Object oriented programming

Binding, Polymorphism, Static binding and Dynamic binding, Overloading, and Overriding

What is Polymorphism?

- Polymorphism is a combination of two words. Poly means many and morphism means form.

- In OOP, polymorphism is the ability of objects of different types to respond to functions of the same name differently.

```
class A{
public: void my_features()
{ cout << “A \n ";
}};
class B : public A {
public: void my_features() {
cout << "B \n ";
}};
class C : public A{
public:
void my_features()
{ cout << "C";
 }};
```

```
int main(){
A *obj1 = new A;
B *obj2 = new B;
C *obj3 = new C;

obj1->my_features();
obj2->my_features();
obj3->my_features();

delete obj1;
delete obj2;
delete obj3;
 return 0;
}
```

- Output

A

B

C

We created three pointer objects *obj1, *obj2 and *obj3. And when we call function my_features() using these pointer objects, the corresponding functions of the classes are executed.

Now, what if we created an intermediate function to which we can pass objects of the corresponding class to invoke functions. This should also do the same.

```
class A{
public: void my_features()
{ cout << "A \n";
}
};

class B : public A {
public: void my_features() {
cout << "B \n";
}
};

class C : public A{
public:
void my_features()
{ cout << "C";
 }
};
```

```
//intermediate function
void intermediate_func(A *a1)
{
a1->my_features();
}

int main()
{
A *obj1 = new A;
B *obj2 = new B;
C *obj3 = new C;
intermediate_func(obj1);
intermediate_func(obj2);
intermediate_func(obj3);
return 0;
}
```

- Output

A

A

A

- However, the desired output was:

A

B

C

How to get desired output? (through polymorphism that is achieved through virtual functions)

Virtual function

- A virtual function is a member function of the base class, that is **overridden** in derived class. The compiler binds virtual function at runtime, hence called **runtime polymorphism**. Use of virtual function allows the program to decide at runtime which function is to be called based on the type of the object pointed by the pointer.
- In C++, the member function of a class is selected at runtime using virtual function. The function in the base class is overridden by the function with the same name of the derived class.
- Once a function is declared virtual in a class then for all its derived classes that function will remain virtual.

Overriding

- Same signatures
- Different scope

C++ virtual function : Syntax

```
class class_name
 {
   public:
     virtual return_type func_name( args.. )
      {
        //function definition
      }
 };
```

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

class B : public A {
public:
virtual void my_features() {
cout << “B \n";
}};
class C : public A{
public:
virtual void my_features()
{ cout << "C";
 }};
```

