VIRTUAL FUNCTIONS

A **virtual function** a member **function** which is declared within base class and is re-defined (Overriden) by 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. They Must be declared in public section of class.

2. Virtual functions cannot be static and also cannot be a friend function of another class.

3. Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.

4. The prototype of virtual functions should be same in base as well as derived class.

5. 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.

6. A class may have virtual destructor but it cannot have a virtual constructor.

PURE VIRTUAL FUNCTIONS

- Sometimes implementation of all **function** cannot be provided in a base class because we don't know the implementation.
- A **pure virtual function** (or abstract **function) in C++** is a **virtual function** for which we don't have implementation, we only declare it.
- A **pure virtual function** is declared by assigning 0 in declaration.

We can change the virtual function area() in the base class to the following −

```
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**.

Interfaces in C++ (Abstract Classes)

An interface describes the behaviour or capabilities of a C++ class without committing to a particular implementation of that class.

The C++ interfaces are implemented using **abstract classes** and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.

A class is made abstract by declaring at least one of its functions as **pure virtual** function.

A pure virtual function is specified by placing "= 0" in its declaration as in following example code:

```
class Box
{
   public:

  // pure virtual function
   virtual double getVolume() = 0;

   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};
```

// CPP program to illustrate concept 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; }
};
```

```
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
```

NOTE: If we have created virtual function in base class and it is being overrided in derived class then we don't need virtual keyword in derived class, functions are automatically considered as virtual functions in derived class.

- The purpose of an **abstract class** is to provide an appropriate base class from which other classes can inherit.
- Abstract classes cannot be used to instantiate objects and serves only as an **interface**.
- Attempting to instantiate an object of an abstract class causes a compilation error.
- Thus, if a subclass of an Abstract classes needs to be instantiated, it has to implement each of the virtual functions, which means that it supports the interface declared by the Abstract class.
- Failure to override a pure virtual function in a derived class, then attempting to instantiate objects of that class, is a compilation error.
- Classes that can be used to instantiate objects are called **concrete classes**.

Example: Abstract Class

Consider the following example where parent class provides an interface to the base class to implement a function called **getArea()** –

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

// Base class
class Shape
 {
  public:

    /* pure virtual function providing
       interface framework.  */

    virtual int getArea() = 0;

    void setWidth(int w)
     {
       width = w;
     }

     void setHeight(int h)
    {
       height = h;
     }

  protected:
     int width;
     int height;
};
```

```
// Derived classes

class Rectangle: public Shape
{
  public:
    int getArea() {
      return (width * height);
    }
};

class Triangle: public Shape
 {
  public:
    int getArea() {
      return (width * height)/2;
    }
};
```

```
int main(void)
{
  Rectangle Rect;
  Triangle  Tri;

  Rect.setWidth(5);
  Rect.setHeight(7);

  // Print the area of the object.
  cout << "Total Rectangle area: " << Rect.getArea() << endl;

  Tri.setWidth(5);
  Tri.setHeight(7);

  // Print the area of the object.
  cout << "Total Triangle area: " << Tri.getArea() << endl;

  return 0;
}
```

Output:

Total Rectangle area: 35
Total Triangle area: 17

You can see how an abstract class defined an interface in terms of getArea() and two other classes implemented same function but with different algorithm to calculate the area specific to the shape.