(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 1](#_Toc126827019)

[Set 2](#_Toc126827020)

[Tham khảo 3](#_Toc126827021)

# Set

## Tổng quan

### Set trong c++ là gì

Set trong C++ là một tập hợp các phần tử duy nhất được sắp xếp theo thứ tự cụ thể, và được sử dụng làm tiêu chuẩn để xử lý các đối tượng chứa nhiều phần tử trong C++.

Mỗi phần tử trong set có giá trị phải là duy nhất, có nghĩa là nó không được trùng lặp với các giá trị còn lại trong set. Ngoài ra thì phần tử trong set không thể thay đổi giá trị, tuy nhiên chúng có thể được chèn hoặc xóa khỏi set.

Về mặt nội bộ, các phần tử trong set luôn được sắp xếp theo thứ tự cụ thể một cách nghiêm ngặt, được chỉ ra bởi đối tượng so sánh nội bộ của nó. Nếu bạn thêm các phần tử mới không theo thứ tự cụ thể vào một set, chúng sẽ tự động sắp xếp lại theo giá trị trước khi được lưu trữ nội bộ.

Trong C++ cũng có một loại dữ liệu khá giống với set là multiset khi các phần tử trong chúng luôn được sắp xếp theo thứ tự. Tuy nhiên thì khác với multiset vốn cho phép các phần tử có thể trùng nhau cùng tồn tại, thì các phần tử trong set không được trùng nhau và luôn phải là duy nhất.

Nói tóm lại thì set trong C+++ sẽ giống như một mảng với các phần tử không trùng lặp và luôn được sắp xếp.

Về mặt tốc độ xử lý thì set có khả năng thêm, xóa, tìm kiếm dữ liệu với tốc độ cực cao là O(log N), và nó còn cao hơn cả vector với tốc là O(1). Tuy nhiên thì do các phần tử được quản lý dạng cây nhị phân nên tốc độ truy cập ngẫu nhiên tới một phần tử chỉ định trong set lại cực thấp.

Do đó, trong trường hợp chúng ta không hay thêm xóa tìm kiếm phần tử thì việc dùng vector sẽ có lợi hơn, do tốc độ cũng tương đương mà lại tiết kiệm bộ nhớ.

Tuy nhiên trong các trường hợp không cần thiết việc truy cập ngẫu nhiên và hay thêm xóa tìm kiếm phần tử thì việc sử dụng set thay cho vector sẽ có lợi nhiều hơn.

Loại Truy cập ngẫu nhiên Thêm xóa tìm kiếm ngẫu nhiên

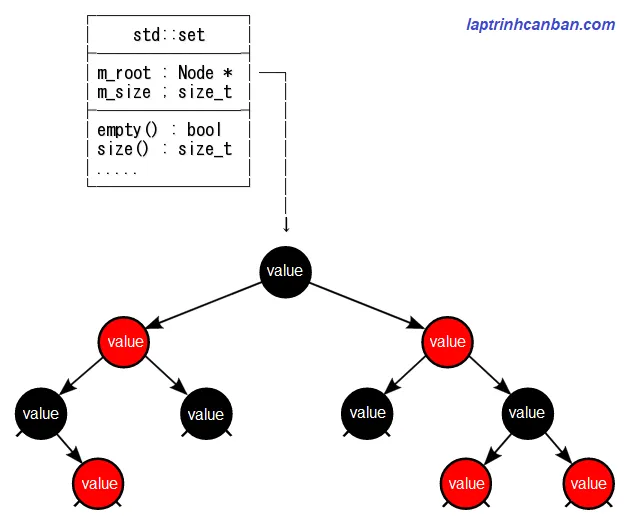
vector O(1) O(N)

set, multiset chậm O(log N)

### Cấu trúc dữ liệu set trong c++

Cấu trúc dữ liệu set trong C++ thuộc dạng Red–black tree (cây đỏ đen) - một cây nhị phân, là một cấu trúc dữ liệu trong khoa học máy tính để tổ chức các thành phần dữ liệu có thể so sánh.

Cụ thể thì cấu trúc dữ liệu set trong C++ có được thể hiện như ví dụ dưới đây. Lưu ý là cấu trúc này có thể khác một chút so với thực tế cấu trúc trong môi trường máy của bạn.



Trong các Node sẽ lưu giữ giá trị (value) cũng như con trỏ của các Node con (trái, phải) của nó.

Các giá trị trong Node thỏa mãn điều kiện giá trị của Node con bên trái < Giá trị Node cha < Giá trị của Node con bên phải. Do trong set không tồn tại giá trị trùng nhau nên dấu < được sử dụng.

Độ sâu của các Node bằng nhau và cây Node thì cân bằng.

### std::set trong C++

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

## Khai báo set trong C++

### Khai báo 1 set trong C++

Cú pháp:

std::set<type> st;

Trong đó st là tên biến set và type là kiểu dữ liệu. Cách viết sử dụng cặp dấu <> như trên được viết theo cú pháp khi sử dụng chức năng template của C++ mà chúng ta sẽ cùng học trong các chuyên đề sau.

Lưu ý, mặc dù chúng ta có thể dùng bất cứ kiểu dữ liệu nào có trong C++ để khai báo type, tuy nhiên do trong set các phần tử cần phải được sắp xếp, nên kiểu của chúng cũng phải là kiểu dữ liệu có thể được so sánh.

Đối với các kiểu dữ liệu nguyên thủy như char, int, double chẳng hạn thì chúng vốn có thể tự so sánh được, nhưng nếu chúng ta sử dụng các kiểu dữ liệu không phải là kiểu dữ liệu nguyên thủy, ví dụ như cấu trúc hoặc class tự tạo chẳng hạn, thì bắt buộc phải tự định nghĩa toán tử so sánh nội bộ operator<() để làm rõ quan hệ lớn nhỏ giữa chúng.

VD:

#include <iostream>

#include <set>

using namespace std;

int main()

{

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

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

}

Khi khai báo 1 set không thuộc kiểu dữ liệu nguyên thủy, ví dụ như struct chẳng hạn thì chúng ta phải tự tạo ra toán tử so sánh nội bộ operator<() để làm rõ quan hệ lớn nhỏ giữa các phần tử như ví dụ sau:

struct Person {

    string m\_name;

    int    m\_height;

};

// Định nghĩa toán tử so sánh nội bộ của struct

bool operator<(const Person &lhs, const Person &rhs)

{

    return lhs.m\_name < rhs.m\_name;

}

/\*Khai báo set thuộc kiểu struct\*/

std::set<Person> st;

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

Cú pháp:

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

VD:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    set<string> name, job, sex;

    set<int> age;

}

## Khởi tạo set trong C++

Cú pháp:

std::set<type> st {value1, value2, value3, ...};

Trong đó

* type là kiểu dữ liệu
* st là tên biến set
* value là các giá trị của set

VD:

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

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

### Khai báo set 2 chiều trong C++

Cú pháp:

using namespace std;

set<set<type> > st {l1, l2, l3, ...};

Trong đó:

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

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

VD:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    /\*Khai báo set 2 chiều\*/

    set<set<string> > all\_user{

        {"Kiyoshi", "male", "Hanoi"},

        {"Honda", "male", "Tokyo"},

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

    return 0;

}

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

#include <iostream>

#include <set>

using namespace std;

int main()

{

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

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

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

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

    /\*Khai báo set 2 chiều\*/

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

    return 0;

}

## Truy cập phần tử trong set C++

Khác với vector hay mảng, do cấu trúc của set theo dạng cây chứ không phải dạng mảng nên chúng ta không thể truy cập ngẫu nhiên vào phần tử bất kỳ trong một set.Do vậy 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 set, thông qua vòng lặp hoặc là trình lặp.

VD:

#include <iostream>

#include <set>

using namespace std;

int main()

{

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

    int n=0, index = 2;

    for (string x: user) {

        /\*Truy cập vào phần tử thứ 2 trong set và kết thúc khi tìm thấy\*/

        if (n == index) {

            cout << x << endl;

            break;

        }

        ++n ;

    }

    return 0;

}

male

Chúng ta cũng có thể truy cập và in ra toàn bộ các phần tử trong set 1 chiều như sau:

#include <iostream>

#include <set>

using namespace std;

int main()

{

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

    for (string x: user) {

        cout << x << endl;

    }

    return 0;

}

Kiyoshi

Tokyo

male

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

#include <iostream>

#include <set>

using namespace std;

int main()

{

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

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

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

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

    /\*Khai báo set 2 chiều\*/

    set<set<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 set trong C++

Như đã phân tích ở trên thì giữa set và vector trong C++ có một số điểm khác biệt như sau:

Vector là mảng động còn set có cấu trúc cây nhị phân tạo bởi các Node

Vector chấp nhận phần tử trùng lặp còn set thì không. Phần tử trong set phải là duy nhất

Phần tử trong vector không được sắp xếp còn phần tử trong set thì được tự động sắp xếp thứ tự cụ thể.

Vector có tốc độ truy cập ngẫu nhiên nhanh hơn set, nhưng có tốc độ thêm xóa tìm kiếm ngẫu nhiên kém hơn set.

Thông thường trong trường hợp chúng ta không hay thêm xóa tìm kiếm phần tử thì việc dùng vector sẽ có lợi hơn, do tốc độ cũng tương đương mà lại tiết kiệm bộ nhớ.

Tuy nhiên trong các trường hợp không cần thiết việc truy cập ngẫu nhiên và hay thêm xóa tìm kiếm phần tử thì việc sử dụng set thay cho vector sẽ có lợi nhiều hơn.

## Duyệt set

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

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

for ( auto& x : v) {

    // Xử lý

}

Trong đó:

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

VD:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    set<int> data{5, 6, 88, -2};

    for (auto x: data) {

        cout << x << endl;

    }

}

-2

5

6

88

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

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

for (int x: data) {

    cout << x << endl;

}

Một cách tương tự chúng ta cũng có thể dùng vòng lặp dựa trên phạm vi để duyệt set 2 chiều trong C++ như sau:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    set<set<int> > data{{5, 2}, {6,3}, {88, -2}};

    for (auto x: data) {

        for (auto y: x) {

            cout << y << endl;

        }

    }

}

-2

88

2

5

3

6

### Duyệt set trong C++ bằng iterator

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

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

cout << \*itr << endl;

}

Trong đó:

* st là tên set
* itr là tên iterator

VD:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    set<int> data{5, 6, 88, -2};

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

        cout << \*itr << endl;

    }

}

-2

5

6

88

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

Cú pháp:

st.size();

Trong đó st là set cần lấy kích thước (số phần tử) chứa trong nó.

VD:

#include <iostream>

#include <set>

using namespace std;

int main ()

{

    set<int> st{3, 1, 4,5};

    cout<< st.size() <<endl;

    return 0;

}

4

Lưu ý, số phần tử hay kích thước của set ở đây được tính sau khi các phần tử được kiểm tra trùng lặp và sắp xếp trong set, chứ không phải là số phần tử mà chúng ta đã dùng khi khai báo set.

Ví dụ, nếu khi khai báo set mà tồn tại phần tử trùng lặp thì số phần tử được đếm bởi hàm size() sẽ khác với số phần tử chỉ định trong khai báo như sau:

#include <iostream>

#include <set>

using namespace std;

int main ()

{

    set<int> st{3, 3,  1, 4, 4, 5};

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout<< "\n "<< st.size() ;

    return 0;

}

1 3 4 5

4

## Thêm chèn phần tử vào set trong C++

### Chèn 1 phần tử vào set trong C++ bằng hàm insert

Cú pháp:

st.insert(value);

Trong đó st là set ban đầu, và value là phần tử cần chèn.

Hàm set insert sẽ trả về một cặp kết quả pair<iterator, bool> với iterator là trình lặp của set kết quả, và bool là việc có thực hiện việc chèn hay không, dưới dạng 0 hoặc 1.

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất set

void dump(set<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    set<int> st{1, 3, 6, 7, 5};

    dump(st);

    //chèn phần tử đã tồn tại trong set

    st.insert(3);

    dump(st);

    //chèn phần tử chưa tồn tại trong set

    st.insert(4);

    dump(st);

    return 0;

}

1 3 5 6 7

1 3 5 6 7

1 3 4 5 6 7

Chúng ta cũng có thể kiểm tra việc chèn phần tử đã thực hiện hay chưa bằng phương thức first() hoặc second() từ kết quả trả về của hàm như sau:

#include <iostream>

#include <set>

using namespace std;

int main ()

{

    set<int> st{1, 3, 6, 7, 5};

    auto r = st.insert(3);

    std::cout << r.second << "\n";

    // 0 : không chèn

    r = st.insert(4);

    std::cout << r.second << "\n";

    // 1 : đã chèn

    return 0;

}

### Chèn nhiều phần tử vào set trong C++ bằng hàm insert

Cú pháp:

st.insert(il);

Trong đó st là set ban đầu, và il là một initializer\_list chứa các phần tử cần chèn vào set.

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất set

void dump(set<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    set<int> st{1, 3, 6, 7, 5};

    dump(st);

    /\*Khai báo initializer\_list chứa phần tử cần chèn \*/

    int myints[]= {2,10,15};

    /\*Chèn initializer\_list vào set\*/

    st.insert (myints,myints+3);

    dump(st);

    return 0;

}

1 3 5 6 7

1 2 3 5 6 7 10 15

## Xoá phần tử

### Xóa 1 phần tử trong set bằng hàm erase c++

Cú pháp:

st.erase(itr);

OR

st.erase(value);

Trong đó st là set ban đầu, itr và value là trình lặp hoặc là giá trị của phần tử cần xóa.

Nếu sử dụng erase(itr) thì hàm sẽ xóa đi phần tử tại vị trí mà trình lặp xác định bởi itr chỉ đến.

Và nếu sử dụng erase(value) thì hàm sẽ tìm phần tử có giá trị bằng với value trong list và tiến hành xóa nó đi.

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất set

void dump(set<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    set<int> st{1, 3, 6, 7, 5};

    dump(st);

    /\*Tạo trình lặp trỏ đến vị trí đầu tiên của set\*/

    auto itr = st.begin();

    ++itr; //Di chuyển trình lặp đến vị trí thứ nhất trong set

    //xóa phần tử tại vị trí itr chỉ đến (vị trí thứ nhất)

    st.erase(itr);

    dump(st);

    //xóa phần tử có giá trị bằng 6 trong set

    st.erase(6);

    dump(st);

    return 0;

}

1 3 5 6 7

1 5 6 7

1 5 7

Chúng ta cũng có thể kiểm tra số phần tử đã được xóa đi từ kết quả trả về của hàm như sau:

#include <iostream>

#include <set>

using namespace std;

int main ()

{

    set<int> st{1, 3, 6, 7, 5};

    auto r = st.erase(2); //Xóa phần tử không tồn tại trong set

    std::cout << r << "\n";

    // 0

    r = st.erase(6); //Xóa phần tử tồn tại trong set

    std::cout << r << "\n";

    // 1

    return 0;

}

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

Cú pháp:

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

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

Lưu ý là phạm vi xóa là [first,last) sẽ được tính từ iterator\_first đến trước iterator\_last, nghĩa là phần tử ở vị trí iterator\_first sẽ được xóa nhưng phần tử ở vị trí iterator\_last sẽ không bị xóa đi.

Và các giá trị trình lặp (iterator) này được tính sau khi các phần tử đã được sắp xếp và lưu trong set, chứ không phải là theo thứ tự các phần tử khi chúng ta khai báo set.

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất set

void dump(set<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    set<int> st{1, 3, 6, 7, 5, 9, 2};

    dump(st);

    int n;

    /\*Khai báo phạm vi cần xóa\*/

    int start = 2, end =5;

    /\*Tạo các trình lặp trỏ tới start và end với giá trị ban đầu\*/

    auto itr\_start = st.begin();

    auto itr\_end = st.begin();

    /\*Thay đổi trình lặp tương ứng tới các vị trí start và end\*/

    for (int i=1; i <= start; i++ )

        ++itr\_start;

    for (int i=1; i <= end; i++ )

        ++itr\_end;

    //Sau đó dùng hàm erase để xóa phạm vi là xong

    st.erase(itr\_start, itr\_end);

    dump(st);

    return 0;

}

1 2 3 5 6 7 9

1 2 7 9

## Sao chép set trong C++

set 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 set vào một set mới.

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

std::set<type> st\_dest(st\_src);

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

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất set

void dump(set<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main()

{

    set<int> st\_src{3, 1, 4,5};

    cout << "Orginary set" << endl;

    dump (st\_src);

    /\*Sao chép set\*/

    set<int> st\_dest(st\_src);

    cout << "Copy set" << endl;

    dump (st\_dest);

    return 0;

}

Orginary set

1 3 4 5

Copy set

1 3 4 5

### Hoán đổi 2 set trong C++

Cú pháp:

st1.swap(st2);

Trong đó st1 và st2 là 2 set cần hoán đổi nội dung cho nhau.

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất set

void dump(set<int>& v)

{

    for (auto x: v) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main()

{

    set<int> st1{3, 1, 4, 1, 5};

    set<int> st2{9, 8, 7};

    cout << "Before swap" << endl;

    dump(st1);

    dump(st2);

    st1.swap(st2);

    cout << "After swap" << endl;

    dump(st1);

    dump(st2);

}

Before swap

1 3 4 5

7 8 9

After swap

7 8 9

1 3 4 5

Ngoài cách dùng hàm set swap, chúng ta cũng có thể dùng function template là std::swap để tiến hành hoán đổi 2 set 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 <set>

using namespace std;

//Tạo hàm xuất set

void dump(set<int>& v)

{

    for (auto x: v) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main()

{

    set<int> st1{3, 1, 4, 1, 5};

    set<int> st2{9, 8, 7};

    cout << "Before swap" << endl;

    dump(st1);

    dump(st2);

    swap(st1,st2);

    cout << "After swap" << endl;

    dump(st1);

    dump(st2);

}

Before swap

3 1 4 1 5

9 8 7

After swap

9 8 7

3 1 4 1 5

## Tính tổng

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

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

Cú pháp:

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

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

VD:

#include <iostream>

#include <numeric>

#include <set>

using namespace std;

int main ()

{

    set<int> st{3, 1, 4, 2, 5};

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

    cout << "SUM = " << sum << endl;

    return 0;

}

SUM = 15

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

VD:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    set<int> st{3, 1, 4, 2, 5};

    int sum = 0;

    for (auto x: st) {

        sum += x;

    }

    cout << "SUM = " << sum << endl;

    return 0;

}

SUM = 15

## Set trống

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

Cú pháp:

st.empty();

Trong đó st là set cần kiểm tra.

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

VD:

#include <iostream>

#include <set>

using namespace std;

int main ()

{

    std::set<int> st;

    if( st.empty() )

        std::cout << "empty.\n";

    else

        std::cout << "not empty.\n";

    std::set<int> st2{1, 2, 3};

    if( st2.empty() )

        std::cout << "empty.\n";

    else

        std::cout << "not empty.\n";

    return 0;

}

empty.

not empty.

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

Cú pháp:

st.clear();

Trong đó st là set cần làm trống.

Khác với vector thì hàm set clear ngoài việc làm trống set chỉ định (xóa đi tất cả phần tử) thì còn giải phóng bộ nhớ sử dụng cho việc lưu trữ dữ liệu đã dùng.

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất set

void dump(set<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    std::set<int> st{3, 1, 4};

    dump(st);

    st.clear();

    dump(st);

}

1 3 4

# Multiset

## Tổng quan

### Multiset trong c++ là gì

multiset trong C++ là một tập hợp các phần tử có thể trùng lặp được sắp xếp theo thứ tự cụ thể, và được sử dụng làm tiêu chuẩn để xử lý các đối tượng chứa nhiều phần tử trong C++.

Các phần tử trong multiset có thể trùng với các phần tử khác, ngoài ra thì phần tử trong multiset không thể thay đổi giá trị, tuy nhiên chúng có thể được chèn hoặc xóa khỏi multiset.

Về mặt nội bộ, các phần tử trong multiset luôn được sắp xếp theo thứ tự cụ thể một cách nghiêm ngặt, được chỉ ra bởi đối tượng so sánh nội bộ của nó. Nếu bạn thêm các phần tử mới không theo thứ tự cụ thể vào một multiset, chúng sẽ tự động sắp xếp lại theo giá trị trước khi được lưu trữ nội bộ.

Trong C++ cũng có một loại dữ liệu khá giống với multiset là set khi các phần tử trong chúng luôn được sắp xếp theo thứ tự. Tuy nhiên thì khác với set vốn không cho phép các phần tử có thể trùng nhau cùng tồn tại, thì các phần tử trong multiset lại có thể trùng với các phần tử khác nó.

Nói tóm lại thì multiset trong C+++ sẽ giống như một mảng với các phần tử luôn được sắp xếp theo một thứ tự cụ thể.

Về mặt tốc độ xử lý thì multiset có khả năng thêm, xóa, tìm kiếm dữ liệu với tốc độ cực cao là O(log N), và nó còn cao hơn cả vector với tốc là O(1). Tuy nhiên thì giống với set thì do các phần tử được quản lý dạng cây nhị phân nên tốc độ truy cập ngẫu nhiên tới một phần tử chỉ định trong multiset lại cực thấp.

Do đó, trong trường hợp chúng ta không hay thêm xóa tìm kiếm phần tử thì việc dùng vector sẽ có lợi hơn, do tốc độ cũng tương đương mà lại tiết kiệm bộ nhớ.

Tuy nhiên trong các trường hợp không cần thiết việc truy cập ngẫu nhiên và hay thêm xóa tìm kiếm phần tử thì việc sử dụng multiset thay cho vector sẽ có lợi nhiều hơn.

