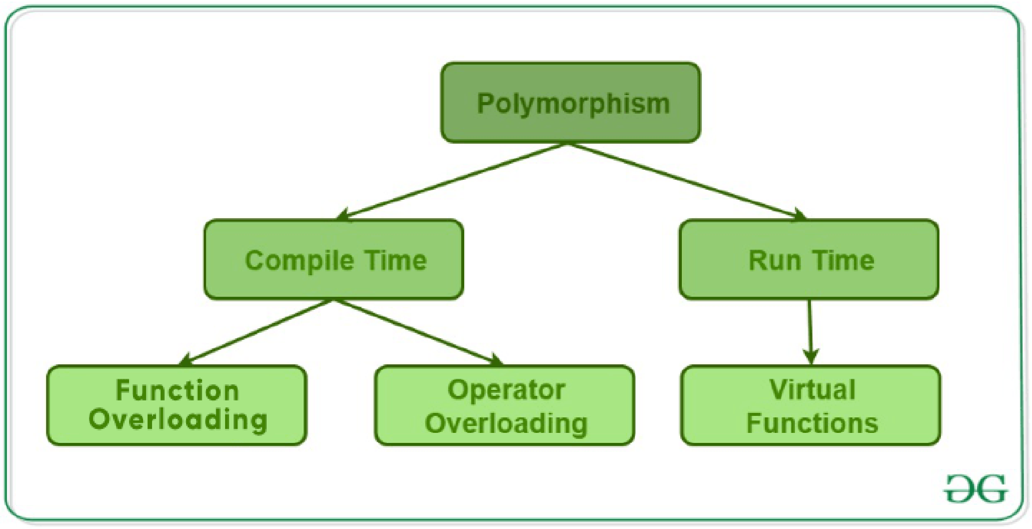
Polymorphism

- 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.
- In C++ polymorphism is mainly divided into two types:
    - Compile time Polymorphism
    - Runtime Polymorphism



- **Function Overloading**:
    - When there are multiple functions with same name but different parameters then these functions are said to be overloaded.
    - Functions can be overloaded by change in number of arguments or/and change in type of arguments.
- **Operator Overloading:**
    - C++ also provide option to overload operators.
    - For example, we can make the operator ('+') for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So, a single operator '+' when placed between integer operands , adds them and when placed between string operands, concatenates them.
- **Function overriding:**
    - It 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.

Function Overloading

- Function overloading is a feature of object-oriented programming where two or more functions can have the same name but different parameters.
- When a function name is overloaded with different jobs it is called Function Overloading.
- In Function Overloading "Function" name should be the same and the arguments should be different.
- **Functions that cannot be overloaded:**

a. Functions can not be overloaded if they differ only in the return type.

b. Member function declarations with the same name and the name parameter-type-list cannot be overloaded if any of them is a static member function declaration.

```
class Test
{
   static void fun(int i) {}
   void fun(int i) {}
};
```

c. Parameter declarations that differ only in a pointer * versus an array [] are equivalent. That is the array declaration is adjusted to become a pointer declaration. Only the second and subsequent array dimensions are significant in parameter types.

```
int fun(int *ptr);
int fun(int ptr[]);    // redeclaration of fun(int *ptr)
```

d. Parameter declarations that differ only in that one is a function type and the other is a pointer to the same function type are equivalent.

```
void h(int ());
void h(int (*)());    // redeclaration of h(int())
```

e. Parameter declarations that differ only in the presence or absence of const and/or volatile are equivalent. That is, the const and volatile type-specifiers for each parameter type are ignored when determining which function is being declared, defined, or called.

```
int f ( int x)
{
   return x+10;
}

int f ( const int x)     // redefinition of int f(int)
{
   return x+10;
}
```

Only the const and volatile type-specifiers at the outermost level of the parameter type specification are ignored in this fashion; const and volatile type-specifiers buried within a parameter type specification are significant and can be used to distinguish overloaded function declarations. In particular, for any type T, "pointer to T," "pointer to const T," and "pointer to volatile T" are considered distinct parameter types, as are "reference to T," "reference to const T," and "reference to volatile T."

f. Two parameter declarations that differ only in their default arguments are equivalent.

```
int f ( int x, int y)
{
   return x+10;
}

int f ( int x, int y = 10)     // redefinition of int f(int, int)
{
   return x+y;
}
```

