Late binding
Ch 15.3
Highlights

- Early / Late binding

```cpp
Parent* x = new Child;

class Person{
    public:
        virtual void swing();
};

class Boxer : public Person
{
    public:
        void swing();
};
```
Today we will deal more with inheritance

Mainly we will focus on how you can store a child class in a parent container (sort of)

```java
Parent p = Child();
```

Questions we will answer:
What is this line of code doing exactly?
Are there other ways of doing this?
Early vs late binding

**Static binding (or early)** is when the computer determines what to use 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.
Static binding

When you go to a fast-food-ish restaurant, you get one tray, regardless of what you order.

The key is before they knew what you were ordering, they determined you needed one tray.
Dynamic binding

When you order a drink, they do not just give you a standard cup and say “fill to this line”

Now, they have to react to what you want and give you the correct cup size (not a predetermined action, thus dynamic binding)
Static binding

Checking out at a grocery store, all items are scanned and added to the bill in the same way.

The same program on the computer runs for all items and just identifies their price.
Dynamic binding

After you pay, you put the food into bags (paper/plastic/your own)

What items go where depends on what you want to use and the item properties (weight, dampness, rigidness, etc.)
All animals need to mate, so we could build a generic Animal class with a function `mate()`

However, the gender roles in `mate()` are very different between species...
Static/dynamic binding

static = rigid/constant
dynamic = flexible/adaptive
Static/dynamic binding

Static/dynamic binding is similar to how we originally made arrays: (static/early binding)

```cpp
// need to know the size when compiling
int x[20];
```

To dynamic memory arrays: (dynamic/late)

```cpp
cin >> size;
// may not know how big x is until this line
int* x = new int[size];
```
Mini-quiz (ungraded)

```cpp
class Parent {
public: // bad bad bad bad
    int x;
};

class Child : public Parent {
public: // bad bad bad bad
    int y;
};

int main()
{
    Parent p;
    p.x = 1;

    Child c;
    c.x = 2;
    c.y = 10;
    p = c;

    int z = 2.5;
}
```

What is in `p` at end of `main()`?
1. x=2
2. x=2, y=10
3. x=1, y=10
4. x=1

(Hint: what happens on this:)

int z = 2.5;
It is debatable how we should interpret line:

```
p = c;
```

In C++ (not some other languages), this just copies the parts of the parent class over:
Mini-quiz (ungraded)

What is at p now?
1. x=2
2. x=2, y=10
3. x=1, y=10
4. x=1

```cpp
class Parent {
public: // bad bad bad bad
    int x;
};

class Child : public Parent {
public: // bad bad bad bad
    int y;
};

int main()
{
    Parent* p = new Parent;
    p->x = 1;

    Child* c = new Child();
    c->x = 2;
    c->y = 10;

    p=c;
}
```
= between parent/child pointers

When the objects are pointers, lines line just changes the object being pointed to (but not any information inside either class)

```
p = c;
```
Dynamic variable binding

If a Parent type is pointing to a Child instance, we cannot directly access them (variables cannot be “virtual”...)

\[ p->y = 20; \ // \ red \ angry \ underlines! \]

Instead, we have to tell it to act like a Child* by casting it:

\[ \text{static\_cast<Child*>}(p)->y = 20; \ // \ happy \]

(see: dynamicObjects.cpp)
Dynamic variable binding

If p points to a Parent instance, the below line is VERY BAD (but it might work... sorta...)

```cpp
Parent* p = new Parent;
static_cast<Child*>(p)->y = 10; // happy..?
```

You will be fooling around in some part of memory that is not really associated p (though you might not crash...)

(see: badMemoryManagement.cpp)
(see: memoryOops.cpp)
EVERY MEAL YOU MAKE,
EVERY BITE YOU TAKE,

I’LL BE WATCHING YOU.

memes.com
Dynamic binding

Consider this relationship:
Dynamic binding

Tell each of them to swing()!
Dynamic function binding

Who's swing function is being run?

```java
Person p = Person();
Boxer b = Boxer();
p = b;
p.swing();
```
Dynamic function binding

Who's swing function is being run?

```java
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?

```cpp
Person* p = new Person();
Boxer* b = new Boxer();
p = b;
p->swing();
```
Dynamic function binding

Who's swing function is being run now?

```cpp
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)

class Person{
public:
    virtual void swing();
};

(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 and this will ignore the Child's part

Always a Person

Can be Person, Boxer or Baseballer
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 an 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)

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