

Thư viện chuẩn C++ Standard Template Library (STL)

Thư viện khuôn mẫu chuẩn - STL

- Thư viện chuẩn C++ bao gồm 32 header file

<algorithm>

<bitset>

<complex>

<deque>

<exception>

<fstream>

<functional>

<iomanip>

<ios>

<iosfwd>

<iostream>

<istream>

<iterator>

<limits>

<list>

<locale>

<map>

<memory>

<new>

<numeric>

<ostream>

<queue>

<set>

<sstream>

<stack>

<stdexcept>

<streambuf>

<string>

<typeinfo>

<utility>

<valarray>

<vector>

Thư viện khuôn mẫu chuẩn - STL

- Thư viện chuẩn C++ gồm 2 phần:
 - Lớp **string**
 - Thư viện khuôn mẫu chuẩn – STL
- Ngoại trừ lớp string, ***tất cả các thành phần còn lại của thư viện*** đều là các khuôn mẫu
- Tác giả đầu tiên của STL là Alexander Stepanov, mục đích của ông là xây dựng một cách thể hiện tư tưởng lập trình tổng quát

Thư viện khuôn mẫu chuẩn - STL

- Các khái niệm trong STL được phát triển ***độc lập với C++***
 - Do đó, ban đầu, STL không phải là một thư viện C++, mà nó đã được chuyển đổi thành thư viện C++
 - Nhiều tư tưởng dẫn đến sự phát triển của STL đã được cài đặt phần nào trong Scheme, Ada, và C

Thư viện khuôn mẫu chuẩn - STL

- Một số lời khuyên về STL
 - STL được thiết kế đẹp và hiệu quả - không có thừa kế hay hàm ảo trong bất kỳ định nghĩa nào
 - Từ tư tưởng lập trình tổng quát dẫn tới những "khối cơ bản" (building block) mà có thể kết hợp với nhau theo đủ kiểu
 - Tuy làm quen với STL tốn không ít thời gian nhưng thành quả tiềm tàng về năng suất rất xứng đáng với thời gian đầu tư

Giới thiệu STL

- Ba thành phần chính của STL
 - Các thành phần rất mạnh xây dựng dựa trên template
 - Container: các cấu trúc dữ liệu template
 - Iterator: giống con trỏ, dùng để truy nhập các phần tử dữ liệu của các container
 - Algorithm: các thuật toán để thao tác dữ liệu, tìm kiếm, sắp xếp, v.v..

Giới thiệu về các Container

- 3 loại container
 - Sequence container – container chuỗi
 - các cấu trúc dữ liệu tuyến tính (vector, danh sách liên kết)
 - first-class container
 - **vector, deque, list**
 - Associative container – container liên kết
 - các cấu trúc phi tuyến, có thể tìm phần tử nhanh chóng
 - first-class container
 - các cặp khóa/giá trị
 - **set, multiset, map, multimap**
 - Container adapter – các bộ tương thích container
 - **stack, queue, priority_queue**

Các hàm thành viên STL

- Các hàm thành viên mọi container đều có
 - Default constructor, copy constructor, destructor
 - empty
 - max_size, size
 - = < <= > >= == !=
 - swap
- Các hàm thành viên của first-class container
 - begin, end
 - rbegin, rend
 - erase, clear

Giới thiệu về Iterator

- Iterator tương tự như con trỏ
 - trỏ tới các phần tử trong một container
 - các toán tử iterator cho mọi container
 - * truy nhập phần tử được trỏ tới
 - ++ trỏ tới phần tử tiếp theo
 - **begin()** trả về iterator trỏ tới phần tử đầu tiên
 - **end()** trả về iterator trỏ tới phần tử đặc biệt chặn cuối container

