(Ghi chú)

Ngôn ngữ lập trình cpp

**Thực hiện: Nguyễn Văn Hào**

**Email:** [hao2205tb@gmail.com](mailto:hao2205tb@gmail.com)

(Tài liệu được biên xoạn lại từ nhiều nguồn và không dùng trong mục đích kinh doanh kiếm tiền)

# Mục lục

[Mục lục **Error! Bookmark not defined.**](#_Toc126778440)

[Vector **Error! Bookmark not defined.**](#_Toc126778441)

[Tham khảo **Error! Bookmark not defined.**](#_Toc126778442)

# Vector

## Tổng quan

Kiểu vector trong C++ hay còn gọi là mảng động hoặc mảng độ dài thay đổi, là kiểu dữ liệu có chức năng tương tự như mảng trong C++ nhưng lại có khả năng tự thay đổi độ dài.

std::vector được cài sẵn trong header file vector và để sử dụng được chức năng này, chúng ta cần thêm dòng 「#include <vector> 」 vào đầu chương trình.

#include <vector>

int main()

{

    std::vector<int> v1;

    std::vector<double> v2;

}

Lại nữa, namespace của std::vector là std, do đó bằng cách khai báo sử dụng namespace này vào đầu chương trình mà chúng ta có thể viết gọn std::vector trong chương trình như sau:

#include <vector>

using namespace std;

int main()

{

   vector<int> v1;

   vector<double> v2;

}

## Khai báo

### Khai báo 1 vector

using namespace std;

vector <type> data;

Trong đó:

* type: kiểu dữ liệu
* data: tên biến vector

VD

#include <iostream>

#include <vector>

using namespace std;

int main()

{

    vector<string> name; //Khai báo vector name kiểu string

    vector<int> age;     //Khai báo vector age kiểu int

}

### Khai báo bao gồm chỉ định số phần tử

std::vector<type> data(length);

Trong đó length chính là độ dài(số phần tử() của vector cần tạo.

Với cách khai báo vector này thì các phần tử trong vector sẽ được gán giá trị mặc định tùy thuộc vào kiểu dữ liệu sử dụng

Ví dụ, nếu dùng kiểu số như int, double thì các phần tử sẽ có giá trị mặc định bằng 0 như sau:

VD

/\*Khai báo vector data kiểu số\*/

std::vector<int> data1(3);    //{0, 0, 0}

std::vector<double> data2(3); //{0, 0, 0}

Tuy nhiên nếu dùng kiểu chữ như char, string thì các phần tử sẽ có giá trị mặc định bằng Null được biểu diễn bởi trống khi in ra màn hình như sau:

VD

/\*Khai báo vector data kiểu chữ\*/

std::vector<char> data1(3)   //{, , }

std::vector<string> data2(3) //{, , }

### Khai báo đồng thời nhiều vector

vector<type> name1, name2, name3, ... ;

VD

#include <iostream>

#include <vector>

using namespace std;

int main()

{

    vector<string> name, job, sex;

    vector<int> age;

}

## Gán giá trị

data[index] = value;

VD

#include <iostream>

#include <vector>

using namespace std;

int main()

{

    vector<int> data(3);

    data[0]=1;

    data[1]=4;

    data[2]=88;

}

## Khởi tạo

### Khởi tạo vector trong C++ với các phần tử riêng biệt

std::vector<type> data {value1, value2, value3, ...};

Trong đó

* type là kiểu dữ liệu
* data là tên biến vector
* value là các giá trị của vector

VD

std::vector<string> user{"Kiyoshi", "male", "Tokyo"};

//{"Kiyoshi", "male", "Tokyo"}

### Khởi tạo vector trong C++ với các phần tử giống nhau

std::vector<type> data(length, value);

Trong đó length là số phần tử(độ dài), và value là giá trị sử dụng để khai báo đồng loạt các phần tử của vector cần tạo.

VD

//Khởi tạo vector data kiểu int gồm 5 phần tử có giá trị bằng 8

std::vector<int> data(5, 8); //{8, 8, 8, 8, 8}

### Khai báo vector 2 chiều

vector<vector<type> > data {v1, v2, v3, ...};

Trong đó:

* data là tên biến vector 2 chiều
* v là các vector 1 chiều được sử dụng như phần tử của vector 2 chiều

Lưu ý, chúng ta cần phải viết thêm dấu cách giữa cặp dấu > > khi khai báo vector 2 chiều. Lý do là để phân biệt với toán tử >> được sử dụng để dịch chuyển bit trong C++.

VD

#include <iostream>

#include <vector>

using namespace std;

int main()

{

    /\*Khai báo vector 2 chiều\*/

    vector<vector<string> > all\_user{

        {"Kiyoshi", "male", "Hanoi"},

        {"Honda", "male", "Tokyo"},

        {"Ajinomoto", "female", "Osaka"}};

    return 0;

}

Chúng ta cũng có thể khởi tạo các vector 1 chiều trước rồi dùng chúng để khai báo vector 2 chiều như sau:

#include <iostream>

#include <vector>

using namespace std;

int main()

{

    /\*Khởi tạo các vector 1 chiều làm phần tử trong vector 2 chiều\*/

    vector<string> user1{"Kiyoshi", "male", "Hanoi"};

    vector<string> user2{"Honda", "male", "Tokyo"};

    vector<string> user3{"Ajinomoto", "female", "Osaka"};

    /\*Khai báo vector 2 chiều\*/

    vector<vector<string> > all\_user{ user1, user2, user3};

    return 0;

}

## Truy cập phần tử

data[index];

Trong đó data là tên biến vector và index là vị trí của cần truy xuất.

#include <iostream>

#include <vector>

using namespace std;

int main()

{

    vector<string> user{"Kiyoshi", "male", "Tokyo"};

    cout << user[0] <<endl;

    cout << user[1] <<endl;

    cout << user[2] <<endl;

    return 0;

}

Kiyoshi

male

Tokyo

## Duyệt vector

### Duyệt vector trong C++ bằng vòng lặp và index

VD

#include <iostream>

#include <vector>

using namespace std;

int main()

{

    vector<int> data{5, 6, 88, -2};

    for (short i=0; i< data.size();i++){

        cout << data[i] <<endl;

    }

    return 0;

}

5

6

88

-2

### Duyệt vector trong C++ bằng vòng lặp dựa trên phạm vi

Bằng cách sử dụng vòng lặp dựa trên phạm vi, chúng ta có thể rút gọn code khi duyệt vector trong C++ với cú pháp như sau:

for ( auto & x : v) {

// Xử lý

}

Trong đó:

* v là tên vector
* x là tên một biến dùng để gán từng phần tử được lấy từ vector
* auto là kiểu suy luận giúp tự xác định kiểu dữ liệu của giá trị lấy từ vector

VD

#include <iostream>

#include <vector>

using namespace std;

int main()

{

    vector<int> data{5, 6, 88, -2};

    for (auto x: data) {

        cout << x << endl;

    }

}

5

6

88

-2

Nếu trong vector chỉ chứa các phần tử thuộc kiểu dữ liệu nguyên thủy, chúng ta cũng có thể thay thế auto bằng tên kiểu, ví dụ vector ở trên chỉ chứa các phần tử thuộc kiểu int nên chúng ta có thể viết:

vector<int> data{5, 6, 88, -2};

for (int x: data) {

    cout << x << endl;

}

### Duyệt vector trong C++ bằng iterator

Trong C++, các kiểu dữ liệu như vector, list, map đều được thêm một chức năng là iterator nhằm giúp biến chúng thành các trình lặp để dễ dàng xử lý.

Bằng cách sử dụng iterator, chúng ta có thể duyệt vector trong C++ với cú pháp như sau:

for(auto itr = v.begin(); itr != v.end(); ++itr) {

// Xử lý

}

Trong đó:

* v là tên vector
* itr là tên iterator

VD

#include <iostream>

#include <vector>

using namespace std;

int main()

{

    vector<int> data{5, 6, 88, -2};

    for(auto itr = data.begin(); itr != data.end(); ++itr) {

        cout << \*itr << endl;

    }

}

5

6

88

-2

## Kích thước vector

### Lấy kích thước vector

vt.size();

Trong đó vt là vector cần kiểm tra.

Hàm size sẽ trả về kích thước của vector, cũng chính là số phần tử có trong vector đó.

VD

#include <iostream>

#include <vector>

using namespace std;

int main()

{

    vector<int> data{5, 6, 88, -2};

    for (short i=0; i< data.size();i++){

        cout << data[i] <<endl;

    }

    return 0;

}

5

6

88

-2

### Thay đổi kích thước vector

vt.resize(n);

Trong đó vt là vector cần thay đổi kích thước, và n là kích thước cần thay đổi.

Nếu n nhỏ hơn kích thước(số phần tử) ban đầu của vector thì các phần tử bị thừa sẽ bị xóa đi.

Ngược lại nếu n lớn hơn kích thước(số phần tử) ban đầu của vector, các giá trị mặc định tùy thuộc vào kiểu dữ liệu sẽ được thêm vào vector cho đủ độ dài. Ví dụ nếu vector thuộc kiểu số như int hay double thì số 0 sẽ được thêm vào. Còn nếu vector thuộc kiểu chữ như char hay vector, giá trị trống sẽ được thêm vào.

VD

#include <iostream>

#include <vector>

using namespace std;

//Tạo hàm xuất vector

void dump(vector<int>& vt)

{

    for (auto x: vt) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    vector<int> vt{3, 1, 4,5};

    dump(vt);

    cout<< "Size before: "<<vt.size()<<endl;

    /\*Thay đổi kích thước nhỏ hơn ban đầu\*/

    vt.resize(2);

    dump(vt);

    cout<< "Size after: "<<vt.size()<<endl;

    /\*Thay đổi kích thước lớn hơn ban đầu\*/

    vt.resize(10);

    dump(vt);

    cout<< "Size after: "<<vt.size()<<endl;

    return 0;

}

3 1 4 5

Size before: 4

3 1

Size after: 2

3 1 0 0 0 0 0 0 0 0

Size after: 10

## Lấy phần tử

### Lấy phần tử đầu tiên

vt.front();

Trong đó vt là vector cần cắt ra phần tử ban đầu. Hàm front() sẽ trả về tham chiếu tới phần tử đầu tiên của vector, qua đó chúng ta có thể cắt lấy giá trị của phần tử đầu tiên này, hoặc là gán một phần tử khác để thay đổi nó.

