ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Chương 7: Ứng dụng các cấu trúc dữ liệu cơ bản

Nội dung

1. Nhắc lại các cấu trúc dữ liệu cơ bản
2. Giới thiệu thư viện STL và cách sử dụng các cấu trúc cơ bản
3. Một số ví dụ ứng dụng

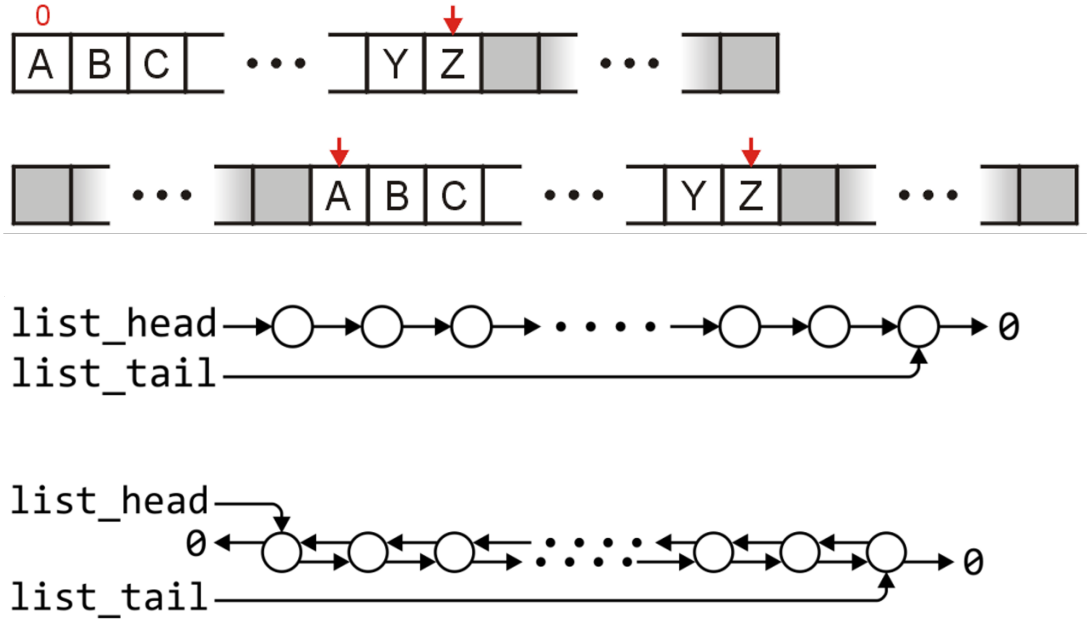
Nhắc lại các CTDL cơ bản

Tại sao cần có các cấu trúc dữ liệu khác nhau?

- Hiệu quả xử lý
  - Mỗi cấu trúc có ưu điểm riêng trong việc lưu trữ và thao tác dữ liệu. Ví dụ:
    - Danh sách liên kết: Hiệu quả cho việc chèn/xóa phần tử
    - Ngăn xếp (stack): Quản lý dữ liệu theo nguyên tắc vào sau ra trước (LIFO)
    - Hàng đợi (queue): Quản lý dữ liệu theo nguyên tắc vào trước ra trước (FIFO)
- Tối ưu hóa bộ nhớ
  - Các cấu trúc khác nhau cho phép sử dụng bộ nhớ một cách linh hoạt và hiệu quả.
- Giải quyết các bài toán cụ thể
  - Mỗi cấu trúc phù hợp với các yêu cầu xử lý khác nhau trong lập trình và thuật toán.
- Tốc độ truy xuất
  - Một số cấu trúc cho phép truy cập dữ liệu nhanh hơn so với các cấu trúc khác.

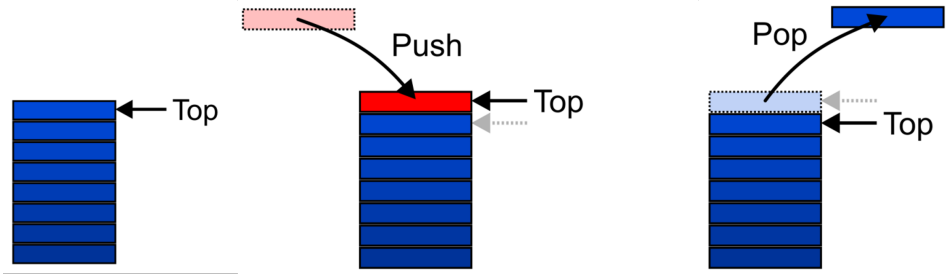
Các cấu trúc dữ liệu cơ bản

- Mảng và danh sách liên kết



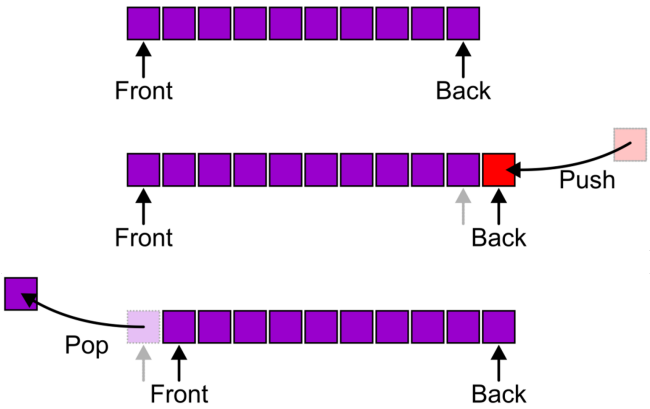
Ngăn xếp (stack)

Tuân theo thứ tự *last-in–first-out* (LIFO)

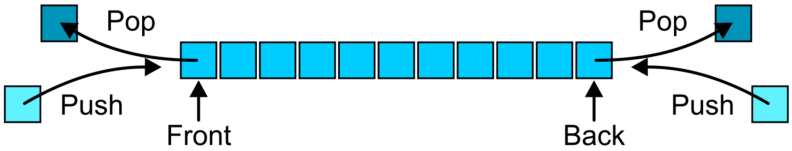


Hàng đợi (queue)

- Tuân theo thứ tự *first-in–first-out* (FIFO)



- Hàng đợi hai đầu (Deque): cho phép chèn và xóa từ hai phía



Giới thiệu thư viện STL và cách sử dụng các cấu trúc dữ liệu cơ bản

Giới thiệu thư viện STL

- STL (Standard Template Library) là thư viện chuẩn của C++, được xây dựng sẵn
- Cài đặt các cấu trúc dữ liệu và thuật toán thông dụng
- Bao gồm các lớp và hàm khuôn mẫu, cho phép làm việc với dữ liệu tổng quát
- Các thành phần chính:
  - Các lớp dữ liệu cơ bản: string, complex
  - Xuất nhập (IO)
  - Các lớp chứa (containers): pair, vector, list, deque, stack, map, set,…
  - Duyệt phần tử của các lớp chứa (iterators)
  - Một số thuật toán thông dụng: tìm kiếm, so sánh, sắp xếp,…
  - Quản lý bộ nhớ, con trỏ
  - Xử lý ngoại lệ (exception handling)

STL Containers

- C++ 98/03 định nghĩa 3 loại container:
  - Sequence Containers: `vector, deque, list`
  - Container Adapters: `stack, queue, priority_queue`
  - Ordered Associative `Containers: [multi]set, [multi]map`
- C++11 bổ sung:
  - `emplace{_front, _back}`
  - Sequence Containers: `array, forward_list`
  - Unordered Associative Containers: `unordered_[multi]set, unordered_[multi]map`
- C++14 bổ sung:
  - Non-member `cbegin, cend, rbegin, rend, crbegin, crend.`

STL Containers

- Màu đỏ nghĩa là có thuật toán độc lập trùng tên
- Không thể duyệt (iterate) theo queue, stack và priority_queue
- Các phương thức * không có trong multi-map

	STL containers	*Some* useful operations
	all containers	size, empty, emplace, erase
Sequence	vector<T>	[], at, clear, insert, back, {emplace,push,pop}_back
	deque<T>	[], at, emplace{,_front,_back}, insert, {,push_,pop_}back, {,push_,pop_}front
	list<T>	insert, emplace, merge, reverse,splice, {,emplace_,push_,pop_}{back,front}, sort
	array<T>	[], at, front, back, max_size
	forward_list<T>	assign, front, max_size, resize, clear, {insert,erase,emplace}_after, {push,pop,emplace}_front
Associative	set<T>, multiset<T>	find, count, insert, clear, emplace, erase, {lower,upper}_bound
	map<T1,T2>, multimap<T1,T2>	[]*, at*, find, count, clear, insert, emplace, erase, {lower,upper}_bound
Unordered Associative	unordered_set<T>, unordered_multiset<T>	find, count, insert, clear, emplace, erase, {lower,upper}_bound
	unordered_map<T1,T2>, unordered_multimap<T1,T2>	[]*, at*, find, count, clear, insert, emplace, erase, hash_function
Container Adaptors	stack	top, push, pop, swap
	queue	front, back, push, pop
	priority_queue	top, push, pop, swap
Other	bitset (N bits)	[], count, any, all, none, set, reset, flip

Trong C++, std::list là một danh sách liên kết kép (doubly linked list).

