Contents

[allocator类 1](#_Toc36574204)

[algorithm库函数 3](#_Toc36574205)

[deque容器类知识 4](#_Toc36574206)

[map容器类知识 7](#_Toc36574207)

[queue类知识 10](#_Toc36574208)

[pair类知识 10](#_Toc36574209)

[stack类知识 11](#_Toc36574210)

[stream类 11](#_Toc36574211)

[string类 11](#_Toc36574212)

[tuple类知识 16](#_Toc36574213)

[vector类向量知识 17](#_Toc36574214)

BLUE keywords/function

*ITALIA* argument/parameter

|  |  |  |
| --- | --- | --- |
| <cstdlib>(stdlib.h) | void srand (unsigned int *seed*) | [F] Initialize random number generator for the subsequent calls to rand  {E}srand(time(NULL)) |
| int rand (void) | [F] Generate random integral number between 0 and RAND\_MAX  {E}rand()%100+1 //rand num 1 to100 |
| <ctime>(time.h) | time\_t time (time\_t\* *timer*) | [F] Get current time; if argument is not NULL, it also sets this value to the object pointed by *timer* |
|  | sizeof(X) | 获得以字节为单位的X的尺寸空间 |
| <cstddef>(stddef.h) | offsetof(X,Y) | 获得X类/结构体中，成员Y相对首地址的起始位置，字节为单位 |
| <cstring>(string.h) | void\* memcpy(void \* dst, const void \* src, size\_t num) | 复制num个字节的数据从src到dst，二者都是不限类型的指针或地址 |
|  |  |  |

# allocator类

**1. 简述**

C++的STL中定义了很多容器，容器的第二个模板参数通常为allocator类型，于是想对这一类型做个透彻的了解，看看到底是怎么回事。标准库中allocator类定义在头文件memory中，用于帮助将内存分配和对象的构造分离开来。它分配的内存是原始的、未构造的。和vector等一样，allocator也是一个模板类，为了定义一个allocator对象，我们需指明这个allocator可以分配的对象类型，这样allocator好根据给定的对象类型来确定合适的内存空间大小和对齐位置，例：

allocator<string> alloc; 定义了一个可以分配string的allocator对象

auto const p=alloc.allocate(n); //分配n个未初始化的string内存，即为n个空string分配了内存，当然正如上面所说，分配的内存是原始的，未构造的。

**2. allocator用法概述**

常见操作总结如下：

allocator<T> a // 定义了一个名为a的allocator对象，它可以为类型T的对象分配内存

a.allocate(n) // 分配一段原始的、未构造的内存，这段内存能保存n个类型为T的对象

a.deallocate(p,n) // 释放T\*指针p地址开始的内存，这块内存保存了n个类型为T的对象，p必须是一个先前由allocate返回的指针，且n必须是p创建时所要求的大小，且在调用该函数之前必须销毁在这片内存上创建的对象。要求还蛮多的哈，这是因为在创建过程中我们分配的是最原始的内存，所以在释放内存时也是只能严格释放这片最原始的内存。

a.construct(p,args) // p必须是一个类型为T\* 的指针，指向一片原始内存，arg将被传递给类型为T的构造函数，用来在p指向的原始内存上构建对象。

a.destory(p) // p为T\*类型的指针，用于对p指向的对象执行析构函数

**3. 详情**

1) allocate用于分配原始内存

正如标题所说，allocator出来的内存是最原始的，未构造的内存。相当于开辟新天地，我们将在这片新天地上盖高楼建大厦。它的construct成员函数接受一个指针和零个或多个额外的参数，在给定位置构造对象，额外的参数是用于初始化构造对象的。

auto q=p; //q指向最后构造的元素之后的位置

alloc.construct(q++); //\*q为空字符串

alloc.construct(q++,10,'c'); //\*q为cccccccccc

alloc.construct(q++,"hi"); //\*q为hi

用完对象后，必须对这种构造的的对象调用destory销毁，它接受一个指针，对指向的对象执行析构函数。

while(q!=p)

alloc.destory(--q);

循环开始处，q是指向最后构造的元素之后的一个位置，调用destory之前我们先对q进行递减操作，所以第一次调用destory销毁的是最后一个元素，依次执行销毁操作直到q和p相等。我们只能对真正构造了的元素进行destory操作。一旦元素被销毁，就可以重新使用这部分内存来保存其他string或归还给系统，释放内存通过调用deallocate完成。

alloc.deallocate(p,n);

其中p不能为空，必须指向allocate分配的内存，而且大小参数n也必须与调用allocated分配内存时提供的大小参数相等。

**4. 两个伴随算法**

allocator还有两个伴随算法，用于在未初始化的内存块中创建对象，这些函数在给定目的位置创建元素，而不是由系统分配内存给他们，同样它们也位于头文件memory中。

uninitialized\_copy(b,e,b2) // 从迭代器b和e指出的输入范围中拷贝元素到迭代器b2指定的未构造的原始内存中，b2指向的内存必须足够大，能容纳输入序列中元素的拷贝。

uninitialized\_copy\_n(b,n,b2) // 从迭代器b指向的元素开始，拷贝n个元素到b2开始的内存中

uninitialized\_fill(b,e,t) // 在迭代器b和e指定的原始内存范围中创建对象，对象的值均为t的拷贝

uninitalized\_fiil\_n(b,n,t) // 在迭代器b指向的内存地址开始创建n个对象，b必须指向足够大的未构造的原始内存，能够容乃给定数量的对象

以上函数将返回一个迭代器，指向最后一个构造的元素之后的位置。

**5. 实例**

假定有一个int的vector，希望将它的内容拷贝到动态内存中，我们将分配一块比vector中元素所占空间大一倍的动态内存，然后将原vector中的元素拷贝到前一半空间，后一半用一个给定值进行填充。

//分配比vi向量所占空间大一倍的动态内存

auto p=alloc.allocate(vi.size()\*2);

//通过拷贝vi中的元素来构造从p开始的元素

auto q=uninitialized\_copy(vi.begin(),vi.end(),p);

//将剩余元素初始化为42

uninitialized\_fill\_n(q,vi.size(),42);

**6. 总结**

  为什么会有allocator?

