STL Data Structures: set, map

DATA STRUCTURES AND ALGORITHMS

Nội dung

- **Giới thiệu STL**
  - **Containers**
  - **Iterators**
  - **Function objects**
  - **Algorithms**
- Cấu trúc dữ liệu set
- Cấu trúc dữ liệu map
- Tài liệu tham khảo

Giới thiệu STL

- STL: Standard Template Library
- STL là trái tim của thư viện chuẩn C++ (C++ standard library)
- STL là thư viện chung cung cấp các giải pháp để quản lý các dữ liệu bằng các thuật toán hiện đại và hiệu quả.
- STL gồm các thành phần sau:
  - Containers
  - Iterators
  - Algorithms
  - Function Objects (Functors)

Giới thiệu STL: Containers

Containers dùng để lưu trữ các đối tượng có cùng kiểu. Gồm các loại sau:

- Cấu trúc tuần tự (Sequence containers)
  - **array:** (C++11) gần giống như một mảng tĩnh trong C++
  - **vector**: tương tự mảng động (hỗ trợ thay đổi kích thước)
  - **list**: doubly linked list
  - **forward_list**: (C++11) singly linked list
  - **deque (double-ended queue):** dữ liệu có thể chèn hoặc xóa ở cả hai đầu hàng đợi
- Container adapter: **stack, queue, priority_queue**

Giới thiệu STL: Containers

- Cấu trúc liên kết có thứ tự (Ordered Associative containers)
  - **set**: các giá trị lưu trữ có thứ tự và không trùng nhau
  - **multiset**: các giá trị lưu trữ có thứ tự và có thể trùng nhau
  - **map**: các phần tử trong map gồm 2 thành phần (key-value), trong đó các key không trùng nhau và các phần tử của map được sắp xếp theo thứ tự của key.
  - **multimap**: tương tự như map, tuy nhiên với multimap thì các phần tử trong map có thể có cùng khóa key.
- Cấu trúc liên kết không có thứ tự (Unordered associative containers): **unordered_set, unordered_multiset, unordered_map, unordered_multimap** (C++11): có thể dùng cho cấu trúc dữ liệu không có thứ tự nhưng vẫn có thể được tìm kiếm nhanh chóng bằng cách sử dụng cấu trúc Hashtable.

Giới thiệu STL: STL Algorithms

- STL Algorithm cung cấp một số thuật toán cơ bản để thao tác với các containers
- Chèn `header <algorithm>` để sử dụng các thuật toán trong STL
- Phân loại các thuật toán:

Loại	Thuật toán
So sánh	`equal, includes, lexicographical_compare, max, min, mismatch`
Sao chép	`copy, copy_backward, partial_sort_copy, remove_copy, remove_copy_if, replace_copy, replace_copy_if, reverse_copy, rotate_copy, unique_copy`
Đếm	`count, count_if`
Toán tử Heap	`make_heap, pop_heap, push_heap, sort_heap`
Khởi tạo	`fill, fill_n, generate, generate_n`

Giới thiệu STL: STL Algorithms

Loại	Thuật toán
Toán tử	`accumulate, adjacent_difference, inner_product, partial_sum`
Tìm kiếm	`adjacent_find, binary_search, equal_range, find, find_end, find_first_of, find_if, lower_bound, max_element, min_element, search, search_n, set_difference, set_intersection, set_symmetric_difference, set_union, upper_bound`
Các thao tác sắp xếp	`inplace_merge, iter_swap, merge, next_permutation, nth_element, partial_sort, partial_sort_copy, partition, prev_permutation, remove, remove_copy, remove_copy_if, remove_if, reverse, reverse_copy, rotate, rotate_copy, sort, stable_partition, stable_sort, swap, unique`
Truy xuất dữ liệu	`for_each, replace, replace_copy, replace_copy_if, replace_if, transform, unique_copy`

Giới thiệu STL: Function Objects (Functors)