Các loại Iterator

- Input (ví dụ: **istream_iterator**)
 - Đọc các phần tử từ một container, hỗ trợ ++, += (chỉ tiến)
- Output (ví dụ: **ostream_iterator**)
 - Ghi các phần tử vào container, hỗ trợ ++, += (chỉ tiến)
- Forward (ví dụ: **hash_set<T> iterator**)
 - Kết hợp input iterator và output iterator
 - Multi-pass (có thể duyệt chuỗi nhiều lần)
- Bidirectional (Ví dụ: **list<T> iterator**)
 - Như forward iterator, nhưng có thể lùi (--, -=)
- Randomaccess (Ví dụ: **vector<T> iterator**)
 - Như bidirectional, nhưng còn có thể nhảy tới phần tử tùy ý

Các loại Iterator được hỗ trợ

- Sequence container
 - **vector**: random access
 - **deque**: random access
 - **list**: bidirectional
- Associative container
(hỗ trợ các loại bidirectional)
 - **set, multiset, map, multimap**
- Container adapter (không hỗ trợ iterator)
 - **stack, queue, priority_queue**

Các phép toán đối với Iterator

- Input iterator
 - `++` , `*p` , `->` , `==` , `!=`
- Output iterator
 - `++` , `*p=` , `p = p1`
- Forward iterator
 - Kết hợp các toán tử của input và output iterator
- Bidirectional iterator
 - các toán tử cho forward, và `--`
- Random iterator
 - các toán tử cho bidirectional, và
`+` , `+=` , `-` , `-=` , `>` , `>=` , `<` , `<=` , `[]`

Giới thiệu các thuật toán – Algorithm

- STL có các thuật toán được sử dụng tổng quát cho nhiều loại container
 - thao tác gián tiếp với các phần tử qua các iterator
 - thường dùng cho các phần tử trong một chuỗi
 - chuỗi xác định bởi một cặp iterator trở tới phần tử đầu tiên và cuối cùng của chuỗi
 - các thuật toán thường trả về iterator
 - ví dụ: **find()** trả về iterator trở tới phần tử cần tìm hoặc trả về **end()** nếu không tìm thấy
 - sử dụng các thuật toán được cung cấp giúp lập trình viên tiết kiệm thời gian và công sức

Sequence Container

- 3 loại sequence container:
 - **vector** – dựa theo mảng
 - **deque** – dựa theo mảng
 - **list** – danh sách liên kết hiệu quả cao

vector Sequence Container

- **vector**
 - **<vector>**
 - cấu trúc dữ liệu với các vùng nhớ liên tiếp
 - truy nhập các phần tử bằng toán tử []
 - sử dụng khi dữ liệu cần được sắp xếp và truy nhập dễ dàng
- Cơ chế hoạt động khi hết bộ nhớ
 - cấp phát một vùng nhớ liên lục lớn hơn
 - tự sao chép ra vùng nhớ mới
 - trả lại vùng nhớ cũ
- sử dụng randomaccess iterator

vector Sequence Container

- Khai báo
 - **`std::vector <type> v;`**
 - *type là kiểu dữ liệu của phần tử dữ liệu (int, float, v.v..)*
- Iterator
 - **`std::vector<type>::iterator iterVar;`**
 - trường hợp thông thường
 - **`std::vector<type>::const_iterator iterVar;`**
 - `const_iterator` không thể sửa đổi các phần tử
 - **`std::vector<type>::reverse_iterator iterVar;`**
 - Visits elements in reverse order (end to beginning)
 - Use **`rbegin`** to get starting point
 - Use **`rend`** to get ending point

vector Sequence Container

- Các hàm thành viên của **vector**
 - **v.push_back(value)**
 - thêm phần tử vào cuối (sequence container nào cũng có hàm này).
 - **v.size()**
 - kích thước hiện tại của vector
 - **v.capacity()**
 - kích thước có thể lưu trữ trước khi phải cấp phát lại
 - khi cấp phát lại sẽ cấp phát kích thước gấp đôi
 - **vector<type> v(a, a + SIZE)**
 - tạo vector từ SIZE phần tử đầu tiên của mảng a