Lưu ý chúng ta không dùng hàm front() cho vector trống.

VD

#include <iostream>

#include <vector>

using namespace std;

//Tạo hàm xuất vector

void dump(vector<int>& vt)

{

    for (auto x: vt) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main() {

    vector<int> vt{3, 1, 4, 1, 5, 7, 9};

    dump(vt);

    //Tạo tham chiếu tới vị trí phần tử đầu tiên của vector

    int front\_element = vt.front();

    cout << front\_element << endl;

    //Thay đổi phần tử đầu tiên

    vt.front() = 2;

    dump(vt);

    return 0;

}

3 1 4 1 5 7 9

3

2 1 4 1 5 7 9

### Lấy phần tử cuối cùng

vt.back();

Trong đó vt là vector cần cắt ra phần tử ban đầu. Hàm back() sẽ trả về tham chiếu tới phần tử cuối cùng của vector, qua đó chúng ta có thể cắt lấy giá trị của phần tử cuối cùng này, hoặc là gán một phần tử khác để thay đổi nó.

Lưu ý chúng ta không dùng hàm back() cho vector trống.

VD

#include <iostream>

#include <vector>

using namespace std;

//Tạo hàm xuất vector

void dump(vector<int>& vt)

{

    for (auto x: vt) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main() {

    vector<int> vt{3, 1, 4, 1, 5, 7, 9};

    dump(vt);

    //Tạo tham chiếu tới vị trí phần tử cuối cùng của vector

    int back\_element = vt.back();

    cout << back\_element << endl;

    //Thay đổi phần tử cuối cùng

    vt.back() = 2;

    dump(vt);

    return 0;

}

3 1 4 1 5 7 9

3

2 1 4 1 5 7 9

### Lấy một phạm vi từ vector

Sao chép từng phần tử trong phạm vi vào vector đích

VD

#include <iostream>

#include <vector>

using namespace std;

//Tạo hàm xuất vector

void dump(vector<int>& vt)

{

    for (auto x: vt) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main() {

    vector<int> vt{3, 1, 4, 1, 5, 7, 9};

    dump(vt);

    //Khai báo vector chứa kết quả

    vector<int> vt2;

    //Khai báo phạm vi cắt

    int start = 2, end = 4;

    //Truy cập vào phạm vi để lấy và lưu các phần tử vào vector kết quả

    for (short i = start; i<= end;i++){

        vt2.push\_back(vt[i]);

    }

    dump(vt2);

    return 0;

}

3 1 4 1 5 7 9

4 1 5

## Thêm chèn phần tử vào vector

### Thêm phần tử vào cuối vector bằng push\_back()

v.push\_back(value);

Trong đó v là tên vector, và value là giá trị sẽ được copy vào phần tử vừa được tạo ra. Nếu phần tử thêm vào có kiểu khác với kiểu của vector thì nó sẽ được chuyển về kiểu của vector trước khi được thêm vào.

Hàm push\_back thuộc kiểu void, do đó nó sẽ không trả giá trị. Bởi vậy nó sẽ thêm phần tử vào vị trí cuối cùng trong vector chứ không trả về chuỗi kết quả.

VD

#include <iostream>

#include <vector>

using namespace std;

//Tạo hàm xuất vector

void dump(vector<int>& v)

{

    for (auto x: v) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main() {

    vector<int> v{3, 1, 4, 1, 5, 7, 9};

    dump(v);

    v.push\_back(88);

    dump(v);

    return 0;

}

3 1 4 1 5 7 9

3 1 4 1 5 7 9 88

Lưu ý, nếu phần tử thêm vào có kiểu khác với kiểu của vector thì nó sẽ được chuyển về kiểu của vector trước khi được thêm vào. Ví dụ nếu thêm một ký tự vào một vector thuộc kiểu int, thì ký tự đó sẽ được chuyển về mã Unicode trước khi được thêm vào vector ban đầu như sau:

vector<int> v{3, 1, 4, 1, 5, 7, 9};

dump(v);

// 3 1 4 1 5 7 9

v.push\_back('a');

dump(v);

// 3 1 4 1 5 7 9 97

### Thêm phần tử vào cuối vector bằng emplace\_back()

v.emplace\_back(args);

Trong đó v là tên vector, và args là các thông tin được sử dụng trong hàm tạo value\_type tương ứng với kiểu giá trị để tạo nên phần tử trong vector.

VD

#include <iostream>

#include <vector>

using namespace std;

//Tạo hàm xuất vector

void dump(vector<int>& v)

{

    for (auto x: v) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main() {

    vector<int> v{3, 1, 4, 1, 5, 7, 9};

    dump(v);

    v.emplace\_back(88);

    dump(v);

    return 0;

}

3 1 4 1 5 7 9

3 1 4 1 5 7 9 88

### Khác biệt giữa push\_back và emplace\_back

Hàm emplace\_back khác với hàm push\_back ở chỗ, nó không nhận trực tiếp giá trị và copy giá trị này vào phần tử được thêm, mà nó sẽ dùng giá trị này như là đối số trong hàm tạo nên phần tử.

Ví dụ, nếu chúng ta thêm phần tử vào vector bằng hàm push\_back:

vector<int> v{3, 1, 4, 1, 5, 7, 9};

v.push\_back(88);

Điều đó có nghĩa là giá trị 88 sẽ được sử dụng trực tiếp để copy vào một phần tử mới được tạo ở cuối vector.

Còn nếu chúng ta thêm phần tử vào vector bằng hàm emplace\_back:

vector<int> v{3, 1, 4, 1, 5, 7, 9};

v.emplace\_back(88);

Điều đó có nghĩa là giá trị 88 được sử dụng như đối số trong hàm tạo thuộc class std:In tương ứng với kiểu int của giá trị 88. Hàm tạo này sẽ tạo ra phần tử mới được tạo ở cuối vector.

Do hàm emplace\_back sẽ tạo trực tiếp phần tử bên trong vector mà không cần phải qua việc copy (di chuyển) giá trị vào trong vector, nên nó được cho là có tốc độ xử lý nhanh hơn hàm push\_back.

### Chèn 1 phần tử vào vị trí chỉ định trong vector C++ bằng hàm insert

v.insert(p, value);

Trong đó v là vector ban đầu, value là phần tử cần chèn, và p là trình lặp trỏ đến vị trí cần chèn trong vector.

Trong trường hợp cần chỉ đến vị trí index thứ n trong vector, chúng ta sẽ viết trình lặp p như sau:

v.insert(v.begin() + i, value);

Trong đó v.begin() chỉ đến vị trí đầu tiên trong vector, và i là index của vị trí phần tử cần chỉ đến.

VD

#include <iostream>

#include <vector>

using namespace std;

//Tạo hàm xuất vector

void dump(vector<int>& v)

{

    for (auto x: v) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

  vector<int> v{3, 1, 4, 1, 5, 7, 9};

  v.insert(v.begin()+2, 55); //Thêm phần tử 55 vào vị trí thứ 2

  dump(v);

 v.insert(v.begin()+4, 22); //Thêm phần tử 22 vào vị trí thứ 4

  dump(v);

  return 0;

}

3 1 55 4 1 5 7 9

3 1 55 4 22 1 5 7 9

## Xoá phần tử

### Xóa phần tử cuối cùng trong vector C++ bằng pop\_back

Hàm pop\_back là một hàm thành viên trong class std:vector, có tác dụng xóa phần tử cuối cùng trong vector cũng như giảm độ dài của nó đi một đơn vị.

Cú pháp sử dụng hàm pop\_back để xóa phần tử cuối cùng trong vector như sau:

v.pop\_back();

Trong đó v là tên vector cần xóa phần tử cuối cùng.

VD

#include <iostream>

#include <vector>

using namespace std;

//Tạo hàm xuất vector

void dump(vector<int>& v)

{

    for (auto x: v) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    vector<int> v{3, 1, 4, 1, 5};

    cout << "Before erase" << endl;

    dump (v);

    v.pop\_back();

    cout << "After erase" << endl;

    dump (v);

    return 0;

}

Before erase

3 1 4 1 5

After erase

3 1 4 1

### Xóa 1 phần tử trong vector bằng vector erase c++

vector erase c++ là một hàm thành viên trong class std:vector, có tác dụng xóa một hoặc nhiều phần tử trong vector C++ cũng như làm giảm độ dài tương ứng của nó.

Cú pháp:

v.erase(p);

Trong đó v là vector ban đầu, và p là trình lặp trỏ đến vị trí cần xóa trong vector.

Trong trường hợp cần chỉ đến vị trí index thứ i trong vector, chúng ta sẽ viết trình lặp p như sau:

v.erase(v.begin() + i);

Trong đó v.begin() chỉ đến vị trí đầu tiên trong vector, và i là index của vị trí phần tử cần chỉ đến.

#include <iostream>

#include <vector>

using namespace std;

//Tạo hàm xuất vector

void dump(vector<int>& v)

{

    for (auto x: v) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    vector<int> v{3, 1, 4, 1, 5};

    v.erase(v.begin() + 1);//Xóa phần tử ở vị trí thứ 1

    dump(v);

    v.erase(v.begin() + 3);//Xóa phần tử ở vị trí thứ 3

    dump(v);

    return 0;

}

3 4 1 5

3 4 1

### Xóa các phần tử trong một phạm vi chỉ định bằng vector erase c++

v.erase( iterator\_first, iterator\_last);

Trong đó v là chuỗi ban đầu, iterator\_first và iterator\_last là phạm vi cần xóa được chỉ định trong trình lặp trỏ đến vị trí cần xóa trong vector.

