CPL ASSIGNMENT - 4

Harshal Deshpande
(BT18CSE079)

(* All diagrams are made using Microsoft Word. The .cpp file of code explained here is also linked in the email body itself *)

Why virtual functions are needed?

Function overriding is when a function with the same signature is present in both parent as well as child class. **Overriding fails** when **pointers of parent objects** are used to **invoke child member functions** (the same pointer can refer to both parent and child objects). This problem is handled by virtual functions.

Definition:

A virtual function is a member function which is declared within a base class and is 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.

P.T.O

Below is a C++ code to demonstrate working of virtual functions :

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

class base
{
    public:
        void function_1() { cout << "base->function_1\n"; }
        virtual void function_2() { cout << "base->function_2\n"; }
        virtual void function_3() { cout << "base->function_3\n"; }
        virtual void function_4() { cout << "base->function_4\n"; }
};

class derived : public base
{
    public:
        void function_1() { cout << "derived->function_1\n"; }
        void function_2() { cout << "derived->function_2\n"; }
        void function_4(int x) { cout << "derived->function_4\n"; }
};

int main()
{
    base *basePtr, obj1;
    basePtr = &obj1;

    basePtr->function_1();  //Early binding because function1() is non-virtual
    basePtr->function_2();    // Late binding
    basePtr->function_3();    // Late binding
    basePtr->function_4();    // Late binding
    basePtr->function_4(10);  // Early binding

    derived obj2;
    basePtr = &obj2;

    basePtr->function_1();  //Early binding because function1() is non-virtual
    basePtr->function_2();    // Late binding
    basePtr->function_3();    // Late binding
    basePtr->function_4();    // Late binding
    basePtr->function_4(10);  // Early binding
}
```

```
virtual.cpp:10:22: note:   candidate expects 0 arguments, 1 provided
ubuntu@ubuntu-HP-Pavilion-Laptop-15-cs2xxx:~/Desktop$ g++ virtual.cpp
ubuntu@ubuntu-HP-Pavilion-Laptop-15-cs2xxx:~/Desktop$ ./a.out
base->function_1
base->function_2
base->function_3
base->function_4
base->function_1
derived->function_2
base->function_3
base->function_4
ubuntu@ubuntu-HP-Pavilion-Laptop-15-cs2xxx:~/Desktop$ g++ -fdump-class-hierarchy virtual.
```

Compiler generates a **Virtual Table (**for all classes that have at least one virtual function and its descendants) which is a **static array that contains function pointers**. As it is a static array it will be allocated memory even if class object is not created. Each class will have a unique virtual table (vtable).

Virtual table base class

Address of Base::function_2() base class
Address of Base::function_3() base class
Address of Base::function_4() base class

Virtual table derived class

Address of Derived::function_2() derived class
Address of Base::function_3() base class
Address of Base:function_4() base class

Above diagrams are just for representation. Below in actual Vtable generated by gcc compiler.

Command used g++ -fdumpclass-hierarchy filename.cpp

```
Vtable for base
base::_ZTV4base: 5 entries
0     (int (*)(...))0
8     (int (*)(...))(& _ZTI4base)
16    (int (*)(...))base::function_2
24    (int (*)(...))base::function_3
32    (int (*)(...))base::function_4
```

