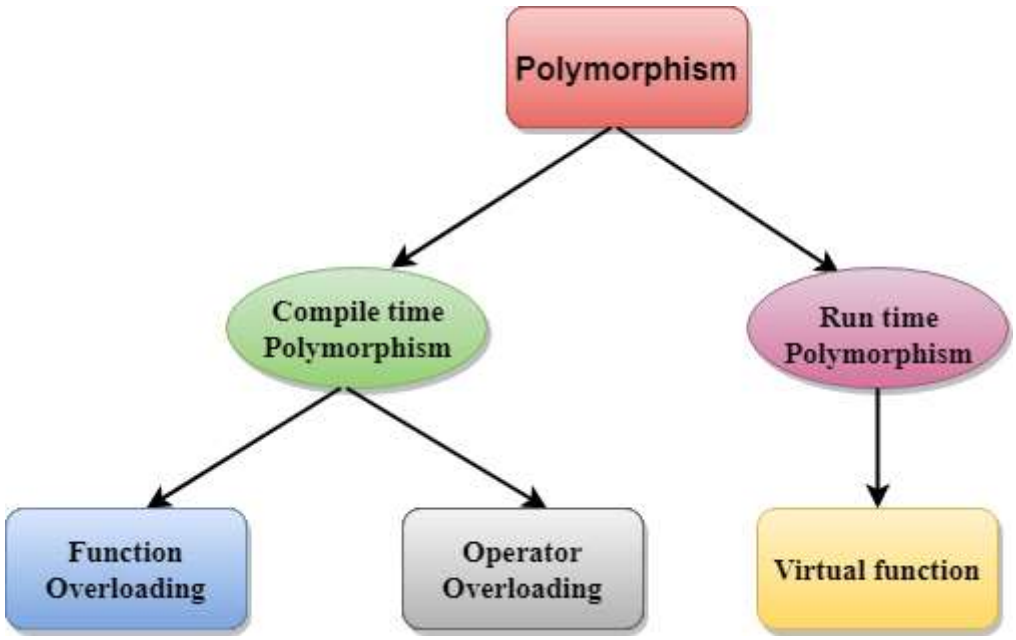
C++ Polymorphism

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word. In object-oriented programming, we use 3 main concepts: inheritance, encapsulation, and polymorphism.

Real Life Example Of Polymorphism

Let's consider a real-life example of polymorphism. A lady behaves like a teacher in a classroom, mother or daughter in a home and customer in a market. Here, a single person is behaving differently according to the situations.

There are two types of polymorphism in C++:



- **Compile time polymorphism**: The overloaded functions are invoked by matching the type and number of arguments. This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time. It is achieved by function overloading and operator overloading

which is also known as static binding or early binding. Now, let's consider the case where function name and prototype is same.

- **Run time polymorphism**: Run time polymorphism is achieved when the object's method is invoked at the run time instead of compile time. It is achieved by method overriding which is also known as dynamic binding or late binding.
-

Differences b/w compile time and run time polymorphism.

Compile time polymorphism	Run time polymorphism
The function to be invoked is known at the compile time.	The function to be invoked is known at the run time.
It is also known as overloading, early binding and static binding.	It is also known as overriding, Dynamic binding and late binding.
Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters.	Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It provides fast execution as it is known at the compile time.	It provides slow execution as it is known at the run time.
It is less flexible as mainly all the things execute at the compile time.	It is more flexible as all the things execute at the run time.

C++ Runtime Polymorphism Example

Let's see a simple example of run time polymorphism in C++.an example without the virtual keyword.

```
#include <iostream>

using namespace std;

class Animal {

   public:

void eat(){

cout<<"Eating...";

   }

};

class Dog: public Animal

{

 public:

 void eat()

   {          cout<<"Eating bread...";

   }

};

int main(void) {

  Dog d = Dog();

  d.eat();

  return 0;
```

```
}
```

