**Data Structures Lab 01**

**Course:** Data Structures (CL2001) **Semester:** Fall 2023

**Instructor:** Taha Ahmed  **T.A:** N/A

**Note:**

* Maintain discipline during the lab.
* Listen and follow the instructions as they are given.
* Just raise your hand if you have any problem.
* Completing all tasks of each lab is compulsory.
* Get your lab checked at the end of the session.

**Revision of Previous Programming Concepts:**

# Debugging

*Using the debugger:*

* The various features of the debugger are obvious. Click the "Debug" icon to run your program and pause at the current source code cursor location; Click "Next Line" to step through the code; Click "Add Watch" to monitor variables.
* Setting breakpoints is as easy as clicking in the blank space (Line Number) next to the line in the source code.

# → Dynamic Memory

C++ supports three types of memory allocation.

* **Static memory allocation** happens for static and global variables. Memory for these types of variables is allocated once when your program is run and persists throughout the life of your program.
* **Automatic memory allocation** happens for function parameters and local variables. Memory for these types of variables is allocated when the relevant block is entered, and freed when the block is exited, as many times as necessary.
* **Dynamic memory allocation** is a way for running programs to request memory from the operating system when needed.

## **new** Operator

* This operator is used to allocate a memory of a particular type.
* This creates an object using the memory and **returns a pointer** containing the memory address.
* The return value is mostly stored in a **pointer** variable.

| // new\_op.cpp  int main()  {  int \*ptr = new int; // allocate memory  \*ptr = 7; // assign value  // allocated memory and assign value  int \*ptr2 = new int(5);  } |
| --- |

## **delete** Operator

* When we allocate memory dynamically, we need to explicitly tell C++ to deallocate this memory.
* **delete** Operator is used to release / deallocate the memory.

| // delete\_op.cpp    #include <iostream>  int main()  {  int \*ptr = new int; // dynamically allocate an integer  int \*otherPtr = ptr; // otherPtr is now pointed at that same memory  delete ptr; // ptr and otherPtr are now dangling pointers.  ptr = 0; // ptr is now a nullptr    // however, otherPtr is still a dangling pointer!  return 0; } |
| --- |

# → Dynamic Arrays

To allocate an array dynamically we use array form of **new** and **delete**

(new[ ] , delete[ ])

| dynamic\_array.cpp  // dynamic\_array.cpp  #include<iostream>  using namespace std;  int main()  {  int array[] = {1,2,3};  cout << array[0];  cout << endl;    // int\* dArray = new int[] {1,2,3};  int\* dArray = new int[3] {1,2,3};  cout << \*dArray+1;  cout << endl;  cout << dArray[2];    delete[] dArray;  } |
| --- |

**1D & 2D Array:**

A **one-dimensional** array (or single dimension array) is a type of linear array. Accessing its elements involves a single subscript which can either represent a row or column index.

**Syntax:**

***char name[5];***

***int mark[5] = {5,11,14,65,85};***

***int mark[] = {5,11,14,65,85};***

Like a 1D array, a **2D array** is a collection of data cells, all of the same type, which can be given a single name. However, a 2D array is organized as a matrix with a number of rows and columns.

**Syntax:**

***float x[3][4];***

***int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};***

***int c[][3] = {{1, 3, 0}, {-1, 5, 9}};***

***int c[2][3] = {1, 3, 0, -1, 5, 9};***

**Dynamic Memory Allocation for arrays:**

Memory in your C++ program is divided into two parts

1. The **stack −** All variables declared inside the function will take up memory from the stack.
2. The **heap −** this is unused memory of the program and can be used to allocate the memory dynamically when the program runs.

A **dynamic array** is an array with a big improvement: **automatic resizing**.

One limitation of arrays is that they're fixed size, meaning you need to specify the number of elements your array will hold ahead of time.

A dynamic array expands as you add more elements. So you don't need to determine the size ahead of time.

Strengths:

1. **Fast lookups**. Just like arrays, retrieving the element at a given index takes O (1) time.
2. **Variable size.** You can add as many items as you want, and the dynamic array will expand to hold them.
3. **Cache-friendly.** Just like arrays, dynamic arrays place items right next to each other in memory, making efficient use of caches.

Weaknesses:

1. **Slow worst-case appends.** Usually, adding a new element at the end of the dynamic array takes **O (1)** time. But if the dynamic array doesn't have any room for the new item, it'll need to expand, which takes **O(n)** time.
2. **Costly inserts and deletes.** Just like arrays, elements are stored adjacent to each other. So adding or removing an item in the middle of the array requires "scooting over" other elements, which takes **O(n)** time.

**Factors impacting performance of Dynamic Arrays:**

The array's initial size and its growth factor determine its performance. Note the following points:

1. If an array has a **small size** and a **small growth factor**, it will keep on **reallocating** memory more often. This will **reduce** the performance of the array.
2. If an array has a **large size** and a **large growth facto**r, it will have a **huge chunk** of **unused** memory. Due to this, resize operations may take longer. This will reduce the performance of the array.

