In C++, the \*\*virtual table\*\* (often abbreviated as \*\*vtable\*\*) is a mechanism used to support dynamic (or late) binding of functions, enabling polymorphism. When a class has one or more virtual functions, the compiler creates a vtable for that class. This table is essentially an array of pointers, where each pointer points to the virtual function implementations of the class.

### How the Vtable Works:

1. \*\*Vtable Creation:\*\*

- When a class with virtual functions is compiled, the compiler generates a vtable for that class.

- The vtable contains pointers to the virtual functions of the class. If the class overrides a virtual function from a base class, the vtable will store a pointer to the overridden function in the derived class.

2. \*\*Vptr (Virtual Table Pointer):\*\*

- Each object of a class that has virtual functions contains a hidden pointer, known as a \*\*vptr\*\* (virtual table pointer). This pointer points to the vtable of the class to which the object belongs.

- The vptr is automatically set up by the compiler in the constructor of the class.

3. \*\*Function Call:\*\*

- When you call a virtual function through a base class pointer or reference, the program uses the vptr to look up the function address in the vtable and then calls the appropriate function.

- This lookup mechanism allows the correct function to be called based on the actual type of the object, not the type of the pointer or reference.

### Example to Illustrate Vtable:

Consider the following simple class hierarchy:

#include <iostream>

class Base {

public:

virtual void show() {

std::cout << "Base show function called." << std::endl;

}

virtual void display() {

std::cout << "Base display function called." << std::endl;

}

};

class Derived : public Base {

public:

void show() override {

std::cout << "Derived show function called." << std::endl;

}

void display() override {

std::cout << "Derived display function called." << std::endl;

}

};

int main() {

Base\* basePtr;

Derived derivedObj;

basePtr = &derivedObj;

basePtr->show(); // Calls Derived::show

basePtr->display(); // Calls Derived::display

return 0;

}

```

### How the Vtable and Vptr Work in This Example:

1. \*\*Vtable for `Base`:\*\*

- The vtable for `Base` contains pointers to `Base::show` and `Base::display`.

2. \*\*Vtable for `Derived`:\*\*

- The vtable for `Derived` contains pointers to `Derived::show` and `Derived::display`.

- These entries in the vtable replace the pointers to `Base::show` and `Base::display`.

3. \*\*Vptr in `derivedObj`:\*\*

- `derivedObj` is an object of type `Derived`. Therefore, its vptr points to the vtable of `Derived`.

4. \*\*Function Call:\*\*

- When `basePtr->show()` is called, the program follows the vptr in `derivedObj` to the `Derived` vtable, finds the pointer to `Derived::show`, and calls it.

- The same happens for `basePtr->display()`; the call is routed to `Derived::display` via the vtable.

### Key Points:

- \*\*Dynamic Binding:\*\* The use of the vtable allows C++ to implement dynamic binding, where the method that gets called is determined at runtime based on the actual object type.

- \*\*Memory Overhead:\*\* There is some memory overhead associated with vtables. Each class with virtual functions needs a vtable, and each object of such a class requires a vptr.

- \*\*Efficiency:\*\* While vtable lookups are fast, they do introduce a small overhead compared to non-virtual function calls, which are resolved at compile-time (static binding).

### Vtable Structure:

Imagine the following simplified structure of vtables:

- For `Base` class:

```

vtable for Base:

+----------------+

| &Base::show |

+----------------+

| &Base::display |

+----------------+

```

- For `Derived` class:

```

vtable for Derived:

+------------------+

| &Derived::show |

+------------------+

| &Derived::display|

+------------------+

```

The vptr in `derivedObj` would point to the `Derived` vtable. When `basePtr->show()` is called, it uses the vptr to access the vtable and call `Derived::show`.

### Summary:

The vtable is an internal structure used by the C++ compiler to implement virtual functions. It enables the correct function to be called for an object, even when using a pointer or reference to the base class, facilitating polymorphism and dynamic binding in C++.