華中科技大學

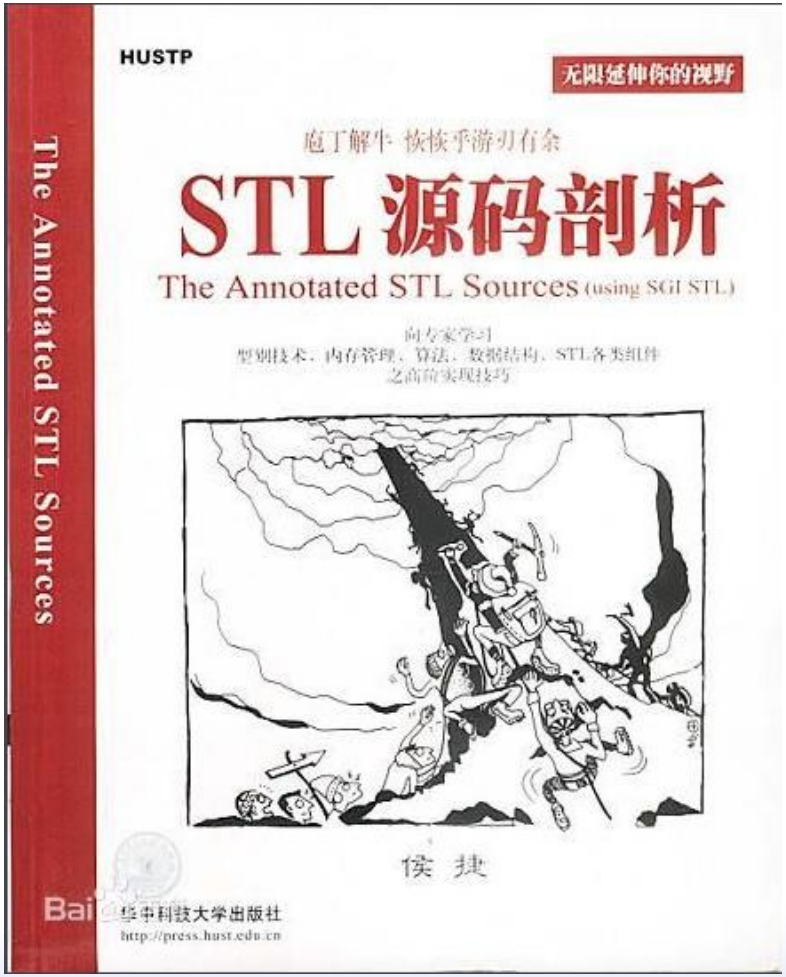
C++的标准模板库 STL

Standard Template Library

许向阳

xuxy@hust.edu.cn

参考书



肖波. 数据结构与STL. 北京邮电大学出版社. 2010年

bilibili

C加加STL

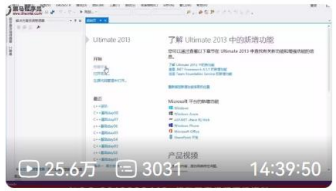
UP 马甲--马甲 · 6-26

清华大佬超全超详细讲解——C++STL看这份视频就够了

UP 编程其实也不难 · 2023-1-10

2024版C++与STL库开发

课堂 远航哥嵌入式 · 共145课时

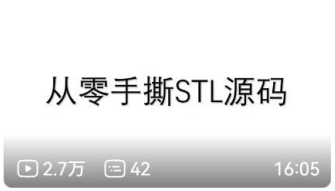


黑马程序员2017C++STL教程（已完结）

UP 可爱的小飞猪 · 2018-2-27

B站讲的最好的STL源码剖析PJ版

UP IT资料小金库 · 11-3



【从零手撕STL源码】1、初识模板

UP 宇文新粥 · 2022-5-28

如何阅读C++ STL 源码?

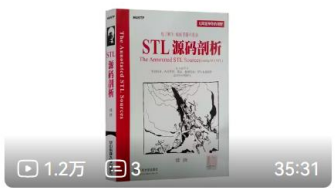
UP mq白cpp · 3-29

STL源码剖析PJ版

UP itshare1024 · 10-7

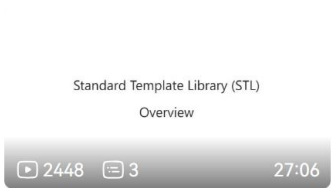
STL深入浅出教程|传智教育

UP 传智教育 · 2019-4-16



《STL源码剖析》_快速讲解

UP 大话孙 · 2022-6-9



课堂目标

- 掌握 STL中的容器、算法、迭代器、仿函数
  空间配置器、配接器　各自的作用
- 了解 STL 的实现机理
- 能看懂用 STL编写的程序
- 会使用 STL 编写程序

提纲

1. 概论
2. STL中的基本概念
3. 容器
4. 迭代器
5. 算法

软件重用

- 面向对象的思想
  - 封装、继承和多态
  - 标准类库 Microsoft Foundation Classes
- 泛型程序设计的思想
  - generic programming
  - 模板机制：函数模板、类模板
  - 标准模板库STL Standard Template Library

泛型程序设计的核心思想

"编写一次，处处使用" - 创建不依赖于具体数据类型的通用算法和数据结构，通过类型参数化实现代码复用，同时保持类型安全和性能。

这种思想让程序员能够：

- 从重复的编码工作中解放出来
- 构建更加灵活和可维护的系统
- 在抽象层面思考问题，而不是陷入具体类型的细节

泛型程序设计的思想

1. 类型参数化

将数据类型作为参数传递，让同一段代码能够处理多种不同的数据类型。

2. 代码复用

避免为相似逻辑但不同类型的数据重写代码。

3. 抽象与通用性

关注算法和逻辑的本质，而不是具体的数据类型。

Don't Repeat Yourself (DRY)

避免代码重复，提高维护性。

编译时多态

在编译时确定具体类型，无运行时开销。

1. 概论

华中科技大学

泛型程序设计——使用模板的程序设计方法

- 常用的数据结构（如数组，链表，二叉树）
- 常用算法（如排序，查找）
- 常用的数据结构和算法写成模板
- 不论数据结构里放什么对象，算法针对什么对象，都不必重新实现数据结构，不必重新编写算法。
- STL主要由 Alex Stepanov 开发，于1998年被添加进C++标准

1. 概论

STL的优点

- 是C++的一部分，内建在编译器内，使用简单
- 算法与数据结构分离，使得STL非常通用
- 高可重用性（使用类模板、函数模板）
- 高性能 （STL 内部实现的性能好）
- 高移植性 （一个项目的程序 用到另一个项目）
- 跨平台（不同的操作系统，不同的开发工具）

