SDF II(15B11CI211)

EVEN Semester 2021

2nd Semester , First Year

Jaypee Institute Of Information Technology (JIIT), Noida

Topics covered:

1.Virtual Functions
2. Pure Virtual Function
3. Abstract Class
4. Virtual Destructor
5. Pure Virtual Destructor

Function Call Binding with class Objects

Connecting the function call to the function body is called Binding. When it is done before the program is run, it is called Early Binding or Static Binding or Compile-time Binding.

```
class Base
{
   public:
   void show()
   {
      cout << "Base class\n";
   }
};
```

```
class Derived: public Base
{
   public:
   void show()
   {
      cout << "Derived Class\n";
   }
}
int main()
{
   Base b;       //Base class object
   Derived d;     //Derived class object
   b.show();     //Early Binding Ocuurs
   d.show();
}
```

Function Call Binding with class Objects

```
class Base
{
   public:
   void show()
   {
      cout << "Base class\n";
   }
};
class Derived: public Base
{
   public:
   void show()
   {
      cout << "Derived Class\n";
   }
}
```

```
int main()
{
   Base* b;       //Base class pointer
   Derived d;     //Derived class object
   b = &d;
   b->show();     //Early Binding Occurs
}
```

In the above example, although, the object is of Derived class, still Base class's method is called. This happens due to Early Binding.

Compiler on seeing Base class's pointer, set call to Base class's show() function, without knowing the actual object type.

Virtual Functions in C++

Virtual Function is a function in base class, which is overridden in the derived class, and which tells the compiler to perform Late Binding on this function.

Virtual Keyword is used to make a member function of the base class Virtual.

```
class Base
{
   public:
   virtual void show()
   {
      cout << "Base class\n";
   }
};
```

```
class Derived:public Base
{
   public:
   void show()
   {
      cout << "Derived Class";
   }
}
int main()
{
   Base* b;       //Base class pointer
   Derived d;     //Derived class object
   b = &d;
   b->show();     //Late Binding Ocurrs
}
```

```
Derived Class
```

On using Virtual keyword with Base class's function, Late Binding takes place and the derived version of function will be called, because base class pointer pointes to Derived class object.

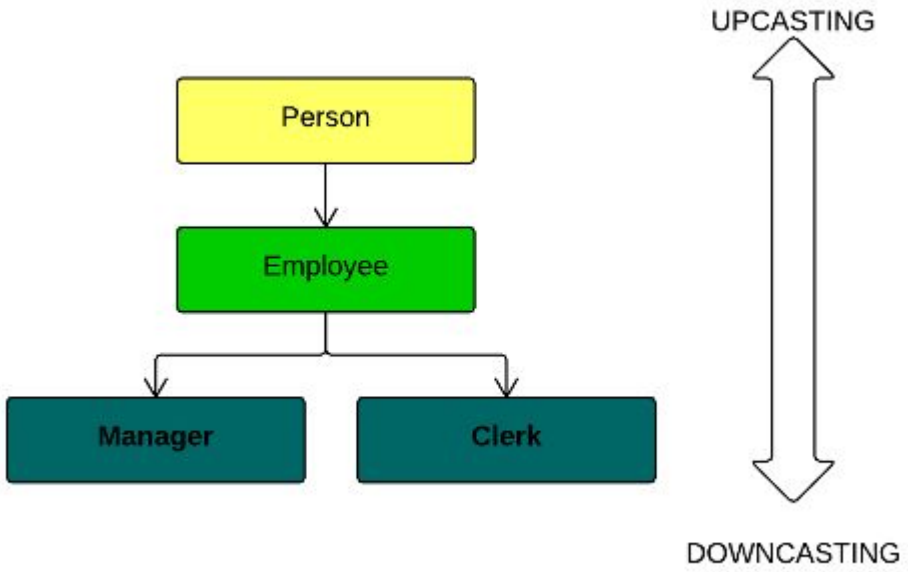
Virtual Functions in C++

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

class Base
{
   public:
   virtual void show()
   {
      cout << "Base class\n";
   }
};
class Derived:public Base
{
   public:
   void show()
   {
      cout << "Derived Class";
   }
};
int main()
{
   Base* b; //Base class pointer
   Derived* f;
   Base e;
   Derived d;     //Derived class object
   b = &d;
   b->show();     //Late Binding Occurs
   b= &e;
   b->show();
   cout<<"derived pointer";
   f = &d;
   f->show();
 // f= &e;         //Error
   f->show();

}
```

Upcasting and Downcasting



C++ allows that a derived class pointer (or reference) to be treated as base class pointer. This is upcasting.

Downcasting is an opposite process, which consists in converting base class pointer (or reference) to derived class pointer.

UPCASTING

Upcasting is a process of treating a pointer or a reference of derived class object as a base class pointer. You do not need to upcast manually. You just need to assign derived class pointer (or reference) to base class pointer, in other words address of derived object in base class pointer.

Virtual Function in C++

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.

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.

Virtual Function in C++

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 Functions in C++