Tương tự như khi xóa 1 phần tử thì cú pháp sử dụng thực tế để xóa các phần tử trong một phạm vi chỉ định trong vector C++ bằng hàm erase sẽ là:

v.erase(v.begin() + start, v.begin() + end);

Trong đó start và end là vị trí index của phạm vi xóa trong vector.

VD

#include <iostream>

#include <vector>

using namespace std;

//Tạo hàm xuất vector

void dump(vector<int>& v)

{

    for (auto x: v) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    vector<int> v{3, 1, 4, 1, 5, 7, 9};

    dump(v);

    //Xóa phần tử trong phạm vi index từ 1 đến 3

    v.erase(v.begin() + 1, v.begin() + 3);

    dump(v);

    //Xóa phần tử trong phạm vi index từ 2 đến 5

    v.erase(v.begin() + 2, v.begin() +5);

    dump(v);

    return 0;

}

3 1 4 1 5 7 9

3 1 5 7 9

3 1

## Sao chép và hoán đổi vector

### Sao chép vector

std::vector<type> vt\_dest(vt\_src);

Trong đó type là kiểu dữ liệu, vt\_src là vector nguồn để copy và vt\_dest là vector đích dùng để dán kết quả sao chép.

VD

#include <iostream>

#include <vector>

using namespace std;

//Tạo hàm xuất vector

void dump(vector<int>& v)

{

    for (auto x: v) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main()

{

    vector<int> vt\_src{3, 1, 4,5};

    cout << "Orginary vector" << endl;

    dump (vt\_src);

    /\*Sao chép vector\*/

    vector<int> vt\_dest(vt\_src);

    cout << "Copy vector" << endl;

    dump (vt\_dest);

    return 0;

}

Orginary vector

3 1 4 5

Copy vector

3 1 4 5

### Hoán đổi 2 vector

vt1.swap(vt2);

Trong đó vt1 và vt2 là 2 vector cần hoán đổi nội dung cho nhau.

VD

#include <iostream>

#include <vector>

using namespace std;

//Tạo hàm xuất vector

void dump(vector<int>& v)

{

    for (auto x: v) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main()

{

    vector<int> vt1{3, 1, 4, 1, 5};

    vector<int> vt2{9, 8, 7};

    cout << "Before swap" << endl;

    dump(vt1);

    dump(vt2);

    vt1.swap(vt2);

    cout << "After swap" << endl;

    dump(vt1);

    dump(vt2);

}

Before swap

3 1 4 1 5

9 8 7

After swap

9 8 7

3 1 4 1 5

Ngoài cách dùng hàm vector swap, chúng ta cũng có thể dùng function template là std::swap để tiến hành hoán đổi 2 vector với nhau, cũng như là để hoán đổi các đối tượng khác như map, vector trong C++.

Lưu ý chúng ta cần phải thêm header file utility vào trong chương trình để có thể sử dụng được function template này

VD

#include <iostream>

#include <utility>

#include <vector>

using namespace std;

//Tạo hàm xuất vector

void dump(vector<int>& v)

{

    for (auto x: v) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main()

{

    vector<int> vt1{3, 1, 4, 1, 5};

    vector<int> vt2{9, 8, 7};

    cout << "Before swap" << endl;

    dump(vt1);

    dump(vt2);

    swap(vt1,vt2);

    cout << "After swap" << endl;

    dump(vt1);

    dump(vt2);

}

Before swap

3 1 4 1 5

9 8 7

After swap

9 8 7

3 1 4 1 5

## Đảo ngược trong vector

### Đảo ngược trong vector C++ bằng std::reverse

std::reverse(vt.begin(), vt.end());

Trong đó vt là tên vector cần đảo ngược thứ tự các phần tử, còn vt.begin() và vt.end() lần lượt là các trình lặp có tác dụng như con trỏ chỉ đến vị trí đầu tiên và cuối cùng của vector.

Để sử dụng hàm std::reverse, chúng ta cần include header file algorithm vào đầu chương trình:

VD

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

//Tạo hàm xuất vector

void dump(vector<int>& vt)

{

    for (auto x: vt) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    vector<int> vt{3, 1, 4, 2, 5};

    cout << "Before reverse" << endl;

    dump (vt);

    reverse(vt.begin(), vt.end());

    cout << "After reverse" << endl;

    dump (vt);

    return 0;

}

Before reverse

3 1 4 2 5

After reverse

5 2 4 1 3

## Xắp xếp vector

### Sắp xếp vector tăng dần bằng hàm sort vector trong C++

sort vector trong C++ là một hàm thành viên trong class std:algorithm, có tác dụng sắp xếp vector trong C++ theo thứ tự tăng dần hoặc giảm dần. Phép sắp xếp vector bằng hàm sort trong C++ sẽ làm thay đổi vector ban đầu.

Để sử dụng hàm sort, chúng ta cần include header file algorithm.

Cú pháp:

std::sort(v.begin(), v.end() );

Trong đó v là vector cần sắp xếp theo thứ tự tăng dần. Các phương thức begin() và end() được dùng để lấy index đầu tiên và cuối cùng trong vector, tương ứng với phạm vi sẽ tiến hành sắp xếp.

VD

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

//Tạo hàm xuất vector

void dump(vector<int>& v)

{

    for (auto x: v) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main()

{

    vector<int> data{3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};

    cout << "Before sort" << endl;

    dump (data);

    /\*Sắp xếp vector theo thứ tự tăng dần\*/

    sort(data.begin(), data.end() );

    cout << "After sort" << endl;

    dump (data);

    return 0;

}

Before sort

3 1 4 1 5 9 2 6 5 3 5

After sort

1 1 2 3 3 4 5 5 5 6 9

### Sắp xếp vector giảm dần bằng hàm sort vector trong C++

Để sắp xếp vector giảm dần, chúng ta sử dụng hàm sort vector trong C++ với cú pháp sau đây:

std::sort(v.begin(), v.end(), std::greater<type>() );

Sự khác biệt duy nhất với cú pháp khi sắp xếp vector theo thứ tự tăng dần bằng hàm sort đó là đối số std::greater<type> có tác dụng chuyển từ sắp xếp tăng dần thành giảm dần, trong đó type là kiểu dữ liệu sử dụng trong vector.

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

void dump(vector<int>& v)

{

    for (auto x: v) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main()

{

    vector<int> data{3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};

    cout << "Before sort" << endl;

    dump (data);

    sort(data.begin(), data.end(), std::greater<int>() );

    cout << "After sort" << endl;

    dump (data);

    return 0;

}

Before sort

3 1 4 1 5 9 2 6 5 3 5

After sort

9 6 5 5 5 4 3 3 2 1 1

## Tính tổng các phần tử

### Tính tổng các phần tử trong vector C++ bằng std::accumulate

Cú pháp:

std::accumulate(vt.begin(), vt.end(), 0);

Trong đó vt là tên vector cần tính tổng các phần tử trong thứ tự các phần tử, còn vt.begin() và vt.end() lần lượt là các trình lặp có tác dụng như con trỏ chỉ đến vị trí đầu tiên và cuối cùng của vector. Đối số cuối cùng (0) là giá trị khởi tạo của tổng.

Để sử dụng hàm std::accumulate, chúng ta cần include header file numeric.

VD

#include <iostream>

#include <numeric>

#include <vector>

using namespace std;

int main ()

{

    vector<int> vt{3, 1, 4, 2, 5};

    int sum = accumulate(vt.begin(), vt.end(), 0);

    cout << "SUM = " << sum << endl;

    return 0;

}

SUM = 15

### Tính tổng các phần tử trong vector C++ bằng vòng lặp

VD

#include <iostream>

#include <vector>

using namespace std;

int main()

{

    vector<int> vt{3, 1, 4, 2, 5};

    int sum = 0;

    for (auto x: vt) {

        sum += x;

    }

    cout << "SUM = " << sum << endl;

    return 0;

}

SUM = 15

## Vector trống

### Kiểm tra vector trống trong C++ bằng hàm empty

Cú pháp của hàm empty trong C++ như sau:

vt.empty();

Trong đó vt là vector cần kiểm tra.

Hàm empty sẽ trả về true nếu vector đã cho là vector trống, và trả về false, nếu vector đã cho có chứa phần tử.

VD

#include <iostream>

#include <vector>

using namespace std;

int main ()

{

    vector<int> vt;

    if( vt.empty() )

        std::cout << "empty.\n";

    else

        std::cout << "not empty.\n";

    vector<int> dq2{1, 2, 3};

    if( dq2.empty() )

        std::cout << "empty.\n";

    else

        std::cout << "not empty.\n";

}

empty.

not empty.

### Làm trống 1 vector trong C++ bằng hàm clear

Cú pháp của hàm clear trong C++ như sau:

vt.clear();

Trong đó vt là vector cần làm trống.

Lưu ý hàm vector clear sẽ làm trống vector chỉ định bằng việc xóa đi tất cả phần tử của nó. Tuy nhiên thì hàm vector clear sẽ không giải phóng bộ nhớ sử dụng cho việc lưu trữ dữ liệu đã dùng.

Trong trường hợp muốn xóa phần tử và giải phóng bộ nhớ, hãy dùng hàm shrink\_to\_fit() để thay thế.

VD

#include <iostream>

#include <vector>

using namespace std;

//Tạo hàm xuất vector

void dump(vector<int>& v)

{

    for (auto x: v) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    vector<int> vt{3, 1, 4};

    dump(vt);

    vt.clear();

    dump(vt);

}

3 1 4

### Giải phóng bộ nhớ trong C++ bằng hàm shrink\_to\_fit

Thông thường một vùng chứa vector có thể được cấp phát nhiều bộ nhớ hơn mức cần thiết để chứa các phần tử hiện tại của nó, nhằm tạo không gian trong mảng động giúp việc xử lý thêm chèn vector nhanh hơn.

Và hàm shrink\_to\_fit được tạo ra nhằm giúp chúng ta khi muốn reset lại vùng bộ nhớ và xóa đi các vùng cấp phát thừa. Lưu ý là hàm chỉ xóa các bộ nhớ cấp phát thừa, chứ không làm giảm kích thước tối thiểu của vùng chứa vector.