2. STL中的基本概念

華中科技大學

容　器：可容纳各种数据类型的数据结构　Containers

Sequence　Containers　序列式容器

Associative Containers　关联式容器

迭代器：可依次存取容器中元素　Iterators

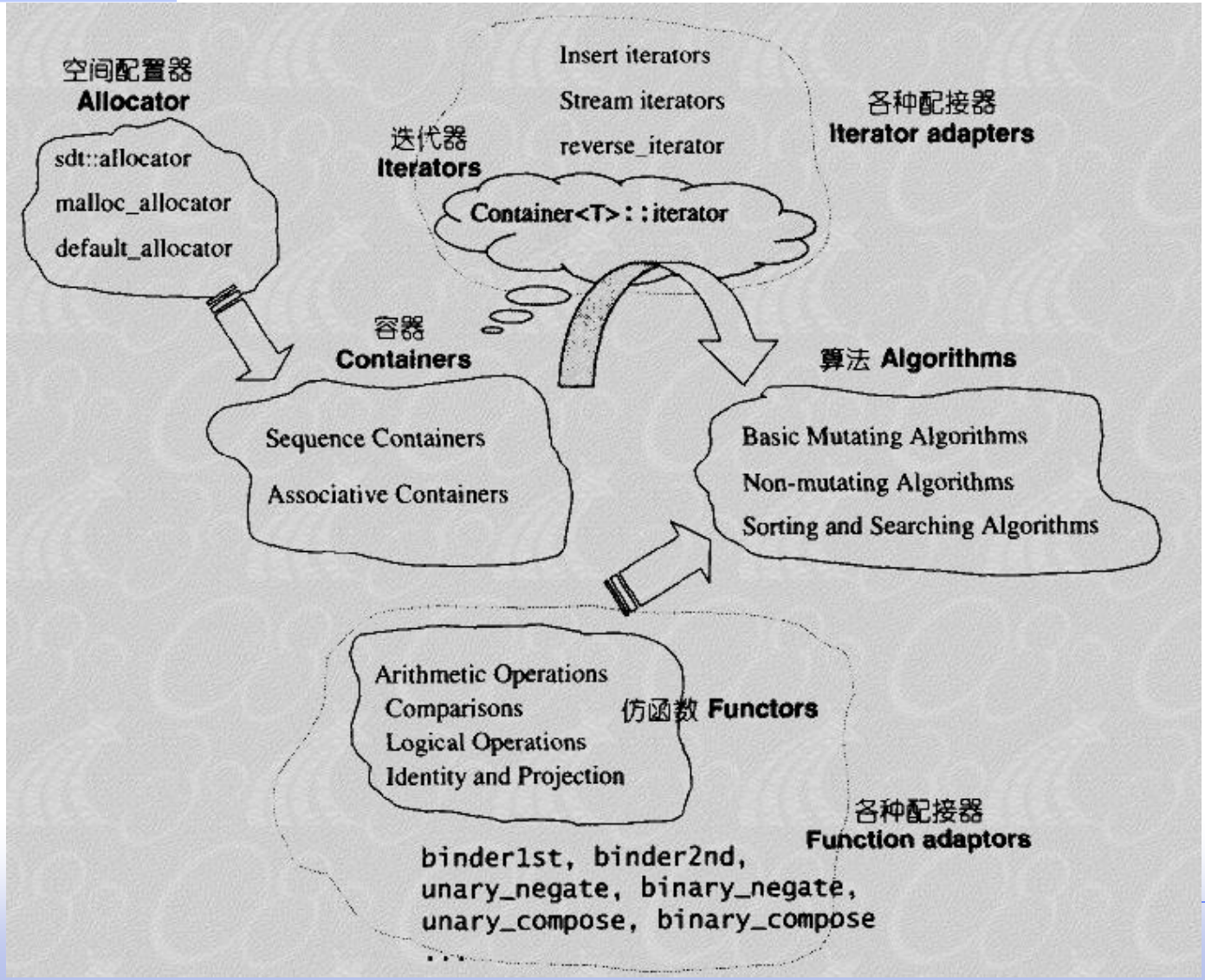
算　法：用来操作容器中的元素的函数模板　Algorithms

空间配置器：　负责空间配置和管理　Allocators

仿函数：类似于函数，重载operator (), functors

配接器：将一个接口转换为接口　Adaptors

2. STL中的基本概念


空间配置器
Allocator
sdt::allocator
malloc_allocator
default_allocator
迭代器
Iterators
Insert iterators
Stream iterators
reverse_iterator
Container<T>::iterator
各种配接器
Iterator adapters
容器
Containers
Sequence Containers
Associative Containers
算法 Algorithms
Basic Mutating Algorithms
Non-mutating Algorithms
Sorting and Searching Algorithms
Arithmetic Operations
Comparisons
Logical Operations
Identity and Projection
仿函数 Functors
各种配接器
Function adaptors
binder1st, binder2nd,
unary_negate, binary_negate,
unary_compose, binary_compose
...

2. STL中的基本概念

Container 、Allocator 、 Iterator 、Algorithm、Functor、Function Adapter 之间的关系

- Container 通过Allocator 取得数据储存空间
- Algorithm 通过Iterator 存取Container 中的内容
- Functor 协助Algorithm 完成不同的策略变化
- Adapter 可以修饰或套接Functor

2. STL中的基本概念

- Algorithm 是函数，Functor 也是一个函数，但Functor 作为Algorithm 的参数

函数作为参数

```
#include <iostream>
using namespace std;
int fadd(int a, int b)        { return a + b; }
int fsubtract(int a, int b)  { return a - b;  }
int f(int a, int b, int (*fp)(int, int))  {
    int temp= fp(a, b);
    a += 10;        cout << "a=" << a << endl;
    return temp;
}
int main( ) {
    cout << f(3, 4, fadd) << endl;
    cout << f(3, 4, fsubtract) << endl;
}
```

2. STL中的基本概念

函数作为参数

```
#include <iostream>
using namespace std;
int fadd(int a, int b)          {  return a + b; }
int f(int a, int b, int (*fp)(int, int))  {
    int temp= fp(a, b);
    a += 10;        cout << "a=" << a << endl;
    return temp;
}
int main( ) {    //  Lambda 表达式为仿函数，作为函数参数
    auto myf = [](int x, int y) {return x + y; };
    cout << f(3, 4, fadd) << endl;
    cout << f(3, 4, myf) << endl;
    cout << f(3, 4, [](int x, int y) {return x + y; }) << endl;
}
```

2. STL中的基本概念

函数作为参数

设有 Person 类, p1,p2,p3三个对象

```
int main( ) {
    vector<Person>  v ={p1, p2, p3};

     sort(v.begin(), v.end(), [ ](const Person& a1, const
Pesron & a2) {return strcmp(a1.name, a2.name)<0; } );
                采用不同的仿函数，作为排序策略
}
```

