Ngôn ngữ lập trình C++

CHƯƠNG III. THƯ VIỆN STL

ThS. Phạm Đức Cường

cuongpd@ptit.edu.vn

Posts and Telecommunications Institute of Technology

Học viện Công nghệ Bưu chính Viễn thông

2024

Nội dung chương

- **Phần 1:** Thư viện lưu trữ
- **Phần 2:** Thư viện thuật toán

- Tính năng cho phép định nghĩa hàm hoặc lớp mà không cần chỉ định kiểu dữ liệu cụ thể
- Giúp mã nguồn trở nên linh hoạt và tái sử dụng
- Template hàm:

```
template <typename T>
T max(T a, T b) {
    return (a > b) ? a : b;
}

template <typename T, typename U>
void printPair(T first, U second) {
    std::cout << first << " " << second << std::endl;
}
```

- Template lớp:

```cpp
template <typename T>
class MyClass1 {
private:
    T value;
public:
    MyClass1(T val) {
        value = val;
    }
    T getValue() {
        return value;
    }
};

template <typename T, typename U>
class MyClass2 {
private:
    T key; U value;
public:
    MyClass2(T val1, U val2) {
        key = val1; value = val2;
    }
    void printInfo() {
        std::cout << key << " " << value << std::endl;
    }
};
```

- Mỗi **container** là một cấu trúc dữ liệu có thể lưu trữ nhiều phần tử cùng một kiểu dữ liệu
- Được tạo thành chủ yếu từ class và template
- **Container** quản lý không gian lưu trữ, cung cấp các hàm thành viên hoặc **iterator** để truy cập và thao tác dữ liệu
- **iterator** là một con trỏ đặc biệt giúp truy cập các phần tử trong **container**, có các toán tử giống như con trỏ
- Các phần tử thuộc **container** được lưu trữ trong vùng nhớ heap
- Kiểu dữ liệu của phần tử trong **container** có thể là kiểu dữ liệu tích hợp, dẫn xuất hoặc kiểu dữ liệu do người dùng định nghĩa
- Các **container** thường chứa một số hàm thành viên giống nhau, và luôn hỗ trợ toán tử gán
- Cần hiểu độ phức tạp của từng **container** để chọn phù hợp cho từng bài toán

- **vector** là một **container** lưu trữ dữ liệu theo kiểu mảng động
- Hàm thành viên:
  - Kích thước:
    - **size()** - trả về số phần tử trong **vector**, độ phức tạp $O(1)$
    - **empty()** - kiểm tra **vector** có rỗng hay không, độ phức tạp $O(1)$
  - Truy cập:
    - **operator []** hoặc **at()** - truy cập phần tử tại vị trí chỉ định, độ phức tạp $O(1)$
    - **front()** - trả về phần tử đầu tiên, độ phức tạp $O(1)$
    - **back()** - trả về phần tử cuối cùng, độ phức tạp $O(1)$
    - **begin()** - trả về **iterator** đến phần tử đầu tiên, độ phức tạp $O(1)$
    - **end()** - trả về **iterator** đến ngay sau phần tử cuối cùng, end() không trỏ đến một phần tử thực sự trong **vector** mà trỏ ra ngoài để làm mốc kết thúc cho các thao tác duyệt hoặc tính toán, độ phức tạp $O(1)$

- Chỉnh sửa:
  - **push_back()** - thêm phần tử vào cuối, độ phức tạp $O(1)$
  - **pop_back()** - xóa phần tử cuối cùng, độ phức tạp $O(1)$
  - **insert()** - chèn phần tử vào vị trí chỉ định, độ phức tạp $O(n)$
  - **erase()** - xóa phần tử tại vị trí chỉ định, độ phức tạp $O(n)$
  - **clear()** - xóa tất cả phần tử, độ phức tạp $O(n)$
  - **resize()** - thay đổi kích thước, độ phức tạp $O(n)$
  - **swap()** - hoán đổi dữ liệu giữa hai **vector**, độ phức tạp $O(1)$

