Lecture Summary

In this lecture, we continued with the topic of template classes, and learned about inheritance between classes; both are separate ways to increase code reuse, and avoid lots of duplication.

1 Templates, and the const keyword

In the previous lecture, we had begun our exploration of templates, as a way to produce the “same” data structure, but allowing it to store objects of different types (int, string, ...). In this way, we only have to write and debug the code once, and can then reuse it for many data types. The syntax for class definitions was:

```cpp
template <class T>
class LinkedList
{
    ...
}
```

When we implement class functions, the syntax is as follows:

```cpp
template <class T>
LinkedList<T>::add (T item) {
    //code
}
```

Note the `template<class T>` line and the `<T>` in the class name. You have to put these in front of each member function definition. Usually, if you get a “template undefined” error, you might have missed a `<T>`. (Note that the identifier `T` is arbitrary — you can use `X` or `Type` or whatever). Templates were kind of grafted on top of C++, and not really part of the initial design. That is why the syntax is rather unpleasant, and the “obvious” way of doing things causes all kinds of compiler and linker problems. In order to get things to work, you will sometimes need to deviate from good coding practices, for instance by having `#include` of a `.cpp` file. Check the course coding guide and Piazza discussions for information on how to deal with some of the most common problems.

**The const keyword:** Because the type (referred to by `T` above) could be large, such as a custom type (think about a linked list each of whose items is itself a linked list), it is usually preferable to pass the parameters by reference, to avoid copying the whole item. So we would revise the syntax for the above definition as follows:

```cpp
template <class T>
LinkedList<T>::add (T & item) {
    //code
}
```
But now, because `item` is passed by reference, it becomes susceptible to overwriting: the function `add` would be allowed to change the fields of `item`. If another person sees only the function signature, how can that person be sure `item` doesn’t get overwritten? They may be worried about the side effects of the `add` function. The way to solve this is to use the `const` keyword, with the following syntax:

```cpp
template <class T>
LinkedList<T>::add (const T & item) {
    //code
}
```

By adding `const`, we ensure that the code inside the function can not alter the value of `item` or any of its fields (if it is a more complex variable). If we include code that tries to alter `item`, the compiler will return an error. The main point of `const` is to make it clear to others using the function(s) about what can and cannot happen to the parameters.

Another way to use `const` is as follows:

```cpp
LinkedList<T>::contains(const T & item) const {
    //code
}
```

In this case, what the `const` says is that the function `contains` will not change the object on which it is called, i.e., after it executes, the object will be in the same state. This means that the function cannot alter the variables of the object. It is helpful to mark all `const` functions as such. Imagine that you are passing a linked list as `item` into the `add` function above; now, you could still call `item.contains()`, since it is guaranteed that it cannot alter `item`, and therefore does not violate the `const` part of the parameter.

### 2 Inheritance

Suppose that a friend has coded an integer (or perhaps templated) bag class based on linked lists and you want to create a deluxe version of it. The deluxe class `DeluxeBag` would do the same thing except that it has a `checkEmpty` function and an upgraded version of your friend’s `print` function. There are two natural ways of accomplishing this.

1. We could just change our friend’s code everywhere. But our friend may not like this; perhaps, he/she needed the particular format of printing the list, and we just destroyed his/her code.

2. We could just copy and paste our friend’s code and write our own modifications into the copied version. But that also has problems:
   - There are now two versions of the original code, so changes made to one have to be carefully copied over to the other. In particular, if our friend discovers a bug in his/her code, it now has to be fixed in two places (or more, if we made different versions).
   - Copying/pasting is very inelegant as a general programming technique.

Both methods also require us to have access to our friend’s complete source code in the first place, and not just the compiled executables and headers.

The solution is called `inheritance`, and it is really the central notion of object-oriented design/programming. Given a class `A`, inheritance allows you to define a new class `B` that ‘inherits’ (automatically copies) all data and functions from `A`. Further, it allows you to modify and add functions as necessary, and overwrite existing functions. The basic syntax is as follows:

```cpp
class B : public A
{
    ...
}
```
What this does is basically just copy-paste everything from A into B (except what you overwrite). The difference is that as soon as something is changed in A, the changed version is also used in B, so you only need to fix things in one place, and — unless you use templates (which create some problems) — you don’t even need access to the implementation of A, just the header file and compiled version.

In class, it was asked if we can have two functions of the same name, for instance, because we had a function already, and add another one. The answer depends:

- If they have same name and the same arguments, then this does not work, just like you cannot have two functions with the same signature in a program. The compiler wouldn’t know which one you are referring to. If you write a function with the same name and signature in the class B, then it overwrites the one from class A.

- If the functions have different signatures (number and/or types of arguments), then this is okay. The compiler can tell apart which one you want based on the arguments you are passing it.

Now that we hopefully understand the basics of inheritance, let’s build DeluxeBag by inheriting from IntLinkedListBag. In class we wrote up our souped-up print function (with “Extra Charge” and pretty asterisks) for our DeluxeBag. Inside that function, to avoid doing too much work ourselves, we want to call the parent class’s (IntLinkedListBag) print function. Here is how that worked:

```cpp
void DeluxeBag::print() {
    cout << "Deluxe Bag: Extra Charge for Printing" << endl;
    cout << "***********************************************************************************" << endl;
    IntLinkedListBag::print();
    cout << "***********************************************************************************" << endl;
}
```

Note how we specify the print function of IntLinkedListBag instead of DeluxeBag’s print function. (Otherwise, print would recursively call itself.) In general, this is how a class calls its parent class’s function: if B inherits from A, the code inside B should look like:

```
A::functionName(parameters);
```

Of course, if we did not add a function of the same name (as we did with print here), we can leave out the A:: part, and just call it as functionName(parameter), since the function is inherited, and part of our new class B.