So sánh các sequence container

Đặc điểm	vector	list	deque	array	forward_list
Cấu trúc	***Mảng động liên tiếp***	***Danh sách liên kết kép***	***Mảng động hai đầu***	***Mảng tĩnh cố định***	***Danh sách liên kết đơn***
Chèn/xóa đầu	Chậm O(n)	Nhanh O(1)	Nhanh O(1)	Không hỗ trợ	Nhanh O(1)
Chèn/xóa cuối	Nhanh O(1)	Nhanh O(1)	Nhanh O(1)	Không hỗ trợ	Không hỗ trợ
Truy cập ngẫu nhiên	Nhanh O(1)	Chậm O(n)	Nhanh O(1)	Nhanh O(1)	Không hỗ trợ
Kích thước	Động	Động	Động	Tĩnh	Động
Bộ nhớ	Liên tiếp	Rải rác	Khối rời rạc	Liên tiếp	Rải rác

Lựa chọn phụ thuộc vào yêu cầu cụ thể của bài toán.

Sequence Containers: vector<T>

- std::vector là mảng động (dynamic array). Hỗ trợ kiểm tra biên (bound checking) khi truy xuất phần tử `vector<T>::at()`
- Các phần tử vector được lưu liên tiếp trong bộ nhớ, do vậy các phép toán số học với con trỏ hoạt động trên vector và đảm bảo truy xuất ngẫu nhiên trong O(1).
- Thực hiện `push_back()` khi vector đã đầy sẽ dẫn tới việc cấp phát mới (reallocation).

Phương pháp truy xuất	Độ phức tạp	API hỗ trợ
Truy xuất ngẫu nhiên	O(1)	`Toán tử[]` hoặc `at()`
Thêm/xóa phần tử cuối	O(1) / O(1)	`push_back` / `pop_back`
Thêm/xóa phần tử đầu	O(n)	Không có
Chèn/xóa ngẫu nhiên	O(n)	`insert / erase`

Sequence Containers: vector<T>

- Khai báo vector

```
std::vector< <data_type> > <vector_name>;
```

- Phương thức khởi tạo

```
vector();//Khởi tạo vector rỗng

vector(size_type n);//Tạo vector có n phần tử. Các phần tử được khởi tạo giá trị mặc định (0 với int)
      size_type (đầy đủ std::vector<int>::size_type, là một kiểu dữ liệu được định nghĩa trong lớp std::vector<int>, thường là std::size_t (một kiểu nguyên không dấu đủ lớn để biểu thị kích thước hoặc chỉ số).

vector(size_type n, const_reference val);//Tạo vector có n phần tử. Mỗi phần tử có giá trị val

vector(vector const &);//Khởi tạo vector bằng cách sao chép vector khác. Tạo bản copy hoàn toàn của vector gốc

vector(vector &&);//Khởi tạo vector bằng cách di chuyển (move). Chuyển dữ liệu từ vector nguồn sang vector đích mà không sao chép

template < class InputIterator >
vector( InputIterator first, InputIterator last );//Tạo vector từ khoảng các phần tử giữa first và last. Có thể từ mảng, vector khác, hoặc bất kỳ container nào

vector( initializer_list<value_type> );//Khởi tạo vector từ danh sách các giá trị. Ví dụ: vector<int> v = {1, 2, 3};
```


Sequence Containers: vector<T>

- Một số ví dụ khởi tạo vector

```
vector<int> vect;//Khởi tạo vector rỗng
vector<int> vect(n, 10);//Khởi tạo vector kích thước n với các giá trị đều bằng 10

vector<int> vect{ 10, 20, 30 };//Khởi tạo vector với danh sách các giá trị

int arr[] = { 10, 20, 30 };
int n = sizeof(arr) / sizeof(arr[0]);
vector<int> vect(arr, arr + n);//Tạo vector từ mảng C. Sao chép các phần tử từ mảng arr. Từ vị trí đầu đến vị trí arr + n

vector<int> vect1{ 10, 20, 30 };
vector<int> vect2(vect1.begin(), vect1.end());//Tạo vector mới bằng cách sao chép từ vector khác. Sao chép toàn bộ phạm vi phần tử của vect1

vector<int> vect1(10);//Tạo vector có 10 phần tử. Các phần tử được khởi tạo với giá trị mặc định (0 đối với kiểu int)

int value = 5;
fill(vect1.begin(), vect1.end(), value);//Điền giá trị value vào toàn bộ vector. Áp dụng sau khi đã khởi tạo vector có sẵn kích thước
```

Sequence Containers: vector<T>

- Phương thức gán
  - Copy assignment:

```
vector& operator= (vector const&);
```

Ví dụ:

```
vector<int> v1 = {1, 2, 3};
vector<int> v2;
v2 = v1; // Sao chép toàn bộ v1 sang v2
// v2 bây giờ là {1, 2, 3}
```

  - Move assignment:

```
vector& operator= (vector&&);//Cú pháp Vector&& other cho phép "di chuyển" tài nguyên thay vì sao chép
```

Ví dụ:

```
vector<int> v1 = {1, 2, 3};
vector<int> v2 = move(v1); // Di chuyển v1 sang v2
// v2 là {1, 2, 3}
// v1 trở thành vector rỗng
```

  - Initializer list assignment:

```
vector& operator= (initializer_list<value_type>);
```

Ví dụ:

```
vector<int> v;
v = {4, 5, 6}; // Gán danh sách giá trị
// v bây giờ là {4, 5, 6}
```


Iterator

- Là một "trừu tượng hóa con trỏ" trong STL, cung cấp giao diện thống nhất để truy cập các phần tử của container.
- So sánh với con trỏ:
  - Giống con trỏ:
    - Truy cập phần tử
    - Di chuyển
    - So sánh
  - Khác con trỏ:
    - Là đối tượng lớp
    - Có thể có logic phức tạp bên trong
    - Được thiết kế riêng cho từng container

```
vector<int> vec = {1, 2, 3};
vector<int>::iterator it = vec.begin();

// Các phép toán tương tự con trỏ
cout << *it;        // Truy cập giá trị
it++;               // Di chuyển đến phần tử kế tiếp
```

Sequence Containers: vector<T>

- Các iterators

  begin end rbegin rend cbegin cend crbegin crend

- **begin():**
  - Trả về iterator trỏ đến phần tử đầu tiên của vector.
- **end():**
  - Trả về iterator trỏ đến vị trí ngay sau phần tử cuối cùng (không phải phần tử cuối).
- **rbegin():**
  - Trả về reverse iterator trỏ đến phần tử cuối cùng (duyệt ngược).
- **rend():**
  - Trả về reverse iterator trỏ đến vị trí ngay trước phần tử đầu tiên (khi duyệt ngược).
- **cbegin():**
  - Trả về const iterator trỏ đến phần tử đầu tiên, không cho phép sửa đổi phần tử.
- **cend():**
  - Trả về const iterator trỏ đến vị trí ngay sau phần tử cuối, không cho phép sửa đổi.
- **crbegin():**
  - Trả về const reverse iterator trỏ đến phần tử cuối cùng, không cho phép sửa đổi.
- **crend():**
  - Trả về const reverse iterator trỏ đến vị trí ngay trước phần tử đầu tiên, không cho phép sửa đổi.

