ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO CHUYÊN ĐỀ
PHƯƠNG PHÁP LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG
ĐỀ TÀI: Thư viện STL

Giảng viên hướng dẫn: ThS Nguyễn Minh Huy
Lớp: 20CTT1TN
Thành viên thực hiện:

- 20120059 – Lê Ngọc Đức
- 20120131 – Nguyễn Văn Lộc
- 20120536 – Võ Trọng Nghĩa
- 20120600 – Lê Minh Trí
- 20120607 – Lê Hữu Trọng

THÀNH PHỐ HỒ CHÍ MINH, THÁNG 11 NĂM 2021

Lời nói đầu

C++ Standard Template Library (Thư viện STL) là một tập hợp các lớp dựng sẵn, cung cấp các cấu trúc dữ liệu phổ biến trong lập trình như: list, array, stack, queue, ... và các hàm phục vụ cho các cấu trúc dữ liệu đó, cũng như các thuật toán thông dụng. Nhận thấy thư viện STL là một chủ đề thú vị, giúp ích cho các lập trình viên cho quá trình lập trình nói chung và lập trình hướng đối tượng nói riêng, nhóm chúng em quyết định chọn đây là đề tài tìm hiểu cho chuyên đề môn học Phương pháp lập trình hướng đối tượng.

Mục lục

Danh sách hình vẽ

1 Tổng quan về STL

Thư viện STL là một thư viện cung cấp các "khuôn mẫu" (template) cho các cấu trúc dữ liệu, cung cấp các thuật toán liên quan.
Thư viện STL có 4 "thành phần" (components) chính: [3]

- Thuật toắn (algorithms): gồm các thuật toán được xây dựng chủ yếu cho các cấu trúc dữ liệu, ví dụ như: sort, search, ..
- Containers: chủ yếu gồm các cấu trúc dữ liệu thông dụng.
- Các hàm (functions): thư viện STL chứa các class có chứa toán tử được nạp chồng ("overload the function call operator"). Các class như vậy được gọi là "function objects" hoặc "functors".
- Iterators: iterators thường được dùng để xử lý một dãy các giá trị.

Theo như thỏa thuận giữa hai nhóm cùng làm chuyên đề *Thư viện STL*, nhóm chúng em sẽ trình bày về **containers** và **iterators**.

2 Các khái niệm

2.1 Iterators

Trong C++, iterator (tạm dịch: biến lặp) là đối tượng bất kỳ, trỏ đến một hoặc một số phần tử trong phạm vi của các phần như (như một mảng hoặc một container), có thể dùng để duyệt các phần tử trong phạm vi đó. [4]
Dạng rõ ràng nhất của iterator là một con trỏ.
Iterator có các toán tử như: [4]

- Toán tử so sánh: ==, !=
- Toán tử gán: =
- Toán tử tăng giảm: +, - với một hằng số, ++, –
- Toán tử lấy giá trị (tương tự như con trỏ): *

Phân loại theo chức năng, iterator gồm 4 loại: [5]

- Random access ỉterator
- Bidirectional iterator
- Forward iterator
- Input iterator, output iterator

Ta có bảng tính chất các loại iterators như sau:[5]

<table>
<tr><th colspan="4">category</th><th>properties</th><th>valid expressions</th></tr>
<tr><td colspan="4">all categories</td><td>copy-constructible, copy-assignable and destructible</td><td>X b(a);
b = a;</td></tr>
<tr><td colspan="4">all categories</td><td>Can be incremented</td><td>++a
a++</td></tr>
<tr><td rowspan="7">Random Access</td><td rowspan="5">Bidirectional</td><td rowspan="4">Forward</td><td>Input</td><td>Supports equality/inequality comparisons</td><td>a == b
a != b</td></tr>
<tr><td>Input</td><td>Can be dereferenced as an rvalue</td><td>*a
a->m</td></tr>
<tr><td>Output</td><td>Can be dereferenced as an lvalue
(only for mutable iterator types)</td><td>*a = t
*a++ = t</td></tr>
<tr><td rowspan="2"></td><td>default-constructible</td><td>X a;
X()</td></tr>
<tr><td>Multi-pass: neither dereferencing nor incrementing affects dereferenceability</td><td>{ b=a; *a++; *b; }</td></tr>
<tr><td colspan="2"></td><td>Can be decremented</td><td>--a
a--
*a--</td></tr>
<tr><td colspan="3" rowspan="4"></td><td>Supports arithmetic operators + and -</td><td>a + n
n + a
a - n
a - b</td></tr>
<tr><td>Supports inequality comparisons (<, >, <= and >=) between iterators</td><td>a < b
a > b
a <= b
a >= b</td></tr>
<tr><td>Supports compound assignment operations += and -=</td><td>a += n
a -= n</td></tr>
<tr><td>Supports offset dereference operator ([])</td><td>a[n]</td></tr>
</table>

Hình 1: Tính chất của các loại iterators

Các hàm thao tác thường dùng trên iterator:

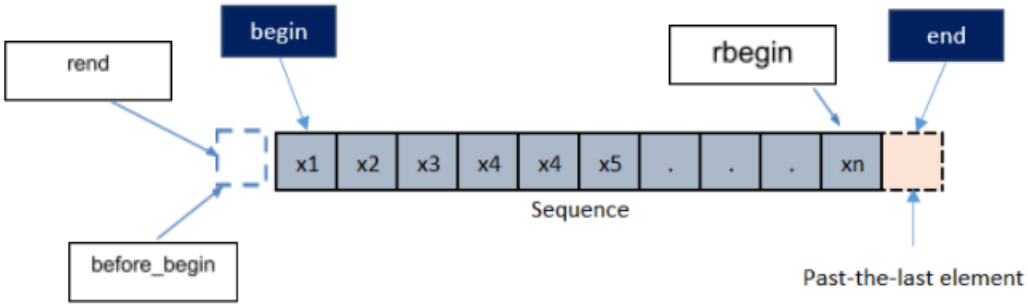
- `next(it, steps)`: tăng biến lặp lên phía trước `steps` lần, nếu không không có giá trị `steps` thì hàm mặc định sẽ tăng 1, trả về `it` sau khi đã tăng.
- `prev(it, steps)`: giảm biến lặp về phía sau `steps` lần, nếu không có giá trị `steps` thì hàm mặc định sẽ giảm 1, trả về `it` sau khi đã giảm.