- **deque** là một **container** lưu trữ dữ liệu theo kiểu hàng đợi hai đầu
- Hàm thành viên:
  - Kích thước:
    - **size()** - trả về số phần tử trong **deque**, độ phức tạp $O(1)$
    - **empty()** - kiểm tra **deque** có rỗng hay không, độ phức tạp $O(1)$
  - Truy cập:
    - **operator []** hoặc **at()** - truy cập phần tử tại vị trí chỉ định, độ phức tạp $O(1)$
    - **front()** - trả về phần tử đầu tiên, độ phức tạp $O(1)$
    - **back()** - trả về phần tử cuối cùng, độ phức tạp $O(1)$
    - **begin()** - trả về **iterator** đến phần tử đầu tiên, độ phức tạp $O(1)$
    - **end()** - trả về **iterator** đến ngay sau phần tử cuối cùng, end() không trỏ đến một phần tử thực sự trong **deque** mà trỏ ra ngoài để làm mốc kết thúc cho các thao tác duyệt hoặc tính toán, độ phức tạp $O(1)$

- Chỉnh sửa:
  - **push_back()** - thêm phần tử vào cuối , độ phức tạp $O(1)$
  - **push_front()** - thêm phần tử vào đầu, độ phức tạp $O(1)$
  - **pop_back()** - xóa phần tử cuối cùng, độ phức tạp $O(1)$
  - **pop_front()** - xóa phần tử đầu tiên, độ phức tạp $O(1)$
  - **insert()** - chèn phần tử vào vị trí chỉ định, độ phức tạp $O(n)$
  - **erase()** - xóa phần tử tại vị trí chỉ định, độ phức tạp $O(n)$
  - **clear()** - xóa tất cả phần tử, độ phức tạp $O(n)$
  - **resize()** - thay đổi kích thước, độ phức tạp $O(n)$
  - **swap()** - hoán đổi dữ liệu giữa hai **deque**, độ phức tạp $O(1)$

- **stack** là một **container** lưu trữ dữ liệu theo kiểu ngăn xếp - LIFO (Last In First Out - vào sau ra trước)
- Hàm thành viên:
  - Kích thước:
    - **size()** - trả về số phần tử trong **stack**, độ phức tạp $O(1)$
    - **empty()** - kiểm tra **stack** có rỗng hay không, độ phức tạp $O(1)$
  - Truy cập:
    - **top()** - trả về phần tử ở đỉnh, độ phức tạp $O(1)$
  - Chỉnh sửa:
    - **push()** - thêm phần tử vào đỉnh, độ phức tạp $O(1)$
    - **pop()** - xóa phần tử ở đỉnh, độ phức tạp $O(1)$
    - **swap()** - hoán đổi dữ liệu giữa hai **stack**, độ phức tạp $O(1)$

- **queue** là một **container** lưu trữ dữ liệu theo kiểu hàng đợi - FIFO (First In First Out - vào trước ra trước)
- Hàm thành viên:
  - Kích thước:
    - **size()** - trả về số phần tử trong **queue**, độ phức tạp $O(1)$
    - **empty()** - kiểm tra **queue** có rỗng hay không, độ phức tạp $O(1)$
  - Truy cập:
    - **front()** - trả về phần tử ở đầu, độ phức tạp $O(1)$
    - **back()** - trả về phần tử ở cuối, độ phức tạp $O(1)$
  - Chỉnh sửa:
    - **push()** - thêm phần tử vào cuối, độ phức tạp $O(1)$
    - **pop()** - xóa phần tử ở đầu, độ phức tạp $O(1)$
    - **swap()** - hoán đổi dữ liệu giữa hai **queue**, độ phức tạp $O(1)$

- **queue** là một **container** lưu trữ dữ liệu theo kiểu hàng đợi - FIFO (First In First Out - vào trước ra trước)
- Hàm thành viên:
  - Kích thước:
    - **size()** - trả về số phần tử trong **queue**, độ phức tạp $O(1)$
    - **empty()** - kiểm tra **queue** có rỗng hay không, độ phức tạp $O(1)$
  - Truy cập:
    - **front()** - trả về phần tử ở đầu, độ phức tạp $O(1)$
    - **back()** - trả về phần tử ở cuối, độ phức tạp $O(1)$
  - Chỉnh sửa:
    - **push()** - thêm phần tử vào cuối, độ phức tạp $O(1)$
    - **pop()** - xóa phần tử ở đầu, độ phức tạp $O(1)$
    - **swap()** - hoán đổi dữ liệu giữa hai **queue**, độ phức tạp $O(1)$