Sequence Containers: vector<T>

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> g1;
    for (int i = 1; i <= 5; i++)
        g1.push_back(i);
    cout << "Output of begin and end: ";
    for (auto i = g1.begin(); i != g1.end(); ++i)
        cout << *i << " ";
    cout << "\nOutput of cbegin and cend: ";
    for (auto i = g1.cbegin(); i != g1.cend(); ++i)
        cout << *i << " ";
    cout << "\nOutput of rbegin and rend: ";
    for (auto ir = g1.rbegin(); ir != g1.rend(); ++ir)
        cout << *ir << " ";
    cout << "\nOutput of crbegin and crend : ";
    for (auto ir = g1.crbegin(); ir != g1.crend(); ++ir)
        cout << *ir << " ";
    return 0;
}
```

```
Output of begin and end: 1 2 3 4 5
Output of cbegin and cend: 1 2 3 4 5
Output of rbegin and rend: 5 4 3 2 1
Output of crbegin and crend : 5 4 3 2 1
```

Ví dụ: begin() vs. cbegin()

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

int main() {
    std::vector<int> vec = {1, 2, 3};

    // Sử dụng begin()
    auto it = vec.begin();
    *it = 10; // Hợp lệ: sửa đổi phần tử đầu tiên thành 10
    std::cout << vec[0] << "\n"; // In: 10

    // Sử dụng cbegin()
    auto cit = vec.cbegin();
    // *cit = 20; // Lỗi biên dịch: cbegin() trả về const iterator, không thể sửa đổi

    return 0;
}
```


Sequence Containers: vector<T>

- Một số cách duyệt phần tử vector khác

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> vect;
    for(int i = 1; i <= 5; i++)
        vect.push_back(i);
    for(int i = 0; i < vect.size(); i++)
        cout << vect[i] << ' ';□ 1 2 3 4 5

    //u là bản sao của từng phần tử. Tạo bản sao mới của mỗi phần tử. Không thể thay đổi giá trị gốc trong vector
    for(auto u : vect)
        cout << u << ' '; □ 1 2 3 4 5
    //&u tham chiếu đến từng phần tử của vector. Không tạo bản sao. Có thể thay đổi trực tiếp giá trị gốc trong vector
    for(auto &u : vect)
        cout << u << ' '; □ 1 2 3 4 5
}
```

Sequence Containers: vector<T>

- Kiểm tra kích thước

```
size_type size() const noexcept;//Trả về số phần tử hiện tại của vector. Kích thước thực tế được sử dụng (const noexcept: không thay đổi container, và không gây ra ngoại lệ.)
size_type capacity() const noexcept;//Trả về dung lượng vector đã cấp phát. Số phần tử tối đa có thể chứa mà không cần cấp phát bộ nhớ mới
size_type maxsize() const noexcept;//Trả về số phần tử tối đa vector có thể chứa. Phụ thuộc vào bộ nhớ hệ thống
void resize( size_type );//Thay đổi kích thước vector thành n. Nếu thu nhỏ, loại bỏ các phần tử ở cuối. Nếu mở rộng, thêm các phần tử mặc định
void resize( size_type, const_reference );//Tương tự resize(n). Các phần tử mới được thêm vào sẽ có giá trị val
bool empty() const noexcept;//Kiểm tra vector có rỗng không. Trả về true nếu không có phần tử
void reserve( size_type );//Đặt trước dung lượng tối thiểu là n. Tránh việc cấp phát lại bộ nhớ khi thêm phần tử
void shrink_to_fit();//Giảm dung lượng xuống bằng kích thước thực tế. Giải phóng bộ nhớ thừa
```


Sequence Containers: vector<T>

- Truy xuất phần tử

```
//Truy xuất phần tử theo chỉ số. Không kiểm tra biên (không an toàn). Nhanh nhưng có thể gây lỗi nếu vượt quá kích thước
reference operator[]( size_type );
const_reference operator[]( size_type ) const;

//Tương tự operator[]. Kiểm tra biên, ném ngoại lệ nếu vượt quá kích thước. An toàn hơn nhưng chậm hơn
reference at( size_type );
const_reference at ( size_type ) const;

//Trả về tham chiếu tới phần tử đầu tiên. Không kiểm tra vector rỗng
reference front();
const_reference front() const;

//Trả về tham chiếu tới phần tử cuối cùng. Không kiểm tra vector rỗng
reference back();
const_reference back() const;

//Trả về con trỏ tới mảng. Cho phép truy cập trực tiếp vùng nhớ vector
pointer data() noexcept;
const_pointer data() const noexcept;
```

Ví dụ:

```
vector<int> v = {10, 20, 30};
cout << v[1];       // 20
cout << v.at(1);    // 20
cout << v.front(); // 10
cout << v.back();  // 30
// Lấy con trỏ đến mảng dữ liệu
int* ptr = v.data();
```

reference vs. const_reference

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

int main() {
    // Vector không const
    std::vector<int> vec = {1, 2, 3};

    // 1. Sử dụng reference operator[](size_type)
    // - Gọi trên vector không const
    // - Cho phép sửa đổi phần tử
    vec[1] = 20; // Sử dụng reference operator[], sửa đổi phần tử tại chỉ số 1
    std::cout << "Sau khi sửa đổi, vec[1] = " << vec[1] << "\n"; // In: 20

    // 2. Sử dụng const_reference operator[](size_type) const
    // - Tạo một vector const
    const std::vector<int> const_vec = {4, 5, 6};
    // - Chỉ cho phép đọc phần tử
    std::cout << "const_vec[1] = " << const_vec[1] << "\n"; // In: 5
    // const_vec[1] = 50; // Lỗi biên dịch: không thể sửa đổi vì const
    return 0;
}
```

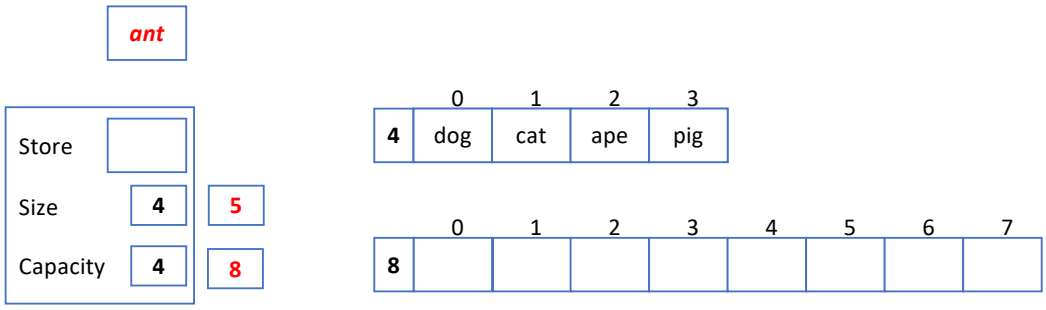

Sequence Containers: vector<T>

- Thay đổi nội dung

```
template < class Iterator >
void assign( Iterator, Iterator );

void assign( size_type, const_reference );
void assign( initializer_list<value_type> );

void push_back( const_reference );
void push_back( value_type&& );
void pop_back();

iterator insert( const_iterator position, const_reference );
iterator insert( const_iterator position, size_type n,
  const_reference );

template < class Iterator >
iterator insert( const_iterator position, Iterator first,
  Iterator last );

iterator insert( const_iterator position, value_type&& );
iterator insert( const_iterator position,
  initializer_list<value_type> );
```

Sequence Containers: vector<T>

- Cơ chế nhân đôi vùng nhớ (Doubling Policy Memory Allocation)
- Giả sử vector đang có capacity là 4 và đã sử dụng hết
- Giờ cần push_back thêm một phần tử mới
- Khi đó một vùng nhớ gấp đôi sẽ được cấp phát, các giá trị cũ được copy sang vùng nhớ mới



