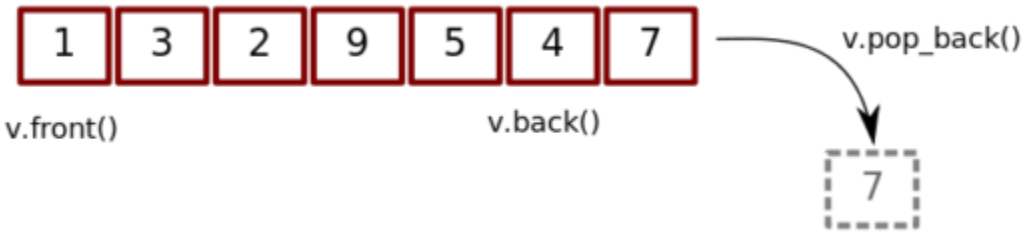
# Vector trong C++ là gì?

Không giống như array (mảng), chỉ một số giá trị nhất định có thể được lưu trữ dưới một tên biến duy nhất. Vector trong C++ giống dynamic array (mảng động) nhưng có khả năng tự động thay đổi kích thước khi một phần tử được chèn hoặc xóa tùy thuộc vào nhu cầu của tác vụ được thực thi, với việc lưu trữ của chúng sẽ được vùng chứa tự động xử lý. Các phần tử vector được đặt trong contiguous storage (bộ nhớ liền kề) để chúng có thể được truy cập và duyệt qua bằng cách sử dụng iterator.




## Vì sao nên dùng Vector

Nếu bạn đã phát chán việc quản lý mảng động qua con trỏ trong C++ hay chán phải tạo mảng mới, copy các phần tử qua mảng mới, rồi lại xóa mảng cũ mỗi khi bạn muốn resize kích thước mảng động trong C++. Thật thừa thải, tốn thời gian và đó là thời điểm ta nhận ra C++ còn có vector.

## Một số điểm nổi trội của Vector

– Bạn không cần phải khai báo kích thước của mảng ví dụ int A[100]…, vì vector có thể tự động nâng kích thước lên.
– Nếu bạn thêm 1 phần tử vào vector đã đầy rồi, thì nó sẽ tự động tăng kích thước của nó lên để dành chỗ cho giá trị mới này.
– Vector còn giúp cho bạn biết số lượng các phần tử mà bạn đang lưu trong đó.
– Dùng số phần tử âm vẫn được trong vector ví dụ A[-6], A[-9], rất tiện trong việc cài đặt các giải thuật.

## Vector có thứ tự trong C++ không?

Không có vector nào không được sắp xếp trong C++. Các phần tử vector được đặt trong bộ nhớ liền kề để chúng có thể được truy cập và di chuyển qua các iterator. Trong vector, dữ liệu được chèn vào cuối. Việc chèn một phần tử vào cuối sẽ mất thời gian chênh lệch, vì đôi khi có thể cần mở rộng vector. Việc xóa phần tử cuối cùng chỉ mất thời gian không đổi vì không xảy ra thay đổi kích thước. Chèn và xóa ở đầu hoặc giữa vector là tuyến tính theo thời gian.

# Các vector được lưu trữ trong C++ như thế nào?

Để tạo một vector, bạn cần thực hiện theo cú pháp dưới đây:

**Cú pháp:**

```
#include <vector>
//...
vector<object_type> variable_name;
```

**Ví dụ:**

```
#include <vector>
int main()
{
    std::vector<int> my_vector;
}
```

Vậy là chúng ta đã có một vector với mỗi phần tử có kiểu dữ liệu (object_type) là int. Sau đó bạn có thể gắn giá trị cho vector như này:

```
vector<int> my_vector = {1,2,3,4,5}
```

Hoặc bạn cũng có thể tạo một vector rồi gán giá trị của một vector khác cho nó bằng cách:

```
vector<int> my_vector = {1,2,3,4,5};
vector<int> your_vector = my_vector;
```

## Cơ chế ngăn chặn rò rỉ bộ nhớ của Vector

Khi một biến **vector** rời khỏi phạm vi đoạn code mà chương trình đang chạy, nó sẽ tự động giải phóng những phần bộ nhớ mà nó kiểm soát (nếu cần). Điều này không chỉ tiện dụng (vì bạn không cần tự tay giải phóng bộ nhớ), mà nó còn giúp ngăn ngừa lỗi rò rỉ bộ nhớ (memory leaks).