2. STL中的基本概念

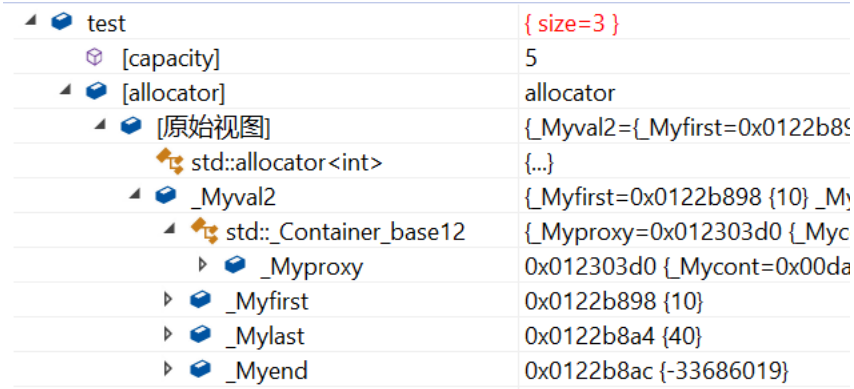
容器 vector

_Myproxy
_Myfirst
_Mylast
_Myend

调试：迭代器的安全性和有效性检查

```
vector<int> test = { 10,20,30,40,50 };
test.pop_back();
test.pop_back();
 // sizeof(test) 是 16
 // test.size( )          3
 // test.capacity( )    5
```

对象体外空间，来自空间配置器



pop_back 从容器中删除了元素 40，主要是改变指针，并没有把相应位置的值清除。

vector 中只有一个数据成员

_Compressed_pair<_Alty, _Scary_val> _Mypair;

2. STL中的基本概念

容器 vector

華中科技大學

test	{ size=3 }
[capacity]	5
[allocator]	allocator
[原始视图]	{_Myval2={_Myfirst=0x0122b8
std::allocator<int>	{...}
_Myval2	{_Myfirst=0x0122b898 {10} _M
std::_Container_base12	{_Myproxy=0x012303d0 {_Myc
_Myproxy	0x012303d0 {_Mycont=0x00da
_Myfirst	0x0122b898 {10}
_Mylast	0x0122b8a4 {40}
_Myend	0x0122b8ac {-33686019}

监视窗口

内存 1

地址: 0x00DAFC10 ← 输入 &test

0x00DAFC10	d0	03	23	01	98	b8	22	01
0x00DAFC18	a4	b8	22	01	ac	b8	22	01

内存窗口

```
_Compressed_pair<_Alty, _Scary_val>  _Mypair;
```

_Mypair . _Myval2;

2. STL中的基本概念 容器 vector


```
template <class _Val_types>
class _Vector_val : public  _Container_base {
public:
    using value_type    = typename _Val_types::value_type;
    using size_type     = typename _Val_types::size_type;
    using pointer       = typename _Val_types::pointer;
    using const_pointer   = typename _Val_types::const_pointer;
……..
    pointer _Myfirst;  // pointer to beginning of array
    pointer _Mylast;  // pointer to current end of sequence
    pointer _Myend;  // pointer to end of array
};

   using _Container_base = _Container_base12;
   _Container_proxy* _Myproxy;    //  <xmemory>
```

2. STL中的基本概念

空间配置器 Allocator

- STL空间配置器分为一、二级配置器。
- 当申请的内存大于128字节时，使用一级配置器，小于128个字节时，利用二级配置器来分配内存。
- 一级配置器是对malloc的简单包装，从系统中申请内存。
- 二级配置器每一次配置一大块内存，并维护其对应的自由链表(free_list)。

- 当客户要求空间时，适配器便会将符合其大小的第一个空间给予客户；
- 当客户返还内存时，直接将其插入对应的自由链表中。

2. STL中的基本概念

空间配置器 Allocator

- STL空间配置器分为一、二级配置器。
- 当申请的内存大于128字节时，使用一级配置器，小于128个字节时，利用二级配置器来分配内存。
- 一级配置器是对malloc的简单包装，从系统中申请内存。
- 二级配置器每一次配置一大块内存，并维护其对应的自由链表(free_list)。

- 当客户要求空间时，适配器便会将符合其大小的第一个空间给予客户；
- 当客户返还内存时，直接将其插入对应的自由链表中。

2. STL中的基本概念

空间配置器 Allocator

```
template <class _Ty, class _Alloc = allocator<_Ty>>
class vector {
    ……
};

template <class _Ty>
class allocator {……
};
```

2. STL中的基本概念　迭代器

迭代器是容器和算法之间的桥梁。它提供了一种方法来顺序或随机访问容器中的元素，而无需暴露容器的内部表示。

- 无论容器是数组、链表、树还是其他数据结构，迭代器都提供了一组统一的操作（如*、++、--、->等），使得算法可以以相同的方式处理不同类型的容器。
- 根据迭代器的类型（输入、输出、前向、双向、随机访问），它们支持不同层次的操作。例如，随机访问迭代器（如vector的迭代器）支持跳跃式访问，而双向迭代器（如list的迭代器）只支持前后移动。

2. STL中的基本概念

迭代器 含有的重载运算符：

+、-、++、--、=、+=、-=

<、<=、==、>、>=、!=、[ ]、 *、->

vector<int>::iterator it1 = v1.begin( );

it1++; ++it1; 指向下一个元素

it1=it1+3; 指针向后移3个元素

*it1 指向的当前元素

auto it1 = v1.begin( ); 简化定义方式，返回是 iterator

vecor<int>::const_iterator cit1 = v1.cbegin( );

auto cit1=v1.cbegin( );

2. STL中的基本概念

vector<int>::iterator it1; 的解释

iterator 是 vector 的嵌套类吗？ 不是

using iterator =_Vector_iterator<_Scary_val>;

```
class B {
public:
    int y;
public:
    B(int v) :y(v){ cout << "B :" <<y<< endl; }
};

class A {
    int x;
public:
    using myclass = B;
    A(int v){ x = v; }
};

class AA {
    int x;
public:
    using myclass = B;
    AA(int v) { x = v; }
};
```

```
A::myclass    uuu(10);
AA::myclass  vvv(100);
```

```
B  :10
B  :100
```

myclass 是 A类下的符号，
是 B 类的代名词
功能等同 嵌套类

2. STL中的基本概念

迭代器 Iterator

vector<int> v1;　　vector<A>::iterator it ;

v1.size( ) 为 3；　　设 v1.capacity( ) 为 5

v1. begin( )
v1.cbegin( )

v1. end( )
v1. cend( )

v1. rend ( )
v1. crend( )

v1. rbegin ( )
v1. crbegin( )

c ： const　　不能通过迭代器进行数据更改操作

r ： reverse

it = v1.begin( ):

2. STL中的基本概念

STL中提供能在各种容器中通用的算法

- 插入
- 删除
- 查找
- 排序
- 大约有70种标准算法。

算法可以处理容器，也可以处理C语言的数组

2. STL中的基本概念

仿函数　　也称为函数对象

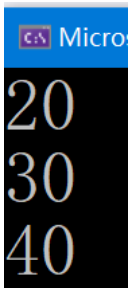
- 是基于C++运算符重载而产生的一个重要工具。
- 仿函数的实现是创建一个类，类中有重载operator()的函数
- 使用的时候是先创建一个对象，然后用这个对象就可以作为函数使用了（或者用operator()创建一个临时对象当作函数也行）。
- 仿函数按操作数可分为一元、二元仿函数；
- 按功能分可分为算法运算、关系运算；
- Lamba 表达式就是一种仿函数