  原因是new在内存分配上面有一些局限性，new的机制是将内存分配和对象构造组合在一起，同样的，delete也是将对象析构和内存释放组合在一起。但当分配一块大块内存时，我们想要自己在这块内存上构建对象，就像建房子，我们弄到一块地，想自己开发更赚钱，或更满足自己的需求，这中情况下我们希望将内存分配和对象构造分离，这样就可实现，我们可以事先得到大块内存，然后真正需要时就在这块内存上创建对象。

# algorithm库函数

**binary\_search**

template <class ForwardIterator, class T, class Compare>

bool binary\_search (ForwardIterator first, ForwardIterator last, const T& val, Compare comp)

// 使用二分搜索在一个范围[first, last)内查找val，注意返回的是bool。comp是可选的比较函数，默认为小于<，注意搜索对象必须按照相同的比较函数排序完毕

**count**

template <class InputIterator, class T>

typename iterator\_traits<InputIterator>::difference\_type

count (InputIterator first, InputIterator last, const T& val)

// 返回在范围[first, last)内的等于val的值的元素个数，typename之后是返回类型

**find**

template <class InputIterator, class T>

InputIterator find (InputIterator first, InputIterator last, const T& val)

// 返回在范围内等于val的元素迭代器，若查找失败，返回last

**for\_each**

template <class InputIterator, class Function>

Function for\_each (InputIterator first, InputIterator last, Function fn);

// 对范围内所有元素使用fn函数对象，并返回move(fn)

**max\_element**

template <class ForwardIterator, class Compare>

ForwardIterator max\_element (ForwardIterator first, ForwardIterator last, Compare comp);

// 返回范围内最大值的迭代器

**min\_element**

// 类似上例

**nth\_element**

template <class RandomAccessIterator, class Compare>

void nth\_element (RandomAccessIterator first, RandomAccessIterator nth,

RandomAccessIterator last, Compare comp);

// 单元素部分排序或n排序或n选取算法，n位置上的元素即为排序好后的元素

**random\_shuffle**

template <class RandomAccessIterator, class RandomNumberGenerator>

void random\_shuffle (RandomAccessIterator first, RandomAccessIterator last, RandomNumberGenerator& gen);

// 以逐个元素遍历并交换的方式打乱范围内的元素，gen是可选的随机数生成器，返回的随机数用于确定哪个位置的元素用来和正在被遍历的元素交换

**remove**

template <class ForwardIterator, class T>

ForwardIterator remove (ForwardIterator first, ForwardIterator last, const T& val);

// 删除值等于val的元素，实际上是将这些元素置于容器后部，然后返回一个新的last迭代器标识所剩元素的结尾位置。

**remove\_if**

template <class ForwardIterator, class UnaryPredicate>

ForwardIterator remove\_if (ForwardIterator first, ForwardIterator last, UnaryPredicate pred);

// 类似上例，但适用于对于除了比较数值外的更复杂的逻辑判定，pred函数对象返回true的元素被删除

**sort**

template <class RandomAccessIterator, class Compare>

void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);

// 对范围内的元素排序，comp是可选的排序逻辑函数对象，默认为第一个参数小于第二个参数时返回true

**swap**

template <class T>

void swap (T& a, T& b);

// 交换二者的值

# deque容器类知识

C++ STL容器deque和vector很类似，也是采用动态数组来管理元素。

使用deque之前需包含头文件：

*#include <deque>*

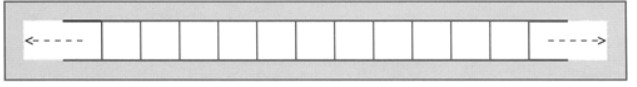
它是定义在命名空间std内的一个class template：

*template<class \_Ty,  
class \_Ax = allocator<\_Ty> >  
class deque;*

第一个template参数用来表示元素型别，第二个可有可无，指定内存模型。一般使用默认的内存模型。

与vector不同的是deque的动态数组首尾都开放，因此能够在首尾进行快速地插入和删除操作。

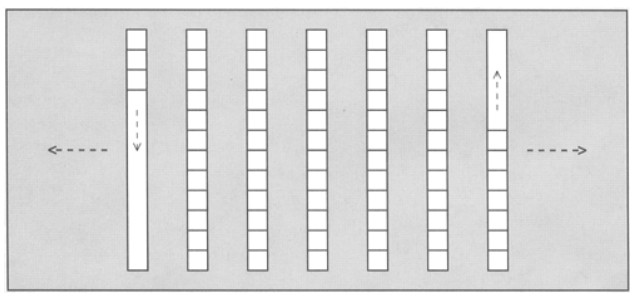
**deque的逻辑结构：**



**deque的内部结构**

deque是一种优化了的对序列两端元素进行添加和删除操作的基本序列容器。通常由一些独立的区块组成，第一区块朝某方向扩展，最后一个区块朝另一方向扩展。它允许较为快速地随机访问但它不像vector一样把所有对象保存在一个连续的内存块，而是多个连续的内存块。并且在一个映射结构中保存对这些块以及顺序的跟踪。

