

Lecture Notes on C++ Multi-Paradigm Programming

Bachelor of Software Engineering, Spring 2014

Wan Hai

whwanhai@163.com

13512768378

Software School, Sun Yat-sen University, GZ

Standard Template Library

(STL, 标准模板库)

Review: class template Stack

```
template <typename ElementType>
```

```
//genericStack.h
```

```
class Stack {
```

```
public:
```

```
Stack();
```

```
~Stack();
```

```
void push(ElementType obj)
```

```
void pop()
```

```
ElementType getTop() const
```

```
bool isEmpty() const;
```

```
private:
```

```
struct Node // 栈结点类型
```

```
{
```

```
    ElementType element;
```

```
    Node* next;
```

```
};
```

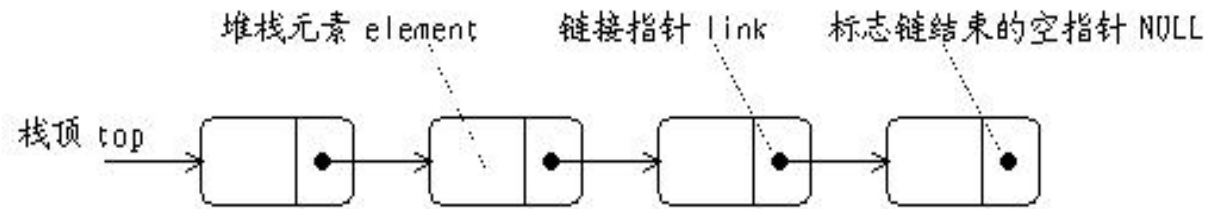
```
Node* top;
```

```
};
```

```
throw(std::bad_alloc);
```

```
throw(std::logic_error);
```

```
throw(std::logic_error);
```



```
// 结点中存放的元素
```

```
// 指向下一结点的指针
```

```
// 栈顶
```

Review: class template Stack

```

: //接上
template <typename ElementType>
void Stack<ElementType>::push( ElementType obj ) throw(std::bad_alloc)
{
    Node* temp;
    try {
        temp = new Node;
        temp -> element = obj;
        temp -> next = top;
        top = temp;
    }
    catch (std::bad_alloc e) { // 内存分配失败时进行异常处理
        throw;                // 重新抛出异常
    }
}

```

堆栈元素 element 链接指针 link 标志链结束的空指针 NULL

栈顶 top

temp

Review: class template Stack

```

:
template <typename ElementType>
void Stack<ElementType>::pop() throw(std::logic_error)
{
    Node* temp;
    if (top != NULL) {
        temp = top;
        top = top -> next;
        delete temp;
    }
    else { // 栈为空时抛出异常
        throw std::logic_error("pop from empty Stack");
    }
}

```

:

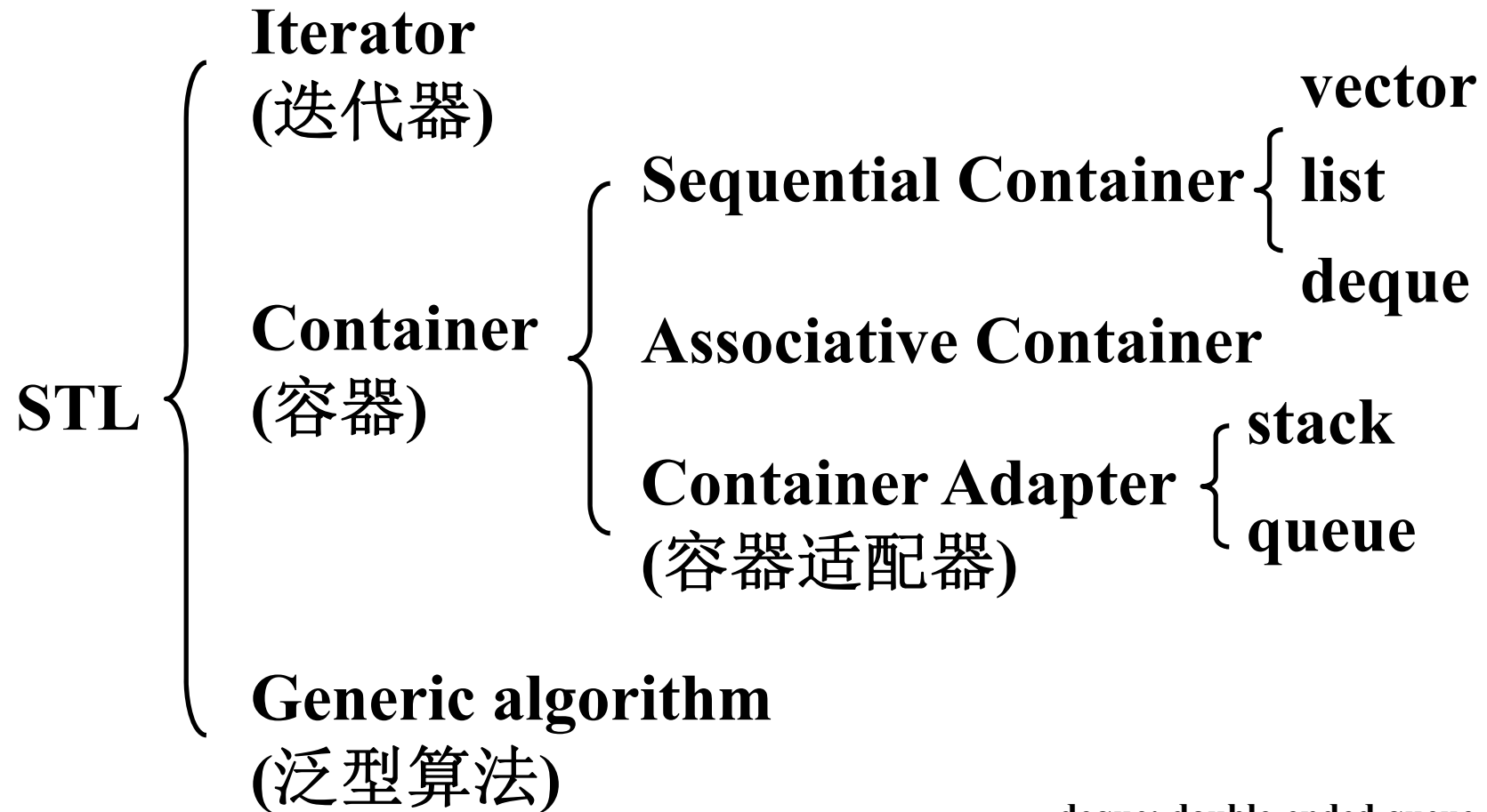
Review: class template Stack

```
#include "genericStack.h"
:
int main()
{
    Stack<int> stack; // 实例化一个保存int型元素的栈

    for (int i = 1; i < 9; i++) // 向栈中压入8个元素
        stack.push(i);

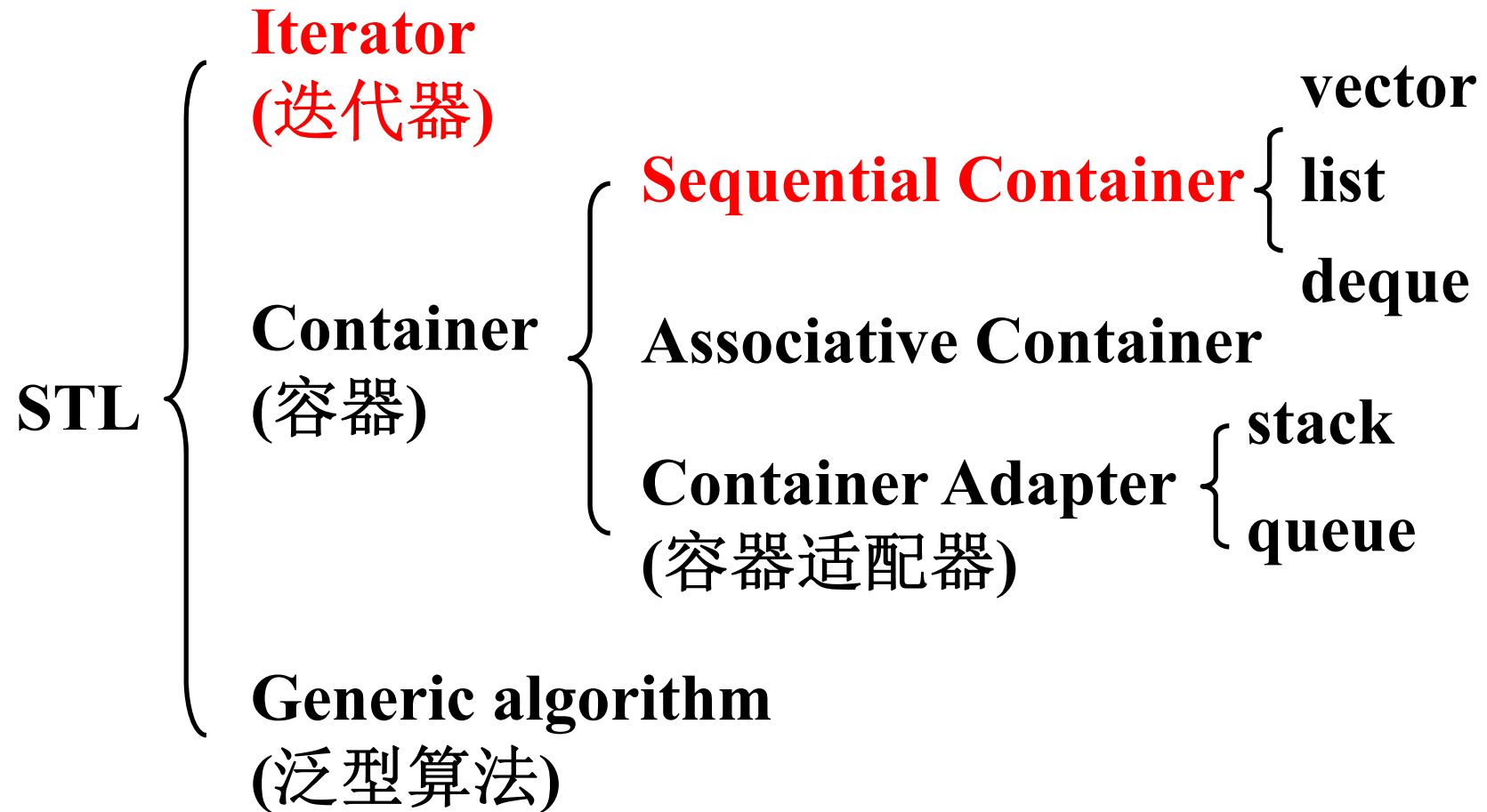
    while (!stack.isEmpty()) { // 栈不为空时循环
        cout << stack.getTop() << " "; // 显示栈顶元素
        stack.pop(); // 弹出栈顶元素
    }
}
```

