Cairo University
Faculty of Engineering
CMP102& CMPN102

Spring 2019

Data Structures and Algorithms LAB #1 Revision: Pointers, Arrays, Functions and Classes

Objectives

After this lab, the student should be able to:

- Declare and initialize pointers
- Use pointer-related operators (***&*** and *******)
- Use ***new*** and ***delete*** operators to handle dynamic memory
- Differentiate between automatic and dynamic allocation
- Declare and use arrays
- Declare and use array of objects and arrays of pointers to objects
- Declare and define functions
- Call and pass parameters to functions
- Differentiate between passing by value, reference or pointer
- Pass arrays as function parameters (arrays of objects or arrays of pointers)
- Pass pointers as function parameters (by value, reference or pointer)
- Write classes' special member functions: constructor, destructor, setters and getters
- Use inheritance and polymorphism
- Write copy constructor and assignment operator and explain when each is called.

To Do

- Open "Examples" solution (**Examples.sln**).
- Run the examples in the order mentioned in the "***Lab Outline***" section
- Don't forget to activate any example before running it:
  (right click on project name + set as startup project)
- For each example, make sure to read the comments, answer the questions in them and ask about answers you're not sure of.
- Open "Exercises" solution (**Exercises.sln**).
- Solve the exercises in the order mentioned in the "***Exercises***" section
- **Note:** If you need to read first about any topic, refer to "***Suggested Readings***" section

Lab Outline

1- Pointers

- ❑ Pointer variables vs. regular variables
- ❑ Declaration and Initialization
- ❑ The reference (the address) operator **(&)** and the dereference operator (*****)
- ❑ Dynamic memory allocation (**new**) and de-allocation (***delete***)
- ❑ Difference between deleting and setting to NULL
- ❑ **See Example1 – Pointers**

2- Arrays and Pointers

- ❑ Dynamic and automatic allocation of arrays
- ❑ Dynamic array allocation (**new [..]**) and de-allocation (***delete [ ]***)
- ❑ Making a pointer points to array element and using **[ ]** operator with it
- ❑ **See Example2 – Arrays_and_Pointers**

3- Array of Objects and Array of Pointers

- ❑ Constructor, destructor, setters and getters of classes
- ❑ Default access specifiers in classes and structs
- ❑ Allocation and de-allocation of arrays of objects (automatic and dynamic)
- ❑ Allocation and de-allocation of arrays of pointers to objects (automatic) [TODO] dynamic allocation of array of pointers
- ❑ **See Example3 – Arr_of_Obj_and_of_Ptrs**

4- Functions and Parameter Passing

- ❑ Pass by value vs. pass by reference
- ❑ Two types of pass by reference: pass by alias and pass by pointer
- ❑ Difference in syntax between dealing with pointers and references
- ❑ Alias variables
- ❑ **See Example4 – Functions and Passing (self-reading)**

5- Passing Array and Pointers as Function Parameters

- ❑ Pointers can be passed as parameters
- ❑ Passing an array of integer as a function parameter
- ❑ Passing pointer by value and by reference
- ❑ **See Example5 – Passing Pointers (self-reading)**

6- Passing Arrays of Objects and Arrays of Pointers

- ❑ Passing an array of objects as a function parameter
- ❑ Passing an array of pointers as a function parameter
- ❑ When to pass arrays pointers themselves by reference
- ❑ **See Example6 – Passing Arrays (self-reading)**

7- Inheritance and Polymorphism

- ❑ Reusability and extendibility of inheritance
- ❑ When to make the function non-virtual, virtual or pure virtual
- ❑ Abstract vs. concrete classes
- ❑ How to use dynamic cast and when it is useful
- ❑ **See Example7 – Inheritance and Polymorphism**
  - ➢ Open the (.h and .cpp) files in the following order: ***CFigure, CRect, CSquare, CFigList then the main source file.***

8- Copy Constructors and Assignment Operators

- ❑ When you need to define copy constructor and assignment operator
- ❑ How to define them
- ❑ When each of them is called
- ❑ **See Example8 – Copy Constructors and Assignment Operators**
  - ➢ This example also reviews on character arrays

Suggested Readings:

Don’t waste your time during the lab in reading. Practice!
You can review the following resources before or after the lab. They’re useful.

