Review Notes: Midterm 1

CSCI2270: Data Structures

C++ Fundamentals

1. Command-line arguments: Remember to typecast necessary arguments (**stof()**, **stoi()**)

```
./commandLine arg1 arg2 arg3
```

4	"commandLine"	"arg1"	"arg2"	"arg3"
argc	argv[0]	argv[1]	argv[2]	argv[3]

2. A structure in C++ is a way to bundle data together. For example, the student information can be maintained using a struct which may contain their student ID, full name, age, GPA, etc.

```
struct student {
      unsigned int student_id; string name; int age; float gpa;
};
```

A class is an extension of a struct which also combines data with associated functions and introduces membership criteria (private v/s public). **A struct is like a class with all members being public.**

```
class student {
      unsigned int student_id; string name; int age; float gpa;
      void print_student_info();
      void update_gpa(float updated_gpa);
};
```

3. Pointers are variables that store (or point to) a memory address. References (&) give memory address of a variable. e.g.: `int x = 10; int *ptrX = &x; student* stu = new student;`
   a. Use **new**, **delete** keywords to allocate memory and store address in a pointer
   b. Dereferencing Operators (`*, ->`): `*ptrX == 10; stu->name = "John Doe";`

4. A function in C++ is a subroutine that can be called by other code routines or subroutines. A function may take arguments and may return a value (both optional). e.g.

```
void reset_value(int c, int d) { c = 7; d = 8; }
void reset_pointers(int * c, int * d) { *c = 7; *d = 8; }
void reset_reference(int & c, int & d) { c = 7; d = 8; }

int main() {
      int a = 3; int b = 4;
      reset(a, b);
      reset_pointer(&a, &b);
      reset_reference(a, b); }
```

a. **<u>Pass by value</u>**: When the ***reset*** function is called from main, the value of *a* in main will be copied over to the argument c, and the value of b in main will be copied over to argument d.
b. **<u>Pass by pointers</u>**: When the ***reset_pointers*** function is called from main, the address of ***a*** in main will be copied over to the argument ***c***, and the address of ***b*** in main will be copied over to argument ***d***. Both parameters ***c*** and ***d*** are pointers to int, and contain the addresses of ***a*** and ***b***. Dereferencing ***c*** and ***d*** will change the values of the original arguments ***a*** and ***b***.
c. **<u>Pass by reference</u>**: When the ***reset_reference*** function is called from main, parameters ***c*** and ***d*** will act as references to the original variables ***a*** and ***b*** in main. Updating ***c*** and ***d*** will change the values of the original arguments ***a*** and ***b***.

<u>Arrays</u>

1. **Basics:** A contiguous chunk of memory, indexed with an integer.
2. **Allocation:** Static (`int array[100]`) or dynamic (`int * array = new int[100]`)
3. **Accessing notation (array/pointer):** `a[i] == *(a+i) == *(i+a) == i[a]; // true`
   a. Pointer accessing the first element of the array: `int * ptr = &a[0];`
   b. Pointer accessing the ith element of the array: `int * ptr = &a[i];`
   c. Pointer arithmetic: `ptr++` will access the next element, `ptr--` will access the previous.

   Don't access the array out of bounds! (< 0 or > array size)!
4. **Deallocation:** You can only manually delete a dynamic array: `delete [] array`

<u>Linked List</u>

1. **Basics:** A linked list can only be accessed using its first node, pointed to by a pointer ***head***. Each node contains ***data*** and a pointer ***next***. ***next*** points to the next node in the list; for the last node ***next* = *nullptr*** or ***NULL***. **Never modify *head* unless inserting or deleting at *head*.**

2. **Traversing the linked list and searching for an element:** Initialize a pointer ***temp*** to ***head*,** and hop ***temp*** through the linked list (***temp*** = ***temp->****next)* using a loop*. <u>Hint</u>: If searching based on index, a for loop is a good idea (or a while loop with a counter). If searching based on the node value, a while loop is easier. (However, with the right design, either loop can be used for both the cases)*

```
while(pres) { pres = pres->next; }
```

3. **Deleting a node:** Use two pointers ***prev*** and ***pres*** to traverse the linked list such that ***prev*** is one node behind ***pres***, until ***pres*** points to the node to be deleted. Then ***prev->***next = ***pres->***next, and then delete ***pres***. Recall how you can do the two pointer traversal:

```
Node * pres = head;
while(pres) {
      prev = pres;
      pres = pres->next;
}
```

4. **Inserting a node after a node *prev*:** Create a new node and manipulate the ***next*** pointers.

```
Node * temp = new Node;
temp->next = prev->next;
prev->next = temp;
```

Important: Always handle corner cases for the above functions!