```
Vtable for derived
derived::_ZTV7derived: 5 entries
0     (int (*)(...))0
8     (int (*)(...))(& _ZTI7derived)
16    (int (*)(...))derived::function_2
24    (int (*)(...))base::function_3
32    (int (*)(...))base::function_4
```

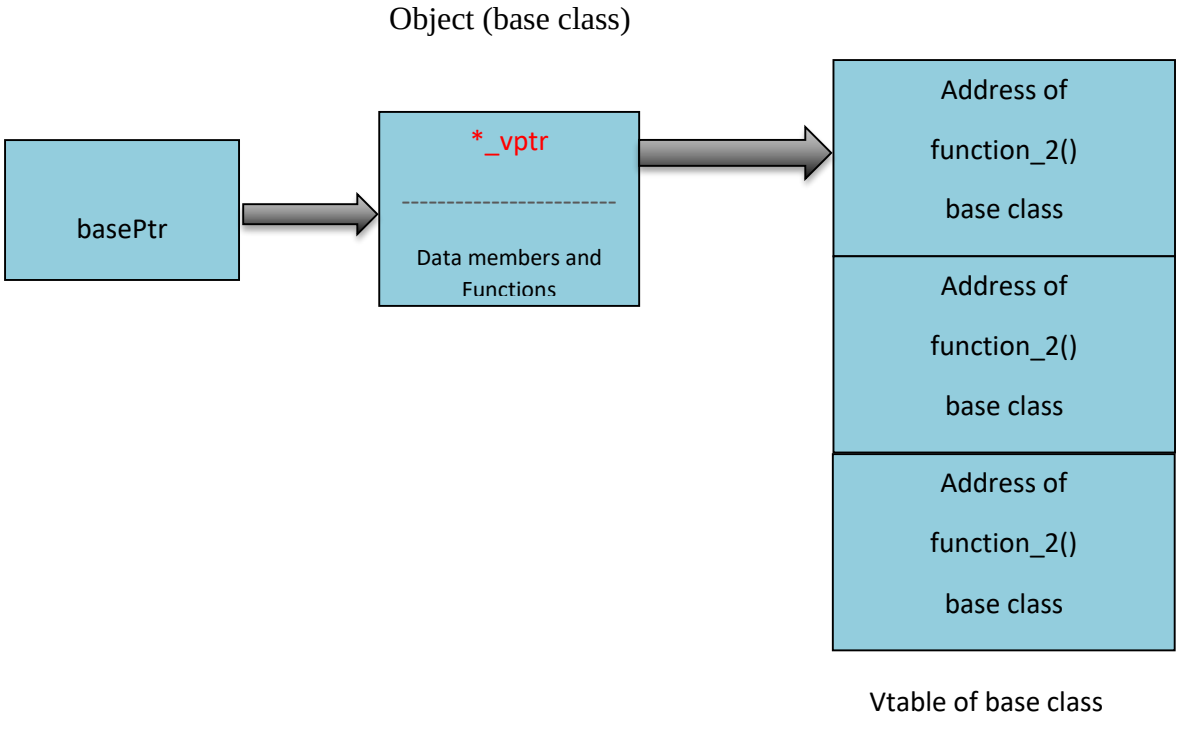
Explanation:

- function_1() is not virtual hence not present in vtable.
- Vtable of class base contains addresses of virtual functions of it.
- Vtable of derived class contains function_3() of base because it is not overridden in it
- Vtable of derived class contains function_4() of base because it is overridden in it but the signature if different

Object creation:

When object is created, compiler allocates memory for it and for every object implicitly adds a **data member** which is a **pointer to the vtable** of that class.

```
base *basePtr, obj1;
basePtr = &obj1;
```



Class base {

*_vptr; // implicitly added by compiler

// Data members and functions

}

Compiler adds this pointer to only base class (To avoid multiple copies during inheritance). When derived class object is created this pointer is inherited (But it now points to respective vtable of that class). Similar steps are followed for object creation of derived classes.

Function Invocation:

1) Using base class pointer and base object -

```
base *basePtr, obj1;
basePtr = &obj1;
basePtr->function_1();    // Early binding
```

Output : base->function_1

Function is not virtual. Hence early binding is done (that is function to be called it determined at compile time).

```
basePtr->function_2();    // Late binding
basePtr->function_3();    // Late binding
basePtr->function_4();    // Late binding
```

Output: base->function_2

base->function_3

base->function_4

These functions being virtual the function to be executed is decided at runtime. BasePtr points to object of base class. Then, *_vptr in that object points to vtable of base class. Hence, using function pointer present in vtable respective function of base class is invoked.

```
basePtr->function_4(10);  // Early binding
```

Here, early binding (because not virtual). Early binding but this function call is illegal (produces error) because pointer is of base type and function is of derived class

2) Base class pointer with derived class object:

```
Derived obj2;
basePtr = &obj2;
basePtr->function_1();
```

Output: base->function_1 (This is why virtual functions are required)

Function is not virtual and hence early binding is done (that is function to be called it determined at compile time). So even if **object is of derived type, but function of base class is invoked** as the reference (pointer) used if of base class. This is disadvantage of overriding.

```
basePtr->function_2();     // Late binding
```

Output: base->function_2

Function is virtual so the function to be executed is decide at runtime. DerivedPtr points to object of derived class. Then, *_vptr in that object points to vtable of derived class. Hence, using function pointer present in vtable respective function of derived class is invoked.

```
basePtr->function_3();     // Late binding
basePtr->function_4();     // Late binding
```

Output: base->function_3

base->function_4

Here even if the functions are virtual but they are either not inherited in the derived class or have different signatures. So, the function pointers in vtable point to the base class functions only!! And hence they are invoked.

```
    derivedPtr->function_4(10);  // Early binding
```

Here, early binding takes place (because it is not a virtual function). But this function call is illegal (produces error) because pointer is of base type so compiler looks for function in case class but no function with given signature is present.