C++ Run time Polymorphism Example: By using two derived class

Let's see another example of run time polymorphism in C++ where we are having two derived classes.// an example with virtual keyword.

```
#include <iostream>
using namespace std;
class Shape {                          //  base class
   public:
virtual void draw(){                   // virtual function
cout<<"drawing..."<<endl;
   }
};
class Rectangle: public Shape              //  inheriting Shape class.
{
 public:
 void draw()
  {
     cout<<"drawing rectangle..."<<endl;
   }
};
class Circle: public Shape                 //  inheriting Shape class.
```

```
{
 public:
 void draw()
  {
     cout<<"drawing circle..."<<endl;
  }
};
int main(void) {
   Shape *s;                    //  base class pointer.
   Shape sh;                    // base class object.
      Rectangle rec;
       Circle cir;
     s=&sh;
    s->draw();
       s=&rec;
    s->draw();
   s=?
   s->draw();
}
```

Runtime Polymorphism with Data Members

Runtime Polymorphism can be achieved by data members in C++. Let's see an example where we are accessing the field by reference variable which refers to the instance of derived class.

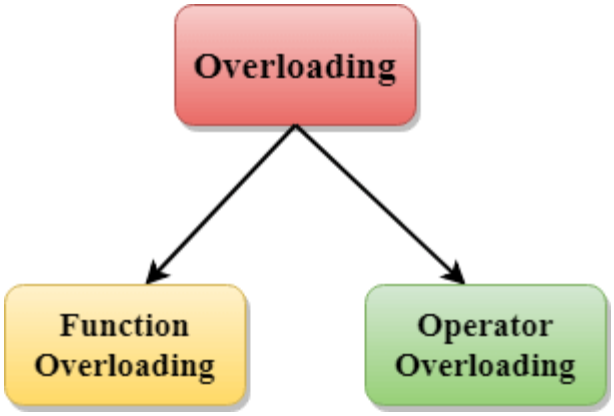
```
#include <iostream>
using namespace std;
class Animal {                          //  base class declaration.
   public:
   string color = "Black";
};
class Dog: public Animal                // inheriting Animal class.
{
 public:
   string color = "Grey";
};
int main(void) {
    Animal d= Dog();
   cout<<d.color;
}
```

C++ Overloading (Function and Operator)

If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:

Types of overloading in C++ are:

- Function overloading
- Operator overloading



C++ Function Overloading

Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++. In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.

The **advantage** of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

C++ Function Overloading Example

Let's see the simple example of function overloading where we are changing number of arguments of add() method.

// program of function overloading when number of arguments vary.

```
#include <iostream>

using namespace std;

class Cal {

   public:
```

```
static int add(int a,int b){
      return a + b;
   }
static int add(int a, int b, int c)
   {
      return a + b + c;
   }
};
int main(void) {
   Cal C;                                    //    class object declaration.
   cout<<C.add(10, 20)<<endl;
   cout<<C.add(12, 20, 23);
  return 0;
}
```

Let's see the simple example when the type of the arguments vary.

// Program of function overloading with different types of arguments.

```
        #include<iostream>
        using namespace std;
        int mul(int,int);
        float mul(float,int);
        int mul(int a,int b)
        {
```

```
    return a*b;
}
float mul(double x, int y)
{
    return x*y;
}
int main()
{
    int r1 = mul(6,7);
    float r2 = mul(0.2,3);
    cout << "r1 is : " <<r1<< endl;
    cout <<"r2 is : "  <<r2<< endl;
    return 0;
}
```

C++ Operators Overloading

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

The advantage of Operators overloading is to perform different operations on the same operand.

Operator that cannot be overloaded are as follows:

- Scope operator (::)
- Sizeof
- member selector(.)
- member pointer selector(*)
- ternary operator(?:)

Rules for Operator Overloading

- Existing operators can only be overloaded, but the new operators cannot be overloaded.
- The overloaded operator contains atleast one operand of the user-defined data type.
- We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
- When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
- When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

C++ Operators Overloading Example

Let's see the simple example of operator overloading in C++. In this example, void operator ++ () operator function is defined (inside Test class).