2.2 Containers

Một container (tạm dịch: thư viện lưu trữ) là một đối tượng lưu trữ một tập các đối tượng khác, được gọi là các phần tử của container đó. Container thường được xây dựng như các "khuôn lớp" (class template).
Các thư viện về “Container" (nằm trong thư viện Template chuẩn - STL) là tập hợp những lớp template và thuật toán, cho phép người lập trình có thể dễ dàng triển khai những cấu trúc dữ liệu như hàng đợi (queue), ngăn xếp (stack), . . .
Container sẽ “sở hữu" các phần tử, tức là vòng đời của phần tử không thể vượt quá vòng đời của chính container chứa nó.
Có một vài cách tiếp cận về việc phân loại các containers, như trong [1] chia containers thành 3 nhóm. Ở đây chúng em sẽ chọn cách chia như trong [6], containers được chia thành 4 nhóm:

- Sequence containers: lưu trữ các phần tử theo thứ tự giống với thứ tự chúng được đưa vào container, bằng một cách thức tuyến tính (a linear manner). Nhóm này gồm `array`, `vector`, `list`, `forward_list`, và `deque`.
- Container adapters: các lớp container thuộc nhóm này thực chất chỉ là các "lớp bao bọc" (wrappers) của các lớp sequence containers. Những container adapters đóng gói (encapsulate) kiểu container được bao bọc, và giới hạn user intefaces của chúng. Nhóm này gồm `stack`, `queue` và `priority_queue`.
- Associative containers: cung cấp các cấu trúc dữ liệu được lưu trữ tự động theo một thứ tự nào đó, cho phép các thao tác tìm kiếm với độ phức tạp $O(\log n)$. Nhóm này thường gồm `set`, `map`, `multiset`, và `multimap`, ngoài ra còn có thể có `hash_set`, `hash_multiset`, `hash_map`, `hash_multimap`. [1]
- Unordered associative containers: cung cấp các cấu trúc dữ liệu chưa được sắp xếp sẵn, có thể được truy cập bằng hash. Thao tác truy cập có độ phức tạp thời gian là $O(n)$ trong trường hợp xấu nhất, nhưng phần lớn phép tính có độ phức tạp tốt hơn nhiều so với độ phức tạp tuyến tính. [6] Nhóm này gồm: `unordered_set`, `unordered_map`, `unordered_multiset`, `unordered_multimap`.

Mỗi container thường có một vài iterators và functions đi kèm, phục vụ cho các thao tác trên container đó. Hình 2 là các iterators thường gặp:



Hình 2: Các iterators thường gặp

Ngoài ra, các hàm thành viên trên sẽ đi kèm với các hàm có tiền tố c- để chỉ rằng hàm trả về biến lặp hằng (const iterator) (ví dụ `cbegin()`, `cend()`).
Trong khuôn khổ báo cáo này, các loại containers sẽ được tìm hiểu cùng với iterators đi kèm.
Sau đây chúng ta sẽ đến với từng loại containers.

3 Sequence containers

Như đã nói trong phần 2.2, sequence containers lưu trữ các phần tử theo thứ tự giống với thứ tự chúng được đưa vào container. Sequence containers gồm: `array`, `vector`, `list`, `forward_list`, và `deque`.

3.1 array

Cấu trúc `array` cung cấp container chứa một mảng tĩnh với số phần tử cố định.
Để sử dụng cấu trúc `array`, ta cần chỉ thị `#include <array>`.
Cấu trúc `array` được khai báo trong header `<array>` như sau:[7]

```
template<class T, std::size_t N>
struct array;
```

3.1.1 Các hàm thành viên

Implicitly-defined member functions: các constructor, destructor, toán tử gán.
Các phương thức truy cập phần tử:

- `at(size_type pos)` : Trả về tham chiếu tới phần tử đối tượng thứ `pos` (bắt đầu từ 0). Hàm sẽ ném ngoại lệ (throw exception) `std::out_of_range` nếu như pos vượt quá phạm vi của mảng (`pos >= N`).
- `operator[](size_type pos)` : Tương tự như trên nhưng không ném ngoại lệ.
- `front()`: Trả về tham chiếu tới phần tử đầu tiên.
- `back()`: Trả về tham chiếu tới phần tử cuối cùng.
- `data()`: Trả về constant pointer (`T*`) của tới mảng nằm trong `array`.

Các iterators như trong phần hình 2.
Các phương thức kiểm tra số lượng (capacity):

- `empty()`: trả về `true` nếu mảng trống và `false` nếu ngược lại.
- `size()`: trả về số lượng phần tử hiện tại của mảng.
- `max_size()`: trả về số lượng phần tử tối đa có thể có của mảng.

Một số phương thức khác:

- `fill(const T& value)`: gán toàn bộ giá trị của mảng bằng `value`.
- `swap(array& other)`: thay đổi nội dung array này với array khác.

Lưu ý: Phải tạo mảng có kích thước lớn hơn 0, nếu không sẽ gây Undefined Behaviour khi truy cập phần tử.
Độ phức tạp của các hàm thành viên có thể được tìm thấy tại [7].

