**Review and Research**

You can review the video of Session 4 here:

[Session 4 - Data Structures #1](https://blizzard.sharepoint.com/portals/hub/_layouts/15/PointPublishing.aspx?app=video&p=p&chid=8aa7fa80%2Dbfa2%2D4021%2Dbf44%2D543dba93f693&vid=47d43cd2%2Db869%2D420c%2D8e80%2D8689b4be837b&from=1)

As always, email me at [semerson@blizzard.com](mailto:semerson@blizzard.com) if you have any questions!

Here’s a nice MS document detailing how to program with arrays:

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/>

And here you can find the official documentation on the List<> and the LinkedList<> respectively. **Please give these a look!** These documents are a little wordy and are heavy on jargon, but they will give you a great overview of what functionality you can get out of these containers. You’ll save a lot of time in the long run if you get into the habit early of learning about what tools the standard libraries offer you:

<https://msdn.microsoft.com/en-us/library/6sh2ey19(v=vs.110).aspx>

<https://msdn.microsoft.com/en-us/library/he2s3bh7(v=vs.110).aspx>

Here’s a good reference for the switch statement. Please note that more recent versions of .NET have expanded the functionality of switch statements somewhat, allowing cases to include simplified compound conditions using the “when” keyword:

<https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/switch>

As a follow-up to our bubble sort exercise, here’s a cool interactive slide deck that Steve found that describes various sorting algorithms in a fair amount of detail complete with visualizations of how they work. The language here is a bit jargon-y in places, and you’ll come across concepts that we haven’t discussed in class (Big O notation, for example), but for those who are interested in learning more I highly recommend checking it out and researching any additional concepts that seem interesting to you:

<https://visualgo.net/en/sorting>

**Glossary**

Array – A collection of objects of similar type, stored contiguously in memory and with a fixed length. Items stored in arrays can be accessed by use of an index (or subscript) notated in square brackets. Index numbering begins with 0 and runs through n – 1, where n is the length of the array. If you attempt to access an array item outside of this range your program will trigger an exception (or error) that informs you that this is the case.

Switch – A switch statement (sometimes called a case switch statement) is a language feature found in many languages, including C#, that allows you to write multiple if / then / else statements in a simpler, more readable format. There are limitations to switch statements; be sure to check out the documentation linked above to learn what they are!

Algorithm – An algorithm is a step-by-step process that solves a particular problem. The first “textbook” algorithm we examined in class was the bubble sort. Learning how to design and implement algorithms is one of the most crucial things for you to do as you seek to improve your skills as a software engineer!

Data structure – A data structure is a way of organizing data such that it can be stored, retrieved, and organized. Many data structures can be thought of as containers or collections of data.

List – In C#, a List is a data structure that contains a resizable array. Lists are powerful, efficient containers that come with a variety of methods to make sorting, interpreting, and accessing data easier. **Note:** The word “list” can be very ambiguous in different programming contexts; for example, while a List in C# represents a contiguous, resizable, homogenous container, in Python a List can contain items of different types, and in C++ the standard library List is actually a linked list.

Linked list – Linked lists are data structures that are similar to arrays in some ways but very different in others. Instead of storing items in contiguous memory, lists are comprised of separate notes that can reside nearly anywhere in memory, each with at least one pointer that indicates where another node in the sequence can be found. Linked lists do not provide random access, and iterating through them is generally slower than iterating through an array; however, they are extremely fast at insertion and deletion.

**Problem Solving Techniques**

In Session 4, we discussed a number of techniques you can use when attempting to solve a problem with code. Here is a quick roundup:

* Understand the problem before you begin planning a solution. Make sure you know what kind of input to expect and what should be done with that input. Think about possible edge cases and how you might deal with them. Most of all, make sure you’re solving the right problem!
* Plan before you code. It is tempting to jump straight into your development tools and begin writing code immediately, but a bit of planning will ultimately save you time, make your code less error-prone, and help prevent the frustration of not knowing how to proceed. If you find that you hit a wall when writing your code, don’t be afraid to stop and re-examine your plan (and revise it if necessary). When planning, you don’t necessarily need to focus on language features; think about the problem in real-life terms, and feel free to sketch out your ideas on paper to help make them more concrete.
* Break big problems down into smaller problems. When we implemented bubble sort in class, we first thought about how to sort a single number in an array rather than *all* the numbers. We then tackled how to repeat the process, how to manage the number of iterations in each loop, how to know when to stop, and how to swap values. By thinking about solutions to each of these smaller problems in their turn, we chip away at the larger problem until we have a workable solution.
* Use concrete examples when you plan. If you’re trying to devise a sorting algorithm, don’t just think about sorting in the abstract: Write down simple sets of actual numbers (or other values) to work with, and from those specific examples begin to create generalized solutions. Problem solving is almost always easier to do when you work with concrete values rather than jumping straight into purely abstract concepts!
* Consider extreme cases, i.e., extremely large cases or extremely small cases. This is useful for solving all kinds of problems! Consider this simple, classic riddle: You have a drawer with 10 red socks, 12 blue socks, and 5 yellow socks inside. If you begin pulling socks out of the drawer at random without looking at the color, how many socks must you take to guarantee that you grab at least one red sock? One way to easily come up with a general solution to this kind of riddle is to use an extreme case. Let’s go for an extremely small case and imagine we have exactly one sock of each color. It’s very easy to see that, in order to be sure you get the red sock, you have to take all three socks out of the drawer; until the yellow sock and blue sock have been removed you’re not guaranteed to get the red sock. Once we understand this, it’s easier to generalize for other cases: To be guaranteed a red sock, you must take the total number of yellow and blue socks, plus one more. Believe it or not, in software engineering we often solve very complex problems using just this kind of thinking! When choosing concrete examples to work with, don’t be afraid to start with extremely simple cases; the solution is usually easier to find, and will often scale up to work for more complex cases as well.