Hàm shrink\_to\_fit hay được sử dụng sau hàm clear(), sau khi bạn đã xóa tất cả các phần tử và cần phải giải phóng cả bộ nhớ đã dùng để chứa vector đó.

Cú pháp của hàm shrink\_to\_fit trong C++ như sau:

vt.shrink\_to\_fit();

Trong đó vt là vector cần tinh chỉnh và làm giảm bộ nhớ.

#include <iostream>

#include <vector>

using namespace std;

int main ()

{

    std::vector<int> vt{3, 1, 4};

    vt.clear();

    vt.shrink\_to\_fit();

}

# Deque

## Tổng quan

Deque trong C++ được viết tắt bởi cụm từ double-ended-queue, là một mảng động tương tự như vector nhưng lại có khả năng thêm xóa phần tử ở cả đầu lẫn cuối mảng đó với tốc độ cao O(1).

Do có khả năng xử lý cả 2 đầu của mảng động nên nó còn được gọi là queue kép (hàng đợi kép) và được sử dụng làm tiêu chuẩn để xử lý mảng động trong C++.

Tương tự như vector thì deque cũng được sử dụng để xử lý mảng động, nên khác với mảng truyền thống vốn có độ dài cố định thì độ dài của deque có thể được tự động thay đổi.

Về cách sử dụng cũng như các hàm thành viên trong deque thì hầu như là giống với vector trong C++, nên bạn hãy sử dụng deque thay cho vector trong các trường hợp cần xử lý ở đầu mảng động nhé.

Lưu ý, khác với vector thì các phần tử của deque không phải lúc nào cũng được lưu trữ tại các địa chỉ liên tiếp trong bộ nhớ, vì vậy việc lấy địa chỉ của một phần tử và chuyển nó dưới dạng một con trỏ đến một hàm khác có thể gây ra sự cố trong xử lý.

Ngoài ra, ngoại trừ ưu thế có thể chèn và xóa ở gần đầu mảng của deque thì hiệu suất của deque kém hơn vector, vì vậy nếu bạn không chèn và xóa thường xuyên gần đầu mảng, bạn nên sử dụng vector.

### std::deque trong C++

std::deque trong C++ là một class rất thuận tiện được sử dụng làm tiêu chuẩn để xử lý deque trong C++.

std::deque được cài sẵn trong header file deque và để sử dụng được chức năng này, chúng ta cần thêm dòng 「#include <deque> 」 vào đầu chương trình.

#include <deque>

int main()

{

    std::deque<int> dq1;

    std::deque<double> dq2;

}

Lại nữa, namespace của std::deque là std, do đó bằng cách khai báo sử dụng namespace này vào đầu chương trình mà chúng ta có thể viết gọn std::deque trong chương trình như sau:

#include <deque>

using namespace std;

int main()

{

   deque<int> dq1;

   deque<double> dq2;

}

## Khai báo deque trong C++

### Khai báo 1 deque trong C++

Cú pháp:

std::deque<type> dq;

Trong đó dq là tên biến deque và type là kiểu dữ liệu.

Lưu ý deque được khai báo với cú pháp này sẽ có 0 phần tử bên trong nó.

VD

#include <iostream>

#include <deque>

using namespace std;

int main()

{

    deque<double> name; //Khai báo deque name kiểu double

    deque<int> age;     //Khai báo deque age kiểu int

}

### Khai báo deque trong C++ bao gồm chỉ định số phần tử

Mặc dù kiểu deque trong C++ là loại mảng động có khả năng tự thay đổi kích thước, tuy nhiên trong trường hợp không chỉ định rõ kích thước của nó, thì việc xử lý nó trong chương trình sẽ tiêu tốn thời gian nhiều hơn so với khi biết kích thước cụ thể của nó. Bởi vậy nếu đã biết kích thước trước của một deque, chúng ta nên chỉ định độ dài (số phần tử) khi khai báo nó, với cú pháp sau đây:

std::deque<type> dq(length);

Trong đó length chính là độ dài(số phần tử() của deque cần tạo.

Với cách khai báo deque này thì các phần tử trong deque sẽ được gán giá trị mặc định tùy thuộc vào kiểu dữ liệu sử dụng. Ví dụ, nếu dùng kiểu số như int, double thì các phần tử sẽ có giá trị mặc định bằng 0 như sau:

VD

/\*Khai báo deque dq kiểu số\*/

std::deque<int> dq1(3);    //{0, 0, 0}

std::deque<double> dq2(3); //{0, 0, 0}

Tuy nhiên nếu dùng kiểu chữ như char, deque thì các phần tử sẽ có giá trị mặc định bằng Null được biểu diễn bởi trống khi in ra màn hình như sau:

VD

/\*Khai báo deque dq kiểu chữ\*/

std::deque<char> dq1(3)   //{, , }

std::deque<string> dq2(3) //{, , }

### Khai báo đồng thời nhiều deque trong C++

Cú pháp:

deque<type> name1, name2, name3, ... ;

VD

#include <iostream>

#include <deque>

using namespace std;

int main()

{

    deque<string> name, job, sex;

    deque<int> age;

}

## Gán deque trong C++

Cú pháp:

dq[index] = value ;

Trong đó dq là tên biến deque, index là vị trí phần tử cần gán giá trị và value là giá trị gán vào phần tử đó.

VD

#include <iostream>

#include <deque>

using namespace std;

int main()

{

    deque<int> dq(3);

    dq[0]=1;

    dq[1]=4;

    dq[2]=88;

}

## Khởi tạo deque trong C++

### Khởi tạo deque trong C++ với các phần tử riêng biệt

Cú pháp:

std::deque<type> dq {value1, value2, value3, ...};

Trong đó

* type là kiểu dữ liệu
* dq là tên biến deque
* value là các giá trị của deque

VD

std::deque<string> user{"Kiyoshi", "male", "Tokyo"};

//{"Kiyoshi", "male", "Tokyo"}

### Khởi tạo deque trong C++ với các phần tử giống nhau

Cú pháp:

std::deque<type> dq(length, value);

Trong đó length là số phần tử(độ dài), và value là giá trị sử dụng để khai báo đồng loạt các phần tử của deque cần tạo.

VD

//Khởi tạo deque dq kiểu int gồm 5 phần tử có giá trị bằng 8

std::deque<int> dq(5, 8); //{8, 8, 8, 8, 8}

### Khai báo deque 2 chiều trong C++

Cú pháp:

deque<deque<type> > dq {l1, l2, l3, ...};

Trong đó:

* dq là tên biến deque 2 chiều
* l là các deque 1 chiều được sử dụng như phần tử của deque 2 chiều

Lưu ý, chúng ta cần phải viết thêm dấu cách giữa cặp dấu > > khi khai báo deque 2 chiều. Lý do là để phân biệt với toán tử >> được sử dụng để dịch chuyển bit trong C++.

VD

#include <iostream>

#include <deque>

using namespace std;

int main()

{

    /\*Khai báo deque 2 chiều\*/

    deque<deque<string> > all\_user{

        {"Kiyoshi", "male", "Hanoi"},

        {"Honda", "male", "Tokyo"},

        {"Ajinomoto", "female", "Osaka"}};

    return 0;

}

Chúng ta cũng có thể khởi tạo các deque 1 chiều trước rồi dùng chúng để khai báo deque 2 chiều như sau:

#include <iostream>

#include <deque>

using namespace std;

int main()

{

    /\*Khởi tạo các deque 1 chiều làm phần tử trong deque 2 chiều\*/

    deque<string> user1{"Kiyoshi", "male", "Hanoi"};

    deque<string> user2{"Honda", "male", "Tokyo"};

    deque<string> user3{"Ajinomoto", "female", "Osaka"};

    /\*Khai báo deque 2 chiều\*/

    deque<deque<string> > all\_user{ user1, user2, user3};

    return 0;

}

## Truy cập phần tử trong deque C++

Cú pháp:

dq[index];

Trong đó dq là tên biến deque và index là vị trí của cần truy xuất.

VD

#include <iostream>

#include <deque>

using namespace std;

int main()

{

    deque<string> user{"Kiyoshi", "male", "Tokyo"};

    cout << user[0] <<endl;

    cout << user[1] <<endl;

    cout << user[2] <<endl;

    return 0;

}

Kiyoshi

male

Tokyo

## Duyệt deque trong C++

### Duyệt deque trong C++ bằng vòng lặp và index

Bằng cách sử dụng vòng lặp for và hàm size() để lấy số phần tử trong deque và chỉ định số vòng lặp, chúng ta có thể truy xuất lần lượt tới các phần tử trong deque và thực hiện việc duyệt deque trong C++, giống như ví dụ sau đây:

#include <iostream>

#include <deque>

using namespace std;

int main()

{

    deque<int> dq{5, 6, 88, -2};

    for (short i=0; i< dq.size();i++){

        cout << dq[i] <<endl;

    }

    return 0;

}

5

6

88

-2

### Duyệt deque trong C++ bằng vòng lặp dựa trên phạm vi

Cú pháp:

for ( auto& x : dq) {

// Xử lý

}

Trong đó:

* dq là tên deque
* x là tên một biến dùng để gán từng phần tử được lấy từ deque
* auto là kiểu suy luận giúp tự xác định kiểu dữ liệu của giá trị lấy từ deque

Truy cập vào phần tử của deque 1 chiều thông qua vòng lặp dựa trên phạm vi.