1. [Optional] Watch these youtube videos:
   - Pointers: http://goo.gl/8FVo4o
   - Arrays: https://goo.gl/uMxLfk
   - 2D Arrays: https://goo.gl/37m22E
   - Passing by Reference vs. Pointers: http://goo.gl/S341oW
   - General C++ Playlist: https://goo.gl/TESQ1Q

2. [Optional] Read:
   - Pointers, References and Dynamic Memory Allocation: https://goo.gl/a1uX6H
     (see only the topics that we talk about)
     – Recommended; It draws memory for clarification
   - General C++ Tutorial: https://goo.gl/Gfh14m
   - Pointers: http://goo.gl/X0S8fm
     (you can discard the last section)
   - Passing by Reference vs. Pointer: http://goo.gl/C7N0MN

3. [Optional] From this textbook, “*Data Abstraction and Problem Solving with C++*”: https://goo.gl/WVCG6h , Read the following chapters:
   - **Appendix A** which reviews C++ up to but NOT including classes.
   - **C++ Interlude 1** which presents C++ classes, template interfaces, inheritance.
   - **C++ Interlude 2** which covers pointers, polymorphism, and dynamic memory allocation.

Practice Exercises

Part 1: Exercises on Pointers Arrays and Functions

Exercise 1 (using Debugger)

1. Open the VS solution "***Exercises.sln***" ➔ "***Exercise1.sln***"
2. Open the file Ex1.cpp
3. Start the **debugger (break point and F5 OR F10)**.
4. Execute the code step by step
5. Answer the questions found in the code to fill the following table

Important Note:

To answer a question, you should first execute all the code preceding that question.

Question	Variables	Values		
1	**x, y, z**			
	&x, &y, &z			
2	**x, y, z**			
	***P, P3**			
3	**P2, P3**			
	***P2, *P3**			
4	**x, y, z**			
5	**P, P[0]**			
	P2, P2[0]			
	P3, P3[0]			

6. Complete the function **CloneArray** that takes an array and its size as parameters. The function then returns a new clone array with the same contents of input array. (**Note**: this function does NOT print anything on the screen.)
7. Complete the main function as instructed by comments

Exercise 2:

Write function "**EvenHalves"** that takes a number greater than 0 and divides it by 2 if the result is even, keep dividing it by 2 until the division result is odd, then stop.
It outputs 3 pieces of information:

a. No. of even halves (use pass by pointer)
b. First odd half (use pass by reference)
c. Return false if the entered number is odd itself and true otherwise (use return statement)

For example, if the entered number is 80, the number of even halves is 3 (40, 20 and 10) and the first odd half is 5. The even halves themselves is not outputted (only their count).
Note: the input number must not change after returning from the function (pass by what?).
It's required to:

- Write the function
- Write a main function that:
    a. Asks the user to enter a number
    b. If the entered number is <= 0, the program stops
    c. Otherwise, calls the function and gives it the entered number
    d. Prints the results of the function to the user
    e. Repeat from step a

Test Cases:

Input	Output		
	No of even halves	**First odd half**	**Entered number is odd?**

80	3	5	True
100	1	25	True
2	0	1	True
1	0	1	False

Exercise 3:

Wite function “**CompressArray”** that takes an array of characters sorted in ascending order, removes duplicates and counts the no. occurrences of each character.
It outputs:

a. A new array that contains the characters after removing duplicates
b. Another array that contains no. of occurrences corresponding to each character of the unique array (array of step a). See the test cases below.
c. The size of the 2 output arrays

Note: the input array must not change.
It’s required to:

- Write the function
- Write a main function that:
    a. Asks the user to enter the number of elements
    b. If it’s <= 0, the program stops
    c. Otherwise, asks the user to enter an array of characters (assume they’re entered sorted)
    d. Calls the function and passes the entered array to it
    e. Prints the 2 output arrays of the function
    f. De-allocates the arrays and repeat from step a

Test Cases:

Input Array	**Output**		
	Unique array	**No of occurrence array**	**The size**
[a, c, c, e, e, e, h, h]	[a, c, e, h]	[1, 2, 3, 2]	4
[e, g, i, p, s, t]	[e, g, i, p, s, t]	[1, 1, 1, 1, 1, 1]	6
[h, h, h, m, m, m, m]	[h, m]	[3, 4]	2
[e, e]	[e]	[2]	1
[h]	[h]	[1]	1

