Friend and virtual functions.

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++

(https://www.javatpoint.com/cpp-friend-function#) (https://www.javatpoint.com/cpp-friend-function#) (https://www.javatpoint.com/cpp-friend-function#)

```
1. {
2. };
```

In the above declaration, the friend function is preceded by the keyword friend. The function can be defined anywhere in the program like a normal C++ function. The function definition does not use either the keyword **friend or scope resolution operator**.

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.

(https://www.javatpoint.com/cpp-friend-function#) (https://www.javatpoint.com/cpp-friend-function#) (https://www.javatpoint.com/cpp-friend-function#)

```
1. using namespace std;
2. {
3.         int length;
4.         Box(): length(0) { }
5. };
6. {
```

```
 7.     return b.length;
 8. int main()
 9.     Box b;
10.     return 0;
```

Output:

```
Length of box: 10
```

Let's see a simple example when the function is friendly to two classes.

(https://www.javatpoint.com/cpp-friend-function#) (https://www.javatpoint.com/cpp-friend-function#) (https://www.javatpoint.com/cpp-friend-function#)

```
 1. using namespace std;
 2. class A
 3.     int x;
 4.     void setdata(int i)
 5.         x=i;
 6.     friend void min(A,B);         // friend function.
 7. class B
 8.     int y;
 9.     void setdata(int i)
10.         y=i;
11.     friend void min(A,B);                 // friend function
12. void min(A a,B b)
13.     if(a.x<=b.y)
14.     else
15. }
16. {
17.    B b;
18.    b.setdata(20);
19.     return 0;
```

Output:

```
10
```

In the above example, min() function is friendly to two classes, i.e., the min() function can access the private members of both the classes A and B.

C++ Friend class

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

Let's see a simple example of a friend class.

(https://www.javatpoint.com/cpp-friend-function#) (https://www.javatpoint.com/cpp-friend-function#) (https://www.javatpoint.com/cpp-friend-function#)

```
1.
2.
3. {
4.     friend class B;          // friend class.
5. class B
6.   public:
7.     {
8.     }
9. int main()
10.     A a;
11.     b.display(a);
12. }
```

Output:

```
value of x is : 5
```

In the above 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.

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.

Late binding or Dynamic linkage

In late binding function call is resolved during runtime. Therefore compiler determines the type of object at runtime, and then binds the function call.

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.

(https://www.javatpoint.com/cpp-virtual-function#) (https://www.javatpoint.com/cpp-virtual-function#) (https://www.javatpoint.com/cpp-virtual-function#)

```
1. using namespace std;
2. {
3.     public:
4.     {
5.     }
6. class B: public A
7.     int y = 10;
8.     void display()
9.         std::cout << "Value of y is : " <<y<< std::endl;
10. };
11. {
12.     B b;
13.    a->display();
14. }
```

Output:

```
Value of x is : 5
```

In the above example, * a is the base class pointer. The pointer can only access the base class members but not the members of the derived class. Although C++ permits the base pointer to point to any object derived from the base class, it cannot directly access the members of the derived class. Therefore, there is a need for virtual function which allows the base pointer to access the members of the derived class.

C++ virtual function Example

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

(https://www.javatpoint.com/cpp-virtual-function#) (https://www.javatpoint.com/cpp-virtual-function#) (https://www.javatpoint.com/cpp-virtual-function#)

```
1. {
2.  virtual void display()
3.   cout << "Base class is invoked"<<endl;
4. };
5. {
6.  void display()
7.   cout << "Derived Class is invoked"<<endl;
8. };
9. {
10.  B b;     //object of derived class
11.  a->display();   //Late Binding occurs
```

Output:

```
Derived Class is invoked
```

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:

(https://www.javatpoint.com/cpp-virtual-function#) (https://www.javatpoint.com/cpp-virtual-function#) (https://www.javatpoint.com/cpp-virtual-function#)

Let's see a simple example:

(https://www.javatpoint.com/cpp-virtual-function#) (https://www.javatpoint.com/cpp-virtual-function#) (https://www.javatpoint.com/cpp-virtual-function#)

```
1. using namespace std;
2. {
3.     virtual void show() = 0;
4. class Derived : public Base
5.     public:
6.     {
7.     }
```

```
8. int main()
9.     Base *bptr;
10.    Derived d;
11.    bptr->show();
12. }
```

Output:

```
Derived class is derived from the base class.
```

In the above example, the base class contains the pure virtual function. Therefore, the base class is an abstract base class. We cannot create the object of the base class.