VD

#include <iostream>

#include <deque>

using namespace std;

int main()

{

    deque<string> user{"Kiyoshi", "male", "Tokyo"};

    //Sử dụng vòng lặp dựa trên phạm vi để truy cập phần tử của deque

    for (auto x: user) {

        cout << x << endl;

    }

    return 0;

}

Kiyoshi

male

Tokyo

Truy cập vào phần tử trong deque 2 chiều trong C++:

VD

#include <iostream>

#include <deque>

using namespace std;

int main()

{

    /\*Khởi tạo các deque 1 chiều làm phần tử trong deque 2 chiều\*/

    deque<string> user1{"Kiyoshi", "male", "Hanoi"};

    deque<string> user2{"Honda", "male", "Tokyo"};

    deque<string> user3{"Ajinomoto", "female", "Osaka"};

    /\*Khai báo deque 2 chiều\*/

    deque<deque<string> > all\_user{ user1, user2, user3};

    for (auto x: all\_user) {

        for (auto y: x) {

            cout << y << endl;

        }

    }

    return 0;

}

Honda

male

Tokyo

Ajinomoto

female

Osaka

### Duyệt deque trong C++ bằng iterator

Cú pháp:

for(auto itr = dq.begin(); itr != dq.end(); ++itr) {

cout << \*itr << endl;

}

Trong đó:

* dq là tên deque
* itr là tên iterator

VD

#include <iostream>

#include <deque>

using namespace std;

int main()

{

    deque<int> dq{5, 6, 88, -2};

    for(auto itr = dq.begin(); itr != dq.end(); ++itr) {

        cout << \*itr << endl;

    }

}

5

6

88

-2

## Lấy kích thước

### Lấy kích thước deque trong C++ bằng hàm size

Cú pháp của hàm size trong C++ như sau:

dq.size();

Trong đó dq là deque cần kiểm tra.

Hàm size sẽ trả về kích thước của deque, cũng chính là số phần tử có trong deque đó.

VD

#include <iostream>

#include <deque>

using namespace std;

int main ()

{

    deque<int> dq{3, 1, 4,5};

    cout<<dq.size()<<endl;

    return 0;

}

4

### Thay đổi kích thước deque trong C++ bằng hàm resize

Cú pháp của hàm resize trong C++ như sau:

dq.resizesize(n);

Trong đó dq là deque cần thay đổi kích thước, và n là kích thước cần thay đổi.

Nếu n nhỏ hơn kích thước(số phần tử) ban đầu của deque thì các phần tử bị thừa sẽ bị xóa đi. Ngược lại nếu n lớn hơn kích thước(số phần tử) ban đầu của deque, các giá trị mặc định tùy thuộc vào kiểu dữ liệu sẽ được thêm vào deque cho đủ độ dài. Ví dụ nếu deque thuộc kiểu số như int hay double thì số 0 sẽ được thêm vào. Còn nếu deque thuộc kiểu chữ như char hay deque, giá trị trống sẽ được thêm vào.

VD

#include <iostream>

#include <deque>

using namespace std;

//Tạo hàm xuất deque

void dump(deque<int>& v)

{

    for (auto x: v) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    deque<int> dq{3, 1, 4,5};

    dump(dq);

    cout<< "Size before: "<<dq.size()<<endl;

    /\*Thay đổi kích thước nhỏ hơn ban đầu\*/

    dq.resize(2);

    dump(dq);

    cout<< "Size after: "<<dq.size()<<endl;

    /\*Thay đổi kích thước lớn hơn ban đầu\*/

    dq.resize(10);

    dump(dq);

    cout<< "Size after: "<<dq.size()<<endl;

    return 0;

}

3 1 4 5

Size before: 4

3 1

Size after: 2

3 1 0 0 0 0 0 0 0 0

Size after: 10

## Cắt deque

### Cắt phần tử đầu tiên từ deque trong C++ bằng hàm front

Cú pháp:

dq.front();

Trong đó dq là deque cần cắt ra phần tử ban đầu.

Hàm front() sẽ trả về tham chiếu tới phần tử đầu tiên của deque, qua đó chúng ta có thể cắt lấy giá trị của phần tử đầu tiên này, hoặc là gán một phần tử khác để thay đổi nó.

Lưu ý chúng ta không dùng hàm front() cho deque trống.

VD

#include <iostream>

#include <deque>

using namespace std;

//Tạo hàm xuất deque

void dump(deque<int>& dq)

{

    for (auto x: dq) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main() {

    deque<int> dq{3, 1, 4, 1, 5, 7, 9};

    dump(dq);

    //Tạo tham chiếu tới vị trí phần tử đầu tiên của deque

    int front\_element = dq.front();

    cout << front\_element << endl;

    //Thay đổi phần tử đầu tiên

    dq.front() = 2;

    dump(dq);

    return 0;

}

3 1 4 1 5 7 9

3

2 1 4 1 5 7 9

### Cắt phần tử cuối cùng từ deque trong C++ bằng hàm back

Cú pháp:

dq.back();

Trong đó dq là deque cần cắt ra phần tử ban đầu.

Hàm back() sẽ trả về tham chiếu tới phần tử cuối cùng của deque, qua đó chúng ta có thể cắt lấy giá trị của phần tử cuối cùng này, hoặc là gán một phần tử khác để thay đổi nó.

Lưu ý chúng ta không dùng hàm back() cho deque trống.

VD

#include <iostream>

#include <deque>

using namespace std;

//Tạo hàm xuất deque

void dump(deque<int>& dq)

{

    for (auto x: dq) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main() {

    deque<int> dq{3, 1, 4, 1, 5, 7, 9};

    dump(dq);

    //Tạo tham chiếu tới vị trí phần tử cuối cùng của deque

    int back\_element = dq.back();

    cout << back\_element << endl;

    //Thay đổi phần tử cuối cùng

    dq.back() = 2;

    dump(dq);

    return 0;

}

3 1 4 1 5 7 9

3

2 1 4 1 5 7 9

### Cắt một phạm vi từ deque

Để cắt một phạm vi từ deque ban đầu, chúng ta sử dụng vòng lặp hoặc trình lặp để truy cập tuần tự vào các phần tử trong deque để lấy các giá trị trong phạm vi cần cắt. Sau đó, chúng ta lưu các giá trị này vào deque kết quả.

Và để thêm phần tử vào deque, chúng ta sẽ cần dùng đến hàm push\_back() trong C++ mà Kiyoshi đã hướng dẫn trong bài .

VD

#include <iostream>

#include <deque>

using namespace std;

//Tạo hàm xuất deque

void dump(deque<int>& dq)

{

    for (auto x: dq) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main() {

    deque<int> dq{3, 1, 4, 1, 5, 7, 9};

    dump(dq);

    //Khai báo deque chứa kết quả

    deque<int> dq2;

    //Khai báo phạm vi cắt

    int start = 2, end = 4;

    //Truy cập vào phạm vi để lấy và lưu các phần tử vào deque kết quả

    for (short i = start; i<= end;i++){

        dq2.push\_back(dq[i]);

    }

    dump(dq2);

    return 0;

}

3 1 4 1 5 7 9

4 1 5

## Thêm, chèn phần tử

### Thêm phần tử vào đầu deque C++ bằng hàm push\_front

Cú pháp:

dq.push\_front(value);

Trong đó dq là tên deque cần thêm phần tử value vào vị trí đầu tiên.

Nếu phần tử thêm vào có kiểu khác với kiểu của deque thì nó sẽ được chuyển về kiểu của deque trước khi được thêm vào.

Hàm push\_front thuộc kiểu void, do đó nó sẽ không trả giá trị. Bởi vậy nó sẽ thêm phần tử vào vị trí đầu tiên trong deque chứ không trả về chuỗi kết quả.

VD

#include <iostream>

#include <deque>

using namespace std;

//Tạo hàm xuất deque

void dump(deque<int>& dq)

{

    for (auto x: dq) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main() {

    deque<int> dq{3, 1, 4, 1, 5, 7, 9};

    dump(dq);

    dq.push\_front(88);

    dump(dq);

    return 0;

}

3 1 4 1 5 7 9

88 3 1 4 1 5 7 9

Lưu ý, nếu phần tử thêm vào có kiểu khác với kiểu của deque thì nó sẽ được chuyển về kiểu của deque trước khi được thêm vào. Ví dụ nếu thêm một ký tự vào một deque thuộc kiểu int, thì ký tự đó sẽ được chuyển về mã Unicode trước khi được thêm vào deque ban đầu như sau:

deque<int> dq{3, 1, 4, 1, 5, 7, 9};

dump(dq);

// 3 1 4 1 5 7 9

dq.push\_front('a');

dump(dq);

// 97 3 1 4 1 5 7 9

Do mã Unicode của ký tự ‘a’ là 97 nên nó đã được chuyển về dạng int là 97, rồi mới được thêm vào deque ban đầu như trên.

### Thêm phần tử vào cuối deque C++ bằng hàm push\_back

Cú pháp:

dq.push\_back(value);

Trong đó dq là tên deque cần thêm phần tử value vào vị trí cuối cùng.

Nếu phần tử thêm vào có kiểu khác với kiểu của deque thì nó sẽ được chuyển về kiểu của deque trước khi được thêm vào.

Hàm push\_back thuộc kiểu void, do đó nó sẽ không trả giá trị. Bởi vậy nó sẽ thêm phần tử vào vị trí cuối cùng trong deque chứ không trả về chuỗi kết quả.

VD

#include <iostream>

#include <deque>

using namespace std;

//Tạo hàm xuất deque

void dump(deque<int>& dq)

{

    for (auto x: dq) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main() {

    deque<int> dq{3, 1, 4, 1, 5, 7, 9};

    dump(dq);

    dq.push\_back(88);

    dump(dq);

    return 0;

}

3 1 4 1 5 7 9

3 1 4 1 5 7 9 88

Lưu ý, nếu phần tử thêm vào có kiểu khác với kiểu của deque thì nó sẽ được chuyển về kiểu của deque trước khi được thêm vào. Ví dụ nếu thêm một ký tự vào một deque thuộc kiểu int, thì ký tự đó sẽ được chuyển về mã Unicode trước khi được thêm vào deque ban đầu như sau:

deque<int> dq{3, 1, 4, 1, 5, 7, 9};

dump(dq);

// 3 1 4 1 5 7 9

dq.push\_back('a');

dump(dq);

// 3 1 4 1 5 7 9 97

Do mã Unicode của ký tự ‘a’ là 97 nên nó đã được chuyển về dạng int là 97, rồi mới được thêm vào deque ban đầu như trên.

### Thêm phần tử vào đầu deque C++ bằng hàm emplace\_front

Khác với hàm push\_front sẽ thêm phần tử bằng cách copy một đối tượng hoặc giá trị vào trong deque thì hàm emplace\_front sẽ sử dụng hàm tạo của kiểu giá trị và chèn trực tiếp vào đầu deque.