其内部结构如下图所示：



**deque的特点：**

1、支持随机访问，即支持[]以及at()，但是性能没有vector好。

2、可以在内部进行插入和删除操作，但性能不及list。

**deque和vector的不同之处：**

1、两端都能够快速插入和删除元素。vector只能在尾端进行。

2、deque的元素存取和迭代器操作会稍微慢一些。因为deque的内部结构会多一个间接过程。

3、迭代器是特殊的智能指针，而不是一般指针。它需要在不同的区块之间跳转。

4、deque可以包含更多的元素，其max\_size可能更大。因为不止使用一块内存。

5、不支持对容量和内存分配时机的控制。

注意：在除了首尾两端的其他地方插入和删除元素，都将会导致指向deque元素的任何pointers、references、iterators失效。不过，deque的内存重分配优于vector。因为其内部结构显示不需要复制所有元素。

6、deque的内存区块不再被使用时，会被释放。deque的内存大小是可缩减的。不过，是不是这么做以及怎么做由实作版本定义。

**deque和vector相似的特性：**

1、在中间部分插入和删除元素相对较慢，因为所有元素都要被移动。

2、迭代器属于随即存取迭代器。

**最好采用deque的情形：**

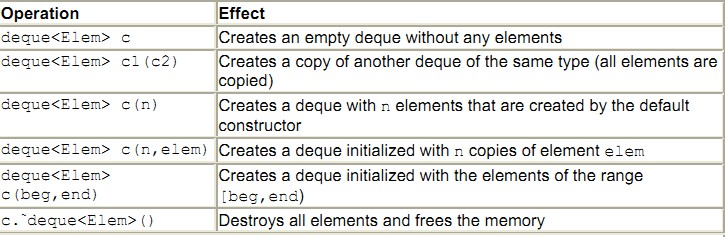
1、需要在两端插入和删除元素。

2、无需引用容器内的元素。

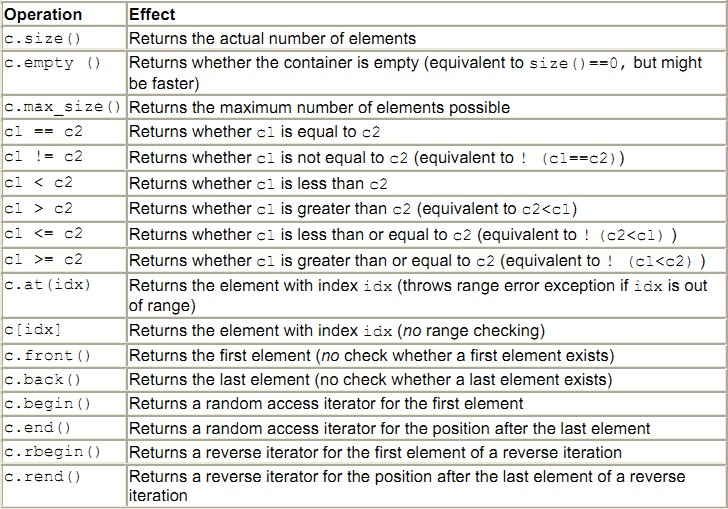
3、要求容器释放不再使用的元素。

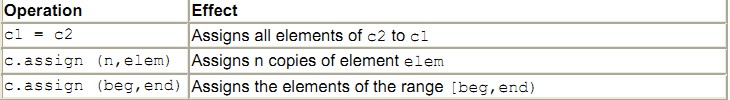
**deque的操作函数**

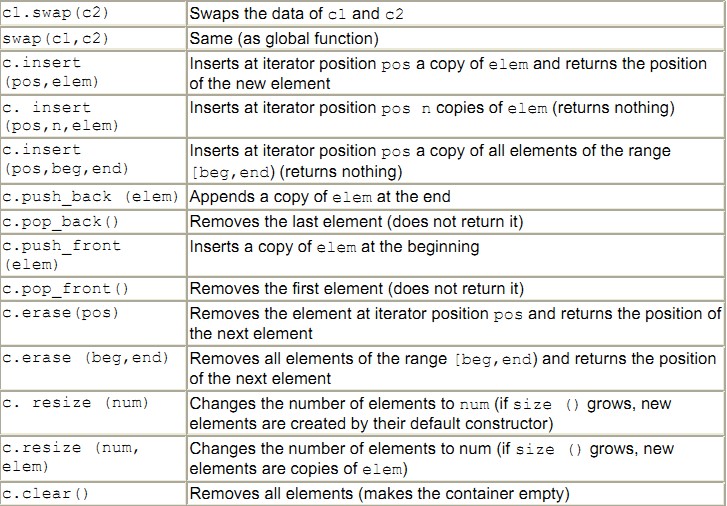
**构造函数和析构函数：**



**非变动性操作：**

  
**变动性操作：**



  
deque的各项操作只有一下两点和vector不同：

deque不提供容量操作：capacity()和reverse()。

deque直接提供函数完成首尾元素的插入和删除。

其他均与vector相同。

注意：

1、除了at()函数，其他成员函数都不会检查索引或迭代器是否有效。

2、元素的插入和删除可能会导致内存重新分配。所以任何插入或删除操作都会使所有指向deque元素的pointers、reference、iterators失效。唯一例外的是在首尾插入元素之后，pointers和reference可能仍然有效。