**Xem hàm dưới đây:**

```
void doSomething(bool earlyExit)
{
	int *array = new int[3]{ 1, 3, 2 };

	if (earlyExit) // thoát khỏi hàm
		return;
```

```
        delete[] array; // trường hợp hàm thoát sớm, array sẽ không
bị xóa
}
```

Nếu biến earlyExit được gán là true, mảng array sẽ không bao giờ được giải phóng, và bộ nhớ sẽ bị rò rỉ.

Tuy nhiên, nếu biến array là một vector, điều này sẽ không xảy ra, bởi vì bộ nhớ sẽ được giải phóng ngay sau khi biến array nằm ngoài phạm vi đoạn code mà chương trình đang chạy (bất kể hàm có bị thoát ra sớm hay không). Điều này làm cho vector an toàn hơn nhiều so với việc bạn phải tự chú ý đến việc giải phóng bộ nhớ.

## Vector tự ghi nhớ độ dài của mình

Không giống như mảng động được tích hợp sẵn của C++, cái mà không biết được độ dài của mảng mà nó đang trỏ tới là bao nhiêu, **std::vectors** tự theo dõi độ dài của chính nó. Chúng ta có thể lấy được độ dài của vector thông qua hàm **size()**:

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

void printLength(const std::vector<int>& array)
{
    std::cout << "The length is: " << array.size() << '\n';
}

int main()
{
    std::vector array { 9, 7, 5, 3, 1 };
    printLength(array);

    return 0;
}
```

**Output:**

```
The length is: 5
```

Tương tự với array, hàm size() sẽ trả về một giá trị thuộc kiểu nested type (kiểu dữ liệu lồng) là size_type, nó là một số nguyên không dấu.

# Các hàm của Vectors trong C ++

Vector trong STL cung cấp cho chúng ta nhiều chức năng hữu ích khác nhau.

1. **Modifiers**
2. **Iterators**
3. **Capacity**
4. **Element access**

## Modifiers

1. **push_back()**: Hàm đẩy một phần tử vào vị trí sau cùng của vector. Nếu kiểu của đối tượng được truyền dưới dạng tham số trong push_back() không giống với kiểu của vector thì sẽ bị ném ra.

```
ten-vector.push_back(ten-cua-phan-tu);
```

2. **assign()**: Nó gán một giá trị mới cho các phần tử vector bằng cách thay thế các giá trị cũ.

```
ten-vector.assign(int size, int value);
```

3. **pop_back()**: Hàm pop_back () được sử dụng để xóa đi phần tử cuối cùng một vector.

```
ten-vector.pop_back();
```

4. **insert()**: Hàm này chèn các phần tử mới vào trước phần tử trước vị trí được trỏ bởi vòng lặp. Chúng ta cũng có thể chuyển một số đối số thứ ba, đếm số lần phần tử được chèn vào trước vị trí được trỏ.

```
ten-vector.insert(position, value);
```

5. **erase()**: Hàm được sử dụng để xóa các phần tử tùy theo vị trí vùng chứa

```
ten-vector.erase(position);
ten-vector.erase(start-position, end-position);
```

6. **emplace()**: Nó mở rộng vùng chứa bằng cách chèn phần tử mới vào

```
ten-vector.emplace(ten-vector.position, element);
```

7. **emplace_back()**: Nó được sử dụng để chèn một phần tử mới vào vùng chứa vector, phần tử mới sẽ được thêm vào cuối vector

```
ten-vector.emplace_back(value);
```

8. **swap()**: Hàm được sử dụng để hoán đổi nội dung của một vector này với một vector khác cùng kiểu. Kích thước có thể khác nhau.

```
ten-vector-1.swap(ten-vector-2);
```

9. **clear()**: Hàm được sử dụng để loại bỏ tất cả các phần tử của vùng chứa vector.