2. STL中的基本概念


```
vector<int> test = { 10,20,30 };
    // 容器中每个元素 加 10
for_each(test.begin(), test.end(), [](int &i) {i = i + 10; });
for_each(test.begin(), test.end(), [](int i) { cout << i << endl; });

class Add_10 {
    public:   void operator( )(int& i) { i += 10; }
};
for_each(test.begin( ), test.end( ), Add_10( ));

auto myfunctor = [](int& i) {i += 10; };
for_each(test.begin( ), test.end( ),  myfunctor);
```



2. STL中的基本概念

类比：函数模板的定义、使用

```
auto myfunctor = [](int& i) {i += 10; };

for_each(test.begin( ), test.end( ),  myfunctor);
```

```
template <class _InIt, class _Fn>
_CONSTEXPR20 _Fn for_each(_InIt _First, _InIt _Last, _Fn _Func) {
    _Adl_verify_range(_First, _Last);
    auto _UFirst      = _Get_unwrapped(_First);
    const auto _ULast = _Get_unwrapped(_Last);
    for (; _UFirst != _ULast; ++_UFirst) {
        _Func(*_UFirst);
    }
    return _Func;
}
```

Q:myfunctor 能不能是一个普通函数？

```
void   myfunctor(int& i)
{   i += 10;   }
```

2. STL中的基本概念

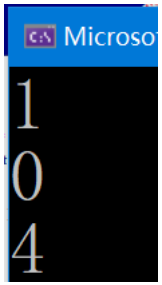
配接器 STL所提供的各种适配器

将一个 class 的接口转换为 另一个class 的接口，使得因接口不兼容而不能互作的 classes 可以一起运作。

改变仿函数（functors）接口者，称为function adapter；
改变容器（containers）接口者，称为container adapter；
改变迭代器（iterators）接口者，称为iterator adapter。

2. STL中的基本概念


```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
using namespace std;
int main()
{
    vector<int> v = { 1,2,3,10,20,30 };
    cout << less<int>( )(3, 5) << endl;
    cout << less<int>( )(5, 3) << endl;
    cout<<count_if(v.begin(), v.end(), bind2nd(less<int>( ), 15));
}
```



less 函数有两个参数，bind2nd 将其换成一元函数

3、容器

序列式容器

vector、list、forward_list、deque、stack 、queue、array

关联式容器

有序关联容器

set、map、multiset、multimap

无序关联容器

unordered_set、unordered_map、

unordered_multiset、unordered_multimap

3、容器

序列式容器 ： 取决于插入的时间、地点，
与插入值的大小无关

vector：动态数组、多种插入/删除方法、可直接访问

list： 双向链表，任意位置插入/删除，不支持随机访问

forward_list：单向链表，任意位置快速插入/删除

deque：双端数组，前部插入/删除，
后部插入/删除，直接访问

stack： 堆栈 LIFO

queue： 队列 FIFO

#include <vector> <list> <deque> <stack> <queue>

3、容器

关联式容器 ：

按照关键字（key）来存储元素，

元素的位置取决于特定的排序准则，与插入顺序无关。

通常使用平衡二叉搜索树（如红黑树）或哈希表实现。

有序关联容器（通常基于红黑树）：

set：　只包含关键字，关键字不可重复。

map：包含键值对，键不可重复。

multiset：允许重复关键字的set。

multimap：允许重复键的map。

3、容器

关联式容器 ：

按照关键字（key）来存储元素，

元素的位置取决于特定的排序准则，与插入顺序无关。

通常使用平衡二叉搜索树（如红黑树）或哈希表实现。

无序关联容器（基于哈希表）：

unordered_set：哈希集合，关键字不可重复。

unordered_map：哈希映射，键不可重复。

unordered_multiset：允许重复关键字的哈希集合。

unordered_multimap：允许重复键的哈希映射。

3、容器

華中科技大學

特性	序列式容器	关联式容器
核心逻辑	维护**插入顺序**	维护**键的逻辑关系**（排序或哈希）
组织方式	线性结构（数组、链表）	树形结构（红黑树）或哈希表
访问方式	主要通过**位置/索引**	主要通过**键**
元素顺序	由用户插入顺序决定	由容器根据键自动决定（有序容器）或无明确顺序（无序容器）
查找效率	顺序查找 O(n)，如果排序后二分查找 O(log n)	**高效查找**，有序容器 O(log n)，无序容器平均 O(1)
典型用途	需要保持元素先后关系的场景	需要根据关键字快速查找、删除的场景

3、容器——vector


```
#include <vector>
vector<int>  intVector;
vector<float>  floatVector;
vector<string>  stringVector;
class  A {……};
vector<A>    classAVector;
vector<A*>  classAPointerVector;
```

向量中的元素可以是各种类型

3、容器——vector

vector：可变大小数组的序列容器　动态数组

vector 采用数组作为容器，空间不够时再重新分配内存，

拷贝原来数组的元素到新分配的数组中。

```
vector<int> t;
for (int i = 0;i < 20;i++) {
    cout << endl << "insert  element " << i << endl;
    t.push_back(i);
    cout << "size =" << t.size()<<endl;
    cout << "capacity =" << t.capacity()<<endl;
}
```

```
t[5] = 33;
```

直接修改数组的某个元素

capacity ： 当前容量

size　　： 实际元素个数

3、容器——vector

```
insert  element 12
size =13
capacity =13

insert  element 13
size =14
capacity =19

insert  element 14
size =15
capacity =19

insert  element 15
size =16
capacity =19
```

vector 采用数组作为容器，

元素不够时再重新分配内存，

拷贝原来数组的元素到新分配的数组中。

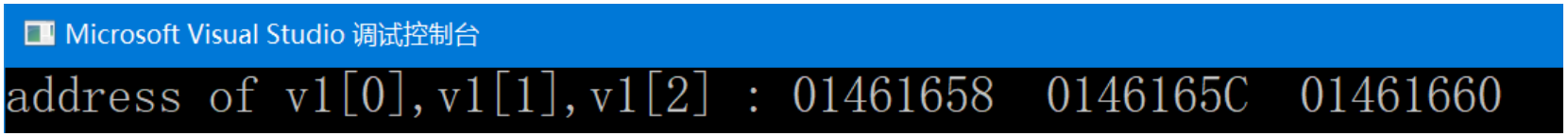
新数组扩大多少？

现有元素个数的一半

3、容器——vector

实验：验证 vector 采用数组作为容器

```
vector<int> v1;
v1.push_back(10);
v1.push_back(20);
v1.push_back(30);
cout << "address of v1[0],v1[1],v1[2] : " << &v1[0]
     << "  " << &v1[1] << "  " << &v1[2] << endl;
```