Loại Truy cập ngẫu nhiên Thêm xóa tìm kiếm ngẫu nhiên

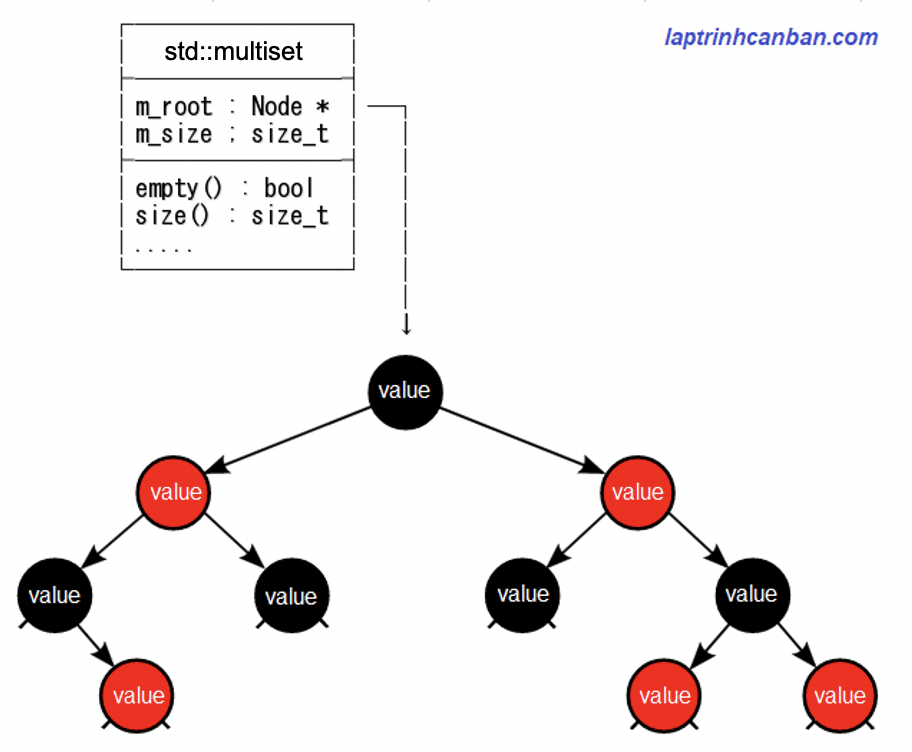
vector O(1) O(N)

multiset, set chậm O(log N)

### Cấu trúc dữ liệu multiset trong c++

Giống với set thì cấu trúc dữ liệu multiset trong C++ thuộc dạng Red–black tree (cây đỏ đen) - một cây nhị phân, là một cấu trúc dữ liệu trong khoa học máy tính để tổ chức các thành phần dữ liệu có thể so sánh.

Cụ thể thì cấu trúc dữ liệu multiset trong C++ có được thể hiện như ví dụ dưới đây. Lưu ý là cấu trúc này có thể khác một chút so với thực tế cấu trúc trong môi trường máy của bạn.



Trong các Node sẽ lưu giữ giá trị (value) cũng như con trỏ của các Node con (trái, phải) của nó.

Các giá trị trong Node thỏa mãn điều kiện giá trị của Node con bên trái <= Giá trị Node cha <= Giá trị của Node con bên phải. Do trong multiset có thể trùng nhau nên dấu <= được sử dụng.

Độ sâu của các Node bằng nhau và cây Node thì cân bằng.

### std::multiset trong C++

std::multiset trong C++ là một thư viện chuẩn được sử dụng để xử lý multiset trong C++.

Giống như std::set thì std::multiset được cài sẵn trong header file set và để sử dụng được chức năng này, chúng ta cần thêm dòng #include <set> vào đầu chương trình.

Cần đặc biệt lưu ý ở đây là #include <set> chứ không phải là #include <multiset> .Khác với các container khác vốn có header file riêng biệt để quản lý thì multiset lại dùng chung header file với set.

#include <set>

int main()

{

    std::multiset<int> st1;

    std::multiset<double> st2;

}

## Khai báo multiset trong C++

### Khai báo 1 multiset trong C++

Cú pháp:

std::multiset<type> st;

Trong đó st là tên biến multiset và type là kiểu dữ liệu.

Lưu ý, mặc dù chúng ta có thể dùng bất cứ kiểu dữ liệu nào có trong C++ để khai báo type, tuy nhiên do trong multiset các phần tử cần phải được sắp xếp, nên kiểu của chúng cũng phải là kiểu dữ liệu có thể được so sánh.

Đối với các kiểu dữ liệu nguyên thủy như char, int, double chẳng hạn thì chúng vốn có thể tự so sánh được, nhưng nếu chúng ta sử dụng các kiểu dữ liệu không phải là kiểu dữ liệu nguyên thủy, ví dụ như cấu trúc hoặc class tự tạo chẳng hạn, thì bắt buộc phải tự định nghĩa toán tử so sánh nội bộ operator<() để làm rõ quan hệ lớn nhỏ giữa chúng.

VD:

#include <iostream>

#include <set>

using namespace std;

int main()

{

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

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

}

Khi khai báo 1 multiset không thuộc kiểu dữ liệu nguyên thủy, ví dụ như struct chẳng hạn thì chúng ta phải tự tạo ra toán tử so sánh nội bộ operator<() để làm rõ quan hệ lớn nhỏ giữa các phần tử như ví dụ sau:

struct Person {

    string m\_name;

    int    m\_height;

};

// Định nghĩa toán tử so sánh nội bộ của struct

bool operator<(const Person &lhs, const Person &rhs)

{

    return lhs.m\_name < rhs.m\_name;

}

/\*Khai báo multiset thuộc kiểu struct\*/

std::multiset<Person> st;

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

Cú pháp:

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

VD:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    multiset<string> name, job, sex;

    multiset<int> age;

}

## Khởi tạo multiset trong C++

Cú pháp:

std::multiset<type> st {value1, value2, value3, ...};

Trong đó

* type là kiểu dữ liệu
* st là tên biến multiset
* value là các giá trị của multiset

VD:

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

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

## Khởi tạo multiset 2 chiều trong C++

Cú pháp:

multiset<multiset<type> > st {l1, l2, l3, ...};

Trong đó:

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

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

#include <iostream>

#include <set>

using namespace std;

int main()

{

    /\*Khai báo multiset 2 chiều\*/

    multiset<multiset<string> > all\_user{

        {"Kiyoshi", "male", "Hanoi"},

        {"Honda", "male", "Tokyo"},

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

    return 0;

}

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

VD:

#include <iostream>

#include <set>

using namespace std;

int main()

{

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

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

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

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

    /\*Khai báo multiset 2 chiều\*/

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

    return 0;

}

## Truy cập phần tử trong multiset C++

Khác với vector hay mảng, do cấu trúc của multiset theo dạng cây chứ không phải dạng mảng nên chúng ta không thể truy cập ngẫu nhiên vào phần tử bất kỳ trong một multiset.

Do vậy 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 multiset, thông qua vòng lặp hoặc là trình lặp mà Kiyoshi đã giới thiệu trong bài Duyệt multiset trong C++.

VD:

#include <iostream>

#include <set>

using namespace std;

int main()

{

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

    int n=0, index = 2;

    for (string x: user) {

        /\*Truy cập vào phần tử thứ 2 trong multiset và kết thúc khi tìm thấy\*/

        if (n == index) {

            cout << x << endl;

            break;

        }

        ++n ;

    }

    return 0;

}

// male

Chúng ta cũng có thể truy cập và in ra toàn bộ các phần tử trong multiset 1 chiều như sau:

#include <iostream>

#include <set>

using namespace std;

int main()

{

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

    for (string x: user) {

        cout << x << endl;

    }

    return 0;

}

Kiyoshi

Tokyo

male

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

#include <iostream>

#include <set>

using namespace std;

int main()

{

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

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

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

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

    /\*Khai báo multiset 2 chiều\*/

    multiset<multiset<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 multiset trong C++

Như đã phân tích ở trên thì giữa multiset và vector trong C++ có một số điểm khác biệt như sau:

Vector là mảng động còn multiset có cấu trúc cây nhị phân tạo bởi các Node

Phần tử trong vector không được sắp xếp còn phần tử trong multiset thì được tự động sắp xếp thứ tự cụ thể.

Vector có tốc độ truy cập ngẫu nhiên nhanh hơn multiset, nhưng có tốc độ thêm xóa tìm kiếm ngẫu nhiên kém hơn multiset.

Thông thường trong trường hợp chúng ta không hay thêm xóa tìm kiếm phần tử thì việc dùng vector sẽ có lợi hơn, do tốc độ cũng tương đương mà lại tiết kiệm bộ nhớ.

Tuy nhiên trong các trường hợp không cần thiết việc truy cập ngẫu nhiên và hay thêm xóa tìm kiếm phần tử thì việc sử dụng multiset thay cho vector sẽ có lợi nhiều hơn.

## Duyệt phần tử

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

Cú pháp:

for ( auto& x : v) {

    // Xử lý

}

VD:

#include <iostream>

#include <set>

using namespace std;

int main()

{

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

    for (auto x: data) {

        cout << x << endl;

    }

}

-2

5

5

6

88

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

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

for (int x: data) {

    cout << x << endl;

}

Một cách tương tự chúng ta cũng có thể dùng vòng lặp dựa trên phạm vi để duyệt multiset 2 chiều trong C++ như sau:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    multiset<multiset<int> > data{{5, 2}, {6,3}, {88, -2}};

    for (auto x: data) {

        for (auto y: x) {

            cout << y << endl;

        }

    }

}

-2

88

2

5

3

6

### Duyệt multiset trong C++ bằng iterator

Cú pháp:

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

cout << \*itr << endl;

}

Trong đó:

* st là tên multiset
* itr là tên iterator

VD:

#include <iostream>

#include <set>

using namespace std;

int main()

{

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

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

        cout << \*itr << endl;

    }

}

-2

5

5

6

88

## Lấy kích thước

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

Cú pháp:

st.size();

Trong đó st là multiset cần lấy kích thước (số phần tử) chứa trong nó.

VD:

#include <iostream>

#include <set>

using namespace std;

int main ()

{

    multiset<int> st{3, 1, 4, 5};

    cout<< st.size() <<endl;

    return 0;

}

4

Khác với set thì trong multiset cho phép tồn tại phần tử trùng lặp, nên khi đếm số phần tử có trong multiset thì hàm size() sẽ trả về toàn bộ các phần tử mà chúng ta đã dùng để khai báo set. Ví dụ:

#include <iostream>

#include <set>

using namespace std;

int main ()

{

    multiset<int> st{3, 3,  1, 4, 4, 5};

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout<< "\n "<< st.size() ;

    return 0;

}

1 3 3 4 4 5

6

## Chèn, thêm phần tử

### Chèn 1 phần tử vào multiset trong C++ bằng hàm insert

Cú pháp:

st.insert(value);

Trong đó st là multiset ban đầu, và value là phần tử cần chèn.

Do multiset cho phép tồn tại các phần tử trùng lặp trong nó, nên chúng ta có thể chèn một phần tử không phân biệt việc nó đã tồn tại hay chưa trong multiset ban đầu.

Lại nữa, phần tử trong multiset được sắp xếp theo thứ tự cụ thể trước khi được lưu, nên các phần tử cần chèn thêm vào multiset sẽ được tự động quyết định vị trí dựa trên bộ sắp xếp, và chúng ta sẽ không quyết định được vị trí cần chèn của phần tử.

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất multiset

void dump(multiset<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    multiset<int> st{1, 3, 6, 7, 5};

    dump(st);

    //chèn phần tử đã tồn tại trong multiset

    st.insert(3);

    dump(st);

    //chèn phần tử chưa tồn tại trong multiset

    st.insert(4);

    dump(st);

    return 0;

}

1 3 5 6 7

1 3 3 5 6 7

1 3 3 4 5 6 7

### Chèn nhiều phần tử vào multiset trong C++ bằng hàm insert

Cú pháp:

st.insert(il);

Trong đó st là multiset ban đầu, và il là một initializer\_list chứa các phần tử cần chèn vào multiset.

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất multiset

void dump(multiset<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    multiset<int> st{1, 3, 6, 7, 5};

    dump(st);

    /\*Khai báo initializer\_list chứa phần tử cần chèn \*/

    int myints[]= {2, 3, 10};

    /\*Chèn initializer\_list vào multiset\*/

    st.insert (myints,myints+3);

    dump(st);

    return 0;

}

1 3 5 6 7

1 2 3 3 5 6 7 10

## Xoá phần tử

### Xóa 1 phần tử trong multiset bằng hàm erase c++

Cú pháp:

st.erase(itr);

OR

st.erase(value);

Trong đó st là multiset ban đầu, itr và value là trình lặp hoặc là giá trị của phần tử cần xóa.

Nếu sử dụng erase(itr) thì hàm sẽ xóa đi phần tử tại vị trí mà trình lặp xác định bởi itr chỉ đến.

Và nếu sử dụng erase(value) thì hàm sẽ xoá đi tất cả các phần tử có giá trị bằng với value trong list và tiến hành xóa nó đi.

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất multiset

void dump(multiset<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    multiset<int> st{1, 2, 3, 6, 7, 5, 3, 9};

    dump(st); // 1 2 3 3 5 6 7 9

    /\*Tạo trình lặp trỏ đến vị trí đầu tiên của multiset\*/

    auto itr = st.begin();

    ++itr; //Di chuyển trình lặp đến vị trí thứ nhất trong multiset với giá trị bằng 2

    //xóa phần tử tại vị trí itr chỉ đến (vị trí thứ nhất)

    st.erase(itr);

    dump(st); //1 3 3 5 6 7 9

    //xóa tất cả các phần tử có giá trị bằng 3 trong multiset

    st.erase(3);

    dump(st); // 1 5 6 7 9

    return 0;

}

Chúng ta cũng có thể kiểm tra số phần tử đã được xóa đi từ kết quả trả về của hàm như sau:

#include <iostream>

#include <set>

using namespace std;

int main ()

{

    multiset<int> st{1, 2, 3, 6, 7, 5, 3, 9};

    auto r = st.erase(4); //Xóa phần tử không tồn tại trong multiset

    std::cout << r << "\n";

    // 0

    r = st.erase(3); //Xóa phần tử tồn tại trong multiset

    std::cout << r << "\n";

    // 2

    return 0;

}

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

Cú pháp:

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

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

Lưu ý là phạm vi xóa là [first,last) sẽ được tính từ iterator\_first đến trước iterator\_last, nghĩa là phần tử ở vị trí iterator\_first sẽ được xóa nhưng phần tử ở vị trí iterator\_last sẽ không bị xóa đi.

Và các giá trị trình lặp (iterator) này được tính sau khi các phần tử đã được sắp xếp và lưu trong multiset, chứ không phải là theo thứ tự các phần tử khi chúng ta khai báo multiset.

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất multiset

void dump(multiset<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    multiset<int> st{1, 2, 3, 6, 7, 5, 3, 9};

    dump(st);    // 1 2 3 3 5 6 7 9

    int n;

    /\*Khai báo phạm vi cần xóa\*/

    int start = 2, end =5;

    /\*Tạo các trình lặp trỏ tới start và end với giá trị ban đầu\*/

    auto itr\_start = st.begin();

    auto itr\_end = st.begin();

    /\*Thay đổi trình lặp tương ứng tới các vị trí start và end\*/

    for (int i=1; i <= start; i++ )

        ++itr\_start;

    for (int i=1; i <= end; i++ )

        ++itr\_end;

    //Sau đó dùng hàm erase để xóa phạm vi là xong

    st.erase(itr\_start, itr\_end);

    dump(st);   // 1 2 6 7 9

    return 0;

}

## Sao chép phần tử

Cú pháp:

std::multiset<type> st\_dest(st\_src);

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

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất multiset

void dump(multiset<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main()

{

    multiset<int> st\_src{3, 1, 4, 1, 5};

    cout << "Orginary multiset" << endl;

    dump (st\_src);

    /\*Sao chép multiset\*/

    multiset<int> st\_dest(st\_src);

    cout << "Copy multiset" << endl;

    dump (st\_dest);

    return 0;

}

Orginary multiset

1 1 3 4 5

Copy multiset

1 1 3 4 5

## Hoán đổi 2 multiset trong C++

Cú pháp:

st1.swap(st2);

Trong đó st1 và st2 là 2 multiset cần hoán đổi nội dung cho nhau.

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất multiset

void dump(multiset<int>& v)

{

    for (auto x: v) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main()

{

    multiset<int> st1{3, 1, 4, 1, 5};

    multiset<int> st2{9, 8, 7};

    cout << "Before swap" << endl;

    dump(st1);

    dump(st2);

    st1.swap(st2);

    cout << "After swap" << endl;

    dump(st1);

    dump(st2);

}

Before swap

1 1 3 4 5

7 8 9

After swap

7 8 9

1 1 3 4 5

Ngoài cách dùng hàm multiset swap, chúng ta cũng có thể dùng function template là std::swap để tiến hành hoán đổi 2 multiset 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

#include <iostream>

#include <utility>

#include <set>

using namespace std;

//Tạo hàm xuất multiset

void dump(multiset<int>& v)

{

    for (auto x: v) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main()

{

    multiset<int> st1{3, 1, 4, 1, 5};

    multiset<int> st2{9, 8, 7};

    cout << "Before swap" << endl;

    dump(st1);

    dump(st2);

    swap(st1,st2);

    cout << "After swap" << endl;

    dump(st1);

    dump(st2);

}

Before swap

1 1 3 4 5

7 8 9

After swap

7 8 9

1 1 3 4 5

## Tính tổng

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

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

Cú pháp:

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

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

VD:

#include <iostream>

#include <numeric>

#include <set>

using namespace std;

int main ()

{

    multiset<int> st{3, 1, 4, 2, 5, 2};

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

    cout << "SUM = " << sum << endl;

    return 0;

}

SUM = 17

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

VD:

#include <iostream>

#include <set>

using namespace std;

int main()

{

    multiset<int> st{3, 1, 4, 2, 5};

    int sum = 0;

    for (auto x: st) {

        sum += x;

    }

    cout << "SUM = " << sum << endl;

    return 0;

}

SUM = 17

## Tìm phần tử

### Tìm phần tử trong multiset C++ bằng hàm find

Cú pháp:

st.find(val);

Trong đó val là giá trị của phần tử cần tìm trong multiset st.

Hàm find() sẽ trả về trình lặp trỏ đến vị trí phần tử, nếu nó tồn tại trong multiset. Và nếu phần tử đó không tồn tại, hàm sẽ trả về trình lặp trỏ đến vị trí cuối cùng trong multiset.

Bằng cách ứng dụng hàm find(), chúng ta có thể tìm ra vị trí của phần tử đó trong multiset, rồi kết hợp với hàm clear() để xóa nó đi như ví dụ cụ thể sau đây:

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất multiset

void dump(multiset<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main() {

    multiset<int> st{3, 1, 4, 5, 2, 9};

    dump(st);

    //Tìm phần tử có giá trị bằng 2 trong multiset

    auto itr = st.find(2);

    //Xóa phần tử vừa tìm thấy

    st.erase (itr);

    dump(st);

    return 0;

}

1 2 3 4 5 9

1 3 4 5 9

### Tìm phần tử trong multiset C++ bằng hàm equal\_range

VD:

st.equal\_range(val);

Trong đó val là giá trị phần tử cần tìm trong multiset st.

Hàm equal\_range() sẽ trả về một cặp giá trị, với giá trị đầu tiên trỏ đến đầu phạm vi, và giá trị thứ hai trỏ đến cuối phạm vi chứa tất cả các phần tử có giá trị giống giá trị chỉ định.

VD:

#include <iostream>

#include <set>

using namespace std;

typedef multiset<int>::iterator It;

//Tạo hàm xuất multiset

void dump(multiset<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main() {

    multiset<int> st{3, 1, 2, 5, 7, 2, 3};

    dump(st);

    //Tìm phạm vi chứa phần tử có giá trị bằng 3 trong multiset

    pair<It,It> ret = st.equal\_range(2);

    //Xoá các phần tử trong phạm vi tìm thấy

    st.erase(ret.first,ret.second);

    dump(st);

    return 0;

}

1 2 2 3 3 5 7

1 3 3 5 7

Giống như trên, toàn bộ phạm vi chứa các phần tử có giá trị chỉ định đã được xác định bởi hàm equal\_range() và được xóa đi.

### Tìm phần tử trong multiset C++ bằng hàm lower\_bound

Cú pháp:

st.lower\_bound(val);

Trong đó val là giá trị của phần tử cần tìm trong multiset st.

Hàm lower\_bound() sẽ trả về trình lặp trỏ đến vị trí vị trí phần tử đầu tiên có giá trị lớn hơn hoặc bằng với giá trị chỉ định. Và nếu không tìm thấy, hàm sẽ trả về trình lặp trỏ đến vị trí cuối cùng trong multiset.

Trong trường hợp chỉ có một phần tử trong multiset có giá trị giống với giá trị chỉ định thì hàm lower\_bound sẽ trả về con trỏ chỉ đến phần tử đó. Còn nếu có nhiều phần tử có giá trị giống với giá trị chỉ định, vị trí của phần tử đầu tiên có giá trị lớn hơn giá trị chỉ định sẽ được hàm lower\_bound() trả về.