STL



deque: double-ended queue
双端队列

STL



EXP1: element access in a sequential container

```
#include <vector>                                     // elementAccessDemo.cpp
using namespace std;

int main()
{
    int ia[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    vector<int> ivec(ia, ia+10);                       // ivec包含10个元素，值分别为0~9

    ivec.front() = 100;                                // 将第0个元素修改为100
    cout << "the first element: " << ivec[0] << endl; // 输出第0个元素的值

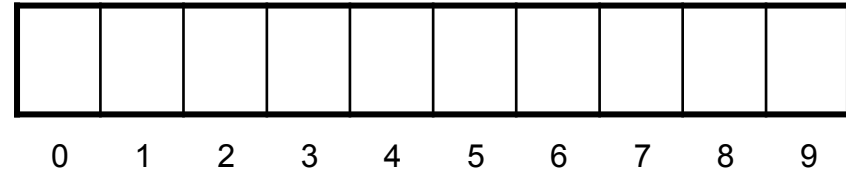
    ivec[1] = 102; // 将第1、第2个元素修改为102、103
    ivec.at(2) = 103;

    cout << "the second element: " << ivec.at(1) << endl; // 输出第1、第2个元素的值
    cout << "the third element: " << ivec[2] << endl;

    ivec.back() = 999;                                 // 将最后一个元素修改为999

    cout << "the last element: " << ivec[9] << endl; // 输出最后一个元素的值
}
```

ivec



EXP1: element access in a sequential container

ivec

100	102	103	3	4	5	6	7	8	999
-----	-----	-----	---	---	---	---	---	---	-----

```
the first element: 100
the second element: 102
the third element: 103
the last element: 999
请按任意键继续. . .
```

EXP2: element access in a sequential container

```
int main()
{
    vector<int> ivec(10, 2);
    vector<int>::iterator iter;
    vector<int>::reverse_iterator riter;

    iter = ivec.begin();
    *iter += 10;

    riter = ivec.rend();
    *(riter-1) += 10;

    iter = ivec.end();
    *(iter-1) = 100;

    riter = ivec.rbegin();
    *riter -= 20;
    :
}
```

// accessElementByIterator.cpp

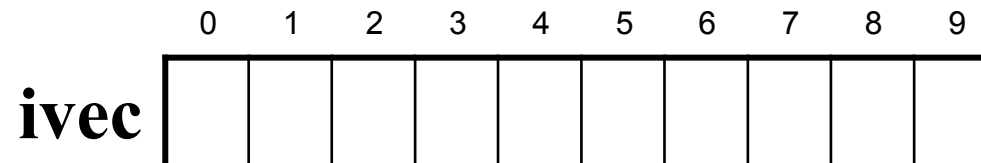
// 创建含10个值为2的元素的vector容器
// 声明迭代器对象
// 声明迭代器对象

// 获取指向第0个元素的迭代器
// 将第0个元素的值加10

// riter指向第0个元素的前一位置
// 将第0个元素的值加10

// iter指向最后一个元素的下一位置
// 将最后一个元素的值改为100

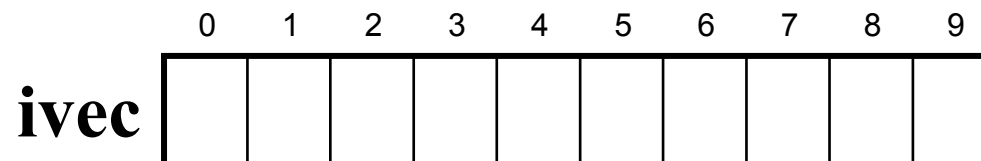
// riter指向最后一个元素
// 将最后一个元素的值减20



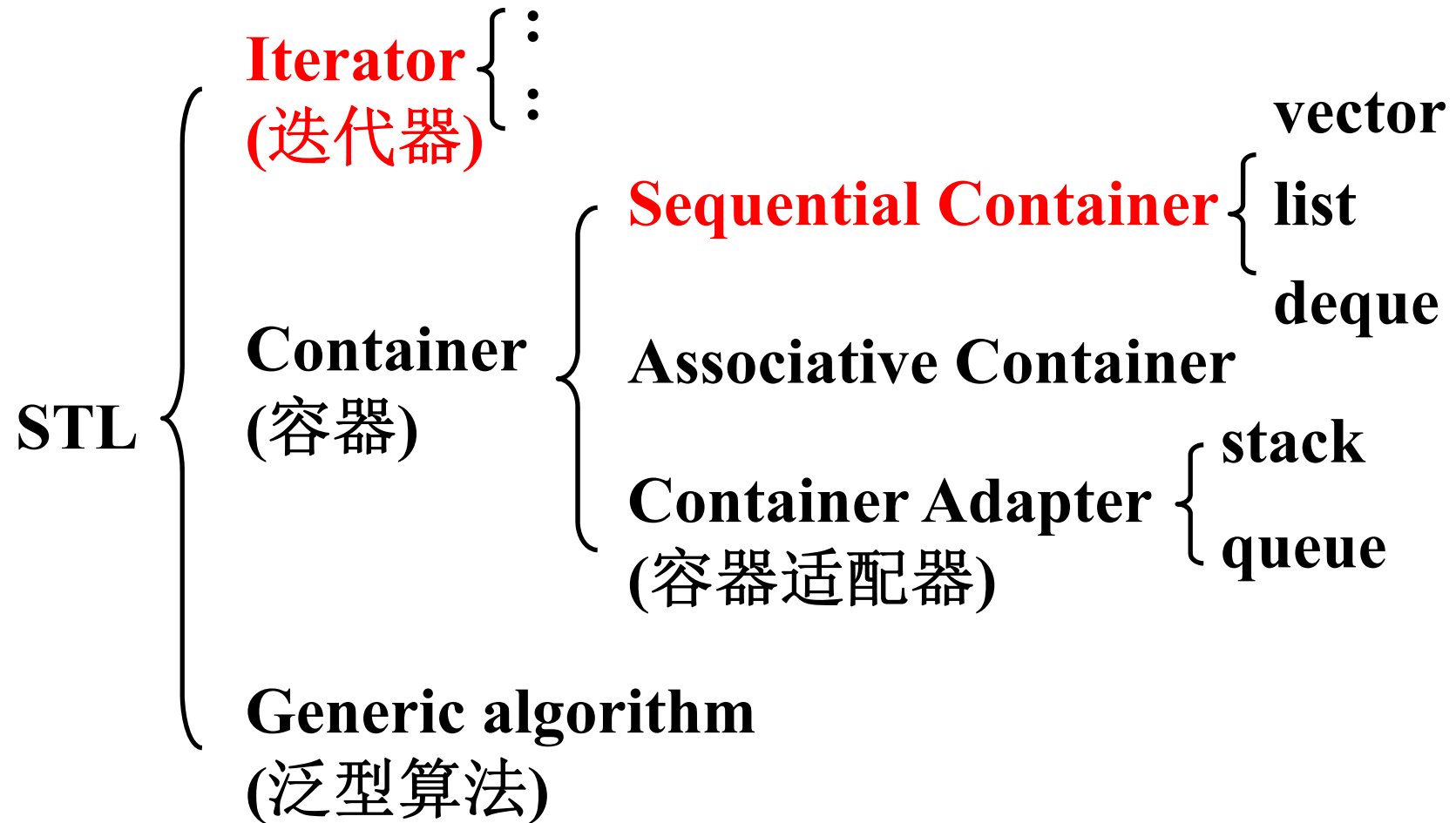
EXP1: element access in a sequential container

```
int main()                                     // accessElementByIterator.cpp
{
    vector<int> ivec(10, 2);                    // 创建含10个值为2的元素的vector容器
    vector<int>::iterator iter;                // 声明迭代器对象
    vector<int>::reverse_iterator riter;        // 声明迭代器对象
    :
    // 输出容器中的所有元素
    for( iter = ivec.begin(); iter != ivec.end(); iter ++ )
    {   cout << *iter << " ";   }

    // 反向输出容器中的所有元素
    for( riter = ivec.rbegin(); riter != ivec.rend(); riter++)
    {   cout << *riter << " ";   }
    :
}
```



STL

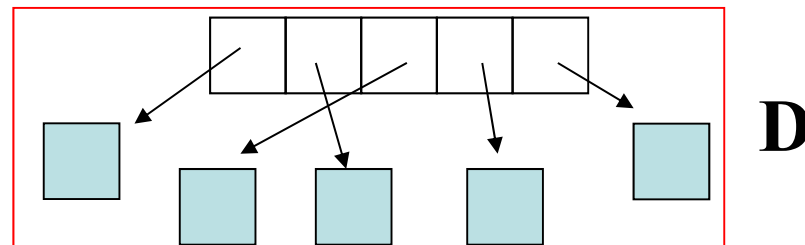
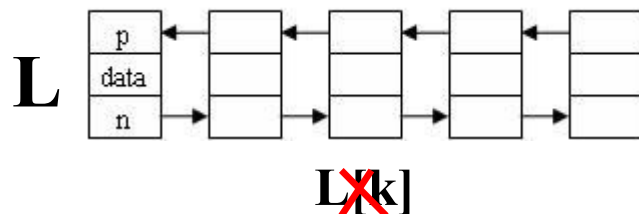


Sequential Container

V

--	--	--	--	--

template	Description (I&D=insertion&deletion)	header
vector	1. Implemented by array. 2. Fast I&D at the tail. 3. Random access supported. V[k]✓	<vector>
list	1. Implemented by double linked list. 2. Fast I&D at arbitrary position. 3. Random access NOT supported. L[k]✗	<list>
deque	1. Implemented by pointer array. 2. Fast I&D at both ends. 3. Random access supported. D[k]✓	<deque>



Sequential Container and its iterators

template	Iterator Type Names	Kinds of iterators
vector	vector <T>::iterator vector <T>::reverse_iterator vector <T>::const_iterator vector <T>::const_reverse_iterator	mutable & random access mutable & random access constant & random access constant & random access
list	list <T>::iterator list <T>::reverse_iterator list <T>::const_iterator list <T>::const_reverse_iterator	mutable & bi-directional mutable & bi-directional constant & bi-directional constant & bi-directional
deque	deque <T>::iterator deque <T>::reverse_iterator deque <T>::const_iterator deque <T>::const_reverse_iterator	mutable & random access mutable & random access constant & random access constant & random access

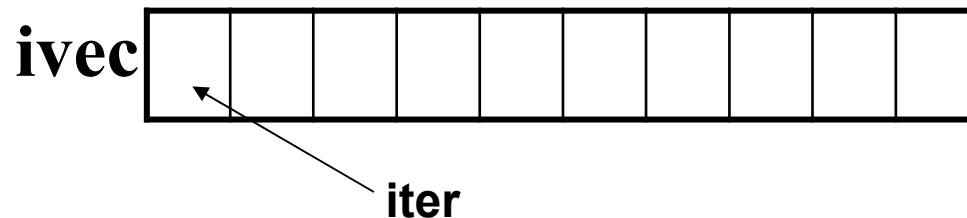
- **mutable**: the element pointed to by a mutable iterator can be modified.
- **constant**: the element pointed to by a constant iterator can NOT be modified.