- **‘const’ keyword in function overloading:**
  - C++ allows member methods to be overloaded on the basis of const type. Overloading on the basis of const type can be useful when a function return reference or pointer.
  - We can make one function const, that returns a const reference or const pointer, other non-const function, that returns non-const reference or pointer.
  - **Example:**

    Input:

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

class Test
{
   protected:
      int x;
   public:
      Test (int i):x(i) { }
      void fun() const
      {
         cout << "fun() const called " << endl;
      }
      void fun()
      {
         cout << "fun() called " << endl;
      }
};

int main()
{
   Test t1 (10);
   const Test t2 (20);
   t1.fun();
   t2.fun();
}
```

Output:

```
fun() called
fun() const called
```

- The two methods 'void fun() const' and 'void fun()' have same signature except that one is const and other is not. Also, if we take a closer look at the output, we observe that, 'const void fun()' is called on const object and 'void fun()' is called on non-const object.
- C++ allows functions to be overloaded on the basis of const-ness of parameters only if the const parameter is a reference or a pointer.
- When we pass by reference or pointer, we can modify the value referred or pointed, so we can have two versions of a function, one which can modify the referred or pointed value, other which can not.

- **Function Overloading with float:**
  - It's a well known fact in Function Overloading, that the compiler decides which function needs to be invoked among the overloaded functions. If the compiler can not choose a function amongst two or more overloaded functions, the situation is – *"Ambiguity in Function Overloading"*.
  - **Example:**

Input:

```
#include<iostream>
using namespace std;
void test(float s,float t)
{
   cout << "Function with float called ";
}

void test(int s, int t)
{
   cout << "Function with int called ";
}

int main()
{
   test(3.5, 5.6);
   return 0;
}
```

Output:

```
In function 'int main()':
13:13: error: call of overloaded 'test(double, double)' is ambiguous
 test(3.5,5.6):
```

- The reason behind the ambiguity in above code is that the floating literals **3.5** and **5.6** are actually treated as double by the compiler.

- As per C++ standard, *floating point literals (compile time constants) are treated as double unless explicitly specified by a suffix.*
- Since compiler could not find a function with double argument and got confused if the value should be converted from double to int or float.
- **Rectifying the error:** We can simply tell the compiler that the literal is a float and not double by providing ***suffix f***.

```
int main()
{
   test(3.5f, 5.6f); // Added suffix "f" to both values to tell compiler, it's a float value
   return 0;
}
```

Does overloading work with Inheritance?

- Overloading doesn't work for derived class in C++ programming language.
- There is no overload resolution between Base and Derived. The compiler looks into the scope of Derived, finds the single function and calls it. It never disturbs with the (enclosing) scope of Base.
- In C++, there is no overloading across scopes – derived class scopes are not an exception to this general rule.

Can main() be overloaded in C++?

- We can not write multiple main() in program to overload it.
- To overload main() function in C++, it is necessary to use class and declare the main as member function.
- Note: main is not reserved word in programming languages.
- The following program shows overloading of main() function in a class.

Input:

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

class Test
{
   public:
   int main(int s)
   {
      cout << s << "\n";
      return 0;
   }
   int main(char *s)
   {
      cout << s << endl;
      return 0;
   }
```

```
    int main(int s ,int m)
    {
        cout << s << " " << m;
        return 0;
    }
};

int main()
{
    Test obj;
    obj.main(3);
    obj.main("I love C++");
    obj.main(9, 6);
    return 0;
}
```

Output:

```
3
I love C++
9 6
```

Operator Overloading

- In C++, we can make operators to work for user defined classes. This means 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 overload an operator '+' in a class like String so that we can concatenate two strings by just using +. Other example classes where arithmetic operators may be overloaded are Complex Number, Fractional Number, Big Integer, etc.
- **Example:**
  Input:

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

class Complex
{
   private:
      int real;
      int imag;

   public:
      Complex(int r = 0, int i = 0)
      {
         real = r;
         imag = i;
      }

      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;
      }
};


int main()
{
   Complex obj1(2, 4);
   obj1.print();
```

```
    Complex obj2(3, 6);
    obj2.print();

    Complex obj3;
    obj3 = obj1 + obj2;
    obj3.print();

    return 0;
}
```

Output:

```
2 + i4
3 + i6
5 + i10
```

- **What is the difference between operator functions and normal functions?**
    - Operator functions are same as normal functions. The only differences are, name of an operator function is always operator keyword followed by symbol of operator and operator functions are called when the corresponding operator is used.
    - **Example:** Following is an example of *global operator function*.
      Input:

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

class Complex
{
    private:
        int real;
        int imag;

    public:
        Complex(int r = 0, int i = 0) : real(r), imag(i) {}

        void print()
        {
            cout << real << " + i" << imag << endl;
        }

        friend Complex operator + (Complex const &obj1, Complex const &obj2);
};

Complex operator + (Complex const &obj1, Complex const &obj2)
{
   return Complex(obj1.real + obj2.real, obj1.imag + obj2.imag);
}

int main()
{
    Complex obj1(2, 4);
    obj1.print();

    Complex obj2(3, 6);
    obj2.print();

    Complex obj3;
    obj3 = obj1 + obj2;
    obj3.print();

    return 0;
}
```

Output:

```
2 + i4
3 + i6
5 + i10
```

- **Can we overload all operators?**
  Almost all operators can be overloaded except few. Following is the list of operators that cannot be overloaded.
  a) . (dot)
  b) ::
  c) ?:
  d) sizeof

- **Important points about operator overloading:**
  - For operator overloading to work, at least one of the operands must be a user defined class object.
  - **Assignment Operator:** Compiler automatically creates a default assignment operator with every class. The default assignment operator does assign all members of right side to the left side and works fine most of the cases (this behavior is same as copy constructor).
  - **Conversion Operator:** We can also write conversion operators that can be used to convert one type to another type. Overloaded conversion operators must be a member method. Other operators can either be member method or global method.
  - Any constructor that can be called with a single argument works as a conversion constructor, means it can also be used for implicit conversion to the class being constructed.

- **When should we write our own assignment operator in C++?**
  - The answer is same as Copy Constructor. If a class doesn't contain pointers, then there is no need to write assignment operator and copy constructor. The compiler creates a default copy constructor and assignment operators for every class. The compiler created copy constructor and assignment operator may not be sufficient when we have pointers or any run time allocation of resource like file handle, a network connection..etc.

- **Copy constructor vs assignment operator in C++**
  - **Example:**

    Input:

```
#include<iostream>
#include<stdio.h>
using namespace std;

class Test
{
   public:
      Test() {}
      Test(const Test &t)
      {
         cout<<"Copy constructor called "<<endl;
      }

      Test& operator = (const Test &t)
      {
         cout<<"Assignment operator called "<<endl;
         return *this;
      }
};

int main()
{
   Test t1, t2;
   t2 = t1;
   Test t3 = t1;
   return 0;
}
```

Output:

```
Assignment operator called
Copy constructor called
```

- Copy constructor is called when a new object is created from an existing object, as a copy of the existing object. And assignment operator is called when an already initialized object is assigned a new value from another existing object.

```
t2 = t1;            // calls assignment operator, same as "t2.operator=(t1);"
Test t3 = t1;       // calls copy constructor, same as "Test t3(t1);"
```

- **Is assignment operator inherited?**
  - In C++, like other functions, assignment operator function is inherited in derived class.
  - **Example:** In the following program, base class assignment operator function can be accessed using the derived class object.

    Input:

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

class A
{
    public:
    A & operator= (A &a)
    {
        cout << "Base class assignment operator called";
        return *this;
   }
};

class B: public A { };

int main()
{
    B a, b;
    // calling base class assignment operator function using derived class
    a.A::operator=(b);
    return 0;
}
```

Output:

```
Base class assignment operator called
```

Function Overriding

- Function Overriding (Runtime Polymorphism) 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**.

- **Virtual function:**
  - A virtual function is a member function which is declared within a base class and is re-defined (Overridden) by a derived class.
  - When we refer to a derived class object using a pointer or a reference to the base class, we can call a virtual function for that object and execute the derived class's version of the function.
  - **Key Points:**
    1. Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
    2. They are mainly used to achieve Runtime polymorphism.
    3. Functions are declared with a virtual keyword in base class.
    4. The resolving of function call is done at Run-time.
  - **What is the use?**
    - Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object.
  - **Rules for Virtual Functions:**
    - Virtual functions cannot be static.
    - A virtual function can be a friend function of another class.
    - Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
    - The prototype of virtual functions should be the same in the base as well as 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 virtual destructor, but it cannot have a virtual constructor.

- **Compile-time (early binding) VS run-time (late binding) behavior:**
  - **Example:**

    Input:
```
#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:
        void print()
        {
            cout << "print derived class" << endl;
        }