Và do hàm emplace\_back sẽ tạo trực tiếp phần tử bên trong vector mà không cần phải qua việc copy (di chuyển) giá trị vào trong vector, nên nó được cho là có tốc độ xử lý nhanh hơn hàm push\_front.

Cú pháp sử dụng hàm emplace\_front để thêm phần tử vào đầu deque như sau:

dq.emplace\_front(value);

Trong đó dq là tên deque cần thêm phần tử value vào vị trí đầu tiên.

Nếu phần tử thêm vào có kiểu khác với kiểu của deque thì nó sẽ được chuyển về kiểu của deque trước khi được thêm vào.

Hàm emplace\_front thuộc kiểu void, do đó nó sẽ không trả giá trị. Bởi vậy nó sẽ thêm phần tử vào vị trí đầu tiên trong deque chứ không trả về chuỗi kết quả.

VD

#include <iostream>

#include <deque>

using namespace std;

//Tạo hàm xuất deque

void dump(deque<int>& dq)

{

    for (auto x: dq) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main() {

    deque<int> dq{3, 1, 4, 1, 5, 7, 9};

    dump(dq);

    dq.emplace\_front(88);

    dump(dq);

    return 0;

}

3 1 4 1 5 7 9

88 3 1 4 1 5 7 9

### Thêm phần tử vào cuối deque C++ bằng hàm emplace\_back

Hàm emplace\_back là một hàm thành viên trong class std:deque, có tác dụng thêm một phần tử vào vị trí cuối cùng trong deque cũng như tăng độ dài của nó thêm một đơn vị, bằng cách sử dụng hàm tạo của kiểu của giá trị được thêm.

Khác với hàm push\_back sẽ thêm phần tử bằng cách copy một đối tượng hoặc giá trị vào trong deque thì hàm emplace\_back sẽ sử dụng hàm tạo của kiểu giá trị và chèn trực tiếp vào cuối deque.

Và do hàm emplace\_back sẽ tạo trực tiếp phần tử bên trong vector mà không cần phải qua việc copy (di chuyển) giá trị vào trong vector, nên nó được cho là có tốc độ xử lý nhanh hơn hàm push\_back.

Cú pháp sử dụng hàm emplace\_back để thêm phần tử vào cuối deque như sau:

dq.emplace\_back(value);

Trong đó dq là tên deque cần thêm phần tử value vào vị trí cuối cùng. Nếu phần tử thêm vào có kiểu khác với kiểu của deque thì nó sẽ được chuyển về kiểu của deque trước khi được thêm vào.

Hàm emplace\_back thuộc kiểu void, do đó nó sẽ không trả giá trị. Bởi vậy nó sẽ thêm phần tử vào vị trí cuối cùng trong deque chứ không trả về chuỗi kết quả.

VD

#include <iostream>

#include <deque>

using namespace std;

//Tạo hàm xuất deque

void dump(deque<int>& dq)

{

    for (auto x: dq) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main() {

    deque<int> dq{3, 1, 4, 1, 5, 7, 9};

    dump(dq);

    dq.emplace\_back(88);

    dump(dq);

    return 0;

}

3 1 4 1 5 7 9

3 1 4 1 5 7 9 88

### Chèn 1 phần tử vào vị trí chỉ định trong deque C++ bằng hàm insert

Cú pháp:

dq.insert(p, value);

Trong đó dq là deque ban đầu, value là phần tử cần chèn, và p là trình lặp trỏ đến vị trí cần chèn trong deque.

Trong trường hợp cần chỉ đến vị trí index thứ n trong deque, chúng ta sẽ viết trình lặp p như sau:

dq.insert(dq.begin() + i, value);

Trong đó dq.begin() chỉ đến vị trí đầu tiên trong deque, và i là index của vị trí phần tử cần chỉ đến.

VD

#include <iostream>

#include <deque>

using namespace std;

//Tạo hàm xuất deque

void dump(deque<int>& dq)

{

    for (auto x: dq) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

  deque<int> dq{3, 1, 4, 1, 5, 7, 9};

  dq.insert(dq.begin()+2, 55); //Thêm phần tử 55 vào vị trí thứ 2

  dump(dq);

  dq.insert(dq.begin()+4, 22); //Thêm phần tử 22 vào vị trí thứ 4

  dump(dq);

  return 0;

}

3 1 55 4 1 5 7 9

3 1 55 4 22 1 5 7 9

## Xoá phần tử

### Xóa phần tử đầu tiên trong deque C++ bằng pop\_front

Cú pháp:

dq.pop\_front();

Trong đó dq là tên deque cần xóa phần tử đầu tiên.

VD

#include <iostream>

#include <deque>

using namespace std;

//Tạo hàm xuất deque

void dump(deque<int>& dq)

{

    for (auto x: dq) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    deque<int> dq{3, 1, 4, 1, 5};

    cout << "Before erase" << endl;

    dump (dq);

    dq.pop\_front();

    cout << "After erase" << endl;

    dump (dq);

    return 0;

}

Before erase

3 1 4 1 5

After erase

1 4 1 5

Hàm pop\_front thuộc kiểu void, do đó nó sẽ không trả giá trị. Bởi vậy nó sẽ xóa phần tử đầu tiên trong deque chứ không trả về phần tử đó. Trong trường hợp muốn lấy phần tử đầu tiên trong deque, hãy sử dụng tới hàm front để thay thế.

deque<int> dq{3, 1, 4, 1, 5};

int last = dq.front();  //3

dq.pop\_front();         //{1, 4, 1, 5}

### Xóa phần tử cuối cùng trong deque C++ bằng pop\_back

Cú pháp:

dq.pop\_back();

Trong đó dq là tên deque cần xóa phần tử cuối cùng.

VD

#include <iostream>

#include <deque>

using namespace std;

//Tạo hàm xuất deque

void dump(deque<int>& dq)

{

    for (auto x: dq) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    deque<int> dq{3, 1, 4, 1, 5};

    cout << "Before erase" << endl;

    dump (dq);

    dq.pop\_back();

    cout << "After erase" << endl;

    dump (dq);

    return 0;

}

Before erase

3 1 4 1 5

After erase

3 1 4 1

Hàm pop\_back thuộc kiểu void, do đó nó sẽ không trả giá trị. Bởi vậy nó sẽ xóa phần tử cuối cùng trong deque chứ không trả về phần tử đó. Trong trường hợp muốn lấy phần tử cuối cùng trong deque, hãy sử dụng tới hàm back để thay thế.

VD

deque<int> dq{3, 1, 4, 1, 5};

int last = dq.back();  //5

dq.pop\_back();         //{3, 1, 4, 1}

### Xóa 1 phần tử trong deque bằng erase c++

Cú pháp:

dq.erase(itr);

Trong đó dq là deque ban đầu, và itr là trình lặp trỏ đến vị trí cần xóa trong deque.

Trong trường hợp cần chỉ đến vị trí index thứ n trong deque, chúng ta sẽ viết trình lặp p như sau:

dq.erase(dq.begin() + i);

Trong đó dq.begin() chỉ đến vị trí đầu tiên trong deque, và i là index của vị trí phần tử cần chỉ đến.

VD

#include <iostream>

#include <deque>

using namespace std;

//Tạo hàm xuất deque

void dump(deque<int>& dq)

{

    for (auto x: dq) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    deque<int> dq{3, 1, 4, 1, 5};

    dq.erase(dq.begin() + 1);//Xóa phần tử ở vị trí thứ 1

    dump(dq);

    dq.erase(dq.begin() + 3);//Xóa phần tử ở vị trí thứ 3

    dump(dq);

    return 0;

}

3 4 1 5

3 4 1

### Xóa các phần tử trong một phạm vi chỉ định bằng deque erase c++

Cú pháp:

dq.erase( iterator\_first, iterator\_last);

Trong đó dq là deque ban đầu, iterator\_first và iterator\_last là các trình lặp trỏ đến phạm vi bắt đầu và kết thúc xóa.

Tương tự như khi xóa 1 phần tử thì cú pháp sử dụng thực tế để xóa các phần tử trong một phạm vi chỉ định trong deque C++ bằng hàm erase sẽ là:

dq.erase(dq.begin() + start, dq.begin() + end);

Trong đó start và end là vị trí index của phạm vi xóa trong deque.

VD

#include <iostream>

#include <deque>

using namespace std;

//Tạo hàm xuất deque

void dump(deque<int>& dq)

{

    for (auto x: dq) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    deque<int> dq{3, 1, 4, 1, 5, 7, 9};

    dump(dq);

    //Xóa phần tử trong phạm vi index từ 1 đến 3

    dq.erase(dq.begin() + 1, dq.begin() + 3);

    dump(dq);

    //Xóa phần tử trong phạm vi index từ 2 đến 5

    dq.erase(dq.begin() + 2, dq.begin() +5);

    dump(dq);

    return 0;

}

3 1 4 1 5 7 9

3 1 5 7 9

3 1

## Sao chép và hoán đổi

### Sao chép deque trong C++

Deque trong C++ thuộc kiểu dữ liệu đối tượng, do vậy khác với các kiểu dữ liệu nguyên thủy, chúng ta không thể sử dụng toán tử bằng = để gán và sao chép một deque vào một deque mới.

Thay vào đó, chúng ta sẽ sử dụng cách copy constructor trong deque với cú pháp như sau:

std::deque<type> dq\_dest(dq\_src);

Trong đó type là kiểu dữ liệu, dq\_src là deque nguồn để copy và dq\_dest là deque đích dùng để dán kết quả sao chép.

VD

#include <iostream>

#include <deque>

using namespace std;

//Tạo hàm xuất deque

void dump(deque<int>& v)

{

    for (auto x: v) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main()

{

    deque<int> dq\_src{3, 1, 4,5};

    cout << "Orginary deque" << endl;

    dump (dq\_src);

    /\*Sao chép deque\*/

    deque<int> dq\_dest(dq\_src);

    cout << "Copy deque" << endl;

    dump (dq\_dest);

    return 0;

}

Orginary deque

3 1 4 5

Copy deque

3 1 4 5

### Hoán đổi 2 deque trong C++

Deque swap trong C++ là một hàm thành viên trong class std:deque, có tác dụng hoán đổi 2 deque trong C++.