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất multiset

void dump(multiset<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main() {

    multiset<int> st{3, 3, 1, 4, 2, 5, 2, 9};

    dump(st);

    /\*Tìm vị trí phần tử đầu tiên có giá trị lớn hơn hoặc bằng 2\*/

    auto itr1 = st.lower\_bound(2); // itr1 trỏ đến 2

    ///Tìm vị trí phần tử đầu tiên có giá trị lớn hơn 5 trong multimap

    auto itr2 = st.upper\_bound(5); // itr2 trỏ đến 9

    //Xóa các phần tử trong phạm vi [itr1, itr2)

    st.erase (itr1, itr2);

    dump(st);

    return 0;

}

1 2 2 3 3 4 5 9

1 9

### Tìm phần tử trong multiset C++ bằng hàm upper\_bound

Cú pháp:

st.upper\_bound(key);

Trong đó key là giá trị của phần tử cần tìm trong multiset st.

Hàm upper\_bound() sẽ trả về trình lặp trỏ đến vị trí phần tử đầu tiên có giá trị lớn hơn giá trị chỉ định. Và nếu không tìm thấy, hàm sẽ trả về trình lặp trỏ đến vị trí cuối cùng trong map.

Trong trường hợp chỉ có một phần tử trong multiset có giá trị giống với giá trị chỉ định thì hàm upper\_bound sẽ trả về con trỏ chỉ đến phần tử ngay sau nó. Còn nếu có nhiều phần tử có giá trị giống với giá trị chỉ định, vị trí của phần tử đầu tiên có giá trị lớn hơn giá trị chỉ định sẽ được hàm upper\_bound() trả về.

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất multiset

void dump(multiset<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main() {

    multiset<int> st{3, 3, 1, 4, 2, 5, 2, 9};

    dump(st);

    /\*Tìm vị trí phần tử đầu tiên có giá trị lớn hơn hoặc bằng 3\*/

    auto itr1 = st.lower\_bound(3); //itr1 trỏ đến 3

    //Tìm vị trí phần tử đầu tiên có giá trị lớn hơn 5

    auto itr2 = st.upper\_bound(5); //itr2 trỏ đến 9

    //In các phần tử trong phạm vi [itr1, itr2)

    for (auto it=itr1; it!=itr2; ++it)

        cout << \*it << ' ';

    return 0;

}

1 2 2 3 3 4 5 9

3 3 4 5

### Đếm số lần xuất hiện của phần tử trong multiset C++ bằng hàm count

Cú pháp:

st.count(val);

Trong đó val là giá trị của phần tử cần đếm số lần xuất hiện trong multiset st.

VD:

#include <iostream>

#include <set>

using namespace std;

int main() {

    multiset<int> st{3, 1, 3, 5, 2, 3};

    //Đếm số lần xuất hiện của phần tử tồn tại trong set

    cout << st.count(3) <<endl;

    //Đếm số lần xuất hiện của phần tử không tồn tại trong set

    cout << st.count(8) <<endl;

    return 0;

}

3

0

## Multiset trống

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

Cú pháp:

st.empty();

Trong đó st là multiset cần kiểm tra.

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

VD:

#include <iostream>

#include <set>

using namespace std;

int main ()

{

    std::multiset<int> st;

    if( st.empty() )

        std::cout << "empty.\n";

    else

        std::cout << "not empty.\n";

    std::multiset<int> st2{1, 2, 3};

    if( st2.empty() )

        std::cout << "empty.\n";

    else

        std::cout << "not empty.\n";

    return 0;

}

empty.

not empty.

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

Cú pháp:

st.clear();

Trong đó st là multiset cần làm trống.

Khác với vector thì hàm multiset clear ngoài việc làm trống multiset chỉ định (xóa đi tất cả phần tử) thì còn giải phóng bộ nhớ sử dụng cho việc lưu trữ dữ liệu đã dùng.

VD:

#include <iostream>

#include <set>

using namespace std;

//Tạo hàm xuất multiset

void dump(multiset<int>& st)

{

    for (auto x: st) {

        cout << " "<< x ;

    }

    cout << endl;

}

int main ()

{

    std::multiset<int> st{3, 1, 4};

    dump(st);

    st.clear();

    dump(st);

}

1 3 4

# Map

### Map trong c++ là gì

Map trong C++ là một tập hợp các phần tử được sắp xếp theo thứ tự cụ thể, mà mỗi phần tử trong đó được hình thành bởi sự kết hợp của một cặp khóa và giá trị (key & value), với mỗi khóa là duy nhất trong map.

Trong map, các khóa (key) được sử dụng để sắp xếp và xác định giá trị (value) tương ứng được liên kết với nó. Mỗi khóa trong map là duy nhất và không được phép trùng lặp. Các giá trị trong map thì có thể trùng lặp, chúng có thể thay đổi giá trị, cũng như là được chèn hoặc xóa khỏi map.

Ví dụ cụ thể, một map chứa thông tin về một người sẽ gồm các khóa như name, old, weight. Tương ứng với các khóa này là các giá trị là thông tin cụ thể của người đó như sau:



Về mặt nội bộ, các phần tử trong map luôn được sắp xếp theo khóa của nó theo thứ tự cụ thể một cách nghiêm ngặt, được chỉ ra bởi đối tượng so sánh nội bộ của map. Nếu bạn thêm các phần tử mới không theo thứ tự cụ thể vào một map, chúng sẽ tự động sắp xếp lại theo khóa trước khi được lưu trữ nội bộ.

Trong C++ cũng có một loại dữ liệu khá giống với map là unordered\_map khi các phần tử cũng được hình thành bởi các cặp khóa và giá trị. Tuy nhiên thì khác với unordered\_map có các phần tử không được sắp xếp, thì phần tử trong một map luôn luôn được sắp xếp theo khóa của chúng.

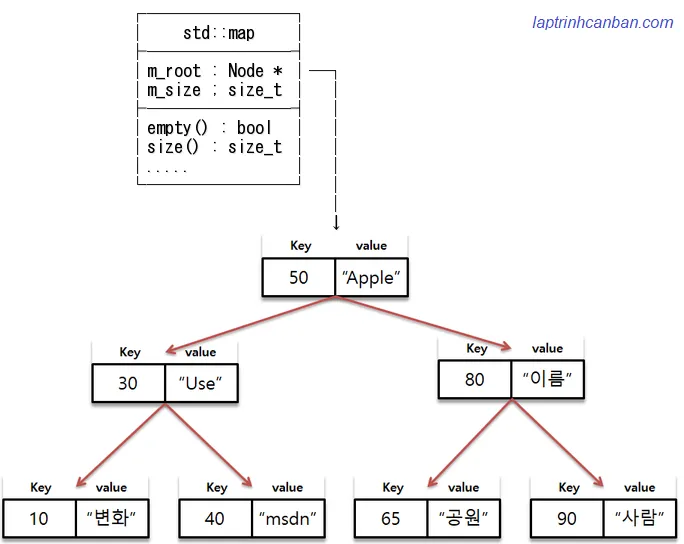
Về mặt tốc độ xử lý thì map có khả năng tìm kiếm dữ liệu theo khóa với tốc độ cực khá cao là O(log N). Tuy nhiên thì nó vẫn kém unordered\_map với tốc độ còn cao hơn là O(1), do unordered\_map sử dụng bảng hash.

Tuy nhiên thì map có lợi thế khi các phần tử của nó được sắp xếp sẵn, nên chúng ta nên dùng map trong các trường hợp cần ưu tiên tính chất này.

### Cấu trúc dữ liệu map trong c++

Cấu trúc dữ liệu map trong C++ thuộc dạng Red–black tree (cây đỏ đen) - một cây nhị phân, là một cấu trúc dữ liệu trong khoa học máy tính để tổ chức các thành phần dữ liệu có thể so sánh.

Cụ thể thì cấu trúc dữ liệu map trong C++ có được thể hiện như ví dụ dưới đây. Lưu ý là cấu trúc này có thể khác một chút so với thực tế cấu trúc trong môi trường máy của bạn.



Trong các Node sẽ lưu giữ cặp khóa:giá trị (key & value) cũng như con trỏ của các Node con (trái, phải) của nó.

Các giá trị trong Node thỏa mãn điều kiện giá trị của Node con bên trái < Giá trị Node cha < Giá trị của Node con bên phải. Do trong map không cho phép các khóa trùng nhau nên dấu < được sử dụng.

Độ sâu của các Node bằng nhau và cây Node thì cân bằng.

Nhờ vào cấu trúc dữ liệu kiểu này mà chúng ta có thể tìm kiếm nhị phân trong map, qua đó có thể tìm kiếm trong map với tốc độ cao O(Log N).

### std::map trong C++

std::map trong C++ là một thư viện chuẩn được sử dụng để xử lý map trong C++.

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

#include <map>

int main()

{

    std::map<std::string, int> mp;

    std::map<char, double> mp2;

}

## Khai báo map trong C++

### Khai báo 1 map trong C++

Cú pháp:

std::map<k\_type, v\_type> mp;

Trong đó mp là tên biến map và k\_type, v\_type lần lượt là kiểu dữ liệu của key và value. Cách viết sử dụng cặp dấu <> như trên được viết theo cú pháp khi sử dụng chức năng template của C++ mà chúng ta sẽ cùng học trong các chuyên đề sau.

Lưu ý, mặc dù chúng ta có thể dùng bất cứ kiểu dữ liệu nào có trong C++ để khai báo type, tuy nhiên do trong map các phần tử cần phải được sắp xếp, nên kiểu của chúng cũng phải là kiểu dữ liệu có thể được so sánh.

Đối với các kiểu dữ liệu nguyên thủy như char, int, double chẳng hạn thì chúng vốn có thể tự so sánh được, nhưng nếu chúng ta sử dụng các kiểu dữ liệu không phải là kiểu dữ liệu nguyên thủy, ví dụ như cấu trúc hoặc class tự tạo chẳng hạn, thì bắt buộc phải tự định nghĩa toán tử so sánh nội bộ operator<() để làm rõ quan hệ lớn nhỏ giữa chúng.

VD:

#include <iostream>

#include <map>

using namespace std;

int main()

{

    map<double, string> user;

    map<int, char> info;

}

Còn khi khai báo 1 map không thuộc kiểu dữ liệu nguyên thủy, ví dụ như struct chẳng hạn thì chúng ta phải tự tạo ra toán tử so sánh nội bộ operator<() để làm rõ quan hệ lớn nhỏ giữa các phần tử như ví dụ sau:

struct Person {

    string m\_name;

    int    m\_height;

};

// Định nghĩa toán tử so sánh nội bộ của struct

bool operator<(const Person &lhs, const Person &rhs)

{

    return lhs.m\_name < rhs.m\_name;

}

/\*Khai báo map thuộc kiểu struct\*/

std::map<Person, int> mp;

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

Cú pháp:

map<k\_type, v\_type> name1, name2, name3, ... ;

#include <iostream>

#include <map>

using namespace std;

int main()

{

    map<string, int> name, job, sex;

    map<int, char> age;

}

## Gán giá trị cho map trong C++

Sau khi đã khai báo map, chúng ta có thể gán giá trị các phần tử vào nó bằng cách sử dụng toán tử [] với cú pháp sau đây:

mp[key] = value;

VD:

std::map<std::string, int> mp;

mp["Kiyoshi"] = 1; // {"Kiyoshi", 1}

mp["Honda"] = 2;   // {"Honda", 2}

mp["Suzuki"] = 3;  // {"Suzuki", 3}

Lại nữa, giá trị của phần tử có thể được ghi đè và thay thế bởi một giá trị mới nhiều lần như sau:

mp["Kiyoshi"] = 4; // {"Kiyoshi", 4}

mp["Kiyoshi"] = 8; // {"Kiyoshi", 8}

mp["Kiyoshi"] = 88;// {"Kiyoshi", 88}

Trong trường hợp thay đổi giá trị nhiều lần, thì giá trị trong lượt thay đổi giá trị cuối cùng sẽ được sử dụng làm giá trị phần tử.

## Khởi tạo map trong C++

Cú pháp:

std::map<k\_type, v\_type> mp = { {k1, v1}, {k2, v2}, {k3, v3}, ...};

Trong đó

* mp là tên biến map
* k\_type, v\_type là kiểu dữ liệu của key và value
* k,v là các cặp key và value

Lưu ý do mỗi khóa trong map là duy nhất, nên nếu chúng ta chỉ định các phần tử có cùng khóa thì dù giá trị của chúng có giống hay khác nhau thì chỉ có duy nhất phần tử viết đầu tiên sẽ được lưu vào trong map mà thôi.

#include <iostream>

#include <map>

using namespace std;

int main ()

{

    map<string,int> mp = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 }

            };

    for (auto& x: mp) {

        std::cout << x.first << ": " << x.second << '\n';

    }

    return 0;

}

alpha: 10

beta: 20

gamma: 30

Trong trường hợp chỉ định các phần tử có khóa giống nhau, bất kể giá trị của chúng có giống hay khác nhau thì chỉ có duy nhất phần tử viết đầu tiên được lưu vào map như sau:

#include <iostream>

#include <map>

using namespace std;

int main ()

{

    map<string,int> mp = {

                { "alpha", 20 },

                { "beta", 20 },

                { "alpha", 10 },

                { "gamma", 30 },

                { "alpha", 10 },

                };

    for (auto x: mp) {

        cout << x.first << ": " << x.second << endl;

    }

    return 0;

}

// alpha: 20

// beta: 20

// gamma: 30

## Truy cập phần tử trong map C++

### Truy cập phần tử trong map C++ bằng toán tử []

Cú pháp:

mp[key]

Nếu như key tồn tại trong map, giá trị tương ứng của key sẽ được trả về. Tuy nhiên nếu không tồn tại, giá trị 0 sẽ được trả về

VD:

#include <iostream>

#include <map>

using namespace std;

int main ()

{

    /\*Khai báo và gán giá trị\*/

    std::map<std::string, int> mp;

    mp["Kiyoshi"] = 1;

    /\*Truy cập vào phần tử vừa gán bằng key\*/

    cout << mp["Kiyoshi"] <<   //1

    /\*Truy cập vào phần tử không tồn tại trong map\*/

    cout << mp["Honda"];       //0

    return 0;

}

Ngoài việc xuất giá trị, chúng ta cũng có thể thay đổi giá trị của một phần tử trong map với cách này. Ví dụ:

/\*Khai báo và gán giá trị\*/

std::map<std::string, int> mp;

mp["Kiyoshi"] = 1;

/\*Truy cập và thay đổi giá trị\*/

mp["Kiyoshi"] = 2;

### Truy cập phần tử trong map C++ bằng hàm at()

Hàm at() trong C++ là một hàm thành viên trong class std:map, có tác dụng truy cập vào phần tử trong map thông qua key của nó.

Nếu như key tồn tại trong map, giá trị tương ứng của key sẽ được trả về. Tuy nhiên nếu không tồn tại, hàm at() sẽ trả về lỗi out\_of\_range.

Cú pháp:

mp.at(key);

Trong đó mp là tên map và key là khóa của phần tử cần truy cập.

/\*Khai báo và gán giá trị\*/

std::map<std::string, int> mp;

mp["Kiyoshi"] = 1;

/\*Truy cập vào phần tử vừa gán bằng key\*/

cout << mp.at("Kiyoshi"); //1

/\*Truy cập vào phần tử không tồn tại trong map\*/

cout << mp.at("Honda"); // throwing an instance of 'std::out\_of\_range

Có thể thấy rõ sự khác biệt giữa cách truy cập phần tử trong map bằng hàm at() và toán tử [] chính là ở kết quả khi chỉ định một key không tồn tại trong map.

## set và map trong c++

Giống với map thì kiểu set trong C++ cũng chứa các phần tử duy nhất được sắp xếp theo thứ tự trong nó. Và cấu trúc dữ liệu của 2 kiểu này cũng đều thuộc dạng cây nhị phân Red–black tree (cây đỏ đen).

Tuy vậy thì giữa chúng có 3 điểm khác biệt rất rõ ràng như sau:

Phần tử trong set tạo bởi 1 giá trị, trong khi trong map tạo bởi một cặp khóa và giá trị.

Phần tử trong set được sắp xếp theo giá trị của chúng, còn trong map thì theo khóa của chúng

Vùng bộ nhớ chứa set sẽ nhỏ hơn so với một map có cùng số phần tử, do trong map ngoài giá trị thì còn lưu trữ cả key nữa

Do vậy chúng ta có thể sử dụng một trong 2 kiểu này để xử lý các đối tượng mà cần có sự sắp xếp phần tử trong nó, tùy theo nhu cầu của mình.

## Duyệt map

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

Cú pháp:

for ( auto& x : mp) {

cout << x.first << “: “ << x.second << endl;

}

VD:

#include <iostream>

#include <map>

using namespace std;

int main()

{

    map<string,int> mp = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

    for (auto x: mp) {

        cout << x.first << ": " << x.second << endl;

    }

}

alpha: 10

beta: 20

gamma: 30

### Duyệt map trong C++ bằng iterator

Cú pháp:

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

cout << itr->first << ": "<< itr->second << "\n";

}

VD:

#include <iostream>

#include <map>

using namespace std;

int main()

{

    map<string,int> mp = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

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

        cout  << itr->first << ": "<< itr->second << "\n";

    }

}

alpha: 10

beta: 20

gamma: 30

## Lấy kích thước

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

Cú pháp:

mp.size();

VD:

#include <iostream>

#include <map>

using namespace std;

int main ()

{

    map<string,int> mp = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

    cout<< mp.size() <<endl;

    return 0;

}

//3

Lưu ý, số phần tử hay kích thước của map ở đây được tính sau khi các phần tử được kiểm tra key có trùng lặp và sắp xếp trong map, chứ không phải là số phần tử mà chúng ta đã dùng khi khai báo map.

Ví dụ, nếu khi khai báo map mà tồn tại các phần tử trùng lặp key thì số phần tử được đếm bởi hàm size() sẽ khác với số phần tử chỉ định trong khai báo như sau:

#include <iostream>

#include <map>

using namespace std;

int main ()

{

    map<string,int> mp = {

                { "alpha", 20 },

                { "beta", 20 },

                { "alpha", 10 },

                { "gamma", 30 },

                { "alpha", 10 },

                };

    for (auto x: mp) {

        cout << x.first << ": " << x.second << endl;

    }

    cout<< mp.size() <<endl;

    return 0;

}

alpha: 20

beta: 20

gamma: 30

3

## Thêm phần tử

### Thêm 1 phần tử vào map trong C++ bằng toán tử []

Cú pháp:

mp[key] = value;

VD:

map<char,int> foo,bar;

foo['x']=100; //()

foo['y']=200;

Lưu ý do mỗi key trong map đều là duy nhất nên nếu chúng ta thêm một phần tử mới vào map nhưng lại có key trùng với một phần tử đã tồn tại trước đó, thì giá trị của phần tử mới này sẽ được dùng để ghi đè lên phần tử trước đó.

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất map

void dump(map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    map<char,int> mymap;

    mymap['x']=100;

    mymap['y']=200;

    mymap['z']=300;

    dump(mymap);        // x:100 y:200 z:300

    mymap['z']=888;

    dump(mymap);        // x:100 y:200 z:888

}

### Chèn 1 phần tử vào map trong C++ bằng hàm insert

Cú pháp:

mp.insert(std::pair<k\_type,x\_type>(k,v));

Trong đó

* mp là map ban đầu
* pair<k\_type,x\_type>(k,v) sử dụng để chỉ định key và value của phần tử cần thêm, trong đó k\_type,x\_type là kiểu và k,v là key và value.

Trong trường hợp không rõ kiểu, hoặc muốn rút bỏ chỉ định kiểu của key và value, chúng ta có thể dùng hàm make\_pair() để thay thế cho pair() trong hàm insert, với cú pháp sau đây:

mp.insert(std::make\_pair(k,v));

Hàm map insert sẽ trả về một cặp kết quả pair<iterator, bool> với iterator là trình lặp trỏ đến map kết quả, và bool là việc có thực hiện việc chèn hay không, dưới dạng 0 hoặc 1.

Bởi vì các phần tử trong một map là duy nhất, nên thao tác chèn sẽ kiểm tra xem mỗi phần tử được chèn đã tồn tại trong map hay chưa. Nếu chưa tồn tại thì phần tử đó sẽ được chèn và ngược lại nếu đã tồn tại thì không được chèn.

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất map

void dump(map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    /\*Khai báo và gán giá trị cho foo và bar\*/

    map<char,int> mp;

    mp['a']=100;

    mp['b']=200;

    // duyệt map ban đầu

    dump(mp);

    //chèn phần tử với key chưa tồn tại trong map

    mp.insert(pair<char,int>('c',300));

    dump(mp);

    //chèn phần tử với key đã tồn tại trong map

    mp.insert(pair<char,int>('a',3));

    dump(mp);

    //chèn phần tử bằng insert và make\_pair

    mp.insert(make\_pair('x',50));

    dump(mp);

    return 0;

}

a:100 b:200

a:100 b:200 c:300

a:100 b:200 c:300 // Do key đã tồn tại nên sẽ không được chèn

a:100 b:200 c:300 x:50

Chúng ta cũng có thể kiểm tra việc chèn phần tử đã thực hiện hay chưa bằng phương thức first() hoặc second() từ kết quả trả về của hàm như sau:

#include <iostream>

#include <map>

using namespace std;

int main ()

{

    /\*Khai báo và gán giá trị cho foo và bar\*/

    map<char,int> mp;

    mp['a']=100;

    mp['b']=200;

    //chèn phần tử với key chưa tồn tại trong map

    auto r = mp.insert(pair<char,int>('z',300));

    cout << r.second << "\n";   // Trả về 1 do được chèn

    //chèn phần tử với key tồn tại trong map,

    r = mp.insert(pair<char,int>('a',3));

    cout << r.second << "\n";  // Trả về 0 do không được chèn

    return 0;

}

### Thêm chèn 1 phần tử vào map trong C++ bằng hàm emplace

Cú pháp:

mp.emplace(k,v);

Trong đó

* mp là map ban đầu
* k,v là key và value.