Sequence Containers: vector<T>

```
int main () {
    std::vector<int>::size_type cap;
    std::vector<int> foo;
    cap = foo.capacity();
    std::cout << "making foo grow:\n";
    for (int i = 0; i < 100; ++i) {
        foo.push_back(i);
        if (cap!=foo.capacity()) {
            cap = foo.capacity();
            std::cout << "Size changed " << foo.size() << ". Capacity
                          changed: " << cap << '\n';
        }
    }
    std::vector<int> bar;
    cap = bar.capacity();
    bar.reserve(100);     // this is the only difference with foo above
    std::cout << "making bar grow:\n";
    for (int i=0; i<100; ++i) {
        bar.push_back(i);
        if (cap!=bar.capacity()) {
            cap = bar.capacity();
            std::cout << "capacity changed: " << cap << '\n';
        }
    }
    return 0;
}
```

```
making foo grow:
size changed 1. Capacity changed: 1
size changed 2. Capacity changed: 2
size changed 3. Capacity changed: 4
size changed 5. Capacity changed: 8
size changed 9. Capacity changed: 16
size changed 17. Capacity changed: 32
size changed 33. Capacity changed: 64
size changed 65. Capacity changed: 128
making bar grow:
capacity changed: 100
```

STL Containers

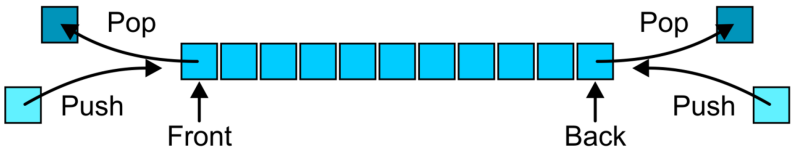
- Màu đỏ nghĩa là có thuật toán độc lập trùng tên
- Không thể duyệt (iterate) theo queue, stack và priority_queue
- Các phương thức * không có trong multi-map

	STL containers	*Some* useful operations
	all containers	size, empty, emplace, erase
Sequence	vector<T>	[], at, clear, insert, back, {emplace,push,pop}_back
Sequence	deque<T>	[], at, emplace{,_front,_back}, insert, {,push_,pop_}back, {,push_,pop_}front
Sequence	list<T>	insert, emplace, merge, reverse,splice, {,emplace_,push_,pop_}{back,front}, sort
Sequence	array<T>	[], at, front, back, max_size
Sequence	forward_list<T>	assign, front, max_size, resize, clear, {insert,erase,emplace}_after, {push,pop,emplace}_front
Associative	set<T>, multiset<T>	find, count, insert, clear, emplace, erase, {lower,upper}_bound
Associative	map<T1,T2>, multimap<T1,T2>	[]*, at*, find, count, clear, insert, emplace, erase, {lower,upper}_bound
Unordered Associative	unordered_set<T>, unordered_multiset<T>	find, count, insert, clear, emplace, erase, {lower,upper}_bound
Unordered Associative	unordered_map<T1,T2>, unordered_multimap<T1,T2>	[]*, at*, find, count, clear, insert, emplace, erase, hash_function
Container Adaptors	stack	top, push, pop, swap
Container Adaptors	queue	front, back, push, pop
Container Adaptors	priority_queue	top, push, pop, swap
Other	bitset (N bits)	[], count, any, all, none, set, reset, flip

Trong C++, std::list là một danh sách liên kết kép (doubly linked list).

Sequence Containers: deque<T>

- Hàng đợi hai đầu "Double-Ended Queue", tương tự như vector nhưng cho phép chèn/xóa nhanh phần tử đầu.
- Truy xuất ngẫu nhiên nhanh, nhưng không đảm bảo các phần tử được lưu giữ kế tiếp nhau trong bộ nhớ
  - Do đó phép toán số học với con trỏ không hoạt động
  - Toán tử [] và at() được đa năng hóa (overloaded) để làm việc đúng



Phương pháp truy xuất	Độ phức tạp	API hỗ trợ
Truy xuất ngẫu nhiên	O(1)	operator[] or at()
Thêm/xóa phần tử cuối	O(1) / O(1)	push_back / pop_back
Thêm/xóa phần tử đầu	O(1) / O(1)	push_front / pop_front
Chèn/xóa ngẫu nhiên	O(n)	insert / erase

Sequence Containers: vector vs. deque

Khía cạnh	Vector	Deque	Lý do
Truy xuất ngẫu nhiên			
Tốc độ	Nhanh hơn	Chậm hơn	Vector: Bộ nhớ liên tục, truy cập trực tiếp Deque: Cần thêm bước tính toán vị trí thật
Cache locality	Tốt hơn	Kém hơn	Vector: Dữ liệu liên tục trong bộ nhớ Deque: Dữ liệu phân tán các khối
Cấp phát lại			
Thêm/xóa đầu	Kém hiệu quả O(n)	Hiệu quả O(1)	Vector: Phải dịch tất cả phần tử Deque: Chỉ cần thêm/xóa ở khối đầu
Thêm/xóa cuối	Hiệu quả amortized O(1)	Hiệu quả O(1)	Vector: Đôi khi cần cấp phát lại Deque: Chỉ cần thêm khối mới nếu cần
Chi phí cấp phát	Cao khi resize	Thấp và từng phần	Vector: Copy toàn bộ sang vùng nhớ mới Deque: Chỉ cấp phát thêm khối khi cần
Ứng dụng			
Truy xuất nhiều	Phù hợp	Ít phù hợp	Vector: Truy cập nhanh và cache friendly Deque: Overhead khi truy cập
Thao tác hai đầu	Ít phù hợp	Phù hợp	Vector: Tốn kém khi thao tác đầu Deque: Hiệu quả ở cả hai đầu
Bộ nhớ	Liên tục lớn	Phân tán nhỏ	Vector: Cần vùng nhớ liên tục lớn Deque: Có thể tận dụng các vùng nhớ nhỏ rời rạc

Sequence Containers

```
#include <vector>
#include <deque>
#include <iostream>
using namespace std;

int main(int argc, char* argv[]) {
// Vector Implementation
   cout << "With a vector:" << endl;
   vector<int> v;
   v.push_back(4); v.push_back(3);
   v.push_back(37); v.push_back(15);
   int* p = &v.back();
   cout << *p << " " << v.at(3) << " "   //must be same
        << p << " " << &v.at(3) << endl; //must be same
   v.push_back(99);
   cout << *p << " " << v.at(3) << " "   //may be different*
        << p << " " << &v.at(3) << endl; //probably different

// Deque Implementation

   cout << "With a deque:" << endl;
   deque<int> d;
   d.push_back(4); d.push_back(3);
   d.push_back(37); d.push_back(15);
   p = &d.back();
   cout << *p << " " << d.at(3) << " "   //must be same
        << p << " " << &d.at(3) << endl; //must be same
   d.resize(32767);            //probably causes realloc
   cout << *p << " " << d.at(3) << " "   //must be same
        << p << " " << &d.at(3) << endl; //must be same
}
```

```
With a vector:
15 15 0x17d32cc 0x17d32cc
0 15 0x17d32cc 0x17d330c
With a deque:
15 15 0x17d338c 0x17d338c
15 15 0x17d338c 0x17d338c
```

Sau khi push_back(99):

1.Nếu capacity đã hết:

- Vector cấp phát vùng nhớ mới lớn hơn
- Copy tất cả phần tử sang vùng nhớ mới
- Con trỏ p trở thành dangling (trỏ đến vùng nhớ cũ)
- Địa chỉ của v.at(3) thay đổi
- *p có thể trả về giá trị không xác định

2.Nếu còn capacity:

- Thêm 99 vào cuối
- Không cấp phát lại
- Con trỏ p và địa chỉ v.at(3) không thay đổi

Kết luận: **Không nên giữ con trỏ đến phần tử vector** vì có thể bị invalid sau các thao tác thêm/xóa.