Kinds of iterators

迭代器类别	功能	支持的操作	备 注
输入迭代器 (input iterator)	读	*、->、=、++、==、!=	输入迭代器的解引用 (*) 操作的结果只能作为右值 ^[1] 使用
输出迭代器 (output iterator)	写	*、++、=	输出迭代器的解引用 (*) 操作的结果只能作为左值 ^[2] 使用
正向迭代器 (forward iterator)	读/写	输入和输出迭代器支持的所有操作	正向迭代器的解引用 (*) 操作的结果既可作为左值使用也可作为右值使用

[1] 右值出现在赋值操作符的右边。

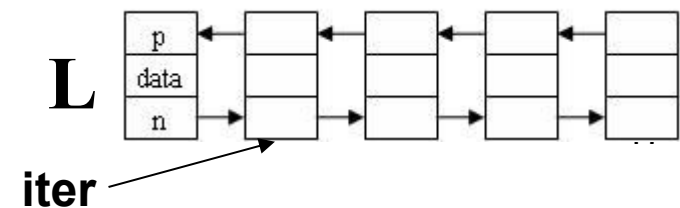
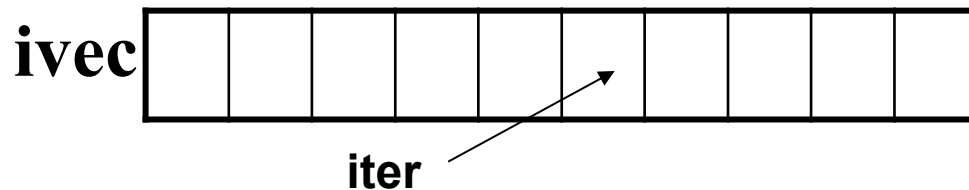
[2] 左值出现在赋值操作符的左边。



```
vector<int> ivec(10, 2);
vector<int>::iterator iter;
iter = ivec.begin();
*iter += 10;
```


Kinds of iterators

迭代器类别	功能	支持的操作	备 注
双向迭代器 (bidirectional iterator)	读/写	正向迭代器支持的所有操作以及--	所有标准库容器所提供的迭代器都至少达到双向迭代器的要求
随机访问迭代器 (random access iterator)	读/写	1.双向迭代器支持的所有操作 2.两个迭代器之间的比较操作<、<=、>、>= 3.迭代器对象与整型值n之间的+、+=、-、-= 4.两个迭代器对象相减(-) 下标操作 ([])	当两个迭代器是同一容器中的迭代器时，比较操作才有意义。下标操作 iter[n] 等价于*(iter+n)



Constructors of container

函数使用形式	说 明
<code>C<T> c;</code>	创建空容器。
<code>C<T> c(cx);</code>	创建容器c作为cx的副本，c和cx必须是同类型且元素类型也相同的容器。
<code>C<T> c(b, e);</code>	创建c，并用迭代器b和e所标示范围内的元素对c进行初始化（c中存放b和e范围内元素的副本）。
<code>C<T> c(n, t);</code>	创建c，并在其中存放n个值为t的元素，t必须是T类型的值，或者可以转换为T类型的值。
<code>C<T> c(n);</code>	创建c，并在其中存放n个元素。每个元素都是T类型的值初始化元素。

- 迭代器**b**和**e**所标示的范围是一个半开区间**[b, e)**，迭代器**e**通常用作处理的结束标记。

EXP: Constructors of container

(1) 分配指定数目的元素，并对这些元素进行值初始化：

```
vector<int> ivec1(10);           // ivec1包含10个0值元素
```

(2) 分配指定数目的元素，并将这些元素初始化为指定值：

```
vector<int> ivec2(10, 1);       // ivec2包含10个值为1的元素
```

(3) 将vector对象初始化为一段元素的副本：

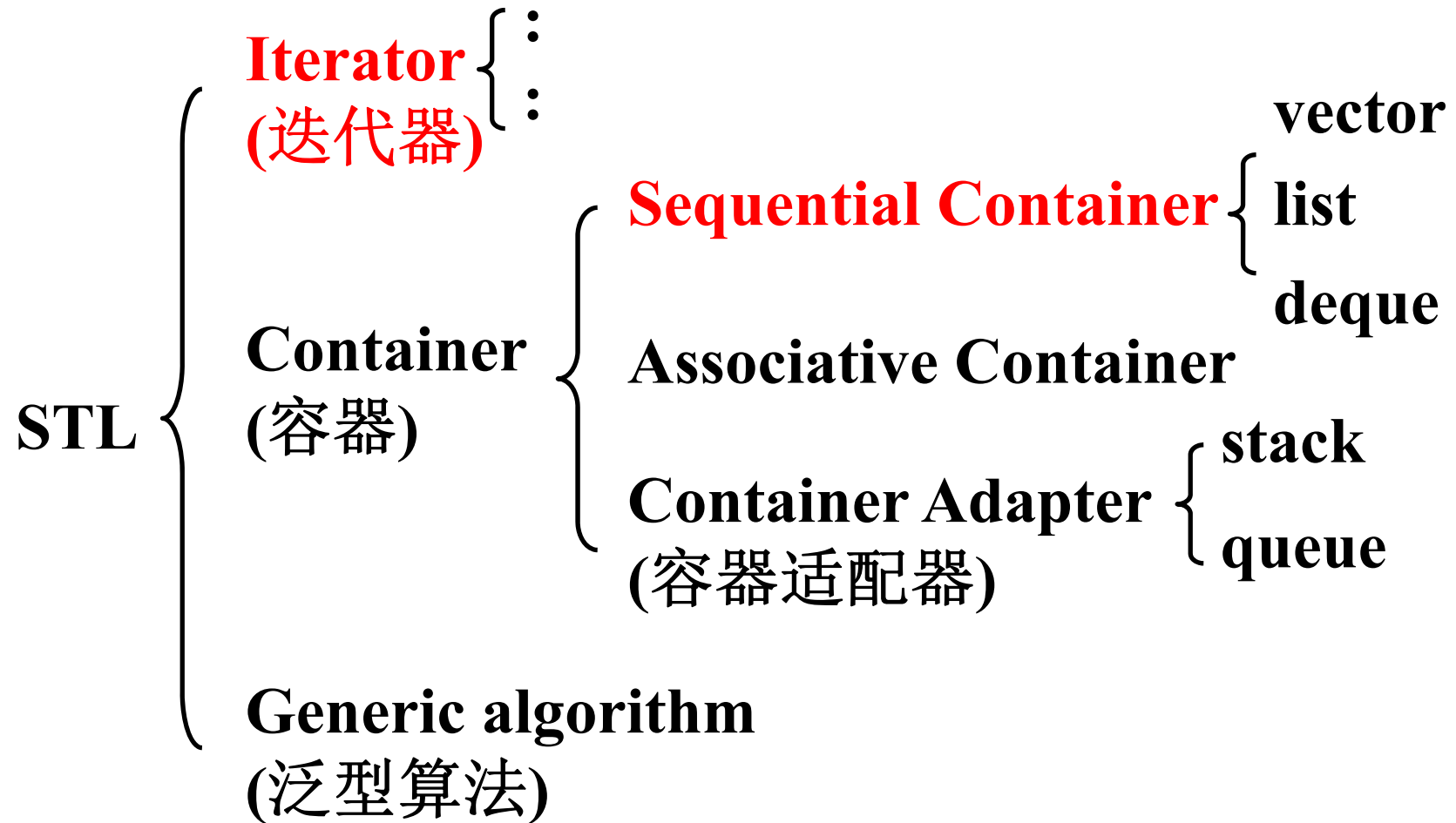
```
int ia[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

```
vector<int> ivec3(ia, ia+10); // ivec3包含10个元素，值分别为0~9
```

(4) 将一个vector对象初始化为另一个vector对象的副本：

```
vector<int> ivec4(ivec3); // ivec4包含值为0~9的元素（与ivec3相同）
```


STL



Operations of sequential containers

Iterator {
(迭代器) :

Sequential Container { **vector**
list
deque

- 
- (1)Element access
 - (2)Element insertion
 - (3)Element deletion
 - (4)Comparison between two containers
 - (5)Operation related to container capacity
 - (6)Assignment and swapping

Element access in sequential containers

- **Two ways to access an element in a sequential container:**
 - (1) using operations provided by the containers. See EXP1.**
 - (2) using iterators of the containers. See EXP2.**

EXP1: element access in a sequential container

```
#include <vector>                                     // elementAccessDemo.cpp
using namespace std;

int main()
{
    int ia[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    vector<int> ivec(ia, ia+10);                       // ivec包含10个元素，值分别为0~9

    ivec.front() = 100;                                // 将第0个元素修改为100
    cout << "the first element: " << ivec[0] << endl; // 输出第0个元素的值

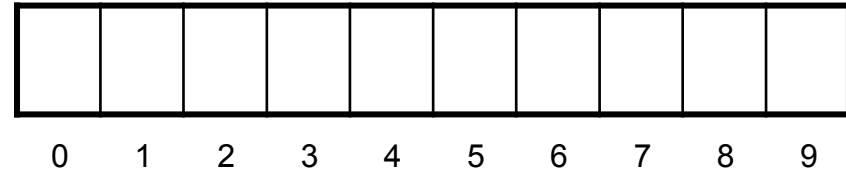
    ivec[1] = 102; // 将第二、第三个元素修改为102、103
    ivec.at(2) = 103;

    cout << "the second element: " << ivec.at(1) << endl; // 输出第1、第2个元素的值
    cout << "the third element: " << ivec[2] << endl;

    ivec.back() = 999;                                 // 将最后一个元素修改为999

    cout << "the last element: " << ivec[9] << endl; // 输出最后一个元素的值
}
```

ivec



Element access operations provided by sequential container

使用形式	形式参数	返回值	备 注
c.back()	无	容器 c 中最后一个元素的引用	若容器为空，则该操作的行为没有定义
c.front()	无	容器 c 中第 0 个元素的引用	若容器为空，则该操作的行为没有定义
c[index]	index 为元素的下标（元素在容器中序号）	返回下标为 index 的元素的引用	list 容器不提供该操作。若下标越界（即小于 0 或大于元素数-1），则该操作的行为没有定义
c.at[index]	index 为元素的下标	返回下标为 index 的元素的引用	list 容器不提供该操作。若下标越界，则该操作的行为没有定义

EXP2: element access in a sequential container

```
int main()
{
    vector<int> ivec(10, 2);
    vector<int>::iterator iter;
    vector<int>::reverse_iterator riter;

    iter = ivec.begin();
    *iter += 10;

    iter = ivec.end();
    *(iter-1) = 100;

    riter = ivec.rbegin();
    *riter -= 20;

    riter = ivec.rend();
    *(riter-1) += 10;
    :
}
```

// accessElementByIterator.cpp

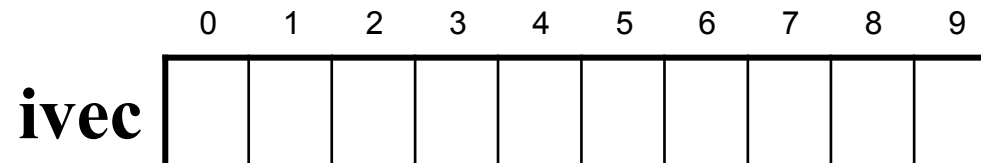
// 创建含10个值为2的元素的vector容器
// 声明迭代器对象
// 声明迭代器对象

// 获取指向第0个元素的迭代器
// 将第0个元素的值加10

// iter指向最后一个元素的下一位置
// 将最后一个元素的值改为100

// riter指向最后一个元素
// 将最后一个元素的值减20

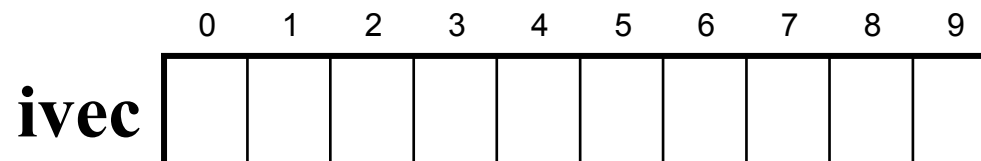
// riter指向第0个元素的前一位置
// 将第0个元素的值加10



EXP1: element access in a sequential container

```
int main()                                     // accessElementByIterator.cpp
{
    vector<int> ivec(10, 2);                    // 创建含10个值为2的元素的vector容器
    vector<int>::iterator iter;                // 声明迭代器对象
    vector<int>::reverse_iterator riter;       // 声明迭代器对象
    :
    // 正向输出容器中的所有元素
    for( iter = ivec.begin(); iter != ivec.end(); iter ++ )
    {   cout << *iter << " ";   }