# map容器类知识

**1. map简介**

map是一类关联式容器，它是模板类。关联的本质在于元素的值与某个特定的键相关联，而并非通过元素在数组中的位置类获取。它的特点是增加和删除节点对迭代器的影响很小，除了操作节点，对其他的节点都没有什么影响。对于迭代器来说，不可以修改键值，只能修改其对应的实值。

默认情况下，map中的元素按照key的一定顺序自动进行排列，可以通过设置特殊的函数或类来改变排列规则，但是必须仍然是针对key进行排序。如果想使用不自动排序的散列表，应使用unordered\_map。

**2. map的功能**

①自动建立Key － value的对应。key 和 value可以是任意你需要的类型，但是需要注意的是对于key的类型，唯一的约束就是必须支持 “<” 操作符。

②根据key值快速查找记录，查找的复杂度基本是Log(N)，如果有1000个记录，最多查找10次，1,000,000个记录，最多查找20次。

③快速插入Key - Value 记录。

④快速删除记录

⑤根据Key 修改value记录。

⑥遍历所有记录。

**3. map的定义**

使用map得包含map类所在的头文件：#include <map> //注意，STL头文件没有扩展名.h

map对象是模板类，需要关键字和存储对象两个模板参数，基本的定义模式如下：

std:map<int, string> personnel;

这样就定义了一个以int为键，值为string的map对象personnel。

map中定义了以下三个类型：

map<K, V>::key\_type : 表示map容器中，索引的类型；

map<K, V>::mapped\_type : 表示map容器中，键所关联的值的类型；

map<K, V>::value\_type : 表示一个pair类型，它的first元素具有const map<K, V>::key\_type类型，而second元素则有map<K, V>::mapped\_type类型

对迭代器进行解引用 “\*” 时，将获得一个引用，指向容器中一个value\_type类型的值，对于map容器，其value\_type是pair类型。

为了使用方便，可以对模板类进行一下类型定义:

typedef map<int, CString> UDT\_MAP\_INT\_CSTRING;

UDT\_MAP\_INT\_CSTRING enumMap;

**4. 在map中添加元素**

给map中添加元素主要有两种方法：

①使用下标操作符获取元素，然后给元素赋值

For example:

map<string, int> word\_count; // 定义了一个空的map对象word\_count;

word\_count["Anna"] = 1;

程序说明：

1）在word\_count中查找键为Anna的元素，没有找到.

2） 将一个新的键-值对插入到word\_count中，它的键是const string类型的对象，保存Anna。 而它的值则采用直初始化，这就意味着在本例中指为0.

3）将这个新的键-值对插入到word\_count中

4）读取新插入的元素，并将它的值赋为1.

**※注意！**使用下标访问map与使用下标访问数组或者vector的行为是截然不同的：使用下 标访问不存在的元素将导致在map容器中添加一个新的元素，他的键即为该下标值。

②使用map::insert方法添加元素

map容器提供的insert操作：

1）map.insert(e) : e是一个用在map中的value\_type类型的值。如果键不存在，则插入一个 值为e.second的新元素；如果键在map中已经存在，那么不进行任何操作。该函数返 回一个pair类型，该pair类型的first元素为当前插入e的map迭代器，pair的second 类型是一个bool类型，表示是否插入了该元素。

2）map.insert(beg, end) : beg和end是迭代器，返回void类型

3）map.insert(iter, e) : e是value\_type类型的值，如果e.first不在map中，则创建新元素，并 以迭代器iter为起点搜索新元素存储的位置，返回一个迭代器，指向map中具有给定 键的元素。

For example:

word\_count.insert(map<sting, int>::value\_type("Anna", 1));

word\_count.insert(make\_pair("Anna", 1));

返回值：如果该键已在容器中，则其关联的值保持不变，返回的bool值为true。

**5. 查找并获取map中的元素**

使用下标获取元素存在一个很危险的副作用：如果该键不在map容器中，那么下标操作会插入一个具有该键的新元素。

因此引入map对象的查询操作：

1） map.count(k) : 返回map中键k的出现次数(对于map而言，由于一个key对应一个value，因此返回只有0和1，因此可以用此函数判断k是否在map中)

2） map.find(k) : 返回map中指向键k的迭代器，如果不存在键k，则返回超出末端迭代器。

For example:

int occurs = 0;

if( word\_count.cout("foobar") )

occurs = word\_count["foobar"];

int occurs = 0;

map<string, int>::iterator it = word\_count.find("foobar");

if( it != word\_count.end() )

occurs = it ->second;

**6. 从map中删除元素**

移除某个map中某个条目用erase()

该成员方法的定义如下:

iterator erase(iterator it); //通过一个条目对象删除

iterator erase(iterator first, iterator last); //删除一个范围

size\_type erase(const Key& key); //通过关键字删除

**7. map对象的迭代遍历**

与其他容器一样，map同样提供begin和end运算，以生成用于遍历整个容器的迭代器。但是迭代器的种类不同。在STL定义的容器中，string，vector与deque提供了随机访问迭代器，list、set、multiset、map、multimap提供了双向迭代器。有以下区别：