Sau khi resize(32767):

1. Deque thêm nhiều khối bộ nhớ mới
2. Nhưng khác với vector:
   - Con trỏ p vẫn hợp lệ
   - Địa chỉ của d.at(3) không thay đổi
   - *p vẫn giữ giá trị 15

Lý do:

- Deque sử dụng cấu trúc map of chunks (bản đồ các khối)
- Khi resize, chỉ thêm các khối mới
- Không di chuyển dữ liệu cũ
- Con trỏ đến phần tử cũ vẫn hợp lệ

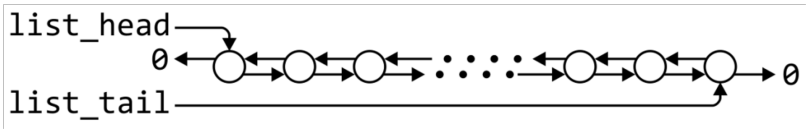
=> Đây là ưu điểm của deque so với vector về mặt quản lý bộ nhớ.

Sequence Containers: std::array (C++11)

- Là một lớp bao (wrapper) của mảng C++, có thể xem như một `vector` kích thước cố định
- `std::array` vs. mảng C++
  - std::array không tự động chuyển thành pointer bởi trình dịch như mảng C++ thường
  - Hỗ trợ các hàm hữu ích
    - Phương thức `at()` cho phép truy xuất phần tử an toàn, kiểm tra biên
    - Phương thức `size()` trả về kích thước mảng được chỉ định khi khởi tạo.
- `std::array` vs. `std::vector`
  - Nếu bạn biết trước kích thước cần dùng thì hãy cố định nó!
  - `std::array` thường được lưu trong bộ nhớ stack hơn là bộ nhớ heap
  - `std::array` nhanh hơn và hiệu quả về mặt bộ nhớ hơn `std::vector`

Sequence Containers: list<T>

- Cài đặt như danh sách liên kết kép doubly-linked list, cho phép chèn và xóa nhanh phần tử.

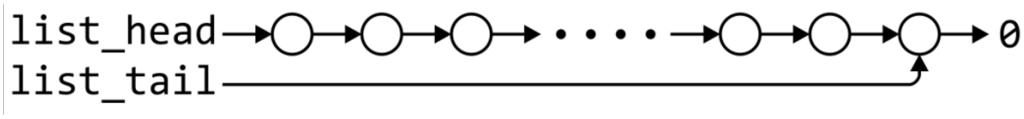


- Chỉ hỗ trợ truy xuất tuần tự thông qua iterators.
  - Không truy xuất ngẫu nhiên qua chỉ số bằng toán tử `[]` hoặc `at()`.

Phương pháp truy xuất	Độ phức tạp	API hỗ trợ
Truy xuất ngẫu nhiên	O(N)	Không hỗ trợ API riêng
Thêm/xóa phần tử cuối	O(1)	`push_back` / `pop_back`
Thêm/xóa phần tử đầu	O(1)	`push_front` / `pop_front`
Chèn/xóa ngẫu nhiên	O(1) (khi đã tới phần tử) O(N) (để tới nơi cần xóa)	`insert` / `erase`

Sequence Containers: std::forward_list (C++11)

- Về cơ bản là danh sách liên kết đơn.
- `std::forward_list` vs. `std::list`
  - Hiệu quả hơn về bộ nhớ, các thao tác chèn/xóa nhanh hơn một chút
  - Không truy xuất ngay lập tức tới cuối danh sách được
    - Nghĩa là không có `push_back()`, `back()`
  - Không thể duyệt ngược danh sách
  - Không có phương thức `size()`



STL Containers

- Màu đỏ nghĩa là có thuật toán độc lập trùng tên
- Không thể duyệt (iterate) theo queue, stack và priority_queue
- Các phương thức * không có trong multi-map

	STL containers	*Some* useful operations
	all containers	size, empty, emplace, erase
Sequence	vector<T>	[], at, clear, insert, back, {emplace,push,pop}_back
	deque<T>	[], at, emplace{,_front,_back}, insert, {,push_,pop_}back, {,push_,pop_}front
	list<T>	insert, emplace, merge, reverse,splice, {,emplace_,push_,pop_}{back,front}, sort
	array<T>	[], at, front, back, max_size
	forward_list<T>	assign, front, max_size, resize, clear, {insert,erase,emplace}_after, {push,pop,emplace}_front
Associative	set<T>, multiset<T>	find, count, insert, clear, emplace, erase, {lower,upper}_bound
	map<T1,T2>, multimap<T1,T2>	[]*, at*, find, count, clear, insert, emplace, erase, {lower,upper}_bound
Unordered Associative	unordered_set<T>, unordered_multiset<T>	find, count, insert, clear, emplace, erase, {lower,upper}_bound
	unordered_map<T1,T2>, unordered_multimap<T1,T2>	[]*, at*, find, count, clear, insert, emplace, erase, hash_function
Container Adaptors	stack	top, push, pop, swap
	queue	front, back, push, pop
	priority_queue	top, push, pop, swap
Other	bitset (N bits)	[], count, any, all, none, set, reset, flip

Trong C++, std::list là một danh sách liên kết kép (doubly linked list).

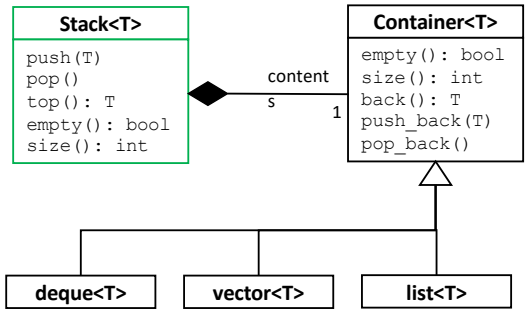
STL Container Adapters

- Đây là các lớp đặc biệt trong C++ STL giúp biến đổi các sequence container cơ bản (như vector, deque, list) thành các cấu trúc dữ liệu chuyên biệt hơn
- Có 3 loại Container Adapters chính:
  1. stack (Ngăn xếp): Hoạt động theo nguyên tắc "vào sau ra trước" (LIFO)
     - Mặc định sử dụng deque, nhưng có thể dùng: vector, deque, list
  2. queue (Hàng đợi): Hoạt động theo nguyên tắc "vào trước ra trước" (FIFO)
     - Mặc định sử dụng deque, có thể dùng: deque, list
  3. priority_queue (Hàng đợi ưu tiên): Các phần tử được sắp xếp theo độ ưu tiên
     - Mặc định sử dụng vector, có thể dùng: vector, deque
- Cách hoạt động:
  - Container Adapters không trực tiếp kế thừa từ container cơ bản
  - Thay vào đó, nó:
    - Tạo một container cơ bản bên trong như một thành phần private
    - Cung cấp các hàm đặc thù (interface) riêng để thao tác với dữ liệu (ví dụ với `stack`, ta cần `push()` và `pop()`)
    - Chuyển các yêu cầu từ interface sang thao tác tương ứng của container bên trong

STL Container Adapters – DP: Adaptor

- Phiên bản đơn giản cài đặt STL Stack

```
// Bên trong stack được cài đặt tương tự như
sau
template <typename T>
class stack {
private:
    deque<T> container;  // Mặc định dùng deque

public:
    void push(const T& value) {
        container.push_back(value);  // Chuyển
                    yêu cầu push sang push_back
của deque
    }

    void pop() {
        container.pop_back();  // Chuyển yêu
                    cầu pop sang pop_back của
deque
    }
};
```



Điểm mạnh của cách thiết kế này:

- Đơn giản hóa interface cho người dùng
- Tăng tính bảo mật dữ liệu
- Linh hoạt trong việc thay đổi container nền tảng
- Tái sử dụng code hiệu quả