3.1.2 Ví dụ

```
#include <array>

//default initialization (non-local = static storage)
std::array<int, 3> global; //{0, 0, 0}
```

```
int main() {
    //default initialization (local = automatic storage)
    std::array<int, 3> first; //{?, ?, ?}

    //initializer-list initializations
    std::array<int, 3> second = {10, 20}; //{10, 20, 0}
    std::array<int, 3> third = {1, 2, 3}; //{1, 2, 3}

    //copy initialization
    std::array<int, 3> fourth(third); //copy

    return 0;
}
```

3.2 forward_list

Lớp `forward_list` triển khai kiểu dữ liệu danh sách liên kết đơn, hỗ trợ các thao tác chèn, xóa nhanh của phần tử ở bất cứ đâu trong container, tuân theo quy tắc RAII để đảm bảo chương trình không bị memory leak.Tuy nhiên không thể truy cập ngẫu nhiên trên container này được.
Để sử dụng lớp `forward_list`, ta cần chỉ thị `#include <forward_list>`.
Lớp `forward_list` được khai báo trong header `<forward_list>` như sau: [8]

```
template<class T, class Allocator = std::allocator<T>>
class forward_list;
namespace pmr {
    template <class T>
    using forward_list = std::forward_list<T,
       std::pmr::polymorphic_allocator<T>>;
}
```

3.2.1 Các hàm thành viên

Các constructor, destructor, toán tử gán.
Phương thức truy cập phần tử: `front()`: trả về tham chiếu giá trị của HEAD trong DSLK đơn.
Các iterators như trong phần hình 2.
Các phương thức kiểm tra số lượng (capacity):

- `empty()`: trả về `true` nếu mảng trống và `false` nếu ngược lại.
- `max_size()`: trả về số lượng phần tử tối đa có thể có của danh sách.

Các phương thức thao tác trên phần tử:

- `insert_after(pos, (n), value)`: chèn `n` giá trị `value` vào sau vị trí iterator `pos`, `value` ở đây có thể là một giá trị hoặc là một`initializer_list`, nếu không có tham số `n` thì mặc định chèn 1 giá trị. Trả về iterator trỏ tới phần tử cuối cùng được thêm vào.

- `insert_after(pos, beginIt, endIt)`: chèn giá trị copy từ iterator `beginIt` đến `endIt` vào sau vị trí iterator `pos`. Trả về iterator trỏ tới phần tử cuối cùng được thêm vào.
- `erase_after(pos)`: xoá phần tử phía sau vị trí iterator `pos`, trả về iterator trỏ tới phần tử sau phần tử đã xoá.
- `erase_after(begin_pos, end_pos)`: xoá phần tử từ vị trí iterator `begin_pos` đến trước `end_pos`, trả về iterator trỏ tới `end_pos`.
- `push_front(val)`: tương đương việc `addHead` của DSLK đơn, đưa giá trị `val` lên đầu.
- `pop_front()`: tương đương việc deleteHead của DSLK đơn, xoá giá trị cuối cùng của danh sách. Nếu danh sách đang rỗng thì sẽ gây Undefined Behavior.
- `swap(f)`: đổi chỗ các phần tử DSLK đơn hiện tại với DSLK đơn `f` (cùng kiểu).
- `remove(val)`: xoá toàn bộ các Node có giá trị là val trong DSLK đơn.
- `remove_if(f)`: xoá toàn bộ các Node có giá trị thoả điều kiện của function `f`.

Các phương thức với DSLK đơn:

- `merge(l, comparator)`: chèn DSLK `l` vào DSLK hiện tại, 2 danh sách này phải được sắp xếp từ trước, sau đó được chèn theo điều kiện của `comparator` (mặc định là "<" cho tăng dần).
- `reverse()`: lật ngược toàn bộ DSLK.
- `unique()`: xoá các phần tử trùng nhau trong DSLK (giữ lại 1 phần tử).
- `sort()`: sắp xếp DSLK tăng dần.
- `clear()`: xóa toàn bộ các phần tử trong DSLK đơn.

Độ phức tạp của các hàm thành viên có thể được tìm thấy tại forlist.

3.2.2 Ví dụ

```
#include <iostream>
#include <forward_list>

int main() {
    std::forward_list<int> first; //default: empty
    std::forward_list<int> second(3, 77); //fill: 3 seventy-sevens
    std::forward_list<int> third(second.begin(), second.end()); //range
       initialization
    std::forward_list<int> fourth(third); //copy constructor
    std::forward_list<int> fifth(std::move(fourth)); //move ctor. (fourth
       wasted)
    std::forward_list<int> sixth = {3, 52, 25, 90}; //initializer_list
       constructor

    std::forward_list<int> mylist;
```

```
    std::forward_list<int>::iterator it; //it -> ()

    it = mylist.insert_after(mylist.before_begin(), 10); //it -> (10)
    it = mylist.insert_after(it, 2, 20); //10 20 (20)
    std::array<int, 3> myarray = {11, 22, 33};
    it = mylist.insert_after(it, myarray.begin(), myarray.end()); //10 20 30
       11 22 (33)
    it = mylist.begin(); //(10) 20 30 11 22 33
    it = mylist.erase_after(it); //10 (30) 11 22 33
    it = mylist.erase_after(it, mylist.end()); //10 30 ()

    return 0;
}
```

3.3 list

Lớp `list`, triển khai kiểu dữ liệu danh sách liên kết đôi, có những ưu điểm so với `std::forward_list<T>` nhưng tốn bộ nhớ hơn cho iterator hai chiều (bidirectional Iterator).
Để sử dụng lớp `list`, ta cần chỉ thị `#include <list>`.
Lớp `list` được khai báo trong header `<list>` như sau: [9]

```
template<class T, class Allocator = std::allocator<T>>
class list;
namespace pmr {
    template <class T>
    using list = std::list<T, std::pmr::polymorphic_allocator<T>>;
}
```

3.3.1 Các hàm thành viên

Các constructor, destructor, toán tử gán.
Các phương thức truy cập phần tử:

- `front()`: trả về tham chiếu giá trị của HEAD trong DSLK đôi.
- `back()`: trả về tham chiếu giá trị của TAIL trong DSLK đôi

Các iterators như trong phần hình 2.
Các phương thức kiểm tra số lượng (capacity):

- `empty()`: trả về `true` nếu mảng trống và `false` nếu ngược lại.
- `max_size()`: trả về số lượng phần tử tối đa có thể có của danh sách.

Các phương thức thao tác trên phần tử:

- `push_front(val)`, `pop_front()`: tương tự `forward_list`.
- `push_back(val)`: tương đương việc `addTail` của DSLK đôi, đưa giá trị `val` về cuối danh sách.

