

# 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 xuất 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, 4, 5, 6 };
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 );
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
```

sử dụng **push\_back** để  
thêm phần tử vào cuối  
**vector**

```

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)**



# Container Adapter

- Container adapter
  - **stack, queue và priority\_queue**
  - Không phải first class container, cho nên
    - Không hỗ trợ iterator
    - Không cung cấp cấu trúc dữ liệu
  - Lập trình viên có thể chọn cách cài đặt (sử dụng cấu trúc dữ liệu nào)
  - đều cung cấp các hàm thành viên **push** và **pop** bên cạnh các hàm thành viên khác.

# stack Adapter

- **stack**

- Header **<stack>**
- chèn và xóa tại một đầu
- Cấu trúc Last-in, first-out (LIFO)
- Có thể chọn cài đặt bằng **vector**, **list**, hoặc **deque** (mặc định)
- Khai báo

**`stack<type, vector<type> > myStack;`**

**`stack<type, list<type> > myOtherStack;`**

**`stack<type> anotherStack; // default deque`**

- chọn cài đặt là **vector**, **list** hay **deque** không làm thay đổi hành vi, chỉ ảnh hưởng tới hiệu quả (cài bằng **deque** và **vector** là nhanh nhất)

```
1 // Fig. 21.23: fig21_23.cpp
2 // Standard library adapter stack test program.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <stack>    // stack adapter definition
9 #include <vector>   // vector class-template definition
10 #include <list>    // list class-template definition
11
12 // popElements function-template prototype
13 template< class T >
14 void popElements( T &stackRef );
15
16 int main()
17 {
18     // stack with default underlying deque
19     std::stack< int > intDequeStack;
20
21     // stack with underlying vector
22     std::stack< int, std::vector< int > > intVectorStack;
23
24     // stack with underlying list
25     std::stack< int, std::list< int > > intListStack;
26
```

fig21\_23.cpp  
(1 of 3)

Tạo stack bằng nhiều  
kiểu cài đặt.

```
27 // push the values 0-9 onto each stack
28 for ( int i = 0; i < 10; ++i ) {
29     intDequeStack.push( i );
30     intVectorStack.push( i );
31     intListStack.push( i );
32
33 } // end for
34
35 // display and remove elements from each stack
36 cout << "Popping from intDequeStack: ";
37 popElements( intDequeStack );
38 cout << "\nPopping from intVectorStack: ";
39 popElements( intVectorStack );
40 cout << "\nPopping from intListStack: ";
41 popElements( intListStack );
42
43 cout << endl;
44
45 return 0;
46
47 } // end main
48
```

sử dụng hàm thành viên **push**. 3.cpp

```
49 // pop elements from stack object to which stackRef refers
50 template< class T >
51 void popElements( T &stackRef )
52 {
53     while ( !stackRef.empty() ) {
54         cout << stackRef.top() << ' '; // view top element
55         stackRef.pop();                 // remove top element
56
57     } // end while
58
59 } // end function popElements
```

**fig21\_23.cpp**  
(3 of 3)

**fig21\_23.cpp**  
output (1 of 1)

```
Popping from intDequeStack: 9 8 7 6 5 4 3 2 1 0
Popping from intVectorStack: 9 8 7 6 5 4 3 2 1 0
Popping from intListStack: 9 8 7 6 5 4 3 2 1 0
```

# Các thuật toán

- Trước STL
  - các thư viện của các hãng khác nhau không tương thích
  - Các thuật toán được xây dựng và gắn vào trong các lớp container
- STL tách rời các container và các thuật toán
  - lợi thế:
    - dễ bổ sung các thuật toán mới
    - hiệu quả hơn, tránh các lời gọi hàm ảo
  - header **<algorithm>**

# remove, remove\_if, remove\_copy và remove\_copy\_if

- **remove**

- **remove( iter1, iter2, value);**
- Bỏ mọi phần tử có giá trị **value** trong khoảng **(iter1 - iter2)** theo cách sau:
  - Chuyển các phần tử có giá trị **value** xuống cuối
  - không thay đổi kích thước của container hoặc thực sự xóa các phần tử
- Trả về iterator tới kết thúc “mới” của container
- các phần tử sau kết thúc mới là không xác định

# remove, remove\_if, remove\_copy và remove\_copy\_if

- **remove\_copy**

- **remove\_copy(iter1, iter2, iter3, value);**

- trong khoảng **iter1-iter2**, chép các phần tử khác **value** vào **iter3 (output iterator)**