STL Container Adapters: stack

- Khai báo

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

  - Đây là khai báo template, cho phép class stack làm việc với nhiều kiểu dữ liệu khác nhau
    - `class T`:
      - Tham số template đầu tiên
      - Đại diện cho kiểu dữ liệu sẽ được lưu trữ trong stack
      - Ví dụ: int, double, string,...
    - `class Container = deque<T>`:
      - Tham số template thứ hai
      - Đại diện cho loại container sẽ được dùng để lưu trữ dữ liệu
      - `deque<T>` là giá trị mặc định
      - Có thể thay đổi thành `vector<T>` hoặc `list<T>`
- Các thao tác với stack

  `empty()` : kiểm tra stack rỗng hay không

  `size()` : Trả về kích thước của stack

  `top()` : Trả về tham chiếu tới phần tử trên cùng của stack

  `push(g)`: Thêm phần tử 'g' lên đỉnh của stack

  `emplace()`: Khởi tạo và chèn phần tử lên đỉnh của stack. Hiệu quả hơn hàm push() do không mất thời gian copy đối tượng

  `pop()` – Xóa phần tử trên cùng của đỉnh stack

```
// Cách 1: Dùng container mặc định (deque)
stack<int> s1;
// Lưu số nguyên, dùng deque
stack<string> s2;
// Lưu chuỗi, dùng deque

// Cách 2: Chỉ định container khác
stack<int, vector<int>> s3;
// Lưu số nguyên, dùng vector
stack<string, list<string>> s4;
// Lưu chuỗi, dùng list

// Ví dụ sử dụng
s1.push(10);
// Thêm số 10 vào stack
s2.push("Hello");
// Thêm chuỗi "Hello" vào stack
```


STL Container Adapters: queue

- Khai báo

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

- Các thao tác với queue

  `empty()`: trả về queue rỗng hay không.

  `size()`: trả về kích thước của queue.

  `front()`: trả về tham chiếu tới phần tử đầu tiên trong queue

  `back()`: trả về tham chiếu tới phần tử cuối cùng trong queue.

  `push(g)`: thêm phần tử 'g' vào cuối queue

  `emplace()`: khởi tạo và chèn phần tử mới vào cuối queue.

  `pop()`: xóa phần tử đầu tiên trong queue

STL Container Adapters: priority_queue

- Khai báo

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

  - Trong đó: class `Compare = less<typename Container::value_type>`:
    - Định nghĩa cách so sánh các phần tử
    - Mặc định là `less<T>` (sắp xếp tăng dần)
    - `typename Container::value_type` là kiểu dữ liệu của phần tử trong container

- Cấu trúc hàng đợi ưu tiên đảm bảo phần tử được ưu tiên nhất luôn ở đầu hàng đợi
- Các thao tác với priority_queue (tương tự queue)

  `empty()`: trả về queue rỗng hay không.
  `size()`: trả về kích thước của queue.
  `front()`: trả về tham chiếu tới phần tử đầu tiên trong queue
  `back()`: trả về tham chiếu tới phần tử cuối cùng trong queue.
  `push(g)`: thêm phần tử 'g' vào cuối queue
  `emplace()`: khởi tạo và chèn phần tử mới vào cuối queue.
  `pop()`: xóa phần tử đầu tiên trong queue

STL Containers

	STL containers	*Some* useful operations
	all containers	size, empty, emplace, erase
Sequence	vector<T>	[], at, clear, insert, back, {emplace,push,pop}_back
	deque<T>	[], at, emplace{,_front,_back}, insert, {,push_,pop_}back, {,push_,pop_}front
	list<T>	insert, emplace, merge, reverse,splice, {,emplace_,push_,pop_}{back,front}, sort
	array<T>	[], at, front, back, max_size
	forward_list<T>	assign, front, max_size, resize, clear, {insert,erase,emplace}_after, {push,pop,emplace}_front
Associative	set<T>, multiset<T>	find, count, insert, clear, emplace, erase, {lower,upper}_bound
	map<T1,T2>, multimap<T1,T2>	[]*, at*, find, count, clear, insert, emplace, erase, {lower,upper}_bound
Unordered Associative	unordered_set<T>, unordered_multiset<T>	find, count, insert, clear, emplace, erase, {lower,upper}_bound
	unordered_map<T1,T2>, unordered_multimap<T1,T2>	[]*, at*, find, count, clear, insert, emplace, erase, hash_function
Container Adaptors	stack	top, push, pop, swap
	queue	front, back, push, pop
	priority_queue	top, push, pop, swap
Other	bitset (N bits)	[], count, any, all, none, set, reset, flip

STL Associative Containers

- Container liên kết có thứ tự (Associative Containers)

  `[multi]map, [multi]set`

  - Là các container lưu trữ và quản lý dữ liệu theo dạng key-value hoặc key
  - Tự động sắp xếp các phần tử
  - Thường được cài đặt bằng cây nhị phân tìm kiếm cân bằng

- Container liên kết không có thứ tự (Unordered Associative Containers)

  `unordered_[multi]map, unordered_[multi]set`

  - Là phiên bản không sắp xếp của các associative containers
  - Sử dụng bảng băm (hash table) để lưu trữ
  - Tối ưu cho tìm kiếm nhanh O(1) trung bình

STL Associative Containers – set<T>

- Container lưu trữ các phần tử duy nhất
- Các phần tử tự động được sắp xếp
- Được cài đặt dựa trên cây đỏ-đen (Red-Black Tree)
- Không cho phép phần tử trùng lặp
- Cung cấp các thao tác chèn, xóa, tìm kiếm với độ phức tạp O(log n)

```
#include <set>

//Khai báo
set<int> mySet;                  // Khai báo set số nguyên
set<string> stringSet;           // Khai báo set chuỗi
set<int, greater<int>> descendingSet;  // Sắp xếp giảm dần
```

set<T> - Các thao tác cơ bản

```
// Thêm phần tử
mySet.insert(10);
mySet.insert(5);
mySet.insert(10);  // Bị bỏ qua vì 10 đã tồn tại

// Xóa phần tử
mySet.erase(5);    // Xóa theo giá trị
auto it = mySet.begin();
mySet.erase(it);   // Xóa theo iterator

// Tìm kiếm
auto found = mySet.find(10);
if(found != mySet.end()) {
    cout << "Tìm thấy: " << *found << endl;
}

// Kích thước và kiểm tra rỗng
int size = mySet.size();
bool empty = mySet.empty();
```


set<T> - Duyệt set

```
// Cách 1: Iterator
for(auto it = mySet.begin(); it != mySet.end(); ++it) {
    cout << *it << " ";
}

// Cách 2: Range-based for loop
for(const auto& element : mySet) {
    cout << element << " ";
}

// Duyệt ngược
for(auto it = mySet.rbegin(); it != mySet.rend(); ++it) {
    cout << *it << " ";
}
```

set<T> - Tùy chỉnh cách so sánh

```
#include <iostream>
#include <set>
#include <string>
#include <functional>

using namespace std;

void printDescendingSet(const set<int, greater<int>>& descendingSet) {
    cout << "Các phần tử trong set sắp xếp giảm dần:" << endl;
    for (int num : descendingSet) {
        cout << num << " ";
    }
    cout << endl;
}

int. main() {
    // Set số nguyên sắp xếp giảm dần
    // Sử dụng greater<> để sắp xếp giảm dần
    set<int, greater<int>> descendingSet;

    // Chèn các phần tử
    descendingSet.insert(10);
    descendingSet.insert(5);
    descendingSet.insert(20);
    descendingSet.insert(15);
    descendingSet.insert(7);

    // In set số nguyên giảm dần
    printDescendingSet(descendingSet);

    return 0;
}
```


set<T> - Tùy chỉnh cách so sánh

```
#include <iostream>
#include <set>
#include <string>
#include <functional>

using namespace std;

void printLengthSet(const set<string, CompareLength>& lengthSet) {
    cout << "Các chuỗi theo thứ tự tăng dần độ dài:" << endl;
    for (const string& str : lengthSet) {
        cout << str << " (độ dài: " << str.length() << ")" << endl;
    }
}

int main() {
    // Định nghĩa bộ so sánh theo độ dài chuỗi
    struct CompareLength {
        bool operator()(const string& a, const string& b) const {
            return a.length() < b.length();
        }
    };

    // Set chuỗi sắp xếp theo độ dài
    set<string, CompareLength> lengthSet;

    // Chèn các chuỗi
    lengthSet.insert("apple");
    lengthSet.insert("banana");
    lengthSet.insert("cherry");
    lengthSet.insert("date");
    lengthSet.insert("elderberry");

    // In set chuỗi theo độ dài
    printLengthSet(lengthSet);

    return 0;
}
```