    // 反向输出容器中的所有元素
    for( riter = ivec.rbegin(); riter != ivec.rend(); riter++)
    {   cout << *riter << " ";   }
    :
}
```



Operations for element access in sequential container

使用形式	返回值	备 注
<code>c.begin()</code>	迭代器。指向容器c中第0个元素	此表中的每个操作都有两个版本：一个是 <code>const</code> 成员，一个是非 <code>const</code> 成员。若容器c为 <code>const</code> 对象，则这些操作返回的迭代器的类型为带 <code>const_</code> 前缀的类型（这样的迭代器是只读迭代器，只能用于读取元素，不能用于修改元素）；否则，返回读写迭代器
<code>c.end()</code>	迭代器。指向容器c中最后一个元素的下一位置	
<code>c.rbegin()</code>	逆向迭代器。指向容器c中最后一个元素	
<code>c.rend</code>	逆向迭代器。指向容器c中第0个元素的前一位置	

Operations for element insertion in sequential container

使用形式	形式参数	返回值	操作效果
c.insert(iter, t)	iter 为迭代器， t 为元素值	指向新插入元素的迭代器	在 iter 所指元素之前插入值为 t 的元素
c.insert(iter, n, t)	iter 为迭代器， t 为元素值	无	在 iter 所指元素之前插入 n 个值为 t 的元素
c.insert(iter, b, e)	iter 、 b 和 e 均为迭代器	无	在 iter 所指元素之前插入 b 和 e 所指范围内的元素（不包括 e 所指向的元素）
c.push_back(t)	t 为元素值	无	在 c 的尾端增加值为 t 的元素
c.push_front(t)	t 为元素值	无	在 c 的头端增加值为 t 的元素

c表示容器对象。**vector**容器不提供**push_front**操作。

EXP: element insertion in a sequential container

// **addElementsDemo.cpp** 功能：演示在顺序容器中增加元素的操作

```
int main()
{
    vector<int> ivec;           // 创建空的vector容器，用于存放int型对象
    deque<string> sdeq;        // 创建空的deque容器，用于存放string型对象
    int iarr[] = {100, 100, 100};

    // 在vector容器中增加元素：在尾端增加10个元素：值为1~10
    for (int i = 1; i < 11; i++)
    {  ivec.push_back(i);  }
    :
```

	0	1	2	3	4	5	6	7	8	9
ivec	1	2	3	4	5	6	7	8	9	10

EXP: element insertion in a sequential container

```
int main()
```

```
{
```

```
    vector<int> ivec;           // 创建空的vector容器，用于存放int型对象
```

```
    deque<string> sdeq;        // 创建空的deque容器，用于存放string型对象
```

```
    int iarr[] = {100, 100, 100};
```

```
        :
```

```
    // 在vector容器头端再增加一个元素，值为20
```

```
    ivec.insert(ivec.begin(), 20);
```

20	1	2	3	4	5	6	7	8	9	10
----	---	---	---	---	---	---	---	---	---	----

```
    // 在vector容器的第四个元素后再增加两个元素，值均为30
```

```
    ivec.insert(ivec.begin() + 4, 2, 30);
```

20	1	2	3	30	30	4	5	6	7	8	9	10
----	---	---	---	----	----	---	---	---	---	---	---	----

```
    //将数组iarr中的元素增加到vector容器尾端。注意：被插入的元素不包括
    //第三个参数所指向的元素因此，要插入iarr中的所有元素，第三个参数应
    //该为iarr加3
```

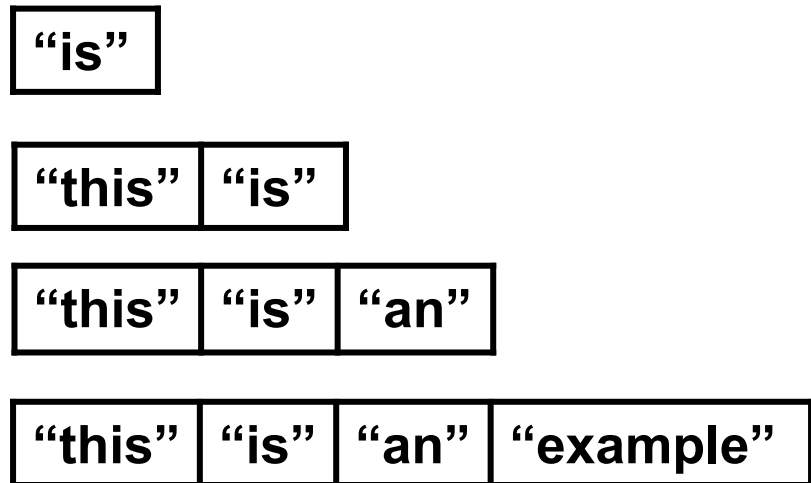
```
    ivec.insert(ivec.end(), iarr, iarr + 3);
```

20	1	2	3	30	30	4	5	6	7	8	9	10	100	100	100
----	---	---	---	----	----	---	---	---	---	---	---	----	-----	-----	-----

EXP: element insertion in a sequential container

```
int main()
{
    vector<int> ivec;      // 创建空的vector容器，用于存放int型对象
    deque<string> sdeq;    // 创建空的deque容器，用于存放string型对象
    int iarr[] = {100, 100, 100};

    :
    // 在deque容器中增加元素
    sdeq.push_back("is");
    sdeq.push_front("this");
    sdeq.insert(sdeq.end(), "an");
    sdeq.insert(sdeq.end(), "example");
    :
```



EXP: element insertion in a sequential container

```
int main()
{
    :
    // 输出vector容器中的元素
    cout << "vector:" << endl;
    for(vector<int>::iterator it = ivec.begin(); it != ivec.end(); it++)
        cout << *it << ' ';
    cout << endl;

    // 输出deque容器中的元素
    cout << "double-ended queue:" << endl;
    for(deque<string>::iterator it = sdeq.begin(); it != sdeq.end(); it++)
        cout << *it << ' ';

    return 0;
}
```

运行程序，结果如下：

vector:

20 1 2 3 30 30 4 5 6 7 8 9 10 100 100 100

double-ended queue:

this is an example

Operations for element deletion in sequential container

使用形式	形式参数	返回值	操作效果	备 注
c.clear()	无	无	删除容器中的所有元素	
c.erase(iter)	iter 为迭代器	迭代器。指向被删除元素的下一元素	删除 iter 所指向的元素	若 iter 等于 c.end() ，则该操作的行为没有定义
c.erase(b, e)	b 和 e 为迭代器	迭代器。指向被删除元素段的下一元素	删除 b 和 e 所指范围内的所有元素（不包括 e 所指向的元素）	
c.pop_back()	无	无	删除容器中最后一个元素	若容器为空，则该操作的行为没有定义
c.pop_front(t)	无	无	删除容器中的第一个元素	若容器为空，则该操作的行为没有定义。 。 vector 容器不提供该操作

EXP: element deletion in a sequential container

// elementDeleteDemo.cpp 功能： 演示顺序容器的元素删除操作

```
int main()
```

```
{
```

```
    int iarr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

1	2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	---	----

```
    deque<int> ideq(iarr, iarr+10);
```

```
    deque<int>::iterator iter;
```

// 输出删除操作之前deque**容器中的所有元素**

```
    cout << "before delete:" << endl;
```

```
    for (iter = ideq.begin(); iter != ideq.end(); iter++) {
```

```
        cout << *iter << " ";
```

```
    }
```

// 删除容器中的第一个及最后一个元素

```
    ideq.pop_front();
```

```
    ideq.pop_back();
```

2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---

:

EXP: element deletion in a sequential container

// elementDeleteDemo.cpp 功能： 演示顺序容器的元素删除操作

```
int main()
```

```
{
```

```
    int iarr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

```
    deque<int> ideq(iarr, iarr+10);
```

```
    deque<int>::iterator iter;
```

:

// 输出删除操作之后list容器中的所有元素

```
    cout << endl << "the first and last element are deleted:" << endl;
```

```
    for (iter = ideq.begin(); iter != ideq.end(); iter++) {
```

```
        cout << *iter << " ";
```

```
    }
```

2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---

```
    iter = ideq.begin();
```

// iter指向ideq中现存的第0个元素

```
    ideq.erase(ideq.erase(iter + 1)); // 删除ideq中现存的第1、第2个元素
```

2	5	6	7	8	9
---	---	---	---	---	---

:

EXP: element deletion in a sequential container

// elementDeleteDemo.cpp 功能： 演示顺序容器的元素删除操作

```
int main()
{
```

:

// 输出删除操作之后list容器中的所有元素

```
cout << endl << "the second and third element are deleted:" << endl;
```

```
for (iter = ideq.begin(); iter != ideq.end(); iter++) {
```

```
    cout << *iter << " ";
```

```
}
```

2	5	6	7	8	9
---	---	---	---	---	---

// 删除容器中现存的前三个元素

```
ideq.erase(ideq.begin(), ideq.begin() + 3);
```

7	8	9
---	---	---

:

```
}
```

EXP: element deletion in a sequential container

// elementDeleteDemo.cpp 功能： 演示顺序容器的元素删除操作

```
int main()
{
    :
    // 输出删除操作之后list容器中的所有元素
    cout << endl << "three elements at front are deleted:" << endl;
    for (iter = ideq.begin(); iter != ideq.end(); iter++) {
        cout << *iter << " ";
    }

    // 删除剩余的所有元素
    ideq.clear();
    cout << endl << "after clear:" << endl;
    if (ideq.empty()) // 容器为空
        cout << "no element in double-ended queue" << endl;

    return 0;
}
```

7	8	9
---	---	---

Container Comparison

比较操作	比较结果
<code>==</code>	若两个容器中的元素个数相同且对应位置上的每个元素都相等，则比较结果为 <code>true</code> ，否则为 <code>false</code>
<code>!=</code>	结果与 <code>==</code> 操作相反
<code><</code> , <code><=</code> , <code>></code> , <code>>=</code>	若一个容器中的所有元素与另一容器中开头一段元素对应相等，则较短的容器小于另一容器；否则，两个容器中第一对不相等元素的比较结果就是容器的比较结果

EXP: Container Comparison

// containerCompare.cpp 功能： 演示顺序容器的比较操作

```
int main()
{
    int iarr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    list<int>  ilist1(iarr, iarr+10);
    list<int>  ilist2(iarr, iarr+5);
    list<int>  ilist3(ilist2);
    list<int>  ilist4(ilist2);

    ilist4.push_back(12);
    ilist4.push_back(7);

    list<int>::iterator iter, ibegin, iend;
    string name;

    :
```

```
list1: 1 2 3 4 5 6 7 8 9 10
list2: 1 2 3 4 5
list3: 1 2 3 4 5
list4: 1 2 3 4 5 12 7
```

EXP: Container Comparison

```
int main()
{   for (int i = 1; i < 5; i++) {    // 输出4个list对象
    switch (i) {
        case 1:
            name = "list1";
            ibegin = ilist1.begin();
            iend = ilist1.end();
            break;

        case 2:
            name = "list2"; ibegin = ilist2.begin(); iend = ilist2.end(); break;
        case 3:
            name = "list3"; ibegin = ilist3.begin(); iend = ilist3.end(); break;
        case 4:
            name = "list4"; ibegin = ilist4.begin(); iend = ilist4.end(); break;
    }

    cout << name << ": ";
    for (iter = ibegin; iter != iend; iter++)
        cout << *iter << ' ';
}
```

```
list1: 1 2 3 4 5 6 7 8 9 10
list2: 1 2 3 4 5
list3: 1 2 3 4 5
list4: 1 2 3 4 5 12 7
```


EXP: Container Comparison