3、容器——vector

实验：验证 在数组尾部插入元素，比在头部插入元素快

```
#include <vector>
#include <iostream>
#include <Windows.h>
using namespace std;
 vector<int> v1;
 int start, end;
 start=GetTickCount();
 for (int i = 0;i < 100000;i++)
        v1.push_back(i);
    // v1.insert(v1.begin(),i);    在头部插入元素
 end = GetTickCount();
 cout << "time : " << end - start << endl;
```

```
time : 31
time : 907
```

vector 未提供 push_front,

猜想原因：怕开发者使用效率低的方法

Vector 的构造函数


```
vector( const Allocator & = Allocator( ));
vector( size_type n,  const & value =T( ),
                        const Allocator & = Allocator( ));
vector(initializer_list<_Ty> _Ilist, const _Alloc& _Al = _Alloc());
vector(_Iter _First, _Iter _Last, const _Alloc& _Al = _Alloc());
vector(const vector& _Right);
vector(vector&& _Right);
```

缺省参数 Allocator，用于指定要使用的空间配置器。

STL提供默认的空间配置器，基本不用管该参数。

3、容器——vector

与 容量 有关的成员函数

size( ) ：　　实际元素个数

max_size( )：　动态增长的数组，最多能长到多大

capacity()：　当前容器的大小

resize( ) ：　　更改容器的大小

empty( )：　　容器中是否含有元素

shrink_to_fit( )：　减少容器的大小，正好装下所有元素

3、容器——vector

增、删、插入 元素的成员函数

多个元素赋值： assign();

末尾添加元素： push_back();

末尾删除元素： pop_back();

任意位置插入元素： insert();

任意位置删除元素： erase();

交换两个向量的元素： swap();

清空向量元素： clear();

在指定位置构造元素：emplace( );

在尾部构造元素：emplace_back( );

3、容器——vector

元素访问的成员函数

下标访问： vec[1]; //不会检查是否越界

at方法访问： vec.at(1); //会检查是否越界，
是则抛出out of range异常

访问第一个元素： vec.front( );

访问最后一个元素： vec.back( );

返回一个指针： 元素类型T * p = vec.data( );

VECTOR示例

vector中常用的函数

```
void assign(_Iter _First, _Iter _Last);
```

将[_First, _Last)区间中的数据赋值给 vector 对象；

替换旧元素为新元素，可以修改向量的大小

```
void assign(_CRT_GUARDOVERFLOW const size_type _Newsize, const
_Ty& _Val);
```

将 _Newsize 个 _Val 的拷贝赋值赋值给 vector 对象

VECTOR insert

華中科技大學

在_Where 位置之前插入，返回指向新数据的 iterator

```
iterator insert(const_iterator _Where, _Ty&& _Val);
    //  插入一个新元素  _Val

iterator insert(const_iterator _Where, _CRT_GUARDOVERFLOW
const size_type _Count, const _Ty& _Val);
    //  插入 _Count 个 _Val ，

iterator insert(const_iterator _Where, _Iter _First, _Iter _Last);
    //  插入在[ _First, _Last)区间的数据。

iterator insert(const_iterator _Where, initializer_list<_Ty> _Ilist);
    //  插入list 中的数据，如 {1, 2, 3} 。
```

VECTOR 的成员函数

assign
at
back
begin
capacity
cbegin
cend
clear
crbegin
crend
data

emplace
emplace_back
empty
end
erase
front
get_allocator
insert
max_size
operator =
operator [ ]
pop_back
push_back

rbegin
rend
reserve
resize
shrink_to_fit
size
swap
vector

_Emplace_reallocate
_Unchecked_begin
_Unchecked_end
~vector

VECTOR 的成员函数—参考文档

https://cplusplus.com/reference/

Containers

<array>	Array header (header)
<bitset>	Bitset header (header)
<deque>	Deque header (header)
<forward_list>	Forward list (header)
<list>	List header (header)
<map>	Map header (header)
<queue>	Queue header (header)
<set>	Set header (header)
<stack>	Stack header (header)
<unordered_map>	Unordered map header (header)
<unordered_set>	Unordered set header (header)
<vector>	Vector header (header)

VECTOR 的成员函数—参考文档

https://en.cppreference.com/

C++ reference

C++11, C++14, C++17, C++20, C++23, C++26 | Compiler support C++11, C++14, C++17, C++20, C++23, C++26

Language
- Keywords – Preprocessor
- ASCII chart
- Basic concepts
  - Comments
  - Names (lookup)
  - Types (fundamental types)
  - The main function
- Expressions
  - Value categories
  - Evaluation order
  - Operators (precedence)
  - Conversions – Literals
- Statements
  - if – switch
  - for – range-for (C++11)
  - while – do-while
- Declarations – Initialization
- Functions – Overloading
- Classes (unions)
- Templates – Exceptions
- Freestanding implementations

Standard library (headers)

Named requirements

Diagnostics library
- Assertions – System error (C++11)
- Exception types – Error numbers
- basic_stacktrace (C++23)
- Debugging support (C++26)

Memory management library
- Allocators – Smart pointers
- Memory resources (C++17)

Metaprogramming library (C++11)
- Type traits – ratio
- integer_sequence (C++14)

General utilities library
- Function objects – hash (C++11)
- Swap – Type operations (C++11)
- Integer comparison (C++20)
- pair – tuple (C++11)
- optional (C++17)
- expected (C++23)
- variant (C++17) – any (C++17)
- bitset – Bit manipulation (C++20)

Containers library
- vector – deque – array (C++11)
- list – forward_list (C++11)
- map – multimap – set – multiset

Strings library
- basic_string – char_traits
- basic_string_view (C++17)
- Null-terminated strings:
  - byte – multibyte – wide

Text processing library
- Primitive numeric conversions (C++17)
- Formatting (C++20)
- locale – Character classification
- text_encoding (C++26)
- Regular expressions (C++11)
  - basic_regex – Algorithms
  - Default regular expression grammar

Numerics library
- Common math functions
- Mathematical special functions (C++17)
- Mathematical constants (C++20)
- Basic linear algebra algorithms (C++26)
- Pseudo-random number generation
- Floating-point environment (C++11)
- complex – valarray

Date and time library
- Calendar (C++20) – Time zone (C++20)

Input/output library

3、容器

关联容器：元素位置取决于特定的排序准则，
与插入顺序无关

set：　按值的大小排列；由结点组成的红黑树
快速查找，无重复元素

multiset：快速查找，可有重复元素

map：　一对一映射，无重复元素，基于关键字查找

multimap：一对一映射，可有重复元素，基于关键字查找

hash_map　hash_multimap

hash_set　hash_multiset

3、容器

容器适配器

stack：LIFO

queue：FIFO

priority_queue：优先级高的元素先出

3、容器

list

- **Element access**:
  front Access first element (public member function)
- back Access last element (public member function)

Modifiers:

- assign Assign new content to container (public member function)
- push_front Insert element at beginning (public member function)
- pop_front Delete first element (public member function)
- push_back Add element at the end (public member function)
- pop_back Delete last element (public member function)
- insert Insert elements (public member function)
- erase Erase elements (public member function)
- swap Swap content (public member function)
- clear Clear content (public member function)

3、容器

- **Iterators**:
  begin Return iterator to beginning (public member function)
- **end** Return iterator to end (public member function)
- **rbegin** Return reverse iterator to reverse beginning (public member function)
- **rend** Return reverse iterator to reverse end (public member function)

Capacity:

empty Test whether container is empty (public member function)

- **size** Return size (public member function)
- **max_size** Return maximum size (public member function)
- **resize** Change size (public member function)

3、容器

Operations:

splice Move elements from list to list (public member function)

- **remove** Remove elements with specific value (public member function)
- **remove_if** Remove elements fulfilling condition (public member function template)
- **unique** Remove duplicate values (member function)
- **merge** Merge sorted lists (public member function)
- **sort** Sort elements in container (public member function)
- **reverse** Reverse the order of elements (public member function)

4、迭代器

Iterator

类似于指针，亦称 广义指针，指向某个对象。

- 为算法提供输入数据
- 遍历容器或者流中的对象
- 不同容器上支持的迭代器功能强弱有所不同。
- 容器的迭代器的功能强弱，决定了该容器是否支持STL中的某种算法。

4、迭代器

容器所支持的迭代器类别

容器	迭代器类别
vector	随机
deque	随机
list	双向
set/multiset	双向
map/multimap	双向
stack	不支持迭代器
queue	不支持迭代器
priority_queue	不支持迭代器

4、迭代器

迭代器类模板的类型

1.普通迭代器：每个容器都定义了自身的迭代器类型，如vector<T>::iterator、list<T>::iterator等。这些迭代器通常是类模板，但具体实现由容器决定。

2.反向迭代器：reverse_iterator，用于逆向遍历容器。

3.插入迭代器：包括insert_iterator、front_insert_iterator、back_insert_iterator，用于在容器中插入元素。

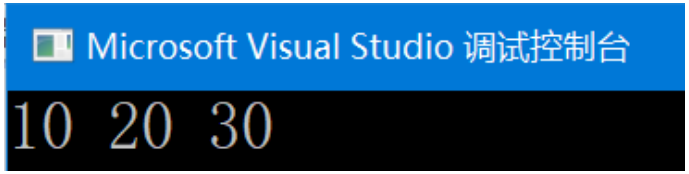
4.流迭代器：包括 istream_iterator和ostream_iterator，用于从流中读取或向流中写入数据。

5.移动迭代器：move_iterator，用于将指向的元素转换为右值引用，从而允许移动而非拷贝。

6.迭代器适配器：例如reverse_iterator，它也是一种适配器，用于反转迭代器的方向。

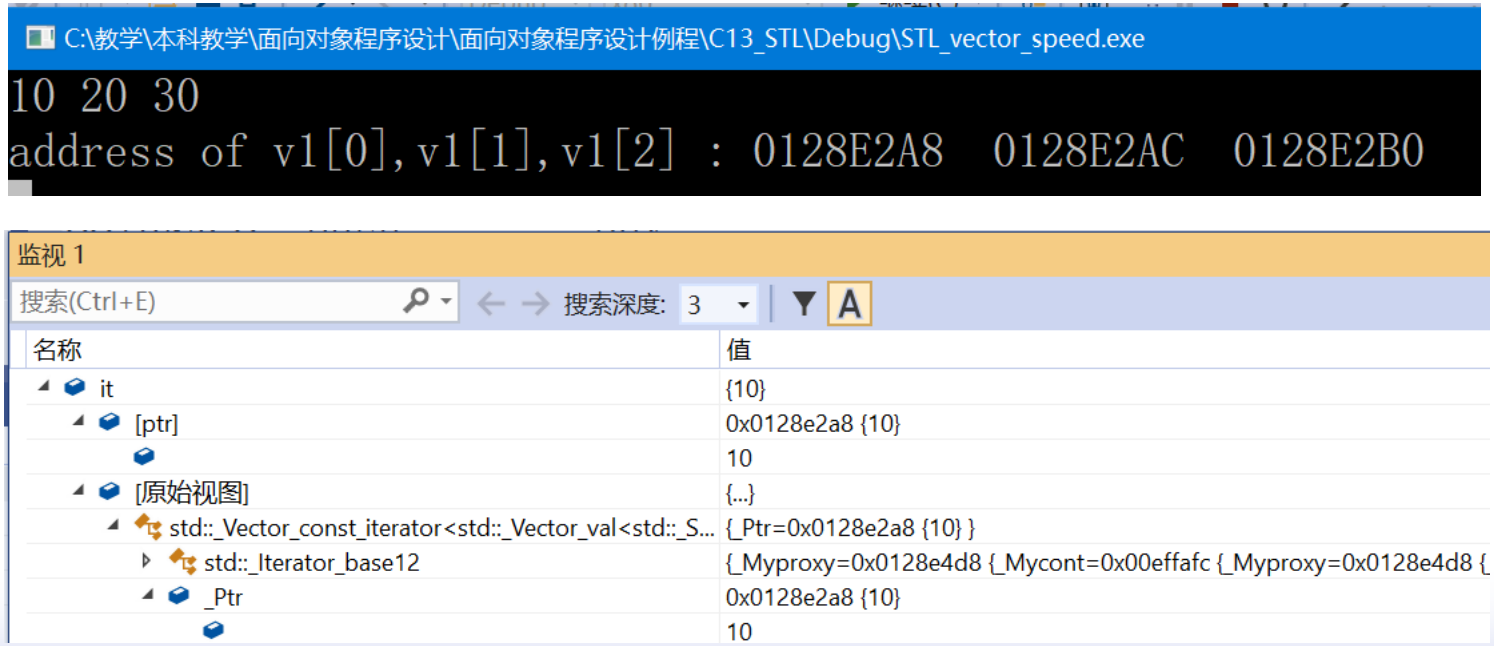
4、迭代器——vector::iterator

```
vector<int> v1;
v1.push_back(10);
v1.push_back(20);
v1.push_back(30);
vector<int>::iterator it;
for (it = v1.begin();it != v1.end(); it++)
    cout << *it << " ";
```



4、迭代器——vector::iterator

```
cout << "address of v1[0],v1[1],v1[2] : " << &v1[0]
     << "  " << &v1[1] << "  " << &v1[2] << endl;
it = v1.begin();
```



4、迭代器——vector::iterator

華中科技大學

正确理解 iterator

```
vector<int>::iterator it;
```

iterator 是一个类

它重载了 前置的 ++、--；后置的++、--；

*； ->；+=；-= 等运算符

里面有指针成员 指向要访问的元素

4、迭代器——vector::iterator

華中科技大學

```
_NODISCARD reference operator*() const {
    return const_cast<reference>(_Mybase::operator*());
}

_NODISCARD pointer operator->() const {
    return _Const_cast(_Mybase::operator->());
}

_Vector_iterator& operator++() {
    _Mybase::operator++();
    return *this;
}

_Vector_iterator operator++(int) {
    _Vector_iterator _Tmp = *this;
    _Mybase::operator++();
    return _Tmp;
}
```

5、算法

STL中提供能在各种容器中通用的算法

- 插入
- 删除
- 查找
- 排序
- 大约有70种标准算法。

算法可以处理容器，也可以处理C语言的数组

5、算法

- 变化序列算法

copy ,remove,fill,replace,random_shuffle,swap, …..