```
ten-vector.clear();
```

**Ví dụ:**

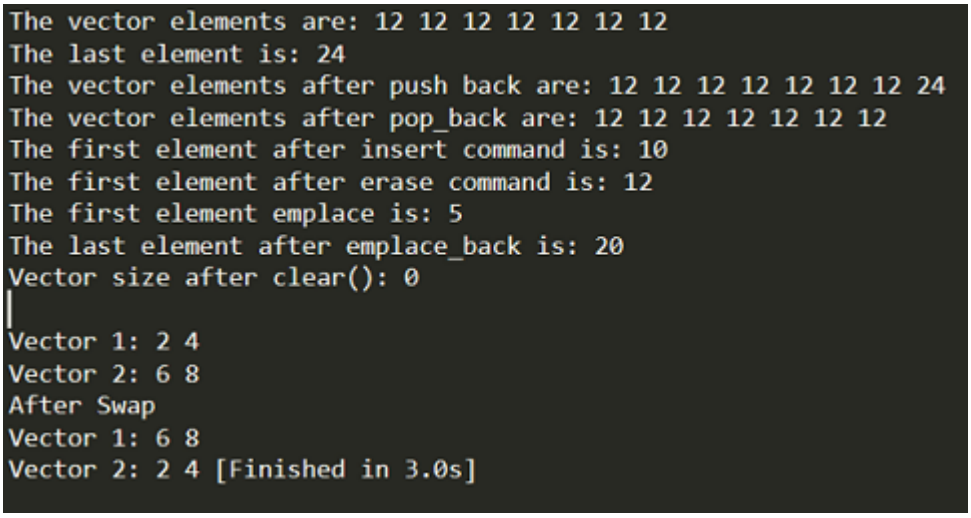
```
/ Modifiers in vector
```

```cpp
#include <bits/stdc++.h>
#include <vector>
using namespace std;

int main()
{
    // Assign vector
    vector<int> vec;

    // fill the array with 12 seven times
    vec.assign(7, 12);

    cout << "The vector elements are: ";
    for (int i = 0; i < 7; i++)
        cout << vec[i] << " ";

    // inserts 24 to the last position
    vec.push_back(24);
    int s = vec.size();
    cout << "nThe last element is: " << vec[s - 1];

     // prints the vector
    cout << "nThe vector elements after push back are: ";
    for (int i = 0; i < vec.size(); i++)
    cout << vec[i] << " ";

    // removes last element
    vec.pop_back();

    // prints the vector
    cout << "nThe vector elements after pop_back are: ";
    for (int i = 0; i < vec.size(); i++)
        cout << vec[i] << " ";

    // inserts 10 at the beginning
    vec.insert(vec.begin(), 10);

    cout << "nThe first element after insert command is: " <<
vec[0];

    // removes the first element
```

```
    vec.erase(vec.begin());

    cout << "nThe first element after erase command is: " <<
vec[0];

    // inserts at the beginning
    vec.emplace(vec.begin(), 5);
    cout << "nThe first element emplace is: " << vec[0];

    // Inserts 20 at the end
    vec.emplace_back(20);
    s = vec.size();
    cout << "nThe last element after emplace_back is: " << vec[s -
1];

    // erases the vector
    vec.clear();
    cout << "nVector size after clear(): " << vec.size();

    // two vector to perform swap
    vector<int> obj1, obj2;
    obj1.push_back(2);
    obj1.push_back(4);
    obj2.push_back(6);
    obj2.push_back(8);

    cout << "nnVector 1: ";
    for (int i = 0; i < obj1.size(); i++)
        cout << obj1[i] << " ";

    cout << "nVector 2: ";
    for (int i = 0; i < obj2.size(); i++)
        cout << obj2[i] << " ";

    // Swaps obj1 and obj2
    obj1.swap(obj2);

    cout << "nAfter Swap nVector 1: ";
    for (int i = 0; i < obj1.size(); i++)
        cout << obj1[i] << " ";
```