Exercise 4:

Write function “**ExpandArray”** that takes the two output arrays of the above problem and constructs the original input array.
Note: the input array must not change inside the function.
It’s required to:

- write the function
- write a main that tests the function and allows entering many test cases as before (returns from the loop if a number <=0 is entered)

Exercise 5:

Write a function that takes 2 sorted arrays of integers A and B then allocates an array of integers A_minus_B that contains elements found in A and not found in B.
It's required the following:

a. Decide the inputs and output of the function and the passing type of each of them then write function prototype
b. Implement the function
c. Write a main function that reads from the use A and B arrays then call the function and print outside the function A_minus_B returned as an output from the function call.
d. Make the appropriate de-allocations of any dynamically allocated arrays

Exercise 6:

Write function "**EvensAndOdds"** that takes as input an array of pointers to integers and outputs the following:

d. An array of pointers "**evens"** whose elements points to the even integer elements of the original array.
e. An array of pointers "**odds"** whose elements points to the odd integer elements of the original array.
f. Deletes the integers that are <=0 from the original arrays and making their pointers equals NULL.

Note: the original array here may be updated if negative integers occur.
Important Note: the evens and odds arrays *do NOT allocate* new integers but point to the already-allocated integers of the original array.
It's required to:

- Write the function
- Write a main function that:
    a. Asks the user to enter the number of elements
    b. If it's <= 0, the program stops
    c. Otherwise, asks the user to enter an array of integers
    d. Calls the function and passes the entered array to it
    e. Prints the 2 output arrays of the function and the updated original array
    Note: print "NULL" if the element is NULL (need a check before printing).
    f. De-allocates any dynamically-allocated memory and repeat from step a

Test cases:
Note: (*) means pointer to

Input Array	**Output**		
	Evens array	**Odds array**	**Updated input array**
[*1, *-3, *5, *6, *11, * 2, *-10]	[*6, *2]	[*1, *5, *11]	[*1, NULL, *5, *6, *11, * 2, NULL]
[*0, *-3, *5, *11, *-10]	[]	[*5, *11]	[NULL, NULL, *5, *11, NULL]
[*0, *-3, *-10]	[]	[]	[NULL, NULL, NULL]
[*1, *2, *2, *1, *1]	[*2, *2]	[*1, *1, *1]	[*1, *2, *2, *1, *1]
[*2, *4, *6]	[*2, *4, *6]	[]	[*2, *4, *6]

Part 2: Exercises on Classes, Inheritance and Polymorphism

Exercise 1: Bag/Shop

1- Create a class that presents **bag** objects in a shop. For each bag you store:
   a. **Size** (**float**)
   b. **Slots (number of slots in the bag)**
   c. **Price**

2- Member Functions of class **Bag**:
   a. **Constructor**: takes the following 2 parameters with these default values:
      i. size: **15.6**
      ii. number of Slots = **5**
   The constructor should initialize price, to **0**.

   b. **Setters for all members**: If the passed parameter is valid, the data member should be changed and the function should return **true**. Otherwise, the member variable **won't be affected** and the function should return **false**.)

   c. **Getters for all members**

   d. **Compare:** this function should compare the current bag with another bag passed to the function. It returns **1** if the current bag is *bigger* than the passed bag, and returns **-1** if the passed bag is bigger than the current one. If the sizes are equal, the bag with more number of slots is considered bigger. Otherwise, the function returns 0 as when both size and slots are equal.

   e. **PrintInfo**: Print all the details of the bag in the following format.

   f. **ReadInfo**: Reads the information of the bag from the user

3- Create a class **Shop** to represent bags shop with members:
   a. **BagList** (Array of 50 **pointers to Bag**)
   b. **Quantity** (Number of bags available in the shop).
   c. **Total profit** (Total profit for the sold bags)

   d. **Constructor**: to initialize Shop members.

   e. **IsAvailabe**: This function takes bag size and no. of slots returns the index of the first bag with the passed parameters. Otherwise, it returns -1.

   f. **AddBag**: Add a new bag to the BagList if there is a vacant place. Then reads the data of the new bag from the user.

   g. **GetBag**: takes and index and returns a pointer to the bag stored at this index if any. Otherwise, returns NULL. If index is out of range, function returns NULL too.

