Lab 9 - Hermione's Handbag

Introduction

Though small physically, the volume of Hermione's beaded handbag is actually quite vast! In fact, **it is infinite (https://en.wikipedia.org/wiki/Magical_objects_in_Harry_Potter#Hermione's_handbag)**. In the *Harry Potter* series, the small sack Hermione carries, pictured at right, was big enough to hold at least a pair of jeans, a sweatshirt, a pair of socks *and* an invisibility cloak! My sister is a world traveler, but not even she could fit that much stuff in luggage that **size (https://harrypotter.fandom.com/wiki/Hermione_Granger%27s_beaded_handbag)**.

Don't worry, you don't have to be a Harry Potter fan to complete this lab. Nor do you have to believe in Wizardry because, in the end, all you need is some C++ and you, too, can create a beaded handbag of your own!

I solemnly swear that I am up to no good. (https://www.hp-lexicon.org/magic/solemnly-swear-no-good/)

Recall our discussion in class about the difference in C++ between *fundamental* (e.g., `char`, `int`, `bool`, etc) and (precisely) *compound* types. A compound type is any type that is *not* a fundamental type. A helpful definition, I know! A good example of a compound type is an array! Another compound type, and one that we will be exploring in this lab, is the *class type*. Class types are also known as user-defined types. In class we have studied the power of user-defined types for defining and implementing our own *abstract data types (ADTs)*! We can see the true power of C++ when we use the language's syntax and semantics to define our own types that look and act just like fundamental types. When defined correctly, our fellow programmer can use our creations just like they would, say, an `int` or a `bool` or a `double`. Pretty neat!

User-defined types *implement* abstract data types (ADTs). Remember the definition of *abstraction* we learned in class:

> The "art" of removing the detail to simplify and focus on the essence.

That's the long-winded, technical definition. The short, loose definition of abstraction is "hiding". And, as we said in class, an ADT hides data *and* process from the user. In the world of professional software engineering, there *is* a difference between an ADT and a user-defined type:

the former (ADTs) are conceptual entities that are implemented (in code -- C++, in this case) in a user-defined type.

In this lab you are going to write the user-defined types that implement the *Item* and *Beaded-Bag* ADTs. In other words, you are going to implement two classes in C++.

But first, let's make sure that we are clear about our terms. We use ADTs in our software and write object-oriented code because these tools allow us to build our software in a way that models our real-world interactions. Each user-defined type that we implement using a `class` (or `struct`) allows us to write code that models an entity with which we are familiar. So, what is the entity or concept that we are modeling with the *Item* ADT? We are modeling anything that Hermione can put in her bag, of course. And what about the *Beaded-Bag* ADT? What does *that* model? Well, I'll leave that to your imagination but I don't think that it'll be too hard to figure out.

The Item and Beaded-Bag ADTs

Before we can implement the ADTs we have to actually define the ADTs. Recall that ADTs *hide* data and processes (i.e., they are a combination of data and process abstraction) which means that ADTs have “a set of data and a set of operations on that data”.

Let's talk first about the *Item* ADT. For data, each *Item* will have a name. The *Item* ADT will have two operations it can perform on its data:

1. Get The Name: This operation will simply return the name of the *Item.*
2. Is Equal?: This operation will simply return whether *this Item* equals another, according to whether the names of the *Item*s match.

The *Item* is a fairly boring ADT. The *Beaded-Bag* ADT is *far* more magical.

First, for the *Beaded-Bag* ADT we will define the data. Like Hermione’s physical bag, the *Beaded-Bag* ADT holds an infinite number of *Item*s.

Having defined the data of the *Beaded-Bag* ADT, we turn our attention to defining its operations. The user of a bag should be able to

1. Insert a new *Item* into the *Beaded Bag,* as long as it is not already in the *Beaded Bag* -- Hermione does *not need* two toothbrushes, after all!
2. Query whether the *Beaded Bag* contains a certain *Item*.
3. Determine how many *Item*s the *Beaded Bag* actually contains.

The following are the *specifications* of the *Beaded-Bag* ADT's operations:

Operation	Task	Input	Output
insert	Insert an *Item* into the *Beaded Bag* as long as it is not already present.	An *Item* to insert.	Nothing.
contains	Check whether a given *Item* is in the *Beaded Bag* or not.	An *Item* potentially contained in the	True or False depending on

Operation	Task	Input	Output
		Beaded Bag.	whether the given *Item* is contained in the *Beaded Bag*.
size	Determine how many *Item*s are contained in the *Beaded Bag*.	Nothing.	The number of *Item*s in the *Beaded Bag*, as an `int`.