Next, we also added a new function to test whether the bag was empty:

```cpp
bool DeluxeBag::isEmpty () {
    return (head == NULL);
}
```

But when we tried to compile this, the compiler noted an error, and claimed that our DeluxeBag did not have a variable called head. Didn’t we inherit from IntLinkedListBag, which has a variable called head?

The problem was that head is declared as private in IntLinkedListBag. The keyword private makes the variable very restricted: not even an inheriting class gets to access the variable (though of course the data item is stored in the class). A version that resembles private but allows all inheriting classes to access the variable directly is protected. Here is a summary of who is and isn’t allowed to access variables or functions of a class:

- **public**: everyone (other classes, this class) can access the field.
- **private**: only objects of the same class can access the field.
- **protected**: only objects of the same or inheriting classes can access the field.
Friend Classes: We did not talk about this in class, but there is a mechanism to give one class access to the private variables of another class. You can declare that class A is a friend of class B. The code would be:

```cpp
class B {
    public:
        friend class A;
    private:
        ...
}
```

This piece of code would ensure that A would be allowed to access all the private parts of B as well.

Inheritance Visibility: You were probably wondering why in inheritance, we were writing `class B : public A`, in particular, why we needed the word `public` here. When you have a class B inheriting from A, sometimes, you want to restrict access to variables that were public in A, i.e., make them private or protected in B. For instance, you may want to inherit the functionality of a linked list to use it, but show the world a different set of functions to call, and keep the other ones only for internal use. Then, you’d want them to become private. To do that, you can inherit in one of three ways:

- **public:** All elements in A have the same protection in B as they had in A, so everything stays as it was.
- **protected:** Private elements of A remain private in B, but protected and public elements of A both become protected in B.
- **private:** All elements of A become private in B.

So by writing `class B : private A` instead, we would ensure that whatever B has inherited from A cannot be accessed from any other classes: neither those inheriting from B, nor those using objects of type B.

Multiple inheritance C++ allows a single class to inherit from multiple classes. We strongly advise you not to use this feature, as quite a few things can go wrong. For instance, if the two classes you are inheriting from have variables or functions of the same name, you get clashes. For some discussion, look up “C++ Diamond of Dread”. If you find yourself inheriting from more than one class, it is often a sign that you should rethink your program’s class structure.

The most frequent “legitimate” use of multiple inheritance is when you want to genuinely inherit the functionality of one class, and add functions to meet all the specs of a pure abstract class. So from the second class, you really don’t “inherit” per se, but you just implement the functions that that class wants. Unfortunately, in C++, the only way to do that is multiple inheritance. In other more naturally object-oriented languages (such as Java), purely abstract classes which specify functions you’d like to see in your class are distinguished from actual classes; in Java, they are called Interface. Then, you can say that your class inherits one class, and implements another interface. Unfortunately, C++ does not have a mechanism for this, so sometimes, you are stuck with multiple inheritance.

2.1 Class Relationships

When you build a large project, you will often have objects of dozens, or even hundreds, of different classes interacting in different ways. While at first, this may seem overwhelming, if you define your class hierarchies and relationships carefully, it actually helps you organize and understand the way your code works, compared to just writing thousands of lines of code without much structure. Forming high-level conceptual ideas about how your project is organized can be really helpful. The following three are very standard relationships between classes. (There is way more along these lines: a topic called “design patterns” explores many different standard patterns in which classes interact.)
Is-A: We say that B is a(n) A if B is a more specific version of A; B has all the functionality of A and then some more. For instance, every cat is a mammal: it has all the abilities of a generic mammal, plus many cat-specific ones.

When you write code, is a relationships are typically implemented by using public inheritance: you want all the functionality of A to remain available, while possibly adding more functions or data to B.

As-A: B as a(n) A refers to implementing a class B using the functionality of A. B will typically look like something completely different to the outside world, but internally, it is implemented by using just A’s functionality. For instance, when you sleep on someone’s couch, you are implementing the functionality of “bed” using a “couch”. (You implement “bed” as a “couch” — even though that’s the opposite as you would refer to it with the verb “use”.) If instead, you sit on the couch, you may be implementing the functionality of “chair” using a “couch”.

When you write code, as a relationships are typically implemented by using protected or private inheritance: you do not want other classes to see the underlying functionality, only the new stuff you implement using it.

Has-A B has a(n) A if one of the fields of B is of type A. For instance, you car probably has a radio. That does not mean that your car itself has a function “switch station”, but that it contains an object which itself has that function.

Has a relationships require no inheritance; they simply involve creating a member variable of type A in B.

The boundaries between has a and as a can be pretty fluid, and sometimes, it makes sense to think about solving a particular design issue in either way. (For instance, are we implementing a Wall as a list of WallPosts, or does a Wall contain a list of WallPosts? A case could be made either way.)