```
int main() // 比较list对象并输出结果
{
    cout << "ilist2 == ilist3 : ";
    if (ilist2 == ilist3)
        cout << "true" << endl;
    else
        cout << "false" << endl;

    cout << "ilist1 < ilist2 : ";
    if (ilist1 < ilist2)
        cout << "true" << endl;
    else
        cout << "false" << endl;

    cout << "ilist3 > ilist4 : ";
    if (ilist3 > ilist4)
        cout << "true" << endl;
    else
        cout << "false" << endl;
}
```

```
list1: 1 2 3 4 5 6 7 8 9 10
list2: 1 2 3 4 5
list3: 1 2 3 4 5
list4: 1 2 3 4 5 12 7
```

EXP: Container Comparison

```
int main() // 比较list对象并输出结果
{
    cout << "ilist1 < ilist4 : ";
    if (ilist1 < ilist4)
        cout << "true" << endl;
    else
        cout << "false" << endl;

    cout << "ilist2 != ilist4 : ";
    if (ilist2 != ilist4)
        cout << "true" << endl;
    else
        cout << "false" << endl;
}
```

```
list1: 1 2 3 4 5 6 7 8 9 10
list2: 1 2 3 4 5
list3: 1 2 3 4 5
list4: 1 2 3 4 5 12 7
```

Operations on capacity in sequential container

使用形式	形式参数	返回值	操作效果
c.empty()	无	若容器为空，则返回 true ；否则返回 false	
c.size()	无	返回容器中目前所存放的元素的数目，类型为 C::size_type	
c.max_size()	无	返回容器中可存放元素的最大数目，类型为 C::size_type	
c.resize(n)	n 为元素数目	无	将容器的大小调整为可存放 n 个元素。若 n < c.size() ，则删除多余元素；否则，在尾端增加相应数目的新元素，新元素均采用值初始化
c.resize(n, t)	n 为元素数目， t 为元素值	无	新增元素取值为 t ，其余效果同 c.resize(n)

Assignment and swapping of sequential containers

使用形式	形式参数	操作效果	备 注
c1 = c2	c2 为容器	首先删除 c1 中的所有元素，然后将 c2 中的元素复制给 c1 。	c1 和 c2 必须是同类型容器且其元素类型也必须相同。
c.assign(b, e)	b 和 e 为一对迭代器	首先删除 c 中的所有元素，然后将迭代器 b 和 e 所指范围内的元素复制到 c 中。	b 和 e 不能指向 c 中的元素。 b 和 e 所指范围内元素的类型不必与 c 的元素类型相同，只需类型兼容即可。
c.assign(n, t)	n 为元素数目， t 为元素值	首先删除 c 中的所有元素，然后在 c 中存放 n 个值为 t 的元素。	
c1.swap(c2)	c2 为容器	交换 c1 和 c2 的所有元素。实际上是 c1 与 c2 交换名称。	c1 和 c2 必须是同类型容器且其元素类型也必须相同。 44

Vector, List or Deque?

- **Efficiency**

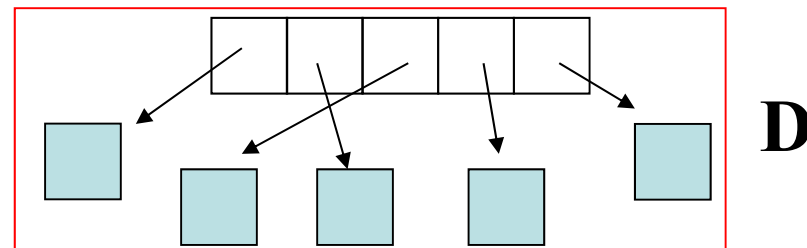
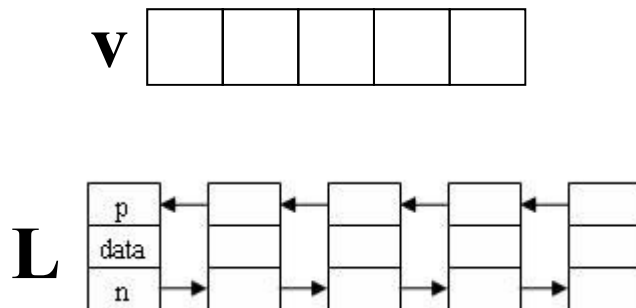
access: **vector** > deque > list

insertion or deletion: **list** > deque > vector

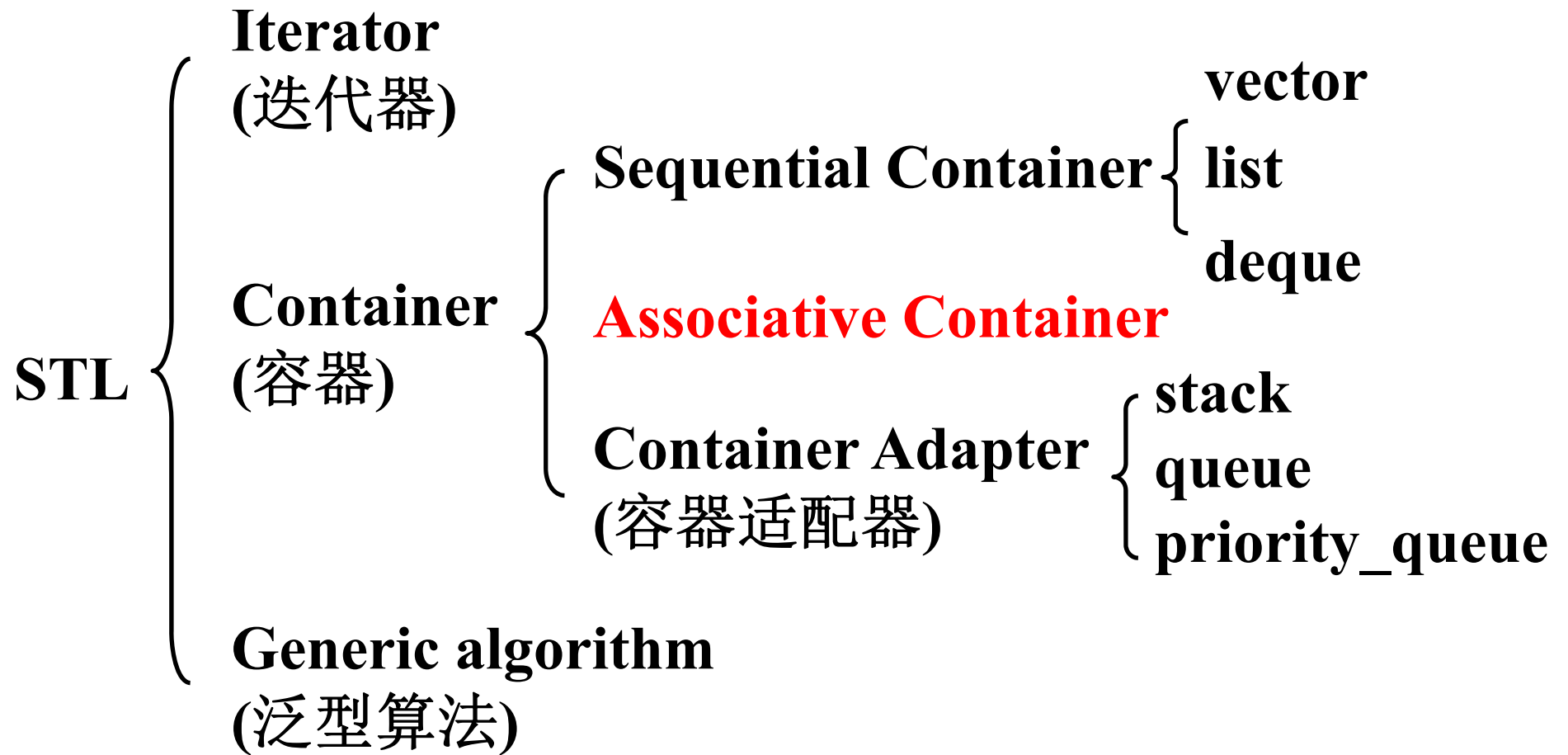
- **Capacity**

vector: **capacity** and **reserve** operations for
controlling the capacity.

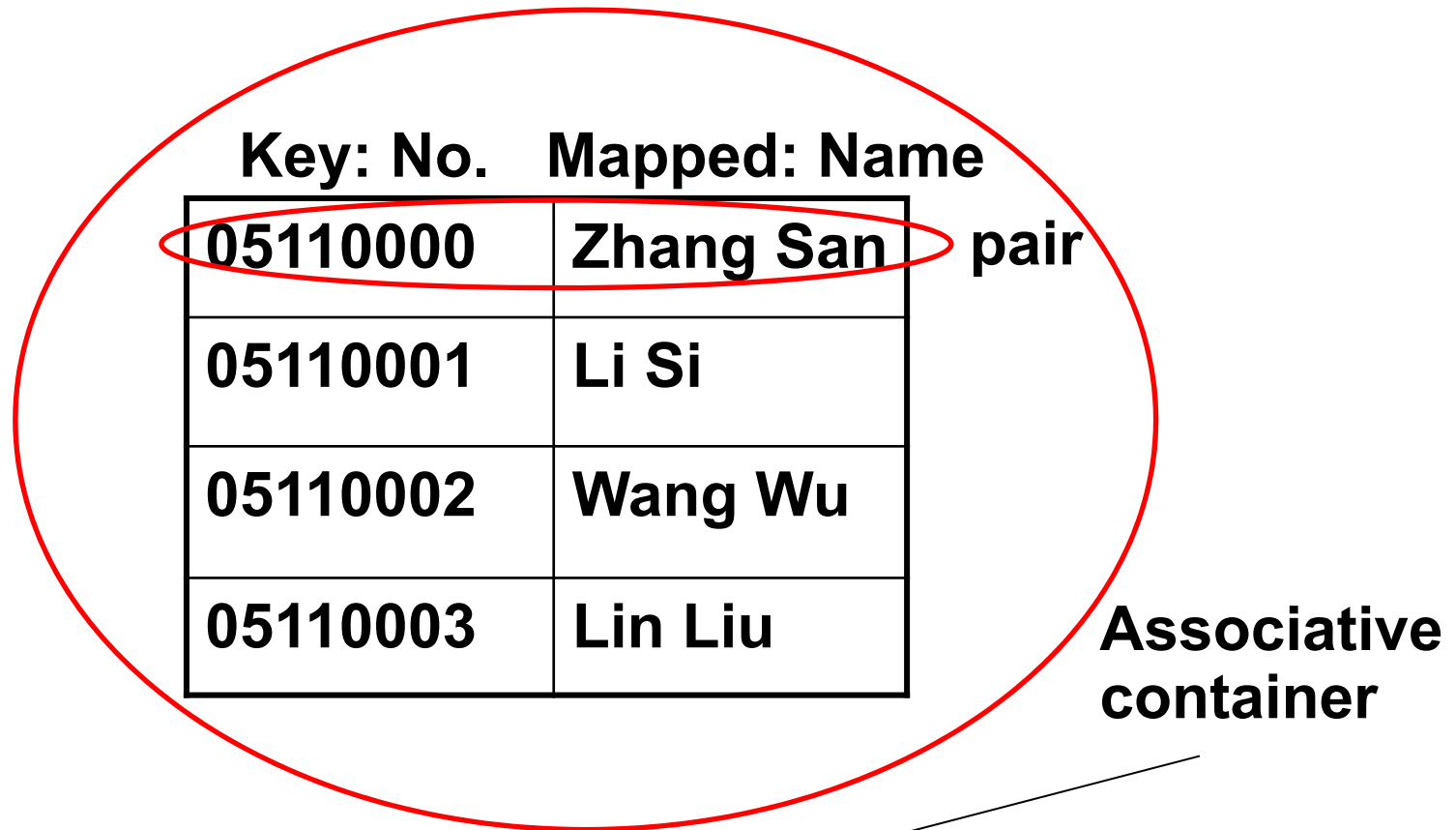
deque, list: NO capacity concepts for a list or a deque.



STL



Associative Container



map multimap set multiset

Associative Container


Key: No. Mapped: Name

05110000	Zhang San
05110001	Li Si
05110002	Wang Wu
05110001	Chen Qi
05110003	Lin Liu

map

Associative Container

Key: Name Mapped: Phone No.



Chen Qi	84111111
Li Si	84111112
Li Si	84111111
Wang wu	84111100
Zhang San	84111123

mutimap

Associative Container

{ 1, ~~2~~, 3, ~~4~~, 5, 6, 7, ~~2~~, 8, 9, ~~4~~, 10 } set

{ 1, 2, 3, 4, 5, 6, 7, 2, 8, 9, 4, 10 } multiset

Associative Container

类 名	说 明	所在头文件
map	通过键进行元素存取的相关数组	<map>
multimap	支持重复键的相关数组	<map>
set	键的集合（集合的元素就是键）	<set>
multiset	支持重复键的集合	<set>

Class template: pair

pair 头文件<utility>

Key: No. Mapped: (Name age score)

05110000	Zhang San	20	90
05110001	Li Si	19	80
05110002	Wang Wu	21	80
05110003	Lin Liu	22	100

map

How to create a pair object

使用形式	说 明
pair<T1, T2> p	创建空的 pair 对象 p ， p 的两个数据成员的类型分别为 T1 和 T2 ，均采用值初始化。
pair<T1, T2> p(v1, v2)	创建 pair 对象 p ， p 的两个数据成员的类型分别为 T1 和 T2 ，成员 first 初始化为 v1 ，成员 second 初始化为 v2 。
make_pair(v1, v2)	标准库函数。使用值 v1 和 v2 创建 pair 对象。

p.first: **T1**类型的数据，作为**key**

p.second: **T2**类型的数据，作为**mapped**

Comparison between pair objects

使用形式	说 明	备 注
p1 < p2	若 p1.first < p2.first 或 !(p2.first < p1.first) && p1.second < p2.second ，则比较结果为 true ，否则为 false	pair 对象的比较使用其元素类型提供的相应比较操作
p1 == p2	若 p1.first == p2.first && p1.second == p2.second ，则比较结果为 true ，否则为 false	

Class template: pair

pair

Key: No. Mapped: (Name age score)

05110000	Zhang San	20	90
05110001	Li Si	19	80
05110002	Wang Wu	21	80
05110003	Lin Liu	22	100

map

Constructors of map

函数使用形式	说 明
map<K, T> m;	创建空的 map 容器 m ，其键类型为 K ，值类型为 T
map< K, T> m(mx);	创建 map 容器 m 作为 mx 的副本。 m 和 mx 的键类型和值类型都必须相同
map< K, T> m(b, e);	创建 map 容器 m ，并用迭代器 b 和 e 所标示范围内的元素对 m 进行初始化（ m 中存放 b 和 e 范围内元素的副本）

Data types of a map

类型别名	说 明
map<K, T>::key_type	元素（ pair 对象）中键的类型
map<K, T>::mapped_type	元素中键所关联的值的类型
map<K, T>::value_type	元素的类型，是一个 pair 类型，其中 first 成员的类型为 const map<K, T>::key_type ， second 成员的类型为 map<K, T>::mapped_type

map类中的**value_type**是**pair**类型，它的**first**成员是**const**类型的，也就是说，**map**容器中元素的键不能修改。若需要修改容器中某元素的键，只能用间接的方式：首先删除该元素，再插入一个新元素，新元素的键设置为所需要的键。

Operations of map

- **Element access**
- **Element insertion**
- **Element deletion**
- **Searching an element**

EXP: Map