- **priority_queue** là một **container** lưu trữ dữ liệu theo kiểu hàng đợi ưu tiên, sao cho phần tử lớn nhất luôn ở đầu
- Phép so sánh mặc định là so sánh lớn hơn, tức là phần tử lớn nhất sẽ ở đầu
- Có thể thay đổi phép so sánh để phục vụ cho các bài toán cụ thể:

```
priority_queue<int, vector<int>, greater<int> > pq1;
struct cmp {
    bool operator() (int a, int b) {
        return a > b;
    }
};
priority_queue<int, vector<int>, cmp> pq2;
```

- Hàm thành viên:
  - Kích thước:
    - **size()** - trả về số phần tử trong **priority_queue**, độ phức tạp $O(1)$
    - **empty()** - kiểm tra **priority_queue** có rỗng hay không, độ phức tạp $O(1)$

- Truy cập:
  - **top()** - trả về phần tử ở đầu, độ phức tạp $O(1)$
- Chỉnh sửa:
  - **push()** - thêm phần tử vào cuối, độ phức tạp $O(\log n)$
  - **pop()** - xóa phần tử ở đầu, độ phức tạp $O(\log n)$
  - **swap()** - hoán đổi dữ liệu giữa hai **queue**, độ phức tạp $O(1)$

- **set** là một **container** lưu trữ dữ liệu theo kiểu tập hợp, không chứa các phần tử trùng lặp, các phần tử trong **set** được sắp xếp theo thứ tự tăng dần
- Có thể thay đổi phép so sánh để phục vụ cho các bài toán cụ thể:

```
set<int, greater<int> > s1;
struct cmp {
    bool operator() (int a, int b) {
        return a > b;
    }
};
set<int, cmp> s2;
```

- Hàm thành viên:
  - Kích thước:
    - **size()** - trả về số phần tử trong **set**, độ phức tạp $O(1)$
    - **empty()** - kiểm tra **set** có rỗng hay không, độ phức tạp $O(1)$

- Truy cập:
  - **begin** - trả về **iterator** đến phần tử đầu tiên, độ phức tạp $O(1)$
  - **end** - trả về **iterator** đến ngay sau phần tử cuối cùng
  - **find** - trả về **iterator** đến phần tử cần tìm kiếm, nếu không tìm thấy trả về **end**, độ phức tạp $O(log(n))$
  - **lower_bound** - trả về **iterator** đến phần tử đầu tiên lớn hơn hoặc bằng giá trị cần tìm, nếu không tìm thấy trả về **end**,độ phức tạp $O(log(n))$
  - **upper_bound** - trả về **iterator** đến phần tử đầu tiên lớn hơn giá trị cần tìm, nếu không tìm thấy trả về **end**, độ phức tạp $O(log(n))$
  - **count** - trả về số lần xuất hiện của giá trị cần tìm, tuy nhiên trong set, các phần tử chỉ xuất hiện một lần, nên hàm này có ý nghĩa là sẽ return 1 nếu khóa có trong **set**, và 0 nếu không có, độ phức tạp $O(log(n))$
- Chỉnh sửa:

- **insert** - chèn phần tử vào **set**, độ phức tạp $O(log(n))$
- **erase** - xóa phần tử tại vị trí chỉ định hoặc phần tử cần xóa, độ phức tạp $O(log(n))$
- **clear** - xóa tất cả phần tử, độ phức tạp $O(n)$
- **swap** - hoán đổi dữ liệu giữa hai **set**, độ phức tạp $O(1)$

- **map** là một **container** lưu trữ dữ liệu theo kiểu ánh xạ, mỗi phần tử trong **map** bao gồm một cặp giá trị (key, value), các phần tử trong map được sắp xếp mặc định theo thứ tự tăng dần của key
- Có thể thay đổi phép so sánh để phục vụ cho các bài toán cụ thể:

```
map<char, int > m1;
struct cmp {
    bool operator() (int a, int b) {
        return a > b;
    }
};
map <char, int, cmp> m2;
```