set<T> - Các thao tác hữu ích khác

```
// Lấy phần tử đầu/cuối
int first = *mySet.begin();
int last = *mySet.rbegin();

// Đếm số lần xuất hiện (0 hoặc 1)
int count = mySet.count(10);

// Lấy vị trí phần tử đầu tiên >= giá trị
auto lower = mySet.lower_bound(10);

// Lấy vị trí phần tử đầu tiên > giá trị
auto upper = mySet.upper_bound(10);

// Lấy cả khoảng [lower_bound, upper_bound]
auto range = mySet.equal_range(10);
```


set<T> - Ví dụ thực tế

```
#include <iostream>
#include <set>
using namespace std;
struct Student {
    string id;
    string name;
    //Sắp xếp tăng dần và không trùng lặp (<) theo id
    bool operator<(const Student& other) const {
        return id < other.id;  // So sánh theo ID
    }
};

int main() {
    set<Student> students;
    students.insert({"001", "Alice"});
    students.insert({"003", ”Mary"});
    students.insert({"002", "Bob"});
    for(auto it = students.begin(); it != students.end(); ++it) {
        cout << it->id << " ";
    }
    return 0;
}
```

STL Associative Containers – map<T>

- Là container lưu trữ các cặp key-value
  - Khai báo:

```
map<T1, T2> m;
```

  - T1 là kiểu khóa (*key field type*)
    - Các key là duy nhất và được sắp xếp
    - Mặc định là toán tử `operator<`
    - Có thể dùng cho lớp do người dùng tự định nghĩa, nhưng phải định nghĩa toán tử `operator<` phù hợp cho lớp, hoặc cung cấp *functor* khi khởi tạo set.
  - `T2` là kiểu giá trị (*value field type*); có thể là kiểu bất kỳ sao cho có thể copy hoặc gán được.

```
// Các cách khai báo
map<string, int> ages;  // Map key string, value int
map<int, string> names; // Map key int, value string
map<string, vector<int>> scores; // Map phức tạp
```


map<T> - Các phương thức chính

```
// Chèn phần tử
ages["Alice"] = 25;    // Chèn hoặc cập nhật
ages.insert({"Bob", 30});

// Truy xuất
int aliceAge = ages["Alice"];
int bobAge = ages.at("Bob");   // Kiểm tra tồn tại

// Kiểm tra
bool exists = ages.count("Alice") > 0;
auto it = ages.find("Alice");

// Xóa
ages.erase("Alice");
```

map<T> - Duyệt map

```
// Duyệt theo từng cặp key-value
for (const auto& [name, age] : ages) {
    cout << name << ": " << age << endl;
}

// Duyệt bằng iterator
for (auto it = ages.begin(); it != ages.end(); ++it) {
    cout << it->first << ": " << it->second << endl;
}
```


map<T> - Tùy chỉnh so sánh key

```
// Sử dụng bộ so sánh tùy chỉnh
struct CompareLength {
    bool operator()(const string& a, const string& b) const {
        return a.length() < b.length();
    }
};
map<string, int, CompareLength> specialMap;
```

map<T> - Các thao tác hữu ích khác

```
ages.size();          // Số lượng phần tử
ages.empty();         // Kiểm tra rỗng
ages.clear();         // Xóa toàn bộ
```


map<T> - Ví dụ

```
#include <iostream>
#include <map>
#include <vector>

using namespace std;

int main() {
     map<string, vector<int>> studentScores;

     // Thêm điểm cho sinh viên
     studentScores["Alice"] = {85, 90, 92};
     studentScores["Bob"].push_back(88);

     // In ra
     for (const auto& [name, scores] : studentScores) {
         cout << name << ": ";
         for (int score : scores) {
             cout << score << " ";
         }
         cout << endl;
     }

     return 0;
}
```

STL Associative Containers

- `multiset`
  - Cho phép các phần tử trùng lặp
  - Các phần tử được tự động sắp xếp
  - Không giới hạn số lần xuất hiện của một phần tử
  - Thao tác thêm, xóa, tìm kiếm: O(log n)

```
multiset<int> numbers;
numbers.insert(5);
numbers.insert(5);  // Cho phép trùng lặp
numbers.insert(3);
// Kết quả: {3, 5, 5}

// Đếm số lần xuất hiện
int count = numbers.count(5);  // Trả về 2
```

- `multimap`
  - Cho phép nhiều key trùng lặp
  - Các key được tự động sắp xếp
  - Mỗi key có thể ánh xạ đến nhiều giá trị
  - Thao tác thêm, xóa, tìm kiếm: O(log n)

```
multimap<string, int> scores;
scores.insert({"Alice", 85});
scores.insert({"Alice", 90});  // Cho phép

     key trùng lặp
scores.insert({"Bob", 88});

// Truy xuất các giá trị của một key
auto range = scores.equal_range("Alice");
for (auto it = range.first; it !=
range.second; ++it) {
    cout << it->second << " ";  // In ra
các điểm của Alice
}
```


STL Unsorted Associative Containers (C++11)

- `unordered_[multi]set` và `unordered_[multi]map`
- Tương tự như các cấu trúc container liên kết có thứ tự, ngoại trừ:
  - Các phần tử không được sắp xếp.
  - Được cài đặt bằng bảng băm (hash tables), độ phức tạp O(1) với các thao tác insert / lookup / remove.
  - Có iterators để duyệt các phần tử trong container nhưng không theo thứ tự *đặc biệt* nào cả

pair

```
template <class T1, class T2> struct pair;
```

- pair chứa hai phần tử có thể thuộc hai kiểu khác nhau
- Truy cập phần tử thứ nhất bằng first
- Truy cập phần tử thứ hai bằng second

```
pair<int, char> PAIR1;
PAIR1.first = 100;
PAIR1.second = 'G';
cout << PAIR1.first << " ";
cout << PAIR1.second << endl;
```


make_pair

```
template <class T1, class T2> pair<T1,T2>
make_pair(T1 x, T2 y);
```

- Tạo ra một đối tượng pair

```
pair <int,int> foo;
pair <int,int> bar;

foo = make_pair (10,20);
bar = make_pair (10.5,'A'); // ok: implicit conversion from
                            // pair<double,char>

cout << "foo: " << foo.first << ", " << foo.second << '\n';
cout << "bar: " << bar.first << ", " << bar.second << '\n';
```

make_pair

- Khởi tạo biến pair

```
pair <int,int> foo = {10, 20};
pair <int,int> bar = {10.5, 'A'};

cout << "foo: " << foo.first << ", " << foo.second << '\n';
cout << "bar: " << bar.first << ", " << bar.second << '\n';
```


Ví dụ: make_pair

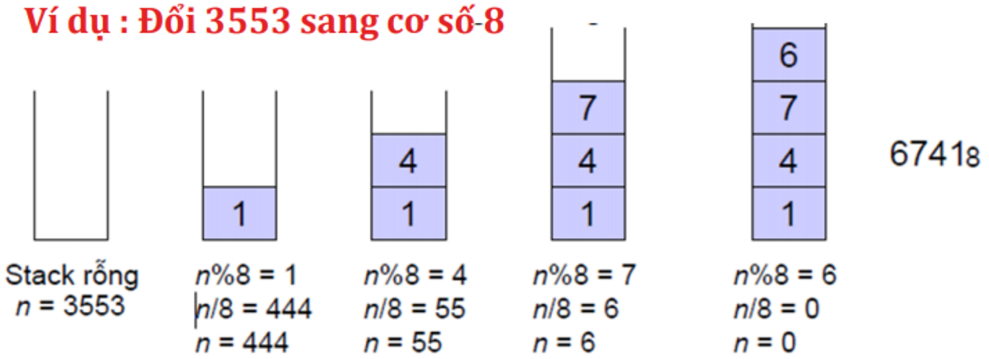
```
#include <iostream>
#include <map>
#include <string>
using namespace std;
int main() {
    multimap<string, string> mdict;
    mdict.insert(make_pair("car", "voiture"));
    mdict.insert(make_pair("car", "auto"));
    mdict.insert(make_pair("car", "wagon"));
    mdict.insert(make_pair("hello", "bonjour"));
    mdict.insert(make_pair("apple", "pomme"));
    cout << "\nPrinting all defs of \"car\"" << endl;
    for(multimap<string, string>::const_iterator it = mdict.lower_bound("car"); it !=
    mdict.upper_bound("car"); it++){
        cout << (*it).first << ": " << ' ' << (*it).second << endl;
    }
    return 0;
}
```

