Virtual Function

Module 4

Virtual Function

- A virtual function is a member function which is declared within a base class and is re-defined(Overriden) 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 call is done at Run-time.

Rules for Virtual Functions

1. Virtual functions cannot be static and also cannot be a friend function of another class.
2. Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
3. The prototype of virtual functions should be same in base as well as derived class.
4. They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
5. A class may have virtual destructor but it cannot have a virtual constructor.

Virtual Function

```
#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;
//calling virtual function
bptr->print();
//calling non-virtual function
bptr->show(); }
```

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.
- **For example:** Consider an employee management software for an organization.
- Let the code has a simple base class *Employee* , the class contains virtual functions
  like *raiseSalary()*, *transfer()*, *promote()*, etc. Different types of employees like *Manager*, *Engineer*, etc. may have their own implementations of the virtual functions present in base class *Employee*.

- In our complete software, we just need to pass a list of employees everywhere and call appropriate functions without even knowing the type of employee. For example, we can easily raise the salary of all employees by iterating through the list of employees. Every type of employee may have its own logic in its class, but we don’t need to worry about them because if *raiseSalary()* is present for a specific employee type, only that function would be called.

```
class Employee {
public:
                virtual void raiseSalary()
                {
                                /* common raise salary code */
                }

                virtual void promote()
                {
                                /* common promote code */
                }
};

class Manager : public Employee {
                virtual void raiseSalary()
                {
                                /* Manager specific raise salary code, may contain
                                increment of manager specific incentives*/
                }

                virtual void promote()
                {
                                /* Manager specific promote */
                }
};

// Similarly, there may be other types of employees

// We need a very simple function
// to increment the salary of all employees
// Note that emp[] is an array of pointers
// and actual pointed objects can
// be any type of employees.
```

Inline virtual function in C++

- Virtual functions in C++ use 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. Virtual functions are resolved late, at the runtime.
- The main use of the virtual function is to achieve Runtime Polymorphism. The inline functions are used to increase the efficiency of the code. The code of inline function gets substituted at the point of an inline function call at compile time, whenever the inline function is called.
- Whenever a virtual function is called using base class reference or pointer it cannot be inlined, but whenever called using the object without reference or pointer of that class, can be inlined because the compiler knows the exact class of the object at compile time.

```
#include<iostream>
using namespace std;
class B {    public:
virtual void s()
{           cout<<" In Base \n";        } };
class D: public B {    public:
void s()
{           cout<<"In Derived \n";        } };
int main(void)
{
B b;
D d;
// An object of class D
B *bptr = &d;
// A pointer of type B* pointing to d
b.s()
;//Can be inlined as s() is called through object of class
bptr->s();
// prints"D::s() called"
//cannot be inlined, as virtualfunction is called through pointer.
return 0;
}
```

Virtual Function

- A **virtual** function is a function in a base class that is declared using the keyword **virtual**. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.
- What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as **dynamic linkage**, or **late binding**.

Pure Virtual Functions

- It is possible that you want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

```
class Shape {
  protected:
    int width, height;
  public:
    Shape(int a = 0, int b = 0) {
      width = a;
      height = b;      }
    // pure virtual function
    virtual int area() = 0; };
```

The = 0 tells the compiler that the function has no body and above virtual function will be called **pure virtual function**.

Compile-time(early binding) VS run-time(late binding) behaviour of Virtual Functions

```
#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;
                }
```