- `pop_back()`: tương đương việc `deleteTail` của DSLK đôi, xoá giá trị cuối cùng của danh sách. Nếu danh sách đang rỗng thì sẽ gây Undefined Behavior.
- `insert(pos, (n), value)`: chèn `n` giá trị `value` vào trước vị trí iterator `pos`, `value` ở đây có thể là một giá trị hoặc là một `initializer_list`, nếu không có tham số `n` thì mặc định chèn 1 giá trị. Trả về iterator trỏ tới phần tử đầu tiên được thêm vào.
- `insert(pos, beginIt, endIt)`: chèn giá trị copy từ iterator `beginIt` đến `endIt` vào trước vị trí iterator `pos`. Trả về iterator trỏ tới phần tử đầu tiên được thêm vào.
- `erase(pos)`: xoá phần tử ở vị trí iterator `pos`, trả về iterator trỏ tới phần tử sau phần tử đã xoá.
- `erase(begin_pos, end_pos)`: xoá phần tử từ vị trí iterator `begin_pos` đến trước `end_pos`, trả về iterator trỏ tới `end_pos`.

Các phương thức với DSLK đôi: tương tự như `forward_list`.
Độ phức tạp của các hàm thành viên có thể được tìm thấy tại [9].

3.3.2 Ví dụ

```
#include <list>

int main() {
    std::list<int> mylist = {1, 2, 3, 4};
    std::list<int>:: it;

    it = mylist.begin(); //(1) 2 3 4
    it = mylist.insert(it, 10); //(10) 1 2 3 4
    it++; //10 (1) 2 3 4
    it = mylist.insert(it, 2, 3); //10 (3) 3 1 2 3 4

    return 0;
}
```

3.4 vector

Lớp `vector` cung cấp mảng động với khả năng thay đổi số lượng phần tử tùy ý. Do là một associative container nên các phần tử của `vector` được lưu trữ một cách tuần tự, theo thứ tự đúng với thứ tự được nhập vào.
Để sử dụng lớp `vector`, ta sử dụng chỉ thị `#include <vector>`.
Lớp `vector` được khai báo trong header `<vector>` như sau: [12]

```
template<class T, class Allocator = std::allocator<T>> class vector;
namespace pmr {
    template <class T>
    using vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;
}
```

3.4.1 Các hàm thành viên

Các constructor, destructor, toán tử gán.
Các phương thức truy cập phần tử: tương tự như `array`.
Các iterators như trong phần hình 2.
Các phương thức kiểm tra số lượng phần tử:

- `empty()` `size()`, `max_size()`: tương tự như `array`.
- `capacity()`: trả về số phần tử nhiều nhất hiện tại mà vector có thể chứa mà không cần phải cấp phát lại.
- `shrink_to_fit()`: giảm vùng nhớ bằng cách giải phóng những ô nhớ không sử dụng.

Các phương thức thao tác trên phần tử:

- `push_back(val)`: thêm phần tử có giá trị `val` vào vị trí cuối vào `vector`.
- `pop_back()`: xoá phần tử cuối khỏi `vector`.
- `insert()`, `erase()`, `clear()`: tương tự như các phần bên trên.

Độ phức tạp của các hàm thành viên có thể được tìm thấy tại [11].

3.4.2 Ví dụ

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

int main() {
    std::vector<int> a;
    for (int i = 0; i <= 10; i++) {
        a.push_back(i);
        std::cout << a[i] << " ";
    }
    //0 1 2 3 4 5 6 7 8 9 10
    std::cout << "\n";

    int sum = 0;
    for (auto it = a.begin(); it != a.end(); it++) {
        if (*it % 2 == 0) {
            sum += *it;
        }
    }
    std::cout << "Sum of odd number: " << sum << "\n"; //25
}
```

Do cách tạo và hủy của `vector`, khi các phần tử trong `vector` là các đối tượng phức tạp, việc gọi nhiều lần các constructor và destructor sẽ rất tốn kém. Các hướng giải quyết cho vấn đề này được đề cập ở [1].

3.5 Deque

Lớp `deque`, viết tắt của double-ended queue (tạm dịch: hàng đợi kết thúc kép, hàng đợi hai đầu), là một cấu trúc dữ liệu mở rộng của hàng đợi, cho phép thêm và xóa các phần tử ở cả hai đầu.
Để sử dụng lớp `deque`, ta sử dụng chỉ thị `#include <deque>`.
Lớp `deque` được khai báo trong header `<deque>` như sau:[13]

```
template<class T, class Allocator = std::allocator<T>>
class deque;
namespace pmr {
    template <class T>
    using deque = std::deque<T, std::pmr::polymorphic_allocator<T>>;
}
```

3.5.1 Các hàm thành viên

Các constructor, destructor, toán tử gán.
Các phương thức truy cập phần tử: tương tự như `array`.
Các iterators như trong phần hình 2.
Các phương thức kiểm tra số lượng phần tử:

- `empty()` `size()`, `max_size()`: tương tự như `array`.
- `shrink_to_fit()`: giảm vùng nhớ bằng cách giải phóng những ô nhớ không sử dụng.

Các phương thức thao tác trên phần tử:

- `push_back(val)`, `pop_back()`: tương tự như `vector`.
- `push_front(val)`: thêm phần tử có giá trị `val` vào đầu `deque`.
- `pop_front()`: xóa phần tử đầu khỏi `deque`.
- `insert()`, `erase()`, `clear()`: tương tự như các phần bên trên.

Độ phức tạp của các hàm thành viên có thể được tìm thấy tại [13].

3.5.2 Ví dụ

```
#include <deque>

int main() {
    std::deque<int> d; //
    d.push_front(8); // 8
    d.push_front(18); //18 8
    d.push_back(9); //18 8 9
    d.push_back(20); //18 8 9 20

    return 0;
}
```

4 Container adapters

Như đã nhắc đến trong phần 2.2, các lớp thuộc nhóm này thực chất chỉ là wrappers của các containers trong phần 3, nghĩa là các lớp này chỉ khai báo các phương thức xử lý đối với các phần tử trong container, còn phần thực thi bên dưới của phương thức thực chất là của các phương thức trong các sequence containers. Nhóm này gồm: `stack`, `queue`, `priority_queue`.

4.1 stack

Lớp `stack` cung cấp cấu trúc dữ liệu ngăn xếp, hỗ trợ lữu trữ và truy xuất các phần tử theo cơ chế của cấu trúc dữ liệu này, nghĩa là dạng vào sau - ra trước (Last In, First Out – LIFO).
Để sử dụng lớp `stack`, ta sử dụng chỉ thị `#include <stack>`.
Lớp `stack` được khai báo trong header `<stack>` như sau: [14]