Hàm map emplace sẽ thực hiện việc chèn phần tử nếu key của nó chưa tồn tại trong map. Ngược lại nếu key đó đã tòn tại, việc chèn thất bại và false sẽ được trả về

Và Hàm map emplace sẽ trả về một trình lặp trỏ tới vị trí phần tử vừa được chèn vào, nếu việc chèn thành công.

Lại nữa, phần tử trong map được sắp xếp theo thứ tự cụ thể trước khi được lưu, nên các phần tử cần chèn thêm vào map sẽ được tự động quyết định vị trí dựa trên bộ sắp xếp, và chúng ta sẽ không quyết định được vị trí cần chèn của phần tử.

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất map

void dump(map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    std::map<char,int> mymap;

    mymap.emplace('x',100);

    mymap.emplace('y',200);

    mymap.emplace('z',100);

    dump(mymap);

    return 0;

}

x:100 y:200 z:100

### Chèn nhiều phần tử vào map trong C++ bằng hàm insert

Cú pháp:

mp.insert(iterator\_first, iterator\_last);

Trong đó

* mp là map ban đầu
* iterator\_first và iterator\_last là các trình lặp xác định phạm vi chứa các phần tử cần chèn ở trong một map khác vào map ban đầu.

Lưu ý ở đây, các phần tử cần chèn được lấy ra từ một phạm vi trong một map khác, và phạm vi này được chỉ định thông qua 2 trình lặp như trên

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất map

void dump(map<int, const char\*>& mp)

{

    for (auto x: mp) {

        cout << x.first << ": " << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    /\* Khởi tạo map mp1\*/

    map<int, const char\*> mp1 =

    {

            {1, "a"},

            {2, "b"},

            {3, "c"},

            {4, "d"},

            {5, "e"},

    };

    // duyệt map mp1

    dump(mp1);

    /\*Khai báo map mp2\*/

    map<int, const char\*> mp2;

    /\*Chèn các phần tử từ đầu tới phần tử có key bằng 4

      từ mp1 vào mp2\*/

    mp2.insert(mp1.begin(),mp1.find(4));

    // duyệt map mp2

    dump(mp2);

    return 0;

}

1: a 2: b 3: c 4: d 5: e

1: a 2: b 3: c

## Xoá phần tử

### Xóa 1 phần tử trong map bằng hàm erase c++

Cú pháp:

mp.erase(itr);

OR

mp.erase(key);

Trong đó mp là map ban đầu, itr và key là trình lặp hoặc là khóa của phần tử cần xóa.

Nếu sử dụng erase(itr) thì hàm sẽ xóa đi phần tử tại vị trí mà trình lặp xác định bởi itr chỉ đến.

Và nếu sử dụng erase(key) thì hàm sẽ tìm phần tử có khóa bằng với key trong map và tiến hành xóa nó đi.

Khác với các container khác thì để chuyển trình lặp chỉ đến vị trí index thứ n trong map, chúng ta không thể đơn giản cộng vào n vào trình lặp, mà cần phải di chuyển lần lượt qua từng vị trí, bằng toán tử ++ với đủ số vòng lặp.

Lưu ý phần tử trong map được sắp xếp theo thứ tự cụ thể trước khi được lưu, nên các phần tử còn lại sau khi xóa phần tử trong map sẽ được tự động quyết định vị trí dựa trên bộ sắp xếp. Do đó chúng ta sẽ không quyết định được vị trí của các phần tử sau khi xóa đi phần tử từ map.

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất map

void dump(map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ": " << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    map<char,int> mp;

    mp['x']=100;

    mp['y']=200;

    mp['z']=300;

    mp['t']=400;

    dump(mp);

    /\*Tạo trình lặp trỏ đến vị trí đầu tiên của map\*/

    auto itr = mp.begin();

    ++itr; //Di chuyển trình lặp đến vị trí thứ nhất trong map

    //xóa phần tử tại vị trí itr chỉ đến (vị trí thứ nhất)

    mp.erase(itr);

    dump(mp);

    //Tìm vị trí và xóa phần tử có key bằng 'z' trong map

    mp.erase('z');

    dump(mp);

    //xóa phần tử có có key bằng 'y' trong map

    mp.erase('y');

    dump(mp);

    return 0;

}

t: 400 x: 100 y: 200 z: 300

t: 400 y: 200 z: 300

t: 400 y: 200

t: 400

Chúng ta cũng có thể kiểm tra số phần tử đã được xóa đi từ kết quả trả về của hàm như sau:

map<char,int> mp;

mp['x']=100;

mp['y']=200;

mp['z']=300;

mp['t']=400;

auto r = mp.erase('t'); /\*r là số phần tử có key bằng 't' bị xóa\*/

std::cout << r;

//1

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

Cú pháp:

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

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

Lưu ý là phạm vi xóa sẽ được tính từ iterator\_first đến trước iterator\_last, nghĩa là phần tử ở vị trí iterator\_first sẽ được xóa nhưng phần tử ở vị trí iterator\_last sẽ không bị xóa đi.

Và các giá trị trình lặp (iterator) này được tính sau khi các phần tử đã được sắp xếp và lưu trong map, chứ không phải là theo thứ tự các phần tử khi chúng ta khai báo map.

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất map

void dump(map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ": " << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    map<char,int> mp;

    mp['a']=100;

    mp['b']=200;

    mp['c']=300;

    mp['d']=400;

    mp['e']=500;

    mp['f']=600;

    dump(mp);

    /\*Khai báo phạm vi cần xóa\*/

    int start = 2, end =5;

    /\*Tạo các trình lặp trỏ tới start và end với giá trị ban đầu\*/

    auto itr\_start = mp.begin();

    auto itr\_end = mp.begin();

    /\*Thay đổi trình lặp tương ứng tới các vị trí start và end\*/

    for (int i=1; i <= start; i++ )

        ++itr\_start;

    for (int i=1; i <= end; i++ )

        ++itr\_end;

    //Sau đó dùng hàm erase để xóa phạm vi là xong

    mp.erase(itr\_start, itr\_end);

    dump(mp);

    return 0;

}

a: 100 b: 200 c: 300 d: 400 e: 500 f: 600

a: 100 b: 200 f: 600

## Sao chép, hoán đổi phần tử

### Sao chép map trong C++

Cú pháp:

std::map<k\_type, v\_type> mp\_des( mp\_src );

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

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất map

void dump(map<string,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ": " << x.second << " ";

    }

    cout << endl;

}

int main()

{

    map<string,int> mp\_src = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

    cout << "Orginary map" << endl;

    dump (mp\_src);

    /\*Sao chép map\*/

   map<string,int> mp\_des(mp\_src);

    cout << "Copy map" << endl;

    dump (mp\_des);

    return 0;

}

Orginary map

alpha: 10 beta: 20 gamma: 30

Copy map

alpha: 10 beta: 20 gamma: 30

### Hoán đổi 2 map trong C++

Cú pháp:

st1.swap(st2);

Trong đó st1 và st2 là 2 map cần hoán đổi nội dung cho nhau.

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất map

void dump(map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ": " << x.second << " ";

    }

    cout << endl;

}

int main()

{

    /\*Khai báo và gán giá trị cho foo và bar\*/

    map<char,int> foo,bar;

    foo['x']=100;

    foo['y']=200;

    bar['a']=11;

    bar['b']=22;

    bar['c']=33;

    cout << "Before swap" << endl;

    dump(foo);

    dump(bar);

    /\*Hoán đổi for cho bar\*/

    foo.swap(bar);

    cout << "After swap" << endl;

    dump(foo);

    dump(bar);

}

Before swap

x: 100 y: 200

a: 11 b: 22 c: 33

After swap

a: 11 b: 22 c: 33

x: 100 y: 200

Ngoài cách dùng hàm map swap, chúng ta cũng có thể dùng function template là std::swap để tiến hành hoán đổi 2 map 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 <map>

using namespace std;

//Tạo hàm xuất map

void dump(map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ": " << x.second << " ";

    }

    cout << endl;

}

int main()

{

    map<char,int> foo,bar;

    foo['x']=100;

    foo['y']=200;

    bar['a']=11;

    bar['b']=22;

    bar['c']=33;

    cout << "Before swap" << endl;

    dump(foo);

    dump(bar);

    swap(foo,bar);

    cout << "After swap" << endl;

    dump(foo);

    dump(bar);

}

Before swap

x: 100 y: 200

a: 11 b: 22 c: 33

After swap

a: 11 b: 22 c: 33

x: 100 y: 200

## Tìm phần tử

### Tìm phần tử trong map C++ bằng hàm find

Cú pháp:

mp.find(key);

Trong đó key là khóa của phần tử cần tìm trong map mp.

Hàm find() sẽ trả về trình lặp trỏ đến vị trí phần tử, nếu nó tồn tại trong map. Và nếu phần tử đó không tồn tại, hàm sẽ trả về trình lặp trỏ đến vị trí cuối cùng trong map.

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất map

void dump(map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main() {

    map<char,int> mp;

    mp.insert(make\_pair('a', 1));

    mp.insert(make\_pair('b', 2));

    mp.insert(make\_pair('c', 3));

    mp.insert(make\_pair('d', 1));

    mp.insert(make\_pair('e', 2));

    dump(mp);

    //Tìm phần tử có khóa bằng 'c' trong map

    auto ret = mp.equal\_range('c');

    //Xóa phần tử vừa tìm thấy

    mp.erase (ret.first,ret.second);

    dump(mp);

    return 0;

}

a:1 b:2 c:3 d:1 e:2

a:1 b:2 d:1 e:2

### Tìm phần tử trong map C++ bằng hàm equal\_range

Cú pháp:

mp.equal\_range(key);

Trong đó key là khóa của phần tử cần tìm trong map mp.

Hàm equal\_range() sẽ trả về một cặp giá trị, với giá trị đầu tiên trỏ đến đầu phạm vi, và giá trị thứ hai trỏ đến cuối phạm vi chứa tất cả các phần tử có khóa giống khóa chỉ định.

Tuy nhiên do trong map thì các khóa là duy nhất, do đó nếu khóa tồn tại trong map thì hàm equal\_range sẽ trả về một khoảng chứa giá trị duy nhất đó mà thôi.

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất map

void dump(map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main() {

    map<char,int> mp;

    mp.insert(make\_pair('a', 1));

    mp.insert(make\_pair('b', 2));

    mp.insert(make\_pair('c', 3));

    mp.insert(make\_pair('d', 1));

    mp.insert(make\_pair('e', 2));

    dump(mp);

    //Tìm phần tử có khóa bằng 'c' trong map

    auto ret = mp.equal\_range('c');

    cout << "lower bound points to: ";

    cout << ret.first->first << " => " << ret.first->second << '\n';

    cout << "upper bound points to: ";

    cout << ret.second->first << " => " << ret.second->second << '\n';

    return 0;

}

a:1 b:2 c:3 d:1 e:2

lower bound points to: c => 3

upper bound points to: d => 1

### Tìm phần tử trong map C++ bằng hàm lower\_bound

Cú pháp:

mp.lower\_bound(key);

Trong đó key là khóa của phần tử cần tìm trong map mp.

Hàm lower\_bound() sẽ trả về trình lặp trỏ đến vị trí phần tử đầu tiên có khóa lớn hơn hoặc bằng với khóa chỉ định. Và nếu không tìm thấy, hàm sẽ trả về trình lặp trỏ đến vị trí cuối cùng trong map.

Và trong trường hợp chỉ có một phần tử trong map có khóa giống với khóa chỉ định thì hàm lower\_bound sẽ trả về con trỏ chỉ đến phần tử đó.

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất map

void dump(map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main() {

    map<char,int> mp;

    mp.insert(make\_pair('a', 1));

    mp.insert(make\_pair('b', 2));

    mp.insert(make\_pair('c', 3));

    mp.insert(make\_pair('d', 1));

    mp.insert(make\_pair('e', 2));

    mp.insert(make\_pair('f', 3));

    dump(mp);

    /\*Tìm vị trí phần tử đầu tiên có khóa lớn hơn hoặc bằng 'b' trong map\*/

    auto itr1 = mp.lower\_bound('b'); // itr1 trỏ đến b:2

    //Tìm vị trí phần tử đầu tiên có khóa lớn hơn 'e' trong map

    auto itr2 = mp.upper\_bound('e'); // itr2 trỏ đến f:3

    //Xóa các phần tử trong phạm vi [itr1, itr2)

    mp.erase (itr1, itr2);

    dump(mp);

    return 0;

}

a:1 b:2 c:3 d:1 e:2 f:3

a:1 f:3

### Tìm phần tử trong map C++ bằng hàm upper\_bound

Hai hàm upper\_bound và lower\_bound đều tìm kiếm phần tử trong một danh sách đã xắp xếp (theo chiều tăng dần như set, map). Nhưng upper\_bound tìm kiếm phần tử đầu tiên có giá trị lớn hơn > giá trị tìm kiếm trong khi lower\_bound tìm kiếm phần tử đầu tiên có giá trị lớn hơn hoặc bằng giá trị trị tìm kiếm.

Tham khảo: <https://www.youtube.com/watch?v=dB2DWSKGLj8> đoạn 38p30’

Cú pháp:

mp.upper\_bound(key);

Trong đó key là khóa của phần tử cần tìm trong map mp.

Hàm lower\_bound() sẽ trả về trình lặp trỏ đến vị trí phần tử đầu tiên có khóa lớn hơn khóa chỉ định. Và nếu không tìm thấy, hàm sẽ trả về trình lặp trỏ đến vị trí cuối cùng trong map.

VD:

#include <iostream>

#include <map>

using namespace std;

int main() {

    map<char,int> mp;

    mp['a']=100;

    mp['b']=200;

    mp['c']=300;

    mp['d']=300;

    mp['e']=300;

    mp['f']=300;

    /\*Duyệt map mp\*/

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl; //a:1 b:2 c:3 d:1 e:2

    //Tìm phần tử có khóa bằng 'b' trong map

    auto itr1 = mp.lower\_bound('b'); // itr1 trỏ đến b:200

    //Tìm vị trí phần tử  đầu tiên có khóa lớn hơn 'e' trong map

    auto itr2 = mp.upper\_bound('e'); // itr2 trỏ đến f:300

    //In các phần tử trong phạm vi (itr1, itr2)

    for (auto it=itr1; it!=itr2; ++it)

        cout << (\*it).first << ":" << (\*it).second << ' ';

    return 0;

}

a:100 b:200 c:300 d:300 e:300 f:300

b:200 c:300 d:300 e:300

### Đếm số lần xuất hiện của phần tử trong map C++ bằng hàm count

Cú pháp:

mp.count(key);

Trong đó key là khóa của phần tử cần đếm số lần xuất hiện trong map mp.

Do trong map các khóa là duy nhất, nên một phần tử nếu tồn tại cũng chỉ có xuất hiện 1 lần duy nhất trong map mà thôi. Do vậy, kết quả trả về của hàm count() cũng chỉ là 0 tương ứng với phần tử không tồn tại, hoặc 1 tương ứng với phần tử tồn tại trong map đó.

VD:

#include <iostream>

#include <map>

using namespace std;

int main() {

    map<char,int> mp;

    mp['a']=100;

    mp['b']=200;

    mp['c']=300;

    //Đếm số lần xuất hiện của phần tử tồn tại trong map

    cout << mp.count('b') <<endl;

    //Đếm số lần xuất hiện của phần tử không tồn tại trong map

    cout << mp.count('f') <<endl;

    return 0;

}

1

0

## Map trống

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

Cú pháp:

mp.empty();

Trong đó mp là map cần kiểm tra.

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

VD:

#include <iostream>

#include <map>

using namespace std;

int main ()

{

    map<int, char> mp;

    if( mp.empty() )

        cout << "empty.\n";

    else

        cout << "not empty.\n";

    map<string,int> mp2 = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

    if( mp2.empty() )

        cout << "empty.\n";

    else

        cout << "not empty.\n";

    return 0;

}

empty.

not empty.

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

Cú pháp:

mp.clear();

Trong đó mp là map cần làm trống.

Khác với vector thì hàm map clear ngoài việc làm trống map chỉ định (xóa đi tất cả phần tử) thì còn giải phóng bộ nhớ sử dụng cho việc lưu trữ dữ liệu đã dùng.

VD:

#include <iostream>

#include <map>

using namespace std;

int main ()

{

    map<char,int> mymap;

    mymap['x']=100;

    mymap['y']=200;

    mymap['z']=300;

    cout << mymap.size() << "\n";       //3

    mymap.clear();

    cout << mymap.size() << "\n";       //0

}

# Multimap

## Tổng quan

### Multimap trong c++ là gì

Multimap trong C++ là một tập hợp các phần tử được sắp xếp theo thứ tự cụ thể, với mỗi phần tử được hình thành bởi sự kết hợp của một cặp khóa và giá trị (key & value), trong đó nhiều phần tử có thể có các khóa giống nhau.

Trong multimap, các khóa (key) được sử dụng để sắp xếp và xác định các giá trị (value) tương ứng được liên kết với nó. Mỗi khóa trong multimap được phép trùng lặp khiến nhiều phần tử có thể có các khóa tương đương. Do vậy, một khóa trong multimap có thể dùng để xác định nhiều giá trị được liên kết với nó.

Về mặt nội bộ, các phần tử trong multimap luôn được sắp xếp theo khóa của nó theo thứ tự cụ thể một cách nghiêm ngặt, được chỉ ra bởi đối tượng so sánh nội bộ của multimap. Nếu bạn thêm các phần tử mới không theo thứ tự cụ thể vào một multimap, chúng sẽ tự động sắp xếp lại theo khóa trước khi được lưu trữ nội bộ.

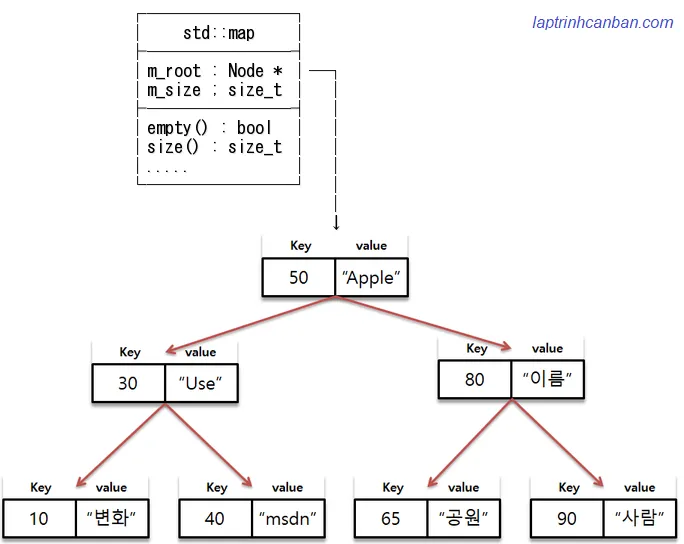
Khác với map không cho phép các khóa trùng lặp thì trong một multimap nhiều phần tử có thể có các khóa giống nhau.

Về mặt tốc độ xử lý thì multimap được cho là chậm hơn unordered\_map khi cần truy cập vào một phần tử ngẫu nhiên, nhưng lại nhanh hơn unordered\_map trong việc thực thi các trình lặp trỏ tới chúng.

### Cấu trúc dữ liệu multimap trong c++

Tương tự như kiểu map thì cấu trúc dữ liệu multimap trong C++ thuộc dạng Red–black tree (cây đỏ đen) - một cây nhị phân, là một cấu trúc dữ liệu trong khoa học máy tính để tổ chức các thành phần dữ liệu có thể so sánh.

Cụ thể thì cấu trúc dữ liệu multimap trong C++ có được thể hiện như ví dụ dưới đây. Lưu ý là cấu trúc này có thể khác một chút so với thực tế cấu trúc trong môi trường máy của bạn.



Trong các Node sẽ lưu giữ cặp khóa:giá trị (key & value) cũng như con trỏ của các Node con (trái, phải) của nó.

Các giá trị trong Node thỏa mãn điều kiện giá trị của Node con bên trái <= Giá trị Node cha <= Giá trị của Node con bên phải. Do trong multimap cho phép khóa trùng nhau nên dấu <= được sử dụng.

Độ sâu của các Node bằng nhau và cây Node thì cân bằng.

Nhờ vào cấu trúc dữ liệu kiểu này mà chúng ta có thể tìm kiếm nhị phân trong multimap, qua đó có thể tìm kiếm trong multimap với tốc độ cao O(Log N).

### std::multimap trong C++

std::multimap trong C++ là một thư viện chuẩn được sử dụng để xử lý multimap trong C++.

Giống như std::map thì std::multimap được cài sẵn trong header file map và để sử dụng được chức năng này, chúng ta cần thêm dòng #include <map> vào đầu chương trình.

#include <map>

int main()

{

    std::multimap<std::string, int> mp;

    std::multimap<char, double> mp2;

}

## Khai báo multimap trong C++

### Khai báo 1 multimap trong C++

Cú pháp:

std::multimap<k\_type, v\_type> mp;

Trong đó mp là tên biến multimap và k\_type, v\_type lần lượt là kiểu dữ liệu của key và value. Cách viết sử dụng cặp dấu <> như trên được viết theo cú pháp khi sử dụng chức năng template của C++ mà chúng ta sẽ cùng học trong các chuyên đề sau.

