Virtual Functions

Name : Kunal Moharkar

Roll no : BT18CSE018

Inheritance :

The capability of a class to derive properties and characteristics from another class is called **Inheritance .** Inheritance is one of the most important feature of **Object Oriented Programming**. The class whose properties are inherited is called **Base/Super/Parent class** and the one which inherits is called **Sub/Derived class .**

What are virtual functions ?

In a class hierarchy , virtual function is a member function of a base class which is then **overridden** or redefined 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. It is used to achieve **run-time polymorphism .**

Key features of virtual functions :

- Virtual Function is a special type of function
- It resolves to the most-derived function that exists between the base and derived class .
- This feature is known as polymorphism .
- This is also called as function overriding .
- Functions should have the same signature name, parameter types, return type .

Pure virtual functions :

- Pure Virtual Function does not have a body .
- It simply acts as a placeholder which is redefined by the derived classes .
- Class with pure virtual function becomes an abstract base class .

- Abstract base class cannot be instantiated .

Need for virtual functions :

Without virtual we get **early binding** . Which implementation of the method is used gets decided at compile time based on the type of the pointer that you call through.

With virtual we get **late binding.** Which implementation of the method is used gets decided at run time based on the type of the pointed-to object - what it was originally constructed as.

Consider the code below :

```
class Sport                   //Base class Sport
{
  public:
    void getId()
    {
      cout<<"Sport id"<<"\n";
    }
    void getNum()
    {
      cout<<"Sport num"<<"\n";
    }
};

class Cricket : public Sport  //class Cricket derived from Sport
{
  public:
   void getId()               //overridden method
    {
      cout<<"Cricket id"<<"\n";
    }
};
```

now consider the following function call :

```
Sport objS;
Cricket *ptrC = &objS;
ptrC->getId();
```

Due to early binding , `getId()` method of class Cricket will be invoked as `ptrC` is a pointer of type Cricket . `Output : Cricket id` . However this is conceptually wrong as `ptrC` points to `objS` which is instance of Sport class and hence , `getId()` of class Sport should be invoked .

`Expected output : Sport id .`

This problem can be solved by using virtual functions that uses late binding by maintaining a Virtual Table during run-time .

Virtual Table :

The virtual table is a lookup table of functions used to resolve function calls in a **late binding** manner. A virtual table contains one entry for each virtual function that can be called by objects of the class. Each entry in this table is simply a function pointer that points to the **most-derived** function accessible by that class. These tables are constructed during the compile time and maintained throughout run-time.

Sample code :

```
class Sport //Base class Sport
{
  private:
    int id;
    int num;

  public:
    virtual void getId()
    {
      cout<<"Sport id"<<"\n";
    }
    virtual void getNum()
    {
      cout<<"Sport num"<<"\n";
    }
};

class Cricket : public Sport  //class Cricket derived from Sport
{
  private:
    char code;

  public:
    virtual void getId()    //overridden method
    {
      cout<<"Cricket id"<<"\n";
    }
};
```

Function Invoking :

1.

```
Sport objS;
Sport *ptrS = &objS;
ptrS->getId();
```

In the above example we create a instance of class Sport as `ObjS` and `ptrS` is the pointer to `objS`. It invokes method `getId()` . Now the pointer to the virtual table of Class Sport is traced and from it the appropriate function is invoked . Here , `S : getId() ;` is invoked .

`Output : Sport id`

2.

```
Cricket objC;
Cricket *ptrC = &objC;
ptrC->getId();
```

Similarly , we create a instance of class Cricket as `ObjC` and `ptrC` is the pointer to `objC`. It invokes method `getId()` . Now the pointer to the virtual table of Class Cricket is traced and from it the appropriate function is invoked . Here , `C : getId() ;` is invoked .

`Output : Cricket id`

3.

```
Cricket objC;
Cricket *ptrC = &objC;
ptrC->getNum();
```

We create a instance of class Cricket as `ObjC` and `ptrC` is the pointer to `objC`. It invokes method `getNum()` . Now the pointer to the virtual table of Class Cricket is traced . but there is no function `getNum()` for derived class Cricket . Hence , getNum() of base class Sport is is invoked . Here , `S:getNum()` is invoked .

`Output : Sport num`

---

4 .