- **remove\_if**

- giống **remove**

- trả về iterator tới phần tử cuối cùng
    - bỏ các phần tử mà hàm trả về **true**

- remove\_if(iter1, iter2, function);**

- các phần tử được truyền cho **function**, hàm này trả về giá trị **bool**



# **remove, remove\_if, remove\_copy và remove\_copy\_if**

- **remove\_copy\_if**
  - giống **remove\_copy** và **remove\_if**  
**remove\_copy\_if(iter1, iter2, iter3, function);**

# Các thuật toán toán học

- **random\_shuffle(iter1, iter2)**
  - xáo trộn các phần tử trong khoảng một cách ngẫu nhiên
- **count(iter1, iter2, value)**
  - trả về số lần xuất hiện của **value** trong khoảng
- **count\_if(iter1, iter2, function)**
  - đếm số phần tử làm **function** trả về **true**
- **min\_element(iter1, iter2)**
  - trả về iterator tới phần tử nhỏ nhất
- **max\_element(iter1, iter2)**
  - trả về iterator tới phần tử lớn nhất

# Các thuật toán toán học

- **accumulate(iter1, iter2)**
  - trả về tổng các phần tử trong khoảng
- **for\_each(iter1, iter2, function)**
  - Gọi hàm **function** cho mỗi phần tử trong khoảng
  - không sửa đổi phần tử
- **transform(iter1, iter2, iter3, function)**
  - gọi **function** cho mọi phần tử trong khoảng **iter1-iter2**, kết quả ghi vào **iter3**

# Các thuật toán tìm kiếm và sắp xếp cơ bản

- **find(iter1, iter2, value)**
  - trả về iterator tới lần xuất hiện đầu tiên (trong khoảng) của **value**
- **find\_if(iter1, iter2, function)**
  - như **find**
  - trả về iterator khi **function** trả về **true**
- **sort(iter1, iter2)**
  - sắp xếp các phần tử theo thứ tự tăng dần
- **binary\_search(iter1, iter2, value)**

```
1 // Fig. 21.31: fig21_31.cpp
2 // Standard library search and sort algorithms.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <algorithm> // algorithm definitions
9 #include <vector>    // vector class-template definition
10
11 bool greater10( int value ); // prototype
12
13 int main()
14 {
15     const int SIZE = 10;
16     int a[ SIZE ] = { 10, 2, 17, 5, 16, 8, 13, 11, 20, 7 };
17
18     std::vector< int > v( a, a + SIZE );
19     std::ostream_iterator< int > output( cout, " " );
20
21     cout << "Vector v contains: ";
22     std::copy( v.begin(), v.end(), output );
23
24     // locate first occurrence of 16 in v
25     std::vector< int >::iterator location;
26     location = std::find( v.begin(), v.end(), 16 );
```

fig21\_31.cpp  
(1 of 4)

```
27
28 if ( location != v.end() )
29     cout << "\n\nFound 16 at location "
30         << ( location - v.begin() );
31 else
32     cout << "\n\n16 not found";
33
34 // locate first occurrence of 100 in v
35 location = std::find( v.begin(), v.end(), 100 );
36
37 if ( location != v.end() )
38     cout << "\n\nFound 100 at location "
39         << ( location - v.begin() );
40 else
41     cout << "\n\n100 not found";
42
43 // locate first occurrence of value greater than 10 in v
44 location = std::find_if( v.begin(), v.end(), greater10 );
45
46 if ( location != v.end() )
47     cout << "\n\nThe first value greater than 10 is "
48         << *location << "\n\nfound at location "
49         << ( location - v.begin() );
50 else
51     cout << "\n\nNo values greater than 10 were found";
52
```

fig21\_31.cpp  
(2 of 4)

```
53 // sort elements of v
54 std::sort( v.begin(), v.end() );
55
56 cout << "\n\nVector v after sort: ";
57 std::copy( v.begin(), v.end(), output );
58
59 // use binary_search to locate 13 in v
60 if ( std::binary_search( v.begin(), v.end(), 13 ) )
61     cout << "\n\n13 was found in v";
62 else
63     cout << "\n\n13 was not found in v";
64
65 // use binary_search to locate 100 in v
66 if ( std::binary_search( v.begin(), v.end(), 100 ) )
67     cout << "\n\n100 was found in v";
68 else
69     cout << "\n\n100 was not found in v";
70
71 cout << endl;
72
73 return 0;
74
75 } // end main
76
```

fig21\_31.cpp  
(3 of 4)

```
77 // determine whether argument is greater than 10
78 bool greater10( int value )
79 {
80     return value > 10;
81
82 } // end function greater10
```

