[CS 225] Advanced C/C++

Lecture 4: Virtual method tables

Sławomir "Swavek" Włodkowski
(Summer 2020)

Agenda

- Generation of virtual method tables
- How a compiler translates function calls?
- A case for virtual destructors
- Can we finally see these virtual tables?!
  (*a.k.a.* a basic tutorial of GDB)

Generation of virtual method tables

Virtual method tables (*virtual function tables*, *vtables*) are:

- Static arrays of addresses and address offsets,
- Created during compilation (may be stored in *rodata* segment),
- Created for each class that contains anything `virtual` itself or inherits from such a class.

Generation of virtual method tables

Thanks to *vtables* we can enjoy:

- Dynamic polymorphism of virtual function calls
  We do not lose the information about the object's real type,
  even when we assign its address to a base type pointer.

- Keeping track of virtual base class data members
  We can find virtual base class' members even if they are not
  at the beginning of an object; a vtable stores an offset to the first member.

- Dynamic casting.
  We can obtain a pointer to a derived class object from a pointer to any of its base classes because their vtables store offsets to the top of the object.

Generation of virtual method tables

A compiler creates a *vtable* in the following steps:

1. **Create**: If a class does not inherit a virtual table, create a new table. Otherwise, copy an existing virtual table from a base class.
2. **Extend**: For every virtual function that does not exist in a base class, add an entry with the address of the new function.
3. **Override**: For every function that overrides a function from a base class, updated an entry with the address of the overriding function.

vtables

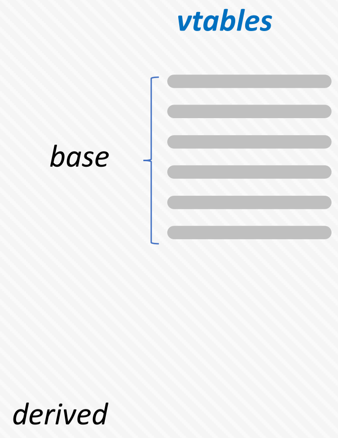
base

derived

Generation of virtual method tables

A compiler creates a *vtable* in the following steps:

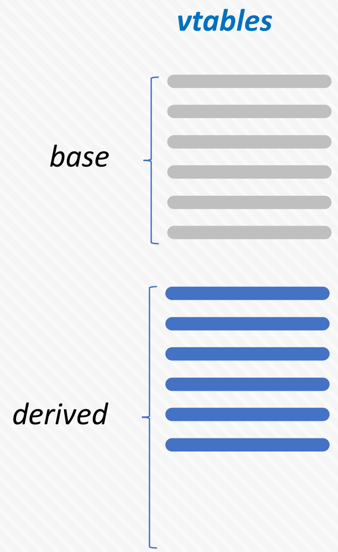
1. **Create**: If a class does not inherit a virtual table, create a new table. Otherwise, copy an existing virtual table from a base class.
2. **Extend**: For every virtual function that does not exist in a base class, add an entry with the address of the new function.
3. **Override**: For every function that overrides a function from a base class, updated an entry with the address of the overriding function.



Generation of virtual method tables

A compiler creates a *vtable* in the following steps:

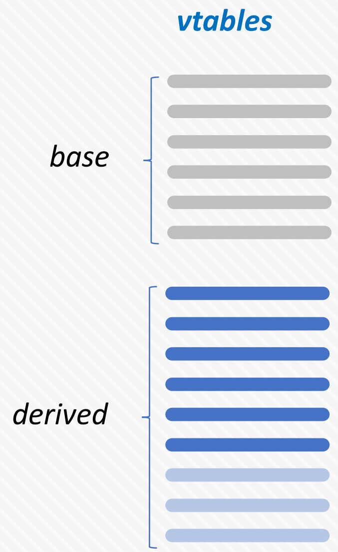
1. **Create**: If a class does not inherit a virtual table, create a new table. Otherwise, copy an existing virtual table from a base class.
2. **Extend**: For every virtual function that does not exist in a base class, add an entry with the address of the new function.
3. **Override**: For every function that overrides a function from a base class, updated an entry with the address of the overriding function.