vector Sequence Container

- Các hàm thành viên của **vector**
 - **v.insert(*iterator*, *value*)**
 - chèn *value* vào trước vị trí của *iterator*
 - **v.insert(*iterator*, *array* , *array* + **SIZE**)**
 - chèn vào vector *SIZE* phần tử đầu tiên của mảng *array*
 - **v.erase(*iterator*)**
 - xóa phần tử khỏi container
 - **v.erase(*iter1*, *iter2*)**
 - xóa bỏ các phần tử bắt đầu từ **iter1** đến hết phần tử liền trước **iter2**

vector Sequence Container

- Các hàm thành viên của **vector**
 - **v.clear()**
 - Xóa toàn bộ container
 - **v.front(), v.back()**
 - Trả về phần tử đầu tiên và cuối cùng
 - **v[elementNumber] = value;**
 - Gán giá trị **value** cho một phần tử
 - **v.at[elementNumber] = value;**
 - Như trên, nhưng kèm theo kiểm tra chỉ số hợp lệ
 - có thể ném ngoại lệ **out_of_bounds**

vector Sequence Container

- `ostream_iterator`
 - `std::ostream_iterator< type > Name(outputStream, separator);`
 - *type*: outputs values of a certain type
 - `outputStream`: iterator output location
 - `separator`: character separating outputs
- Example
 - `std::ostream_iterator< int > output(cout, " ");`
 - `std::copy(iterator1, iterator2, output);`
 - Copies elements from `iterator1` up to (not including) `iterator2` to output, an `ostream_iterator`

```

1 // Fig. 21.14: fig21_14.cpp
2 // Demonstrating standard library vector class template.
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include <vector> // vector class-template definition
10
11 // prototype for function template printVector
12 template < class T >
13 void printVector( const std::vector< T > &integers2 );
14
15 int main()
16 {
17     const int SIZE = 6;
18     int array[ SIZE ] = { 1, 2, 3,
19
20     std::vector< int > integers;
21
22     cout << "The initial size of integers is: "
23         << integers.size()
24         << "\nThe initial capacity of integers is: "
25         << integers.capacity();
26

```

Tạo một **vector** chứa các giá trị **int**

Gọi các hàm thành viên.


```
27 // function push_back is in every sequence collection
```

```
28 integers.push_back( 2 );
```

```
29 integers.push_back( 3 );
```

```
30 integers.push_back( 4 );
```

sử dụng **push_back** để
thêm phần tử vào cuối
vector



```
31  
32 cout << "\nThe size of integers is: " << integers.size()
```

```
33     << "\nThe capacity of integers is: "
```

```
34     << integers.capacity();
```

```
35  
36 cout << "\n\nOutput array using pointer notation: ";
```

```
37  
38 for ( int *ptr = array; ptr != array + SIZE; ++ptr )
```

```
39     cout << *ptr << ' ';
```

```
40  
41 cout << "\n\nOutput vector using iterator notation: ";
```

```
42 printVector( integers );
```

```
43  
44 cout << "\n\nReversed contents of vector integers: ";
```

```
45
```

```

46  std::vector< int >::reverse_iterator reverseIterator;
47
48  for ( reverseIterator = integers.rbegin();
49        reverseIterator != integers.rend();
50        ++reverseIterator )
51      cout << *reverseIterator << ' ';
52
53  cout << endl;
54
55  return 0;
56
57 } // end main
58
59 // function template for outputting vector elements
60 template < class T >
61 void printVector( const std::vector< T > &integers2 )
62 {
63     std::vector< T >::const_iterator constIterator;
64
65     for ( constIterator = integers2.begin();
66           constIterator != integers2.end();
67           constIterator++ )
68         cout << *constIterator << ' ';
69
70 } // end function printVector

```

Duyệt ngược vector bằng một **reverse_iterator**.

Template function để duyệt vector theo chiều tiến.

```
The initial size of v is: 0  
The initial capacity of v is: 0  
The size of v is: 3  
The capacity of v is: 4
```

```
Contents of array a using pointer notation: 1 2 3 4 5 6  
Contents of vector v using iterator notation: 2 3 4  
Reversed contents of vector v: 4 3 2
```

fig21_14.cpp
output (1 of 1)