Program Design Task

As my dad always says, "If you are going to wear cargo shorts, at least make sure that they have lots of pockets."

Program Design Requirements

Your pseudocode or flow chart should be a tool that you find useful for completing the entire lab. However, you *must* describe in your pseudocode how you plan on implementing the insert method of the *Beaded-Bag* ADT. In particular, you *must* describe completely how you will determine whether the item that the user wants to insert into the Beaded Bag is already present which will determine whether or not you add it to Hermione's Beaded Bag. You may choose to write the flow chart or the pseudocode at any level of detail but, remember, this is a tool for you!

Programming Task and Requirements

Your programming task for this lab is to complete the implementation of the user-defined type that implements the *Item* ADT and write the entire implementation of the *Beaded-Bag* ADT. Your implementation of the *Item* ADT will be done with the `Item` class. Your implementation of the *Beaded-Bag* ADT will be done with the `BeadedBag` class. If you are a Windows-based C++ programmer, begin with this **skeleton (https://uc.instructure.com/courses/1657742/files/179158039?wrap=1)** **(https://uc.instructure.com/courses/1657742/files/179158039/download?download_frd=1)** . If you are a macOS-based C++ programmer, begin with this **skeleton (https://uc.instructure.com/courses/1657742/files/179158807?wrap=1)** **(https://uc.instructure.com/courses/1657742/files/179158807/download?download_frd=1)** . If you are a Linux-based C++ programmer, start with this **skeleton (https://uc.instructure.com/courses/1657742/files/179158705?wrap=1)** **(https://uc.instructure.com/courses/1657742/files/179158705/download?download_frd=1)** .

Before beginning to code, familiarize yourself with the contents of each of the files:

- `item.h`: Contains the class definition of the `Item` class that implements the *Item* ADT.
- `item.cpp`: Contains the implementation of the member functions in the `Item` class.
- `beadedbag.h`: Contains the class definition of the `BeadedBag` class that implements the *Beaded-Bag* ADT.
- `beadedbag.cpp`: Contains the implementation of the member functions in the `BeadedBag` class.

`Item` Class

The `Item` class has the following member variables declared:

- `m_name`: A `std::string` used to hold the name of the *Item*.

The `Item` class has a constructor defined/implemented that takes a single parameter, the default name of the item, as a `std::string`, and assigns that parameter to the `m_name` member variable.

The `Item` class has the following member functions *declared*:

- `std::string getName();` This member function will return the `m_name` of the *Item*.
- `bool isEqual(Item other);` This member function returns `true` if `other`'s `m_name` is equal to this *Item*'s `m_name`.

Stubs of the implementation of those member functions are present in the `item.cpp` file. Your job is to complete their implementation according to the specification above.

`BeadedBag` Class

You are flying solo on the implementation of the `BeadedBag` class! You must implement the following member functions for the `BeadedBag` class according to these specifications:

- `void insert(Item to_insert);`: This function will insert `to_insert` into the *Beaded Bag as long as it is not already present*.
- `bool contains(Item maybe_contained_item);`: This function will return `true` if `maybe_contained_item` is already present in the *Beaded Bag*; it will return `false` otherwise.
- `int size();`: This function will return the number of *Item*s in the *Beaded Bag*.

You *must* define the `BeadedBag` class in `beadedbag.h` and you *must* implement the member functions in the `beadedbag.cpp` file.

Just how do you hold the group or list of *Item*s in the *Beaded Bag*? It seems like a `std::vector` might do the trick? Yes, that's definitely a good idea! When you declare/define a `std::vector` to hold the *Item*s, make sure that it is a private member variable. You are free to give the member variable any name that you want -- after all, *that* is the point of writing user-defined types that implement ADTS. The internals of the type are hidden from the user and the user doesn't care what happens behind the curtain!

Check Me Out

Like previous labs, you will *not* write a `main` function for this lab. One is provided for you. The provided `main` function will test your implementation. When your `main` function prints

```
Success
```

you can be confident that your code is correct and you should submit to Gradescope for testing.

