30 - Arrays in C++

quarta-feira, 19 de fevereiro de 2025 07:42

Arrays in C++

- cOLECTION OF ELEMENTS IN A SPECIFIC ORDER, USUALLY OF THE SAME TYPE
- Elementos of the same type grouped in one variable
- Arrays do well with loops, because it's easier to go all over everything
- Alocate memory sequentially
- It has fixed memory, can't increase even if we want to

```cpp
static const int exampleSieOfTheArray30 = 5;
int exemple30[exampleSieOfTheArray30]; // Definition of an array of 5 integers // Gets destroied wen getting out of the scope
int* ptr30 = exemple30; // creating ap ointer to the first element of the array
// If allocated in the stack, we can get the size with the sizeof() method. But we need to convert
    // It's actually the size of the aray with the specific data type
    // The convertion to find the number of elemnts is
    // better way is to keep the size by our selfs
// static const int exampleSieOfTheArray30 = 5;
int count = sizeof(exemple30) / sizeof(int);
std::cout << "Size of the array : " << count << std::endl;

// Creating array with the keyword new
    // This one is created on the heap, we need to manualy delete it
    // Be arround until we delete it... can be good... and usefull
    // we holds a pointer
    // If... we anocate the entire array, it'll be alive after we move out of the scope
    // but if we allocate just the pointer to the array, it'll no be thete ( int* )
        // However, we have the current adress to the current data, so we don'tactually delete it
        // To access we need to jump arround to find the data
        // It's a performance issue, don't do it
    // There is no way to know the size of the array, like get_size() method for example
int* example30_2 = new int[5];

// c++ 11 array example ( standard array)
// Advantages on tools to use and security on acessing memory
// been super safe, use it.
std::array<int, exampleSieOfTheArray30> example30_3;


//To access an array - index starts with 0
exemple30[0] = 10; // This is any other integer and we can set it
exemple30[4] = 14; // lest element of the array
exemple30[3] = 3;
// we dereference the pointer to the array and increase 3 positions
// same as acessing the pointer using the index
*(ptr30 + 3) = 4;

std::cout << exemple30[0] << std::endl;
std::cout << exemple30[4] << std::endl;
std::cout << exemple30[3] << std::endl; // suppose to be 4 and not 3 because we changed using the pointer to the beginning of the array
// This is bad... can cause problems difficult to debug
// std::cout << example30[-1] << std::endl; // Memory access violation
// std::cout << example30[5] << std::endl; // Memory access violation

// loop with arrays - very usefull
for (int i=0; i<5; i++)
{
    exemple30[i] = 2;
    example30_2[i] = 3;
    example30_3[i] = 4;
}

std::cout << exemple30[0] << std::endl;
std::cout << exemple30[4] << std::endl;

std::cout << example30_2[0] << std::endl;
std::cout << example30_2[4] << std::endl;

std::cout << example30_3[0] << std::endl;
std::cout << example30_3[4] << std::endl;

delete[] example30_2; // delete an array created with new
```