**The new Keyword:**

In C++, we can create a dynamic array using the **new keyword**. The number of items to be allocated is specified within a pair of square brackets. The type name should precede this. The requested number of items will be allocated.

**Syntax:**

***int \*ptr1 = new int;***

***int \*ptr1 = new int[5];***

***int \*array { new int[10]{}};***

***int \*array { new int[10]{1,2,3,4,5,6,7,8,9,10}};***

**Resizing Arrays:**

The length of a dynamic array is set during the allocation time. However, C++ doesn't have a built-in mechanism of resizing an array once it has been allocated. You can, however, overcome this challenge by allocating a new array dynamically, copying over the elements, then erasing the old array.

**Dynamically Deleting Arrays:**

A dynamic array should be deleted from the computer memory once its purpose is fulfilled. The delete statement can help you accomplish this. The released memory space can then be used to hold another set of data. However, even if you do not delete the dynamic array from the computer memory, it will be deleted automatically once the program terminates.

**Syntax:**

***delete ptr;***

***delete[] array;***

**NOTE:** To delete a dynamic array from the computer memory, you should use delete[], instead of delete. The [] instructs the CPU to delete multiple variables rather than one variable. The use of delete instead of delete[] when dealing with a dynamic array may result in problems. Examples of such problems include **memory leaks, data corruption, crashes,** etc.

**Example:**

**Single Dimensional Array:**

***#include <iostream>***

***using namespace std;***

***main(){***

***int\* darray = new int[3] {1,2,3}; //Initializing a dynamic array***

***cout << \*darray+1 << endl;***

***cout << darray[2];***

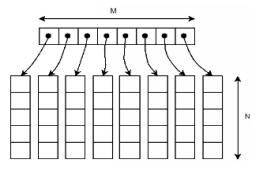
***delete[] darray; //Deleting the dynamic array to save memory space***

***//cout <<darray[2] << endl; If we try to print the array we would get random values***

***}***

**Two Dimensional Array Using Array of Pointers:**

We can dynamically create an array of pointers of size M and then dynamically allocate memory of size N for each row as shown below.



**Example:**

***// Dynamically Allocate Memory for 2D Array in C++***

***int main(){***

***int\* A = new int\*[M]; // dynamically create array of pointers of size M***

***srand (time(NULL)); /\* initialize random seed: \*/***

***for (int i = 0; i < M; i++) // dynamically allocate memory of size N for each row***

***A[i] = new int[N]; // assign values to allocated memory***

***cout << A[i]; // print the 2D array***

***for (int i = 0; i < M; i++) // deallocate memory using delete[] operator***

***delete[] A[i];***

***delete[] A;***

***return 0;***

***}***

**Safe Array:**

In C++, there is **no check** to determine whether the **array index** is **out of bounds**.

During program execution, an out-of-bound array index can cause **serious problems**. Also, recall that

in C++ the array index starts at 0.

Safe array solves the out-of-bound array index problem and allows the user to begin the array index starting at any integer, positive or negative.

**"Safely"** in this context would mean that access to the array elements must not be **out of range**. i.e. the position of the element must be **validated** prior to access.

For example, in the member function to allow the user to set a value of the array at a particular location:

***void set(int pos, Element val){ //set method***

***if (pos<0 || pos>=size){ //this line can also be written as (pos<0 or pos>=size)***

***cout<<"Boundary Error\n";***

***}***

***else{***

***Array[pos] = val;***

***}***

***}***

Jagged Array:

**Jagged array** is similar to an array but the difference is that it's **an array of arrays** in which the member arrays can be in different sizes.

Table

Description automatically generated with medium confidence

**Example:**

***int \*arr = new int\*[3];***

***int Size[3];***

***int i,j,k;***

***for(i=0;i<3;i++){***

***cout<<"Row "<<i+1<< " size: ";***

***cin>>Size[i];***

***arr[i] =new int[Size[i]];***

***}***

***for(i=0;i<3;i++){***

***for(j=0;j<Size[i];j++){***

***cout<<"Enter row " <<i+1<<" elements: ";***

***cin>>\*(\*(arr + i) + j);***

***}***

***}***

***// print the array elements using loops***

***// deallocate memory using delete[] operator as mentioned in the previous example***

**Lab Tasks**

Q1. Implement a 1D Dynamic Array in C++. Create a class named DynamicArray and implement the following features through constructors or member functions. Take input from the user for how many elements should be in the array. Then allocate the space accordingly, take input of elements and show the output after storage.

Q2. Implement a 2D Dynamic Array in C++. Create a class named DynamicArray and implement the following features through constructors or member functions. Take input from the user for how many rows and columns should be in the 2D array. Then allocate the space accordingly, take input of elements and show the output after storage.

Q3. Implement a 2D Dynamic Safe Array in C++. Create a class named DynamicSafeArray and implement the following features through constructors or member functions. Same as Q2, except you must implement an out of bounds check to make it a safe array.  
  
Q4. Implement a 2D Dynamic Jagged Array in C++. Create a class named DynamicJaggedArray and implement the following features through constructors or member functions. Same as Q2, except the columns in each row can be different.