```
template<class T, class Container = std::deque<T>>
class stack;
```

4.1.1 Các hàm thành viên

Các constructor, destructor, toán tử gán.
Các phương thức truy cập phần tử: chỉ có phần tử ở đỉnh của ngăn xếp có thể truy cập bằng phương thức `top()`.
Theo như [14], lớp `stack` không cung cấp các iterators.
Các phương thức kiểm tra số lượng phần tử:

- `empty()`, `size()`: tương tự như các containers trong phần 3.

Các phương thức thao tác trên phần tử:

- `push(val)`: thêm một phần tử vào đỉnh của ngăn xếp.
- `pop()`: bỏ phần tử ở đỉnh của ngăn xếp.

Độ phức tạp của các hàm thành viên có thể được tìm thấy tại [14].

4.1.2 Ví dụ

```
#include <iostream>
#include <stack>

int main() {
    std::stack<int> s;
    for (int i = 0; i < 10; i++) {
        s.push(i);
    }

    while (s.empty() == false) {
```

```
        std::cout << s.top() << " ";
        s.pop();
    }
    //9 8 7 6 5 4 3 2 1 0
    return 0;
}
```

4.2 queue

Lớp `queue` cung cấp cấu trúc dữ liệu hàng đợi, hỗ trợ lưu tru và truy xuất các phần tử theo cơ chế của cấu trúc dữ liệu này, nghĩa là theo cơ chế vào trước - ra trước (First In, First Out – FIFO). Nhưng ta chỉ có thêm phần tử vào một đầu và lấy phần tử ra ở đầu còn lại, có thể nhận thấy rằng đây là một trường hợp đặc biệt của `deque`.
Để sử dụng lớp `queue`, ta sử dụng chỉ thị `#include <queue>`.
Lớp `queue` được khai báo trong header `<queue>` như sau: [15]

```
template<class T, class Container = std::deque<T>>
class queue;
```

4.2.1 Các hàm thành viên

Các constructor, destructor, toán tử gán.
Các phương thức truy cập phần tử:

- `front()`: lấy phần tử ở đầu hàng đợi (theo kiểu tham chiếu).
- `back()`: lấy phần tử mới thêm vào cuối hàng đợi (theo kiểu tham chiếu).

Theo như [15], lớp `queue` không cung cấp các iterators.
Các phương thức kiểm tra số lượng phần tử:

- `empty()`, `size()`: tương tự như các containers trong phần 3.

Các phương thức thao tác trên phần tử:

- `push(val)`: thêm một phần tử vào cuối hàng đợi.
- `pop()`: bỏ phần tử ở đầu hàng đợi.

Độ phức tạp của các hàm thành viên có thể được tìm thấy tại [15].

4.2.2 Ví dụ

```
#include <iostream>
#include <queue>

int main() {
    std::queue<int> q;
    for (int i = 0; i < 10; i++) {
        q.push(i);
```

```
    }

    std::cout << q.back() << "\n"; //9
    std::cout << q.front() << "\n"; //0

    while (q.empty() == false) {
        std::cout << q.front() << " ";
        q.pop();
    }
    //0 1 2 3 4 5 6 7 8 9
    return 0;
}
```

4.3 priority_queue

Lớp `priority_queue` cung cấp cấu trúc dữ liệu hàng đợi có độ ưu tiên, cung cấp các thao tác tìm kiếm với chi phí hằng số cho phần tử có độ ưu tiên cao nhất.
Để sử dụng lớp `priority_queue`, ta sử dụng chỉ thị `#include <queue>`.
Lớp `priority_queue` được khai báo trong header `<queue>` như sau: [16]

```
template<class T,
class Container = std::vector<T>,
class Compare=std::less<typename Container::value_type>
> class priority_queue;
```

4.3.1 Các hàm thành viên

Các constructor, destructor, toán tử gán.
Các phương thức truy cập phần tử: tương tự như ngăn xếp, chỉ có phần tử ở đỉnh của hàng đợi ưu tiên có thể truy cập bằng phương thức `top()`.
Theo như [16], lớp `priority_queue` không cung cấp các iterators.
Các phương thức kiểm tra số lượng phần tử:

- `empty()`, `size()`: tương tự như các containers trong phần 3.

Các phương thức thao tác trên phần tử:

- `push(val)`: thêm một phần tử vào hàng đợi có độ ưu tiên và sắp xếp lại thứ tự ưu tiên của các phần tử.
- `pop()`: bỏ phần tử ở đỉnh của hàng đợi ưu tiên.

Độ phức tạp của các hàm thành viên có thể được tìm thấy tại [16].

4.3.2 Ví dụ

```
#include <iostream>
#include <queue>
```

```
int main() {
    std::priority_queue<int> p;
    p.push(1);
    p.push(5);
    p.push(30);
    p.push(10);
    p.push(20);

    std::cout << p.top() << "\n"; //30
    while (p.empty() != false) {
        std::cout << p.top();
        p.pop();
    }
    //30 20 10 5 1