Lưu ý, mặc dù chúng ta có thể dùng bất cứ kiểu dữ liệu nào có trong C++ để khai báo type, tuy nhiên do trong multimap các phần tử cần phải được sắp xếp, nên kiểu của chúng cũng phải là kiểu dữ liệu có thể được so sánh.

Đối với các kiểu dữ liệu nguyên thủy như char, int, double chẳng hạn thì chúng vốn có thể tự so sánh được, nhưng nếu chúng ta sử dụng các kiểu dữ liệu không phải là kiểu dữ liệu nguyên thủy, ví dụ như cấu trúc hoặc class tự tạo chẳng hạn, thì bắt buộc phải tự định nghĩa toán tử so sánh nội bộ operator<() để làm rõ quan hệ lớn nhỏ giữa chúng.

VD:

#include <iostream>

#include <map>

using namespace std;

int main()

{

    multimap<double, string> user;

    multimap<int, char> info;

}

Còn khi khai báo 1 multimap không thuộc kiểu dữ liệu nguyên thủy, ví dụ như struct chẳng hạn thì chúng ta phải tự tạo ra toán tử so sánh nội bộ operator<() để làm rõ quan hệ lớn nhỏ giữa các phần tử như ví dụ sau:

struct Person {

    string m\_name;

    int    m\_height;

};

// Định nghĩa toán tử so sánh nội bộ của struct

bool operator<(const Person &lhs, const Person &rhs)

{

    return lhs.m\_name < rhs.m\_name;

}

/\*Khai báo multimap thuộc kiểu struct\*/

std::multimap<Person, int> mp;

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

Cú pháp:

using namespace std;

multimap<k\_type, v\_type> name1, name2, name3, ... ;

VD:

#include <iostream>

#include <map>

using namespace std;

int main()

{

    multimap<string, int> name, job, sex;

    multimap<int, char> age;

}

## Khởi tạo multimap trong C++

Cú pháp:

std::multimap<k\_type, v\_type> mp = { {k1, v1}, {k2, v2}, {k3, v3}, ...};

Trong đó

* mp là tên biến multimap
* k\_type, v\_type là kiểu dữ liệu của key và value
* k,v là các cặp key và value

Lưu ý khác với map thì khóa trong multimap có thể trùng lặp, nên nếu chúng ta chỉ định các phần tử có cùng khóa thì tất cả chúng cũng sẽ được lưu vào map.

VD:

#include <iostream>

#include <map>

using namespace std;

int main ()

{

    multimap<string,int> mp = {

                { "alpha", 10 },

                { "alpha", 20 },

                { "beta", 20 },

                { "alpha", 10 },

                { "gamma", 30 }

            };

    for (auto& x: mp) {

        cout << x.first << ": " << x.second << '\n';

    }

    return 0;

}

alpha: 10

alpha: 20

alpha: 10

beta: 20

gamma: 30

## Truy cập phần tử trong multimap C++

Để truy cập vào phần tử trong multimap, cần sử dụng vòng lặp

#include <iostream>

#include <map>

using namespace std;

int main ()

{

    multimap<string,int> mp = {

                { "alpha", 10 },

                { "alpha", 20 },

                { "beta", 20 },

                { "alpha", 10 },

                { "gamma", 30 }

            };

    for (auto& x: mp) {

        if(x.first == "alpha") cout << x.first <<": " << x.second << '\n';

    }

    return 0;

}

## multimap vs map, unordered\_map trong c++

Trong C++ tồn tại 3 loại map gồm map, unordered\_map và multimap. Ba loại map này đều có phần tử được tạo thành bởi một cặp khóa và giá trị (key & value), tuy nhiên giữa chúng có những điểm hoàn toàn khác nhau như sau:

Phần tử trong map và multimap được sắp xếp theo thứ tự, còn unordered\_map thì không có thứ tự.

Phần tử trong map và multimap được lưu theo thứ tự của khóa, còn trong unordered\_map thì theo thứ tự giá trị hash của khóa.

map và multimap tuân theo cấu trúc cây nhị phân Red–black tree, còn unordered\_map thì theo giá trị trong bảng hash.

Key trong map và unordered\_map là duy nhất, còn trong multimap thì có thể trùng nhau.

Về tốc độ xử lý thì unordered\_map có tốc độ truy cập phần tử ngẫu nhiên nhanh hơn map và multimap, tuy nhiên tốc độ xử lý vòng lặp lại kém hơn so với 2 anh em của nó.

## Duyệt phần tử

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

Cú pháp:

for ( auto& x : mp) {

cout << x.first << “: “ << x.second << endl;

}

Trong đó:

* mp là tên multimap.
* auto là kiểu suy luận giúp tự xác định kiểu dữ liệu của giá trị lấy từ multimap.
* x là tên một biến dùng để gán từng phần tử được lấy từ multimap.
* x.first và x.second lần lượt được sử dụng để lấy key và value của phần tử

VD:

#include <iostream>

#include <map>

using namespace std;

int main()

{

    multimap<string,int> mp = {

                { "alpha", 10 },

                { "beta", 20 },

                { "alpha", 20 },

                { "gamma", 30 } };

    for (auto x: mp) {

        cout << x.first << ": " << x.second << endl;

    }

}

alpha: 10

alpha: 20

beta: 20

gamma: 30

### Duyệt multimap trong C++ bằng iterator

Cú pháp:

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

cout << itr->first << ": "<< itr->second << "\n";

}

Trong đó:

* mp là tên multimap
* itr là tên iterator dùng để trỏ đến phần tử
* itr->first và itr->second lần lượt được sử dụng để lấy key và value của phần tử

VD:

#include <iostream>

#include <map>

using namespace std;

int main()

{

    multimap<string,int> mp = {

                { "alpha", 10 },

                { "beta", 20 },

                { "alpha", 20 },

                { "gamma", 30 } };

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

        cout  << itr->first << ": "<< itr->second << "\n";

    }

}

alpha: 10

alpha: 20

beta: 20

gamma: 30

## Lấy kích thước

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

Cú pháp:

mp.size();

Trong đó mp là multimap cần lấy kích thước (số phần tử) chứa trong nó.

VD:

#include <iostream>

#include <map>

using namespace std;

int main ()

{

    multimap<string,int> mp = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

    cout<< mp.size() <<endl;

    return 0;

}

//3

Khác với map thì trong multimap cho phép tồn tại phần tử trùng lặp, nên khi đếm số phần tử có trong multiset thì hàm size() sẽ trả về toàn bộ các phần tử mà chúng ta đã dùng để khai báo map. Ví dụ:

#include <iostream>

#include <map>

using namespace std;

int main ()

{

    multimap<string,int> mp = {

                { "alpha", 20 },

                { "beta", 20 },

                { "alpha", 10 },

                { "gamma", 30 },

                { "alpha", 10 },

                };

    for (auto x: mp) {

        cout << x.first << ": " << x.second << endl;

    }

    cout<< mp.size() <<endl;

    return 0;

}

alpha: 20

alpha: 10

alpha: 10

beta: 20

gamma: 30

5

## Thêm phần tử

### Sự khác biệt khi thêm chèn phần tử vào multimap với các container khác

Để thêm chèn phần tử vào multimap trong C++, chúng ta sử dụng tới hàm thành viên insert() hoặc emplace(). Do khi chèn phần tử vào multimap thì vị trí chèn sẽ được tự động quyết định tùy thuộc vào bộ sắp xếp trong multimap, nên lưu ý khác với các containers khác như list hay vector thì trong multimap không tồn tại các hàm push\_back() hay push\_front() để thêm phần tử vào đầu hay cuối multimap.

Và khác với map thì trong multimap cũng không tồn tại toán tử [] nên chúng ta chỉ có thể dùng 2 hàm thành viên ở trên để thêm chèn phần tử mà thôi.

### Thêm chèn 1 phần tử vào multimap trong C++ bằng hàm insert

Cú pháp:

mp.insert(std::pair<k\_type,x\_type>(k,v));

Trong đó

* mp là multimap ban đầu
* pair<k\_type,x\_type>(k,v) sử dụng để chỉ định key và value của phần tử cần thêm, trong đó k\_type,x\_type là kiểu và k,v là key và value.

Trong trường hợp không rõ kiểu, hoặc muốn rút bỏ chỉ định kiểu của key và value, chúng ta có thể dùng hàm make\_pair() để thay thế cho pair() trong hàm insert, với cú pháp sau đây:

mp.insert(std::make\_pair(k,v));

Hàm multimap insert sẽ trả về một trình lặp trỏ tới vị trí phần tử vừa được chèn vào.

Lại nữa, phần tử trong multimap được sắp xếp theo thứ tự cụ thể trước khi được lưu, nên các phần tử cần chèn thêm vào multimap sẽ được tự động quyết định vị trí dựa trên bộ sắp xếp, và chúng ta sẽ không quyết định được vị trí cần chèn của phần tử.

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất multimap

void dump(multimap<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ": " << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    /\*Khởi tạo multimap\*/

    multimap<char,int> mp = {

            { 'a', 10 },

            { 'b', 20 },

            { 'c', 30 } };

    // duyệt multimap ban đầu

    dump(mp);

    //chèn phần tử bằng insert và pair

    mp.insert(pair<char,int>('z',40));

    dump(mp);

    //chèn phần tử bằng insert và make\_pair

    mp.insert(make\_pair('x',50));

    dump(mp);

    return 0;

}

a: 10 b: 20 c: 30

a: 10 b: 20 c: 30 z: 40

a: 10 b: 20 c: 30 x: 50 z: 40

Chúng ta cũng có thể lấy vị trí đã chèn của phần tử vào multimap bằng trình lặp thu về từ hàm, rồi truy xuất nó thông qua trình lặp như sau:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất multimap

void dump(multimap<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ": " << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    /\*Khởi tạo multimap\*/

    multimap<char,int> mp = {

            { 'a', 10 },

            { 'b', 20 },

            { 'c', 30 } };

    auto itr = mp.insert(make\_pair('x',50));

    cout << (\*itr).first << " => " << (\*itr).second ;

    return 0;

}

x => 50

### Thêm chèn 1 phần tử vào multimap trong C++ bằng hàm emplace

Cú pháp:

mp.emplace(k,v);

Trong đó

* mp là multimap ban đầu
* k,v là key và value.

Hàm multimap emplace sẽ trả về một trình lặp trỏ tới vị trí phần tử vừa được chèn vào.

Lại nữa, phần tử trong multimap được sắp xếp theo thứ tự cụ thể trước khi được lưu, nên các phần tử cần chèn thêm vào multimap sẽ được tự động quyết định vị trí dựa trên bộ sắp xếp, và chúng ta sẽ không quyết định được vị trí cần chèn của phần tử.

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất multimap

void dump(multimap<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    /\*Khởi tạo multimap\*/

    multimap<char,int> mp = {

            { 'a', 10 },

            { 'b', 20 },

            { 'c', 30 } };

    // duyệt multimap ban đầu

    dump(mp);

    //chèn phần tử bằng emplace

    mp.emplace('z',40);

    dump(mp);

    //chèn và lấy khóa và giá trị của phần tử vừa chèn

    auto itr = mp.emplace('x',50);

    cout << (\*itr).first << " => " << (\*itr).second ;

    return 0;

}

a:10 b:20 c:30

a:10 b:20 c:30 z:40

x => 50

### Chèn nhiều phần tử vào multimap trong C++ bằng hàm insert

Cú pháp:

mp.insert(iterator\_first, iterator\_last);

Trong đó

* mp là multimap ban đầu
* iterator\_first và iterator\_last là các trình lặp xác định phạm vi chứa các phần tử cần chèn ở trong một multimap khác vào multimap ban đầu.

Lưu ý ở đây, các phần tử cần chèn được lấy ra từ một phạm vi trong một multimap khác, và phạm vi này được chỉ định thông qua 2 trình lặp như trên

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất multimap

void dump(multimap<int, const char\*>& mp)

{

    for (auto x: mp) {

        cout << x.first << ": " << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    /\* Khởi tạo multimap mp1\*/

    multimap<int, const char\*> mp1 =

    {

            {1, "a"},

            {2, "b"},

            {3, "c"},

            {4, "d"},

            {5, "e"},

    };

    // duyệt multimap mp1

    cout <<"mp1= ";

    dump(mp1);

    /\*Khai báo multimap mp2\*/

    multimap<int, const char\*> mp2;

    /\*Chèn các phần tử từ đầu tới phần tử có key bằng 4

      từ mp1 vào mp2\*/

    mp2.insert(mp1.begin(),mp1.find(4));

    // duyệt multimap mp2

    cout <<"mp2= ";

    dump(mp2);

    return 0;

}

mp1= 1: a 2: b 3: c 4: d 5: e

mp2= 1: a 2: b 3: c

## Xoá phần tử

### Xóa 1 phần tử trong multimap bằng hàm erase c++

Cú pháp:

mp.erase(itr);

OR

mp.erase(key);

Trong đó mp là multimap ban đầu, itr và key là trình lặp hoặc là khóa của phần tử cần xóa.

Nếu sử dụng erase(itr) thì hàm sẽ xóa đi phần tử tại vị trí mà trình lặp xác định bởi itr chỉ đến.

Và nếu sử dụng erase(key) thì hàm sẽ tìm phần tử có khóa bằng với key trong multimap và tiến hành xóa nó đi.

Hàm multimap erase sẽ trả về số lượng phần tử đã được xóa đi từ multimap ban đầu.

Lưu ý phần tử trong multimap được sắp xếp theo thứ tự cụ thể trước khi được lưu, nên các phần tử còn lại sau khi xóa phần tử trong multimap sẽ được tự động quyết định vị trí dựa trên bộ sắp xếp. Do đó chúng ta sẽ không quyết định được vị trí của các phần tử sau khi xóa đi phần tử từ multimap.

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất multimap

void dump(multimap<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    multimap<char,int> mp;

    mp.insert(make\_pair('a', 1));

    mp.insert(make\_pair('b', 2));

    mp.insert(make\_pair('c', 5));

    mp.insert(make\_pair('d', 5));

    mp.insert(make\_pair('e', 6));

    dump(mp);

    /\*Tìm trình lặp trỏ đến phần tử có khóa bằng 'c' \*/

    auto itr = mp.find('c');

    //xóa phần tử tại vị trí itr chỉ đến

    mp.erase(itr);

    dump(mp);

    //xóa phần tử có khóa bằng 'd' trong multimap

    mp.erase('d');

    dump(mp);

    return 0;

}

a:1 b:2 c:5 d:5 e:6

a:1 b:2 d:5 e:6

a:1 b:2 e:6

Chúng ta cũng có thể kiểm tra số phần tử đã được xóa đi từ kết quả trả về của hàm như sau:

VD:

auto r = mp.erase('d'); /\*r là số phần tử có key bằng 'd' bị xóa\*/

cout << r; //2

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

Cú pháp:

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

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

Lưu ý là phạm vi xóa bằng [first,last) sẽ được tính từ iterator\_first đến trước iterator\_last, nghĩa là phần tử ở vị trí iterator\_first sẽ được xóa nhưng phần tử ở vị trí iterator\_last sẽ không bị xóa đi.

Và các giá trị trình lặp (iterator) này được tính sau khi các phần tử đã được sắp xếp và lưu trong multimap, chứ không phải là theo thứ tự các phần tử khi chúng ta khai báo multimap.

Khác với các container khác thì để chuyển trình lặp chỉ đến vị trí index thứ n trong multiset, chúng ta không thể đơn giản cộng vào n vào trình lặp, mà cần phải di chuyển lần lượt qua từng vị trí, bằng toán tử ++ với đủ số vòng lặp.

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất multimap

void dump(multimap<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    multimap<char,int> mp;

    mp.insert(make\_pair('a', 1));

    mp.insert(make\_pair('b', 2));

    mp.insert(make\_pair('c', 5));

    mp.insert(make\_pair('d', 5));

    mp.insert(make\_pair('e', 6));

    dump(mp);

    /\*Khai báo phạm vi cần xóa\*/

    int start = 2, end =5;

    /\*Tạo các trình lặp trỏ tới start và end với giá trị ban đầu\*/

    auto itr\_start = mp.begin();

    auto itr\_end = mp.begin();

    /\*Thay đổi trình lặp tương ứng tới các vị trí start và end\*/

    for (int i=1; i <= start; i++ )

        ++itr\_start;

    for (int i=1; i <= end; i++ )

        ++itr\_end;

    //Sau đó dùng hàm erase để xóa phạm vi là xong

    mp.erase(itr\_start, itr\_end);

    dump(mp);

    return 0;

}

a:1 b:2 c:5 d:5 e:6

a:1 b:2

## Sao chép, hoán đổi phần tử

### Sao chép multimap trong C++

multimap 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 multimap vào một multimap mới.

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

std::multimap<k\_type, v\_type> mp\_des( mp\_src );

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

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất multimap

void dump(multimap<string,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ": " << x.second << " ";

    }

    cout << endl;

}

int main()

{

    multimap<string,int> mp\_src = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

    cout << "Orginary multimap" << endl;

    dump (mp\_src);

    /\*Sao chép multimap\*/

   multimap<string,int> mp\_des(mp\_src);

    cout << "Copy multimap" << endl;

    dump (mp\_des);

    return 0;

}

Orginary multimap

alpha: 10 beta: 20 gamma: 30

Copy multimap

alpha: 10 beta: 20 gamma: 30

### Hoán đổi 2 multimap trong C++

Cú pháp:

st1.swap(st2);

Trong đó st1 và st2 là 2 multimap cần hoán đổi nội dung cho nhau.

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất multimap

void dump(multimap<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ": " << x.second << " ";

    }

    cout << endl;

}

int main()

{

    /\*Khởi tạo các multimap là foo và bar\*/

    multimap<char,int> foo = {

            { 'a', 10 },

            { 'b', 20 },

            { 'c', 30 } };

    multimap<char,int> bar = {

            { 'x', 11 },

            { 'y', 21 } };

    cout << "Before swap" << endl;

    dump(foo);

    dump(bar);

    /\*Hoán đổi for cho bar\*/

    foo.swap(bar);

    cout << "After swap" << endl;

    dump(foo);

    dump(bar);

}

Before swap

a: 10 b: 20 c: 30

x: 11 y: 21

After swap

x: 11 y: 21

a: 10 b: 20 c: 30

Ngoài cách dùng hàm multimap swap, chúng ta cũng có thể dùng function template là std::swap để tiến hành hoán đổi 2 multimap với nhau, cũng như là để hoán đổi các đối tượng khác như multimap, 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 <map>

#include <utility>

using namespace std;

//Tạo hàm xuất multimap

void dump(multimap<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ": " << x.second << " ";

    }

    cout << endl;

}

int main()

{

    /\*Khởi tạo các multimap là foo và bar\*/

    multimap<char,int> foo = {

            { 'a', 10 },

            { 'b', 20 },

            { 'c', 30 } };

    multimap<char,int> bar = {

            { 'x', 11 },

            { 'y', 21 }};

    cout << "Before swap" << endl;

    dump(foo);

    dump(bar);

    /\*Hoán đổi for cho bar\*/

    swap(foo,bar);

    cout << "After swap" << endl;

    dump(foo);

    dump(bar);

}

Before swap

a: 10 b: 20 c: 30

x: 11 y: 21

After swap

x: 11 y: 21

a: 10 b: 20 c: 30

## Tìm phần tử

### Tìm phần tử trong multimap C++ bằng hàm find

Cú pháp:

mp.find(key);

Trong đó key là khóa của phần tử cần tìm trong multimap mp.

Hàm find() sẽ trả về trình lặp trỏ đến vị trí phần tử, nếu nó tồn tại trong multimap. Và nếu phần tử đó không tồn tại, hàm sẽ trả về trình lặp trỏ đến vị trí cuối cùng trong multimap.

Lưu ý là hàm find() chỉ trả về một trình lặp cho một phần tử duy nhất mà thôi. Để có được toàn bộ phạm vi các phần tử có khóa chỉ định, hãy dùng hàm equal\_range mà Kiyoshi sẽ giới thiệu ở phần dưới.

Bằng cách ứng dụng hàm find(), chúng ta có thể tìm ra vị trí của một phần tử thông qua khóa trong multimap, rồi kết hợp với hàm clear() để xóa nó đi như ví dụ cụ thể sau đây:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất multimap

void dump(multimap<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main() {

    multimap<char,int> mp;

    mp.insert(make\_pair('a', 1));

    mp.insert(make\_pair('b', 2));

    mp.insert(make\_pair('e', 5));

    mp.insert(make\_pair('c', 3));

    mp.insert(make\_pair('d', 1));

    mp.insert(make\_pair('e', 2));

    mp.insert(make\_pair('f', 3));

    mp.insert(make\_pair('e', 8));

    dump(mp);

    //Tìm phần tử có khóa bằng 'b' trong multimap

    auto itr = mp.find('e');

    //Xóa phần tử vừa tìm thấy

    mp.erase (itr);

    dump(mp);

    return 0;

}

a:1 b:2 c:3 d:1 e:5 e:2 e:8 f:3

a:1 b:2 c:3 d:1 e:2 e:8 f:3

Giống như trên, mặc dù có 3 phần tử có khóa ‘e’ giống với khóa chỉ định, nhưng hàm find() chỉ trả về trình lặp trỏ đến phần tử đầu tiên mà thôi.

### Tìm phần tử trong multimap C++ bằng hàm equal\_range

Cú pháp:

mp.equal\_range(key);

Trong đó key là khóa của phần tử cần tìm trong multimap mp.