```
    cout << "nVector 2: ";
    for (int i = 0; i < obj2.size(); i++)
        cout << obj2[i] << " ";
}
```

**Output:**




Bạn có thể thấy cách sử dụng các hàm thuộc nhóm **Modifiers** mà chúng ta đã nghiên cứu ở trên qua ví dụ thực tiễn.

## Iterators

1. **begin()**: đặt iterator đến phần tử đầu tiên trong vector`ten-vector.begin();`
2. **end()**: đặt iterator đến sau phần tử cuối cùng trong vector`ten-vector.end();`
3. **rbegin()**: đặt reverse iterator (trình lặp đảo) đến phần tử cuối cùng trong vector (reverse begin). Nó di chuyển từ phần tử cuối cùng đến phần tử đầu tiên
   `ten-vector.rbegin();`
4. **rend()**: đặt reverse iterator (trình lặp đảo) đến phần tử đầu tiên trong vector (reverse end)
   `ten-vector.rend();`
5. **cbegin()**: đặt constant iterator (trình vòng lặp) đến phần tử đầu tiên trong vector`ten-vector.cbegin();`
6. **cend()**: đặt constant iterator (trình vòng lặp) đến phần tử cuối cùng trong vector`ten-vector.cend();`

7. **crbegin()**: đặt constant reverse iterator (trình lặp đảo liên tục) đến phần tử cuối cùng trong vector (reverse begin). Nó di chuyển từ phần tử cuối cùng đến phần tử đầu tiên`ten-vector.cbregin();`
8. **crend()**: đặt constant reverse iterator (trình lặp đảo liên tục) đến phần tử đầu tiên trong vector (reverse end)`ten-vector.crend();`

**Ví dụ 1:**

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

using namespace std;

int main()
{
    vector<int> vec1;

    for (int i = 1; i <= 10; i++)
        vec1.push_back(i);

    cout << "Understanding begin() and end() function: " << endl;
    for (auto i = vec1.begin(); i != vec1.end(); ++i)
        cout << *i << " ";

    return 0;
}
```

**Output:**

```
Understanding begin() and end() function:
1 2 3 4 5 6 7 8 9 10 [Finished in 1.9s]
```

Trong ví dụ trên, chúng ta có thể thấy cách sử dụng hàm begin() và end(). Đầu tiên, chúng ta xác định một vector là vec1, chúng đẩy lùi các giá trị trong đó từ 1 đến 10 bằng cách sử dụng vòng lặp for. Sau đó, chúng in các giá trị của các vector của chúng tôi bằng cách sử dụng vòng lặp for, chúng tôi sử dụng hàm begin() và end() để chỉ định điểm đầu và điểm cuối của vòng lặp for của chúng tôi.

**Ví dụ 2:**

```cpp
// C++ program to illustrate the
// iterators in vector
#include <iostream>
#include <vector>
```

```cpp
using namespace std;

int main()
{
    vector<int> g1;

    for (int i = 1; i <= 5; i++)
        g1.push_back(i);

    cout << "Output of begin and end: ";
    for (auto i = g1.begin(); i != g1.end(); ++i)
        cout << *i << " ";

    cout << "\nOutput of cbegin and cend: ";
    for (auto i = g1.cbegin(); i != g1.cend(); ++i)
        cout << *i << " ";

    cout << "\nOutput of rbegin and rend: ";
    for (auto ir = g1.rbegin(); ir != g1.rend(); ++ir)
        cout << *ir << " ";

    cout << "\nOutput of crbegin and crend : ";
    for (auto ir = g1.crbegin(); ir != g1.crend(); ++ir)
        cout << *ir << " ";

    return 0;
}
```

**Output:**

```
Output of begin and end: 1 2 3 4 5
Output of cbegin and cend: 1 2 3 4 5
Output of rbegin and rend: 5 4 3 2 1
Output of crbegin and crend : 5 4 3 2 1
```

**Ví dụ 3:**

```cpp
// CPP program to illustrate working of crbegin()
// crend()
#include <iostream>
#include <vector>
using namespace std;

int main ()
```

```
{
  // initializing vector with values
  vector<int> vect = {10, 20, 30, 40, 50};

  // for loop with crbegin and crend
  for (auto i = vect.crbegin(); i != vect.crend(); i++)
    cout << ' ' << *i;  //printing results

  cout << '\n';
  return 0;
}
```