    return 0;
}
```

5 Associative containers

Như đã nhắc đến trong phần 2.2, các lớp thuộc nhóm Associative containers gồm các phần tử được lưu trữ tự động theo một thứ tự nào đó được định nghĩa sẵn (ví dụ như tăng dần hay giảm dần).Nhóm này gồm: `set`, `map`, `multiset`, `multimap`, `hash_set`, `hash_map`, `hash_multiset`, và `hash_multimap`. [1] Trong khuôn khổ chuyên đề này, chúng em chỉ trình bày `set`, `map`, `multiset`, và `multimap`.

5.1 set

Lớp `set` cung cấp một cấu trúc dữ liệu cho phép lưu trữ tập hợp các phần tử cùng kiểu dữ liệu theo một thứ tự nhất định, trong đó các phần tử là duy nhất .Ví dụ, mảng có thể lưu trữ các phần tử trùng nhau như `1, 2, 3, 1, 2, 2, 3, 3, 3`, nhưng `set` chỉ có thể lưu các phần tử khác nhau: `1, 2, 3`.
Trong một `set`, giá trị của một phần tử cũng là khóa (key) nhận diện nó. Giá trị của một phần tử trong `set` không thể thay đổi, nhưng có thể `insert` giá trị mới và `remove` giá trị trong `set`. Các phần tử trong một `set` thường được sắp xếp theo một trật tự yếu nghiêm ngặt (strict weak ordering) được xác định bởi các thành phần so sánh của nó (its internal comparison object). [18]
Để sử dụng lớp `set`, ta cần chỉ thị `#include <set>`.
Lớp `set` được khai báo trong header `<set>` như sau: [17]

```
template<
    class Key,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<Key>
> class set;
namespace pmr {
    template <class Key, class Compare = std::less<Key>>
```

```
    using set = std::set<Key, Compare, std::pmr::polymorphic_allocator<Key>>;
}
```

5.1.1 Các hàm thành viên

Các constructor, destructor, toán tử gán.
Các phương thức truy cập phần tử: theo như [17], lớp `set` không hỗ trợ việc truy cập phần tử.
Các iterators như trong phần hình 2.
Các phương thức kiểm tra số lượng phần tử: `empty()` `size()`, `max_size()`: tương tự như `array`.
Các phương thức thao tác trên phần tử:

- `insert(val)`: chèn phần tử có giá trị `val` vào `set` nếu chưa có giá trị `val` trong `set`.
- `erase(val)`: xóa phần tử có giá trị `val` khỏi `set`.
- `find(val)`: trả về một iterator trỏ đến vị trí của giá trị `val` (nếu có) trong `set`, nếu không có trỏ đến `end()`.
- `count(val)`: trả về số lượng của giá trị `val` trong `set`, do mỗi phần tử trong `set` là duy nhất nên hàm này chỉ trả về giá trị 0 hoặc 1.

Độ phức tạp của các hàm thành viên có thể được tìm thấy tại [17].

5.1.2 Ví dụ

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

int main() {
    std::set<int> s;
    s.insert(1);
    s.insert(2);
    s.insert(3);
    s.insert(1);

    std::cout << s.count(1) << "\n"; // 1
    s.erase(1);
    std::cout << s.count(1) << "\n"; // 0
}
```

5.2 multiset

Lớp `multiset` là một cấu trúc dữ liệu tương tự như `set`, nghĩa là lưu trữ tập hợp các phần tử cùng kiểu dữ liệu theo một thứ tự nhất định, nhưng khác với `set`, `multiset` cho phép các phần tử trùng nhau cùng tồn tại.
Để sử dụng lớp `multiset`, ta sử dụng chỉ thị `#include <set>`.
Lớp `multiset` được khai báo trong header `<set>` như sau: [19]

```
template<
    class Key,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<Key>
    > class multiset;
namespace pmr {
    template <class Key, class Compare = std::less<Key>>
    using multiset = std::multiset<Key, Compare,
                                   std::pmr::polymorphic_allocator<Key>>;
}
```

5.2.1 Các hàm thành viên

Các hàm thành viên của lớp `multiset` hoàn toàn tương tự của lớp `set`, ngoại trừ một vài điểm khác sau đây:

- `insert(val)`: cho phép chèn `val` nếu `val` đã có sẵn trong `set`.
- `erase(val)`: xóa ***tất cả*** phần tử có giá trị `val` khỏi `set`.
- `count(val)`: giá trị trả về của hàm này có thể lớn hơn 1.

5.2.2 Ví dụ

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

int main() {
    std::multiset<int> s = {2, 4, 4, 4, 6, 6, 6, 6, 6};

    for (auto x : s) {
        std::cout << x << " appears " << s.count(x) << " times\n";
    }
    /*
    2 appears 1 times
    4 appears 3 times
    4 appears 3 times
    4 appears 3 times
    6 appears 5 times
    6 appears 5 times
    6 appears 5 times
    6 appears 5 times
    6 appears 5 times
    */
}
```

5.3 map

Lớp `map`, hay còn được gọi là `Dictionary` trong một số ngôn ngữ lập trình như C# hay Python, là một cấu trúc dữ liệu ánh xạ giữa một khóa (key value) sang giá trị của khóa đó (mapped value), tương tự như mảng. Nhưng khác với mảng, key của `map` không nhất thiết phải là một số nguyên mà có thể là bất kỳ một kiểu dữ liệu nào.
Trong một `map`, key value thường được dùng để xác định duy nhất phần tử của map đó. Các phần tử trong một map thường được sắp xếp theo một trật tự yếu nghiêm ngặt (strict weak ordering) được xác định bởi các thành phần so sánh của nó (its internal comparison object). [20]
Để sử dụng lớp `map`, ta sử dụng chỉ thị `#include <map>`.
Lớp `map` được khai báo trong header `<map>` như sau: [21]

```
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T> >
> class map;
namespace pmr {
    template <class Key, class T, class Compare = std::less<Key>>
    using map = std::map<Key, T, Compare,
                         std::pmr::polymorphic_allocator<std::pair<const Key,T>>>
}
```

5.3.1 Các hàm thành viên

Các constructor, destructor, toán tử gán.
Các phương thức truy cập phần tử: hàm `at` tương tự bên trên, toán tử `[]` cho phép truy xuất tới phần tử với khóa của nó.
Các iterators như trong phần hình 2.
Các phương thức kiểm tra số lượng phần tử: `empty()` `size()`, `max_size()`: tương tự như `array`.
Các phương thức thao tác trên phần tử:

- `insert()`: hỗ trợ chèn phần tử vào `map` theo nhiều cách khác nhau.
- `erase()`: xóa phần tử khỏi `map`.
- `count()`: trả về số lượng phần tử với một key xác định, tương tự như `set`, hàm này chỉ trả về 0 hoặc 1.

Độ phức tạp của các hàm thành viên có thể được tìm thấy tại [20].

5.3.2 Ví dụ