```
void intermediate_func(A *a1)
{
a1->my_features();
}
int main()
{
A *obj1 = new A;
B *obj2 = new B;
C *obj3 = new C;
intermediate_func(obj1);
intermediate_func(obj2);
intermediate_func(obj3);
return 0;
}
```

V-table

- Every class that has one or more virtual member functions in it has a vTable associated with it. vTable is a kind of function pointer array that contains the addresses of all virtual functions of this class. Compiler builds this vTable at compile time.
- A virtual table contains one entry as a function pointer for each virtual function which is the most derived version of the function.

V-pointer

- For every object of a class that has a vTable associated with it, contains a vPointer. This vPointer points to the vTable of that class. This vPointer will be used to find the actual function address from vTable at run rime.

- This vtable pointer or _vptr, is a hidden pointer added by the Compiler to the base class. And this pointer is pointing to the virtual table of that particular class.

- _vptr is inherited to all the derived classes.

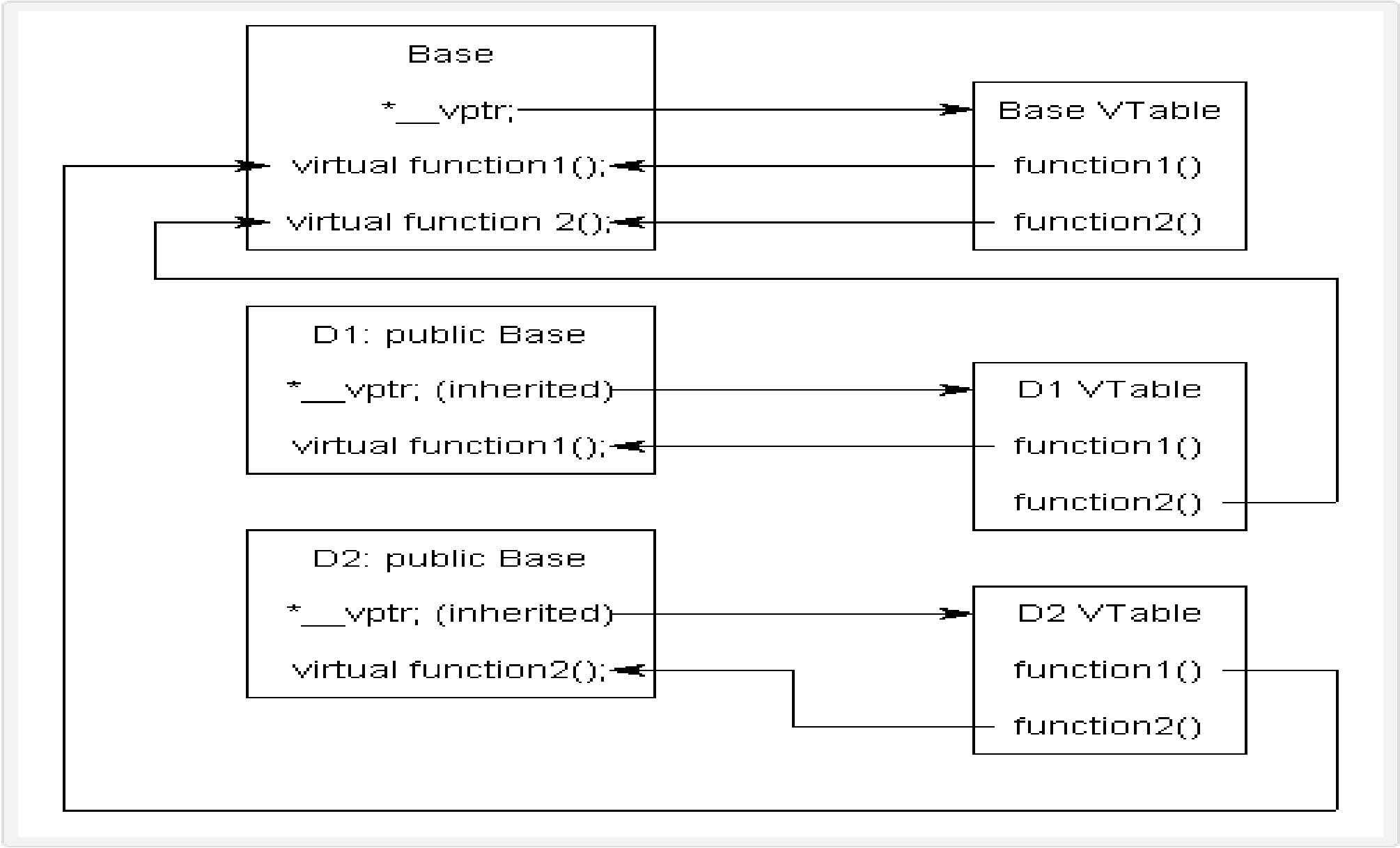
- Each object of a class with virtual functions transparently stores this _vptr.

- Call to a virtual function by an object is resolved by following this hidden _vptr.

```
class Base
{
public:
   virtual void function1() {}
   virtual void function2() {}
};

class D1: public Base
{
public:
   virtual void function1() {}
};

class D2: public Base
{
public:
   virtual void function2() {}
};
```

- Because there are 3 classes here, the compiler will set up 3 virtual tables: one for Base, one for D1, and one for D2.
- The compiler also adds a hidden pointer to the most base class that uses virtual functions.