**Output:**

```
50 40 30 20 10
```

## Capacity

1. **size()**: hàm sẽ trả về số lượng phần tử đang được sử dụng trong vector`ten-vector.size();`
2. **max_size()**: hàm trả về số phần tử tối đa mà vector có thể chứa`ten-vector.max_size();`
3. **capacity()**: hàm trả về số phần tử được cấp phát cho vector nằm trong bộ nhớ`ten-vector.capacity();`
4. **resize(n)**: Hàm này thay đổi kích thước vùng chứa để nó chứa đủ n phần tử. Nếu kích thước hiện tại của vector lớn hơn n thì các phần tử phía sau sẽ bị xóa khỏi vector và ngược lại nếu kích thước hiện tại nhỏ hơn n thì các phần tử bổ sung sẽ được chèn vào phía sau vector`ten-vector.resize(int n, int value);`
5. **empty()**: Trả về liệu vùng chứa có trống hay không, nếu trống thì trả về True, nếu có phần tử thì trả về False`ten-vector.empty();`
6. **shrink_to_fit()**: Giảm dung lượng của vùng chứa để phù hợp với kích thước của nó và hủy tất cả các phần tử vượt quá dung lượng`ten-vector.shrink_to_fit();`
7. **reserve(n)**: hàm cấp cho vector số dung lượng vừa đủ để chứa n phần tử`ten-vector.reserve(n);`

**Ví dụ:**

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

using namespace std;

int main()
```

```
{
    vector<int> vec1;

    for (int i = 1; i <= 10; i++)
        vec1.push_back(i);

    cout << "Size of our vector: " << vec1.size();
    cout << "nCapacity of our vector: " << vec1.capacity();
    cout << "nMax_Size of our vector: " << vec1.max_size();

    // resizes the vector size to 4
    vec1.resize(4);

    // prints the vector size after resize()
    cout << "nSize of our vector after resize: " << vec1.size();

    // checks if the vector is empty or not
    if (vec1.empty() == false)
        cout << "nVector is not empty";
    else
        cout << "nVector is empty";

    return 0;
}
```

**Output:**

```
Size of our vector: 10
Capacity of our vector: 16
Max_Size of our vector: 1073741823
Size of our vector after resize: 5
Vector is not empty[Finished in 1.9s]
```

**Chú ý:** Khi thay đổi kích thước mảng vector, các giá trị của phần tử hiện có cần giữ nguyên, thì các phần tử mới được khởi tạo bằng giá trị mặc định của kiểu dữ liệu mảng.

Chúng ta có thể thấy cách hoạt động của các hàm capacity như đã thảo luận ở trên.

## Element access

1. **at(g)**: Trả về một tham chiếu đến phần tử ở vị trí ‘g’ trong vector`ten-vector.at(position);`

2. **data()**: Trả về một con trỏ trực tiếp đến (memory array) bộ nhớ mảng được vector sử dụng bên trong để lưu trữ các phần tử thuộc sở hữu của nó`ten-vector.data();`
3. **front()**: hàm dùng để lấy ra phần tử đầu tiên của vector`ten-vector.front();`
4. **back()**: hàm dùng để lấy ra phần tử cuối cùng của vector`ten-vector.back();`

**Ví dụ:**

```cpp
// C++ program to illustrate the
// element accesser in vector
#include <bits/stdc++.h>
using namespace std;

int main()
{
    vector<int> g1;

    for (int i = 1; i <= 10; i++)
        g1.push_back(i * 10);

    cout << "\nReference operator [g] : g1[2] = " << g1[2];

    cout << "\nat : g1.at(4) = " << g1.at(4);

    cout << "\nfront() : g1.front() = " << g1.front();

    cout << "\nback() : g1.back() = " << g1.back();

    // pointer to the first element
    int* pos = g1.data();

    cout << "\nThe first element is " << *pos;
    return 0;
}
```

**Output:**

```
Reference operator [g] : g1[2] = 30
at : g1.at(4) = 50
front() : g1.front() = 10
back() : g1.back() = 100
The first element is 10
```

# Kết

Với những cú pháp mẫu và ví dụ thực tiễn trên, chúng ta kết thúc phần tìm hiểu về vector trong C++. Tôi hy vọng bạn đã phân biệt được các hàm khác nhau của vector và nắm được cách hoạt động của từng hàm. Vì vector hỗ trợ rất tốt trong việc thao tác với mảng động, đảm bảo an toàn và dễ dàng hơn. Bạn nên sử dụng vector trong hầu hết các trường hợp đụng tới mảng động.