```
#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();
}
```

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

Virtual Function in C++

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.

Virtual Functions in C++

```
#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
    p->fun_2();
    // Late binding
    p->fun_3();
    // Late binding
    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);
}
```

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

Can static functions be virtual in C++?

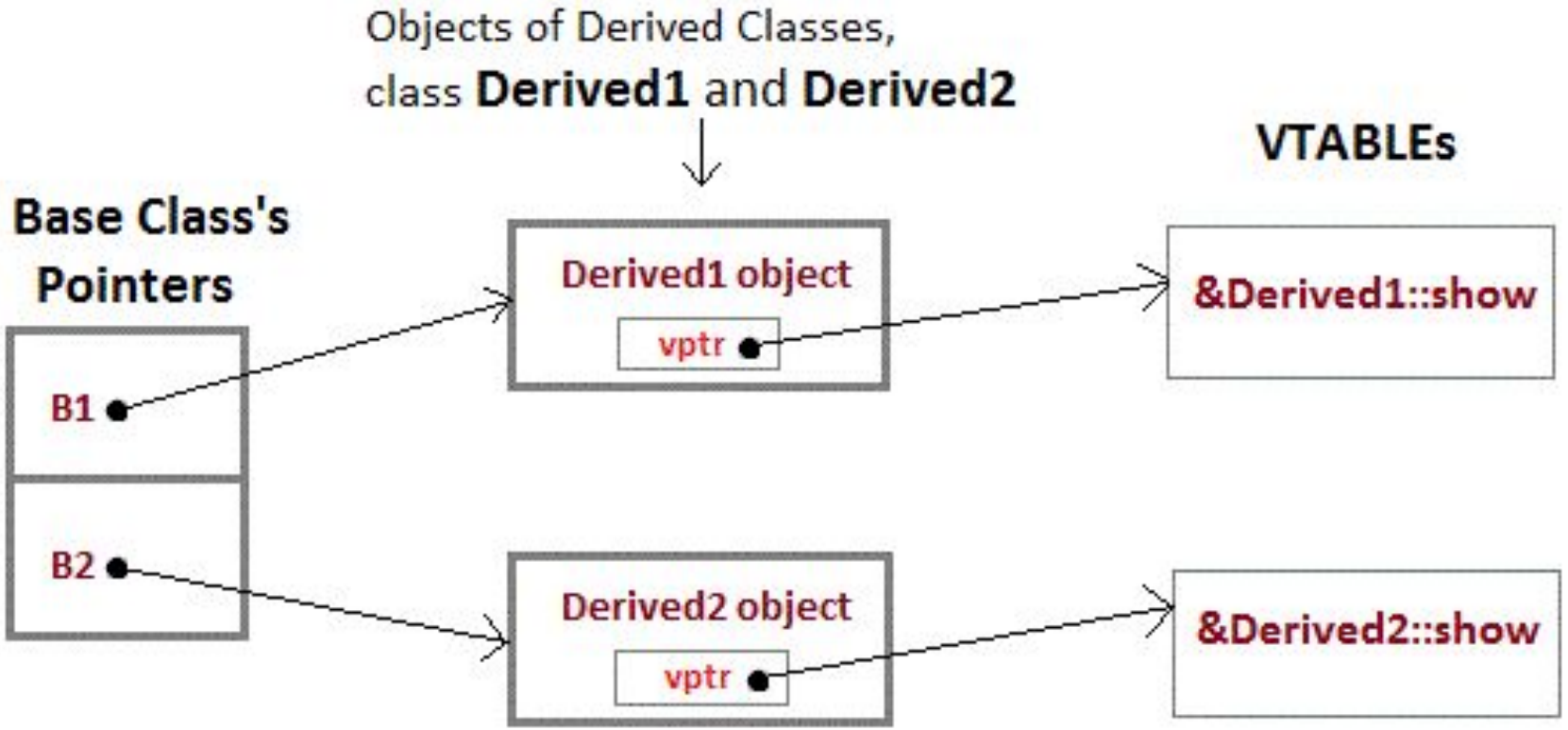
In C++, a static member function of a class cannot be virtual. For example, below program gives compilation error.

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

class Test
{
  public:
     // Error: Virtual member functions
cannot be static
    virtual static void fun()  { }
};
```

Virtual functions are invoked when you have a pointer/reference to an instance of a class. Static functions aren't tied to a particular instance, they're tied to a class. C++ doesn't have pointers-to-class, so there is no scenario in which you could invoke a static function virtually.

Mechanism of Late Binding in C++



vptr, is the vpointer, which points to the Virtual Function for that object.

VTABLE, is the table containing address of Virtual Functions of each class.

Mechanism of Late Binding in C++

To accomplish late binding, Compiler creates VTABLEs, for each class with virtual function. The address of virtual functions is inserted into these tables. Whenever an object of such class is created the compiler secretly inserts a pointer called vpointer, pointing to VTABLE for that object. Hence when function is called, compiler is able to resolve the call by binding the correct function using the vpointer.

