CSCI 1300 – Practicum 2 Review Guide

Strings

- Sequence of characters
- Characters: represented as integers - ASCII correspondence table
- We study “library strings”, so #include <string> is necessary
- Every character has an index
- We can print every character individually by *traversing the string*:
  - Usually done using a *for* loop

- Operations with strings (from Ch. 2.5)
  - Concatenating strings with +
  - If not initialized at declaration, a string variable will not get “garbage”, it will be assigned the empty string ””
  - User input for strings: it will save everything up to a space or a new line character. Example: user enters “Harry Potter”, but only “Harry” is assigned to the cin variable
  - The *substr* function (it’s overloaded):
    - Two parameters: the starting index and the length of the desired substring
    - One parameter: the starting index, in which case the substring will go to the end
- Traversing strings with *for* loops:
  - Looking for a certain char:
    - How many ‘a’s are in the string? (counting matches)
    - How many lowercase letters are in the string?
  - Print all the characters in the string
    - Print reverse
  - What is the first occurence of a character (or a substring)?

Common Loop Algorithms (4.7)

- Some algorithms concern strings (counting matches)
- Accumulator loops. With a series of numbers (array, or entered by user via keyboard):
  - Compute the sum of all numbers, or
  - The average of all numbers,
  - The maximum value, or
  - The minimum value
- Finding duplicates (i.e. adjacent characters that have the same value in a string)

Arrays (Ch. 6; *Vectors will be studied Week 10, not on Practicum II*)

- Collections of items of the same type
- Declaring array variables:
  - Type must be there
  - Num elements could be missing
  - {} notation at initialization. If size is 10 and you only specify values for 5 elements, the remaining elements get **default** value
- Referencing an element with []
- Index goes from 0 to … well, we have no function to tell us the length, like with strings
  - We always carry with us a variable to hold the number of elements in the array
  - We could reference an element with an index passed the bounds of the array, but it could lead to segmentation faults - no error when compiling

Common array algorithms:

- Fill an array - with user input via keyboard or file (later this week and next week)
- Copy an array
- Max, Min, Sum, Product, Avg
- Find the position of a certain value
- Swapping array elements
- Traversing arrays with for loops
  - we need to know the size of the array
- Reading an unknown number of inputs - *partially filled arrays*

Arrays and functions (6.3)

- Distinction between the function declaration (where the array is a parameter) and the function call
  - Where does the [ ] show up? Not in the function call.
  - When we pass a variable we only put its name as an argument.
- Reference parameters = we are in fact passing access directly to the elements of the array (in memory). As opposed to before, when we were passing a copy of the value of the argument.
- You cannot return an array in a return statement. But we can modify the elements of the array passed as a parameter

2D arrays:

- Think of it as a matrix, or a table
- Declaring 2D arrays [ ] [ ]:
  - How do we write initialization with { }?
  - Think about it as an array where every element is an array
- Accessing elements with 2 indices [i][j]
- Traversing every element of the 2D array (for printing for example):
  - with a nested for loop
- Traversing elements in a row
- Traversing elements in a column
- Passing 2D arrays to a functions:
  - Do we need to pass both dimensions to the function as well?
  - Answer: we specify the number of columns at function declaration (example: this function can only accept 2D arrays with 5 columns), and then pass the number of rows as a parameter.

File I/O - Reading and writing text files:

- A file is a stream of characters
  - Even if we don't see them they are still characters (blanks: space, \n, \t)
- Input stream is a source of data (usually a file)
- Output stream is a destination for data (also a file)
- We need:
  - A new library <fstream>
  - File stream objects that will help us read and write from/to files:
    - ifstream for reading
    - ofstream for writing
  - Think about ifstream and ofstream as new data types (int, double, char). So we need to declare variables of type ifstream or ofstream

- Template (in file *filetemplate.cpp*):
    1. Create a file stream variable
    2. Open a stream. This will associate a file stream variable with a file (be careful how you write the path to the file as a string). Use the member function .open("filename.txt")
    3. Read the data: we usually read the stream of characters in the file line by line, using getline()
    4. Process the stream: as you read each line, you can parse, convert it into numerical values, display it (cout) or save it for future use.
    5. Write stuff to the output file (if necessary)
    6. Don't forget to close the file connection: .close() - does not require the file name anymore, it is already associated with the file.
- *Reading from a stream*:
    - Line by line (whole lines): using getline() - we will use this method the most (by far). Use getline() in a loop until no line left
    - Word by word using the >> operator. Recommended when we know the structure of the file (for example: there will be 3 real numbers separated by tabs on each line in the file)
    - Reading character by character: using get() - we will use this method when we are looking for a certain character that would help us parse the text.
        - Good useful thing to know: "Reading a number only if it's a number"
        - Useful things to know when reading char by char, or when parsing: isdigit(), isalpha, islower, isupper, isspace
- *Writing to a stream*:
    - Same as cout << but using the ofstream variable instead