Hàm equal\_range() sẽ trả về một cặp giá trị, với giá trị đầu tiên trỏ đến đầu phạm vi, và giá trị thứ hai trỏ đến cuối phạm vi chứa tất cả các phần tử có khóa giống khóa chỉ định.

Bằng cách ứng dụng hàm equal\_range(), chúng ta có thể tìm ra phạm vi chứa kết quả tìm kiếm, rồi kết hợp với hàm clear() để xóa nó đi như ví dụ cụ thể sau đây:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất multimap

void dump(multimap<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main() {

    multimap<char,int> mp;

    mp.insert(make\_pair('a', 1));

    mp.insert(make\_pair('b', 2));

    mp.insert(make\_pair('e', 5));

    mp.insert(make\_pair('c', 3));

    mp.insert(make\_pair('d', 1));

    mp.insert(make\_pair('e', 2));

    mp.insert(make\_pair('f', 3));

    mp.insert(make\_pair('e', 8));

    dump(mp);

    //Tìm phần tử có khóa bằng 'b' trong multimap

    auto ret = mp.equal\_range('e');

    //Xóa phần tử vừa tìm thấy

    mp.erase (ret.first,ret.second);

    dump(mp);

    return 0;

}

a:1 b:2 c:3 d:1 e:5 e:2 e:8 f:3

a:1 b:2 c:3 d:1 f:3

Giống như trên, toàn bộ phạm vi chứa các phần tử có khóa giống khóa ‘e’ chỉ định đã được xác định bởi hàm equal\_range() và được xóa đi.

### Tìm phần tử trong multimap C++ bằng hàm lower\_bound

Cú pháp:

mp.lower\_bound(key);

Trong đó key là khóa của phần tử cần tìm trong multimap mp.

Hàm lower\_bound() sẽ trả về trình lặp trỏ đến vị trí phần tử đầu tiên có khóa lớn hơn hoặc bằng với khóa chỉ định. Và nếu không tìm thấy, hàm sẽ trả về trình lặp trỏ đến vị trí cuối cùng trong map.

Trong trường hợp chỉ có một phần tử trong multimap có khóa giống với khóa chỉ định thì hàm lower\_bound sẽ trả về con trỏ chỉ đến phần tử đó. Còn nếu có nhiều phần tử có khóa giống với khóa chỉ định, vị trí của phần tử đầu tiên có khóa lớn hơn khóa chỉ định sẽ được hàm lower\_bound() trả về.

VD:

#include <iostream>

#include <map>

using namespace std;

//Tạo hàm xuất multimap

void dump(multimap<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main() {

    multimap<char,int> mp;

    mp.insert(make\_pair('a', 1));

    mp.insert(make\_pair('b', 2));

    mp.insert(make\_pair('e', 5));

    mp.insert(make\_pair('c', 3));

    mp.insert(make\_pair('d', 1));

    mp.insert(make\_pair('e', 2));

    mp.insert(make\_pair('f', 3));

    mp.insert(make\_pair('e', 8));

    mp.insert(make\_pair('b', 9));

    dump(mp);   //a:1 b:2 b:9 c:3 d:1 e:5 e:2 e:8 f:3

    /\*Tìm vị trí phần tử đầu tiên có khóa lớn hơn hoặc bằng 'b' trong multimap\*/

    auto itr1 = mp.lower\_bound('b'); // itr1 trỏ đến b:2

    //Tìm vị trí phần tử đầu tiên có khóa lớn hơn 'e' trong multimap

    auto itr2 = mp.upper\_bound('e'); // itr2 trỏ đến f:3

    //Xóa các phần tử trong phạm vi [itr1, itr2)

    mp.erase (itr1, itr2);

    dump(mp);   //a:1 f:3

    return 0;

}

a:1 b:2 c:3 d:1 e:2 f:3

a:1 f:3

### Tìm phần tử trong multimap C++ bằng hàm upper\_bound

Cú pháp:

mp.upper\_bound(key);

Trong đó key là khóa của phần tử cần tìm trong multimap mp.

Hàm upper\_bound() sẽ trả về trình lặp trỏ đến vị trí phần tử đầu tiên có khóa lớn hơn khóa chỉ định. Và nếu không tìm thấy, hàm sẽ trả về trình lặp trỏ đến vị trí cuối cùng trong map.

Trong trường hợp chỉ có một phần tử trong multimap có khóa giống với khóa chỉ định thì hàm upper\_bound sẽ trả về con trỏ chỉ đến phần tử ngay sau nó. Còn nếu có nhiều phần tử có khóa giống với khóa chỉ định, vị trí của phần tử đầu tiên có khóa lớn hơn khóa chỉ định sẽ được hàm upper\_bound() trả về.

VD:

#include <iostream>

#include <map>

using namespace std;

int main() {

    multimap<char,int> mp;

    mp.insert(make\_pair('a', 1));

    mp.insert(make\_pair('b', 2));

    mp.insert(make\_pair('e', 5));

    mp.insert(make\_pair('c', 3));

    mp.insert(make\_pair('d', 1));

    mp.insert(make\_pair('e', 2));

    mp.insert(make\_pair('f', 3));

    mp.insert(make\_pair('e', 8));

    mp.insert(make\_pair('b', 9));

    /\*Duyệt map mp\*/

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

    //a:1 b:2 b:9 c:3 d:1 e:5 e:2 e:8 f:3

    /\*Tìm vị trí phần tử đầu tiên có khóa lớn hơn hoặc bằng 'b' trong map\*/

    auto itr1 = mp.lower\_bound('b'); // itr1 trỏ đến b:2

    //Tìm vị trí phần tử đầu tiên có khóa nhỏ hơn 'e' trong map

    auto itr2 = mp.upper\_bound('e'); // itr2 trỏ đến f:3

    //In các phần tử trong phạm vi [itr1, itr2)

    for (auto it=itr1; it!=itr2; ++it)

        cout << (\*it).first << ":" << (\*it).second << ' ';

    return 0;

}

a:1 b:2 b:9 c:3 d:1 e:5 e:2 e:8 f:3

b:2 b:9 c:3 d:1 e:5 e:2 e:8

### Đếm số lần xuất hiện của phần tử trong multimap C++ bằng hàm count

Cú pháp:

mp.count(key);

Trong đó key là khóa của phần tử cần đếm số lần xuất hiện trong multimap mp.

Hàm count() sẽ trả về số phần tử có khóa giống khóa chỉ định được tìm thấy trong multimap.

VD:

#include <iostream>

#include <map>

using namespace std;

int main() {

    multimap<char,int> mp;

    mp.insert(make\_pair('a', 100));

    mp.insert(make\_pair('b', 200));

    mp.insert(make\_pair('e', 500));

    mp.insert(make\_pair('c', 300));

    mp.insert(make\_pair('d', 100));

    mp.insert(make\_pair('e', 200));

    mp.insert(make\_pair('f', 300));

    //Đếm số lần xuất hiện của phần tử tồn tại trong multimap

    cout << mp.count('e') <<endl;

    //Đếm số lần xuất hiện của phần tử không tồn tại trong multimap

    cout << mp.count('x') <<endl;

    return 0;

}

2

0

## Multimap trống

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

Cú pháp:

mp.empty();

Trong đó mp là multimap cần kiểm tra.

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

VD:

#include <iostream>

#include <map>

using namespace std;

int main ()

{

    multimap<int, char> mp;

    if( mp.empty() )

        cout << "empty.\n";

    else

        cout << "not empty.\n";

    multimap<string,int> mp2 = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

    if( mp2.empty() )

        cout << "empty.\n";

    else

        cout << "not empty.\n";

    return 0;

}

empty.

not empty.

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

Cú pháp:

mp.clear();

Trong đó mp là multimap cần làm trống.

Khác với vector thì hàm multimap clear ngoài việc làm trống multimap chỉ định (xóa đi tất cả phần tử) thì còn giải phóng bộ nhớ sử dụng cho việc lưu trữ dữ liệu đã dùng.

VD:

#include <iostream>

#include <map>

using namespace std;

int main ()

{

    multimap<string,int> mymap = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

    cout << mymap.size() << "\n";       //3

    mymap.clear();

    cout << mymap.size() << "\n";       //0

}

# Unordered\_map

## Tổng quan

### Unordered\_map trong c++ là gì

Unordered\_map trong C++ là một tập hợp các phần tử không có thứ tự, mà mỗi phần tử trong đó được hình thành bởi sự kết hợp của một cặp khóa và giá trị (key & value), với các khóa không được trùng nhau.

Trong unordered\_map, các khóa (key) được sử dụng để xác định giá trị (value) tương ứng được liên kết với nó. Mỗi khóa trong unordered\_map là duy nhất và không được phép trùng lặp. Các value trong unordered\_map thì có thể trùng lặp, chúng có thể thay đổi giá trị, cũng như là được chèn hoặc xóa khỏi unordered\_map.

Khác với map trong C++ với các phần tử được sắp xếp và lưu trữ theo thứ tự của các khóa, thì các phần tử trong unordered\_map lại được lưu trữ dựa trên giá trị hash của khóa chứ không phải dựa trên thứ tự của các khóa.

Ở đây, giá trị hash của khóa là giá trị thu về sau khi tính toán khóa bằng một hàm hash. Còn hàm hash là giải thuật nhằm sinh ra các giá trị hash tương ứng với mỗi khối dữ liệu (có thể là một chuỗi ký tự, một đối tượng trong lập trình hướng đối tượng, v.v…) để phân biệt các khối dữ liệu với nhau.

Điều đó có nghĩa là giá trị chỉ định làm key trong unordered\_map phải thuộc kiểu giá trị có thể được hash bằng hàm hash.

Về mặt tốc độ xử lý thì unordered\_map có khả năng truy cập vào các phần tử riêng lẻ nhanh hơn so với map, tuy nhiên thì việc lặp trên một phạm vi trong đó thì lại chậm hơn map.

Tuy nhiên thì trong unordered\_map các phần tử không được sắp xếp sẵn, nên để xử lý một số lượng phần tử không quá lớn thì chúng ta chỉ cần dùng tới map là đủ rồi.

### Cấu trúc dữ liệu unordered\_map trong c++

Khác với map hay multimap vốn có cấu trúc dữ liệu unordered\_map trong C++ thuộc dạng Red–black tree (cây đỏ đen), thì cấu trúc dữ liệu của unordered\_map trong c++ lại hoàn toàn khác.

Nó dựa vào hash table- một bảng hash chứa các giá trị đã được hash để so sánh các key của phần tử, và quyết định thứ tự lưu giữ các phần tử này bên trong nó.

Chúng ta cần đặc biệt lưu ý 3 điểm sau đây về Cấu trúc dữ liệu unordered\_map trong c++:

* Khác với vector thì unordered\_map không dựa vào index mà dựa vào key để truy cập phần tử.
* Các phần tử trong unordered\_map được lưu vào dựa vào giá trị sau khi đã hash của các key, chứ không phải là được sắp xếp theo giá trị của key.
* Giống với map thì key trong unordered\_map là duy nhất và một key chỉ có thể dùng để xác định một giá trị duy nhất mà thôi.

Và cũng nhờ việc phần tử trong unordered\_map được lưu giữ theo giá trị hash của key mà chúng ta có thể truy cập vào phần tử riêng lẻ trong nó với tốc độ tức thì O(1).

### std::unordered\_map trong C++

std::unordered\_map trong C++ là một thư viện chuẩn được sử dụng để xử lý unordered\_map trong C++.

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

#include <iostream>

#include <unordered\_map>

int main()

{

    std::unordered\_map<std::string, int> mp;

    std::unordered\_map<char, double> mp2;

}

## Khai báo unordered\_map trong C++

### Khai báo 1 unordered\_map trong C++

Cú pháp:

std::unordered\_map<k\_type, v\_type> mp;

Trong đó mp là tên biến unordered\_map và k\_type, v\_type lần lượt là kiểu dữ liệu của key và value.

Tuy nhiên đối với kiểu của key, chúng ta chỉ có thể chỉ định các kiểu dữ liệu có khả năng được hash bởi hash function mà thôi. Ví dụ như các kiểu dữ liệu nguyên thủy thuộc kiểu số, kiểu chữ, hoặc là con trỏ chẳng hạn.

Nhưng nếu chúng ta sử dụng các kiểu dữ liệu không phải là kiểu dữ liệu nguyên thủy, ví dụ như cấu trúc hoặc class tự tạo chẳng hạn, thì bắt buộc phải thêm vào đối số thứ ba, là một hàm hash sử dụng để hash các dữ liệu này.

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

int main()

{

    unordered\_map<double, string> user;

    unordered\_map<int, char> info;

}

Còn khi khai báo 1 unordered\_map không thuộc kiểu dữ liệu nguyên thủy, ví dụ như struct chẳng hạn thì chúng ta phải thêm vào đối số thứ 3 là hash function như ví dụ sau:

class HashVI {  // Hàm hash object

public:

    size\_t operator()(const vector<int> &x) const {

        const int C = 997;

        size\_t t = 0;

        for (int i = 0; i != x.size(); ++i) {

            t = t \* C + x[i];

        }

        return t;

    }

};

std::unordered\_map<vector<int>, int, HashVI> mp2;

### Khai báo đồng thời nhiều unordered\_map trong C++

Cú pháp:

using namespace std;

unordered\_map<k\_type, v\_type> name1, name2, name3, ... ;

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

int main()

{

    unordered\_map<string, int> name, job, sex;

    unordered\_map<int, char> age;

}

## Gán giá trị cho unordered\_map trong C++

Cú pháp:

mp[key] = value;

Trong đó

* mp là tên biến unordered\_map
* key và value là khóa và giá trị của phần tử cần gán vào unordered\_map

VD:

std::unordered\_map<std::string, int> mp;

mp["Kiyoshi"] = 1; // {"Kiyoshi", 1}

mp["Honda"] = 2;   // {"Honda", 2}

mp["Suzuki"] = 3;  // {"Suzuki", 3}

Lại nữa, giá trị của phần tử có thể được ghi đè và thay thế bởi một giá trị mới nhiều lần như sau:

mp["Kiyoshi"] = 4; // {"Kiyoshi", 4}

mp["Kiyoshi"] = 8; // {"Kiyoshi", 8}

mp["Kiyoshi"] = 88;// {"Kiyoshi", 88}

Trong trường hợp thay đổi giá trị nhiều lần, thì giá trị trong lượt thay đổi giá trị cuối cùng sẽ được sử dụng làm giá trị phần tử.

## Khởi tạo unordered\_map trong C++

Cú pháp:

std::unordered\_map<k\_type, v\_type> mp = { {k1, v1}, {k2, v2}, {k3, v3}, ...};

Trong đó

* mp là tên biến unordered\_map
* k\_type, v\_type là kiểu dữ liệu của key và value
* k,v là các cặp key và value

Lưu ý do mỗi khóa trong unordered\_map là duy nhất, nên nếu chúng ta chỉ định các phần tử có cùng khóa thì dù giá trị của chúng có giống hay khác nhau thì chỉ có duy nhất phần tử viết đầu tiên sẽ được lưu vào trong unordered\_map mà thôi.

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

int main ()

{

    unordered\_map<string,int> mp = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 }

            };

    for (auto& x: mp) {

        std::cout << x.first << ": " << x.second << '\n';

    }

    return 0;

}

gamma: 30

beta: 20

alpha: 10

Trong trường hợp chỉ định các phần tử có khóa giống nhau, bất kể giá trị của chúng có giống hay khác nhau thì chỉ có duy nhất phần tử viết đầu tiên được lưu vào unordered\_map như sau:

#include <iostream>

#include <unordered\_map>

using namespace std;

int main ()

{

    unordered\_map<string,int> mp = {

                { "alpha", 20 },

                { "beta", 20 },

                { "alpha", 10 },

                { "gamma", 30 },

                { "alpha", 10 },

                };

    for (auto x: mp) {

        cout << x.first << ": " << x.second << endl;

    }

    return 0;

}

// beta: 20

// gamma: 30

// alpha: 20

## Truy cập phần tử trong unordered\_map C++

### Truy cập phần tử trong unordered\_map C++ bằng toán tử []

Cú pháp:

mp[key]

Trong đó

* mp là tên biến unordered\_map
* key là khóa của phần tử cần truy cập trong unordered\_map

Nếu như key tồn tại trong unordered\_map, giá trị tương ứng của key sẽ được trả về. Tuy nhiên nếu không tồn tại, giá trị 0 sẽ được trả về

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

int main ()

{

    /\*Khai báo và gán giá trị\*/

    unordered\_map<string, int> mp;

    mp["Kiyoshi"] = 1; // {"Kiyoshi", 1}

    mp["Honda"] = 2;   // {"Honda", 2}

    mp["Suzuki"] = 3;  // {"Suzuki", 3}

    /\*Truy cập vào phần tử vừa gán bằng key\*/

    cout << mp["Kiyoshi"] ;   //1

    /\*Truy cập vào phần tử không tồn tại trong unordered\_map\*/

    cout << mp["Honda"];       //0

    return 0;

}

Ngoài việc xuất giá trị, chúng ta cũng có thể thay đổi giá trị của một phần tử trong unordered\_map với cách này. Ví dụ:

/\*Khai báo và gán giá trị\*/

std::unordered\_map<std::string, int> mp;

mp["Kiyoshi"] = 1;

/\*Truy cập và thay đổi giá trị\*/

mp["Kiyoshi"] = 2;

### Truy cập phần tử trong unordered\_map C++ bằng hàm at()

Cú pháp:

mp.at(key);

Trong đó mp là tên unordered\_map và key là khóa của phần tử cần truy cập.

Nếu như key tồn tại trong unordered\_map, giá trị tương ứng của key sẽ được trả về. Tuy nhiên nếu không tồn tại, hàm at() sẽ trả về lỗi out\_of\_range.

VD:

/\*Khai báo và gán giá trị\*/

std::unordered\_map<std::string, int> mp;

mp["Kiyoshi"] = 1;

/\*Truy cập vào phần tử vừa gán bằng key\*/

cout << mp.at("Kiyoshi"); //1

/\*Truy cập vào phần tử không tồn tại trong unordered\_map\*/

cout << mp.at("Honda"); // throwing an instance of 'std::out\_of\_range

## Duyệt phần tử

### Duyệt unordered\_map trong C++ bằng vòng lặp dựa trên phạm vi

Cú pháp:

for ( auto& x : mp) {

cout << x.first << “:” << x.second << endl;

}

Trong đó:

* mp là tên unordered\_map.
* auto là kiểu suy luận giúp tự xác định kiểu dữ liệu của giá trị lấy từ unordered\_map.
* x là tên một biến dùng để gán từng phần tử được lấy từ unordered\_map.
* x.first và x.second lần lượt được sử dụng để lấy key và value của phần tử

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

int main()

{

    unordered\_map<string,int> mp = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

    for (auto x: mp) {

        cout << x.first << ":" << x.second << endl;

    }

}

gamma:30

beta:20

alpha:10

### Duyệt unordered\_map trong C++ bằng iterator

Cú pháp:

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

cout << itr->first << ":"<< itr->second << "\n";

}

Trong đó:

* mp là tên unordered\_map
* itr là tên iterator dùng để trỏ đến phần tử
* itr->first và itr->second lần lượt được sử dụng để lấy key và value của phần tử

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

int main()

{

    unordered\_map<string,int> mp = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

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

        cout  << itr->first << ":"<< itr->second << "\n";

    }

}

gamma:30

beta:20

alpha:10

## Lấy kích thước

### Lấy kích thước unordered\_map trong C++ bằng hàm size

Cú pháp:

mp.size();

Trong đó mp là unordered\_map cần lấy kích thước (số phần tử) chứa trong nó.

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

int main ()

{

    unordered\_map<string,int> mp = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

    cout<< mp.size() <<endl;

    return 0;

}

//3

Lưu ý, số phần tử hay kích thước của unordered\_map ở đây được tính sau khi các phần tử được kiểm tra key có trùng lặp và sắp xếp trong unordered\_map, chứ không phải là số phần tử mà chúng ta đã dùng khi khai báo unordered\_map.

Ví dụ, nếu khi khai báo unordered\_map mà tồn tại các phần tử trùng lặp key thì số phần tử được đếm bởi hàm size() sẽ khác với số phần tử chỉ định trong khai báo như sau:

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

int main ()

{

    unordered\_map<string,int> mp = {

                { "alpha", 20 },

                { "beta", 20 },

                { "alpha", 10 },

                { "gamma", 30 },

                { "alpha", 10 },

                };

    for (auto x: mp) {

        cout << x.first << ":" << x.second << endl;

    }

    cout<< mp.size() <<endl;

    return 0;

}

beta:20

gamma:30

alpha:20

3

## Thêm, chèn phần tử

### Thêm 1 phần tử vào unordered\_map trong C++ bằng toán tử []

Cú pháp:

mp[key] = value;

Trong đó

* mp là tên biến unordered\_map
* key và value là khóa và giá trị của phần tử cần thêm vào unordered\_map

VD:

unordered\_map<char,int> foo,bar;

foo['x']=100; //()

foo['y']=200;

Lưu ý do mỗi key trong unordered\_map đều là duy nhất nên nếu chúng ta thêm một phần tử mới vào unordered\_map nhưng lại có key trùng với một phần tử đã tồn tại trước đó, thì giá trị của phần tử mới này sẽ được dùng để ghi đè lên phần tử trước đó.

