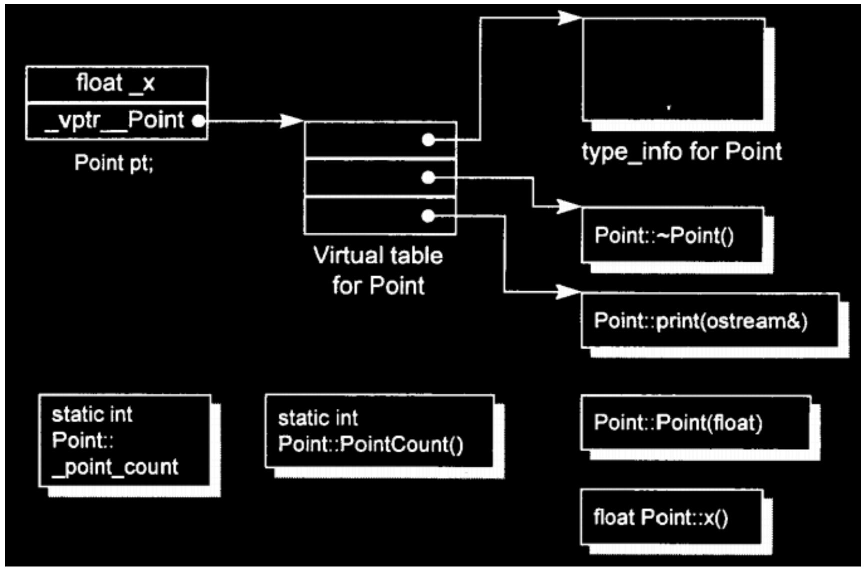
Part 1: All About Virtual Keyword in C++: How Does Virtual Function Works Internally?



Email Print WhatsApp LinkedIn Reddit Twitter Facebook Messenger

Reading Time: 7 minutes

"All About Virtual Keyword in C++" is a series of articles(total of three, PART 1, PART 2, PART 3) describe working of the virtual keyword in different scenarios. This article mostly focuses on "How Does Virtual Function Works Internally?". In other words, How dynamic dispatch done in C++! Although I am not a compiler writer, but this is what I have learned so far from various sources, courses, books & disassembly of C++ program.

RECENT ARTICLES

C++20 Coroutine: Under The Hood

Coroutine in C Language

Mastering C++: Books | Courses | Tools | Tutorials | Blogs | Communities

Regex C++

Using std::map Wisely With Modern C++

Before diving into the How virtual function works internally!, I would like to clarify two things

1. Implementation of dynamic dispatch(i.e. virtual function) is purely compiler dependent.
2. C++ standard does not define the implementation. It only states the behaviour.

I always ask my self before learning anything new "Why do we need it in the first place?". So, let's start there:

Contents [hide]

Why Do We Need a Virtual Function?

- Let's understand it with an example. Suppose you want to connect to the network or to other mobile using your smartphone.
- So, you have two choices Bluetooth or Wifi. Although these two are completely different technologies, still some things are common in them at an abstract/behavioural level like both are communication protocol, both need authentication, etc.
- For example, we have a class of them as follows:

```
class wifi_t{
    private:
        char _pass[15];
        // storage ...
    public:
        void authenticate();
        void connect();
        // operations ...
};

class bluetooth_t{
    private:
        char _pass[15];
        // storage ...
    public:
        void authenticate();
        void connect();
        // operations ...
};
```

- Now, below is the main application in which you want to connect your device to others.

```
int main()
{
    wifi_t          *wifi = new wifi_t;
    bluetooth_t     *bluetooth = new bluetooth_t;

    int pt = selectProtocol();

    if(pt == BLUETOOTH){
        bluetooth->authenticate();
        bluetooth->connect();
    }
    else if(pt == WIFI){
        wifi->authenticate();
        wifi->connect();
    }
    return 0;
}
```

- If you observe above code then you will find that despite selecting any protocol some steps are the same.
- In such case, you can leverage virtual functions of C++ as follows:

```
class protocol_t {
    private:
        uint8_t _type;
        // storage ...
    public:
        virtual void authenticate(){};
        virtual void connect(){};
        // operations ...
};

class wifi_t : public protocol_t {
    private:
        char _pass[15];
        // storage ...
    public:
        void authenticate(){};
        void connect(){};
        // operations ...
};