Generation of virtual method tables

A compiler creates a *vtable* in the following steps:

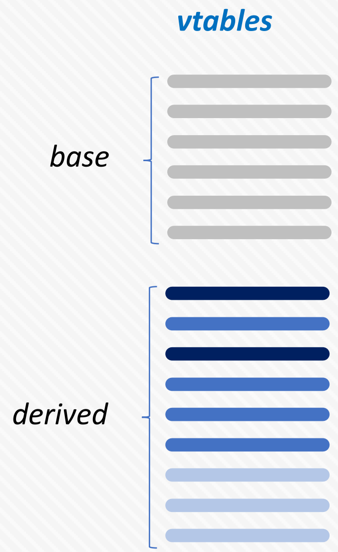
1. **Create**: If a class does not inherit a virtual table, create a new table. Otherwise, copy an existing virtual table from a base class.
2. **Extend**: For every virtual function that does not exist in a base class, add an entry with the address of the new function.
3. **Override**: For every function that overrides a function from a base class, updated an entry with the address of the overriding function.



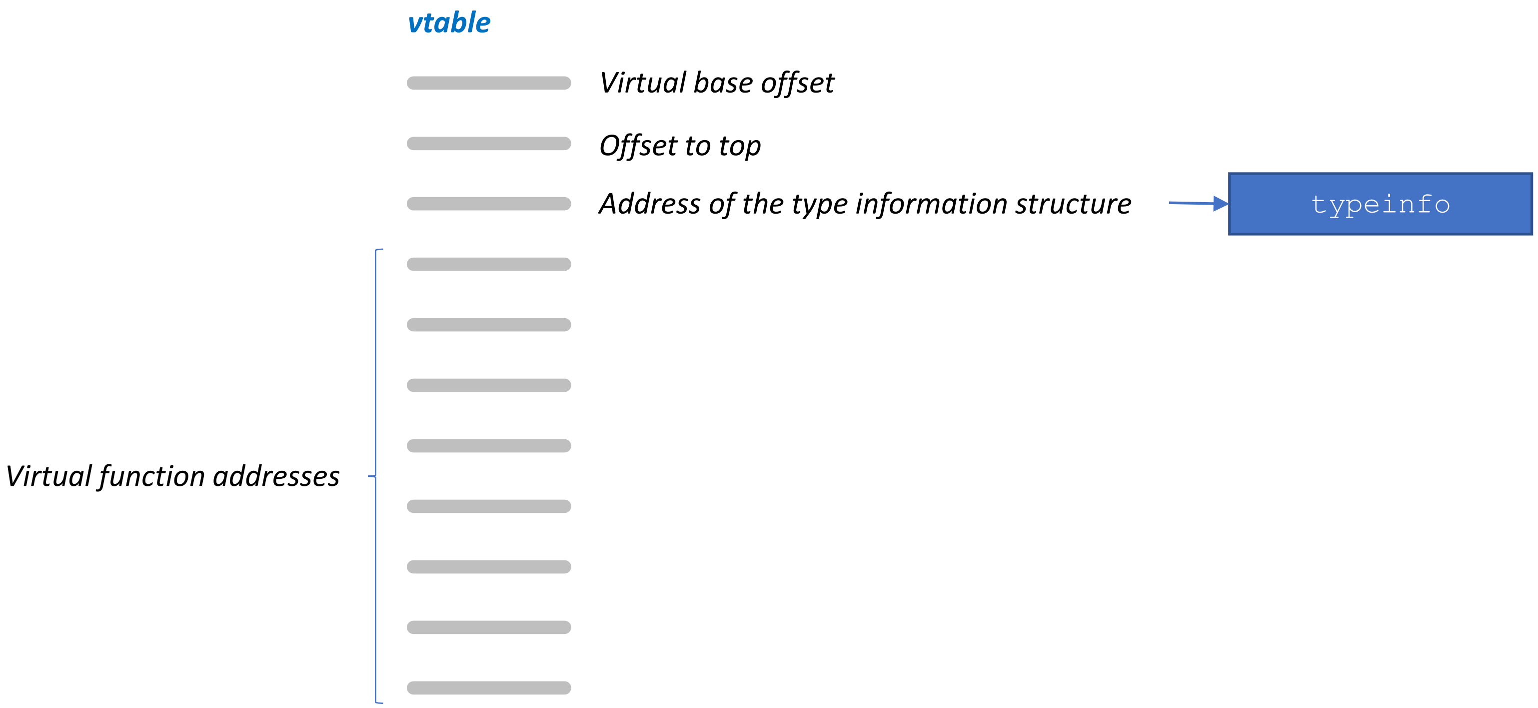
Generation of virtual method tables

A compiler creates a *vtable* in the following steps:

1. **Create**: If a class does not inherit a virtual table, create a new table. Otherwise, copy an existing virtual table from a base class.
2. **Extend**: For every virtual function that does not exist in a base class, add an entry with the address of the new function.
3. **Override**: For every function that overrides a function from a base class, updated an entry with the address of the overriding function.



Generation of virtual method tables



How a compiler translates function calls?

Here we will see various types of function calls:

How their C++ code looks like…

```
Code snippet
```

…and their hypothetical interpretation by a compiler:

```
Generated pseudo-code
```

How a compiler translates function calls?

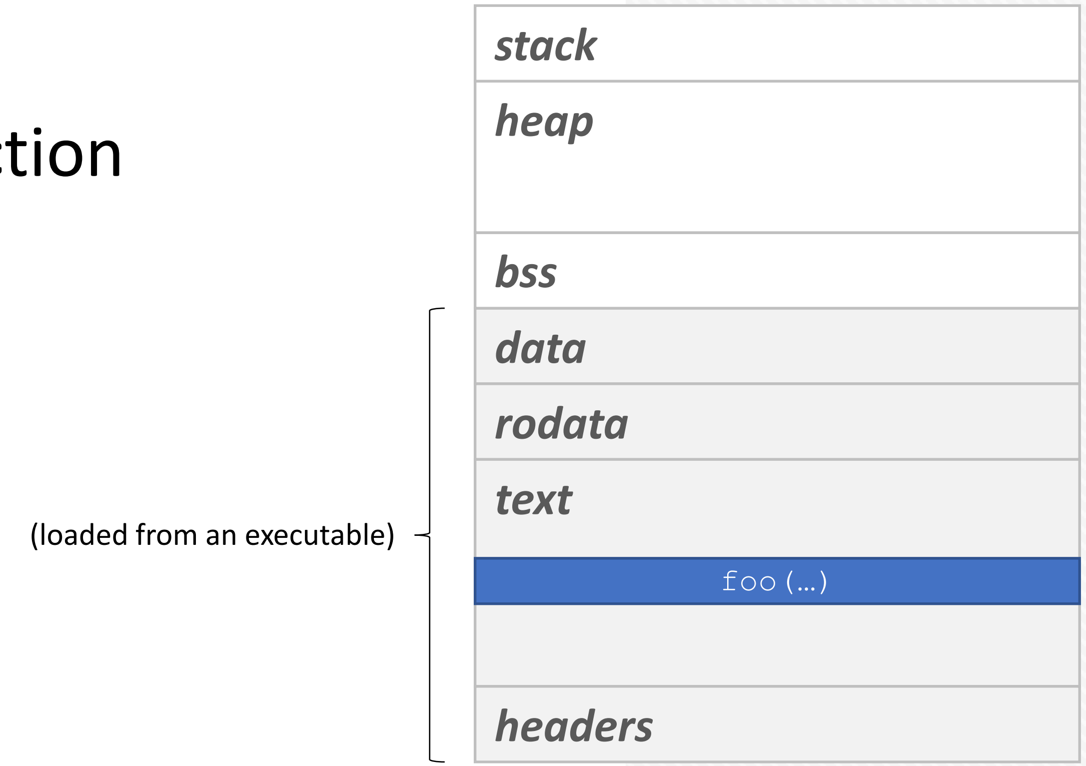
Global functions – the ***address*** of a function is known after compilation in every call.

The following call:

```
foo();
```

Can be translated as:

```
(&foo)();
```



How a compiler translates function calls?

Static member functions – the ***address*** of a function is known after compilation in every call.

The following calls:

```
obj.foo();
ObjClass::foo();
```

Can be translated as:

```
(&ObjClass::foo)();
```

How a compiler translates function calls?