```
#include <iostream>
#include <string>
#include <map>
```

```
int main() {
    std::map<std::string, int> m;
    m["banana"] = 1;
    m["apple"] = 2;
    m["orange"] = 3;
    m["pear"] = 4;
    m["grape"] = 5;
    m["banana"]++;

    for (auto x : m) {
        std::cout << x.first << " " << x.second << "\n";
    }

    /*
    apple 2
    banana 2
    grape 5
    orange 3
    pear 4
    */
}
```

Ta nhận thấy trong ví dụ này, các phần tử được sắp xếp tăng dần theo giá trị `key`.

5.4 multimap

Lớp `multimap` cung cấp một cấu trúc dữ liệu tương tự như `map`, nhưng các phần tử khác nhau có thể có cùng giá trị khóa (key value).
Để sử dụng lớp `multimap`, ta cần chỉ thị `#include <map>`.
Lớp `multimap` được khai báo trong header `<map>` như sau: [22]

```
template<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T> >
> class multimap;
namespace pmr {
    template <class Key, class T, class Compare = std::less<Key>>
    using multimap = std::multimap<Key, T, Compare,
                                   std::pmr::polymorphic_allocator<std::pair<const
                                      Key,T>>>;
}
```

5.4.1 Các hàm thành viên

Các hàm thành viên của lớp `multimap` hoàn toàn tương tự của lớp `map`, ngoại trừ việc không thể truy xuất bằng toán tử `[]` do nhiều phần tử có thể có cùng giá trị khóa.

5.4.2 Ví dụ

```
#include <iostream>
#include <string>
#include <map>

int main() {
    std::multimap<std::string, int> m;
    m.insert(std::pair<std::string, int>("banana", 1));
    m.insert(std::pair<std::string, int>("banana", 2));
    m.insert(std::pair<std::string, int>("banana", 3));

    for (auto x : m) {
        std::cout << x.first << " " << x.second << "\n";
    }

    return 0;
    /*
    banana 1
    banana 2
    banana 3
    */
}
```

6 Unordered associative containers

Như đã nói trong phần 2.2, Unordered associative containers cung cấp các cấu trúc dữ liệu chưa được sắp xếp sẵn, có thể được truy cập bằng bảng băm (hash table). Nhóm này gồm: `unordered_set`, `unordered_map`, `unordered_multiset`, và `unordered_multimap`.

6.1 unordered_set

Lớp `unordered_set` cung cấp một cấu trúc dữ liệu tương tự như `set`, nghĩa là các phần tử là duy nhất, nhưng khác với `set`, dữ liệu trong `unordered_set` không có thứ tự (chèn vào một thứ tự nhưng lấy ra sẽ có thứ tự khác).
Để sử dụng lớp `unordered_set`, ta cần chỉ thị `#include <unordered_set>`.
Lớp `unordered_set` được khai báo trong header `<unordered_set>` như sau: [23]

```
template<
    class Key,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<Key>
> class unordered_set;
namespace pmr {
    template <class Key,
              class Hash = std::hash<Key>,
              class Pred = std::equal_to<Key>>
```

```
    using unordered_set = std::unordered_set<Key, Hash, Pred,
                                             std::pmr::polymorphic_allocator<Key>>;
}
```

6.1.1 Các hàm thành viên

Các hàm thành viên của `unordered_set` hoàn toàn tương tự các hàm thành viên của `set`, tuy nhiên có thể có sự khác nhau về độ phức tạp.
Độ phức tạp của các hàm thành viên có thể tìm thấy tại [23].

6.1.2 Ví dụ

```
#include <iostream>
#include <unordered_set>

int main() {
    std::unordered_set<std::string> s;

    s.insert("hello");
    s.insert("world");
    s.insert("today");
    s.insert("is");
    s.insert("a");
    s.insert("sunny");
    s.insert("Friday");

    for (auto it : s) {
        std::cout << it << "\n";
    }

    /*
    sunny
    hello
    world
    today
    Friday
    is
    a
    */
}
```

Lưu ý: thứ tự xuất ra màn hình có thể khác so với ví dụ trên.

6.2 unordered_multiset

Lớp `unordered_multiset` cung cấp một cấu trúc dữ liệu tương tự như `multiset`, nhưng khác với `multiset`, dữ liệu trong `unordered_multiset` không có thứ tự (chèn vào một thứ tự nhưng lấy ra sẽ có thứ tự khác.

Để sử dụng lớp `unordered_multiset`, ta cần chỉ thị `#include <unordered_set>`.
Lớp `unordered_multiset` được khai báo trong header `<unordered_set>` như sau: [24]

```
template<
    class Key,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<Key>
> class unordered_multiset;
namespace pmr {
    template <class Key,
              class Hash = std::hash<Key>,
              class Pred = std::equal_to<Key>>
    using unordered_multiset = std::unordered_multiset<Key, Hash, Pred,
                                  std::pmr::polymorphic_allocator<Key>>
}
```

6.2.1 Các hàm thành viên

Các hàm thành viên của `unordered_multiset` hoàn toàn tương tự các hàm thành viên của `multiset`, tuy nhiên có thể có sự khác nhau về độ phức tạp.
Độ phức tạp của các hàm thành viên có thể được tìm thấy tại [25].

6.2.2 Ví dụ

```
#include <iostream>
#include <unordered_set>

void printMulSet(std::unordered_multiset<int> mts) {
    std::cout << "Unordered multiset: \n";
    for (auto it = mts.begin(); it != mts.end(); it++) {
        std::cout << *it << " ";
    }
    std::cout << "\n";
}

int main() {
    std::unordered_multiset<int> mts({1, 2, 3, 4, 1, 2, 6, 7});
    printMulSet(mts);

    return 0;