- **Function objects** (Functors) là các đối tượng được sử dụng với cấu trúc tương tự như hàm bằng cách định nghĩa hàm thành viên operator() trong lớp.
- Có thể sử dụng function objects được khai báo sẵn trong header `<functional>` hoặc tự định nghĩa.
- Functors phổ biến gồm: `arithmetic operations, Comparison operations, logical operations, function Adaptors, …`

Giới thiệu STL: Function Objects (Functors)

Ví dụ: Hàm sort mặc định sẽ dùng thứ tự **operator<** (nghĩa là sắp xếp theo thứ tự tăng dần). Nếu muốn sắp xếp theo thứ tự giảm dần, thì ta sử dụng function object của lớp functor greater, hoặc tự khai báo lớp class_greater với ý nghĩa tương đương **operator>.**

Ví dụ: Khai báo mảng int a[]={2, 11, 5, 3, 7};

- Kết quả mảng a là {2 3 5 7 11} nếu gọi hàm std::sort(a, a+5);

- Kết quả mảng a là {11 7 5 3 2} nếu gọi hàm sort như 2 cách dưới đây:

  + Cách 1: std::sort (a, a+5, std::greater<int>());

  + Cách 2: struct class_greater{

  bool operator() (const int& i, const int& j) const {return i>j;}

 };

std::sort (a, a+5, class_greater());

Giới thiệu STL: Iterators

- Iterator giống như một con trỏ, dùng để duyệt qua các phần tử trong các containers.
- Những container có chứa định nghĩa class iterator sẽ có những phương thức trả về giá trị kiểu iterator tương ứng.
- Ví dụ: Cách khai báo lớp iterator cho các container tương ứng:

list<string> => list<string>::iterator

vector<int> => vector<int>::iterator

set<string> => set<string>::iterator

map<int, char*> => map<int, char*>::iterator

Các phép toán trên iterator

- **Copy contructor iterator**
- **Phép gán (operator=)**: gán giá trị mà iterator trỏ đến
- **++** (postfix hoặc prefix): di chuyển iterator đến phần tử tiếp trong container
- **--** (postfix hoặc prefix): di chuyển iterator đến phần tử liền trước trong container
- ***** : trả về một tham chiếu đến đối tượng được lưu trữ tại vị trí của iterator
- **==** : trả về true nếu 2 iterator đang trỏ tới cùng một vị trí, ngược lại trả về false
- **!=** : trả về true nếu 2 iterator đang trỏ đến các vị trí khác nhau, ngược lại trả về false

Giới thiệu STL: Iterators

Ví dụ: Đoạn code dưới đây dùng để duyệt và in các giá trị của vecto v={1, 2, 3} từ phần tử đầu đến phần tử cuối.

```
vector<int> v={1, 2, 3};
for( vector<int>::iterator itr = v.begin( ); itr != v.end( ); ++itr )
    cout << *itr << ' ';
```

Hoặc dùng vòng lặp while:

```
vector<int>::iterator itr = v.begin( );
while( itr !=v.end( ) )   cout << *itr++ << ' ';
```

Nội dung

- Giới thiệu STL
  - Containers
  - Iterators
  - Function objects
  - Algorithms
- **Cấu trúc dữ liệu set**
- Cấu trúc dữ liệu map
- Tài liệu tham khảo

Cấu trúc set

- Các giá trị được lưu trữ **không trùng nhau** và **có thứ tự** tăng dần (mặc định) hoặc giảm dần.
- Giá trị của các **phần tử trong set không thể sửa** (các phần tử luôn là const), nhưng có thể được chèn hoặc xóa.
- Hỗ trợ các thao tác chèn, xóa và tìm kiếm trong trường hợp xấu nhất là logarit.
- Cài đặt sử dụng cây nhị phân tìm kiếm cân bằng => **cây đỏ đen** sẽ được dùng trong trường hợp này thay vì cây AVL
- Thêm header `<set>` và `namespace std` để sử dụng set
- Khai báo iterator tương ứng để duyệt tập hợp
  set<DataType>::iterator

Cấu trúc set

- Cấu trúc: std::set <DataType> set_name;
- Ví dụ:

```
std::set<int> s1= {8, 10, 0, 7, 1};
// copy constructor
std::set<int> s2(s1);
// range constructor
std::set<int> s3(s1.begin(), s1.end());
// default constructor
std::set<int, greater<int>> s4;
// operator =
s4=s1;
```