**fig21\_31.cpp**  
**(4 of 4)**

Vector v contains: 10 2 17 5 16 8 13 11 20 7

Found 16 at location 4

100 not found

The first value greater than 10 is 17  
found at location 2

Vector v after sort: 2 5 7 8 10 11 13 16 17 20

13 was found in v

100 was not found in v

**fig21\_31.cpp**  
**output (1 of 1)**



# Function Object – functor

- (**<functional>**)
  - Các đối tượng có thể gọi như hàm bằng toán tử ()

STL function objects	Type
<code>divides&lt; T &gt;</code>	arithmetic
<code>equal_to&lt; T &gt;</code>	relational
<code>greater&lt; T &gt;</code>	relational
<code>greater_equal&lt; T &gt;</code>	relational
<code>less&lt; T &gt;</code>	relational
<code>less_equal&lt; T &gt;</code>	relational
<code>logical_and&lt; T &gt;</code>	logical
<code>logical_not&lt; T &gt;</code>	logical
<code>logical_or&lt; T &gt;</code>	logical
<code>minus&lt; T &gt;</code>	arithmetic
<code>modulus&lt; T &gt;</code>	arithmetic
<code>negate&lt; T &gt;</code>	arithmetic
<code>not_equal_to&lt; T &gt;</code>	relational
<code>plus&lt; T &gt;</code>	arithmetic
<code>multiplies&lt; T &gt;</code>	arithmetic

```

1 // Fig. 21.42: fig21_42.cpp
2 // Demonstrating function objects.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 #include <vector>      // vector class-template definition
9 #include <algorithm>   // copy algorithm
10 #include <numeric>     // accumulate algorithm
11 #include <functional>  // binary_function definition
12
13 // binary function adds square of its second argument and
14 // running total in its first argument, then returns sum
15 int sumSquares( int total, int value )
16 {
17     return total + value * value;
18
19 } // end function sumSquares
20

```

fig21\_42.cpp  
(1 of 4)

Tạo một hàm để dùng  
với **accumulate**.

```

21 // binary function class template defines overloaded operator()
22 // that adds square of its second argument and running total in
23 // its first argument, then returns sum
24 template< class T >
25 class SumSquaresClass : public std::binary_function< T, T, T > {
26
27 public:
28
29     // add square of value to total and return result
30     const T operator()( const T &total, const T &value )
31     {
32         return total + value * value;
33
34     } // end function operator()
35
36 }; // end class SumSquaresClass
37

```

fig21\_42.cpp

Tạo một function object  
(nó còn có thể đóng gói  
dữ liệu).  
Overload **operator()**.

```

38 int main()
39 {
40     const int SIZE = 10;
41     int array[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
42
43     std::vector< int > integers( array, array + SIZE );
44
45     std::ostream_iterator< int > output( cout, " " );
46
47     int result = 0;
48
49     cout << "vector v contains:\n";
50     std::copy( integers.begin(), integers.end(), output );
51
52     // calculate sum of squares of elements of vector integers
53     // using binary function sumSquares
54     result = std::accumulate( integers.begin(), integers.end(),
55                               0, sumSquares );
56
57     cout << "\n\nSum of squares of elements in integers using "
58           << "binary\nfunction sumSquares: " << result;
59

```

fig21\_42.cpp  
(3 of 4)

đầu tiên, **accumulate** truyền **0** và **phần tử thứ nhất** lần lượt làm các tham số. Sau đó, nó dùng kết quả trả về làm tham số thứ nhất, và lặp qua các phần tử còn lại.

```

60 // calculate sum of squares of elements of vector integers
61 // using binary-function object
62 result = std::accumulate( integers.begin(), integers.end(),
63     0, SumSquaresClass< int >() );
64
65 cout << "\n\nSum of squares of elements in integers using "
66     << "binary\nfunction object of type "
67     << "SumSquaresClass< int >: " << result << endl;
68
69 return 0;
70
71 } // end main

```

dùng accumulate với  
một function object.

fig21\_42.cpp  
(4 of 4)

fig21\_42.cpp  
output (1 of 1)

```

vector v contains:
1 2 3 4 5 6 7 8 9 10

```

```

Sum of squares of elements in integers using binary
function sumSquares: 385

```

```

Sum of squares of elements in integers using binary
function object of type SumSquaresClass< int >: 385

```