Hàm swap sẽ hoán đổi toàn bộ nội dung của 2 deque đã cho cho nhau và làm thay đổi nội dung cũng như độ dài của chúng.

Cú pháp hàm Deque swap để hoán đổi 2 deque trong C++ như sau:

dq1.swap(dq2);

Trong đó dq1 và dq2 là 2 deque cần hoán đổi nội dung cho nhau.

VD

#include <iostream>

#include <deque>

using namespace std;

//Tạo hàm xuất deque

void dump(deque<int>& v)

{

    for (auto x: v) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main()

{

    deque<int> dq1{3, 1, 4, 1, 5};

    deque<int> dq2{9, 8, 7};

    cout << "Before swap" << endl;

    dump(dq1);

    dump(dq2);

    dq1.swap(dq2);

    cout << "After swap" << endl;

    dump(dq1);

    dump(dq2);

}

Before swap

3 1 4 1 5

9 8 7

After swap

9 8 7

3 1 4 1 5

Ngoài cách dùng hàm deque swap, chúng ta cũng có thể dùng function template là std::swap để tiến hành hoán đổi 2 deque với nhau, cũng như là để hoán đổi các đối tượng khác như map, vector trong C++.

Lưu ý chúng ta cần phải thêm header file utility vào trong chương trình để có thể sử dụng được function template này

VD

#include <iostream>

#include <utility>

#include <deque>

using namespace std;

//Tạo hàm xuất deque

void dump(deque<int>& v)

{

    for (auto x: v) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main()

{

    deque<int> dq1{3, 1, 4, 1, 5};

    deque<int> dq2{9, 8, 7};

    cout << "Before swap" << endl;

    dump(dq1);

    dump(dq2);

    swap(dq1,dq2);

    cout << "After swap" << endl;

    dump(dq1);

    dump(dq2);

}

Before swap

3 1 4 1 5

9 8 7

After swap

9 8 7

3 1 4 1 5

## Đảo ngược

### Đảo ngược trong deque C++ bằng std::reverse

Để sử dụng hàm std::reverse, chúng ta cần include header file algorithm vào đầu chương trình như sau:

Cú pháp:

std::reverse(dq.begin(), dq.end());

Trong đó dq là tên deque cần đảo ngược thứ tự các phần tử, còn dq.begin() và dq.end() lần lượt là các trình lặp có tác dụng như con trỏ chỉ đến vị trí đầu tiên và cuối cùng của deque.

VD

#include <iostream>

#include <algorithm>

#include <deque>

using namespace std;

//Tạo hàm xuất deque

void dump(deque<int>& dq)

{

    for (auto x: dq) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    deque<int> dq{3, 1, 4, 2, 5};

    cout << "Before reverse" << endl;

    dump (dq);

    reverse(dq.begin(), dq.end());

    cout << "After reverse" << endl;

    dump (dq);

    return 0;

}

Before reverse

3 1 4 2 5

After reverse

5 2 4 1 3

## Xắp xếp

### Sắp xếp deque tăng dần bằng hàm sort deque trong C++

Để sử dụng hàm sort, chúng ta cần include header file algorithm.

Cú pháp:

std::sort(dq.begin(), dq.end() );

Trong đó dq là deque cần sắp xếp theo thứ tự tăng dần. Các phương thức begin() và end() được dùng để lấy index đầu tiên và cuối cùng trong deque, tương ứng với phạm vi sẽ tiến hành sắp xếp.

VD

#include <iostream>

#include <algorithm>

#include <deque>

using namespace std;

//Tạo hàm xuất deque

void dump(deque<int>& v)

{

    for (auto x: v) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main()

{

    deque<int> dq{3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};

    cout << "Before sort" << endl;

    dump (dq);

    /\*Sắp xếp deque theo thứ tự tăng dần\*/

    sort(dq.begin(), dq.end() );

    cout << "After sort" << endl;

    dump (dq);

    return 0;

}

Before sort

3 1 4 1 5 9 2 6 5 3 5

After sort

1 1 2 3 3 4 5 5 5 6 9

### Sắp xếp deque giảm dần bằng hàm sort deque trong C++

Cú pháp:

std::sort(dq.begin(), dq.end(), std::greater() );

Sự khác biệt duy nhất với cú pháp khi sắp xếp deque theo thứ tự tăng dần bằng hàm sort đó là đối số std::greater<type> có tác dụng chuyển từ sắp xếp tăng dần thành giảm dần, trong đó type là kiểu dữ liệu sử dụng trong deque.

VD

#include <iostream>

#include <algorithm>

#include <deque>

using namespace std;

//Tạo hàm xuất deque

void dump(deque<int>& dq)

{

    for (auto x: dq) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main()

{

    deque<int> dq{3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};

    cout << "Before sort" << endl;

    dump (dq);

    /\*Sắp xếp deque theo thứ tự giảm dần\*/

    sort(dq.begin(), dq.end(), std::greater<int>() );

    cout << "After sort" << endl;

    dump (dq);

    return 0;

}

Before sort

3 1 4 1 5 9 2 6 5 3 5

After sort

9 6 5 5 5 4 3 3 2 1 1

## Tính tổng

### Tính tổng các phần tử trong deque C++ bằng std::accumulate

Để sử dụng hàm std::accumulate, chúng ta cần include header file numeric.

Cú pháp:

std::accumulate(dq.begin(), dq.end(), 0);

Trong đó dq là tên deque cần tính tổng các phần tử trong thứ tự các phần tử, còn dq.begin() và dq.end() lần lượt là các trình lặp có tác dụng như con trỏ chỉ đến vị trí đầu tiên và cuối cùng của deque. Đối số cuối cùng (0) là giá trị khởi tạo của tổng.

VD

#include <iostream>

#include <numeric>

#include <deque>

using namespace std;

int main ()

{

    deque<int> dq{3, 1, 4, 2, 5};

    int sum = accumulate(dq.begin(), dq.end(), 0);

    cout << "SUM = " << sum << endl;

    return 0;

}

SUM = 15

### Tính tổng các phần tử trong deque C++ bằng vòng lặp

VD

#include <iostream>

#include <deque>

using namespace std;

int main()

{

    deque<int> dq{3, 1, 4, 2, 5};

    int sum = 0;

    for (auto x: dq) {

        sum += x;

    }

    cout << "SUM = " << sum << endl;

    return 0;

}

SUM = 15

## Deque trống

### Kiểm tra deque trống trong C++ bằng hàm empty

Cú pháp của hàm empty trong C++ như sau:

dq.empty();

Trong đó dq là deque cần kiểm tra.

Hàm empty sẽ trả về true nếu deque đã cho là deque trống, và trả về false, nếu deque đã cho có chứa phần tử.

VD

#include <iostream>

#include <deque>

using namespace std;

int main ()

{

    deque<int> dq;

    if( dq.empty() )

        std::cout << "empty.\n";

    else

        std::cout << "not empty.\n";

    deque<int> dq2{1, 2, 3};

    if( dq2.empty() )

        std::cout << "empty.\n";

    else

        std::cout << "not empty.\n";

    return 0;

}

empty.

not empty.

### Làm trống 1 deque trong C++ bằng hàm clear

Cú pháp của hàm clear trong C++ như sau:

dq.clear();

Trong đó dq là deque cần làm trống.

Lưu ý hàm deque clear sẽ làm trống deque chỉ định bằng việc xóa đi tất cả phần tử của nó. Tuy nhiên thì hàm deque clear sẽ không giải phóng bộ nhớ sử dụng cho việc lưu trữ dữ liệu đã dùng.

Trong trường hợp muốn xóa phần tử và giải phóng bộ nhớ, hãy dùng hàm shrink\_to\_fit() để thay thế.

VD

#include <iostream>

#include <deque>

using namespace std;

//Tạo hàm xuất deque

void dump(deque<int>& v)

{

    for (auto x: v) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    deque<int> dq{3, 1, 4};

    dump(dq);

    dq.clear();

    dump(dq);

}

3 1 4

### Giải phóng bộ nhớ trong C++ bằng hàm shrink\_to\_fit

Hàm shrink\_to\_fit được tạo ra nhằm giúp chúng ta khi muốn reset lại vùng bộ nhớ và xóa đi các vùng cấp phát thừa. Lưu ý là hàm chỉ xóa các bộ nhớ cấp phát thừa, chứ không làm giảm kích thước tối thiểu của vùng chứa deque.

Hàm shrink\_to\_fit hay được sử dụng sau hàm clear(), sau khi bạn đã xóa tất cả các phần tử và cần phải giải phóng cả bộ nhớ đã dùng để chứa deque đó.

Cú pháp của hàm shrink\_to\_fit trong C++ như sau:

dq.shrink\_to\_fit();

Trong đó dq là deque cần tinh chỉnh và làm giảm bộ nhớ.

VD

#include <iostream>

#include <deque>

using namespace std;

int main ()

{

    deque<int> dq{3, 1, 4};

    dq.clear();

    dq.shrink\_to\_fit();

}

# List

## Tổng quan

List trong C++ là một danh sách liên kết đôi được sử dụng làm tiêu chuẩn để xử lý các đối tượng chứa nhiều phần tử trong C++.

Danh sách liên kết đôi ở đây có nghĩa là từng Node trong danh sách sẽ chứa thông tin vị trí của Node đứng trước và sau nó, nên việc xác định vị trí của một Node trong danh sách sẽ trở nên dễ dàng hơn với độ phức tạp của thuật toán sẽ là O(1) mà thôi.

Điều đó tạo ra ưu điểm của list trong C++ đó là, việc chèn và xóa một vị trí bất kỳ trong list có thể được thực hiện trong một thời gian cố định với tốc độ cao O(1).