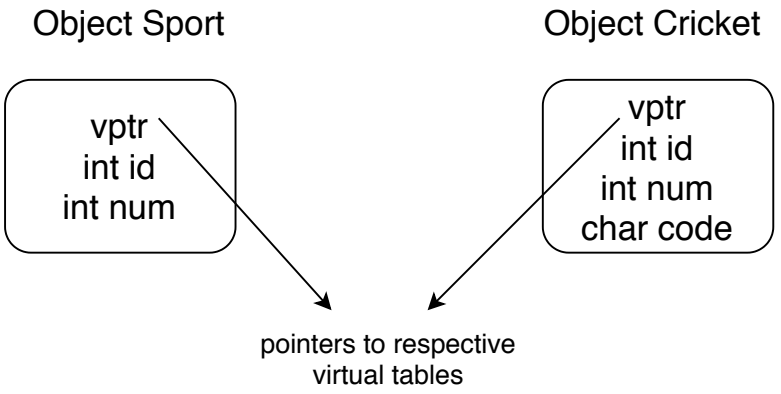
```
Sport objS;
Cricket *ptrC = &objS;
ptrC->getId();
```

This is case we discussed earlier that outputs : `C:getId() : Cricket id ;` without virtual functions . But now due to late binding the function invoking is resolved appropriately using the virtual table . and `S:getId() : Sport id ;` is the output .

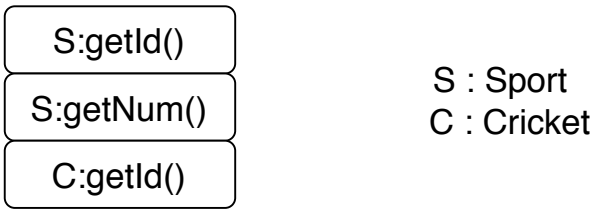
Schematic Representation :

Virtual Tables

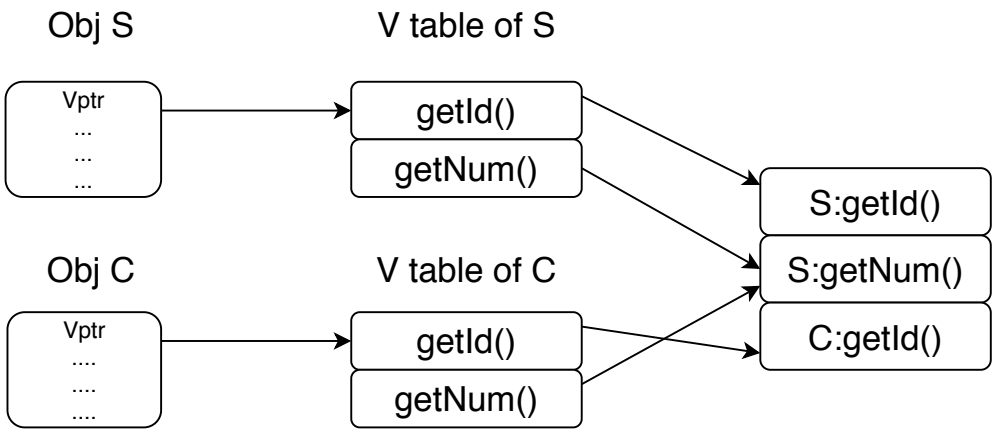
Objects



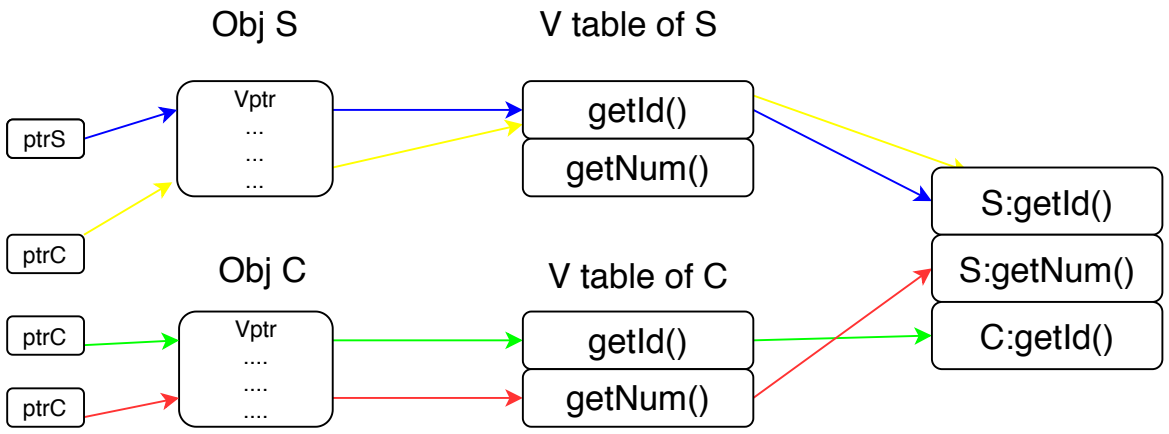
Function segment



Linkings



Function Invoking



1. ptrS ->getId();
2.ptrC ->getId();
3. ptrC ->getNum();
4. ptrC->getId();