①所有迭代器都应该实现自增算符：iter++,++iter

②Bidirectional迭代器：是在前向迭代器的基础上，多了单步向后遍历的能力。也就是--iter,iter--。

③Random Access迭代器：在双向迭代器基础上，具有直接访问各数据元素的能力。随机迭代器增加了“迭代器算术运算”：

iter+=i 迭代器递增i位

iter-=i 迭代器递减i位

iter+i 加i位后的迭代器

iter-i 减i位后的迭代器

iter[i] 加i位后的迭代器的解引用

iter<iter1 如果迭代器iter的位置在iter1前，返回true，否则返回false

iter<=iter1 如果iter的位置在iter1的前面或同一位置时返回true，否则返回false

iter>iter1 如果迭代器iter的位置在iter1后，返回true，否则返回false

iter>=iter1 如果iter的位置在iter1的后面或同一位置时返回true，否则返回false

# queue类知识

**1.成员函数**

empty()

size()

front() // 访问下一元素

back() // 访问队末元素

push(val) // 插入队末元素

template <class… Args> void emplace(Args&&… args)

// 插入队末元素，参数赋予元素的构造函数

void pop() 弹出下一元素，注意不返回该元素

void swap(queue& x) 和x队列交换内容

# pair类知识

**1. pair的应用**

template <class T1, class T2> struct pair

pair是一个类模板，确切的说是结构体模板，将2个数据组合成一个数据，当需要这样的需求时就可以使用pair，如STL中的map就是将key和value放在一起来保存。另一个应用是，当一个函数需要返回2个数据的时候，可以选择pair。 pair的实现是一个结构体，主要的两个成员变量是first 和second 。

**2. make\_pair函数**

template pair make\_pair(T1 a, T2 b) { return pair(a, b); }

很明显，我们可以使用pair的构造函数也可以使用make\_pair来生成我们需要的pair。 一般make\_pair都使用在需要pair做参数的位置，可以直接调用make\_pair生成pair对象很方便，代码也很清晰。另一个使用的方面就是pair可以接受隐式的类型转换，这样可以获得更高的灵活度。灵活度也带来了一些问题如：

std::pair<int, float>(1, 1.1);

std::make\_pair(1, 1.1);

是不同的，第一个就是float，而第2个会自己匹配成double。

**3. 具体用法：**

①定义（构造）：

1 pair<int, double> p1; //使用默认构造函数

2 pair<int, double> p2(1, 2.4); //用给定值初始化

3 pair<int, double> p3(p2); //拷贝构造函数

②访问两个元素（通过first和second）：

1 pair<int, double> p1; //使用默认构造函数

2 p1.first = 1;

3 p1.second = 2.5;

4 cout << p1.first << ' ' << p1.second << endl;

输出结果：1 2.5

③赋值operator = ：

（1）利用make\_pair：

1 pair<int, double> p1;

2 p1 = make\_pair(1, 1.2);

（2）变量间赋值：

pair<int, double> p1(1, 1.2);

pair<int, double> p2 = p1;

# stack类知识

**1. 成员函数**

empty()

size()

top() // 访问栈顶元素

push(x) // 压入元素至栈顶

emplace(…) // 压入元素至栈顶，参数用于初始化元素的构造函数，详见queue中同名函数

void pop() // 弹出栈顶元素，不返回它

void swap(stack& x) // 交换本栈和x栈的内容

# stream类

**0. stream类通用**

①成员函数：

•streambuf\* rdbuf() cosnt

返回当前流（缓存）对象的指针，该指针可以用来将当前流对象的所有内容通过流插入 运算符<<，插入到另一个流对象中

Example：

std::ifstream vertexShaderFile(vShaderFile);

std::stringstream vShaderStream;

vShaderStream << vertexShaderFile.rdbuf(); //将文件流内容插入到字符串流中

•streambuf\* rdbuf (streambuf\* sb)

将sb所指的流对象设为当前流对象的内容

**1.stringstream类**

①成员函数：

•string str()：返回一个string类对象，内容是现在stream中的内容

**2. fstream类**

详见C++ OOP学习笔记

# string类

要想使用标准C++中string类，必须要包含#include <string>// 注意是<string>，不是<string.h>，带.h的是C语言中的头文件

using std::string;

using std::wstring;或

using namespace std;

下面你就可以使用string/wstring了，它们两分别对应着char和wchar\_t。

string和wstring的用法是一样的，以下只用string作介绍：

**0.string类的内存分配：**

string类对象初始会占据28字节的空间，小于等于15个字符的内容时，字符内容存储于该28字节空间内。超过15个字符后，所有字符串内容转移至一片更大的区域存储，然而原28字节仍属该string类对象，且首地址（string对象的指针）不变。

**1.string类的构造函数：**

string(const char \*s); //用c字符串s初始化

string(int n,char c); //用n个字符c初始化

此外，string类还支持默认构造函数和复制构造函数，如string s1；string s2="hello"；都是正确的写法。当构造的string太长而无法表达时会抛出length\_error异常 ；

**2.string类的字符操作：**

const char &operator[](int n)const;

const char &at(int n)const;

char &operator[](int n);

char &at(int n);

operator[]和at()均返回当前字符串中第n个字符的位置，但at函数提供范围检查，当越界时会抛出out\_of\_range异常，下标运算符[]不提供检查访问。

const char \*data()const;//返回一个非null终止的c字符数组

const char \*c\_str()const;//返回一个以null终止的c字符串

int copy(char \*s, int n, int pos = 0) const;//把当前串中以pos开始的n个字符拷贝到以s为起始位置的字符数组中，返回实际拷贝的数目

**3.string的特性描述:**

int capacity()const; //返回当前容量（即string中不必增加内存即可存放的元素个数）

int max\_size()const; //返回string对象中可存放的最大字符串的长度

int size()const; //返回当前字符串的大小

int length()const; //返回当前字符串的长度

bool empty()const; //当前字符串是否为空

void resize(int len,char c);//把字符串当前大小置为len，并用字符c填充不足的部分

**4.string类的输入输出操作:**

string类重载运算符operator>>用于输入，同样重载运算符operator<<用于输出操作。

函数istream& getline (istream& is, string& str, char delim = ‘\n’)，从is对象中提取出一行字符串，或者是delim决定的字符处为止，存储至str中