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

//Tạo hàm xuất unordered\_map

void dump(unordered\_map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    unordered\_map<char,int> mymap;

    mymap['x']=100;

    mymap['y']=200;

    mymap['z']=300;

    dump(mymap);        // z:300 y:200 x:100

    mymap['z']=888;

    dump(mymap);        // z:888 y:200 x:100

}

### Chèn 1 phần tử vào unordered\_map trong C++ bằng hàm insert

Cú pháp:

mp.insert(std::pair<k\_type,x\_type>(k,v));

Trong đó

* mp là unordered\_map ban đầu
* pair<k\_type,x\_type>(k,v) sử dụng để chỉ định key và value của phần tử cần thêm, trong đó k\_type,x\_type là kiểu và k,v là key và value.

Trong trường hợp không rõ kiểu, hoặc muốn rút bỏ chỉ định kiểu của key và value, chúng ta có thể dùng hàm make\_pair() để thay thế cho pair() trong hàm insert, với cú pháp sau đây:

mp.insert(std::make\_pair(k,v));

Hàm unordered\_map insert sẽ trả về một cặp kết quả pair<iterator, bool> với iterator là trình lặp trỏ đến unordered\_map kết quả, và bool là việc có thực hiện việc chèn hay không, dưới dạng 0 hoặc 1.

Bởi vì các phần tử trong một unordered\_map là duy nhất, nên thao tác chèn sẽ kiểm tra xem mỗi phần tử được chèn đã tồn tại trong unordered\_map hay chưa. Nếu chưa tồn tại thì phần tử đó sẽ được chèn và ngược lại nếu đã tồn tại thì không được chèn.

Trong trường hợp tất cả các phần tử chèn vào vốn đã tồn tại từ trước trong unordered\_map, thì bản thân unordered\_map ban đầu sẽ được trả về.

Lại nữa, phần tử trong unordered\_map được sắp xếp theo thứ tự cụ thể trước khi được lưu, nên các phần tử cần chèn thêm vào unordered\_map sẽ được tự động quyết định vị trí dựa trên bộ sắp xếp, và chúng ta sẽ không quyết định được vị trí cần chèn của phần tử.

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

//Tạo hàm xuất unordered\_map

void dump(unordered\_map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    /\*Khai báo và gán giá trị cho foo và bar\*/

    unordered\_map<char,int> mp;

    mp['a']=100;

    mp['b']=200;

    // duyệt unordered\_map ban đầu

    dump(mp);

    //chèn phần tử với key chưa tồn tại trong unordered\_map

    mp.insert(pair<char,int>('c',300));

    dump(mp);

    //chèn phần tử với key đã tồn tại trong unordered\_map

    mp.insert(pair<char,int>('a',3));

    dump(mp);

    //chèn phần tử bằng insert và make\_pair

    mp.insert(make\_pair('x',50));

    dump(mp);

    return 0;

}

b:200 a:100

c:300 b:200 a:100

c:300 b:200 a:100 // Do key đã tồn tại nên sẽ không được chèn

x:50 c:300 b:200 a:100

Chúng ta cũng có thể kiểm tra việc chèn phần tử đã thực hiện hay chưa bằng phương thức first() hoặc second() từ kết quả trả về của hàm như sau:

#include <iostream>

#include <unordered\_map>

using namespace std;

int main ()

{

    /\*Khai báo và gán giá trị cho foo và bar\*/

    unordered\_map<char,int> mp;

    mp['a']=100;

    mp['b']=200;

    //chèn phần tử với key chưa tồn tại trong unordered\_map

    auto r = mp.insert(pair<char,int>('z',300));

    cout << r.second << "\n";   // Trả về 1 do được chèn

    //chèn phần tử với key tồn tại trong unordered\_map,

    r = mp.insert(pair<char,int>('a',3));

    cout << r.second << "\n";  // Trả về 0 do không được chèn

    return 0;

}

### Thêm chèn 1 phần tử vào unordered\_map trong C++ bằng hàm emplace

Cú pháp:

mp.emplace(k,v);

Trong đó

* mp là unordered\_map ban đầu
* k,v là key và value.

Hàm unordered\_map emplace sẽ thực hiện việc chèn phần tử nếu key của nó chưa tồn tại trong unordered\_map. Ngược lại nếu key đó đã tồn tại, việc chèn thất bại và false sẽ được trả về.

Hàm unordered\_map emplace sẽ trả về một trình lặp trỏ tới vị trí phần tử vừa được chèn vào, nếu việc chèn thành công.

Lại nữa, phần tử trong unordered\_map được lưu vào dựa theo giá trị hash của các khóa, nên các phần tử cần chèn thêm vào unordered\_map sẽ được tự động quyết định vị trí dựa trên giá trị hash ở trên bảng hash, và chúng ta sẽ không quyết định được vị trí cần chèn của phần tử.

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

//Tạo hàm xuất unordered\_map

void dump(unordered\_map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    std::unordered\_map<char,int> mymap;

    mymap.emplace('x',100);

    mymap.emplace('y',200);

    mymap.emplace('z',100);

    dump(mymap);

    return 0;

}

z:100 y:200 x:100

### Chèn nhiều phần tử vào unordered\_map trong C++ bằng hàm insert

Cú pháp:

mp.insert(iterator\_first, iterator\_last);

Trong đó

* mp là unordered\_map ban đầu
* iterator\_first và iterator\_last là các trình lặp xác định phạm vi chứa các phần tử cần chèn ở trong một unordered\_map khác vào unordered\_map ban đầu.

Lưu ý ở đây, các phần tử cần chèn được lấy ra từ một phạm vi trong một unordered\_map khác, và phạm vi này được chỉ định thông qua 2 trình lặp như trên.

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

//Tạo hàm xuất unordered\_map

void dump(unordered\_map<int, const char\*>& mp)

{

    for (auto x: mp) {

        cout << x.first << ": " << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    /\* Khởi tạo unordered\_map mp1\*/

    unordered\_map<int, const char\*> mp1 =

    {

            {1, "a"},

            {2, "b"},

            {3, "c"},

            {4, "d"},

            {5, "e"},

    };

    // duyệt unordered\_map mp1

    dump(mp1);

    /\*Khai báo unordered\_map mp2\*/

    unordered\_map<int, const char\*> mp2;

    /\*Chèn các phần tử từ đầu tới phần tử có key bằng 2

      từ mp1 vào mp2\*/

    mp2.insert(mp1.begin(),mp1.find(2));

    // duyệt unordered\_map mp2

    dump(mp2);

    return 0;

}

5: e 4: d 3: c 2: b 1: a

3: c 4: d 5: e

## Xoá phần tử

### Xóa 1 phần tử trong unordered\_map bằng hàm erase c++

Cú pháp:

mp.erase(itr);

OR

mp.erase(key);

Trong đó mp là unordered\_map ban đầu, itr và key là trình lặp hoặc là khóa của phần tử cần xóa.

Nếu sử dụng erase(itr) thì hàm sẽ xóa đi phần tử tại vị trí mà trình lặp xác định bởi itr chỉ đến.

Và nếu sử dụng erase(key) thì hàm sẽ tìm phần tử có khóa bằng với key trong unordered\_map và tiến hành xóa nó đi.

Khác với các container khác thì để chuyển trình lặp chỉ đến vị trí index thứ n trong unordered\_map, chúng ta không thể đơn giản cộng vào n vào trình lặp, mà cần phải di chuyển lần lượt qua từng vị trí, bằng toán tử ++ với đủ số vòng lặp.

Lưu ý do phần tử trong unordered\_map không được sắp xếp. Do đó chúng ta sẽ không quyết định được vị trí của các phần tử sau khi xóa đi phần tử từ unordered\_map.

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

//Tạo hàm xuất unordered\_map

void dump(unordered\_map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    unordered\_map<char,int> mp;

    mp['a']=100;

    mp['b']=200;

    mp['c']=300;

    mp['d']=400;

    mp['e']=400;

    dump(mp);

    /\*Tìm trình lặp trỏ đến phần tử có khóa bằng 'c' \*/

    auto itr = mp.find('c'); // Chỉ đến c:300

    //xóa phần tử tại vị trí itr chỉ đến

    mp.erase(itr);

    dump(mp);

    //xóa phần tử có khóa bằng 'd' trong multimap

    mp.erase('d');

    dump(mp);

    return 0;

}

e:400 d:400 c:300 b:200 a:100

e:400 d:400 b:200 a:100

e:400 b:200 a:100

Chúng ta cũng có thể kiểm tra số phần tử đã được xóa đi từ kết quả trả về của hàm như sau:

unordered\_map<char,int> mp;

mp['x']=100;

mp['y']=200;

mp['z']=300;

mp['t']=400;

auto r = mp.erase('t'); /\*r là số phần tử có key bằng 't' bị xóa\*/

std::cout << r;

//1

### Xóa các phần tử trong một phạm vi chỉ định bằng unordered\_map erase c++

Cú pháp:

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

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

Lưu ý là phạm vi xóa là [iterator\_first,iterator\_last) sẽ được tính từ iterator\_first đến trước iterator\_last, nghĩa là phần tử ở vị trí iterator\_first sẽ được xóa nhưng phần tử ở vị trí iterator\_last sẽ không bị xóa đi.

Ứng dụng điều này, chúng ta có thể chỉ định trình lặp và xóa đi các phần tử trong phạm vi chỉ định từ unordered\_map như sau:

#include <iostream>

#include <unordered\_map>

using namespace std;

//Tạo hàm xuất unordered\_map

void dump(unordered\_map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main ()

{

    unordered\_map<char,int> mp;

    mp['a']=100;

    mp['b']=200;

    mp['c']=300;

    mp['d']=400;

    mp['e']=500;

    mp['f']=600;

    dump(mp);

    /\*Khai báo phạm vi cần xóa\*/

    int start = 2, end =5;

    /\*Tạo các trình lặp trỏ tới start và end với giá trị ban đầu\*/

    auto itr\_start = mp.begin();

    auto itr\_end = mp.begin();

    /\*Thay đổi trình lặp tương ứng tới các vị trí start và end\*/

    for (int i=1; i <= start; i++ )

        ++itr\_start;

    for (int i=1; i <= end; i++ )

        ++itr\_end;

    //Sau đó dùng hàm erase để xóa phạm vi là xong

    mp.erase(itr\_start, itr\_end);

    dump(mp);

    return 0;

}

f:600 e:500 d:400 c:300 b:200 a:100

f:600 e:500 a:100

## Sao chép phần tử

### Sao chép unordered\_map trong C++

Cú pháp:

std::unordered\_map<k\_type, v\_type> mp\_des( mp\_src );

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

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

//Tạo hàm xuất unordered\_map

void dump(unordered\_map<string,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main()

{

    unordered\_map<string,int> mp\_src = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

    cout << "Orginary unordered\_map" << endl;

    dump (mp\_src);

    /\*Sao chép unordered\_map\*/

   unordered\_map<string,int> mp\_des(mp\_src);

    cout << "Copy unordered\_map" << endl;

    dump (mp\_des);

    return 0;

}

Orginary unordered\_map

gamma:30 beta:20 alpha:10

Copy unordered\_map

gamma:30 beta:20 alpha:10

### Hoán đổi 2 unordered\_map trong C++

Cú pháp:

st1.swap(st2);

Trong đó st1 và st2 là 2 unordered\_map cần hoán đổi nội dung cho nhau.

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

//Tạo hàm xuất unordered\_map

void dump(unordered\_map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main()

{

    /\*Khai báo và gán giá trị cho foo và bar\*/

    unordered\_map<char,int> foo,bar;

    foo['x']=100;

    foo['y']=200;

    bar['a']=11;

    bar['b']=22;

    bar['c']=33;

    cout << "Before swap" << endl;

    dump(foo);

    dump(bar);

    /\*Hoán đổi for cho bar\*/

    foo.swap(bar);

    cout << "After swap" << endl;

    dump(foo);

    dump(bar);

}

Before swap

y:200 x:100

c:33 b:22 a:11

After swap

c:33 b:22 a:11

y:200 x:100

Ngoài cách dùng hàm unordered\_map swap, chúng ta cũng có thể dùng function template là std::swap để tiến hành hoán đổi 2 unordered\_map với nhau, cũng như là để hoán đổi các đối tượng khác như unordered\_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

#include <iostream>

#include <utility>

#include <unordered\_map>

using namespace std;

//Tạo hàm xuất unordered\_map

void dump(unordered\_map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main()

{

    /\*Khai báo và gán giá trị cho foo và bar\*/

    unordered\_map<char,int> foo,bar;

    foo['x']=100;

    foo['y']=200;

    bar['a']=11;

    bar['b']=22;

    bar['c']=33;

    cout << "Before swap" << endl;

    dump(foo);

    dump(bar);

    /\*Hoán đổi for cho bar\*/

    swap(foo,bar);

    cout << "After swap" << endl;

    dump(foo);

    dump(bar);

}

Before swap

y:200 x:100

c:33 b:22 a:11

After swap

c:33 b:22 a:11

y:200 x:100

## Tìm phần tử

### Tìm phần tử trong unordered\_map C++ bằng hàm find

Cú pháp:

mp.find(key);

Trong đó key là khóa của phần tử cần tìm trong unordered\_map mp.

Hàm find() sẽ trả về trình lặp trỏ đến vị trí phần tử, nếu nó tồn tại trong unordered\_map. Và nếu phần tử đó không tồn tại, hàm sẽ trả về trình lặp trỏ đến vị trí cuối cùng trong unordered\_map.

Bằng cách ứng dụng hàm find(), chúng ta có thể tìm ra vị trí của phần tử đó trong unordered\_map, rồi kết hợp với hàm clear() để xóa nó đi như ví dụ cụ thể sau đây:

#include <iostream>

#include <unordered\_map>

using namespace std;

//Tạo hàm xuất unordered\_map

void dump(unordered\_map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main() {

    unordered\_map<char,int> mp;

    mp.insert(make\_pair('a', 1));

    mp.insert(make\_pair('b', 2));

    mp.insert(make\_pair('c', 3));

    mp.insert(make\_pair('d', 1));

    mp.insert(make\_pair('e', 2));

    dump(mp);

    //Tìm phần tử có khoá bằng 'c' trong multiset

    auto itr = mp.find('c');

    //Xóa phần tử vừa tìm thấy

    mp.erase (itr);

    dump(mp);

    return 0;

}

e:2 d:1 c:3 b:2 a:1

e:2 d:1 b:2 a:1

### Tìm phần tử trong unordered\_map C++ bằng hàm equal\_range

Cú pháp:

mp.equal\_range(key);

Trong đó key là khóa của phần tử cần tìm trong unordered\_map mp.

Hàm equal\_range() sẽ trả về một cặp giá trị, với giá trị đầu tiên trỏ đến đầu phạm vi, và giá trị thứ hai trỏ đến cuối phạm vi chứa tất cả các phần tử có khóa giống khóa chỉ định.

Tuy nhiên do trong unordered\_map thì các khóa là duy nhất, do đó nếu khóa tồn tại trong unordered\_map thì hàm equal\_range sẽ trả về một khoảng chứa giá trị duy nhất đó mà thôi.

Bằng cách ứng dụng hàm equal\_range(), chúng ta có thể tìm ra phạm vi chứa phần tử có khóa chỉ định trong unordered\_map như sau

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

//Tạo hàm xuất unordered\_map

void dump(unordered\_map<char,int>& mp)

{

    for (auto x: mp) {

        cout << x.first << ":" << x.second << " ";

    }

    cout << endl;

}

int main() {

    unordered\_map<char,int> mp;

    mp.insert(make\_pair('a', 1));

    mp.insert(make\_pair('b', 2));

    mp.insert(make\_pair('c', 3));

    mp.insert(make\_pair('d', 1));

    mp.insert(make\_pair('e', 2));

    dump(mp);

    //Tìm phần tử có khóa bằng 'c' trong unordered\_map

    auto ret = mp.equal\_range('c');

    cout << "lower bound points to: ";

    cout << ret.first->first << ":" << ret.first->second << '\n';

    cout << "upper bound points to: ";

    cout << ret.second->first << ":" << ret.second->second << '\n';

    return 0;

}

e:2 d:1 c:3 b:2 a:1

lower bound points to: c:3

upper bound points to: b:2

### Đếm số lần xuất hiện của phần tử trong unordered\_map C++ bằng hàm count

Cú pháp:

mp.count(key);

Trong đó key là khóa của phần tử cần đếm số lần xuất hiện trong unordered\_map mp.

Do trong unordered\_map các khóa là duy nhất, nên một phần tử nếu tồn tại cũng chỉ có xuất hiện 1 lần duy nhất trong unordered\_map mà thôi.

Do vậy, kết quả trả về của hàm count() cũng chỉ là 0 tương ứng với phần tử không tồn tại, hoặc 1 tương ứng với phần tử tồn tại trong unordered\_map đó.

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

int main() {

    unordered\_map<char,int> mp;

    mp['a']=100;

    mp['b']=200;

    mp['c']=300;

    //Đếm số lần xuất hiện của phần tử tồn tại trong unordered\_map

    cout << mp.count('b') <<endl;

    //Đếm số lần xuất hiện của phần tử không tồn tại trong unordered\_map

    cout << mp.count('f') <<endl;

    return 0;

}

1

0

## Unordered\_map trống

### Kiểm tra unordered\_map trống trong C++ bằng hàm empty

Cú pháp:

mp.empty();

Trong đó mp là unordered\_map cần kiểm tra.

Hàm empty sẽ trả về true nếu unordered\_map đã cho là unordered\_map trống, cũng trả về false, nếu unordered\_map đã cho có chứa phần tử.

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

int main ()

{

    unordered\_map<int, char> mp;

    if( mp.empty() )

        cout << "empty.\n";

    else

        cout << "not empty.\n";

    unordered\_map<string,int> mp2 = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

    if( mp2.empty() )

        cout << "empty.\n";

    else

        cout << "not empty.\n";

    return 0;

}

empty.

not empty.

### Làm trống 1 unordered\_map trong C++ bằng hàm clear

Cú pháp:

mp.clear();

Trong đó mp là unordered\_map cần làm trống.

Khác với vector thì hàm unordered\_map clear ngoài việc làm trống unordered\_map chỉ định (xóa đi tất cả phần tử) thì còn giải phóng bộ nhớ sử dụng cho việc lưu trữ dữ liệu đã dùng.

VD:

#include <iostream>

#include <unordered\_map>

using namespace std;

int main ()

{

    unordered\_map<char,int> mymap;

    mymap['x']=100;

    mymap['y']=200;

    mymap['z']=300;

    cout << mymap.size() << "\n";       //3

    mymap.clear();

    cout << mymap.size() << "\n";       //0

}

## Bucket

### Bucket trong C++ là gì

Bucket trong C++ là một không gian bộ nhớ trong bảng hash của container (unordered\_map), trong đó các phần tử sẽ được chỉ định dựa trên giá trị hash của khóa của chúng. Phạm vi giới hạn hợp lệ của các bucket là từ 0 đến bucket\_count - 1.

### Lấy vị trí bucket chứa phần tử có khóa chỉ định bằng hàm unordered\_map bucket

Cú pháp:

mp.bucket(k);

Trong đó mp là unordered\_map và k là khóa cần tìm số bucket chứa phần tử được xác định bởi nó.

Hàm bucket sẽ trả về vị trí (số thứ tự) của bucket chứa phần tử có khóa k được chỉ định.

VD:

#include <iostream>

#include <string>

#include <unordered\_map>

using namespace std;

int main ()

{

    unordered\_map<string,int> mymap = {

                { "alpha", 10 },

                { "beta", 20 },

                { "gamma", 30 } };

  for (auto& x: mymap) {

    std::cout << "Phần tử [" << x.first << ":" << x.second << "]";

    std::cout << " nằm tại bucket số #" << mymap.bucket (x.first) << std::endl;

  }

  return 0;

}

Phần tử [gamma:30] nằm tại bucket số #2

Phần tử [beta:20] nằm tại bucket số #0

Phần tử [alpha:10] nằm tại bucket số #0

### Lấy số bucket tạo ra bởi unordered\_map bằng hàm bucket\_count

Cú pháp:

mp.bucket\_count();

Trong đó mp là unordered\_map cần đếm số bucket đã được tạo ra từ nó.

Hàm bucket\_count sẽ trả về số bucket được tạo ra từ unordered\_map chỉ định.

VD:

#include <iostream>

#include <string>

#include <unordered\_map>

using namespace std;

int main ()

{

    unordered\_map<char,int> mp;

    mp.insert(make\_pair('a', 1));

    mp.insert(make\_pair('b', 2));

    mp.insert(make\_pair('c', 3));

    mp.insert(make\_pair('d', 1));

    mp.insert(make\_pair('e', 2));

    unsigned n = mp.bucket\_count();

    cout << "Có tất cả " << n << " buckets.\n";

    for (unsigned i=0; i<n; ++i) {

        cout << "bucket #" << i << " chứa: ";

        for (auto it = mp.begin(i); it!=mp.end(i); ++it)

            cout << "[" << it->first << ":" << it->second << "] ";

        cout << "\n";

    }

  return 0;

}

Có tất cả 13 buckets.

bucket #0 chứa:

bucket #1 chứa:

bucket #2 chứa:

bucket #3 chứa:

bucket #4 chứa:

bucket #5 chứa:

bucket #6 chứa: [a:1]

bucket #7 chứa: [b:2]

bucket #8 chứa: [c:3]

bucket #9 chứa: [d:1]

bucket #10 chứa: [e:2]

bucket #11 chứa:

bucket #12 chứa:

### Lấy số phần tử lưu trữ trong bucket bằng hàm bucket\_size

Cú pháp:

mp.bucket\_size(i);

Trong đó mp là unordered\_map và i là số thứ tự của bucket cần đếm số phần tử trong nó.