// program to overload the unary operator ++.

```
#include <iostream>
using namespace std;
class Test
{
  private:
    int num;
```

```
  public:
     Test(): num(8){}
     void operator ++()
   {
        num = num+2;
     }
     void Print()
{
        cout<<"The Count is: "<<num;
     }
};
int main()
{
   Test tt;
   ++tt;  // calling of a function "void operator ++()"
   tt.Print();
   return 0;
}
```

Let's see a simple example of overloading the binary operators.

// program to overload the binary operators.

```
#include <iostream>
using namespace std;
class A
{
   int x;
     public:
     A(){}
   A(int i)
   {
```

```
    x=i;
  }
  void operator+(A);
  void display();
};
void A :: operator+(A a)
{

  int m = x+a.x;
  cout<<"The result of the addition of two objects is : "<<m;
}
int main()
{
  A a1(5);
  A a2(4);
  a1+a2;
  return 0;
}
```

C++ Function Overriding

If derived class defines same function as defined in its base class, it is known as function overriding in C++. It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the function which is already provided by its base class.

C++ Function Overriding Example

Let's see a simple example of Function overriding in C++. In this example, we are overriding the eat() function.

```
#include <iostream>
using namespace std;
class Animal {
   public:
void eat(){
cout<<"Eating...";
   }
};
class Dog: public Animal
{
 public:
 void eat()
   {
      cout<<"Eating bread...";
   }
};
int main(void) {
  Dog d = Dog();
  d.eat();
  return 0;
}
```

C++ virtual function

- A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
- A 'virtual' is a keyword preceding the normal declaration of a function.
- When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

Rules of Virtual Function

- Virtual functions must be members of some class.
- Virtual functions cannot be static members.
- They are accessed through object pointers.
- They can be a friend of another class.
- A virtual function must be defined in the base class, even though it is not used.
- The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.
- We cannot have a virtual constructor, but we can have a virtual destructor

- Consider the situation when we don't use the virtual keyword.

C++ virtual function Example

Let's see the simple example of C++ virtual function used to invoked the derived class in a program.

```
#include <iostream>
{
 public:
 virtual void display()
 {
  cout << "Base class is invoked"<<endl;
 }
};
class B:public A
{
 public:
 void display()
 {
  cout << "Derived Class is invoked"<<endl;
 }
};
int main()
{
 A* a;    //pointer of base class
```

```
 B b;     //object of derived class
 a = &b;
 a->display();   //Late Binding occurs
}
```

C++ virtual function example

```
#include <iostream>
using namespace std;
class disha
{
public:
 virtual void display()
{
 cout<<"Welcome to DiSHA Computer Institute"<<endl;
}
};

class coding: public disha
{
 public:
 void display()
  {
     cout<< "C,C++,Java,Python"<<endl;
   }
```

```
};
int main()
{


   disha *d;
   disha d1;
   d1.display();
   coding c;
   d=&c;
   d->display();
}
```

Pure Virtual Function

- A virtual function is not used for performing any task. It only serves as a placeholder.
- When the function has no definition, such function is known as "**do-nothing**" function.
- The "**do-nothing**" function is known as a **pure virtual function**. A pure virtual function is a function declared in the base class that has no definition relative to the base class.
- A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes.
- The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

Pure virtual function can be defined as:

```
virtual void display() = 0;
```

Let's see a simple example:

```
#include <iostream>
using namespace std;
class Base
{
   public:
   virtual void show() = 0;
};
class Derived : public Base
{
   public:
   void show()
   {
      std::cout << "Derived class is derived from the base class." << std::endl;
   }
};
int main()
{
   Base *bptr;
   //Base b;
   Derived d;
   bptr = &d;
```

```
    bptr->show();
    return 0;
}
```

C++ pur virtual function

```
//do nothing function
#include <iostream>
using namespace std;
class disha
{
   public:
virtual void display()=0;
 void info()
  {
     cout<< "Disha Computer Institute Dange Chowk
Branch"<<endl;
   }
};

class coding: public disha
{
 public:
 void display()
  {
     cout<< "C,C++,Java,Python"<<endl;
```

```
    }
};
int main()
{
        disha a1;
        a1.display();
        a1.info();
        coding d1;
        d1.display();
        d1.info();
}
```