Stack and Queue

1. **Stack:** Last-In First-Out (LIFO), First-In Last-Out (FILO)
2. **Queue:** First-In First-Out (FIFO), Last-In Last-Out (LILO)
3. **Linked List Implementation**:

	Linked List	Stack (LL)	Queue (LL)
Members	head	head	head, tail
Insertion	Anywhere	Push: At head	Enqueue: At tail
Deletion	Anywhere	Pop: At head	Dequeue: At head

Complexity

1. Time complexity: number of times a set of operations is repeated, parameterized by input size `n`
2. Space complexity: additional space required, parameterized by input size `n`
3. Big-Oh notation: Keep the highest power, ignore constants. E.g.: $O(n^3 + 100n^2 + 1000) = O(n^3)$

$$O(1) < O(\log(n)) < O(n) < O(n\log(n)) < O(n^k) < O(2^n) < O(n!)$$

4. Loops**:** Literally repetitive sections of code.
   a. **O(n)**: `for(int i = 0; i < n; i++) { … }`
   b. **O(mn)**: `for(int i = 0; i < m; i++) { for(int j = 0; j < n; j++) { … } }`

Practice

Array

Recall array doubling, except here we are going to use an extend_factor as well as a twist in the copying mechanism. We will call this function **`extend_array`**, and it has the following prototype:

```
int* extend_array(int* A, int &array_size, int extend_factor);
```

1. Create a new dynamic array **`B`** of size = **`(array_size * extend_factor)`**
2. Then, copy all the elements from **`A`** to **`B`** in an interleaving manner, i.e., copy all odd-index elements contiguously to B followed by all even-index elements.
3. Modify **`array_size`** to the new size
4. Finally, return the pointer **`B`** and (in main) store this returned pointer in **`A`**.

A	12	16	20	24	28
	0	1	2	3	4

`A = extend_array(A, array_size, 2);`

A	16	24	12	20	28					
	0	1	2	3	4	5	6	7	8	9

5. Questions
    a. What is the algorithmic complexity of this function?
    b. If **`extend_array`** is a void function, how will you modify the implementation?
    c. Complete using dereference operator (*) and pointer arithmetic instead of [ ] to index the array.

<u>Linked List</u>

Implement a class called **`LinkedList`**. The class should maintain a single data member (also private), **`head`** which is a pointer to a **`Node`**. Create separate header and source files.

Use the following definition for a Linked List node:

```
struct Node { int id; string name; float gpa; };
```

Implement the following subroutines for this class:

1. **`LinkedList()`**: Initialize head to NULL
2. **`~LinkedList()`**: Delete the entire list (don't just delete head)
3. **`Node* CreateNode(int id, string name, float gpa)`**: Dynamically allocate a new node with the provided arguments and return a pointer to it
4. **`void PrintNode(Node* n)`**: Print passed node n's data members in a single line
5. **`void PrintList()`**: Traverse the list and print each node's details in sequence using **`PrintNode`**
6. **`Node* Tail()`**: Find and return a pointer to the tail of the list; return NULL if list is empty
7. **`Node* Search(string name)`**: Find and return the first node (starting at head) in the list which consists of the same **`name`** data member as the passed argument.
8. **`Node* Search(int ix)`**: Find and return the node at index = **`ix`**.
9. **`int Index(Node* n)`**: Find and return the index of the passed node n. **`head`** has index 0.
10. **`void Filter(float low, float high)`**: Traverse the list & print nodes whose gpa ∈ [low, high]
11. **`Node* CreateLoop(string name)`**: Similar to A4, create a loop in the list, return tail pointer.
12. **`void Insert(Node* p, int ix)`**: Insert node **`p`** at index **`ix`**. If the list is empty, insert as head.
13. **`void Delete(int ix)`**: Delete node at index ix.
14. **`int Length()`**: Compute and return the number of nodes in the list

These notes were prepared by Varad Deshmukh and Sanskar Katiyar.