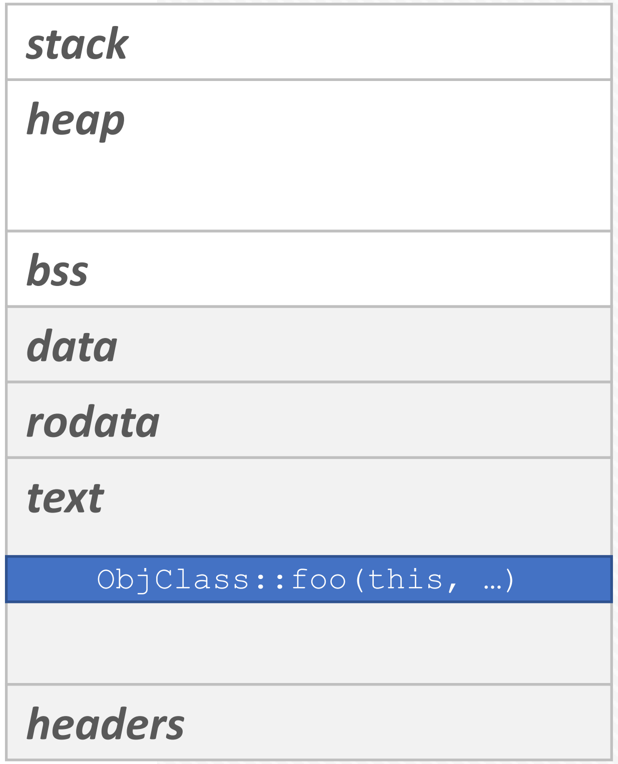
Non-virtual, non-static member functions – the ***address*** of a function is known after compilation in every call.

The following call:

```
obj.foo();
```

Can be translated as:

```
(&ObjClass::foo)(&obj);
```



How a compiler translates function calls?

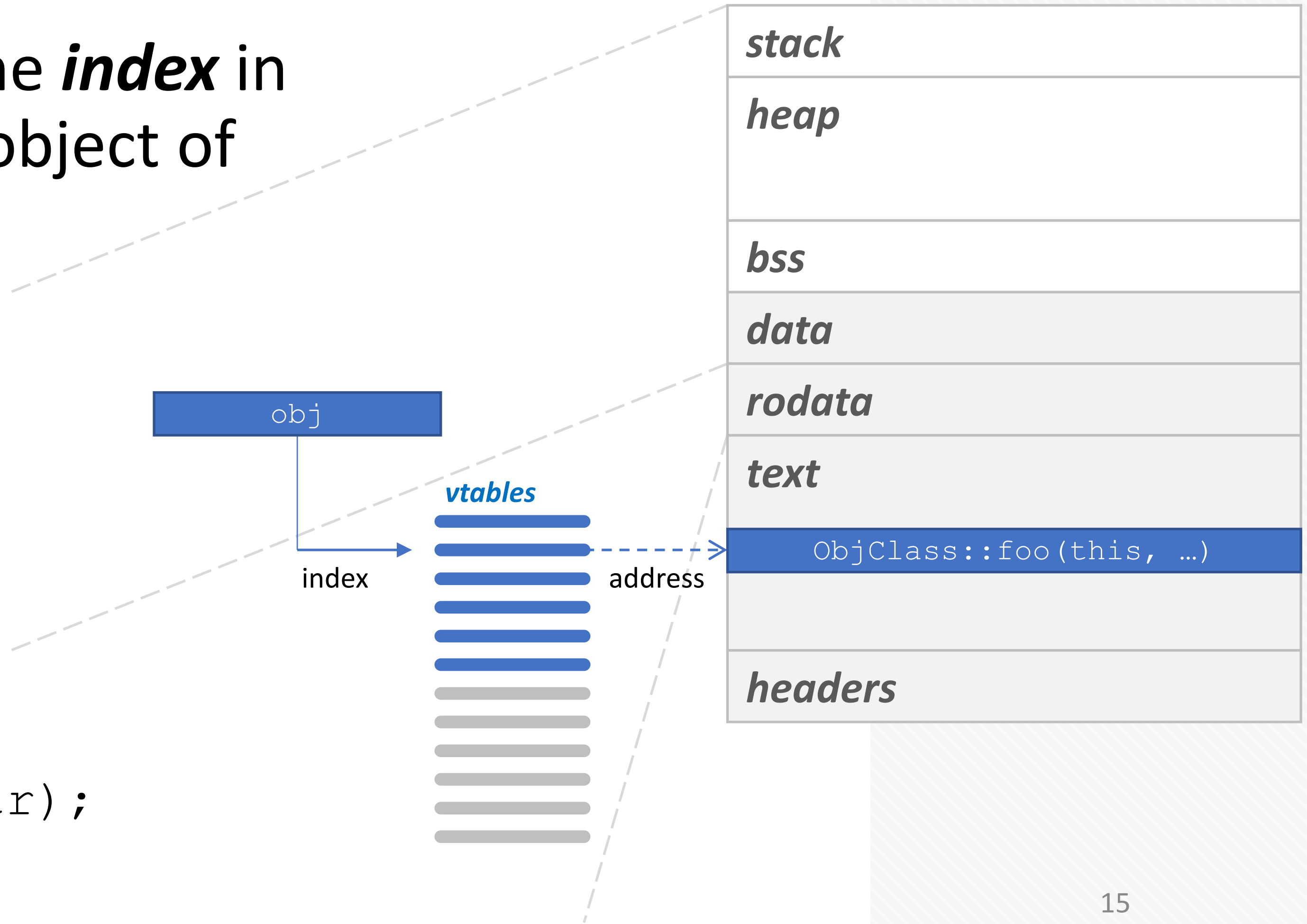
Virtual member functions – the ***index*** in a virtual table is known; each object of this type starts with *vptr*.