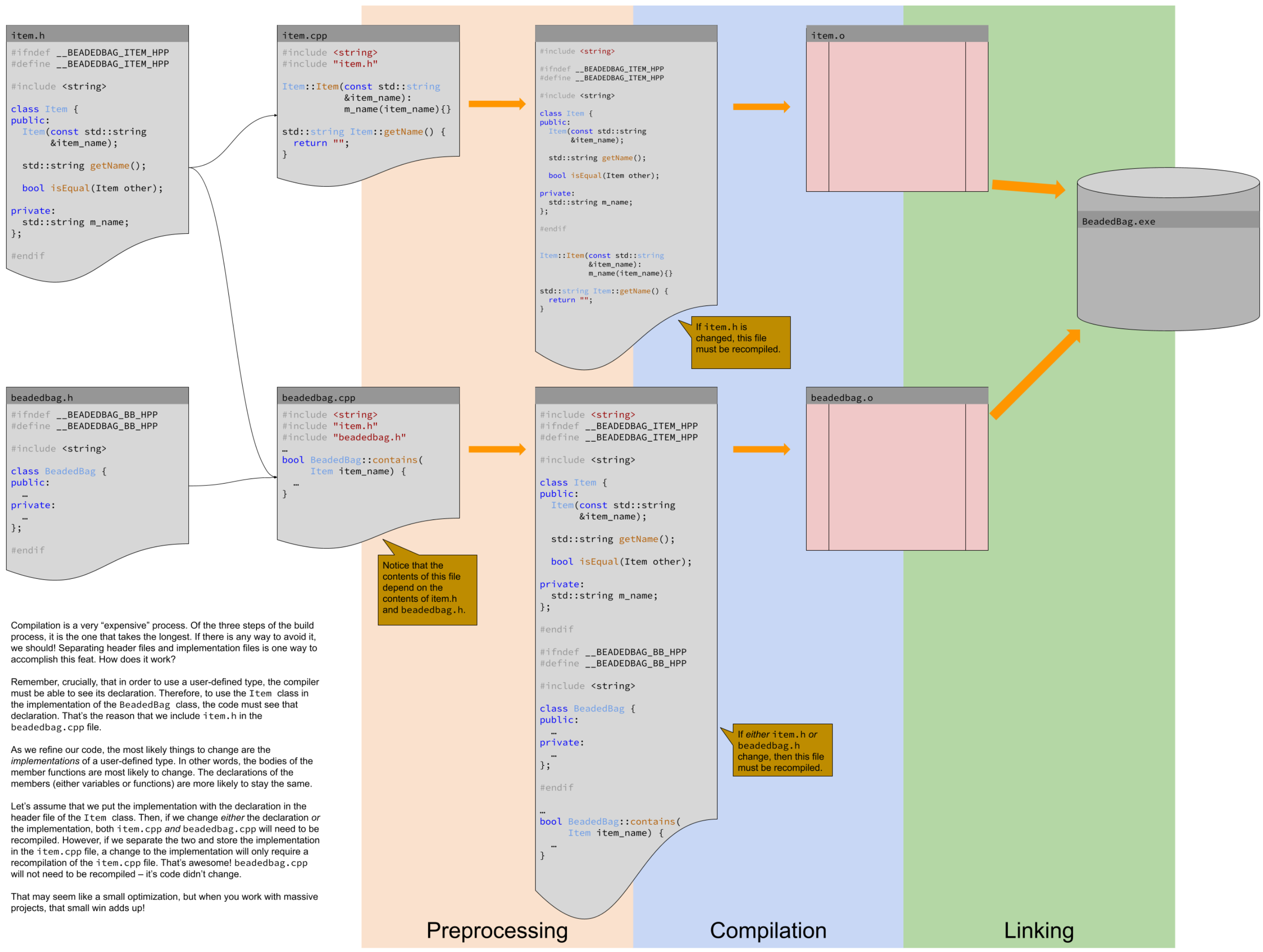
Documentation

All member functions that you write must be commented. You must comment the member functions at their definition (i.e., in the `.h` files) according to the following specification:

```
/*
 * <function name>
 *
 * <short description of what the function does>
 *
 * input: <short description of all input parameters>
 * output: <short description of all output parameters
 *          and the return value>
 */
```

