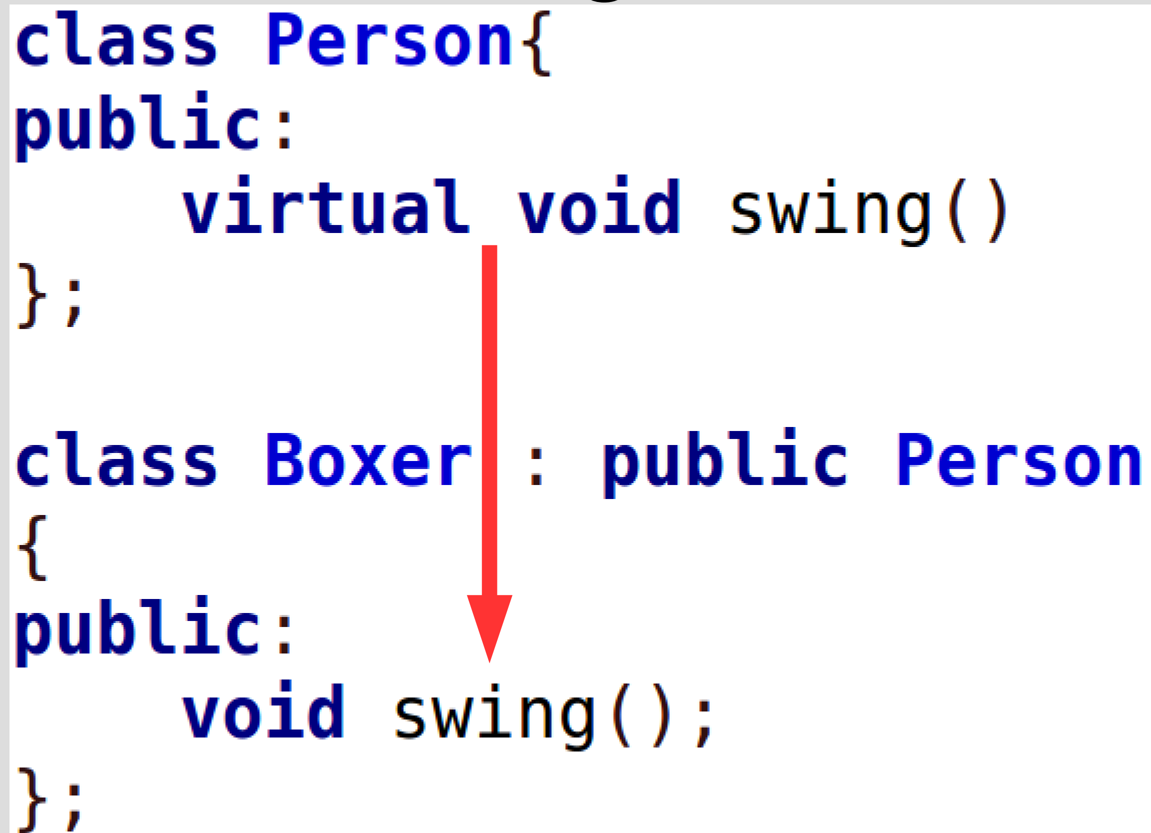
Late binding

Ch 15.3

Highlights

- Late binding for functions



Review: Storing types

Last time we discussed how to properly store a Child object inside a Parent (using pointer)

```
Parent* x = new Child;
```

If we did not use a pointer, it would not work:

```
Parent x = Child;
```

This will only copy the Parent's part of a Child into itself (then delete child)

Early vs late binding

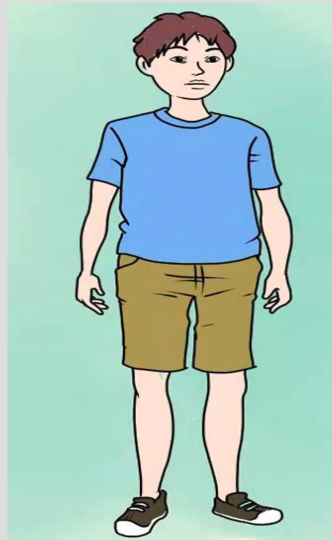
Static binding (or early) is when the computer determines what to do when you hit the compile button

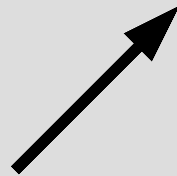
Dynamic binding (late) is when the computer figures out the most appropriate action when it is actually running the program

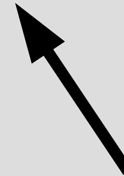
Much of what we have done in the later parts of class is similar to late binding