The following call:

```
obj_ptr->foo();
```

Can be translated as:

```
(obj_ptr->vptr[index])(obj_ptr);
```



How a compiler translates function calls?

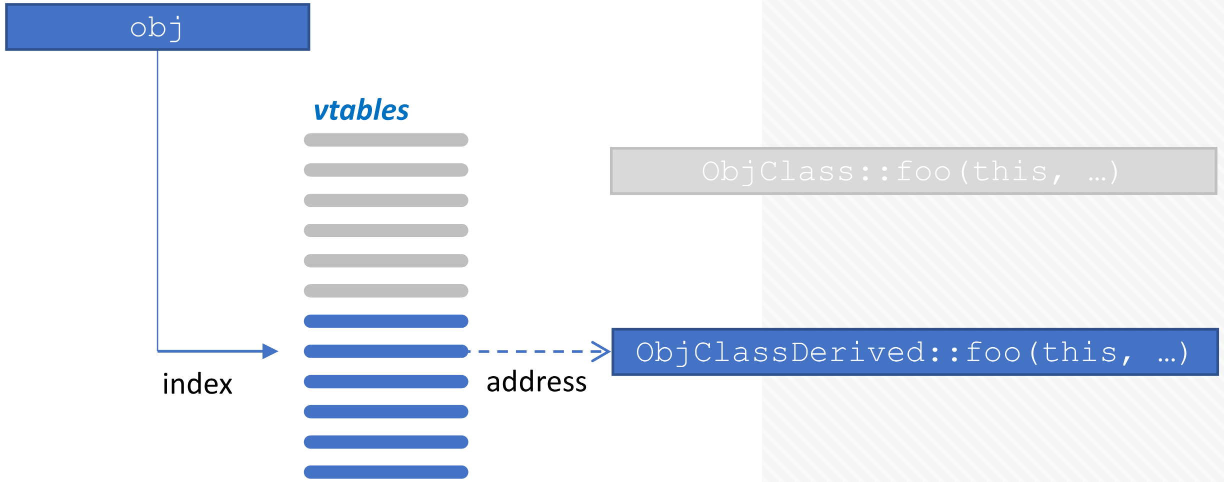
Override member functions – the ***index*** in a virtual table is the same as of the function in a base class; derived class inherits *vptr* from a base, but points to a different table.

