CSCI 2270
Data Structures and Algorithms
Lecture 12—back over more details of C++ classes

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

Administrivia

- First midterm exam is Monday, here, at 2-2:50
  - Open book, open note,
  - No calculator, phone, computer
  - Review sheet posted. (Answers won't be, but help hours will be there.)
  - Bring questions to class Friday.
- I have to shift 0.5 office hour from Thursday to Wednesday permanently (starting today!)
- Interview grading slots will post for HW2 this week
  - I'll send out an email when the slots post
  - Find a slot and please don't miss the appointment. We don't promise you we can make a second date for this.

We defined a class in C++ in 3 parts

We defined your class in C++ in 3 parts:

    BagInterface.h

    ArrayBag.h

    ArrayBag.cxx

And none of these compiled without the test code,

    ArrayBagTester1.cxx

BagInterface class

template<class ItemType>

class BagInterface { … }

Defines one Bag ancestor class, BagInterface, with a set of empty public methods, like

```
virtual bool add(const ItemType& newEntry) = 0;
```

Think of these methods as the core set of functions a user expects from any sort of Bag

In BagInterface.h, none of these methods do anything yet (= 0); virtual functions are just placeholders.

But any class that inherits from the ancestor (BagInterface.h) will need to implement these classes before it can define a fully functional object.

ArrayBag inherits from BagInterface

Remember the BagInterface class name

```
template<class ItemType>
class BagInterface { … }
```

ArrayBag is defined as a descendant of the BagInterface here, after the colon:

```
template<class ItemType>
class ArrayBag : public BagInterface<ItemType>
```

The public keyword here means that anything defined as public in BagInterface is also public in ArrayBag (and anything defined as private in BagInterface would also be private in ArrayBag).

BagInterface virtual methods

With those virtual functions,

If ArrayBag.cxx doesn’t implement them, the compiler complains. (Try commenting out the ArrayBag’s add method, and you’ll see this.)

All the public virtual functions must be implemented (made actual) in the descendant.

If ArrayBag implements a new function that’s not in the interface, then the compiler doesn’t complain. And the new function doesn’t need to go in the interface. (It’s ok to add new functions, just not to skip the interface ones.)

ArrayBag.h and ArrayBag.cxx

In C++, it's conventional to make a split in a class definition, for 2 reasons:

1. It's a (clumsy) way to separate the interface for the function (what it does) from its implementation (how it does that). Clumsy because the class member variables are exposed in the header, as are private functions; too much information for an interface.
2. It allows any other files that use this class to compile with implicit trust that this class and its methods exist (the implementation code is finally included later in a final compilation step called linking).

We defined a class in C++ in 3 parts

How did ArrayBagTester1.cxx know what to do?

1. ArrayBagTester1.cxx has the line:

```
#include "ArrayBag.h"
```

At compile time, this line of text gets replaced with the contents of ArrayBag.h (a copy & paste job).

Now ArrayBagTester1 believes that the ArrayBag class exists.

We defined a class in C++ in 3 parts

How did ArrayBagTester1.cxx know what to do?

2. ArrayBag.h has the line:

```
#include "BagInterface.h"
```

so BagInterface.h pastes in over that line at compile time. Then the ArrayBag.h code specifies the public and private variables and functions that all ArrayBags use, and then the line:

```
#include "ArrayBag.cxx"
```

brings in the implementation code you wrote. Now, we’ve dragged all the code in by compiling the test file.

Macro guards

ArrayBag.h and BagInterface.h contain macro guards, as well.

```
#ifndef _ARRAY_BAG
#define _ARRAY_BAG
            … class goes here …
#endif
```

Why? It’s perfectly likely that 2 files in a project might #include the same header file (and this might well be needed for them to compile in C++).

But the compiler freaks out if it sees a function or class being defined twice! (It doesn’t know which one you mean it to use.)

Macro guard case 1

The first time the compiler sees ArrayBag.h, it checks if it has a variable called _ARRAY_BAG defined. If it's never seen ArrayBag.h before in the compilation, this variable is not defined, so #ifndef _ARRAY_BAG (which means if _ARRAY_BAG is *not* defined) returns true. This lets the code define _ARRAY_BAG for the first time, and read in the header file. The #endif statement is needed to finish the #ifndef statement.

```
#ifndef _ARRAY_BAG
#define _ARRAY_BAG
        … class goes here …
#endif
```

Macro guard case 2

The second time the compiler sees ArrayBag.h in the course of compiling, it checks if it has a variable called _ARRAY_BAG defined. Now, this variable is is already defined, so #ifndef (which means is not defined) returns false. This lets the code sidestep reading in the header file, and avoids the duplicate definition of the ArrayBag functions. The #endif still marks the end of the compiler's #ifndef statement.