**5.string的赋值：**

string &operator=(const string &s);//把字符串s赋给当前字符串

string &assign(const char \*s);//用c类型字符串s赋值

string &assign(const char \*s,int n);//用c字符串s开始的n个字符赋值

string &assign(const string &s);//把字符串s赋给当前字符串

string &assign(int n,char c);//用n个字符c赋值给当前字符串

string &assign(const string &s,int start,int n);//把字符串s中从start开始的n个字符赋给当前字符串

string &assign(const\_iterator first,const\_itertor last);//把first和last迭代器之间的部分赋给字符串

**6.string的连接：**

string &operator+=(const string &s);//把字符串s连接到当前字符串的结尾

string &append(const char \*s); //把c类型字符串s连接到当前字符串结尾

string &append(const char \*s,int n);//把c类型字符串s的前n个字符连接到当前字符串结尾

string &append(const string &s); //同operator+=()

string &append(const string &s,int pos,int n);//把字符串s中从pos开始的n个字符连接到当前字符串的结尾

string &append(int n,char c); //在当前字符串结尾添加n个字符c

string &append(const\_iterator first,const\_iterator last);//把迭代器first和last之间的部分连接到当前字符串的结尾

**7.string的比较：**

bool operator==(const string &s1,const string &s2)const;//比较两个字符串是否相等

运算符">","<",">=","<=","!="均被重载用于字符串的比较；

int compare(const string &s) const;//比较当前字符串和s的大小

int compare(int pos, int n,const string &s)const;//比较当前字符串从pos开始的n个字符组成的字符串与s的大小

int compare(int pos, int n,const string &s,int pos2,int n2)const;//比较当前字符串从pos开始的n个字符组成的字符串与s中pos2开始的n2个字符组成的字符串的大小

int compare(const char \*s) const;

int compare(int pos, int n,const char \*s) const;

int compare(int pos, int n,const char \*s, int pos2) const;

compare函数在>时返回1，<时返回-1，==时返回0

**8.string的子串：**

string substr(size\_t pos = 0,size\_t len = npos) const;// 返回一个子字符串substring，注意不等于裁剪母字符串，母字符串未变；pos代表起始位置（首字符位置为0），默认为0；len代表substring长度，默认为npos（取值为-1，代表直至末尾），size\_t即无符号整型

**9.string的交换：**

void swap(string &s2); //交换当前字符串与s2的值

**10.string类的查找函数：**

int find(char c, int pos = 0) const;//从pos开始查找字符c在当前字符串中第一次出现的位置，结果返回一个整数，代表被搜索字符（串）在搜索范围（即母对象）内的起始位置

int find(const char \*s, int pos = 0) const;//从pos开始查找字符串s在当前串中的位置

int find(const char \*s, int pos, int n) const;//从pos开始查找字符串s中前n个字符在当前串中的位置

int find(const string &s, int pos = 0) const;//从pos开始查找字符串s在当前串中的位置

//查找成功时返回所在位置，失败返回string::npos的值

int rfind(char c, int pos = npos) const;//从pos开始从后向前查找字符c在当前串中的位置

int rfind(const char \*s, int pos = npos) const;

int rfind(const char \*s, int pos, int n = npos) const;

int rfind(const string &s,int pos = npos) const;

//从pos开始从后向前查找字符串s中前n个字符组成的字符串在当前串中的位置，成功返回所在位置，失败时返回string::npos的值

int find\_first\_of(char c, int pos = 0) const;//从pos开始查找字符c第一次出现的位置

int find\_first\_of(const char \*s, int pos = 0) const;

int find\_first\_of(const char \*s, int pos, int n) const;

int find\_first\_of(const string &s,int pos = 0) const;

//从pos开始查找当前串中第一个在s的前n个字符组成的数组里的字符的位置。查找失败返回string::npos

int find\_first\_not\_of(char c, int pos = 0) const;

int find\_first\_not\_of(const char \*s, int pos = 0) const;

int find\_first\_not\_of(const char \*s, int pos,int n) const;

int find\_first\_not\_of(const string &s,int pos = 0) const;

//从当前串中查找第一个不在串s中的字符出现的位置，失败返回string::npos

int find\_last\_of(char c, int pos = npos) const;

int find\_last\_of(const char \*s, int pos = npos) const;

int find\_last\_of(const char \*s, int pos, int n = npos) const;

int find\_last\_of(const string &s,int pos = npos) const;

int find\_last\_not\_of(char c, int pos = npos) const;

int find\_last\_not\_of(const char \*s, int pos = npos) const;

int find\_last\_not\_of(const char \*s, int pos, int n) const;

int find\_last\_not\_of(const string &s,int pos = npos) const;

//find\_last\_of和find\_last\_not\_of与find\_first\_of和find\_first\_not\_of相似，只不过是从后向前查找