Tuy nhiên thì ngược lại cũng tạo ra nhược điểm của list trong C++, đó là chúng ta không thể thực hiện việc truy cập vào vị trí ngẫu nhiên trong list. Do vậy để truy cập vào một vị trí bất kỳ trong list thì chương trình luôn phải truy cập tuần tự bằng một trình lặp, khiến tốc độ xử lý sẽ trở thành O(n).

Tương tự như vector trong C++ thì list cũng có khả năng tự thay đổi kích thước sau khi được tạo ra.

Tuy nhiên so với list thì vector lại có ưu thế trong việc truy cập vào vị trí ngẫu nhiên với tốc độ cao.

Nói tóm lại thì quan hệ giữa vector và list là đối lập nhau, và tùy vào nhu cầu mà chúng ta quyết định dùng list hay vector để xử lý các đối tượng chứa nhiều phần tử trong C++.

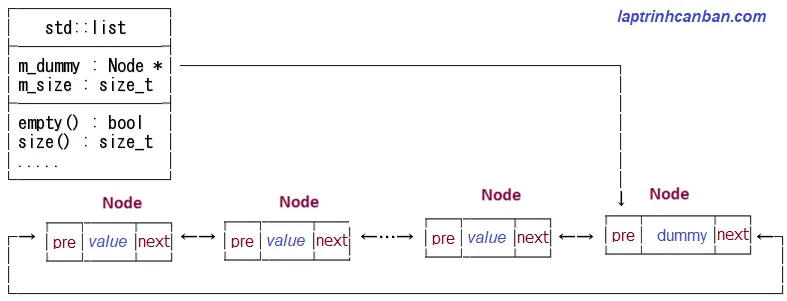
Loại Truy cập ngẫu nhiên Thêm xóa ngẫu nhiên

vector O(1) O(N)

list O(N) O(1)

### Cấu trúc dữ liệu của list trong C++

Cấu trúc dữ liệu của list trong C++ như hình sau:



Khác với vector (với các phần tử được lưu trong một mảng động) thì phần tử trong list và forward\_list lại được lưu trong các phân vùng bộ nhớ độc lập được gọi là Node. Và những node này được liên kết tuần tự với nhau thông qua con trỏ của chúng.

Với forward\_list, mỗi node sẽ lưu trữ con trỏ chỉ tới node sau nó, trong khi với list, con trỏ của cả các node trước và sau node đó đều được lưu giữ. Vì thế, list còn được gọi là danh sách liên kết đôi trong C++.

### std::list trong C++

std::list trong C++ là một thư viện chuẩn được sử dụng làm tiêu chuẩn để xử lý danh sách liên kết đôi trong C++.

std::list được cài sẵn trong header file list và để sử dụng được chức năng này, chúng ta cần thêm dòng 「#include <list> 」 vào đầu chương trình.

## Khai báo list trong C++

### Khai báo 1 list trong C++

Cú pháp:

std::list<type> lst;

Trong đó lst là tên biến list và type là kiểu dữ liệu.

VD

#include <iostream>

#include <list>

using namespace std;

int main()

{

    list<double> name; //Khai báo list name kiểu double

    list<int> age;     //Khai báo list age kiểu int

}

### Khai báo list trong C++ bao gồm chỉ định số phần tử

Mặc dù kiểu list trong C++ là loại mảng động có khả năng tự thay đổi kích thước, tuy nhiên trong trường hợp không chỉ định rõ kích thước của nó, thì việc xử lý nó trong chương trình sẽ tiêu tốn thời gian nhiều hơn so với khi biết kích thước cụ thể của nó. Bởi vậy nếu đã biết kích thước trước của một list, chúng ta nên chỉ định độ dài (số phần tử) khi khai báo nó, với cú pháp sau đây:

std::list<type> lst(length);

Trong đó length chính là độ dài(số phần tử() của list cần tạo.

Với cách khai báo list này thì các phần tử trong list sẽ được gán giá trị mặc định tùy thuộc vào kiểu dữ liệu sử dụng. Ví dụ, nếu dùng kiểu số như int, double thì các phần tử sẽ có giá trị mặc định bằng 0 như sau:

/\*Khai báo list lst kiểu số\*/

std::list<int> lst1(3);    //{0, 0, 0}

std::list<double> lst2(3); //{0, 0, 0}

Tuy nhiên nếu dùng kiểu chữ như char, string thì các phần tử sẽ có giá trị mặc định bằng Null được biểu diễn bởi trống khi in ra màn hình như sau:

/\*Khai báo list lst kiểu chữ\*/

std::list<char> lst1(3)   //{, , }

std::list<string> lst2(3) //{, , }

### Khai báo đồng thời nhiều list trong C++

Cú pháp:

list<type> name1, name2, name3, ... ;

VD

#include <iostream>

#include <list>

using namespace std;

int main()

{

    list<string> name, job, sex;

    list<int> age;

}

## Khởi tạo list trong C++

### Khởi tạo list trong C++ với các phần tử riêng biệt

Cú pháp:

std::list<type> lst {value1, value2, value3, ...};

Trong đó

* type là kiểu dữ liệu
* lst là tên biến list
* value là các giá trị của list

VD

std::list<string> user{"Kiyoshi", "male", "Tokyo"};

//{"Kiyoshi", "male", "Tokyo"}

### Khởi tạo list trong C++ với các phần tử giống nhau

Cú pháp:

std::list<type> lst(length, value);

Trong đó length là số phần tử(độ dài), và value là giá trị sử dụng để khai báo đồng loạt các phần tử của list cần tạo.

VD

//Khởi tạo list lst kiểu int gồm 5 phần tử có giá trị bằng 8

std::list<int> lst(5, 8); //{8, 8, 8, 8, 8}

### Khai báo list 2 chiều trong C++

Cú pháp:

using namespace std;

list<list<type> > lst {l1, l2, l3, ...};

Trong đó:

* lst là tên biến list 2 chiều
* l là các list 1 chiều được sử dụng như phần tử của list 2 chiều

Lưu ý, chúng ta cần phải viết thêm dấu cách giữa cặp dấu > > khi khai báo list 2 chiều. Lý do là để phân biệt với toán tử >> được sử dụng để dịch chuyển bit trong C++.

VD

#include <iostream>

#include <list>

using namespace std;

int main()

{

    /\*Khai báo list 2 chiều\*/

    list<list<string> > all\_user{

        {"Kiyoshi", "male", "Hanoi"},

        {"Honda", "male", "Tokyo"},

        {"Ajinomoto", "female", "Osaka"}};

    return 0;

}

Chúng ta cũng có thể khởi tạo các list 1 chiều trước rồi dùng chúng để khai báo list 2 chiều như sau:

VD

#include <iostream>

#include <list>

using namespace std;

int main()

{

    /\*Khởi tạo các list 1 chiều làm phần tử trong list 2 chiều\*/

    list<string> user1{"Kiyoshi", "male", "Hanoi"};

    list<string> user2{"Honda", "male", "Tokyo"};

    list<string> user3{"Ajinomoto", "female", "Osaka"};

    /\*Khai báo list 2 chiều\*/

    list<list<string> > all\_user{ user1, user2, user3};

    return 0;

}

## Truy cập phần tử trong list C++

Khác với vector hay mảng, chúng ta không thể truy cập ngẫu nhiên vào phần tử bất kỳ trong một list. Do đó chúng ta cũng không thể sử dụng index của các phần tử để truy cập vào nó theo cách thông thường được.

Thay vào đó, chúng ta cần phải tiến hành truy cập tuần tự vào các phần tử của list, thông qua vòng lặp hoặc là trình lặp mà Kiyoshi đã giới thiệu trong bài Duyệt list trong C++.

VD

#include <iostream>

#include <list>

using namespace std;

int main()

{

    list<string> user{"Kiyoshi", "male", "Tokyo"};

    //Sử dụng vòng lặp dựa trên phạm vi để truy cập phần tử của list

    for (auto x: user) {

        cout << x << endl;

    }

    return 0;

}

Kiyoshi

male

Tokyo

Tượng tự khi chúng ta cần truy cập vào phần tử trong list 2 chiều trong C++:

#include <iostream>

#include <list>

using namespace std;

int main()

{

    /\*Khởi tạo các list 1 chiều làm phần tử trong list 2 chiều\*/

    list<string> user1{"Kiyoshi", "male", "Hanoi"};

    list<string> user2{"Honda", "male", "Tokyo"};

    list<string> user3{"Ajinomoto", "female", "Osaka"};

    /\*Khai báo list 2 chiều\*/

    list<list<string> > all\_user{ user1, user2, user3};

    for (auto x: all\_user) {

        for (auto y: x) {

            cout << y << endl;

        }

    }

    return 0;

}

Honda

male

Tokyo

Ajinomoto

female

Osaka

### Vector vs list trong C++

Như đã phân tích ở trên thì sự khác biệt lớn nhất giữa vector và list trong C++ đó chính là ở tốc độ xử lý khi truy cập ngẫu nhiên và thêm xóa ngẫu nhiên các vị trí trong đối tượng.

List là danh sách liên kết đôi (double linked list) với từng Node trong nó chứa các thông tin vị trí của các Node đứng trước và sau nó, nên nó có thể thêm xóa ngẫu nhiên phần tử với tốc độ cao, nhưng lại không có khả năng truy cập ngẫu nhiên phần tử.

Ngược lại vector là mảng động (dynamic array) với khả năng truy cập ngẫu nhiên phần tử với tốc độ cao, nhưng lại chỉ có thể thêm xóa phần tử ở cuối vector mà thôi.

Tuy vậy, với các đối tượng có ít phần tử (từ 10 phần tử trở xuống) thì việc thêm xóa phần tử bằng vector cũng OK nên thông thường chúng ta sử dụng vector là đủ rồi.

Chỉ đối với các đối tượng có nhiều hơn dữ liệu khiến ảnh hưởng tới tốc độ xử lý, thì chúng ta mới cần cân nhắc lựa chọn việc sử dụng giữa vector và list mà thôi.

# Tham khảo

Lập trình căn bản:

<https://laptrinhcanban.com/cpp/lap-trinh-cpp-co-ban/>