```
class Base
{
public:
   FunctionPointer *__vptr;
   virtual void function1() {}
   virtual void function2() {}
};

class D1: public Base
{
public:
   virtual void function1() {}
};

class D2: public Base
{
public:
   virtual void function2() {}
};
```


Base
*__vptr;
virtual function1();
virtual function 2();
Base VTable
function1()
function2()
D1: public Base
*__vptr; (inherited)
virtual function1();
D1 VTable
function1()
function2()
D2: public Base
*__vptr; (inherited)
virtual function2();
D2 VTable
function1()
function2()

```
int main()
{
   D1 d1;
}
```

- d1 is a D1 object, d1 has its *__vptr set to the D1 virtual table.
- Now, add another line to the above code that sets a base pointer to D1

```
int main()
{
   D1 d1;
   Base *dPtr = &d1;
    return 0;
}
```

because dPtr is a base pointer, it only points to the Base portion of d1. However, also note that *__vptr is in the Base portion of the class, so dPtr has access to this pointer. Finally, note that dPtr->__vptr points to the D1 virtual table! Consequently, even though dPtr is of type Base, it still has access to D1's virtual table (through __vptr).

- what happens when we try to call dPtr->function1()?

```
int main()
{
    D1 d1;
    Base *dPtr = &d1;
    dPtr->function1();

    return 0;
}
```

- First, the program recognizes that function1() is a virtual function. Second, the program uses dPtr->__vptr to get to D1's virtual table. Third, it looks up which version of function1() to call in D1's virtual table. This has been set to D1::function1(). Therefore, dPtr->function1() resolves to D1::function1()

- "But what if dPtr really pointed to a Base object instead of a D1 object. Would it still call D1::function1()?".

```
int main()
{
    Base b;
    Base *bPtr = &b;
    bPtr->function1();

    return 0;
}

//which function1 will be called?
```

- In this case, when b is created, __vptr points to Base's virtual table, not D1's virtual table. Consequently, bPtr->__vptr will also be pointing to Base's virtual table. Base's virtual table entry for function1() points to Base::function1(). Thus, bPtr->function1() resolves to Base::function1(), which is the most-derived version of function1() that a Base object should be able to call.

- When a class object is created, *__vptr is set to point to the virtual table for that class. For example, when a object of type Base is created, *__vptr is set to point to the virtual table for Base. When objects of type D1 or D2 are constructed, *__vptr is set to point to the virtual table for D1 or D2 respectively.

- Because there are only two virtual functions here, each virtual table will have two entries (one for function1(), and one for function2()). Remember that when these virtual tables are filled out, each entry is filled out with the most-derived function an object of that class type can call.

- The virtual table for Base objects is simple. An object of type Base can only access the members of Base. Base has no access to D1 or D2 functions. Consequently, the entry for function1 points to Base::function1(), and the entry for function2 points to Base::function2().

- An object of type D1 can access members of both D1 and Base. However, D1 has overridden function1(), making D1::function1() more derived than Base::function1(). Consequently, the entry for function1 points to D1::function1(). D1 hasn’t overridden function2(), so the entry for function2 will point to Base::function2().

- The virtual table for D2 is similar to D1, except the entry for function1 points to Base::function1(), and the entry for function2 points to D2::function2().

What is Binding?

- Binding is a kind of mapping of a function call with the function's definition i.e. function's address. For example,
  When we make a function call like,

Obj.display();

- then before its execution, it gets bonded to display() function definition i.e. function's address, so that while code execution, correct function should be called.

Static Binding Vs Dynamic Binding

- **Static or Compile time or Early Binding:**

By Default C++ Compiler do the early binding for all function calls i.e. while linking when compiler sees a function call, then it binds that call with the particular function's address / definition.

- **Dynamic or Run Time or Late Binding:**

When we make a member function virtual then compiler performs run time binding for that function i.e. any call to that virtual function will not be linked to any function's address during compile time. Actual function's address to this call will be calculated at run time. To resolve the actual function's address or definition at run time, C++ compiler adds some additional data structure around virtual functions i.e.

- vTable
- vPointers

Overloading Vs Overriding

- Overloading
  - Same scope but different signatures
  - Early or static binding

- Overriding
  - Same signatures but different scope
  - Late or dynamic binding

References

- C++ How to Program By Deitel & Deitel
- The C++ Programming Language By Bjarne Stroustrup
- Object oriented programming using C++ by Tasleem Mustafa, Imran Saeed, Tariq Mehmood, Ahsan Raza
- https://www.tutorialspoint.com/cplusplus
- https://www.learncpp.com/cpp-tutorial/125-the-virtual-table/
- https://thispointer.com/how-virtual-functions-works-internally-using-vtable-and-vpointer/