→ Code like that is highly coupled to the Book class hierarchy and may indicate bad design.
→ Better: use virtual methods, or write a visitor!

Note: dynamic casting only works on classes with at least one virtual method.

Nov 27/16

Dynamic Casting:

→ Also works with references:

```
Text t {...}
Book &b = t;
Text &t2 = dynamic_cast<Text&>b
```

→ if b "points to" a Text, t2 is a ref to the same Text
→ if not ...? (no such thing as a null reference)
— raises exception bad_cast

→ With dynamic casting, we can solve the polymorphic assignment problem.

```
Text &Text::operator=(const Book &other) {        // virtual
    Text &textother = dynamic_cast<Text&>(other);
    if (this == &textother) return *this;
    Book::operator=(other);
    topic = textother.topic;
    return *this
}
```

Throws exception if other is not a text.

How Virtual Methods Work:

```
class Vec {
    int x, y;
public:
    int doSomething();
};
```

```
class Vec2 {
    int x, y;
public:
    virtual int doSomething;
```

What's the difference (in terms of what they are)?

```
Vec v{1,2};
Vec2 w{1,2};
```

} Do they look the same in memory?

```
cout << sizeof(v) << " " << sizeof(w);
        8                   16
```

Not same size!

8 = space for two ints, methods are not stored inside the object.
→ Compiler turns methods into ordinary functions, stores them separately from objects.

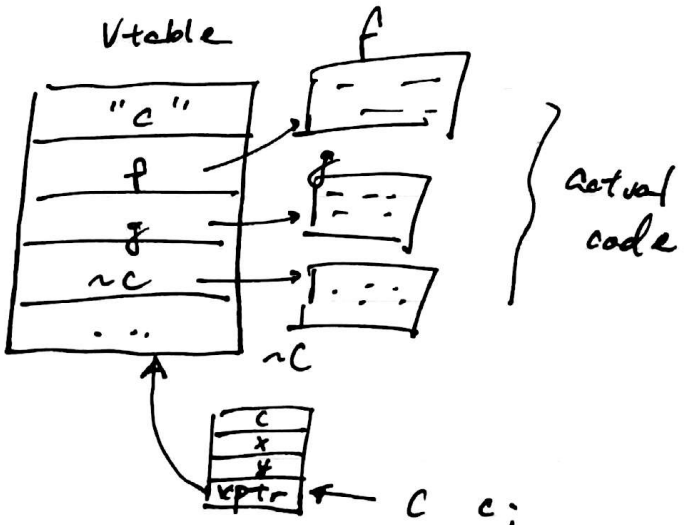
Recall: `Book *pb = new { Book / Text / Comic };`

```
pb -> IsItHeavy();
```

→ If IsItHeavy() is virtual, the choice of which version of IsItHeavy to run is based on the type of the actual object — which the compiler can't know in advance.

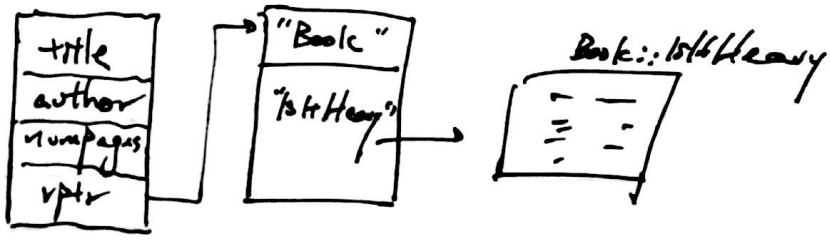
→ For each class with virtual methods, the compiler creates a table of function ptrs (the vtable).

E.g.
```
class C {
    int x, y;
    virtual void f();
    virtual void g();
    void h();
    virtual ~C();
};
```

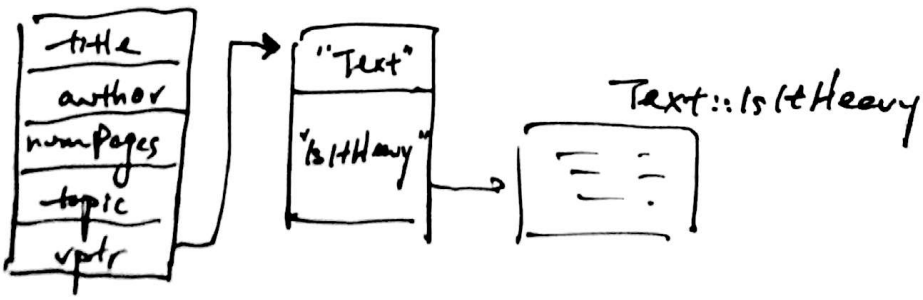


Objects of class C have an extra ptr (the vptr) to its vtable

E.g. `Book b;`



`Text t;`



Calling a virtual method:
- → follow vptr to vtable
- → fetch ptr to actual method from table
- → follow the function pointer to call the function

} at runtime.

