A virtual function is a member function which 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 function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a **virtual** keyword in base class.
- The resolving of function calls is done at runtime.

Rules for Virtual Functions

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 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 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 virtual destructor but it cannot have a virtual constructor.

Example without using virtual function:

```
#include <iostream>
using namespace std;
class Base{
        public:
        void show(){
        cout << "I am from Base!" << endl;
        }
};
class Derv1:public Base{
        public:
        void show(){
        cout << "I am from Derv1!" << endl;
        }
};
class Derv2:public Base{
        public:
        void show(){
        cout << "I am from Derv2!" << endl;
```

```
        }
};
int main() {
        Derv1 objDerv1;
        Derv2 objDerv2;
        Base *ptr;

        ptr = &objDerv1;
        ptr->show();

        ptr = &objDerv2;
        ptr->show();
        return 0;
}
```

Example Using Virtual Function:

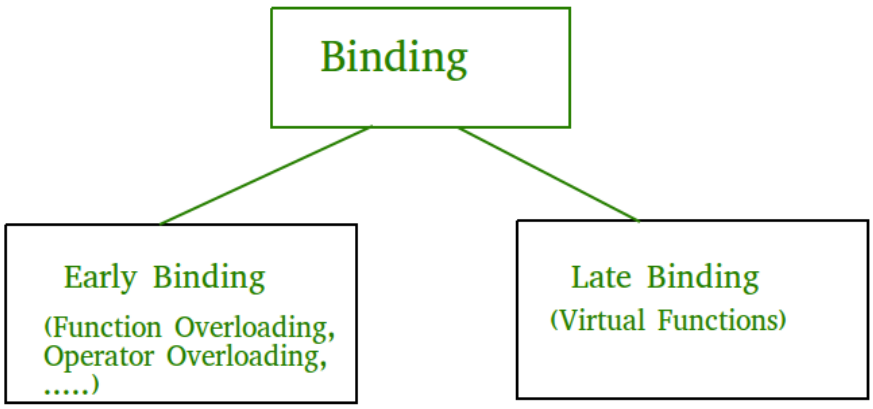
```
#include <iostream>
using namespace std;
class Base{
        public:
        virtual void show(){
        cout << "I am from Base!" << endl;
        }
};
class Derv1:public Base{
        public:
        void show(){
        cout << "I am from Derv1!" << endl;
        }
};
class Derv2:public Base{
        public:
        void show(){
        cout << "I am from Derv2!" << endl;
        }
};
int main() {
        Derv1 objDerv1;
        Derv2 objDerv2;
        Base *ptr;

        ptr = &objDerv1;
```

```
        ptr->show();

        ptr = &objDerv2;
        ptr->show();
        return 0;
}
```

Binding refers to the process of converting identifiers (such as variable and performance names) into addresses. Binding is done for each variable and functions. For functions, it means matching the call with the right function definition by the compiler. It takes place either at compile time or at runtime.



Runtime (late binding) behavior of Virtual Functions

```
// CPP 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 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 runtime (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

```
#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;
}
```

C++ Abstract Class and Pure Virtual Function

A pure virtual function is a virtual function in C++ for which we need not to write any function definition and only we have to declare it. It is declared by assigning 0 in the declaration.

An abstract class is a class in C++ which have at least one pure virtual function.

- Abstract class can have normal functions and variables along with a pure virtual function.
- Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.
- Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.
- If an Abstract Class has derived class, they must implement all pure virtual functions, or else they will become Abstract too.
- We can't create object of abstract class as we reserve a slot for a pure virtual function in Vtable, but we don't put any address, so Vtable will remain incomplete.

Example:

```
#include<iostream>
using namespace std;
class B {
  public:
    virtual void s() = 0; // Pure Virtual Function
};

class D:public B {
  public:
    void s() {
      cout << "Virtual Function in Derived class\n";
    }
};

int main() {
  B *b;
  D dobj;
  b = &dobj;
  b->s();
}
```

Output

```
Virtual Function in Derived class
```

C++ Pure Virtual Functions

Pure virtual functions are used

- if a function doesn't have any use in the base class
- but the function must be implemented by all its derived classes

Let's take an example,

Suppose, we have derived Triangle, Square and Circle classes from the Shape class, and we want to calculate the area of all these shapes.

In this case, we can create a pure virtual function named calculateArea() in the Shape. Since it's a pure virtual function, all derived classes Triangle, Square and Circle must include the calculateArea() function with implementation.

A pure virtual function doesn't have the function body and it must end with = 0. For example,

```
class Shape {
   public:

     // creating a pure virtual function
     virtual void calculateArea() = 0;
};
```

Note: The = 0 syntax doesn't mean we are assigning 0 to the function. It's just the way we define pure virtual functions.

Abstract Class

A class that contains a pure virtual function is known as an abstract class. In the above example, the class Shape is an abstract class.

We cannot create objects of an abstract class. However, we can derive classes from them, and use their data members and member functions (except pure virtual functions).

```
// C++ program to calculate the area of a square and a circle

#include <iostream>
using namespace std;

// Abstract class
class Shape {
  protected:
        float dimension;

  public:
        void getDimension() {
```

```
        cin >> dimension;
        }

        // pure virtual Function
        virtual float calculateArea() = 0;
};

// Derived class
class Square : public Shape {
  public:
        float calculateArea() {
        return dimension * dimension;
        }
};

// Derived class
class Circle : public Shape {
  public:
        float calculateArea() {
        return 3.14 * dimension * dimension;
        }
};

int main() {
        Square square;
        Circle circle;

        cout << "Enter the length of the square: ";
        square.getDimension();
        cout << "Area of square: " << square.calculateArea() << endl;

        cout << "\nEnter radius of the circle: ";
        circle.getDimension();
        cout << "Area of circle: " << circle.calculateArea() << endl;

        return 0;
}
```

Output

```
Enter the length of the square: 4
Area of square: 16

Enter radius of the circle: 5
Area of circle: 78.5
```

In this program, virtual float calculateArea() = 0; inside the Shape class is a pure virtual function.

That's why we must provide the implementation of calculateArea() in both of our derived classes, or else we will get an error.

Some Interesting Facts:
1) *A class is abstract if it has at least one pure virtual function.*

2) *We can have pointers and references of abstract class type.*

3) *If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.*

4) *An abstract class can have constructors.*

5) *An abstract class in C++ can also be defined using* ***struct*** *keyword.*

E.g. :

```
struct shapeClass

{

virtual void Draw()=0;

}
```