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

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

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

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

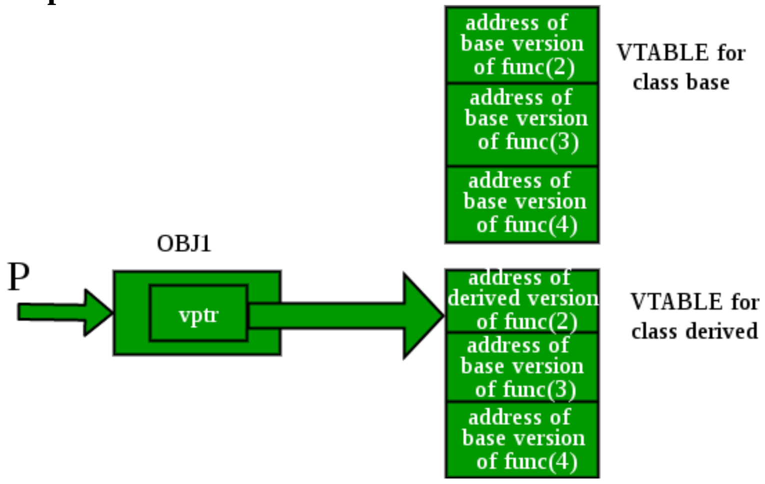
Output:

```
print derived class
show base class
```

Explanation: Runtime polymorphism is achieved only through a pointer (or reference) of base class type. Also, a base class pointer can point to the objects of base class as well as to the objects of derived class. In above code, base class pointer 'bptr' contains the address of object 'd' of derived class. Late binding (Runtime) is done in accordance with the content of pointer (i.e. location pointed to by pointer) and Early binding (Compile time) is done according to the type of pointer, since print() function is declared with virtual keyword so it will be bound at run-time (output is *print derived class* as 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 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 virtual keyword in the derived class, functions are automatically considered as virtual functions in the derived class.

- **Working of Virtual functions:**
  - The compiler maintains two things to serve this purpose:
    1. **vtable:** A table of function pointers, maintained per class.
    2. **vptr:** A pointer to vtable, maintained per object instance.
  - If object of that class is created, then a virtual pointer (vptr) is inserted as a data member of the class to point to vtable of that class. For each new object created, a new virtual pointer is inserted as a data member of that class.
  - Irrespective of object is created or not, a static array of function pointer called vtable where each cell contains the address of each virtual function contained in that class.
  - **Example:**


address of base version of func(2)
address of base version of func(3)
address of base version of func(4)
VTABLE for class base
OBJ1
P
vptr
address of derived version of func(2)
address of base version of func(3)
address of base version of func(4)
VTABLE for class derived

Input:

```
#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);
}
```

Output:

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

Explanation: Initially, we create a pointer of 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. Similar concept of **Late and Early Binding** is used as in above example. For fun_1() function call, base class version of function is called, fun_2() is overridden in derived class so derived class version is called, fun_3() is not overridden in derived class and is virtual function so base class version is called, similarly fun_4() is not overridden so base class version is called.

NOTE: fun_4(int) in derived class is different from virtual function fun_4() in base class as prototype of both the function is different.

- **Virtual functions in derived classes:**
  - In C++, once a member function is declared as a virtual function in a base class, it becomes virtual in every class derived from that base class. In other words, it is not necessary to use the keyword virtual in the derived class while declaring redefined versions of the virtual base class function.

- **Default arguments in Virtual function:**
  - **Example 1:**

    Input:

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

class Base
{
   public:
       virtual void fun ( int x = 0 )
       { cout << "Base::fun(), x = " << x << endl; }
};

class Derived : public Base
{
   public:
       virtual void fun ( int x )
       { cout << "Derived::fun(), x = " << x << endl; }
};

int main()
{
   Derived d1;
   Base *bp = &d1;
   bp->fun();
}
```

Output:

```
Derived::fun(), x = 0
```

Explanation: If we take a closer look at the output, we observe that fun() of derived class is called and default value of base class fun() is used. Default arguments do not participate in signature of functions. So, signatures of fun() in base class and derived class are considered same, hence the fun() of base class is overridden. Also, the default value is used at compile time. When compiler sees that an argument is missing in a function call, it substitutes the default value given. Therefore, in the above program, value of x is substituted at compile time, and at run time derived class's fun() is called.

- **Example 2:**

  Input:

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

class Base
{
   public:
      virtual void fun ( int x = 0 )
      { cout << "Base::fun(), x = " << x << endl; }
};

class Derived : public Base
{
   public:
      virtual void fun ( int x = 10 )         // NOTE THIS CHANGE
      { cout << "Derived::fun(), x = " << x << endl; }
};

int main()
{
   Derived d1;
   Base *bp = &d1;
   bp->fun();
}
```

Output:

```
Derived::fun(), x = 0
```

Explanation: The output of this program is same as the previous program. The reason is same, the default value is substituted at compile time. The fun() is called on bp which is a pointer of Base type. So, compiler substitutes 0 (not 10). In general, it is a best practice to avoid default values in virtual functions to avoid confusion.

