We went over inheritance and ways of implementing it in lecture about a week ago. Today, we’ll see how this memory layout plays out in C++.

Note: For this section, when we ask you to draw the layout of the classes, you don’t need to draw the vtable itself (if there is one). Simply state where a pointer to the vtable is stored.

**Ex.1** In C++, only *virtual* functions necessitate the virtual method table.

```cpp
class Base {
public:
    int x;
    int f() { return 1; }
};

class Derived : public Base {
public:
    int f() { return 2; }
};
```

Draw the layout of the above classes. Note that the methods are not virtual, so there is no vtable!

What does the following code print?

```cpp
Derived a;
std::cout << a.f() << std::endl;
std::cout << ((Base&) a).f() << std::endl;
```

**Ex.2** Suppose the `f` methods above were both marked virtual. Draw the layout of the classes now.
If the methods are marked virtual, what does the following code print?

```cpp
Derived a;
std::cout << a.f() << std::endl;
std::cout << ((Base&) a).f() << std::endl;
```

**Ex.3** In either of the above cases, what does the following print?

```cpp
Derived a; Base& b = (Base&) a;
std::cout << &a << std::endl; // Suppose this prints 0x10
std::cout << &b << std::endl; // What does this print?
```

**Ex.4** Suppose now we have

```cpp
class Base { virtual void g() { }; };
class Base2 { virtual void f() { }; };
class Derived : public Base, public Base2 { };
```

(a) What does the following print?

```cpp
Derived a;
Base &b = (Base&) a;
Base2 &b2 = (Base2&) a;
std::cout << &a << std::endl; // Assume this prints 0x10.
std::cout << &b << std::endl;
std::cout << &b2 << std::endl;
```

(b) What is the layout of the classes in the previous question?

**Ex.5** Now, we explore the *stub* trick. This trick was described in lecture to fix the multiple inheritance with instance methods issue. Suppose we have the following:

```cpp
class Base {
public:
    int foo;
```
virtual int f() {
    std::cout << this << std::endl;
    return this->foo;
}
};

class Base2 {  
public:
    int bar;
    virtual int g() {
        std::cout << this << std::endl;
        return this->bar;
    }
};

class Derived : public Base, public Base2 {  
public:
    int baz;
    virtual int h() {
        std::cout << this << std::endl;
        return this->baz;
    }
};

Then, we try the following

Derived a;
da.f(); // Calls Base::f, which expects foo at offset 0.
da.g(); // Calls Base2::g, which expects bar at offset 0 as well!
da.h();

To overcome this issue, compilers will move the this pointer so that each method sees what it expects. For simplicity, they actually create stub methods that will move the this pointer before calling the original method. For example, the compiler may generate

Base::f1() { move_this_pointer(); f() }

(a) Roughly sketch the class layout of the above classes.
(b) What will the following print?

```
Derived a;
std::cout << &a << std::endl; // Suppose this prints 0x10.
a.f();
a.g();
a.h();
```