Một số ví dụ áp dụng

Đổi cơ số

- Dữ liệu vào: số thập phân n
- Kết quả: số hệ cơ số b tương đương
- Thuật toán:
  1. Lưu phần dư của n cho b (n % b) vào Stack.
  2. Thay n= n / b (để tìm các số tiếp theo).
  3. Lặp lại bước 1-2 cho đến khi n = 0.
  4. Rút lần lượt các chữ số lưu trong Stack, chuyển sang dạng ký tự tương ứng với hệ cơ số trước khi in ra kết quả

Ví dụ : Đổi 3553 sang cơ số-8



Đổi cơ số

```
#include <iostream>
#include <stack>
using namespace std;
void base_change(int n, int b) {
    char *digits = "0123456789ABCDEF";
    stack<int> stk;
    do {
        stk.push(n % b);
        n /= b;
    } while (n);

    while(!stk.empty()){
        int u = stk.top();
        stk.pop();
        cout << digits[u];
    }
}
int main() {
   int n=3553, b=8;
   base_change(n,b);
   return 0;
}
```

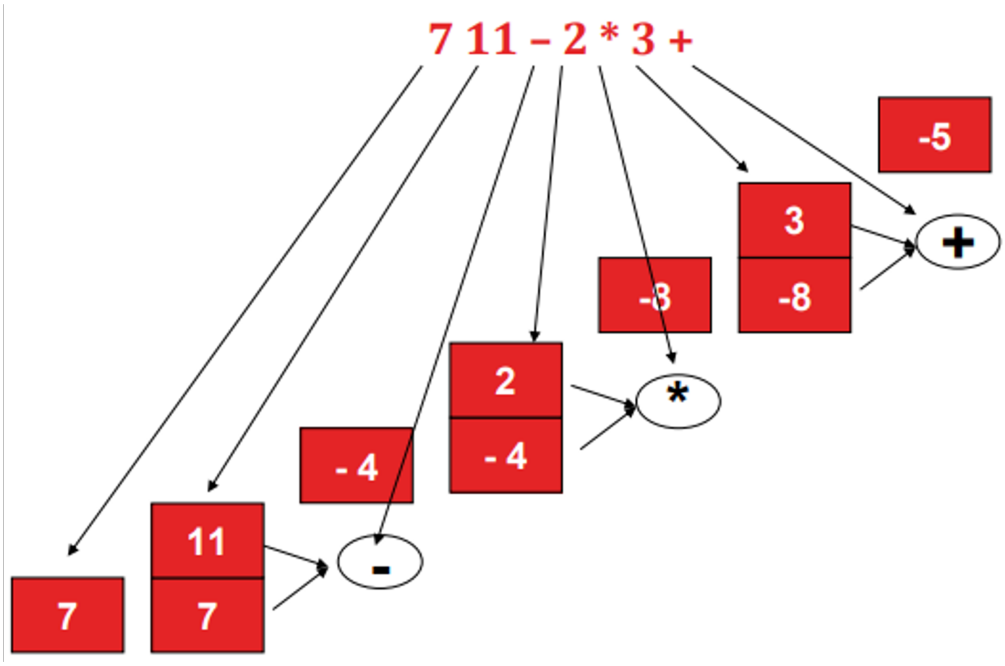

Tính toán biểu thức ký pháp hậu tố

- Ký pháp hậu tố:
  - Toán hạng đặt trước toán tử
  - Không cần dùng các dấu ().
- Ví dụ:

```
a*b*c*d*e*f => ab*c*d*e*f*
1 + (-5) / (6 * (7+8))
=> 1 5 -6 7 8 + * / +
(x/y-a*b) * ((b+x) -y )
=> x y / a b * -b x + y y ^ -*
(x*y*z -x^2 / (y*2 -z^3) + 1/z) * (x -y)
=> xy*z*x2^y2*z3^ -/ -1z/+xy -*
```

Tính toán biểu thức ký pháp hậu tố

- Ví dụ: (7 – 11) * 2 + 3



Tính toán biểu thức ký pháp hậu tố

```
bool isOperator(char op) {
    return op == '+' || op == '-' ||
           op == '*' || op == '%' ||
           op == '/' || op == '^' ;
}

int compute(int left, int right, char op) {
    int value;
    switch(op){
        case '+' : value = left + right; break;
        case '-' : value = left - right; break;
        case '*' : value = left * right; break;
        case '%' : value = left % right; break;
        case '/' : value = left / right; break;
        case '^' : value = pow(left, right);
    }
    return value;
}
```

Tính toán biểu thức ký pháp hậu tố

```
int expr_eval(string expr) {
    int left, right, ans;
    char ch;
    stack<int> stk;
    for(int i = 0; i < expr.length(); i++){
        ch = expr[i];
        if (isdigit(ch)) stk.push(ch-'0');
        else if (isOperator(ch)) {
            left = stk.top();
            stk.pop();
            right = stk.top();
            stk.pop();
            ans =compute(left, right, ch);
            stk.push(ans);
        } else cout << "Expression is not correct" << endl;
    }
    return stk.top();
}
```


Water jug

- Có hai bình, một bình dung tích a lít và một bình dung tích b lít (a, b là các số nguyên dương nhỏ hơn 1000). Có một máy bơm với lượng nước không giới hạn. Cho một số nguyên dương c, làm thế nào để đong được chính xác c lít nước với số bước nhỏ nhất.
- Gọi state (x, y): lượng nước trong hai bình
- Từ trạng thái (x, y) có thể chuyển sang các trạng thái sau:
  (x, 0)
  (0, y)
  (a, y)
  (x, b)
  (a, x + y - a) nếu x + y >= a
  (x + y, 0) nếu x + y < a
  (x + y - b, b) nếu x + y >= b
  (0, x + y) nếu x + y < b
- Trạng thái cần đạt: (c, y) hoặc (x; c)

```
#include <bits/stdc++.h>
using namespace std;
#define ii pair<int, int>

int cnt[1000][1000] = {};
queue<ii> q;
int a, b, c;

void push_and_count(int x, int y, int cur_steps){
    if((x < 0) || (y < 0) || (x > a) || (y > b)) return;
    if(!cnt[x][y]){
        q.push({x, y}); cnt[x][y] = ++cur_steps;
    }
}

void water_jug();

int main(){
    a = 6; b = 8; c = 4;
    water_jug();
    return 0;
}
```



```
void water_jug(){
    int x, y;
    q.push({a, b});
    cnt[a][b] = 1;
    while(!q.empty()){
        ii u = q.front();
        q.pop();
        x = u.first; y = u.second;
        if (x == c || y == c) {
            cout << cnt[x][y] - 1;
            return;
        }
        push_and_count(x, 0, cnt[x][y]);
        push_and_count(0, y, cnt[x][y]);
        push_and_count(x, b, cnt[x][y]);
        push_and_count(a, y, cnt[x][y]);
        push_and_count(a, x + y - a, cnt[x][y]);
        push_and_count(x + y, 0, cnt[x][y]);
        push_and_count(x + y - b, b, cnt[x][y]);
        push_and_count(0, x + y, cnt[x][y]);
    }
}
```

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Xin cảm ơn!

soict.hust.edu.vn/ fb.com/groups/soict