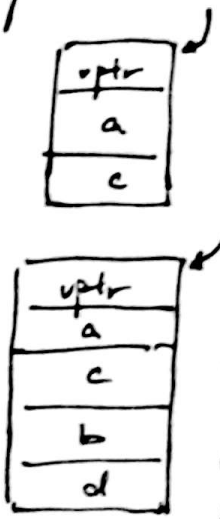
→ Virtual functions incur a small overhead cost in both time and space.

Also: Declaring at least one virtual method adds at least one vptr to the object, so classes with no virtual methods produce smaller objects.

→ Dynamic casting relies on vtable to know what subclass the object is — the identifier label, therefore will only work if there is at least 1 virtual method.

Concretely, how is an object laid out? — compiler dependent.


```
class A {
    int a, c;
    virtual void f();
};
class B : public A {
    int b, d;
};
```



vptr first so that a ptr to B looks like a pointer to A if last 2 fields are ignored.

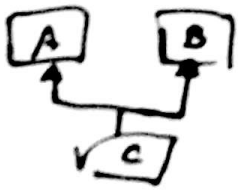
Multiple Inheritance

→ a class can inherit from more than 1 class

```
class A {
  public:
    int a;
};
```

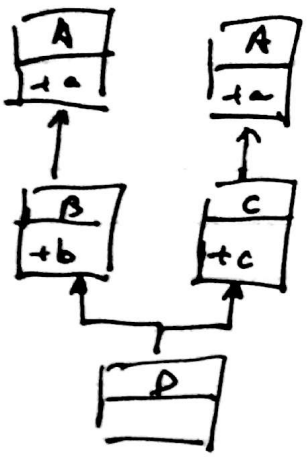
```
class B {
  public:
    int b;
};
```

```
class C : public A, public B {
  void f() {
    cout << a << " " << b;
  }
};
```



→ This is benign!

Challenges: suppose:



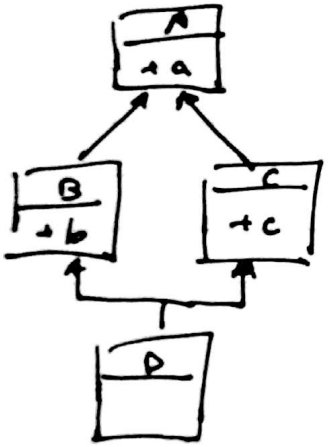

```
class D: public B, public C {
    public:
        int d;
};
```

```
D dobj;
dobj.a;
```

↑ is this B's a field or C's a field?

fix: specify
→ we want `dobj.B::a;`
or `dobj.C::a;`

→ But: if B, C inherit from A, should there be one a part of D, or two a parts of D?
i.e. should B::A and C::A be the same or different?

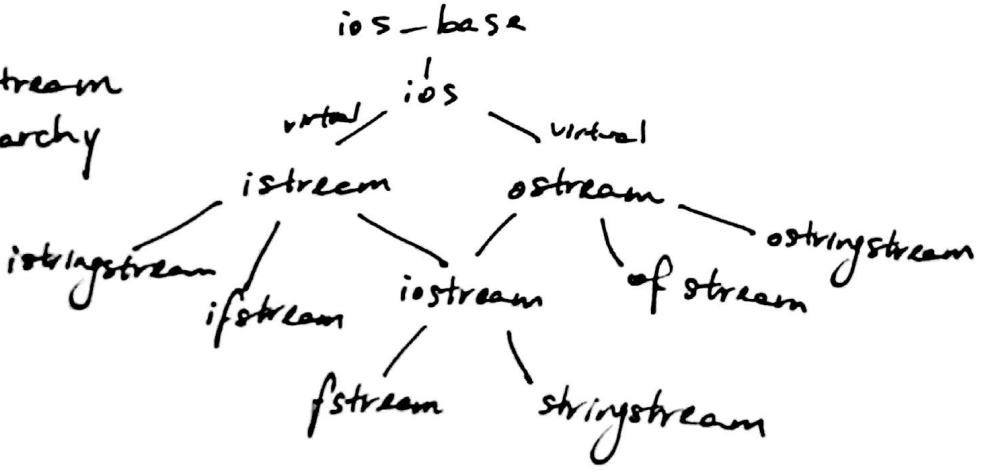


"Deadly Diamond (of Death)"

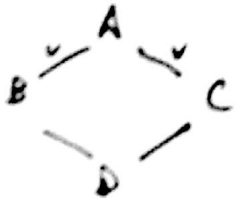
→ make A a virtual base class
i.e. employ virtual inheritance

```
class B: virtual public A {... };
class C: virtual public A {... };
```

e.g. IO stream hierarchy



How will this be laid out?



→ doesn't work! does not look like a C object.

What does g++ do?

→ gives object all three vptrs!

g++:

ptr to B — but not really!
B object should not know about C or D objects/fields!

ptr to C

ptr to A

→ Distance from B to its superclass-part A is not always the same.