- Lưu ý: hàm greater<int>> trong thư viện <functional>

Cấu trúc set: Hàm cơ bản

- Một số hàm cơ bản của set:
  - `begin()`: Trả về iterator trỏ đến phần tử đầu tiên trong tập hợp
  - `end()`: Trả về iterator trỏ đến (lý thuyết) sau phần tử cuối cùng trong tập hợp (không trả về kết quả gì).
  - `size()`: Trả về số lượng phần tử của tập hợp
  - `max_size()`: Trả về số lượng tối đa mà tập hợp có thể lưu trữ
  - `empty()`: Kiểm tra tập hợp rỗng hay không
  - `clear()`: Xóa tất cả các giá trị trong tập hợp

Cấu trúc set: Dùng iterator duyệt tập hợp

- Khai báo iterator tương ứng để duyệt tập hợp set<int>::iterator
- Ví dụ:

```
#include <iostream>
#include <set>

int main () {
   std::set<int> s1= {8, 10, 0, 7, 1};
   std::set<int>::iterator it;
   for (it=s1.begin(); it!=s1.end(); ++it)
      std::cout << *it << ' ';

   return 0;
}
```

Kết quả in ra màn hình:
0 1 7 8 10

Cấu trúc set: Hàm cơ bản

Ví dụ:

```
int a[] = {8, 10, 0, 7, 1};
set<int> s (a, a+5);
for (set<int>::iterator it=s.begin(); it!=s.end(); ++it)
    cout << ' ' << *it;
cout << endl;
cout << "size    : " << s.size() << endl;
cout << "empty   : " << s.empty() << endl;
cout << "max_size: " << s.max_size() << endl;
```

Kết quả:

```
0 1 7 8 10
size     : 5
empty  : false
max_size: 461168601842738790
```

Cấu trúc set: Hàm tìm kiếm

```
iterator find(const Object & x) const;
```

- Hàm trả về iterator trỏ đến phần tử x nếu tìm thấy, ngược lại thì trả về set::end

- Ví dụ:

```
set<int> s={1, 2, 3, 4, 5};
set<int>::iterator it;
it=s.find(7);
if(it==s.end()) cout << "Khong Tim thay.";
else cout << "Tim thay.";
```

Cấu trúc set: Hàm insert

Cấu trúc 1: pair<iterator, bool> insert(const Object & x);

Cấu trúc 2: pair<iterator, bool> insert(iterator hint, const Object & x);

Cấu trúc 3: void insert (InputIterator first, InputIterator last);

- **Tham số:**
  - x: Giá trị cần chèn
  - hint: hint là vị trí gợi ý (không bắt buộc) mà x có thể chèn sau hint, điều này nhằm tối ưu khóa thao tác chèn. Tuy nhiên đây là tập có thứ tự, nên nếu vị trí hint không phù hợp để chèn x thì sẽ đi tìm vị trí thích hợp khác để chèn.
  - first, last: chèn tất cả các phần tử trong phạm vi [first, last), lưu ý giá trị vị trí last sẽ không được chèn.
- **Giá trị trả về:**
  - pair<iterator, bool>: pair::first là iterator trỏ tới phần tử mới được chèn hoặc phần tử bằng với phần tử x cần chèn. pair::second trả về true nếu chèn thành công, trả về false nếu giá trị chèn đã tồn tại.
- **Độ phức tạp:** log(N) với N là số lượng phần tử trong tập set.

