selection sort code searches an array looking for the smallest element in the array. Then, the

smallest element is swapped with the first element of the array. The process is repeated for the

sub-array beginning with the second element of the array. Each pass of the array results in one

element being placed in its proper location. When the sub-array being processed contains one

element, the array is sorted. Write C++ code for this selection sort and output must be stored in

“output.txt” file (having all passes

selection sort code searches an array looking for the smallest element in the array. Then, the

smallest element is swapped with the first element of the array. The process is repeated for the

sub-array beginning with the second element of the array. Each pass of the array results in one

element being placed in its proper location. When the sub-array being processed contains one

element, the array is sorted. Write C++ code for this selection sort and output must be stored in

“output.txt” file (having all passes

# Department of Computing

# School of Electrical Engineering and Computer Science

**CS-250: Data Structure and Algorithms**

**Class: BS-EE 13**

# 

# Lab 2: DMA in Pointers

**Date: 5th February, 2025**

**Time: 02:00 PM – 04:50 PM**

# Instructor: Ms.Ayesha Sarwer

# Lab Engineer: Areeba Rameen

**Lab 2: Dynamic Memory Management**

**Introduction**

In this lab, we will explore two ways of allocating memory for arrays in C++: static arrays and dynamic arrays. Static arrays have a fixed size at compile time, while dynamic arrays can be resized at runtime using dynamic memory allocation. Understanding these concepts is essential for effective memory management and efficient program design.

**Objectives**

This lab will revise the old concepts taught to the students in the previous semesters.

**Tools/Software Requirement**

Visual Studio C++

**Static Arrays (Method A):**

A static array is created with a fixed size that is determined at compile time. Memory for static arrays is allocated on the activation stack, and its lifetime depends on its scope:

**Method A:**

const int size=5;

int x[size];

for (int i = 0; i < size; i++)

{

x[i] = i + 1;

cout << "x[" << i << "] = "<< x[i] << endl;

}

**Dynamic Arrays (Method A):**

A dynamic array is created using the new operator at runtime, allowing you to specify its size during execution.

**Method B**

int size; // Note that size variable is const in variant A whereas it isn’t in variant B. Find out the logic behind it.

cout << "Enter size of array: ";

cin >> size;

int \*x = new int[size];

for (int i = 0; i < size; i++)

{

x[i] = i + 1;

cout << "x[" << i << "] = "<< x[i] << endl;

}

**TASK 1:**

**Create a C++ program that asks the user to input the size of an array, dynamically allocates memory for the array, fills it with values from 1 to N (where N is the user-specified size), calculates and prints the sum and average of the array elements, and finally frees the allocated memory once it's no longer needed.**

**Debugging Pointers**

**TASK 2:**

**Code**

#include <iostream>

using namespace std;

int main() {

int\* ptr = nullptr; // Pointer initialized to nullptr

// Try to dereference the pointer

cout << "Value pointed by ptr: " << \*ptr << endl; // This will cause a crash

return 0;

}

**Questions:**

1. What happens when you try to dereference a null pointer, and why does the program crash?
2. Modify the code to safely handle the null pointer and avoid the crash. What check would you add?
3. What is the role of a nullptr in pointer safety, and why is it important to initialize pointers with it?

**OOP Concept**

**Task 3: Banking Management System**

Develop a C++ program to model a Banking System with the following requirements:

1. Implement a base class `Account` that contains An `accountNumber` (int), A `balance` (double) and customerName` (string)OwnerName (string).

Implement methods:

* A constructor to initialize these fields.
* A pure virtual method `deposit(double amount)` to deposit money into the account.
* A pure virtual method `withdraw(double amount)` to withdraw money.
* A method `displayInfo()` to display account details.

1. Create three derived classes:

* **SavingsAccount:**
* Has an `interestRate` (double) as an additional attribute.
* Override the `deposit()` method to calculate the interest.
* Implement `withdraw()` such that no withdrawal is allowed if the balance falls below a certain minimum.
* **CheckingAccount:**
* Has an `overdraftLimit` (double).
* Implement `withdraw()` such that withdrawals beyond the balance are allowed up to the overdraft limit.
* **InvestmentAccount:**
* Has an `investmentDuration` (int) and `riskFactor` (double).
* Override `deposit()` and `withdraw()` to handle risk-based profit/loss when depositing or withdrawing.

1. Implement \*\*operator overloading\*\* to compare two accounts based on their balances (`>`, `<`, `==`).
2. In the \*\*main\*\* function:
   * Create objects for all three types of accounts.
   * Perform deposit and withdraw operations.
   * Compare the accounts using the overloaded operators.
   * Use polymorphism to manage the different types of accounts through base class pointers.

* Ensure proper memory management using virtual destructors.

**Deliverables**

Compile a single word document by filling in the solution part and submit this Word file on LMS. The name of word document should follow this format. i.e. **YourFullName(reg)\_Lab#.** You must show the implementation of the tasks in a complete Word document to get your work graded. You must also submit this Word document on the LMS.

**Note:** Students are required to upload the lab on LMS before deadline.

Use proper indentation and comments. Lack of comments and indentation will result in deduction of marks.

**Solution:**

|  |
| --- |
| Solution |
| Task 1 Code Screenshots (full window screen):  Task 1 Output Screenshots (full window screen):  Task 2 Code Screenshots (full window screen):  Task 2 Output Screenshots (full window screen): |