```

```
class bluetooth_t : public protocol_t {
    private:
        char _pass[15];
        // storage ...
    public:
        void authenticate(){};
        void connect(){};
        // operations ...
};

void makeConnection(protocol_t *protocol) {
    protocol->authenticate();
    protocol->connect();
}

int main() {
    int pt = selectProtocol();

    // You can not compile this line, but i have kept it that way for simplicity
    makeConnection( (pt == WIFI) ? new wifi_t : new bluetooth_t);

    return 0;
}
```

Following are the benefits we have achieved through virtual keywords:

1. **Run time polymorphism**: Behavioural functions identified automatically at runtime & would called by their type like if `protocol` is wifi then execute `wifi_t::authenticate()` & `wifi_t::connect()`.
2. **Reusability of code**: Observe `makeConnection` function there is an only single call to behavioural functions we have removed the redundant code from main.
3. **Code would be compact**: Observe earlier `main` function & newer one.

How Does Virtual Function Works Internally?

- When you declare any function virtual, the compiler will transform(augment is the precise word here) some of your code at compile time.
- For instance, in our case class `protocol_t` the class object will be augmented by a pointer called `_vptr` which points to the virtual table.
- In other words, this is nothing but a pointer(`_vptr`) which points to an array of a function pointer. That includes offset/address of your virtual functions. So that it can call your function through that table rather than calling it directly.

So if you call the function `authenticate()` using a pointer of type `protocol_t` as below:

```
protocol_t *protocol;
// .... assignment to `protocol`

protocol->authenticate();
```

then it would probably augmented by a compiler like this

```
(*protocol->vptr[ 1 ])( protocol );
```

Where the following holds:

1. `_vptr` represents the internally generated virtual table pointer inserted within each object whose class declares or inherits one or more virtual functions. In practice, its name mangled. There may be multiple _vptrs within a complex class derivation.
2. 1 in _vptr[1] is the index into the virtual table slot associated with `authenticate()`. This index is decided by compiler & fixed throughout the inheritance tree.
3. `protocol` in its second occurrence(i.e. in the argument) represents the `this` pointer.

When we inherit `wifi_t` class from `protocol_t` class, a new virtual table will be created by the compiler with overridden polymorphic function slot. Each virtual function has a fixed index in the virtual table, no matter how long the inheritance hierarchy is.

If `derived` class introduce a new virtual function not present in the base class, the virtual table will be grown by a slot and the address of the function is placed within that slot.

If you want to summarize virtual keyword functionality in two words then its `indirect calling` of a polymorphic function. And to visualize virtual function footprint you can take a look at my earlier article memory layout of a C++ object.

Let’s Address Some of the FAQs Around Virtual Function & Virtual Table

Q. Is virtual table per object or per class?
– This usually depends on compiler implementation.
– But Generally, a ***virtual table is per class*** and the ***virtual table pointers(`_vptr`) is to object***.
– There might be more than one virtual table pointers too depending upon type of inheritance.

Q. Where & how does virtual table/pointer code augments by the compiler?

– The code necessary to fill/override virtual table slot generated by the compiler at the time of compilation. To reiterate it in short, the ***virtual table is generated statically at the compile time*** by the compiler.
in constructors right before user-written code.
– Virtual table pointer(i.e. `_vptr`) has fixed offset. And the ***code to override `_vptr` is generated at the time of object construction*** by the compiler.
– This is the reason that you should not call the virtual function in constructor. Read more about it here.
Q. How do we know at runtime that pointer `protocol` will execute a right function(of the object pointed to)?
– In general, we don't know the exact type of the object `protocol` addresses at each invocation of `authenticate()`. However, we do know the virtual table pointer(`_vptr`) offset(which is fixed) associated with the object's class.
– And using this `_vptr`, we can access the virtual table of the object pointed by `protocol` pointer. Again the index of function `authenticate()` in a virtual table fixed throughout the inheritance hierarchy. This way right `authenticate()` function execution guaranteed.

How `Pure` Virtual Function Works?

- When you declare any function as pure virtual, the compiler automatically fills the slot of that pure virtual function with dummy function or so-called place holder `pure_virtual_called()` library instance. And the run-time exception placed if somehow this place holder called.
- In addition, rest of calling & virtual table slot mechanism would be the same as a normal virtual function.