**11.string类的替换函数：**

string &replace(int p0, int n0,const char \*s);//删除从p0开始的n0个字符，然后在p0处插入串s

string &replace(int p0, int n0,const char \*s, int n);//删除p0开始的n0个字符，然后在p0处插入字符串s的前n个字符

string &replace(int p0, int n0,const string &s);//删除从p0开始的n0个字符，然后在p0处插入串s

string &replace(int p0, int n0,const string &s, int pos, int n);//删除p0开始的n0个字符，然后在p0处插入串s中从pos开始的n个字符

string &replace(int p0, int n0,int n, char c);//删除p0开始的n0个字符，然后在p0处插入n个字符c

string &replace(iterator first0, iterator last0,const char \*s);//把[first0，last0）之间的部分替换为字符串s

string &replace(iterator first0, iterator last0,const char \*s, int n);//把[first0，last0）之间的部分替换为s的前n个字符

string &replace(iterator first0, iterator last0,const string &s);//把[first0，last0）之间的部分替换为串s

string &replace(iterator first0, iterator last0,int n, char c);//把[first0，last0）之间的部分替换为n个字符c

string &replace(iterator first0, iterator last0,const\_iterator first, const\_iterator last);//把[first0，last0）之间的部分替换成[first，last）之间的字符串

**12.string类的插入函数：**

string &insert(int p0, const char \*s);

string &insert(int p0, const char \*s, int n);

string &insert(int p0,const string &s);

string &insert(int p0,const string &s, int pos, int n);

//前4个函数在p0位置插入字符串s中pos开始的前n个字符

string &insert(int p0, int n, char c);//此函数在p0处插入n个字符c

iterator insert(iterator it, char c);//在it处插入字符c，返回插入后迭代器的位置

void insert(iterator it, const\_iterator first, const\_iterator last);//在it处插入[first，last）之间的字符

void insert(iterator it, int n, char c);//在it处插入n个字符c

**13.string类的删除函数**

iterator erase(iterator first, iterator last);//删除[first，last）之间的所有字符，返回删除后迭代器的位置

iterator erase(iterator it);//删除it指向的字符，返回删除后迭代器的位置

string &erase(int pos = 0, int len = npos);//删除pos开始的n个字符，返回修改后的字符串

**14.string类的迭代器处理：**

string类提供了向前和向后遍历的迭代器iterator，迭代器提供了访问各个字符的语法，类似于指针操作，迭代器不检查范围。

用string::iterator或string::const\_iterator声明迭代器变量，const\_iterator不允许改变迭代的内容。常用迭代器函数有：

const\_iterator begin()const;

iterator begin(); //返回string的起始位置

const\_iterator end()const;

iterator end(); //返回string的最后一个字符后面的位置

const\_iterator rbegin()const;

iterator rbegin(); //返回string的最后一个字符的位置

const\_iterator rend()const;

iterator rend(); //返回string第一个字符位置的前面

rbegin和rend用于从后向前的迭代访问，通过设置迭代器string::reverse\_iterator, string::const\_reverse\_iterator实现

**15.字符串流处理：**

通过定义ostringstream和istringstream变量实现，#include <sstream>头文件中

例如：

string input("hello,this is a test");

istringstream is(input);

string s1,s2,s3,s4;

is>>s1>>s2>>s3>>s4;//s1="hello,this",s2="is",s3="a",s4="test"

ostringstream os;

os<<s1<<s2<<s3<<s4;

cout<<os.str();

**16.字符串和数据之间的转换：**

<string>头文件中定义了一些可以直接使用的转换函数，注意它们不是string类的成员函数。且c++11标准后才可使用。

·Convert from strings

int stoi(const string& str, size\_t\* idx = nullptr, int base = 10) //用于将str转换为int，idx用以返回str中被转换的字符串之后的首位置（非特殊条件下不常用），base用于设置字符串采用的数制，如果设0则由函数根据字符串特征自行决定。

stol //Convert string to long int

stoul //Convert string to unsigned integer

stoll //Convert string to long long

stoull //Convert string to unsigned long long

stof //Convert string to float

stod //Convert string to double

stold //Convert string to long double

·Convert to strings

to\_string(X) //Convert numerical value X to string

to\_wstring(X) //Convert numerical value X to wide string

# tuple类知识

tuple容器(元组), 是表示元组容器, 是不包含任何结构的,快速而低质(粗制滥造, quick and dirty)的, 可以用于函数返回多个返回值;

tuple容器, 可以使用直接初始化, 和"make\_tuple()"初始化, 访问元素使用"get<>()"方法, 注意get里面的位置信息, 必须是常量表达式(const expression);

可以通过"std::tuple\_size<decltype(t)>::value"获取元素数量; "std::tuple\_element<0, decltype(t)>::type"获取元素类型;

如果tuple类型进行比较, 则需要保持元素数量相同, 类型可以比较, 如相同类型, 或可以相互转换类型(int&double);

无法通过普通的方法遍历tuple容器, 因为"get<>()"方法, 无法使用变量获取值;

以下代码包含一些基本的用法, 详见注释;

/\*

\* CppPrimer.cpp

\*

\* Created on: 2013.12.9

\* Author: Caroline

\*/

/\*eclipse cdt, gcc 4.8.1\*/

#include <iostream>

#include <vector>

#include <string>

#include <tuple>

using namespace std;

std::tuple<std::string, int>

giveName(void) {

std::string cw("Caroline");

int a(2013);

std::tuple<std::string, int> t = std::make\_tuple(cw, a);

return t; }

int main() {

std::tuple<int, double, std::string> t(64, 128.0, "Caroline");

std::tuple<std::string, std::string, int> t2 =

std::make\_tuple("Caroline", "Wendy", 1992);

//返回元素个数

size\_t num = std::tuple\_size<decltype(t)>::value;

std::cout << "num = " << num << std::endl;

//获取第1个值的元素类型

std::tuple\_element<1, decltype(t)>::type cnt = std::get<1>(t);

std::cout << "cnt = " << cnt << std::endl;

