Array Representation

This C++ file is an excellent practical guide to understanding **arrays** and how they relate to **memory management**. It's structured to build knowledge from the ground up, starting with basic array declarations and moving to more complex, low-level concepts involving pointers and dynamic memory.

The file begins by exploring the various ways to declare and access one-dimensional arrays, highlighting the deep connection between array indexing and pointer arithmetic. It then clearly contrasts arrays created on the **stack** (automatic memory) with dynamic arrays created on the **heap** (manual memory using malloc and new). A key practical lesson is shown in the "Increasing the size of a dynamic array" section, which walks through the essential technique of creating a larger memory block, copying data, and reassigning pointers.

The final and most detailed section is dedicated to **2D arrays**. It brilliantly showcases three distinct methods for creating them:

1. A simple, single block of memory on the stack.
2. An array of pointers on the stack, with each pointer leading to a separate row on the heap.
3. A fully dynamic approach using a double pointer, where both the row pointers and the rows themselves reside on the heap.

This progression provides a crystal-clear picture of the trade-offs between simplicity, flexibility, and manual memory management. Overall, this file is a perfect resource for anyone wanting to master how arrays truly work "under the hood" in C and C++.

**Different ways to declare and access an array**

This section covers the various syntaxes for declaring and initializing 1D arrays. It also cleverly demonstrates that standard array indexing is just a more readable version of pointer arithmetic.

* int c[] = {1, 2, 3, 4, 5}; This shows that you can let the compiler **automatically determine the size** of the array by providing an initializer list. Here, c will be created with a size of 5.
* printf("%d \n", 2[b]); This is an unusual but completely valid syntax. In C++, the expression b[2] is interpreted by the compiler as \*(b + 2). Because addition is commutative, \*(2 + b) is also valid, which leads to the 2[b] syntax. It's a great example of how arrays are really just pointers.
* printf("%d \n", \*(b+2)); This is the explicit **pointer arithmetic** way to access an element. It tells the program to go to the starting address of the array b, move forward by 2 element-sizes, and then dereference (\*) that memory location to get the value.

**Stack and Dynamic Array**

This code snippet provides a direct comparison between a standard array allocated on the **stack** and a dynamic array allocated on the **heap**. It highlights the fundamental difference in how they are created and managed.

* int A[5] = {2, 4, 6, 8, 10}; This creates an array on the **stack**. Memory for this array is allocated and deallocated automatically by the program when the function starts and ends.
* p = (int \*)malloc(5\*sizeof(int)); This creates a dynamic array on the **heap**. malloc (memory allocation) requests a block of memory large enough for 5 integers. This memory persists until it is explicitly freed.
* free(p); This is a crucial step for heap-allocated memory. The free() function **releases the memory** that was previously allocated. Forgetting to do this causes a **memory leak**.

**Increasing the size of dynamic array**

This section demonstrates the standard method for "resizing" a dynamic array. Since an array's size is fixed once allocated, the only way to expand it is to create a new, larger array and copy the contents over.

* int \*p = new int[5]; int \*q = new int[10]; Two dynamic arrays are created. p is the original small array, and q is the new, larger array that we want to use.
* for(int i=0; i<5; i++){ q[i] = p[i]; } This loop **copies all the elements** from the old array (p) into the beginning of the new, larger array (q).
* delete []p; p=q; q = NULL; This is the critical cleanup and reassignment phase:
  1. delete []p;: The memory used by the original small array is **released** to prevent a memory leak.
  2. p=q;: The pointer p is now made to **point to the new, larger array**.
  3. q = NULL;: The pointer q is set to NULL so it no longer points to the memory block (which p now controls). This prevents issues like trying to delete the same memory twice.

**2D array (Method 1)**

This is the simplest and most common way to create a 2D array. A single, contiguous block of memory is allocated on the **stack**.

* int A[3][4] = { ... }; This declares a 2D array named A with 3 rows and 4 columns. All 12 integer values are stored one after another in a single block of memory, making it very cache-friendly and efficient.

**2D array (Method 2)**

This method creates a 2D array where the rows are stored on the heap. It uses an array of pointers on the stack, where each pointer is responsible for a different row.

* int \*p[3]; This declares an **array of 3 integer pointers** on the stack. At this point, no memory has been allocated for the actual integer data yet.
* p[0] = (int \*)malloc(4\*sizeof(int)); Here, memory for the first row (4 integers) is allocated on the **heap**, and the address of this memory block is stored in the first pointer, p[0]. This step is repeated for each row.

**2D array (Method 3)**

This method creates a fully dynamic 2D array where everything, including the array of pointers, is stored on the **heap**. This offers the most flexibility, as even the number of rows can be decided at runtime.

* int \*\*p; A **double pointer** p is created on the stack. This pointer will point to an array of pointers.
* p = (int \*\*)malloc(3\*sizeof(int \*)); Memory is allocated on the **heap** for an array of 3 integer pointers. The double pointer p now points to the beginning of this pointer array.
* p[0] = (int \*)malloc(4\*sizeof(int)); For each pointer in the heap-allocated pointer array (p[0], p[1], p[2]), we now allocate another block of memory on the **heap** for the actual integer data of that row.