```
// *****
```

```
// phoneNumberBook.cpp
```

```
// 功能： 利用map容器类实现电话号码本
```

```
// 电话号码本中的条目按姓名排列（假设没有重名的条目）
```

```
// 程序支持用户创建并维护一个电话号码本：
```

```
// 可以往电话号码本中添加条目、可以删除指定条目、
```

```
// 可以修改指定条目中的电话号码、
```

```
// 可以指定姓名查询电话号码、可以显示电话号码本的内容。
```

```
// *****
```

Name Phone No.

ChenQi	84111111
LiMing	84111112
LiSi	84111111
WangWu	84111100
ZhangSan	84111123

phoneNumBook

```
void printMenu();
```

```
int main()
```

```
{
```

```
    map<string, string> phoneNumBook;
```

```
    string name;
```

```
    string endName;
```

```
    string phoneNumber;
```

// 电话号码本

// 姓名

// 要删除的最后一个姓名

// 电话号码

// 用于访问容器中元素的迭代器

```
    map<string, string>::iterator iter, beginIter, endIter;
```

```
    :
```

EXP: Map

```
int main()
{   int choice = 1;
    while (choice != 0) {
        printMenu(); // 显示菜单
        // 获取用户选择
        cout << "Enter your choice:";
        cin >> choice;
```

// 根据用户选择分别进行处理

```
switch (choice) {
    case 1: // 插入条目
        cout << "Enter the name you want to insert: ";
        cin >> name;
        cout << "Enter the phone number(s) : ";
        cin >> phoneNumber;
        phoneNumBook.insert( make_pair(name, phoneNumber) );
        break;
```

ChenQi	84111111
LiMing	84111112
LiSi	84111111
WangWu	84111100
ZhangSan	84111123

:

EXP: Map

```
int main()
{
    case 2:           // 删除一个条目
        cout << "Enter the name you want to delete: ";
        cin >> name;
        phoneNumBook.erase(name);
        break;

    case 3:           // 根据名字查找号码
        cout << "Enter the name you want to search: ";
        cin >> name;
        iter = phoneNumBook.find(name);
        if (iter == phoneNumBook.end())
            cout << "No such name in the phone number book." << endl;
        else
            cout << "The phone number of " << name << " is "
                << (*iter).second << endl;
        break;
}
```

ChenQi	84111111
LiMing	84111112
LiSi	84111111
WangWu	84111100
ZhangSan	84111123

EXP: Map

```
int main()
{
    case 4:      // 删除多个条目
    cout << "Enter the first name you want to delete: ";
    cin >> name;
    cout << "Enter the last name you want to delete: ";
    cin >> endName;

    beginIter = phoneNumBook.find( name );
    endIter = phoneNumBook.find( endName );

    // erase操作不删除第二个迭代器所指向的元素，所以先将迭代器向后移动一
    // 个元素
    endIter++;
    phoneNumBook.erase(beginIter, endIter);
    break;
```

ChenQi	84111111
LiMing	84111112
LiSi	84111111
WangWu	84111100
ZhangSan	84111123

:

EXP: Map

```
int main()
{
    case 5:                                // 修改指定条目中的电话号码
    cout << "Enter the item you want to modify: ";
    cin >> name;
    if (phoneNumBook.count(name) == 0) {
        cout << "No such name in the phone number book!" << endl;
        break;
    }
    cout << "Enter the new phone number:";
    cin >> phoneNumber;
    phoneNumBook[name] = phoneNumber;
    break;
    :
```

ChenQi	84111111
LiMing	84111112
LiSi	84111111
WangWu	84111100
ZhangSan	84111123

EXP: Map

```
int main()
{
    case 6:                                // 列出电话号码本的内容

    cout << "content of the phone number book: " << endl;
    for ( iter = phoneNumBook.begin();
        iter != phoneNumBook.end();
        iter++ )
        cout << (*iter).first << "\t" << (*iter).second << endl;

    break;

    case 0:                                // 退出
    break;
}
}

return 0;
}
```

ChenQi	84111111
LiMing	84111112
LiSi	84111111
WangWu	84111100
ZhangSan	84111123

EXP: Map

```
void printMenu()  
// 功能： 输出选择菜单  
{  
    cout << endl;  
    cout << "*****" << endl;  
    cout << "  1--insert          2--delete a item" << endl;  
    cout << "  3--search          4--delete some items" << endl;  
    cout << "  5--modify          6--display" << endl;  
    cout << "  0--quit" << endl;  
    cout << "*****" << endl;  
}
```

Element access in map

//假设name="Wang Wu"

phoneNumBook[name] = phoneNumber;

Key	Mapped
ChenQi	84111111
LiMing	84111112
LiSi	84111111
WangWu	84111100
ZhangSan	84111123

iter → (points to the row containing LiSi)

phoneNumBook["Wang Wu"] → (points to the value 84111111 in the Mapped column of the LiSi row)

***iter** → (points to the value 84111111 in the Mapped column of the LiSi row)

Element access in map

- 容器的元素下标可以像内置数组那样是整型，也可以是其他类型（例如，**string**类型），而且后者更常见，因为这里的下标是作为键（关键字）使用的。
- 使用下标访问元素时，如果该元素存在，则返回元素中的值（也就是键所对应的值）。如果指定的元素不存在，将会导致在容器中增加一个新元素，该元素中“键”的取值就是给定的下标值，该元素中的“值”采用值初始化。
- map**容器下标操作的返回值类型为**map**容器中定义的**mapped_type**类型，而容器的迭代器的解引用（*）操作的返回值类型则为容器中定义的**value_type**类型。

ChenQi	84111111
LiMing	84111112
LiSi	84111111
WangWu	84111100
ZhangSan	84111123

