runtime polymorphism.
- The prototype of virtual functions should be the same in the base as well as the derived class.
- 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.
- A class may have a virtual destructor but it cannot have a virtual constructor.

Consider the following simple program showing the runtime behavior of virtual functions.

```
// 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

Explanation: Runtime polymorphism is achieved only through a pointer (or reference) of the base class type. Also, a base class pointer can point to the objects of the base class as well as to the objects of the derived class. In the above code, the base class pointer 'bptr' contains the address of object 'd' of the derived class.

Compile time (early binding) VS runtime (late binding) behavior of Virtual Functions

Late binding (Runtime) is done in accordance with the content of the pointer (i.e. location pointed to by pointer) and Early binding (Compile-time) is done according to the type of pointer since the print() function is declared with the virtual keyword so it will be bound at runtime (output is print derived class as the pointer is pointing to object of derived class) and show() is non-virtual so it will be bound during compile time (output is show base class as the pointer is of base type).

Note: If we have created a virtual function in the base class and it is being overridden in the derived class then we don't need a virtual keyword in the derived class, functions are automatically considered virtual functions in the derived class.

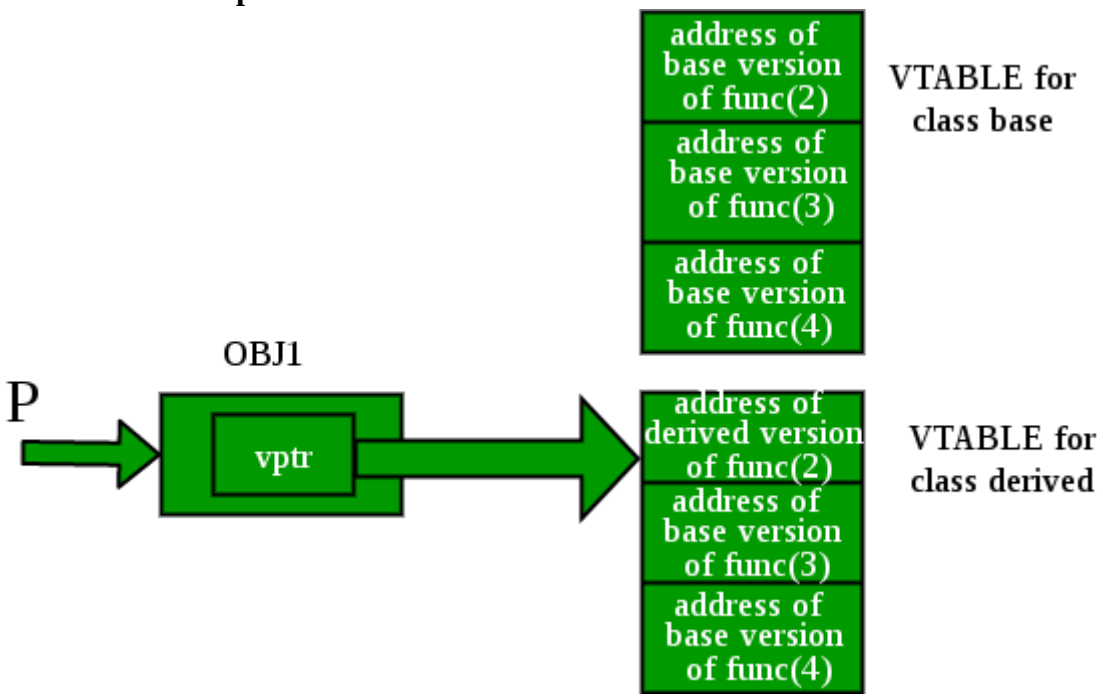
Working of Virtual Functions (concept of VTABLE and VPTR)

As discussed here, if a class contains a virtual function then the compiler itself does two things.

- If an object of that class is created then a virtual pointer (VPTR) is inserted as a data member of the class to point to the VTABLE of that class. For each new object created, a new virtual pointer is inserted as a data member of that class.

- Irrespective of whether the object is created or not, the class contains as a member a static array of function pointers called VTABLE. Cells of this table store the address of each virtual function contained in that class.

Consider the example below:




```
// C++ program to illustrate
// working of Virtual Functions
#include <iostream>
using namespace std;

class base {
public:
  void fun_1() { cout << "base-1\n"; }
  virtual void fun_2() { cout << "base-2\n"; }
  virtual void fun_3() { cout << "base-3\n"; }
  virtual void fun_4() { cout << "base-4\n"; }
};

class derived : public base {
public:
  void fun_1() { cout << "derived-1\n"; }
  void fun_2() { cout << "derived-2\n"; }
  void fun_4(int x) { cout << "derived-4\n"; }
};

int main()
{
  base* p;
```

```
  derived obj1;
  p = &obj1;

  // Early binding because fun1() is non-virtual
  // in base
  p->fun_1();

  // Late binding (RTP)
  p->fun_2();

  // Late binding (RTP)
  p->fun_3();

  // Late binding (RTP)
  p->fun_4();

  // Early binding but this function call is
  // illegal (produces error) because pointer
  // is of base type and function is of
  // derived class
  // p->fun_4(5);

  return 0;
}
```

Output:

```
base-1
derived-2
base-3
base-4
```

Explanation: Initially, we create a pointer of the type base class and initialize it with the address of the derived class object. When we create an object of the derived class, the compiler creates a pointer as a data member of the class containing the address of VTABLE of the derived class.

A similar concept of **Late and Early Binding** is used as in the above example. For the fun_1() function call, the base class version of the function is called, fun_2() is overridden in the derived class so the derived class version is called, fun_3() is not overridden in the derived class and is a virtual function so the base class version is called, similarly fun_4() is not overridden so base class version is called.

Note: fun_4(int) in the derived class is different from the virtual function fun_4() in the base class as prototypes of both functions are different.

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.