CSCI 2270
Data Structures and Algorithms
Lecture 5—Classes part 1

Elizabeth White
elizabeth.white@colorado.edu
Office hours: ECCS 128 or ECCS 112
Wed 9:30am-11:00am
Thurs 10:00am-11:30am

Administrivia

- HW1 posted
  - It will be due Sunday, Feb. 2nd
  - We'll cover function parameters and classes Friday
- Read pp.117-133 (skip recursive part) for hw1
- Skim pp.47-64 for description of general Bag
  - We'll reinforce the important parts in lecture
- No lab this week; lab resumes next week
- Clicker scores have posted for Wednesday
  - Make sure to set the room frequency (AB)
  - Register your clicker

C++ variables to arrays to structs

You've seen how single variables (ints, doubles) work and how arrays of those variables work.

You've also seen generic variables (ItemType) work singly and in arrays.

To describe more complex problems, we can stick a bunch of variables together into something called a struct. For instance, a student in 2270 could be described using her name (a string), ID number (an integer), and course grade (a double).

```
student s;                    s.name = "Zorro";
s.id = 800292663;             s.grade = 0.75;
```

C++ structs to classes

Structs are still not ideal, though. This all looks ok:

```
student s;                    s.name = “Zorro”;
s.id = 800292663;             s.grade = 0.75;
```

But how about those member variables now?

```
s.name = “~%6*!”;
s.id = -5;
s.grade = -900;
```

We can’t just let people set these values to anything…

C++ structs to classes

What we'd like better is for the struct to have member functions that make sure its member variables are set to correct values.

How would you check if a student id is a valid number?

How would you check if the student name is valid?

A struct that includes these member functions is called a class.

C++ structs to classes

Big idea here: By using member functions that control access to the member variables, our code becomes safer!

Corollary: we expect the users of our code to be, well, kind of dumb, and we plan to protect our code from their bad ideas.

C++ classes to objects

Consider for a moment the relationship between a type and an instance. One example of a type is an int (which describes a set of possible integer numbers).

```
int a = 5;       // a is an instance of the int type
int b = -2984989;   // b is another instance of int
```

Any valid integer running in your code is an *instance* of the int type. Many possible instances exist for ints.

C++ classes to objects

Your classes, like ArrayBag, are also types; they're just bigger and more complicated types than integers, booleans, and other 'primitive' types in programming languages. When your classes construct new objects:

```
ArrayBag<int> oscar;
```

those objects you're making are *instances whose type is your class.* oscar's just one of all the possible ArrayBags in the world you could have created here.

Because class instances are more complicated, we like to call them *objects*.

C++ classes

Designing classes is HARD. We'll start out with a simple class, called a bag.

Bag class: stores a collection of things

Unsorted, for now

Finite capacity, for now

Bag knows how to:

create itself, empty itself,

add items, remove items,

list all of its contents, count its contents,

tell if an item is present in its contents,

and tell how many copies of that item exist.

C++ classes, interface part

Notice that nothing in the bag description is specific to C++: this is just a list of things we expect that bags should know how to do. A list of behaviors for a class, at the highest level, is called an *interface*.

Bag interface: a bag can

create itself, empty itself,

add items, remove items,

list all of its contents, count its contents,

tell if an item is present in its contents,

and tell how many copies of that item exist.

C++ classes, interface and implementation

How we decide to program those baggy behaviors is much more specific, because we have to commit to a particular way of writing bags.

You know that a list of behaviors for a class, at the highest level, is called an *interface*. How we program those behaviors specifically is called the class *implementation*.

C++ classes, interface and implementation

What's so important about this distinction?

Interfaces are like contracts you make with the user of your code. You promise, for example, that bags will store items and not forget them.

Implementation is considered the programmer's private business. As long as the user thinks the Bag is working normally, he doesn't care how you wrote the code to get it to do that. (Note that the TAs, LAs, and I care how you wrote the code, too.)

C++ classes, interface and implementation

What's so important about this distinction?

Interface: BagInterface.h lists everything a bag should do for a user. Don't sweat the virtual functions or the weird = 0 statements; just notice that the function names all line up with functions you write in ArrayBag.h and ArrayBag.cxx.

Implementation: ArrayBag.h and ArrayBag.cxx are your implementation files, because they describe exactly how we made the bag do its thing.