//比较

std::tuple<int, int> ti(24, 48);

std::tuple<double, double> td(28.0, 56.0);

bool b = (ti < td);

std::cout << "b = " << b << std::endl;

//tuple作为返回值

auto a = giveName();

std::cout << "name: " << get<0>(a)

<< " years: " << get<1>(a) << std::endl;

return 0; }

输出：

num = 3

cnt = 128

b = 1

name: Caroline years: 2013

# vector类向量知识

**一、向量的介绍**

向量 vector 是一种对象实体, 能够容纳许多其他类型相同的元素, 因此又被称为容器。 与string相同, vector 同属于STL(Standard Template Library, 标准模板库)中的一种自定义的数据类型, 可以广义上认为是数组的增强版。

在使用它时, 需要包含头文件 vector, #include<vector>

vector 容器与数组相比其优点在于它能够根据需要随时自动调整自身的大小以便容下所要放入的元素。此外, vector 也提供了许多的方法来对自身进行操作。

**二、向量的声明及初始化**

vector 型变量的声明以及初始化的形式也有许多, 常用的有以下几种形式:

vector<int> a //声明一个int型向量a

vector<int> a(10) //声明一个初始size为10的向量，每个元素已被默认初始化

vector<int> a(10, 1) //声明一个初始大小为10且初始值都为1的向量

vector<int> b(a) //声明并用向量a初始化向量b

vector<int> b(a.begin(), a.begin()+3)

//将a向量中从第0个到第2个(共3个)作为向量b的初始值

除此之外, 还可以直接使用数组来初始化向量:

int n[] = {1, 2, 3, 4, 5}

vector<int> a(n, n+5) //将数组n的前5个元素作为向量a的初值

vector<int> a(&n[1], &n[4]) //将n[1] - n[4]范围内的元素作为向量a的初值

**三、元素的输入及访问**

元素的输入和访问可以像操作普通的数组那样, 用cin>>进行输入, cout<<a[n]这样进行输出。在元素的输出上, 还可以使用遍历器(又称迭代器)进行输出控制。在 vector<int> b(a.begin(), a.begin()+3) ; 这种声明形式中, (a.begin()、a.begin()+3) 表示向量起始元素位置到起始元素+3之间的元素位置。(a.begin(), a.end())则表示起始元素和最后一个元素之外的元素位置。向量元素的位置便成为遍历器, 同时, 向量元素的位置也是一种数据类型, 在向量中遍历器的类型为: vector<int>::iterator。 遍历器不但表示元素位置, 还可以再容器中前后移动。

例：

vector<int>::iterator t ;

for(t=a.begin(); t!=a.end(); t++)

cout<<\*t<<" " ;

\*t 为指针的间接访问形式, 意思是访问t所指向的元素值。

**四、向量的基本操作**

1. a.size() //获取向量中的元素个数

2. a.empty() //判断向量是否为空

3. a.clear() //清空向量中的元素

4. 复制

a = b ; //将b向量复制到a向量中

5. 比较

保持 ==、!=、>、>=、<、<= 的惯有含义 ;

如: a == b ; //a向量与b向量比较, 相等则返回1

6. 插入 - insert

①a.insert(a.begin(), 1000); //将1000插入到向量a的起始位置前

②a.insert(a.begin(), 3, 1000) //将1000分别插入到向量元素位置的0-2处(共 3个元素)

③vector<int> a(5, 1) ;

vector<int> b(10) ;

b.insert(b.begin(),a.begin(),a.end()) //将a.begin(), a.end()之间的全部元素插入到b.begin()前

7. 删除 - erase

①b.erase(b.begin()) ; //将起始位置的元素删除

②b.erase(b.begin(), b.begin()+3) ; //将(b.begin(), b.begin()+3)之间的元素删除

8.交换 - swap

b.swap(a) ; //a向量与b向量进行交换

**五、二维向量**

与数组相同, 向量也可以增加维数, 例如声明一个m\*n大小的二维向量方式可以像如下形式:

vector< vector<int> > b(10, vector<int>(5)); //创建一个10\*5的int型二维向量

在这里, 实际上创建的是一个向量中元素为向量的向量。同样可以根据一维向量的相关特性对二维向量进行操作。

**六、迭代器（iterator）**

迭代器的作用类似于指针，即有指向性的一个实体，一般指向的都是容器内的对象；在实际应用中，不同类型的容器（vector，list等）有不同的迭代器，因此迭代器是这些类（模板）中的一个类型（type）。

迭代器可以使用的运算符有：\*（dereference），++（increment），使用方法也类似指针。其他细节见STL:map中的迭代器知识。

举例：

//定义了一个向量类的迭代器，指向向量vec的第一个元素

vector<int>::iterator v = vec.begin();

迭代器经常使用在类模板自带的成员函数中，对成员函数的总结见下：

**七、其他成员函数**

T& front() //返回容器中第一个元素的引用

T& back() //返回容器中最后一个元素的引用，注意不是end()迭代器，且容器不能为空

void push\_back(X) //在vector的最后增加一个元素，X为新元素

void pop\_back() //删除最后一个元素，注意不返回该元素

iterator begin() //返回指向首元素的迭代器

iterator end() //返回指向尾元素之后位置的迭代器，注意不是指向尾元素

void reserve(X) //设置容量capacity为X，注意只能设置得比当前的更大

int capacity() //返回当前总容量，注意容量不是size（实际存储元素个数）

void shrink\_to\_fit() //缩减容量至和size吻合