The following call:

```
obj_ptr->foo();
```

Can be translated as:

```
(obj_ptr->vptr[index])(obj_ptr);
```



How a compiler translates function calls?

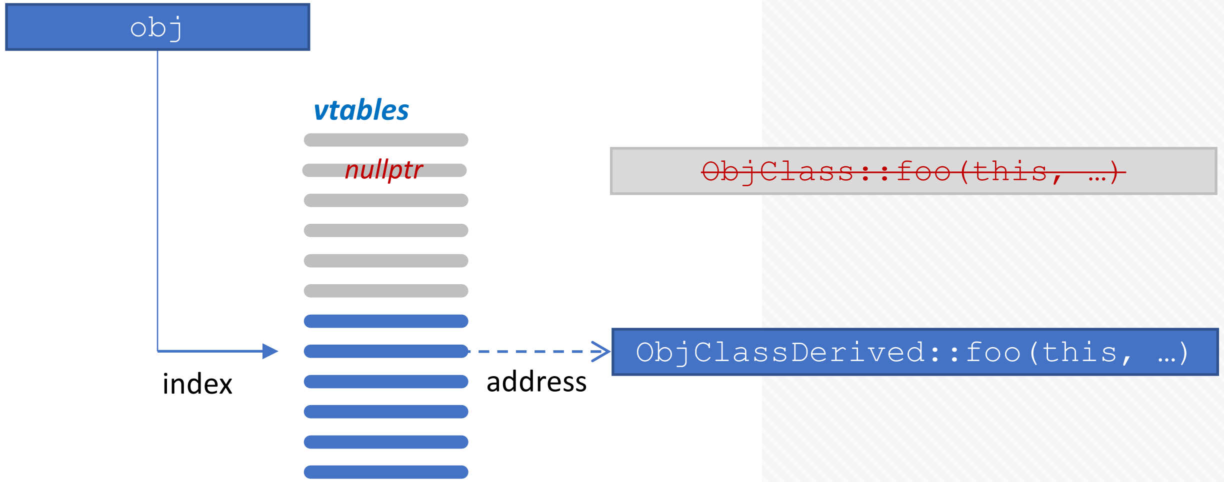
Pure virtual functions without definition – the *index* in a virtual table is known; derived types start with *vptr*. Virtual tables may be generated with `nullptr`, but not stored.

The following call:

```
obj_ptr->foo();
```

Can be translated as:

```
(obj_ptr->vptr[index])(obj_ptr);
```

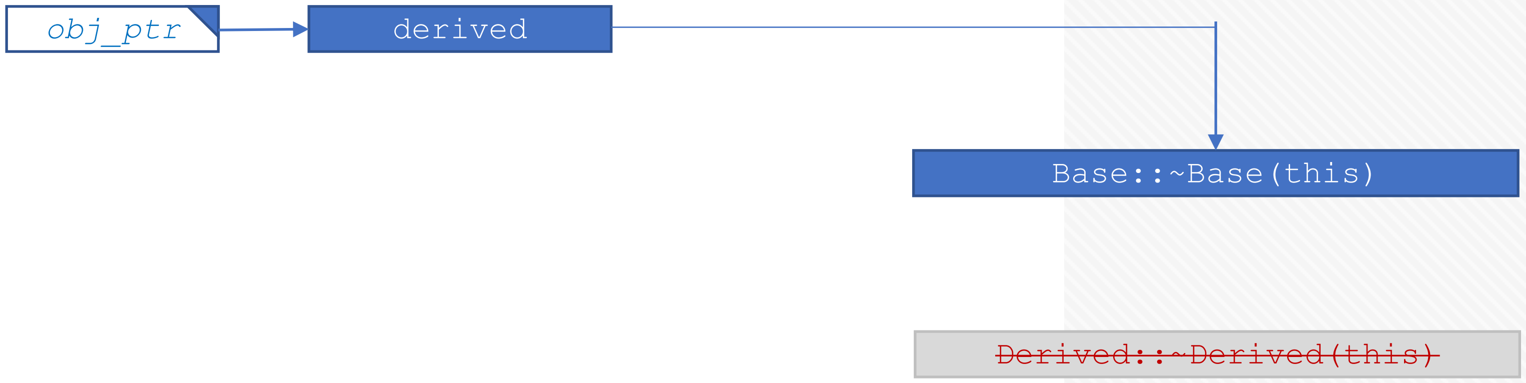


A case for virtual destructors

Any base class used for destroying derived objects **must** have a virtual destructor. If it does not… we will invoke only a base class destructor (non-virtual, non-static).

