Arrays

quarta-feira, 16 de outubro de 2024 13:47

→ Sequence of objects of the same type that occupies a contiguous area of memory

↳ Usually the source for many bugs C-style

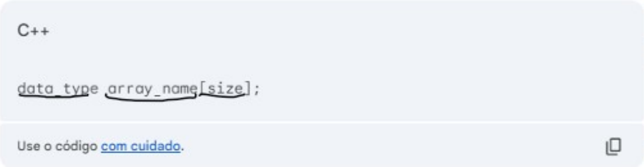
↳ Use std::vector or array instead → Safer

C-style → Bad Practice

Declaration and Initialization:

→ Declaration is simple and easy to understand

- **Declaration:**

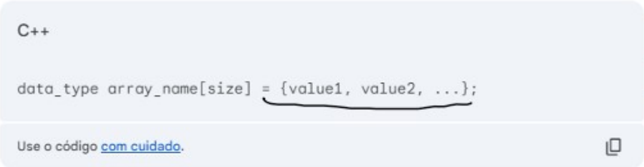


```
C++

data_type array_name[size];
```

Use o código com cuidado.

- `data_type`: The type of elements the array will store (e.g., `int`, `double`, `char`).
- `array_name`: The name of the array.
- `size`: The number of elements the array can hold.

- **Initialization:**



```
C++

data_type array_name[size] = {value1, value2, ...};
```

Use o código com cuidado.

HOWEVER

↳ Not memory safe, meaning you can accidently access memory outside the array's bound !!

↳ Array // Vector has more conventional features than the C-style

Methods like:

size() // data()

VECTOR

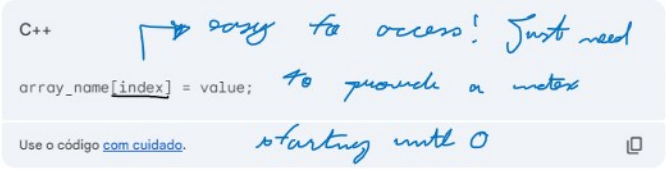
↳ Can resize itself

push_back() begin()
pop_back() end()

↳ Can prevent memory leaks

Accessing Elements:

- Use the index operator `[]` to access individual elements. The index starts from 0.



```
C++

array_name[index] = value;
```

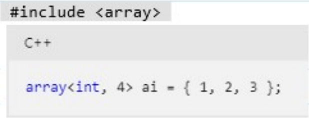
Use o código com cuidado.

→ easy to access! Just need to provide a index starting until 0

Array Class

array (Ty, N)

↳ Control a sequence of N elements of type (Ty)



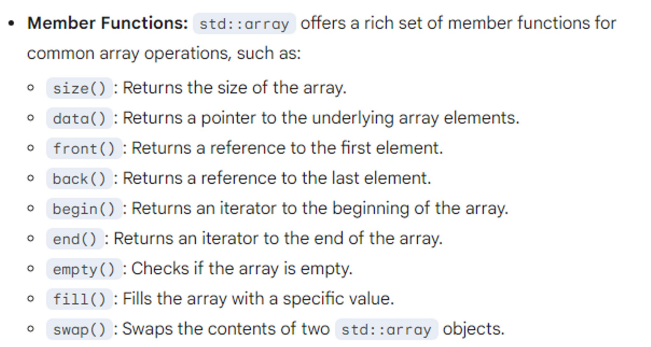
```
#include <array>
C++

array<int, 4> ai = { 1, 2, 3 };
```

creates the object ai that holds four integer values, initializes the first three elements to the values 1, 2, and 3, respectively, and initializes the fourth element to 0.

FIXED SIZE → At compiler time → Can't change it

→ Use them !!!

