**National University of Computer and Emerging Sciences** CS-2001 Data Structure

### Objective: Implementing a DynamicSafeArray with one/two dimensional pointers.

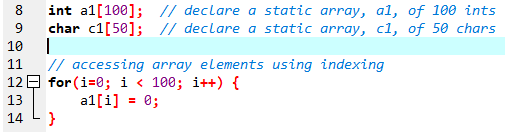
### This lab covers how dynamic memory is used to implement 1D and 2D arrays which are more powerful as compared to the default array mechanism of the programming language C/C++ supports. .

### Description:

### 1D-Array:

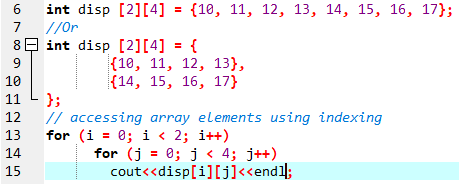
A **one-dimensional** array (or single dimension array) is a type of linear array. Accessing its elements involves a single subscript which can either represent a row or column index.

**Example:**



2D-Array:

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. **Example:**

****

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 quite similar to a regular array, but its size is modifiable during program **runtime**. DynamArray elements occupy a **contiguous block of memory**.

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

Once an array has been created, its size cannot be changed. However, a dynamic array is different. A dynamic array can expand its size even after it has been filled.

During the creation of an array, it is allocated a predetermined amount of memory. This is not the case with a dynamic array as it grows its memory size by a certain factor when there is a need.

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:

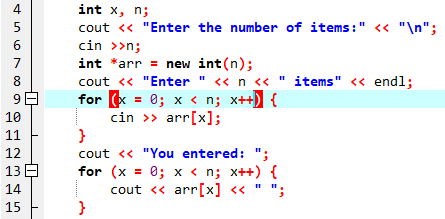
The array's initial size and its growth factor determine its performance. Note the following points:

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.

**Example:**

****

**NOTE:** In the above example, the user is allowed to specify any size for the array during run time. This means the array's size is determined during runtime.

Initializing dynamically allocated arrays:

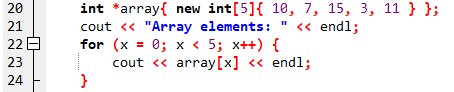
It's easy to initialize a dynamic array to 0.

**Syntax:**

****

In the above syntax, the length denotes the number of elements to be added to the array. Since we need to initialize the array to 0, this should be left empty.

**Example:**



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.

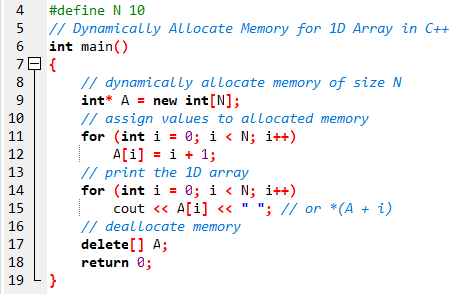
**Example:**

****

**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.

1. Single Dimensional Array:

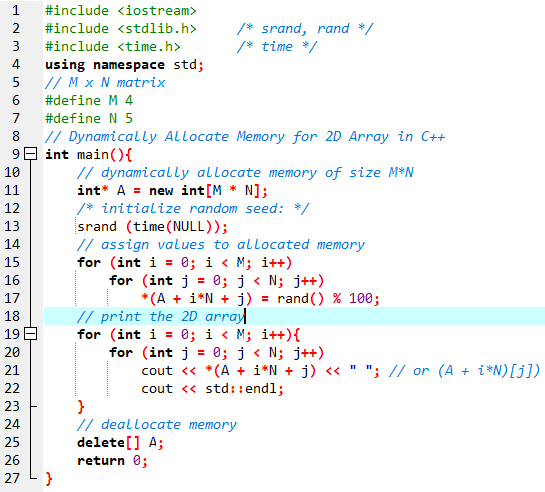
**Example:**

****

1. Two Dimensional Array Using Single Pointer:

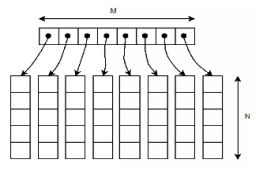
In this approach, we simply allocate one large block of memory of size M\*N dynamically and assign it to the pointer. Then we can use pointer arithmetic to index 2D arrays.

**Example:**

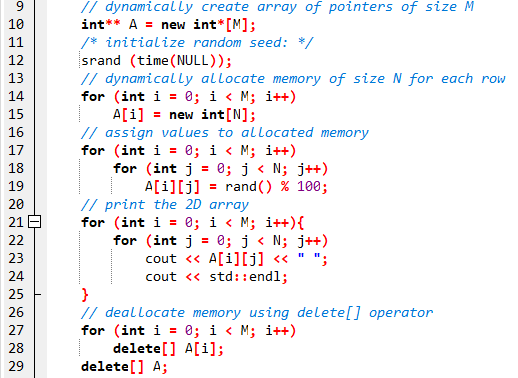


3. Two Dimensional Array Using Array of Pointers:

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:**



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**. ie. 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:

Let’s code a 1D and 2D array:

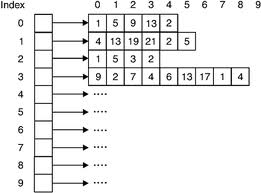
We will see the Bayesian Networks implementation through a notebook file present in google drive named as **“Week02\_1D\_safe\_array.cpp”** and **“Week02\_2D\_safe\_array.cpp”**.

Jagged Array:

A **jagged array**, is **an array of arrays** of which the **member arrays** can be of **different lengths** producing rows of jagged edges when visualized as output. In contrast, two-dimensional arrays are always rectangular so jagged arrays should not be confused with multidimensional arrays, but the former is often used to emulate the latter.

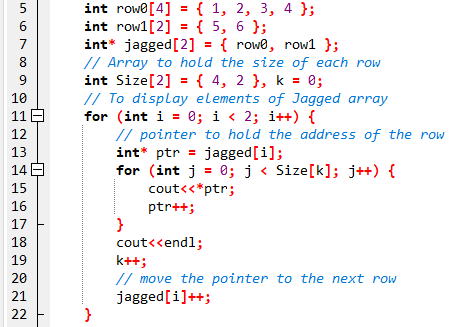
OR

Jagged array is nothing but it is an array of arrays in which the member arrays can be in different sizes.



1. Static Jagged Array(Using array and a pointer ):
2. First declare 1-D arrays with the number of rows you will need,
3. The size of each array (array for the elements in the row) will be the number of columns (or elements) in the row,
4. Then declare a 1-D array of pointers that will hold the addresses of the rows,
5. The size of the 1-D array is the number of rows you want in the jagged array.

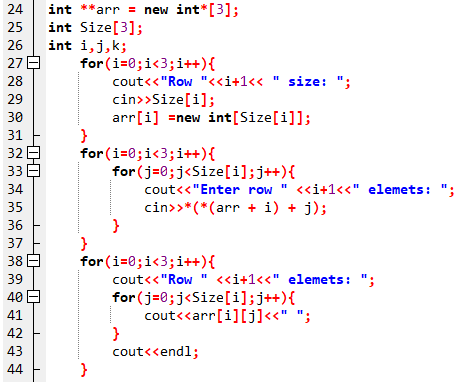
**Example:**



2. Dynamic Jagged Array(Using an array of pointer):

1. Declare an array of pointers (jagged array),
2. The size of this array will be the number of rows required in the Jagged array
3. Then for each pointer in the array allocate memory for the number of elements you want in this row.

**Example:**

****