```
phoneNumBook[ name ] = phoneNumber;
```

```
map<K, T>::key_type
```

```
map<K, T>::mapped_type
```

```
map<K, T>::value_type
```

Element insertion in map

使用形式	形式参数	返回值	操作效果
m.insert(e)	e 为元素值（ 一个pair对象）	一个pair对象（ 其first成员是一个指向被插入元素的迭代器，其second成员是一个bool对象，表示是否插入了元素）	若键 e.first 不在容器 m 中，则插入元素 e ； 否则， m 保持不变
m.insert(iter, e)	iter 为迭代器，表示搜索新元素存储位置的起点 e 为元素值	一个迭代器，指向键为 e.first 的元素	在 iter 所指元素之后插入元素 e （若键 e.first 已在容器 m 中，则 m 保持不变）

Element insertion in map

使用形式	形式参数	返回值	操作效果
m.insert (begin, end)	begin 和 end 均为迭代器 ，表示要插入的元素的 范围	无	将 begin 和 end 所指范围内的元素（不包括 end 所指向的元素）插入到 m 中（若某元素的键在 m 中已存在，则不插入该元素）

- **map**容器不提供**push_back**和**push_front**操作。
- 如果欲插入的元素所对应的键已在容器中存在，则**insert**将不做任何操作。

Element searching in map

使用形式	形式参数	返回值
m.find(k)	k 为要查找的键	若容器 m 中存在与 k 对应的元素，则返回指向该元素的迭代器；否则，返回指向 m 中最后一个元素的下一位置的迭代器(即 m.end())。
m.count(k)	k 为要查找的键	k 在容器 m 中的出现次数。

Element deletion in map

使用形式	形式参数	返回值	操作效果	备 注
m.erase(k)	k 为要删除元素的键	被删除元素的个数，其类型为 map 容器中定义的 size_type	若容器 m 中存在键为 k 的元素，则删除该元素	若返回值为 0 ，表示要删除的元素不存在
m.erase(iter)	iter 为指向要删除元素的迭代器	无	删除 iter 所指向的元素	若 iter 等于 c.end() ，则该操作的行为没有定义
m.erase(b, e)	b 和 e 为迭代器，表示要删除元素的范围	无	删除 b 和 e 所指范围内的所有元素（不包括 e 所指向的元素）	要么 b 和 e 相等（此时删除范围为空，不删除任何元素），要么 b 所指向的元素出现在 e 所指向的元素之前

Multimap

- 不支持下标操作。
- **insert**操作每调用一次都会增加新的元素（**multimap**容器中，键相同的元素相邻存放）。
- 以键值为参数的**erase**操作删除该键所关联的所有元素，并返回被删除元素的数目。
- **count**操作返回指定键的出现次数。
- **find**操作返回的迭代器指向与被查找键相关联的第一个元素。
- 结合使用**count**和**find**操作依次访问**multimap**容器中与特定键关联的所有元素。

ChenQi	84111111
LiSi	84111112
LiSi	84111111
LiSi	84111234
LiSi	84110100
Wangwu	84111100
ZhangSan	84111123

Multimap

使用形式	说 明	备 注	
m.lower_bound(k)	该操作返回一个迭代器，指向容器 m 中第一个键 $\geq k$ 的元素	若键 k 在容器中不存在，则这两个操作所返回的迭代器相同，都指向 k 应该插入的位置	
m.upper_bound(k)	该操作返回一个迭代器，指向容器 m 中第一个键 $> k$ 的元素		
m.equal_range(k)	该操作返回包含一对迭代器的pair对象，其first成员等价于 m.lower_bound(k) ，其second成员则等价于 m.upper_bound(k)	ChenQi	84111111
		LiSi	84111112
		LiSi	84111111
		LiSi	84111234
		LiSi	84110100
		Wangwu	84111100
		ZhangSan	84111123

Set

- **map**支持的操作**set**基本上都支持，但有区别。如下：
 - 1) 不支持下标操作。
 - 2) 没有定义**mapped_type**类型
 - 3) **set**容器定义的**value_type**类型不是**pair**类型，而是与**key_type**相同，指的都**set**中元素的类型。

set { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }

EXP: Set

// setDemo.cpp 功能： 演示set容器类的使用

```
int main()
```

```
{
```

```
    int iarr[] = {1, 1, 2, 3, 3};
```

```
    double darr[] = {4.4, 5.6, 2.1, 7.8, 8.8, 9.8, 1.1};
```

// 用内置数组iarr的所有元素对set对象iset进行初始化: iset中将仅包

// 含3个元素: 1, 2, 3

```
    set<int> iset(iarr, iarr + 5);
```

```
    set<double> dset;
```

// 创建空的set对象

// 输出set对象iset中的元素

```
    cout << "content of the integer set container:" << endl;
```

```
    for (set<int>::iterator iter = iset.begin(); iter != iset.end(); iter++)
```

```
        cout << *iter << " ";
```

```
:
```

```
iset { 1, 2, 3, }
```

EXP: Set

```
int main()
{
    int iarr[] = {1, 1, 2, 3, 3};
    set<int> iset(iarr, iarr + 5);
    set<double> dset;

    // 在iset对象中查找特定元素
    if (iset.find(2) != iset.end())
        cout << "2 is a element in the integer set container" << endl;
    else
        cout << "2 is not a element in the integer set container" << endl;
    if (iset.find(6) != iset.end())
        cout << "6 is a element in the integer set container" << endl;
    else
        cout << "6 is not a element in the integer set container" << endl;
    :
}
```

iset { 1, 2, 3, }

EXP: Set

```
int main()
{
    double darr[] = {4.4, 5.6, 2.1, 7.8, 8.8, 9.8, 1.1};
    set<double> dset;

    dset.insert(1.2); // 向set对象dset中插入元素
    dset.insert(3.4);
    dset.insert(3.4);
    dset.insert(darr, darr + 7);

    // 输出set对象dset中的元素
    cout << "content of the double set container:" << endl;
    for (set<double>::iterator iter = dset.begin(); iter != dset.end(); iter++)
        cout << *iter << " ";
    :
```

EXP: Set

```
int main()
{
    : dset {1.2, 2.1, 3.4, 4.4, 5.6, 7.8, 8.8, 8.9 }

    dset.erase(1.1); // 删除dset对象中的元素1.1

    // 删除dset对象中大于等于3.4且小于7.8的所有元素
    dset.erase(dset.find(3.4), dset.find(7.8));

    // 输出set对象dset中的元素
    cout << "content of the double set container(after delete):" << endl;
    for (set<double>::iterator iter = dset.begin(); iter != dset.end(); iter++)
        cout << *iter << " ";

    dset {1.2, 2.1, 7.8, 8.8, 8.9 }

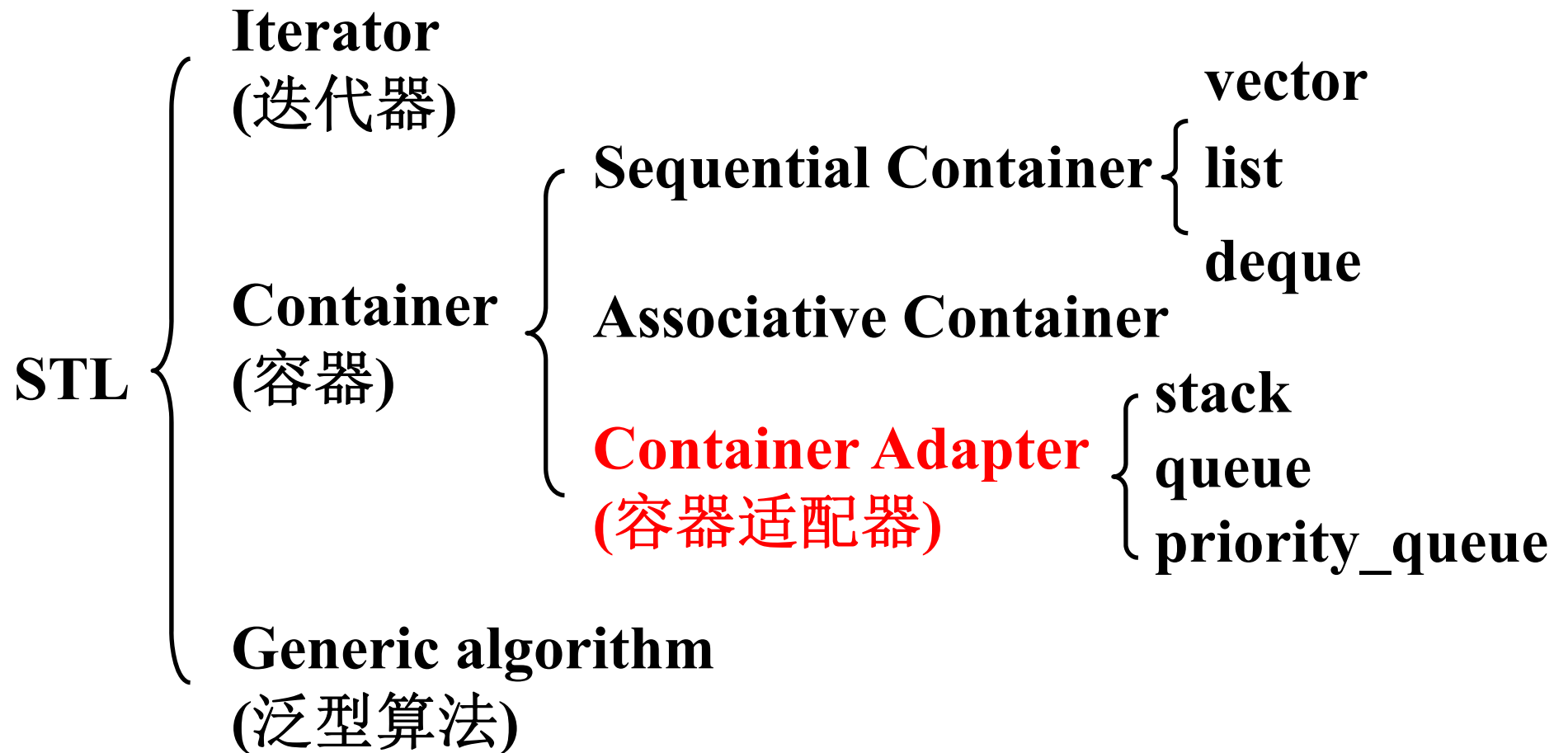
    return 0;
}
```

Set and multiset

$\{ 1, \cancel{2}, 3, \cancel{4}, 5, 6, 7, \cancel{2}, 8, 9, \cancel{4}, 10 \}$ set

$\{ 1, 2, 3, 4, 5, 6, 7, 2, 8, 9, 4, 10 \}$ multiset

STL

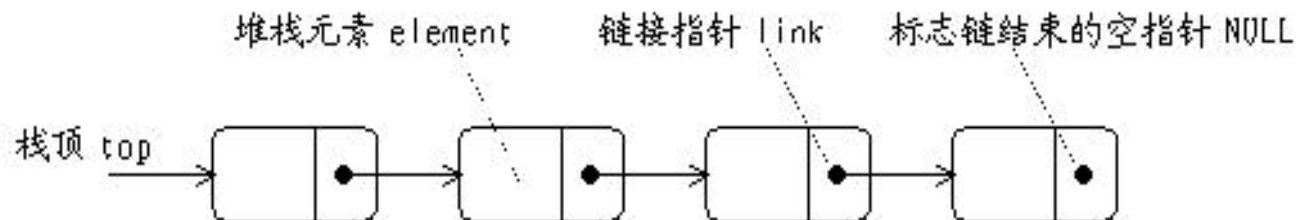


Container Adapter

类 名	说 明	所在头文件
stack	栈。元素的插入和删除只在栈顶进行，支持后进先出（ LIFO ）的元素访问	<stack>
queue	队列。在尾端插入元素，在头端删除元素，支持先进先出（ FIFO ）的元素访问	<queue>
priority_deque	带优先级管理的队列。元素按优先级从高到低排列，使用元素类型的<操作确定优先级	<queue>

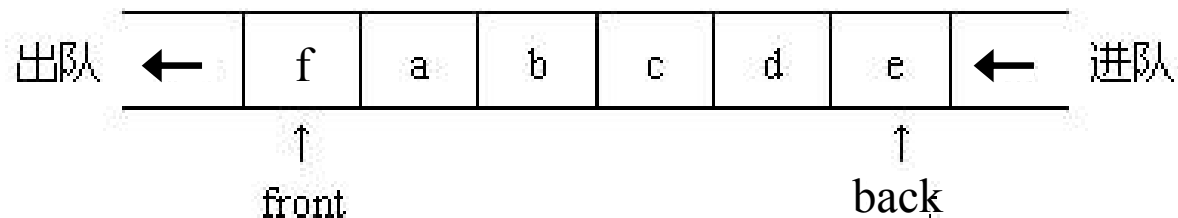
Operations of stack

操 作	操作效果	实现方式
push(item)	将值为item的新元素压入栈顶，无返回值	调用基础容器的 push_back 操作实现
pop()	删除栈顶元素，无返回值	调用基础容器的 pop_back 操作实现
top()	返回栈顶元素的值	调用基础容器的 back 操作实现
empty()	若栈为空，则返回 true ；否则，返回 false	调用基础容器的 empty 操作实现
size()	返回栈中元素的数目	调用基础容器的 size 操作实现



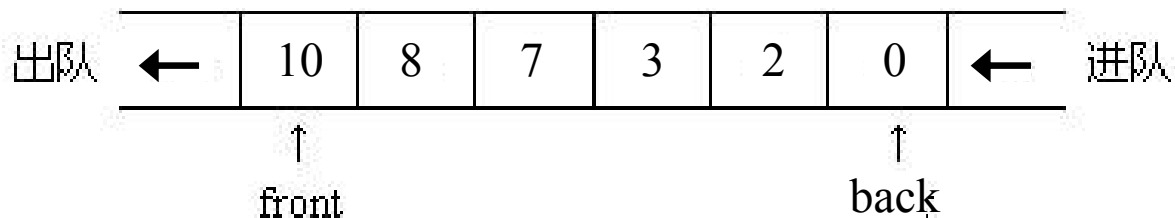
Operations of queue

操 作	操作效果	实现方式
push(item)	将值为item的新元素插入队尾，无返回值	调用基础容器的 push_back 操作实现
pop()	删除队首元素，无返回值	调用基础容器的 pop_front 操作实现
front()	返回队首元素的值	调用基础容器的 front 操作实现
back()	返回队尾元素的值	调用基础容器的 back 操作实现
empty()	若队列为空，则返回 true ；否则，返回 false	调用基础容器的 empty 操作实现
size()	返回队列中元素的数目	调用基础容器的 size 操作实现



Operations of priority_queue

操 作	操作效果	实现方式
push(item)	按优先级顺序将值为 item 的新元素插入到队列中适当位置，无返回值	首先调用基础容器的 push_back 操作，然后调用堆排序算法 push_heap 对元素进行重新排列
pop()	删除队首元素（优先级最高的元素），无返回值	首先调用堆操作算法 pop_heap 删除堆顶元素，然后调用基础容器的 pop_back 操作
top()	返回队首元素的值	调用基础容器的 front 操作实现
empty()	若优先级队列为空，则返回 true ；否则，返回 false	调用基础容器的 empty 操作实现
size()	返回优先级队列中元素的数目	调用基础容器的 size 操作实现



Container Adapter

- 标准库中定义的容器适配器都是基于顺序容器建立的。
- 程序员在创建适配器对象时可以选择相应的基础容器类。
 - 1) **stack**适配器可以建立在**vector**、**list**或**deque**容器上。
 - 2) **queue**适配器只能建立在**list**或**deque**容器上。
 - 3) **priority_queue**适配器只能建立在**vector**或**list**容器上。
- 如果创建适配器对象时不指定基础容器，则**stack**和**queue**默认采用**deque**实现，而**priority_queue**则默认采用**vector**实现。

EXP: Stack

// **palindrome.cpp** 功能： 判断给定文本是否回文（正读反读都一样的文本）
// 演示**stack**适配器的使用