- Hàm thành viên:
  - Kích thước:
    - **size()** - trả về số phần tử trong **map**, độ phức tạp $O(1)$
    - **empty()** - kiểm tra **map** có rỗng hay không, độ phức tạp $O(1)$

- Truy cập:
  - **operator[]** - truy cập phần tử tại key chỉ định nếu đã tồn tại, nếu không tự tạo ra 1 phần tử có key đó trong **map**, độ phức tạp $O(log(n))$
  - **begin** - trả về **iterator** đến phần tử đầu tiên, độ phức tạp $O(1)$
  - **end** - trả về **iterator** đến ngay sau phần tử cuối cùng
  - **find** - trả về **iterator** đến phần tử cần tìm kiếm dựa trên key, nếu không tìm thấy trả về **end**, độ phức tạp $O(log(n))$
  - **lower_bound** - trả về **iterator** đến phần tử đầu tiên lớn hơn hoặc bằng giá trị cần tìm dựa trên key, nếu không tìm thấy trả về **end**,độ phức tạp $O(log(n))$
  - **upper_bound** - trả về **iterator** đến phần tử đầu tiên lớn hơn giá trị cần tìm dựa trên key, nếu không tìm thấy trả về **end**, độ phức tạp $O(log(n))$
  - **count** - trả về số lần xuất hiện của giá trị key trong map, tuy nhiên trong map, các phần tử key chỉ xuất hiện một lần, nên hàm này có ý nghĩa là sẽ return 1 nếu key có trong **map**, và 0 nếu không có, độ phức tạp $O(log(n))$

- Chỉnh sửa:
  - **insert** - chèn phần tử vào **map**, phần tử phải là kiểu **pair**, độ phức tạp $O(log(n))$
  - **erase** - xóa phần tử tại vị trí chỉ định hoặc có key cần xóa, độ phức tạp $O(log(n))$
  - **clear** - xóa tất cả phần tử, độ phức tạp $O(n)$
  - **swap** - hoán đổi dữ liệu giữa hai **map**, độ phức tạp $O(1)$

- **min** - trả về giá trị nhỏ nhất giữa hai giá trị
- **max** - trả về giá trị lớn nhất giữa hai giá trị
- **next_permutation** - sinh ra hoán vị tiếp theo của một hoán vị, trả về **true** nếu hoán vị tiếp theo tồn tại, ngược lại trả về **false**
- **prev_permutation** - sinh ra hoán vị trước của một hoán vị, trả về **true** nếu hoán vị trước tồn tại, ngược lại trả về **false**

- Sắp xếp theo một thứ tự nào đó, mặc định là tăng dần, độ phức tạp trung bình $O(n \log n)$
- Có thể thay đổi phép so sánh để phục vụ cho các bài toán cụ thể:

```
sort(arr, arr + n, greater<int>());

struct cmp1 {
    bool operator() (int a, int b) {
        return a > b;
    }
};
sort(arr, arr + n, cmp1());

bool cmp2(int a, int b) {
    return a > b;
}
sort(arr, arr + n, cmp2);
```

- Tìm kiếm nhị phân với các **container** đã được sắp xếp, độ phức tạp trung bình $O(\log n)$
- Hàm **binary_search** trả về **true** nếu phần tử cần tìm kiếm tồn tại, ngược lại trả về **false**
- Hàm **lower_bound** trả về **iterator** đến phần tử đầu tiên lớn hơn hoặc bằng giá trị cần tìm, nếu không tìm thấy trả về **end**, có thể sử dụng phép so sánh tùy chỉnh để trả về phần tử đầu tiên nhỏ hoặc bằng giá trị cần tìm, độ phức tạp $O(\log n)$
- Hàm **upper_bound** trả về **iterator** đến phần tử đầu tiên lớn hơn giá trị cần tìm, nếu không tìm thấy trả về **end**, có thể sử dụng phép so sánh tùy chỉnh để trả về phần tử đầu tiên lớn hơn giá trị cần tìm, độ phức tạp $O(\log n)$