Cấu trúc set: Hàm insert

- Thự tự mặc định của set sử dụng function object **less<Object>**, bằng cách gọi **operator<** cho đối tượng.
- Có thể thay đổi thứ tự bằng cách gọi (trong thư viện **<functional>**) hoặc tự định nghĩa function object tương ứng.
- Ví dụ 1: Tạo tập hợp lưu trữ các đối tượng chuỗi không phân biệt chữ hoa chữ thường bằng cách định nghĩa `CaseInsensitiveCompare`.

```
struct CaseInsensitiveCompare {
    bool operator( )( const string & lhs,
                      const string & rhs ) const
    { return strcasecmp( lhs.c_str( ), rhs.c_str( ) ) < 0; }
};
// Lưu ý: strcasecmp trong thư viện <ctring>
```

```
set<string> s;
s.insert("Hello"); s.emplace("HeLLo");
cout << "The size of s is : " << s.size( ) << endl;
```

Kết quả:
The size of s1 is: 2

```
set<string, CaseInsensitiveCompare> s1;
s1.insert("Hello"); s1.emplace("HeLLo");
cout << "The size of s1 is: " << s1.size( ) << endl;
```

Kết quả:
The size of s1 is: 1

Ví dụ 2: Tạo tập hợp chứa các phần tử có thứ tự giảm dần.

```
struct class_greater {
   bool operator() (const int& i, const int& j)const {
      return i>j;
   }
};
std::set<int, class_greater> s;

for (int i=1; i<10; i++)
    s.insert(i);
```

Kết quả:
9 8 7 6 5 4 3 2 1

Hay:

```
std::set<int, greater<int>> s; // sử dụng header <functional>

for (int i=1; i<10; i++)
    s.insert(i);
```

Cấu trúc set: Hàm insert

Ví dụ:

```
set<int, greater<int>> s1;
set<int> s2, s3, s4;
std::set<int>::iterator it;

for(int i = 1; i <=9; i++ )
    s1.insert(s1.end(), i);
for(int i = 9; i >= 1; i-- )
    s2.insert(i);
s3.insert(s2.find(4), s2.find(7));
int a[] = {1, 4, 2, 3, 5};
s4.insert(a, a+4);
```

Kết quả:

s1: 9 8 7 6 5 4 3 2 1

s2: 1 2 3 4 5 6 7 8 9

s3: 4 5 6

s4: 1 2 3 4

Cấu trúc set: Hàm emplace

```
template <class... Args>  pair<iterator, bool> emplace (Args&&... args);
```

Trong đó:

- Các đối số được chuyển tiếp để xây dựng phần tử mới.
- pair<iterator, bool>: pair::first là iterator trỏ tới phần tử mới được chèn hoặc phần tử bằng với phần tử x cần chèn. pair::second trả về true nếu chèn thành công, trả về false nếu giá trị chèn đã tồn tại.

So sánh emplace và insert:

- Hàm emplace là thực hiện chèn in-place tránh copy đối tượng.
- Hàm insert sẽ tạo một đối tượng và sau đó chèn giá trị vào
- Hàm emplace được ưu tiên hơn insert khi thao tác trên các đối tượng

Cấu trúc set: Hàm emplace

Ví dụ 1:

```
set<string> s{};
s.emplace("I"); s.emplace("am"); s.emplace("UITer");
// UITer sẽ không được thêm vào set nữa vì đã tồn tại
s.emplace("UITer");
```

Kết quả:
I UITer am

Ví dụ 2:

```
set<pair<char, int>> s;
s.emplace('y', 9);
// Không dùng được: ms.insert('b', 25);
s.insert(make_pair('h', 8));
```

Kết quả:
h 8
y 9

Lưu ý: pair, make_pair trong thư viện <utility>

Cấu trúc set: Hàm erase

```
Cấu trúc 1: int erase(const Object &x);

Cấu trúc 2: iterator erase(iterator itr);

Cấu trúc 3: iterator erase(iterator start, iterator end);
```

Tham số:

- x: Giá trị cần xóa
- itr: iterator trỏ đến một phần tử cần xóa khỏi set
- start, end: xóa tất cả các phần tử trong phạm vi [start, end), lưu ý giá trị vị trí end sẽ không được xóa

- **Giá trị trả về:**
  - int: số lượng phần tử được xóa trong danh sách: 0 hoặc 1
  - iterator: trả về iterator trỏ đến phần tử sau phần tử bị xóa (hoặc set::end, nếu phần tử cuối cùng bị xóa)
- **Độ phức tạp:** log(N) với N là số lượng phần tử trong tập set