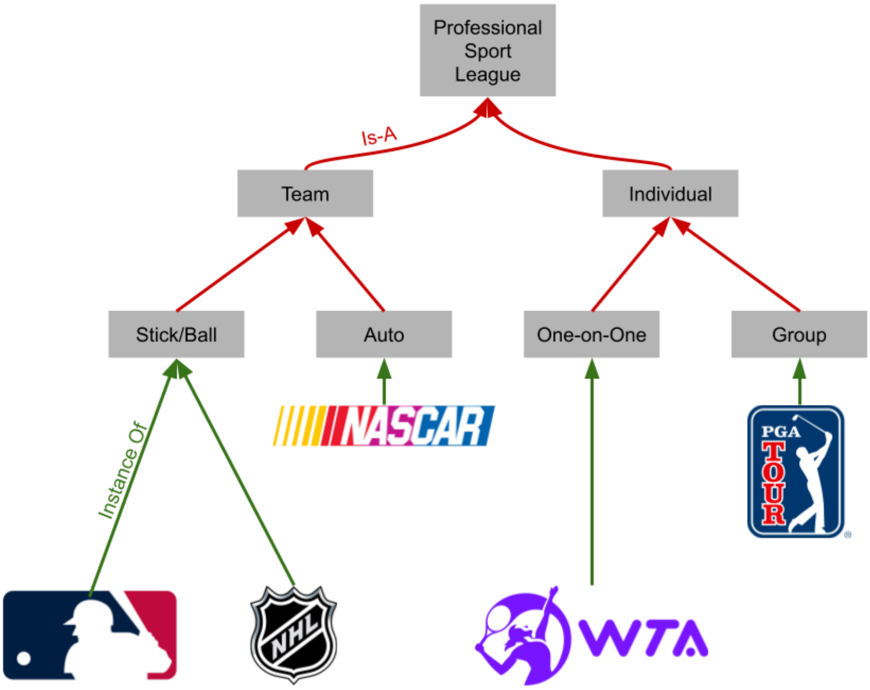
Why Separate Knob?

Just why are there so many files in this lab? Take a look below for a complete explanation!



Critical-Thinking Task

There are many examples of inheritance hierarchies in the real world. For instance, the professional sports leagues in the United States can be ordered into an inheritance hierarchy:



Your job in this Critical-Thinking Task is to come up with a real-world example of an inheritance hierarchy and document it!

Critical-Thinking Requirement:

In writing (500 words or less) or graphically, describe a real-world example of an inheritance hierarchy. The real-world example that you document must have at least two layers of base/derived classes. You must clearly indicate the Is-A relationships among the classes and which elements are instances of a base/derived class. The more creativity that you use in documenting your example, the better!

Question Task:

For this lab you deployed your newfound knowledge of the object-oriented programming style. It is a way of writing programs that is quite a bit different than the way that we have been writing programs for the first 2/3 of the semester. After doing this lab *and* reflecting on object-oriented programming, what do you think is *the* aspect of the object-oriented programming style that will change the way you think about writing code? Is there something about our discussion of object-oriented programming so far that you do not understand? Is there a question you have about how to use object-oriented programming style most effectively? Please feel free to ask anything related to anything that we've learned so far (or will learn)!

Question Requirement:

Submit a file named `question.pdf` which contains your question. Take particular notice of the fact that the question does not have to be subtle, or deep or philosophical for you to receive full points. The goal of the question is for me to be able to get a sense of whether there are common questions about the material which would indicate where I failed to teach the material appropriately.

Submission

1. The pseudocode describing *at least* your plan for implementing the insert operation on the *Beaded-Bag* ADT (named `design.pdf`).
2. The C++ source code for your classes (named `beadedbag.h, beadedbag.cpp, item.h and item.cpp`).
3. A written or graphical example of a real-world instance of an object-oriented, is-a hierarchy (named `hierarchy.pdf`).
4. A written question about the material covered so far in this class in PDF format (name the file `question.pdf`).

Grading

Your submission for this lab will be graded according to the following rubric:

Points	**Task**	**Requirements**
45	Programming	`BeadedBag` passes all tests executed by the autograder.
10	Programming	All member functions that implement the B*eaded-Bag* ADT's operations are properly documented according to the standards set forth in this lab.
5	Programming	All variables are given proper types and meaningful names.
10	Programming	All member functions that implement the Beaded-Bag ADT's operations are in the `beadedbag.cpp` file.
5	Programming	`BeadedBag` class contains *only* private member variables.
20	Critical Thinking	Your response to the critical thinking task documents a real-world example of an inheritance hierarchy, is at least two levels deep, clearly indicates which are the base/derived classes and labels all Is-A relationships.

Points	**Task**	**Requirements**
5	Question	Your question shows an engagement with the material.

Associated Learning Objectives

1. Implement basic ADTs using C++.
2. Use `std::vector`s.
3. Distinguish between ADTs and data structures.
4. Understand class hierarchies and the Is-A relationship.