- **Pure Virtual Functions and Abstract Classes:**
  - Sometimes implementation of all function cannot be provided in a base class because we don't know the implementation. Such a class is called abstract class.
  - We cannot create objects of abstract classes.
  - A pure virtual function (or abstract function) in C++ is a virtual function for which we can have implementation, But we must override that function in the derived class, otherwise the derived class will also become abstract class.
  - **Example:**

```
// An abstract class
class Test
{
   /* Data members of class */

   public:
      // Pure Virtual Function
      virtual void show() = 0;

   /* Other members */
};
```

  - **Key Points:**
    - A class is abstract if it has at least one pure virtual function. Ans we can not create an object of it.
    - We can have pointers and references of abstract class type.
    - If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.
    - If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.

  - **Interface vs Abstract Classes:**
    - An interface does not have implementation of any of its methods, it can be considered as a collection of method declarations. In C++, an interface can be simulated by making all methods as pure virtual.
    - We can think of Interface as header files in C++, like in header files we only provide the body of the class that is going to implement it.

- **Virtual Constructor:**
  - In C++, the constructor cannot be virtual, because when a constructor of a class is executed there is no virtual table in the memory, means no virtual pointer defined yet. So, the constructor should always be non-virtual.

- **Virtual Destructor:**
  - Deleting a derived class object using a pointer of base class type that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor.
  - Making base class destructor virtual guarantees that the object of derived class is destructed properly, i.e., both base class and derived class destructors are called.
  - As a guideline, any time we have a virtual function in a class, we should immediately add a virtual destructor (even if it does nothing).

- **Pure virtual destructor:**
  - It is possible to have pure virtual destructor. Pure virtual destructors are legal in standard C++.
  - One of the most important things to remember is that if a class contains a pure virtual destructor, it must provide a function body for the pure virtual destructor.
  - The reason is because destructors (unlike other functions) are not actually ‘overridden’, rather they are always called in the reverse order of the class derivation. This means that a derived class’ destructor will be invoked first, then base class destructor will be called.
  - If the definition of the pure virtual destructor is not provided, then what function body will be called during object destruction? Therefore, the compiler and linker enforce the existence of a function body for pure virtual destructors.
  - It is important to note that a class becomes abstract class when it contains a pure virtual destructor.

- **Can virtual functions be private?**
  - In C++, virtual functions can be private and can be overridden by the derived class.
  - Base class defines a public interface and derived class overrides it in its implementation even though derived has a private virtual function. But vice versa is not possible.

- **Can static functions be virtual?**
  - A *static* member function of a class cannot be *virtual*.
  - Also, *static* member function cannot be *const* and *volatile*.

- **Can virtual functions be Inlined?**
  - Inline functions are used for efficiency. The whole idea behind the inline functions is that whenever inline function is called code of inline function gets inserted or

substituted at the point of inline function call at compile time. Inline functions are very useful when small functions are frequently used and called in a program many times.

- By default, all the functions defined inside the class are implicitly or automatically considered as inline except virtual functions (Note that inline is a request to the compiler and its compilers choice to do inlining or not).
- Whenever virtual function is called using base class reference or pointer it cannot be inlined (because call is resolved at runtime), but whenever called using the object (without reference or pointer) of that class, can be inlined because compiler knows the exact class of the object at compile time.
- **Example:**

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

class Base
{
   public:
      virtual void who()
      {
         cout << "I am Base\n";
      }
};

class Derived: public Base
{
   public:
      void who()
      {
         cout << "I am Derived\n";
      }
};

int main()
{
   // Virtual function who() is called through object of the class,
   // it will be resolved at compile time, so it can be inlined.
   Base b;
   b.who();

   // Virtual function is called through pointer, so it cannot be inlined.
   Base *ptr = new Derived();
   ptr->who();

   return 0;
}
```