会改变容器

- 非变化序列算法

adjacent-find, equal, mismatch,find ,count, search, count_if, for_each, search_n

- 以上函数模板都在<algorithm> 中定义
- 此外还有其他算法

5、算法

- ostream_iterator<int> output(cout , "*");

  定义了一个 ostream_iterator 对象，可以通过cout输出以 * 分隔的一个个整数

- copy (v.begin(),v.end(),output);

  导致v的内容在 cout上输出

5、算法

- copy 函数模板(算法)

  template<class InIt, class OutIt>

  OutIt copy(InIt first, InIt last, OutIt x);

  本函数对每个在区间[0, last - first)中的N执行一次 *(x+N) = * (first + N) , 返回 x + N

- copy (v.begin(),v.end(),output)

  first 和 last 的类型是

  vector<int>::const_iterator

  output 的类型是 ostream_iterator<int>

5、算法

排序和查找算法

- **Sort**

  template<class RanIt>

  void sort(RanIt first, RanIt last);

  void sort(const _RanIt _First, const _RanIt _Last, _Pr _Pred)

- **find**

  template<class InIt, class T>

  InIt find(InIt first, InIt last, const T& val);

  Find 返回一个 iterator

5、算法


```
int main()  {
      const int SIZE = 10;
      int a1[] = { 2,8,1,50,3,100,8,9,10,2 };
      vector<int> v(a1,a1+SIZE);
      ostream_iterator<int> output(cout," ");
      vector<int>::iterator location;
      location = find(v.begin(),v.end(),10);
      if( location != v.end()) {
            cout << endl << "1) " << location - v.begin();
      }
       sort(v.begin(),v.end());
       if( binary_search(v.begin(),v.end(),9))
           cout << endl << "3) " << "9 found";
      else    cout << endl << " 3) " << " 9 not found";
      return 0;
}
```

5、算法


```
#include <algorithm>
#include <numeric>
```

查找算法

find(beg, end, v)	在迭代区间[beg, end)内查找等于 v 的元素，找到返回对应的迭代器，否则返回 end
find_first_of(beg, end, beg2, end2)	在迭代区间[beg, end)内查找与区间[beg2, end2)内任意元素匹配的元素，然后返回一个迭代器，指向第一个匹配的元素。如果找不到元素，则返回第一个范围的 end 迭代器
find_end(beg, end, beg2, end2)	与 find_first_of 类似，区别：查找最后一个匹配的元素。

find_if（beg，end，func）：函数find的带一个函数参数的_if版本，与find功能相同

5、算法

搜索与统计算法

search(beg, end, beg2, end2)	在迭代区间[beg, end)内查找子序列[beg2, end2)
search_n(beg, end, n, v)	在迭代区间[beg, end)内查找连续 n 个元素 v
count(beg, end, v)	统计等于 v 的元素个数
lower_bound(beg, end, v)	查找非递减序列内第一个大于 v 的元素
upper_bound(beg, end, v)	查找非递减序列内第一个小于 v 的元素

http://blog.csdn.net/

count_if（beg，end，func）：函数count的_if版本。

5、算法

可变序列算法包括元素复制，变换，替换，填充，移除和随机生成等。

copy(beg, end, beg2)	将迭代区间[beg, end)元素复制到以 beg2 开始的区间
transform(beg,end,beg2,func)	功能同上，只是每个元素需要经过函数 func 处理
replace(beg, end, v1, v2)	将区间[beg, end)内等于 v1 的元素替换为 v2
fill(beg, end, v)	区间内元素都写入 v
fill_n(beg, n, v)	从位置 beg 开始的 n 个元素写入 v
generate(beg, n ,rand)	向从 beg 开始的 n 个位置随机填写数据
remove(beg, end)	移除区间[beg, end)内的元素，注意：并不真正删除
unique(beg, end)	剔除相邻重复的元素，注意：并不真正删除

5、算法

copy，transform，fill_n和generate都需要保证：输出序列有足够的空间。

删除函数并不真正删除元素，只是将要删除的元素移动到容器的末尾，删除元素需要容器擦除函数来操作。同理，独特的函数也不会改变容器的大小，只是这些元素的顺序改变了，是将无重复的元素复制到序列的前端，从而覆盖相邻的重复元素.unique返回的迭代器指向超出无重复的元素范围末端的下一位置。

remove_if(beg，end，func)：remove的_if版本。

replace_if(beg，end，func，v2)：replace的_if版本。

_copy版本，**需注意**：必须保证输出序列的大小不小于输入序列的大小。

remove_copy(beg，end，dest)：remove的_copy版本，将反转后的序列输出到从dest开始的区间。

remove_copy_if(beg，end，dest，func)：remove_copy的_if版本。

replace_copy(beg，end，dest，v1，v2)：replace的_copy版本。

replace_copy_if(beg，end，dest，func，v2)：replace_copy的_if版本。

5、算法

排序算法

sort(beg, end)	区间[beg, end)内元素按字典次序排列
stable_sort(beg, end, func)	同上，不过保存相等元素之间的顺序关系
partial_sort(beg, mid, end)	将最小值顺序放在[beg, mid)内
random_shuffle(beg, end)	区间内元素随机排序
reverse(beg, end)	将区间内元素反转
rotate(beg, mid, end)	将区间[beg, mid) 和 [mid, end)旋转，使mid为新的起点
merge(beg, end, beg2. end2, nbeg)	将有序区间[beg, end)和[beg2, end2)合并到一个新的序列nbeg 中，并对其排序

http://blog.csdn.net/

partial_sort对区间[beg，end]内的mid - beg个元素进行排序，将最小的mid - beg个元素有序放在序列的前mid - beg的位置上。

reverse_copy（beg，end，dest）：reverse的_copy版本。

rotate_copy（beg，mid，end，dest）：rotate的_copy版本。

5、算法

关系算法

equal(beg, end, beg2, end2)	判断两个区间元素是否相等
includes(beg, end, beg2, end2)	判断[beg, end)序列是否被第二个序列[beg2, end2)包含
max_element(beg, en)	返回序列最大元素的位置
min_element(beg, end)	返回序列最小元素的位置
mismatch(beg, end, beg2, end2)	查找两个序列中第一个不匹配的元素，返回一对iterator，标记第一个不匹配元素的位置

http://blog.csdn.net/

标准库还提供求最大值，最小值的max和min函数。

5、算法

堆算法

make_heap(beg, end)	以区间[beg, end)内元素建立堆
pop_heap(beg, end)	重新排序堆，使第一个与最后一个交换，并不真正弹出最大值
push_heap(beg, end)	重新排序堆，把新元素放在最后一个位置
sort_heap(beg, end)	对序列重新排序

