**What are vtable and vptr?**

* **vtable** stands for **virtual table**.
* **vptr** stands for **virtual pointer**.

They are **behind-the-scenes mechanisms** used in **C++** (and similar languages) to implement **runtime polymorphism** — that is, deciding *at runtime* which function to call when you have virtual functions.

**Why do we need them?**

Suppose you have a base class pointer pointing to a derived class object:

class Base {

public:

virtual void show() { cout << "Base show" << endl; }

};

class Derived : public Base {

public:

void show() override { cout << "Derived show" << endl; }

};

Base\* b = new Derived();

b->show(); // Should call Derived::show() at runtime!

* The compiler **can't decide at compile-time** which version of show() to call.
* It needs some **runtime lookup system**.  
  That's where **vptr** and **vtable** come in.

**How does it work internally?**

**1. vtable (Virtual Table)**

* **vtable** is **an array (or table) of function pointers**.
* Each class with **at least one virtual function** gets its **own vtable**.
* The **vtable** contains addresses of the **most derived** version of the virtual functions.

Example:

| **Class** | **vtable** |
| --- | --- |
| Base | Base::show() |
| Derived | Derived::show() |

Each entry in vtable points to the correct function implementation.

**2. vptr (Virtual Pointer)**

* Each object **of a class with virtual functions** has a **hidden pointer** called **vptr** inside it.
* The **vptr** points to the **vtable** corresponding to the object's type.
* When a virtual function is called, the program:
  + Looks up the **vtable** via the **vptr**.
  + Fetches the correct function address.
  + Calls the function.

In our example:

Base\* b = new Derived();

* b's vptr points to Derived's vtable.
* When b->show() is called:
  + It looks into Derived's vtable.
  + Finds Derived::show().
  + Calls Derived::show() at runtime!

**Diagram (Very Important)**

**Memory layout:**

Object 'b' (Base\*)

└── [ vptr ] ---> vtable of Derived

└── [ &Derived::show ]

**Calling process:**

* Access vptr inside the object.
* Follow vptr to vtable.
* Get the function pointer.
* Call the function.

**Key Points**

| **Aspect** | **Details** |
| --- | --- |
| Who creates vtable? | Compiler |
| Who maintains vptr? | Compiler inserts code in constructor to set the correct vptr. |
| Does every object have vptr? | Only objects of classes with virtual functions. |
| Size impact | Size of object increases by size of a pointer (vptr). |
| Inheritance | Derived classes modify vtable if they override virtual functions. |

**Practical Example**

class Animal {

public:

virtual void speak() { cout << "Animal speaks" << endl; }

};

class Dog : public Animal {

public:

void speak() override { cout << "Dog barks" << endl; }

};

int main() {

Animal\* a = new Dog();

a->speak(); // Which speak() will be called?

}

* a's vptr points to Dog's vtable.
* Dog's vtable has pointer to Dog::speak.
* So Dog::speak() is called even though a is of type Animal\*.

**Bonus: When does vptr get set?**

* **In the constructor** of the class.
* Every class constructor assigns the vptr to the correct vtable.
* If you slice an object (copy base part only), vptr can cause strange behavior.

**Simple Pseudocode for How Virtual Call Works:**

function virtual\_call(object, function\_index):

vtable = object->vptr

function\_ptr = vtable[function\_index]

call function\_ptr

**Note:**

* function\_index is determined by the order of virtual functions in the class.

**Summary**

| **Term** | **What it is** | **Where it lives** | **What it points to** |
| --- | --- | --- | --- |
| vptr | Hidden pointer | Inside each object | vtable |
| vtable | Table of function pointers | Per class | Correct function addresses |

**How vtable looks in memory (with an example dump)**, or **simulate the compiler behavior manually**?

**🧠 1. Simulating Compiler Behavior**

Let's manually build what the compiler would generate.

**Suppose you have:**

#include <iostream>

using namespace std;

class Base {

public:

virtual void show() { cout << "Base::show" << endl; }

virtual void print() { cout << "Base::print" << endl; }

};

class Derived : public Base {

public:

void show() override { cout << "Derived::show" << endl; }

};

👉 Base has two virtual functions: show() and print()  
👉 Derived overrides only show().

Now, **how would the compiler internally set up vtables?**

**Base vtable:**

| **Index** | **Function Pointer** |
| --- | --- |
| 0 | &Base::show |
| 1 | &Base::print |

**Derived vtable:**

| **Index** | **Function Pointer** |
| --- | --- |
| 0 | &Derived::show |
| 1 | &Base::print |

**Memory Layout (Manually)**

Base object:

[ vptr ] ---> Base vtable

├── &Base::show

└── &Base::print

Derived object:

[ vptr ] ---> Derived vtable

├── &Derived::show

└── &Base::print

**🔥 2. Example Code Simulating vptr + vtable**

Manually writing code similar to what compiler might do:

#include <iostream>

using namespace std;

// function declarations

void Base\_show() { cout << "Base::show" << endl; }

void Base\_print() { cout << "Base::print" << endl; }

void Derived\_show() { cout << "Derived::show" << endl; }

// Simulated vtables

void (\*Base\_vtable[])() = { Base\_show, Base\_print };

void (\*Derived\_vtable[])() = { Derived\_show, Base\_print };

// Simulated object structures

struct BaseObject {

void (\*\*vptr)(); // pointer to vtable

};

struct DerivedObject {

void (\*\*vptr)();

};

// Simulated function call

void call\_show(BaseObject\* obj) {

obj->vptr[0](); // Call first function in vtable

}

void call\_print(BaseObject\* obj) {

obj->vptr[1](); // Call second function in vtable

}

int main() {

// Create Base object

BaseObject b;

b.vptr = Base\_vtable;

// Create Derived object

DerivedObject d;

d.vptr = Derived\_vtable;

// Base object calls

cout << "Base object calls:" << endl;

call\_show(&b);

call\_print(&b);

// Derived object calls

cout << "\nDerived object calls:" << endl;

call\_show((BaseObject\*)&d);

call\_print((BaseObject\*)&d);

return 0;

}

**🚀 Output:**

Base object calls:

Base::show

Base::print

Derived object calls:

Derived::show

Base::print

✅ **Notice**:

* Even though d is cast to BaseObject\*, because its vptr points to Derived\_vtable, the correct overridden function Derived::show is called.
* print() was **not overridden**, so it calls Base::print().

**📚 Summary Visually**

Think of each object carrying an **invisible book (vtable)** that tells:  
*"If someone asks me to do* ***show()****, turn to page 1 and call* ***this function****!"*

Each object carries its own **book reference** (vptr) depending on what class type it is at runtime.

**⭐ Final Tips to Remember**

| **Topic** | **Remember** |
| --- | --- |
| vptr | Hidden pointer in every polymorphic object |
| vtable | Array of function pointers per class |
| How call works | object->vptr[index]() |
| Override | vtable entry replaced in derived class |
| No virtual functions | No vptr/vtable created at all! |