Lecture Summary

In this lecture, we went in detail over the class policies, and saw an overview of what we will be learning in class this semester. If you have not studied the course web site, please do so carefully — it contains a lot of information on how this course will be run. The topics of this class are all centered around “data”: data structures are how you organize your data to support fast operations. Object-oriented programming puts the focus of programming on the data, often resulting in cleaner and modular code. The two often go hand in hand.

As a result of our introductory CS classes (CSCI 101, or CSCI 103 in the future), most students are probably somewhat proficient in basic programming, including the following basic features:

Data types int, String, float/double, struct, arrays, ....

Arithmetic

Loops for, while and do-while

Tests if, else and switch

Pointers

Functions and some recursion

Other I/O and other useful functions

In principle, these (in fact, just while loops, if tests, integers, and arithmetic) are enough to solve any programming problem. Everything that we learn beyond those basics is “just” there to help us write better, faster, or more easily maintained or understood code.

The way most students are probably thinking about programming at this point is that there is some input data (maybe as a few data items, or an array), and the program executes commands to get an output from this. Thus, the sequence of instructions — the algorithm — is at the center of our thinking.

As one becomes a better programmer, it’s often helpful to put the data themselves at the center of one’s thinking, and think about how data get transformed or changed throughout the execution of the program. Of course, in the end, the program will still mostly do the same, but this view often leads to much better (more maintainable, easier to understand, and sometimes also more efficient) code. Along the way, we’ll see how thinking about the organization of our data can be really important for determining how fast our code executes.

Example 1 As an example, let’s think of phone books. Those were thick books in which all registered owners of phones were listed alphabetically, with their numbers next to their names. Usually, by exploiting the alphabetical sorting, one can find a number corresponding to a given name quite fast, in a few seconds.

Now imagine what would happen if the entries were in random order, or sorted by phone number. We could still solve the lookup problem, but it would be much, much slower. By going from a sorted array to an unsorted one, we have drastically altered the efficiency of an operation we would like to execute.
Phone books are long, and they are updated rather infrequently, but referenced quite frequently. This makes sorting them very desirable, since we would like to optimize lookup time.

If the updates were very frequent, while the lookups happen rarely, things would be different. Now, we should optimize the update time, and sorting the entire book may be too expensive.

Similarly, if the phone book only contained a few entries (like perhaps one’s own list of friends), then sorting them may not be worth the trouble, as it is easy to scan through the short list when needed.

The main take-away message is that the answer to “What is the best way to organize my data?” is almost always “It depends”. You will need to consider how you will be interacting with your data, and design an appropriate structure in the context of its purpose. Will you be searching through it often? Will data be added in frequent, short chunks, or in occasional huge blocks? Will you ever have to consolidate multiple versions of your structures? What kinds of queries will you need to ask of your data?

The most obvious data structures we can think of will be similar to arrays (sorted or unsorted), since that’s how we usually organize data on paper. However, some of the organization of data even in printed form is a little more elaborate. For instance, think about the tax tables in IRS forms. Based on a rough grouping of your income, you are sent to different pages of the document, where you look up stuff based on marital status and precise income. The whole thing looks more like trees (which we will see later in class), implemented in an array; this is also a technique we will learn later.

Once we start realizing the important role that data organization plays in good programming, we may change the way we think about code interacting with data. When starting to program, as we mentioned above, most students think about code that gets passed a data structure (array, structs, etc.) and processes it. Instead, we can think of the data structure itself as having functions that help with processing the data in it. In this way of thinking, an array in which you can only add things at the end is different from one in which you are allowed to overwrite everywhere, even though the actual way of storing data is the same.

Example 2 To illustrate this, let’s return to a variation of the phone book question, which we’ll call Dictionary. Our dictionary will contain words and their definitions, say, both stored as strings. (By making the “definitions” numbers, we get our phone book back.) In order to make such a data structure useful, we need to be able to do three things:

- Insert word-definition pairs into the dictionary.
- Remove words from the dictionary. (We may be able to get away without this, perhaps.)
- Look up words in the dictionary, and return their definitions, or report that the word is not in the dictionary.

These supported operations — rather than any particular way in which we store the data — are what makes a dictionary useful as such. In fact, we will see several implementations for dictionaries throughout the semester, optimized for different frequencies of the operations.

The important thing to take away is that what makes a dictionary a dictionary is the combination of the data it contains and the operations it supports. This combination is what we call an abstract data type.

When implementing abstract data types — and for many other aspects of good programming — object-oriented design is a very natural fit. An object consists of data and code; it is basically a struct with functions inside in addition to the data fields. This opens up a lot of ideas for developing more legible, maintainable, and “intuitive” code, while achieving modularity, i.e., ensuring that different pieces of code can be analyzed and tested in isolation more easily.

Returning to our example of dictionaries, imagine that you are developing an application with a friend. In order to support something you want to do, you need a dictionary data structure. You charge your friend with developing it for you. The important thing to notice is that so long as it supports the three operations correctly (and fast enough), you don’t care how he/she implements it. This is called abstraction — separating that “what” from the “how”. It also is closely related to encapsulation — shielding as much
of the inside of an object as possible from the outside, so that it can be changed without negatively affecting the code around it. The textbook refers to this as “walls” around data.

The learning goals of this class are twofold:

1. Learn about good programming practice with object-oriented design, in particular as it relates to implementing data storing objects.

2. Learn the basic and advanced techniques for actually implementing data structures to provide efficient functionality. Some of these techniques will require strong fundamentals, and their analysis will enter into more mathematical territory, which is why we will be drawing on material you will be learning simultaneously in CSCI 170.