Hàm bucket\_size sẽ trả về số phần tử của unordered\_map được lưu trữ trong bucket có số thứ tự chỉ định.

VD:

#include <iostream>

#include <string>

#include <unordered\_map>

using namespace std;

int main ()

{

    unordered\_map<char,int> mp;

    mp.insert(make\_pair('a', 1));

    mp.insert(make\_pair('b', 2));

    mp.insert(make\_pair('c', 3));

    mp.insert(make\_pair('d', 1));

    mp.insert(make\_pair('e', 2));

    unsigned nbuckets = mp.bucket\_count();

    cout << "Có tất cả " << nbuckets << " buckets.\n";

    for (unsigned i=0; i<nbuckets; ++i) {

        cout << "bucket #" << i << " có " << mp.bucket\_size(i) << " phần tử.\n";

    }

  return 0;

}

Có tất cả 13 buckets.

bucket #0 có 0 phần tử.

bucket #1 có 0 phần tử.

bucket #2 có 0 phần tử.

bucket #3 có 0 phần tử.

bucket #4 có 0 phần tử.

bucket #5 có 0 phần tử.

bucket #6 có 1 phần tử.

bucket #7 có 1 phần tử.

bucket #8 có 1 phần tử.

bucket #9 có 1 phần tử.

bucket #10 có 1 phần tử.

bucket #11 có 0 phần tử.

bucket #12 có 0 phần tử.

# File

## Sử dụng hàm thừa kế từ ngôn ngữ C

### File trong C++ là gì

Trong thế giới máy tính, file là một loại “Tài liệu” nhằm lưu dữ dữ liệu và thông tin. Có rất nhiều kiểu file khác nhau nhằm lưu dữ các kiểu dữ liệu và thông tin khác nhau, ví dụ như file text, file Excel, file Json, file XML, hay là file CSV v.v…

Để có thể thao tác với các loại file này, người dùng thông thường cần các phần mềm chuyên dùng để mở, đọc ghi và lưu chúng. Tuy nhiên thì với các lập trình viên như chúng ta thì bằng cách sử dụng sử dụng ngôn ngữ C++, chúng ta cũng có thể dễ dàng thực hiện các thao tác tương tự với file như vậy.

Cũng giống như mảng, chuỗi hay biến thì file trong C++ là một loại dữ liệu trong chương trình, và một file khi nhập vào chương trình cũng sẽ được lưu giữ tại một vùng nào đó trên bộ nhớ máy tính. Tuy nhiên khác với các loại dữ liệu khác có thể gán vào một biến để xử lý, thì để thao tác với file, chúng ta cần tạo ra một thực thể của kiểu cấu trúc FILE để chứa thông tin của file cần thao tác, sau đó sử dụng tới một con trỏ để chỉ đến vị trí của thực thể FILE chứa thông tin file đó trên bộ nhớ, qua đó thực hiện các thao tác với file thông qua các hàm có sẵn, với các chức năng cụ thể như đóng, mở, ghi hay lưu file.

Do vậy muốn thao tác được với file, bạn cần nắm vững các kiến thức cơ bản về con trỏ và kiểu cấu trúc trong C++. Bạn có thể tham khảo các bài viết chi tiết về con trỏ trong chuyên đề Con trỏ trong C++, cũng như về kiểu cấu trúc tại chuyên đề Kiểu cấu trúc trong C++.

### Xử lý file trong C++

Tương tự như với ngôn ngữ C thì quy trình xử lý file trong C++ sẽ gồm các bước như sau:



### Tạo con trỏ file

Mỗi file trong chương trình C++ được xử lý dưới dạng một thực thể của kiểu cấu trúc FILE - một kiểu cấu trúc được quy định sẵn trong ngôn ngữ C++ phục vụ cho việc xử lý file.

Về căn bản thì cách sử dụng kiểu cấu trúc này cũng tương tự như các kiểu cấu trúc mà chúng ta đã học trong chuyên đề Kiểu cấu trúc trong C++.

Để thao tác với file, trước tiên chúng ta cần phải tạo một con trỏ chỉ đến thực thể của kiểu cấu trúc FILE chứa thông tin của file đó trên bộ nhớ với cú pháp sau đây:

FILE \*fp;

Sau đó, bằng cách truy cập vào địa chỉ này thông qua con trỏ, chúng ta mới có thể thực hiện các thao tác với file.

### Mở file

Thông thường khi mở file trong máy tính, chúng ta có thể click đúp chuột vào nó, hoặc là mở nó trong một chương trình đặc định. Tuy nhiên khi mở file trong chương trình C++, chúng ta cần phải sử dụng tới một số hàm chuyên dụng như fopen() hay fopen\_s().

Việc mở file và gán địa chỉ file vào con trỏ thường được tiến hành đồng thời. Ví dụ bạn có thể mở một file sample.txt bằng hàm fopen() và gán vào con trỏ file fp như sau:

fp = fopen("sample.txt", "r");

### Đọc dữ liệu từ file

Sau khi mở file, chúng ta đã có thể đọc các dữ liệu từ file vào trong chương trình để xử lý.

Có nhiều phương pháp để đọc dữ liệu từ một file. Ví dụ bạn có thể đọc từng dòng của file, hoặc đọc toàn bộ nội dung file. Nội dung file có thể được đọc dưới dạng từng ký tự, dạng chuỗi, hoặc dưới dạng mảng để có thể dễ dàng xử lý trong chương trình.

Để đọc file trong C++, chúng ta có thể sử dụng các hàm kế thừa từ C như là fgets() hay fgetc() chẳng hạn.

Ngoài ra với từng định dạng file mà chúng ta sẽ có các phương pháp đọc file khác nhau trong C++. Ví dụ như cùng sử dụng hàm fscanf() hoặc hàm sscanf() để đọc file, nhưng cú pháp đọc file txt sẽ khác cách đọc file CSV chẳng hạn.

Ví dụ cụ thể, sau khi mở file sample.txt ở trên, chúng ta có thể đọc từng dòng file đó bằng hàm fgets như sau:

#include <iostream>

using namespace std;

int main()

{

    FILE \* fp = NULL;

    char arr[128];

    //Mở file bằn hàm fopen

    fp= fopen("sample.txt", "r");

    fgets(arr, 128, fp);

    cout <<  arr;

    return 0;

}

### Ghi file

Chúng ta có thể tạo một file mới rồi ghi nội dung vào file đó, hoặc là mở một file sẵn và ghi thêm nội dung vào đó.

Chúng ta có thể sử dụng các hàm có sẵn như fputc,fputs và fprintf để làm được việc này.

Ví dụ cụ thể, sau khi mở file sample.txt ở trên, chúng ta có thể ghi dòng chữ “Hello Vietnam!” vào file đó bằng hàm fprintf như sau:

FILE \* fp = NULL;

fp = fopen("sample.txt", "r");

fprintf(fp, "%s", "Hello Vietnam!");

### Đóng file

Sau khi đã xử lý xong file trong chương trình, chúng ta cần phải đóng file đó lại. Việc đóng file sẽ giúp kết thúc phiên làm việc với file, và giải phóng bộ nhớ.

Nếu không đóng file thì file đó vẫn tồn tại trên bộ nhớ, dẫn đến xảy ra các sự cố về bộ nhớ trong chương trình.

Để đóng một file trong C++, chúng ta cần dùng đến hàm fclose() với cú pháp sau đây:

fclose(fp);

Trong đó fp là con trỏ dùng để mở file.

VD:

FILE \* fp = NULL;

fp = fopen("sample.txt", "r");

fprintf(fp, "%s", "Hello Vietnam!");

fclose(fp);

### Xử lý lỗi khi thao tác với file trong C++

Chúng ta có thể sử dụng hàm fopen() để mở một file trong C++, nhưng trong quá trình mở có thể gặp một số lỗi nào đó, dẫn dến việc không phải lúc nào bạn cũng có thể mở file đó thành công.

Sẽ tất nguy hiểm khi thực hiện quá trình thao tác với một tập tin mà không thể mở được, do đó thay vì để chương trình tiếp tục chạy như cũ thì chúng ta cần phải thiết kế các xử lý để chương trình tự xử lý lỗi, và chạy chương trình một cách chính xác.

Nếu hàm fopen() không thể mở file chính xác, nó sẽ trả về giá trị NULL. Chúng ta có thể sử dụng điều này để phán đoán khi nào mở file thất bại và xử lý lỗi khi cần.

Giả sử bạn đã mở một file sample.txt và gán vào con trỏ file fp như sau:

fp = fopen("sample.txt", "r");

Khi đó, nếu mở file thất bại thì giá trị của fp sẽ trở thành NULL, và chúng ta có thể viết xử lý tránh lỗi như sau:

if (fp == NULL) {

//Nội dung xử lý khi lỗi mở file xảy ra

}

Về nội dung xử lý lỗi, thông thường chúng ta sẽ xuất một dòng thông báo “Đã xảy ra lỗi”, và sau đó buộc chương trình phải kết thúc.

Và để buộc chương trình phải kết thúc, chúng ta thường sử dụng tới hàm exit() trong header file cstdlib nhằm cưỡng chế kết thúc chương trình. Để sử dụng hàm này thì chúng ta cần include header file vào đầu chương trình, và gọi hàm tại bất cứ vị trí nào muốn kết thúc chương trình với cú pháp sau đây:

exit(status);

Trong đó status là trạng thái thoát chương trình theo cách bình thường hay bất thường. Đối số status có thể chỉ định bằng một trong hai giá trị là 0 và 1, tương ứng với việc kết thúc bình thường và kết thúc bất thường.

Ngoài ra, chúng ta cũng có thể chỉ định giá trị của status là EXIT\_SUCCESS tương ứng với 0, và EXIT\_FAILURE tương ứng với 1.

Ví dụ, các cách viết sau đều OK khi sử dụng hàm exit() để kết thúc chương trình C++.

// Kết thúc chương trình bình thường

exit(0);

exit(EXIT\_SUCCESS);

// Kết thúc chương trình bất thường

exit(1);

exit(EXIT\_FAILURE);

Lưu ý nếu bạn mở file bên trong hàm main(), thì thay vì dùng hàm exit() thì bạn cũng có thể dùng lệnh return như sau để xử lý lỗi khi mở file trong C++:

//Mở file bằng hàm fopen, và trả về NULL nếu mở file thất bại.

fp = fopen(fname, "r");

if(fp == NULL) {

    cout << fname <<" file not open!\n";

    return -1;

}

Ứng dụng các kiến thức trên, chúng ta có thể viết xử lý lỗi khi mở file trong C++ như sau:

#include <iostream>

#include < cstdlib>

using namespace std;

int main()

{

    FILE \* fp = NULL;

    fp = fopen("sample.txt", "r");

    if (fp == NULL) {

        cout << "Error opening file!"<<endl;

        exit(0);

    }

    fp2 = fopen("sample2.txt", "r");

    if (fp2 == NULL) {

        cout << "Error opening file!"<<endl;

        exit(1);

    }

}

## Mở và đóng file

### Mở file trong C++ bằng hàm fopen

Có 2 loại hàm fopen trong C++ là hàm fopen thông thường và hàm fopen\_s cao cấp. Hàm fopen\_s là một phiên bản an toàn và bảo mật hơn của hàm fopen, và chúng ta sẽ tìm hiểu về nó ở phần sau.

Chúng ta sử dụng hàm fopen thông thường trong C++ với cú pháp sau đây:

fp = fopen(filepath, mode);

Trong đó:

* fp là con trỏ file dùng để gán kiểu cấu trúc FILE được trả về từ hàm nếu mở file thành công
* filepath là đường dẫn tới file cần mở. Đường dẫn này có thể là đường dẫn tương đối, hoặc là đường dẫn tuyệt đối.
* mode là chế độ mở file.

Hàm fopen sẽ trả về một thực thể được tạo ra từ kiểu cấu trúc FILE chứa các thông tin của file đã mở để chúng ta có thể xử lý file. Chúng ta cũng gọi thực thể này là file handle - một đối tượng để xủ lý file trong C++. Nếu mở file thất bại, một giá trị NULL sẽ được trả về.

VD:

#include <iostream>

using namespace std;

int main()

{

    //Khai báo con trỏ file

    FILE \* fp = NULL;

    //Mở file và gán file handle vào con trỏ file

    fp = fopen("sample.txt", "r");

    return 0;

}

Các mode để mở file trong C++

Các chế độ có thể sử dụng để mở file bằng hàm fopen trong C++ như sau:

| **Mode** | **Xử lý** | **Chức năng** |
| --- | --- | --- |
| r | Mở để đọc | Chỉ cho phép đọc file Nếu file không tồn tại thì trả về NULL |
| w | Mở để ghi đè | Xoá nội dung cũ và ghi đè nội dung mới Nếu file không tồn tại thì tạo file mới |
| a | Mở để ghi chèn | Ghi chèn nội dung mới vào cuối file Nếu file không tồn tại thì tạo file mới |
| r+ | Mở để đọc và ghi đè | Cho phép cả đọc và ghi đè Nếu file không tồn tại thì trả về NULL |
| w+ | Mở để đọc và ghi đè | Cho phép cả đọc và ghi đè Nếu file không tồn tại thì tạo file mới |
| a+ | Mở để đọc và ghi chèn | Cho phép cả đọc và ghi chèn Nếu file không tồn tại thì tạo file mới |

### Mở file binary file trong C++

Lại nữa, bằng cách thêm ký tự b vào đằng trước tên mode thì chúng ta cũng có thể mở các binary file - tập tin nhị phân trong C++. Khác với các tập tin văn bản mà chúng ta có thể đọc được các chữ cái, văn bản ghi trong đó, thì Binary File là một tập tin chứa nội dung là các chuỗi nhị phân như sau:

binary file



Các mode dùng để mở binary file trong C++ sẽ là:

| **Mode** | **Xử lý** | **Chức năng** |
| --- | --- | --- |
| rb | Đọc | Chế độ đọc file binary |
| wb | Ghi đè | Chế độ ghi đè file binary |
| ab | Ghi chèn | Chế độ ghi chèn file binary |

### Mở file trong C++ bằng hàm fopen\_s

Hàm fopen\_s trong C++ là một phiên bản an toàn và bảo mật hơn của hàm fopen, và hàm này cũng giúp chúng ta mở file trong C++.

Chúng ta sử dụng hàm fopen\_s trong C++ với cú pháp sau đây:

errno\_t err = fopen\_s(fp, filepath, mode);

Trong đó:

* errno\_t là kiểu lỗi trong C++, và err là tên biến để gán giá trị lỗi nếu mở file thất bại
* fp là con trỏ file dùng để gán kiểu cấu trúc FILE được trả về từ hàm nếu mở file thành công
* filepath là đường dẫn tới file cần mở. Đường dẫn này có thể là đường dẫn tương đối, hoặc là đường dẫn tuyệt đối.
* mode là chế độ mở file. Các mode có thể dùng tương tự như với hàm fopen mà chúng ta đã học ở trên.

Hàm fopen\_s sẽ trả về một số tự nhiên biểu thị số hiệu của lỗi khi mở file. Nếu số này bằng 0 thì việc mở file thành công, và nếu số này khác 0 thì chúng ta có thể tìm ra nội dung lỗi bằng cách đối chiếu số hiệu của lỗi với bảng lỗi.

Trong trường hợp không cần xác định tới kiểu lỗi nếu có khi mở file, chúng ta cũng có thể lược bỏ đi việc gán errno\_t err mà dùng hàm trực tiếp với cú pháp sau đây:

fopen\_s(fp, filepath, mode);

Sau khi mở file thành công, thực thể được tạo ra từ kiểu cấu trúc FILE chứa các thông tin của file đã mở sẽ được gán vào con trỏ fp đã khai báo. Và chúng ta có thể thực hiện các thao tác với file thông qua con trỏ file này như bình thường.

#include <iostream>

using namespace std;

int main(){

    //Mở file bằng hàm fopen\_s mà không kiểm tra lỗi

    FILE \* fp1 = NULL;

    fopen\_s(&fp1, "sample.txt", "r");

    fclose(fp1);

    //Mở file bằng hàm fopen\_s có kiểm tra lỗi

    FILE \* fp2 = NULL;

    errno\_t no2;

    no2 = fopen\_s(&fp2, "memo.txt", "w");

    if (no2 != 0) cout << "Mo file thanh cong";

    fclose(fp2);

}

### Sự khác biệt giữa hàm fopen và fopen\_s

Hàm fopen\_s là một phiên bản an toàn và bảo mật hơn của hàm fopen. Do đó, hàm này cấm mở một file cùng lúc khi đang mở file đó ở chế độ ghi.

Bởi vậy, nếu chúng ta đã mở một file bằng hàm fopen\_s rồi thì chương trình tiếp sẽ kết thúc bất thường vào lần gọi thứ hai của hàm fopen\_s.

Sự khác biệt lớn nhất giữa giữa hàm fopen và fopen\_s cũng chính là điều này. Chúng ta có thể mở một file cùng lúc khi đang mở file đó ở chế độ ghi bằng hàm fopen, nhưng sẽ xảy ra lỗi nếu chúng ta làm việc này với hàm fopen\_s.

Ví dụ, do hàm fopen\_s được gọi 2 lần với cùng môt file, nên chương trình sẽ tự động kết thúc như sau:

#include <iostream>

using namespace std;

int main()

{

    FILE \* fp1 = NULL;

    FILE \* fp2 = NULL;

    errno\_t no1;

    errno\_t no2;

    //Lần mở file đầu tiên với fopen\_s

    no1 = fopen\_s(&fp1, "memo.txt", "w");

    if (no1 != 0)

    {

        cout << "ERR("<<no1<<"): mo lan 1"<<endl;

    }

    //Lần mở cùng một file thứ 2 với fopen\_s

    no2 = fopen\_s(&fp2, "memo.txt", "w");

    if (no2 != 0)

    {

        cout << "ERR("<<no2<<"): mo lan 2"<<endl;

    }

    return 0;

}

RR(13): mo lan 2

### Xử lý lỗi khi mở file trong C++

Như đã phân tích trong bài Xử lý file trong C++ thì chúng ta cần phải xử lý lỗi khi mở một file thất bại trong C++, để chương trình tự xử lý lỗi, và chạy chương trình một cách chính xác.

VD:

//Mở file bằng hàm fopen, và trả về NULL nếu mở file thất bại.

fp = fopen(fname, "r");

if(fp == NULL) {

    cout << fname<<" file not open!\n";

    exit(0);

}

Lưu ý nếu bạn mở file bên trong hàm main(), thì bằng cách dùng lệnh return như sau thì chúng ta cũng có thể xử lý lỗi khi mở file trong C++:

//Mở file bằng hàm fopen, và trả về NULL nếu mở file thất bại.

fp = fopen(fname, "r");

if(fp == NULL) {

    cout << fname<<" file not open!\n";

    return -1;

}

### Tạo file mới trong C++

Bằng cách sử dụng một trong 2 hàm mở file trên, chúng ta có thể tạo một file mới và mở nó ra để xử lý. Ví dụ chúng ta sẽ tạo một file mới có tên memo2.txt bằng hàm fopen() với mode w như sau:

#include <iostream>

using namespace std;

int main()

{

    FILE \* fp = NULL; //Khai báo con trỏ file

    fp = fopen("memo2.txt", "w"); // Mở file và gán file handle vào con trỏ file

    return 0;

}

Bạn cũng có thể sử dụng mode a+ để tạo file mới trong C++:

#include <iostream>

using namespace std;

int main()

{

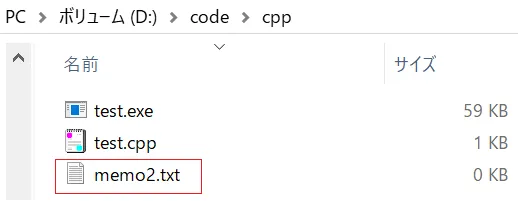
    FILE \* fp = NULL; //Khai báo con trỏ file

    fp = fopen("memo2.txt", "a+"); // Mở file và gán file handle vào con trỏ file

    return 0;

}

Sau khi compile và chạy chương trình, một file trống memo2.txt với kích thước 0 KB sẽ được tạo ra như dưới đây:



### Đóng file trong C++ bằng hàm fclose

Sau khi đã thao tác xong với file trong C++, chúng ta cần đóng file đó lại để kết thúc phiên làm việc với file và giải phóng bộ nhớ trong chương trình.

Và để đóng file trong C++ chúng ta sẽ dùng tới hàm fclose.

Hàm fclose là một hàm có sẵn trong thư viện chuẩn, có tác dụng đóng một file đã mở từ trước đó. Tên hàm fclose được viết tắt bởi cụm từ file và close, được dịch theo tiếng Việt chính xác là hàm đóng file.

Chúng ta sử dụng hàm fclose trong C++ với cú pháp sau đây:

int fclose(FILE \* fp);

Trong đó fp là con trỏ của file cần đóng.

Hàm fclose sẽ trả về 0 trong trường hợp kết thúc file bình thường. Và một giá trị kiểu EOF được trả lại trong trường hợp kết thúc file bất thường.

Ví dụ cụ thể, chúng ta đóng một file sau khi đã mở và hoàn thành các xử lý với nó như sau:

#include <iostream>

using namespace std;

int main(){

    //Mở file

    FILE \* fp = NULL;

    fp = fopen("memo.txt", "w");

    //Đóng file

    fclose(fp);

    return 0;

}

# Tham khảo

Lập trình căn bản:

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

Thuật toán tìm kiếm nhị phân:

<https://www.youtube.com/watch?v=dB2DWSKGLj8>