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.

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 with 0

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() pop_back() begin() and ()

↳ Can prevent memory leaks

Array Class

array (T, N)

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

```
#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 compiles 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.
  - `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.

↳ 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 with a loop

Basic Usage:

C++

```
#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.

Vectors class

↳ Allow dynamic resizing → Eliminate the need of manual memory management

↳ handles memory allocation and deallocation (prevent memory leaks)

`std::vector` **increases its size dynamically as needed.** This means that you don't have to specify the exact size of the vector upfront, and it can grow or shrink as you add or remove elements.

Not ideal to grow indefinitely
↳ But too faster access !!

Here's how the resizing process works:

1. **Capacity:** `std::vector` maintains an internal capacity, which is the maximum number of elements it can store without reallocating memory.
2. **Push Back:** When you add an element to the vector using `push_back()`, the vector checks if its current capacity is sufficient.
3. **Reallocation:** If the capacity is not enough, the vector reallocates memory to accommodate the new element. This involves creating a new array with a larger capacity, copying the existing elements to the new array, and then releasing the old array.
4. **Growth Factor:** The growth factor determines how much the vector's capacity increases when it needs to be reallocated. Typically, the growth factor is a constant value greater than 1, such as 1.5 or 2. This means that the capacity will increase by a certain percentage each time a reallocation is necessary.

Key Points:

- Reallocation can be expensive, especially for large vectors.
- The growth factor can affect the performance of the vector, as a larger growth factor can lead to fewer reallocations but may also waste memory.
- You can use the `reserve()` method to preallocate a specific capacity for the vector, which can help to reduce the number of reallocations.

1. Creating a `std::vector`

A `std::vector` is essentially a dynamic array that automatically resizes as elements are added or removed.

cpp — Copy code

```cpp
#include <iostream>
#include <vector>

int main() {
    std::vector<int> numbers; // Creates an empty vector of integers
}
```

↳ Still need to specify the type !

You can also initialize a vector with a specific size and default value:

cpp — Copy code

```cpp
std::vector<int> numbers(5, 10); // Vector of size 5, all elements initia
```

Size of 5 with all elements equal 10

2. Adding Elements

You can add elements to a `std::vector` using the `push_back()` method, which appends an element to the end of the vector.

cpp — Copy code

```cpp
std::vector<int> numbers;
numbers.push_back(5);  // Adds 5 at the end
numbers.push_back(10); // Adds 10 at the end
```

Inserted in the middle of the array

3. Accessing Elements

You can access elements using the index, just like an array:

cpp — Copy code

```cpp
int firstElement = numbers[0]; // Access the first element
int secondElement = numbers.at(1); // Access the second element using the
```

↳ prefer to use at()

The difference between `[]` and `at()` is that `at()` checks bounds and throws an exception if the index is out of range, while `[]` does not.

4. Iterating Through a `std::vector`

You can use different ways to iterate through a vector:

Range-based for loop (C++11):

cpp — Copy code

```cpp
for (int num : numbers) {
```

Yes, you can insert an element in the middle of a `std::vector` using the `insert()` method. This method takes two parameters:

- The iterator pointing to the position where you want to insert the element.
- The value of the element to insert.

Here's an example:

C++

```cpp
#include <iostream>
#include <vector>

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

    // Insert the value 100 at index 2 (between 2 and 3)
    numbers.insert(numbers.begin() + 2, 100);

    // Print the vector
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Use o código com cuidado.

Output:

```
1 2 100 3 4 5
```

In this example, the `insert()` method is used to insert the value 100 at index 2 of the `numbers` vector. This shifts the elements at index 2 and beyond to the right, making room for the new element.

You can also use the `insert()` method to insert multiple elements at once. For example:

C++

```cpp
numbers.insert(numbers.begin() + 2, 3, 0);
```

Use o código com cuidado.

This will insert three elements with the value 0 at index 2 of the `numbers` vector.

Range-based for loop (C++11):

```cpp
for (int num : numbers) {
    std::cout << num << " ";
}
```

Using iterators:

```cpp
for (std::vector<int>::iterator it = numbers.begin(); it != numbers.end(); ++it) {
    std::cout << *it << " ";
}
```

5. Removing Elements

You can remove the last element using `pop_back()`:

```cpp
numbers.pop_back(); // Removes the last element
```

You can remove specific elements or ranges of elements using `erase()`:

```cpp
numbers.erase(numbers.begin() + 1); // Removes the second element
```

6. Getting Size and Capacity

- `size()`: Returns the number of elements in the vector.
- `capacity()`: Returns the amount of space allocated for the vector (it may be greater than the size).

```cpp
std::cout << "Size: " << numbers.size() << std::endl;
std::cout << "Capacity: " << numbers.capacity() << std::endl;
```

7. Resizing and Clearing

You can resize a vector:

```cpp
numbers.resize(10); // Resizes the vector to hold 10 elements
```

And you can clear all elements in the vector:

```cpp
numbers.clear(); // Removes all elements, size becomes 0
```

This will insert three elements with the value 0 at index 2 of the `numbers` vector.