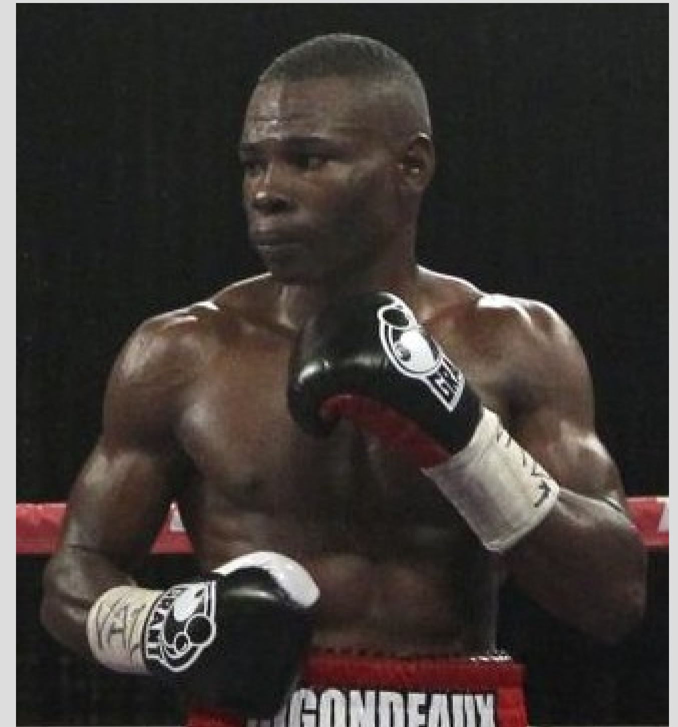
Dynamic binding

Consider this relationship:



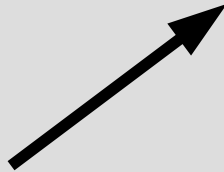
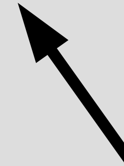






Dynamic binding

Tell each of them to swing()!

Dynamic function binding

Who's swing function is being run?

```
Person p = Person();
Boxer b = Boxer();
p = b;
p.swing();
```

Dynamic function binding

Who's swing function is being run?

```
Person p = Person();
Boxer b = Boxer();
p = b;
p.swing();
```

Answer: the Person's

If you have normal variables, p=b only copies b's Person parts into p's Person box, so you still only have one swing function

Dynamic function binding

Who's swing function is being run now?

```
Person* p = new Person();
Boxer* b = new Boxer();
p = b;
p->swing();
```

Dynamic function binding

Who's swing function is being run now?

```
Person* p = new Person();
Boxer* b = new Boxer();
p = b;
p->swing();
```

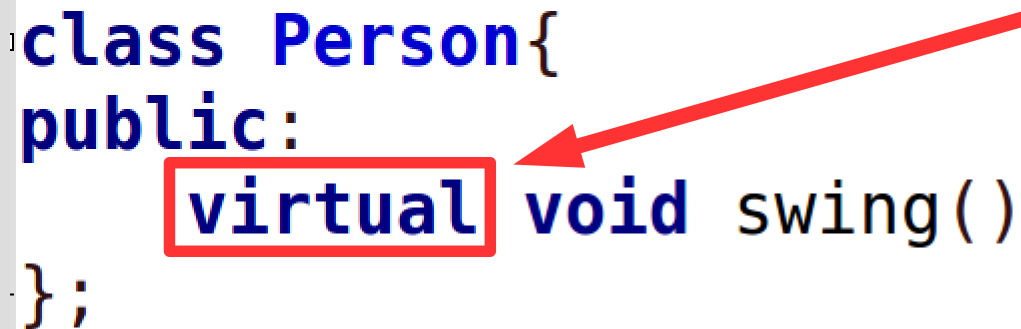
Answer: the Person's still...

p is pointing to a full Boxer object, but it only thinks there is the Person part due to type (see: incorrectChildFunction.cpp)

Dynamic function binding

If we want the computer to not simply look at the "type" of pointer and instead determine what action to take based on the object...

... we need to add virtual (this is slower)



(see: dynamicBindingFunctions.cpp)

Dynamic function binding

If you use a function to run an object and you want to use virtualization, you need to pass-by-reference (i.e. use an &)

If you do not, it will make a copy an this will ignore the Child's part

Can be Person, Boxer or Baseballer

Always a Person

```
void doSwing(Person p)
{
    p.swing();
}
```

```
void doSwing(Person& p)
{
    p.swing();
}
```

Dynamic function binding

If you want to use this virtualization:

1. Pass in a pointer
2. Pass by reference (i.e. use &)

Needs to be memory address so the computer can look at what type is actually there

If you give it a Parent box, it cannot do anything but run normal Parent stuff
(see: dynamicBindingFunctionV2.cpp)

virtual deconstructors

If you use Parent* to dynamically create a instance of a Child class, by default it will ONLY run the parent's deconstructor

With a virtual deconstructor it will run the deconstructor for whatever it is pointing at (the Child's deconstructor in this case)

```
class Parent {
public:
    virtual ~Parent();
};
```

Thus it avoids memory leak
(see: yetAnotherMemoryLeak.cpp)