Cấu trúc set: Hàm erase

Ví dụ:

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
std::set<int, std::greater<int>> s(a, a+10);

s.erase (s.find(8));
// Nếu tìm xóa 15 sẽ báo lỗi
// s.erase (s.find(15));
s.erase (6);

s.erase (s.find (3), s.end());
```

Kết quả:
10 9 7 5 4

Nội dung

- Giới thiệu STL
  - Containers
  - Iterators
  - Function objects
  - Algorithms
- Cấu trúc dữ liệu set
- **Cấu trúc dữ liệu map**
- Tài liệu tham khảo

Cấu trúc map

- Mỗi phần tử trong map là một cặp **khóa và giá trị <key - value>**, kiểu dữ liệu của key và value có thể khác nhau.
- Các phần tử trong map có **khóa key không trùng nhau** và **được sắp xếp** tăng dần (mặc định) hoặc giảm dần **theo key**.
- Hỗ trợ các thao tác chèn, xóa và tìm kiếm trong trường hợp xấu nhất là logarit.
- Cài đặt sử dụng cây nhị phân tìm kiếm cân bằng => **cây đỏ đen** sẽ được dùng trong trường hợp này thay vì cây AVL.
- Thêm header `<map>` và namespace std để sử dụng map.
- Dữ liệu trong map có thể được truy xuất trực tiếp bằng khóa tương ứng của chúng bằng cách sử dụng toán tử dấu ngoặc ((toán tử []).

Cấu trúc map

- Cấu trúc:

```
std::map <DataType> map_name;
```

- Ví dụ:

```
//default constructor
std::map<char, int> mp;
mp.insert(std::pair<char,int>('a', 1)); mp['b']= 2; mp['g']= 7;
// range constructor
std::map<char, int> mp1(mp.begin(), mp.end());
// copy constructor
std::map<char, int> mp2(mp);
std::map<char, int> mp3;
// operator =
mp3=mp;
```

Class pair

- Thêm header <utility> và namespace std để sử dụng pair
- pair cho phép gộp 2 giá trị cùng kiểu dữ liệu hoặc khác kiểu thành 1 cặp.
- Lớp pair có các thuộc tính first và second cho phép lấy dữ liệu
- Cấu trúc: pair<dataType1, dataType2> pair_name;
- Ví dụ:

```
pair<string, double> a;

pair<string, double> b("one", 3.5);

pair<string, double> c(b);

std::pair <string, double> d;

d = make_pair ("two", 5.5);

pair <string, double> *e=&b;


cout << d.first << " -> " << d.second << endl;

cout << e->first << " -> " << e->second  << endl;
```

Kết quả:

two -> 5.5

one -> 3.5

Cấu trúc map: Các hàm cơ bản

- `begin()`: Trả về iterator trỏ đến phần tử đầu tiên trong map
- `end()`: Trả về iterator trỏ đến (lý thuyết) sau phần tử cuối cùng trong map.
- `size()`: Trả về số lượng phần tử của map
- `max_size()`: Trả về số lượng tối đa mà map có thể lưu trữ
- `empty()`: Kiểm tra map rỗng hay không
- `clear()`: Xóa tất cả các giá trị trong map
- ...

Cấu trúc map: find

```
iterator find (const key_type& k);
const_iterator find (const key_type& k) const;
```

- Hàm trả về iterator trỏ đến phần tử có khóa key bằng k, ngược lại thì trả về map::end
- Ví dụ:

```
map<string, int> m;
m["one"] = 1; m["two"] = 2; m["three"] = 3;
map<string, int>::iterator it;
it=m.find("two");
if(it==m.end()) cout << "Khong Tim thay.";
else cout << "Tim thay.";
```

Kết quả:

Tim thay.

Cấu trúc map: Chèn sử dụng operator[]

- Có thể chèn các thành phần vào map sử dụng hàm thành viên map::operator[].
- Ví dụ:

```
#include <iostream>
#include <map>
using namespace std;

int main() {
    std::map<char, int> mp;
    mp['a']= 1; mp['b']= 2; mp['c']= 3; mp['d']= 4; mp['e']= 5;
    cout << "map mp:\nkey\tvalue\n";
    for (auto itr = mp.begin(); itr != mp.end(); ++itr)
        cout << itr->first << '\t' << (*itr).second << endl;
    cout << endl << endl;
    return 0;
}
```

Kết quả:

map mp:

key	value
a	1
b	2
c	3
d	4
e	5

Cấu trúc map: insert

```
pair<iterator, bool> insert (const value_type& x);

iterator insert(iterator hint, const value_type& x);

void insert(InputIterator first, InputIterator last);
```

- **Tham số:**
  - x: Giá trị cần chèn
  - hint: hint là vị trí gợi ý (không bắt buộc) mà x có thể chèn sau hint, điều này nhằm tối ưu khóa thao tác chèn. Tuy nhiên đây là map có thứ tự, nên nếu vị trí hint không phù hợp để chèn x thì sẽ đi tìm vị trí thích hợp khác để chèn.
  - first, last: chèn tất cả các phần tử trong phạm vi [first, last), lưu ý giá trị vị trí last sẽ không được chèn.
- **Giá trị trả về:**
  - pair<iterator, bool>: pair::first là iterator trỏ tới phần tử mới được chèn hoặc phần tử có key bằng với key của giá trị x cần chèn. pair::second trả về true nếu chèn thành công, trả về false nếu giá trị chèn đã tồn tại.
- **Độ phức tạp:** log(N) với N là số lượng phần tử trong tập map.

Cấu trúc map: insert

Ví dụ:

```
std::map<char, int> mp, mp1;

mp.insert(std::pair<char,int>('a',1));

mp.insert(std::pair<char,int>('b',2));

mp.insert({'c', 3});

mp.insert({'d', 4});

mp['g']= 7;
mp.insert(mp.find('d'), {'e', 5});



mp1.insert(mp.begin(), mp.find('d'));
```

Sau đoạn lệnh, ta có:

map mp:

key	value
a	1
b	2
c	3
d	4
e	5
g	7

map mp1:

key	value
a	1
b	2
c	3

Cấu trúc map: Hàm erase

```
iterator  erase(const_iterator itr);
size_type erase(const key_type& k);
iterator  erase(const_iterator start, const_iterator end);
```

Tham số:

- k: phần tử cần xóa có khóa key bằng với k
- itr: iterator trỏ đến một phần tử (**đang tồn tại trong map**) cần xóa khỏi map.
- start, end: xóa tất cả các phần tử trong phạm vi [start, end), lưu ý giá trị vị trí end sẽ không được xóa

- **Giá trị trả về:**
  - size_type: số lượng phần tử được xóa trong map: 0 hoặc 1
  - iterator: trả về iterator trỏ đến phần tử sau phần tử bị xóa (hoặc map::end, nếu phần tử cuối cùng bị xóa)
- **Độ phức tạp:** log(N) với N là số lượng phần tử trong tập map

Ví dụ:

```
std::map<char, int> mp;

mp['a']= 1; mp['b']= 2; mp['c']= 3; mp['d']= 4;

mp['e']= 5; mp['f']= 6; mp['g']= 7;


// Xóa k sẽ gặp lỗi do k không có trong ds

// mp.erase(mp.find('k'));

mp.erase('a');

mp.erase(mp.find('c'));

mp.erase(mp.begin(), mp.find('e'));
```

Danh sách trước xóa:

a	1
b	2
c	3
d	4
e	5
f	6
g	7

Danh sách sau xóa:

key	value
e	5
f	6
g	7

Tài liệu tham khảo

- *Data Structures and Algorithm Analysis in C++,* Fourth Edition (2014), MarkAllen Weiss
- *The C++ Standard Library*, Second Edition (2012), Nicolai M. Josuttis
- *The Standard Template Library* (1995), Alexander Stepanov, Meng Lee
- *STL Tutorial and Reference Guide C++ Programming with the Standard Template Library*, Second Edition (2001) - David R.Muser - Grillmer J.Derge - Atul Saini (forword by Alexander Stepanov)

Chúc các em học tốt!