Late binding

Dec. 8, Ch 15.3
Announcements

Final exam conflicts with physics

Homework 10 is extra credit (will use GraphX)

Fill out evaluations

Lab this week extends last lab, so if you did not finish all problems (including challenge), look at posted code ON MOODLE
Highlights

- Late binding for functions

```cpp
class CheckOut {
public:
  virtual void bag();
};

class PaperCheckOut : public CheckOut {
public:
  void bag();
};
```

- Late binding for variables

```cpp
CheckOut* co = new PaperCheckOut;
```
Today we will deal more with inheritance.

Mainly we will focus on how functions can be passed on and changed by a child class.

Do we run the parent's version or child's?
Early vs late binding

**Static binding** (or early) is when the computer determines what function to use when you hit the compile button.

**Dynamic binding** (late) is when the computer figures out the most appropriate function 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...

- snack
- caring
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Static/dynamic binding is similar to how we originally made arrays: (static/early binding)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];
```
Dynamic function binding

Suppose we are doing the checkout bagging.

To tell the computer to use dynamic binding, we simply add `virtual` before the function (similar to how we used `friend`).

```cpp
class CheckOut {
public:
    virtual void bag();
};
```

(see: dynamicBindingFunction.cpp)
Dynamic function binding

Why do we need the call-by-reference (the & in runBag())?

If we remove this, it will create a copy that only has CheckOut information.

Thus CheckOut::bag() is run and not using dynamic binding, as the type of other is simply CheckOut and cannot be more...
Dynamic function binding

To use dynamic binding without the silly runBag() function, we need to use pointers (why call-by-reference works)

If a variable of type CheckOut* points to an instance of PaperCheckOut...

Then the use of virtual on bag(), will use the most appropriate version (PaperCheckOut's) (see: dynamicBindingFunctionV2.cpp)
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;
}
```

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:)

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

```c
p = c;
```

In C++ (not some other languages), this just copies the parts of the parent class over the child class:
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
    int x;
};

class Child : public Parent {
public: // 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;
}
```
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

Between `Parent` and `Child` pointers:

- `Parent`:
  - `int x = 1`
  - `Parent* p = go`

- `Child`:
  - `int y = 10`
  - `Child* c = go`

- `Parent* p` and `Child* c` point to the same location.
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: dynamicBindingVariables.cpp)
Dynamic variable binding

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

```cpp
static_cast<Child*>(p)->y = 20; // happy
```

You will be fooling around in some part of memory that is not really associated p

While it might not crash, you will might be changing the value of some other variables (see: badMemoryManagement.cpp)
virtual deconstructors

I purposely did not properly delete dynamic memory in examples, as it gives a warning:

warning: deleting object of polymorphic class type ‘CheckOut’ which has non-virtual destructor might cause undefined behaviour

TLDR:
virtual function+non-virtual deconstructor=bad

But why?
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)

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