    //Unordered multiset: 7 6 4 3 2 2 1 1
}
```

6.3 unordered_map

Lớp `unordered_map` cung cấp một cấu trúc dữ liệu tương tự như `map`, nghĩa là ánh xạ một khóa với một giá trị, nhưng khác với `map`, dữ liệu trong `unordered_map` không có thứ tự (chèn vào một thứ tự nhưng lấy ra sẽ có thứ tự khác.
Để sử dụng lớp `unordered_map`, ta cần chỉ thị `#include <unordered_map>`.
Lớp `unordered_map` được khai báo trong header `<unordered_map>` như sau: [26]

```
template<
    class Key,
    class T,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator< std::pair<const Key, T> >
> class unordered_map;
namespace pmr {
    template <class Key,
              class T,
              class Hash = std::hash<Key>,
              class Pred = std::equal_to<Key>>
    using unordered_map = std::unordered_map<Key, T, Hash, Pred,
                            std::pmr::polymorphic_allocator<std::pair<const
                               Key,T>>>;
}
```

6.3.1 Các hàm thành viên

Các hàm thành viên của `unordered_map` hoàn toàn tương tự các hàm thành viên của `map`, tuy nhiên có thể có sự khác nhau về độ phức tạp.
Độ phức tạp của các hàm thành viên có thể được tìm thấy tại [26].

6.3.2 Ví dụ

```
#include <iostream>
#include <string>
#include <unordered_map>

int main() {
    std::unordered_map<std::string, int> m;

    m.insert(std::make_pair("ten", 1));
    m.insert(std::make_pair("points", 2));
    m.insert(std::make_pair("in", 3));
    m.insert(std::make_pair("oop", 10));

    for (auto x : m) {
        std::cout << x.first << " " << x.second << "\n";
    }
```

```
    return 0;
    /*
        oop 10
        in 3
        points 2
        ten 1
    */
}
```

6.4 unordered_multimap

Lớp `unordered_multimap` cung cấp một cấu trúc dữu liệu tương tự như `multimap`, nhưng khác với `multimap` ở chỗ dữ liệu trong `unordered_multimap` chưa được sắp xếp.
Để sử dụng lớp `unordered_multimap`, ta cần chỉ thị `#include <unordered_map>`.
Lớp `unordered_multimap` được khai báo trong header `<unordered_map>` như sau: [27]

```
template<
    class Key,
    class T,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator< std::pair<const Key, T> >
> class unordered_multimap;
namespace pmr {
    template <class Key, class T,
             class Hash = std::hash<Key>,
             class Pred = std::equal_to<Key>>
    using unordered_multimap = std::unordered_multimap<Key, T, Hash, Pred,
                                std::pmr::polymorphic_allocator<std::pair<const
                                    Key,T>>>;
}
```

6.4.1 Các hàm thành viên

Cá hàm thành viên của `unordered_multimap` hoàn toàn tương tự các hàm thành viên của `multimap`, tuy nhiên có thể có sự khác nhau về độ phức tạp.
Độ phức tạp của các hàm thành viên có thể được tìm thấy tại [27].

6.4.2 Ví dụ

```
#include <iostream>
#include <string>
#include <unordered_map>

void printMulMap(std::unordered_multimap<std::string, int> mtm) {
    std::cout <<"Unordered multimap:\n";
    for (auto it = mtm.begin(); it != mtm.end(); ++it) {
        std::cout << it->first << "," << it->second << "\n";
```

```
    }
    std::cout << "\n";
}

int main() {
    std::unordered_multimap<std::string, int> mtm({{"orange", 1000}, {"mango",
       2000}, {"mango", 3000}});
    mtm.insert(std::make_pair("apple", 5000));
    mtm.insert(std::make_pair("mango", 4000));
    printMulMap(mtm);
    return 0;
    /*
        Unordered multimap:
        apple,5000
        orange,1000
        mango,4000
        mango,3000
        mango,2000
    */
}
```

Tài liệu

[1] Trần Đan Thư, Đinh Bá Tiến, Nguyễn Tấn Trần Minh Khang, ***Lập trình hướng đối tượng***, NXB Khoa học và Kỹ thuật, 2010.

[2] Trần Đan Thư, Nguyễn Thanh Phương, Đinh Bá Tiến, Trần Minh Triết, Đặng Bình Phương, ***Kỹ thuật lập trình***, NXB Khoa học và Kỹ thuật, 2014.

[3] https://www.geeksforgeeks.org/the-c-standard-template-library-stl/

[4] https://vnoi.info/library/56/4958/

[5] https://www.cplusplus.com/reference/iterator/

[6] https://embeddedartistry.com/blog/2017/08/02/an-overview-of-c-stl-containers/

[7] https://en.cppreference.com/w/cpp/container/array

[8] https://en.cppreference.com/w/cpp/container/forward_list

[9] https://en.cppreference.com/w/cpp/container/list

[10] https://www.cplusplus.com/reference/list/list/

[11] https://www.geeksforgeeks.org/vector-in-cpp-stl/

[12] https://en.cppreference.com/w/cpp/container/vector

[13] https://en.cppreference.com/w/cpp/container/deque

[14] https://en.cppreference.com/w/cpp/container/stack

[15] https://en.cppreference.com/w/cpp/container/queue

[16] https://en.cppreference.com/w/cpp/container/priority_queue

[17] https://en.cppreference.com/w/cpp/container/set

[18] https://www.cplusplus.com/reference/set/set/

[19] https://en.cppreference.com/w/cpp/container/multiset

[20] https://www.cplusplus.com/reference/map/map/

[21] https://en.cppreference.com/w/cpp/container/map

[22] https://en.cppreference.com/w/cpp/container/multimap

[23] https://en.cppreference.com/w/cpp/container/unordered_set

[24] https://en.cppreference.com/w/cpp/container/unordered_multiset

[25] https://www.geeksforgeeks.org/unordered_multiset-and-its-uses/

[26] https://en.cppreference.com/w/cpp/container/unordered_map

[27] https://en.cppreference.com/w/cpp/container/unordered_multimap

[28] https://www.geeksforgeeks.org/unordered_multimap-and-its-application/