Pure Virtual Function and Abstract Class in C++

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.

```
class Base
{
   public:
   virtual void show() = 0;    // Pure Virtual Function
};
class Derived:public Base
{
 public:void show()
{
cout << "Implementation of Virtual Function in Derived class\n";
 }
};
int main()
{
   Base obj;   //Compile Time Error
   Base *b;
   Derived d;
   b = &d;
   b->show();
}
```

Pure Virtual definitions

```
// Abstract base class
class Base
{
   public:
   virtual void show() = 0;     //Pure Virtual Function
};
void Base :: show()     //Pure Virtual definition
{
   cout << "Pure Virtual definition\n";
}
class Derived: public Base
{   public:
   void show()
   {
      cout << "Implementation of Virtual Function in Derived class\n";
   }
};
```

```
int main()
{
   Base *b;
   Derived d;
   b = &d;
   b->show();
}
```

Pure Virtual functions can be given a small definition in the Abstract class, which you want all the derived classes to have. Still, you cannot create object of Abstract class.
Also, the Pure Virtual function must be defined outside the class definition. If you will define it inside the class definition, complier will give an error. Inline pure virtual definition is Illegal.

Pure Virtual Function and Abstract Class in C++

Abstract Class is a class which contains **at least** one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Characteristics of Abstract Class

Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.

Abstract class can have normal functions, constructor and variables along with a pure virtual function.

Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.

Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

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

Abstract Class

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. See the following example.

```
/ / An abstract class
class Test
{
   // Data members of class
public:
   // Pure Virtual Function
   virtual void show() = 0;

  /* Other members */
};
```

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

The **C++ interfaces** are implemented using abstract classes.

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 follows −

```
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
};
```

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.

```
class Test
{
int x;
public:
    virtual void show() = 0;
    int getX() { return x; }
};
int main(void)
{
    Test t;
    return 0;
}
```

```
Compiler Error: cannot declare variable 't' to be of abstract
 type 'Test' because the following virtual functions are pure
within 'Test': note:     virtual void Test::show()
```

An abstract class can have constructors.

```
#include<iostream>
using namespace std;
// An abstract class with constructor
class Base
{
protected:
int x;
public:
virtual void fun() = 0;
Base(int i) { x = i; }
};
class Derived: public Base
{
    int y;
public:
Derived(int i, int j):Base(i) { y = j; }
void fun() { cout << "x = " << x << ", y = " << y; }
};
int main(void)
{
    Derived d(4, 5);
    d.fun();
    return 0;
}
```

Abstract class can have normal functions, constructor and variables along with a pure virtual function.

```
#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;
 }
```

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

```
#include<iostream>
using namespace std;
class Base
{
public:
    virtual void show() = 0;
};
class Derived : public Base { };
int main(void)
{
  Derived d;
  return 0;
}
```

```
Compiler Error: cannot declare variable 'd' to be of abstract type
'Derived'  because the following virtual functions are pure within
'Derived': virtual void Base::show()
```

Virtual Destructor in C++

Deleting a derived class object using a pointer to a base class that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor. For example, following program results in undefined behavior.

```
#include<iostream>
using namespace std;
class base {
 public:
   base()
   { cout<<"Constructing base \n"; }
   ~base()
   { cout<<"Destructing base \n"; }
};
class derived: public base {
  public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
};
int main(void)
{
  derived *d = new derived();
  base *b = d;
  delete b;
  getchar();
  return 0;
}
```

```
Constructing base
Constructing derived
Destructing base
```

Virtual Destructor in C++

Deleting a derived class object using a pointer to a base class that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor. For example, following program results in undefined behavior.

```
#include<iostream>
using namespace std;
class base {
  public:
    base()
    { cout<<"Constructing base \n"; }
    virtual ~base()
    { cout<<"Destructing base \n"; }
};
class derived: public base {
  public:
    derived()
    { cout<<"Constructing derived \n"; }
    ~derived()
    { cout<<"Destructing derived \n"; }
};
int main(void)
{
  derived *d = new derived();
  base *b = d;
  delete b;
  getchar();
  return 0;
}
```

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```

Pure Virtual Destructor in C++

Yes, it is possible to have pure virtual destructor

```
#include <iostream>
class Base
{
public:
    virtual ~Base()=0; // Pure virtual destructor
};
Base::~Base()
{
    std::cout << "Pure virtual destructor is
called";
}
class Derived : public Base
{
public:
    ~Derived()
    {
        std::cout << "~Derived() is executed\n";
    }
};

int main()
{
    Base *b = new Derived();
    delete b;
    return 0;
}
```

The Content is prepared with the help of existing websites and textbooks mentioned below:

- John Hubbard, Schaum's Outline of Programming with C++, McGraw-Hill, 2nd Edition, 2000
- Herbert Schildt, C++: The Complete Reference, McGraw-Hill Osborne Media, 4th Edition, 2017
- https://www.geeksforgeeks.org/pure-virtual-functions-and-abstract-classes/
- https://www.studytonight.com/cpp/virtual-functions.php
- https://www.programiz.com/cpp-programming/pure-virtual-funtion