**Practice Exercises**

These exercises are grouped around concepts we’ve covered in class and range from very simple to more complex. As always, a few reminders:

1. When compiling and running these exercises in Visual Studio, be sure to do so in Debug Mode. You can do this with the hotkey combination ctl+F5.
2. These exercises are meant to be a start to your practice, but if you want more ideas, contact me and let me know. I’ll be happy to make additional suggestions!
3. Don’t be afraid to repeat an exercise several times. Repetition in your practice is very helpful, especially in the beginning when you’re trying to get comfortable with syntax as well as new concepts.
4. If you get stuck on a problem or don’t understand why something is happening, please contact me and let me know. I’ll be happy to help.
5. An exercise may occasionally require you to use a concept we haven’t covered in class; when this is the case, the exercise will be marked with an asterisk there will be a link to research you can do help you find the new information you need.
6. Don’t be afraid to use Visual Studio’s debugging features to help you understand how your program is working! This can be useful not only for fixing problems but also for gaining a better understanding of how your program is working.

*Data structures and problem solving*

NOTE: The last few exercises in this list are a bit more challenging; be sure to think them through and plan carefully, and be ready to debug and make modifications as necessary!

Ex 1. Declare an array of 50 integers. Assign each item in the array to consecutive multiples of 5, starting with 5: ary[0] = 5, ary[1] = 10, ary[2] = 15, etc.

Ex 2. Write a function that, given an array of strings as input, prints out the length of each string. If you need help, do some research on strings to learn how to determine their length.

Ex 3. Modify the function in Ex. 2 to return the length of the longest string (or strings, it there are ties) in the array.

Ex 4. Write a function that, given a string as input, returns a new string that reverses the characters of the input string. For example, if the input is “Apple”, the return string should be “elppA”. HINT: Remember that you can access the individual characters in a string by using an array-like index:

String str = “Hello!”;

char c = str[1]; // c will be equal to ‘e’.

Ex. 5. Write a function that, given a two-dimensional array of integers, returns the sum of every integer in the array.

Ex. 6. Create a class that represents a student. Provide the class with a field to represent the student’s name and a private list of floating-point point values to represents the student’s grades. Provide the class with an AddGrade(float) method that allows you to add a grade to the student’s grade list, and a GradeAverage() method that returns the student’s current average score.

Ex. 7. Bonus: Add a GPA() method to the above class that returns the student’s grade point average based on the average grade.

Ex. 8. Write a function that, given a list of integers, returns a new list that contains all of the even integers in the input list.

Ex. 9. Write a variation of the function described in Ex. 8. Instead of returning a new list, have your function modify the input list by *removing* all odd integers.

Ex. 10. Write a simple encryption function. Given a string as input, the function should return a new string that shifts each letter up by one. For example:

“apple” becomes “bqqmf”

Assume all input will be in lower-case letters with no numbers or special characters. The character ‘z’ should wrap around and become ‘a’. HINT: Characters are very closely related to integers and can be converted to integers and manipulated with mathematical operations. Try doing some research to learn how.

Ex. 11. Write a function that takes two lists of strings as input. The function should read each string in the second list and, if there are any instances of that string in the first list, remove them. Given the following lists as input:

List1 contents: {“Alice”, “Bob”, “Alice”, “Carol”, “Daphne”}

List2 contents: {“Alice”, “Daphne”}

…the function should modify the first list so that it contains the following:

List1 contents: {“Bob”, “Carol”}

Ex. 12. Write a function that, given two strings as input, determines whether or not the first string contains the second string. For example, given “abcdefg” as the first string and “cde” as the second string, the function should return true, because the first string contains “cde.”