The following call:

```
delete obj_ptr;
```



Can be translated as:

```
Base::~Base(obj_ptr); // Incorrect destructor!
```

A case for virtual destructors

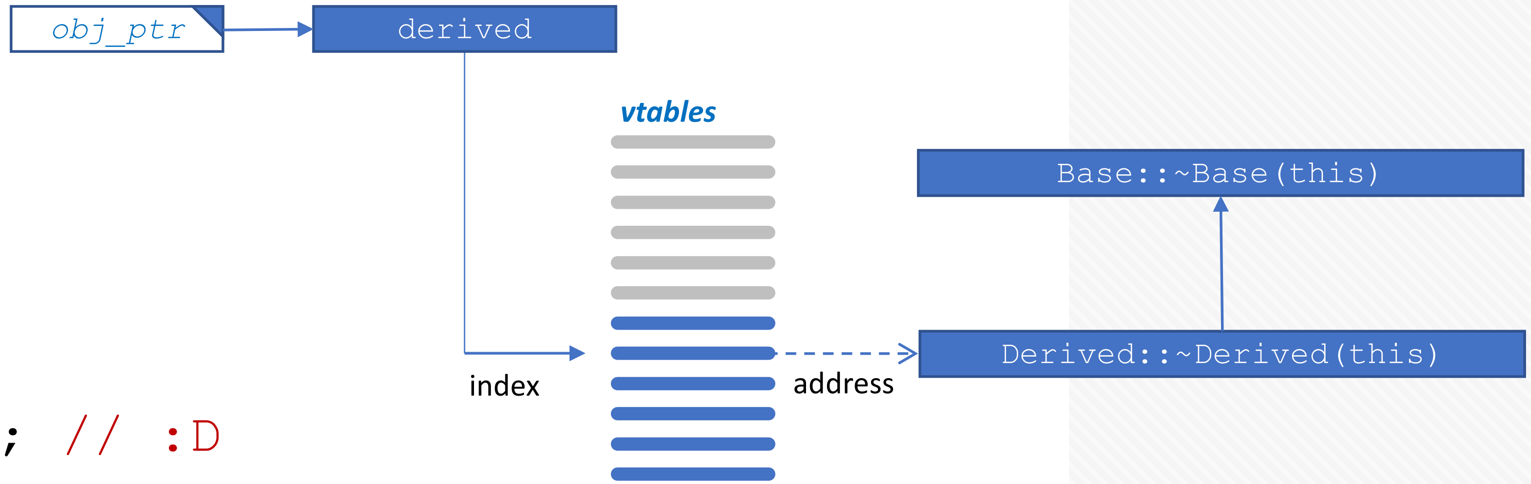
When a destructor is virtual, for an object of a derived type a derived class destructor is called. This destructor automatically calls a base class destructor at the end.

The following call:

```
delete obj_ptr;
```

Can be translated as:

```
(obj->vptr[index])(&obj); // :D
```



Show us the vtables!

GNU Debugger (GDB) cheat sheet:

- Compile a program with g++ with an additional flag (`-g`)
- Launch the debugger (`gdb main.exe`)
- Set printing names from the source, not the binary file (`set print demangle on`)
- Set printing names from the source, not the assembler code (`set print asm-demangle on`)
- Insert a breakpoint at the beginning of the `main` function (`b main`)
- Step into a function (`s`)
- Step to the next instruction (`n`)
- Print members of an object (`p variableName`)
- Print an address of an object (`p &variableName`)
- View the memory dump of `40` bytes at the given address (`x/40xb 0x12345678`)
- From the object read *vptr* and view the memory dump of a *vtable* (take note of the endianness).
- View the assembly of a function at the address range from a *vtable* (`disas /m 0x20000000,0x200000100`)
- From the object read *vptr* and view the memory dump of a *vtable* at the address 24 bytes earlier.