# CSCI 2270
# Data Structures and Algorithms
# Lecture 6—Classes part 2

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.31-46 on C++ classes
- Did you get email from me at about 12:40 today?
- Clicker scores have posted for Wednesday
  - Make sure to set the room frequency (AB)
  - Register your clicker

## Function parameters

```
void fun1(int bobo) {  bobo *= 2; }
void fun2(int& bobo) {  bobo *= 2; }
void fun3(const int& bobo) {  bobo *= 2; }

int main()
{
        int homeslice = 6;
        fun1(homeslice);
        cout << homeslice << endl;      // no change
}
```

## Function parameters by value

```
void fun1(int bobo) {  bobo *= 2; }

int main()
{
        int homeslice = 6;
        fun1(homeslice);   // homeslice is copied to fun1
                           // copied homeslice doubles
        cout << homeslice << endl;
                           // original homeslice abides…
}
```

# Function parameters by value

```
void fun1(int bobo) {  bobo *= 2; }
void fun2(int& bobo) {  bobo *= 2; }
void fun3(const int& bobo) {  bobo *= 2; }

int main()
{
        int homeslice = 6;
        fun2(homeslice);
        cout << homeslice << endl;        // changes
}
```

# Function parameters by reference

```
void fun2(int& bobo) {  bobo *= 2; }
int main()
{
        int homeslice = 6;
        fun2(homeslice);    // address of homeslice is
                            //      passed to fun2 and
                            //      value at this address
                            //      doubles
        cout << homeslice << endl;
                            // homeslice is now doubled
}
```

# Function parameters by constant reference

Passing in an address to a variable can be faster than passing in a copy. But passing in an address also risks exposing the variable to changes. If we pass the address in as a constant reference, we get the speed and the protection for our variable:

```
void fun3(const int& bobo) {  bobo *= 2; }
```

The compiler will actually refuse to build this code at all, because it's breaking the promise (that we made with the const int&) to keep bobo the same.

# Function parameters by constant reference

As long as our fun3 code does not change bobo, it will compile and run just fine. Like this:

```
void fun3(const int& bobo) {  cout << bobo << endl; }
```

Our Bag functions make use of this const reference trick.

## Back to classes, via single variables, arrays, and structs

We talked about how data types can be:

single variables,

arrays of variables,

structs (a bunch of different variables, stuck together to make a new data type),

or classes (again with different variables, but now with member functions to change them in proper ways).

## Types and instances

We also talked about the relationship between types and instances. The type of a variable determines the set of values that each variable instance can have. Below, the types are double and bool, and the instances are n1 and done.

```
double n1 = 1/3.0;  // 0.3333… is allowed
bool done = false;  // only 2 values allowed here
```

Both C++ and Java are fairly strict about types, especially when assigning or converting between types

Exception is generic data type (ItemType)

## Classes and objects

When we make a variable whose type is a class, like

```
ArrayBag<string> papas_brand_new_bag;
```

we can still think of the class ArrayBag as being a type, and the variable papas_brand_new_bag as being an instance of the ArrayBag class.

But convention dictates that we usually call the variable an *object* if its type is a class.

So types define instances and classes define objects.
*It's the same relationship, with different names.*

## Bag class constructor

Makes a new empty bag, initializes it

Check ArrayBag.h and notice that the item array is built automatically for this first bag:

```
// defines a size for all ArrayBags (static variable)
static const int DEFAULT_CAPACITY = 25;
// builds an array of items
ItemType items[DEFAULT_CAPACITY];
```

(For hw2, with a resizable bag, we can't rely on this mechanism.)

# Bag class constructor

Makes a new empty bag, initializes it

Check ArrayBag.h and notice that the itemCount is defined as a member variable:

```
int itemCount;
```

In ArrayBag.cxx, we just set

```
itemCount = 0;
```

(Why can't we say

```
int itemCount = 0;
```

?)

# Bag size functions

getCapacity: returns the current capacity of the Bag

getCurrentSize: returns the current number of items in the Bag

isEmpty:

```
if (itemCount == 0) return true;
else return false;
```

Or…

```
return (itemCount == 0);
```