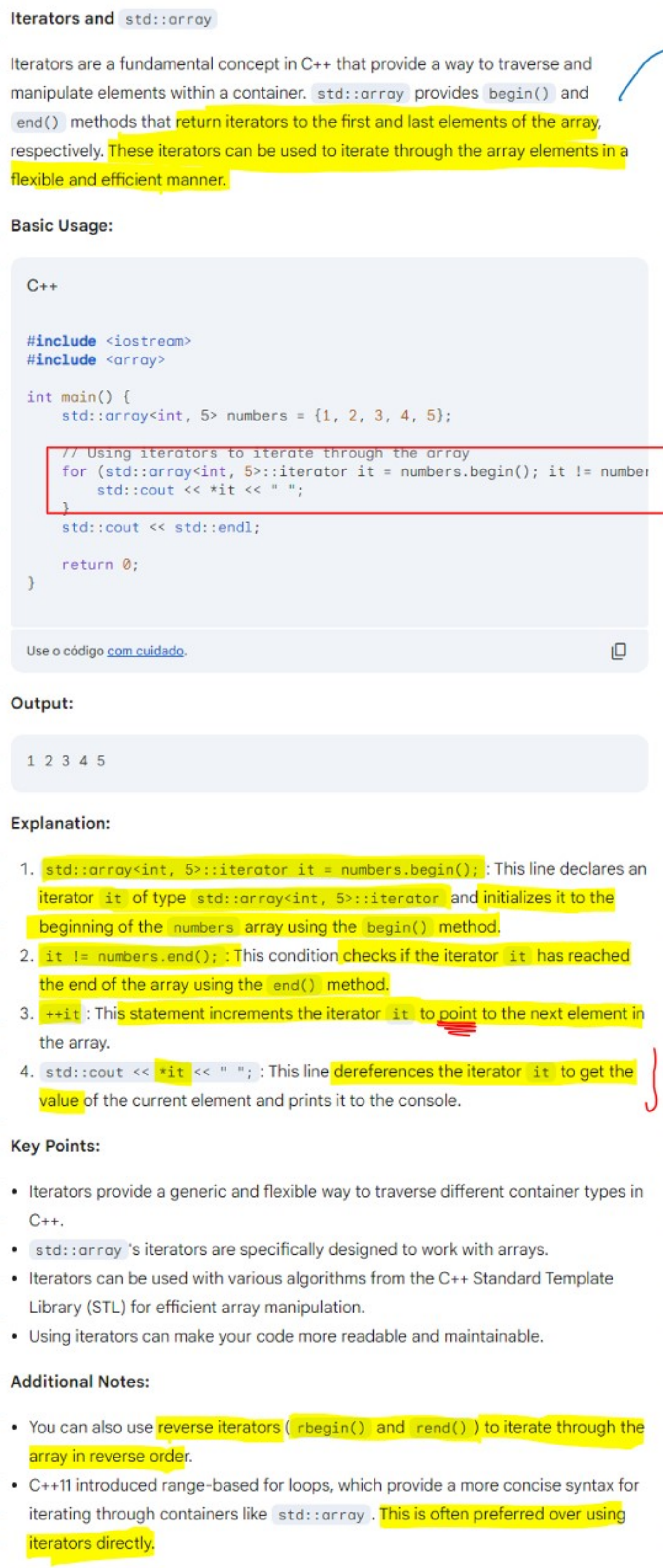
- **Member Functions:** `std::array` offers a rich set of member functions for common array operations, such as:
  - `size()`: Returns the size of the array.
  - `data()`: Returns a pointer to the underlying array elements.



- **Member Functions:** `std::array` offers a rich set of member functions for common array operations, such as:
    - `size()` : Returns the size of the array.
    - `data()` : Returns a pointer to the underlying array elements.
    - `front()` : Returns a reference to the first element.
    - `back()` : Returns a reference to the last element.
    - `begin()` : Returns an iterator to the beginning of the array.
    - `end()` : Returns an iterator to the end of the array.
    - `empty()` : Checks if the array is empty.
    - `fill()` : Fills the array with a specific value.
    - `swap()` : Swaps the contents of two `std::array` objects.

← use them !!!

↳ One of the big problems here is to insert an array
here is some explanation



Iterators and `std::array`

Iterators are a fundamental concept in C++ that provide a way to traverse and manipulate elements within a container. `std::array` provides `begin()` and `end()` methods that return iterators to the first and last elements of the array, respectively. These iterators can be used to iterate through the array elements in a flexible and efficient manner.

Safer than passing index within a loop

Basic Usage:

C++

```cpp
#include <iostream>
#include <array>

int main() {
    std::array<int, 5> numbers = {1, 2, 3, 4, 5};

    // Using iterators to iterate through the array
    for (std::array<int, 5>::iterator it = numbers.begin(); it != numbe
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Use o código com cuidado.

Output:

```
1 2 3 4 5
```

Explanation:

1. `std::array<int, 5>::iterator it = numbers.begin();` : This line declares an iterator `it` of type `std::array<int, 5>::iterator` and initializes it to the beginning of the `numbers` array using the `begin()` method.
2. `it != numbers.end();` : This condition checks if the iterator `it` has reached the end of the array using the `end()` method.
3. `++it` : This statement increments the iterator `it` to point to the next element in the array.
4. `std::cout << *it << " ";` : This line dereferences the iterator `it` to get the value of the current element and prints it to the console.

It's a pointer

Key Points:

- Iterators provide a generic and flexible way to traverse different container types in C++.
- `std::array` 's iterators are specifically designed to work with arrays.
- Iterators can be used with various algorithms from the C++ Standard Template Library (STL) for efficient array manipulation.
- Using iterators can make your code more readable and maintainable.

Additional Notes:

- You can also use reverse iterators (`rbegin()` and `rend()`) to iterate through the array in reverse order.
- C++11 introduced range-based for loops, which provide a more concise syntax for iterating through containers like `std::array` . This is often preferred over using iterators directly.