Virtual Function Support Under Multiple Inheritances

- Now with multiple inheritance things will get a little bit tricky. To understand this behaviour let us take another simplified example as follow :

```
struct base1 {
    int base1_var;

    virtual void base1_func() { }
    virtual void print() { }
};

struct base2 {
    int base2_var;

    virtual void base2_func() { }
    virtual void print() { }
};

struct derived : base1, base2 {
    int derived_var;

    void print() { }
};
```

- Here we have derived class with two base classes. In such a case, when we declare an object of the derived class, two virtual table pointers(_vptr) created in the derived class object. One for base1 & other for base2, which are overridden with the address of derived class virtual table.

```
|                        |
|------------------------| <------ derived object memory layout
|  base1::base1_var      |
|------------------------|          |--------->|----------------------|
|  base1::_vptr_base1    |----------|          |   type_info derived  |
|------------------------|                     |----------------------|
|  base2::base2_var      |                     |   base1::base1_func  |
|------------------------|                     |----------------------|
|  base2::_vptr_base2    |----------|          |    derived:::print   |
|------------------------|          |          |----------------------|
|  derived::derived_var  |          |          |------GUARD_AREA------|
|------------------------|          |--------->|----------------------|
|                        |                     |   type_info derived  |
|                        |                     |----------------------|
|                        |                     |   base2::base2_func  |
|                        |                     |----------------------|
                                               |    derived::print    |
                                               |----------------------|
```

- To understand that, first, let's assign a base2 pointer with the address of a derived class object allocated on the heap:

```
base2 *pb = new derived;
```

- The address of the new derived object must be adjusted to address its base2 subobject. The code augmentation done by compiler would look like:

```
base2* pb = static_cast<base2 *>(new derived());

// Equivalent to "address of derived object + sizeof(base1)"
```

- Visualizing memory object of above adjustment.

```
        |                        |
        |------------------------| <------ derived object memory layout
        |  base1::base1_var      |
        |------------------------|          |--------->|----------------------|
                |  base1::_vptr_base1    |----------|          |   type_info derived  |
        pb ---> |------------------------|                     |----------------------|
                |  base2::base2_var       |                     |   base1::base1_func  |
                |------------------------|                     |----------------------|
                |  base2::_vptr_base2    |----------|          |    derived:::print   |
                |------------------------|          |          |----------------------|
                |  derived::derived_var  |          |          |------GUARD_AREA------|
                |------------------------|          |--------->|----------------------|
                |                        |                     |   type_info derived  |
                |                        |                     |----------------------|
                |                        |                     |   base2::base2_func  |
                |                        |                     |----------------------|
                                                               |    derived:::print   |
                                                               |----------------------|
```

- Without this adjustment, any nonpolymorphic use of the pointer would fail, such as

```
pb->base2_var = 5;
```

- And the call to the polymorphic function `print()`

```
pb->print();
```

would probably transforms into

```
( * pb->_vptr_base2[ 2 ])( pb );
```

Summary

I hope you liked this material & helps you to clarify many doubt around virtual function. There is one extraordinary case of virtual destructor which we will see in-depth in PART-3 of this series. Followings are the points which sum up this article in concise way:

1. C++ standard does not define implementation & it only states the behaviour of the dynamic dispatch(i.e. virtual function)
2. virtual table pointers(`_vptr`) is to object
3. virtual table pointer(`_vptr`) has fixed offset
4. code to override `_vptr` is generated at the time of object construction
5. virtual table is per class
6. virtual table is generated statically at the compile time

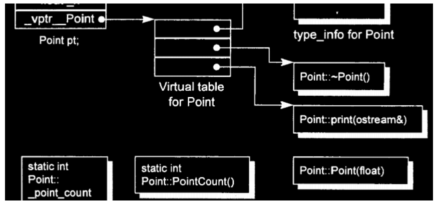
Do you like it ☝️? Get such articles directly into the inbox…!?

Enter your email

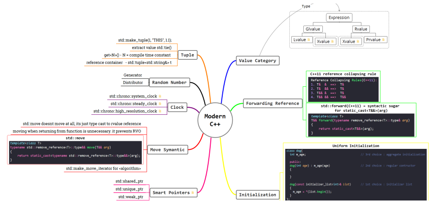
FOLLOW

Email Print WhatsApp LinkedIn Reddit Twitter Facebook Messenger

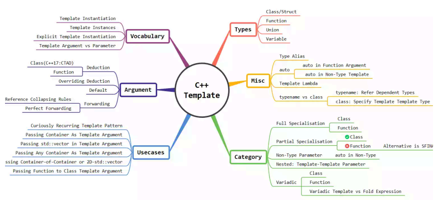
Related Articles



Part 2: All About Virtual Keyword in C++: How Does Virtual Base Class Works Internally?



Understanding unique_ptr with Example in C++11



C++ Template: A Quick UpToDate Look(C++11/14/17/20)