```
bool palindrome(const string& text);
int main()
{
    string text;
    while (true) {
        cout << "Enter the text(\"quit\" to end program):" << endl; // 输入文本
        getline(cin, text);
        if (text == "quit")
            break;
        if (palindrome(text)) // 是回文
            cout << "The text you typed is a palindrome." << endl;
        else // 不是回文
            cout << "The text you typed is NOT a palindrome." << endl;
    }

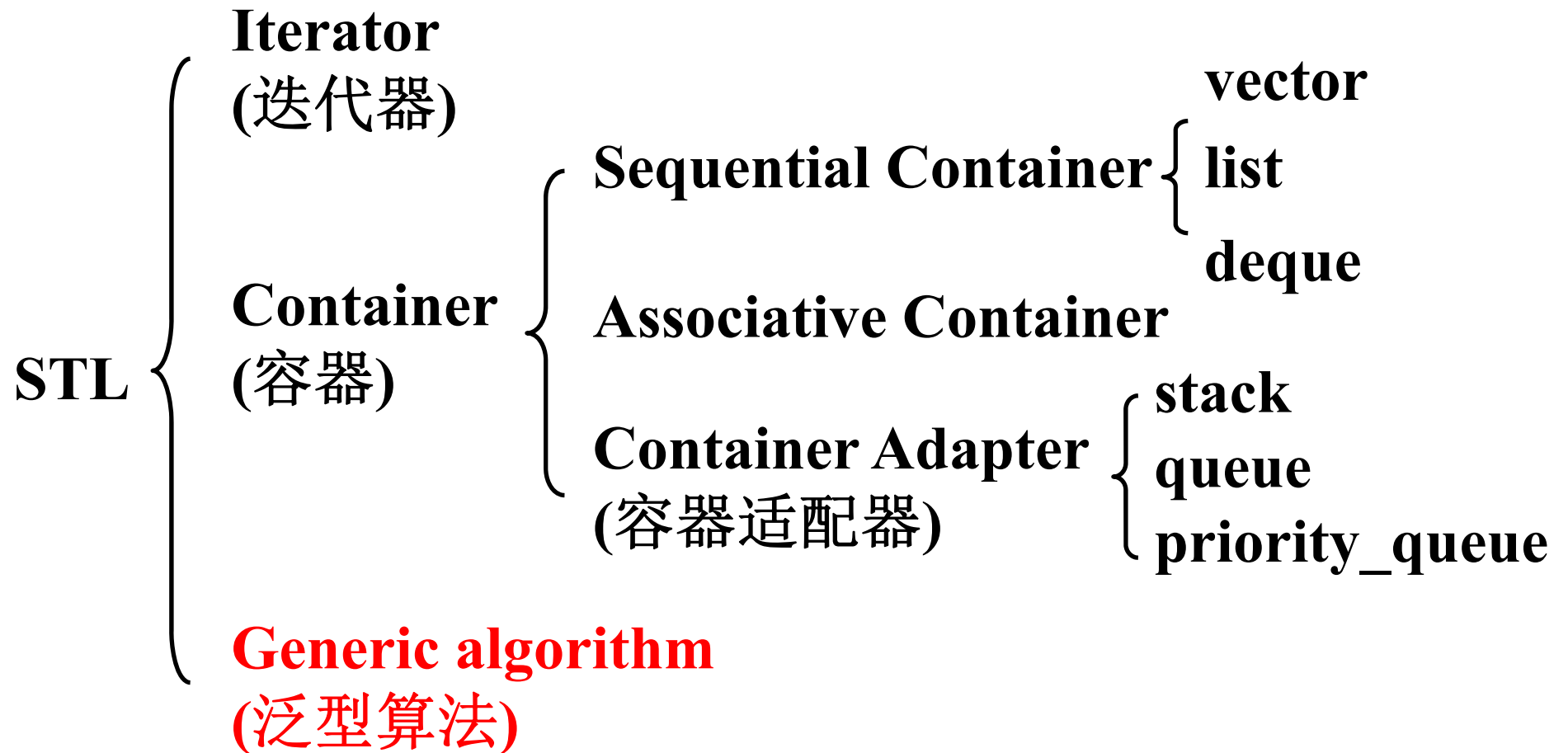
    return 0;
}
```

EXP: Stack

```
bool palindrome(const string& text)
{
    stack<char> cstack;
    size_t length = text.size();
    size_t comparePos;
    for (size_t i = 0; i < length/2; i++)
        cstack.push(text[i]);
    // 将文本text的后半部分逐个字符与前半部分的对应字符进行比较
    if (length % 2 == 0)
        comparePos = length / 2;
    else
        comparePos = length / 2 + 1;
    while (!cstack.empty()) {
        if (text[comparePos] != cstack.top())
            break;
        cstack.pop();
        comparePos++;
    }
    if (cstack.empty()) return true;
    else return false;
}
```

// 比较位置
// 将文本text的前半部分压入栈cstack
// 设定比较起点
// 比较对应字符
// 对应字符不相同
// 对应字符出栈
// 比较位置后移一个字符

STL



Generic algorithm

- “泛型”：算法与具体的容器类型无关，而且一般也不依赖于元素的类型（除了要求元素类型必须支持比较操作之外）。
- 四大类：
 - 1、不修改序列的算法（**non-modifying sequence algorithm**）
 - 2、变更序列的算法（**mutating-sequence algorithm**）
 - 3、排序及相关算法（**sorting and related algorithm**）
 - 4、泛化算术算法（**generalized numeric algorithm**）
- 前三类算法都在头文件<**algorithm**>中定义，泛化算术算法在头文件<**numeric**>中定义。

算法形参规范

alg (first, last, otherParams);

alg (first, last, result, otherParams);

alg (first, last, first2, otherParams);

alg (first, last, first2, last2, otherParams);

算法形参规范： **otherParams**

- **value**（元素类型的值）。例：

count(first, last, value)

- **comp**（表示比较关系的函数）。例

sort(first, last, comp)

comp是一个函数名，该函数必须接受两个形参（形参类型必须与元素类型相同），返回可作为条件检测的值（通常为**bool**类型）。

- **pred**（表示测试条件的函数）。例：

count_if(first, last, pred)

pred是一个函数名，该函数接受一个形参（形参类型必须与元素类型相同），返回可作为条件检测的值（通常为**bool**类型）。

算法命名规范（1）

- 加_if后缀。带_if后缀的算法使用程序员提供的比较或测试函数，不带_if后缀的同类算法则采用默认关系操作<或==进行比较或测试。例：

（1）count_if(first, last, pred)

//返回在迭代器范围[first, last)内，使得测试函数pred返回非0值的元素出现的次数。

（2）count(first, last, value)

//返回在迭代器范围[first, last)内，与value相等（==）的元素出现的次数。

算法命名规范（2）

- 加_**copy**后缀。带_**copy**后缀的算法对元素进行复制。例如

(1) **remove(first, last, value)**

//将迭代器范围[**first, last**)内与**value**相等的元素去掉。

(2) **remove_copy(first, last, result, value)**

//将迭代器范围[**first, last**)内的元素复制到迭代器**result**所指向的位置（去掉其中与**value**相等的元素）。迭代器范围[**first, last**)内的元素将保持不变。

算法命名规范（3）

- 若同类算法的不同版本能够在形参个数上有所区别，则采用相同的名字（构成函数重载）。例如：

（1）`sort(first, last)`

`//对迭代器范围[first, last)内的元素进行升序排列（使用<操作进行元素比较）。`

（2）`sort(first, last, comp)`

`//对迭代器范围[first, last)内的元素进行排序，排序时使用函数comp进行元素比较。`

EXP1: count_if

// CountIf1

```
bool GreaterThan( int ival )
{
    if ( ival > 50 ) return true;
    else             return false;
}
```

```
int main()
{
    int ia[10] = { 11, 20, 21, 60, 71, 40, 42, 50, 90, 100 };
    vector<int> ivec(ia,ia+10);
    int number;

    number = count_if( ivec.begin(), ivec.end(), GreaterThan );
    :
}
```

•count_if算法要求的pred形参是只接受一个形参的函数。

•50固化在函数中不灵活。但增加参数后（如下所示），就不能使用在count_if中。

```
//-----
bool GreaterThan( int ival, int ref )
{
    if ( ival > ref ) return true;
    else             return false;
}
```

如何突破参数个数的限制？

EXP2: count_if

//CountIf2。计算某段数值范围内的元素个数。使用“函数对象”。

```
class BetweenCls {
public:
    BetweenCls(int ival1, int ival2): lowerBound(ival1), upperBound(ival2)
    { }
    bool operator() (const int& ival)
    { return (ival > lowerBound && ival < upperBound); }

private:
    int lowerBound, upperBound;
};

int main()
{
    :
    number = count_if( ivec.begin(), ivec.end(), BetweenCls( 35,95 ) );
    :
}
```

临时变量() -> 临时变量.operator()