5、算法

容器特有的算法

- list容器上的迭代器是双向的，而不是随机访问类型。
- 在此容器上不能使用需要随机访问迭代器的算法，

  如 sort及其相关的算法。
- 有一些其他的泛型算法，如合并，删除，反向和唯一，虽然可以用在list上，但性能低。
- 若算法利用列表容器实现的特点，则可以更高效地执行。
- 标准库为list容器定义了更精细的操作集合，使它不必只依赖于泛型操作。

5、算法

容器特有的算法

lst.merge(lst2) lst.merge(lst2,comp)	将 lst2 的元素合并到 lst 中。这两个 list 容器对象都必须排序。lst2 中的元素将被删除。合并后，lst2 为空。分别使用 < 操作符和 comp 函数比较
lst.remove(val)	调用 lst.erase 删除所有等于指定值的元素
lst.reverse()	反向排列 lst 中的元素
lst.sort	对 lst 中的元素排序
lst.splice(iter, lst2)	将 lst2 的元素移到 lst 中迭代器 iter 指向的元素前面；合并后lst2为空，不能是同一个 list
lst.splice(iter,lst2,iter2)	将lst2中iter2 所指向的元素移到lst 的iter 前面，可以是同一个list
lst.splice(iter, beg, end)	将[beg, end)内元素移动到iter 前面，如果iter 也指向这个区间，则不作任何处理
lst.unique() lst.unique(func)	调用 erase 删除同一个值的副本。分别用 == 操作符和函数 func 比较

6、空间管理器

華中科技大學

一般用户使用　new、malloc、delete、free

高级用户：改变内存分配策略，

采用自己定义的策略来实现内存管理

allocator

每种容器中，都隐藏了 allocator，默默完成内存的配置与释放，对象构造与析构的工作。

6、空间管理器

vector 头文件：

```
template <class _Ty, class _Alloc= allocator<_Ty> >
class vector { // varying size array of values
    ……
};
```

6、空间管理器

类型定义（typedef）：

value_type：分配的元素类型

size_type：表示分配大小的类型，通常是std::size_t

difference_type：表示指针差值的类型，通常是std::ptrdiff_t

pointer：指向value_type的指针，即value_type*

const_pointer：指向const value_type的指针

reference：value_type的引用，即value_type&

const_reference：const value_type的引用

propagate_on_container_move_assignment：一个类型，通常为true_type，表示在容器移动赋值时是否传播分配器

rebind：一个模板结构，允许分配器为其他类型分配内存

6、空间管理器

allocator()：默认构造函数

allocator(const allocator&)：复制构造函数

template <class U> allocator(const allocator<U>&)：

从其他类型的分配器复制的构造函数

~allocator()：析构函数

pointer address(reference x) const：返回对象的地址

const_pointer address(const_reference x) const：返回const对象的地址

pointer allocate(size_type n, const void* hint = 0)：

分配内存，足够存储n个value_type对象

void deallocate(pointer p, size_type n)：释放之前分配的内存

size_type max_size() const：返回分配器能分配的最大大小

template <class U, class... Args> void construct(U* p, Args&&... args)：

在已分配的内存p上构造U类型的对象

template <class U> void destroy(U* p)：销毁p指向的对象

VECTOR 示例

- 向量允许在序列末尾插入和删除;
- 若要在矢量中间插入或删除元素，则需要线性时间;
- 增加到超过其当前存储容量时，将进行矢量重新分配;
- 插入和删除均可能改变序列中各个元素的存储地址;
- Deque 类容器在序列的开头和结尾处速度更快;
- 列表类容器在序列内任何位置的插入和删除速度更快。

VECTOR示例

華中科技大學

运算符

v[n]　　：返回第 n 个元素

v1=v2　　：把v1的元素替换为v2元素的副本

v1==v2　：判断两者是否相同

!=、<、<=、>、>=　　：保持这些操作符惯有含义

VECTOR示例

vector容器的构造函数

```
typedef  vector<int, allocator<int>>   INTVECT;
                  // typedef  vector<int>   INTVECT;

INTVECT  v;      //   vector<int>    v;
                 //    vector <Elem> v  ;
                 //    创建一个空的vector。

vector <elem>  v(n);    // 创建一个vector，含有n个数据
                        //   数据均已缺省构造产生。
                        //   vector<int>  v(10);
                        //   INTVECT    v(10);
```

VECTOR示例

vector容器的构造函数

```
vector <Elem> v1(v) ;  //  复制一个vector。

vector <Elem> v(n, e)  ;
                    //  创建一个含有n个e 拷贝的vector。
                    //  INTVECT  v(10, 5);

INTVECT  v3({ 10,20,30,45,-1,-15,25 });

vector <Elem> v(iterator first, iterator last)   ;
                    //  用两个迭代器区间的数据构造
INTVECT  v4(v3.begin( ) + 2,  v3.end( ) - 2);
```

VECTOR 示例


```
typedef vector<int, allocator<int>> INTVECT;
void f()
{
        INTVECT  myVector(10);
        int  i;
        for (i=0;i<10;i++)  myVector[i]=i;
        cout<<myVector.at(5) <<endl;

       INTVECT::iterator   myVectPtr = myVector.begin();

        myVector.insert(myVectPtr,  25);

        myVector.insert(myVector.begin()+3, 2, 28);

                        // 从第三个位置开始，插入 2个 28
}
void main()
{
    f();
}
```

VECTOR 示例


```
INTVECT::iterator  myVectPtr;
for (myVectPtr = myVector.begin(); myVectPtr != myVector.end(); myVectPtr++)
        cout << *myVectPtr << " ";

myVector.pop_back();    // 删除最后一个元素
cout << myVector.capacity() << endl;  // 显示容器大小
cout << myVector.size() << endl;        // 显示实际元素个数

myVector.push_back(15);    // 在末尾处，插入元素 15
myVector[5]=100;              //  将第5个元素改为 100
```

LIST 例子


```
typedef list<int, allocator<int>> INTLIST;

void f()
{
    INTLIST  myList;
    INTLIST::iterator myListPtr;
    int  i;
    for (i=9;i>=0;i--)
            myList.push_front(i);

    for (myListPtr=myList.begin(); myListPtr!=myList.end(); myListPtr++)
            cout<<*myListPtr<<endl;
}

void main()
{
      f();
}
```

头文件

C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Tools\MSVC\14.22.27905\include

<algorithm>、<deque>、<functional>、

<iterator>、<vector>、<list>、

<map>、<memory>、<numeric>、

<queue>、<set>、<stack>、<utility>

总结

理解 STL 的基本组成

容器　迭代器　算法　空间配置器　仿函数

熟悉 主要模板的用法

理解 STL 内部实现的机制

选择适当的模板和调用方法，提高程序运行效率