   h. **Sell:** this function takes bag size and no. of slots and searches for the first bag with the passed parameters. If found, it removes (sells) the bag from the list and adds its price to the total profit.
   [Note]: Call **IsAvailable** function to check the availability of the required bag.

   i. **PrintAllBags**: Prints information of all bags in the shop

4- Write a program (**main function**) that:
   a. Create a Shop object **S**
   b. Add 3 bags to the shop with arbitrary values for their members.
   c. Print the info of all bags.
   d. Declare Bag **Pointer Bp**.
   e. Dynamically allocate a bag using the following constructor parameters: (Size = **15.6**, Number of slots = **7**) and make **Bp** point to it.
   f. Set the price of **Bp** to **130**
   g. Call GetBag and point to the returned Bag by pointer **ptr**.
   h. Compare the object pointed to by **Bp** with the object pointed to by **ptr** and print the comparison result"
   i. Sell the 2nd bag in the shop.
   j. Print the info of all bags.

Exercise 2: Faculty

The problem keeps track of the graduate and undergraduate students of a faculty. The system consists of 4 classes: **Student** class which is the base class of **Graduate** and **Undergraduate** classes. It also includes class **Faculty** that contains a list of students.

1- Create class **Student** that contains:
- ❑ student name and id data members
- ❑ **a non-default constructor** that initializes its data members and validates them
- ❑ **getters** for data members
- ❑ a member function **PrintInfo()** that prints the data members of the student

2- Create class **Graduate** student that contains:
- ❑ grad_year data member which represents the graduation year
- ❑ **a non-default constructor** that initializes its data members and validates them
- ❑ a member function **PrintInfo()** that prints the data members of the graduate student

3- Create class **Undergraduate** student that contains:
- ❑ current_year that represents the faculty year the student currently in (**Assume:** the number of years of this faculty is 4 years)
- ❑ **a non-default constructor** that initializes its data members and validates them
- ❑ **getters** for data members
- ❑ a member function **bool Pass()** that increments the current_year of the student and returns true if he passed his 4th year and graduated, otherwise returns false.
- ❑ a member function **PrintInfo()** that prints the data members of the undergraduate student

4- Create another class, **Faculty**, which contains:
- ❑ an array of 200 **Student pointers**
- ❑ **a default constructor** that makes any needed initializations
- ❑ a member function **AddStudent (Student * pS)** that adds a student to the list
- ❑ a member function **DropStudent (int index)** that takes an array index and drops the student (that pointed to by the pointer of this index) from the array by:
  - o making its pointer points to the last array element and making the pointer of the last element points to NULL then decrementing the elements count of the array.
- ❑ a member function **PassAll()** that:
  - o calls function **Undergraduate::Pass()** for all undergraduate students in the list
  - o if the **Undergraduate::Pass()** of a student returns true (he finished the 4 years of the faculty), the **PassAll()** function should:
    - ▪ drops this student from the list using function **DropStudent**
    - ▪ creates a new **Graduate** student with the same information of the just-

graduated student (***Assume*:** the current year is 2017)
  - adds this graduate student to the list using **AddStudent** function
- ❑ a member function **PrintInfo()** that prints this information of the faculty:
  - o the number of its <u>undergraduate students</u>
  - o the number of its <u>graduate students</u>
  - o the basic information of its <u>graduate</u> students.

5- Write the **main program** to test your classes. You first need to
- ❑ create <u>one</u> object of class **Graduate** student
- ❑ create <u>two</u> objects of class **Undergraduate** student with current_year: 3 and 4
- ❑ creates a **Faculty** object and adds the 3 created students to it using **AddStudent**
- ❑ calls **PrintInfo()** function of the faculty object
- ❑ repeats the following 2 steps <u>3 times:</u>
  - o calls **PassAll()** function of the faculty object
  - o calls **PrintInfo()** function of the faculty object

Exercise 3: Social Media

1. Create an abstract class ***Profile:***

a. Data members:

- ***i.*** Name: **char***
- ***ii.*** N_Followers**: int**
- ***iii.*** N_Following**: int**
- ***iv.*** Followers: **Profile* []**
- ***v.*** Following: **Profile* []**

b. Member functions:

- ***i.*** **Non-default constructor:** sets the name of the profile, and initializes the **Followers** and **Following** fields to 0.
- ***ii.*** **Setters and Getters.**
- ***iii.*** **IsFollow:** takes as a parameter another **Profile** and checks whether the current **Profile** follows the passed profile.
- ***iv.*** **IsFollowed:** takes as a parameter another **Profile** and checks whether the current **Profile** is followed by the passed profile.
- ***v.*** **Follow:** takes as a parameter another **Profile**. If the current profile doesn't follow the passed profile:
  - The current profile should add the passed profile to its list of followers.
  - The current profile should increment its followers.
  - The passed profile should add the current profile to its list of following.
  - The passed profile should increment its following.

  Hint 1: This function should be pure virtual function.
  Hint 2: It's better to define 2 functions: **AddToFollowers** and **AddToFolowing** that take a **Profile** and add it in the appropriate array and increment its counter. Then, function **Follow** should use these two functions inside it. Should these 2 functions be non-virtual, virtual or pure virtual?
- ***vi.*** **Unfollow*:*** takes as a parameter another **Profile**. If the current profile already follows the passed profile, it should do the reverse operation of **Follow**.
  Hint: you can define 2 more functions for removing from the arrays and updating the counters too.
- vii. **PrintInfo:** Prints all information of the profile.

2. Create class **Twitter Profile** that is derived from class **Profile**:

a. Data members:

- i. Tweets : **int**

    ii. isProtected**: bool**

- b. Member functions**:**
    - **i.** **Constructor**: An initial Twitter profile should be **unprotected** and have **zero** tweets.
    - **ii.** **SetProtected**: sets the profile to be protected or unprotected.
    - **iii.** **Setters and Getters**.
    - **iv.** **Follow**: Override **Follow** function in the base class to support the new feature that protected profiles should be notified first. It should print a message to the user that "You have a new follow request from *profile X, Do you accept it*?"
        - If they **accept** the request, the profile should be added.
        - If they **reject** the request, the profile shouldn't be added and nothing changes.

      Note: Only Twitter profiles can follow Twitter profiles. An error message should be printed otherwise.
    - **v.** **PrintInfo**: Override **printInfo** function in the base class to print the new information.

3. Create class **LinkedIn Profile** that is derived from class **Profile:**
    - **a.** Extra data members:
        - i. University: **char***
        - ii. Work**: char***
    - ***b.*** Member functions:
        - **i.** **Constructor with default parameters:** sets the name of the university and the company of the account holder
        - **ii.** **Setters and Getters.**
        - **iii.** **Follow**: Override **Follow** function in the base class to support the new feature that LinkedIn profiles can follow each other **ONLY** if they were in the same university or if they work at the same company or both.

          Note: Only LinkedIn profiles can follow LinkedIn profiles. An error message should be printed otherwise.
        - **iv.** **PrintInfo**: Override **printInfo** function in the base class to print the new information.

4. Write a main function that does the following:
    - ***a.*** Declare six pointers to **Profile** and dynamically allocate them as follows:
        - **3 LinkedIn** profiles with fields:
            - "**Ahmed**" studied in **Cairo University** and works in **IBM.**
            - **"Mai"** is currently studying at **Cairo University**.
            - **"Omar"** studied in **Alexandria University** and works in **Microsoft**.
        - **3 Twitter** profiles with profile names "**Adam**", "**Kareem**", and "**Dina**".
    - b. **Ahmed** follows **Mai** and **Omar**.
    - c. Print the information of **Ahmed**, **Mai** and **Omar**.
    - d. **Omar** follows **Mai**.
    - e. Print the information of **Ahmed**, **Mai** and **Omar**.
    - f. **Kareem** sets his profile protected.
    - g. **Kareem** follows **Adam** and **Dina.**
    - h. **Adam** follows back **Kareem.**
    - i. Print the information of **Adam, Kareem** and **Dina.**
    - j. **Kareem** unprotects his profile.
    - k. **Dina** follows back **Kareem**.
    - l. Print the information of **Adam, Kareem** and **Dina.**
    - m. **Adam** unfollows **Kareem**.
    - n. Check whether **Adam** follows each of **Dina** and **Kareem.**
    - o. **Adam** follows **Omar**
    - p. Print the information of all profiles.