1. STL 概论

长久以来，软件界一直希望建立一种可重复利用的东西，以及一种得以制造出“可重复运用的东西”的方法，函数（functions）、类别（classes）、函数库（function libraries）、类别库（class libraries）、各种组件、模块化设计、面向对象（object oriented），为的就是复用性的提升。

复用性必须建立在某种标准之上。但是在许多环境下，就连软件开发最基本的数据结构（data structures）和算法（algorithm）都未能有一套标准。大量程序员被迫从事大量重复的工作，竟然是为了完成前人已经完成而自己手上并未拥有的程序代码，这不仅是人力资源的浪费，也是挫折与痛苦的来源。

为了建立数据结构和算法的一套标准，并且降低他们之间的耦合关系，以提升各自的独立性、弹性、交互操作性(相互合作性 interoperability)，诞生了 STL。

1.1 STL 基本概念

STL（Standard Template Library 标准模板库），是惠普实验室开发的一系列软件的统称。现在主要出现在 C++ 中，但是在引入 C++ 之前该技术已经存在很长时间了。

STL 从广义上分为: 容器（container）、算法（algorithm）、迭代器（iterator），容器和算法之间通过迭代器进行无缝连接。STL 几乎所有的代码都采用了模板类或者模板函数，这相比传统的由函数和类组成的库来说提供了更好的代码重用机会。

1.2 STL 六大组件简介

STL 提供了六大组件，彼此之间可以组合套用，这六大组件分别是：容器、算法、迭代

器、仿函数、适配器（配接器）、空间配置器。

容器：各种数据结构，如 vector、list、deque、set、map 等，用来存放数据，从实现角度来看，STL 容器是一种 class template

算法：各种常用的算法，如 sort、find、copy、for_each 等，从实现的角度来看，STL 算法是一种 function tempalte

迭代器：扮演了容器与算法之间的胶合剂，共有五种类型，从实现角度来看，迭代器是一种将 operator* ，operator-> ，operator++，operator-- 等指针相关操作予以重载的 class template。所有 STL 容器都附带有自己专属的迭代器，只有容器的设计者才知道如何遍历自己的元素。原生指针（native pointer）也是一种迭代器。

仿函数：行为类似函数，可作为算法的某种策略。从实现角度来看，仿函数是一种重载了 operator() 的 class 或者 class template

适配器：用来修饰容器或者仿函数或迭代器接口

空间配置器：负责空间的配置与管理。从实现角度看，配置器是一个实现了动态空间配置、空间管理、空间释放的 class tempalte

STL 六大组件的交互关系，容器通过空间配置器取得数据存储空间，算法通过迭代器存储容器中的内容，仿函数可以协助算法完成不同的策略的变化，适配器可以修饰仿函数。

1.3 STL 优点

- STL 是 C++ 的一部分，因此不用额外安装什么，它被内建在你的编译器之内
- STL 的一个重要特性是将数据和操作分离。数据由容器类别加以管理，操作则由可定制的算法定义。迭代器在两者之间充当“粘合剂”,以使算法可以和容器交互运作
- 程序员可以不用思考 STL 具体的实现过程，只要能够熟练使用 STL 就 OK

了。这样就可以把精力放在程序开发的别的方面。

- STL 具有高可重用性，高性能，高移植性，跨平台的优点

高可重用性：STL 中几乎所有的代码都采用了模板类和模版函数的方式实现，这相比于传统的由函数和类组成的库来说提供了更好的代码重用机会。

高性能：如 map 可以高效地从十万条记录里面查找出指定的记录，因为 map 是采用红黑树的变体实现的。

高移植性：如在项目 A 上用 STL 编写的模块，可以直接移植到项目 B 上。

2. STL 三大组件

2.1 容器

容器，置物之所也。研究数据的特定排列方式，以利于搜索或排序或其他特殊目的，这一门学科我们称为数据结构。任何特定的数据结构都是为了实现某种特定的算法。STL 容器就是将运用最广泛的一些数据结构实现出来。

常用的数据结构：数组（array）、链表（list）、树（tree）、栈（stack）、队列（queue）、集合（set）、映射表（map），根据数据在容器中的排列特性，这些数据分为序列式容器和关联式容器两种。

- 序列式容器强调值的排序，序列式容器中的每个元素均有固定的位置，除非用删除或插入的操作改变这个位置。vector 容器、deque 容器、list 容器等。
- 关联式容器是非线性的树结构，更准确的说是二叉树结构。各元素之间没有严格的物理上的顺序关系，也就是说元素在容器中并没有保存元素置入容器时的逻辑顺序。关联式容器另一个显著特点是：在值中选择一个值作为关键字 key，这个关键

字对值起到索引的作用，方便查找。set/multiset 容器、map/multimap 容器

2.2 算法

算法，问题之解法也。以有限的步骤，解决逻辑或数学上的问题，这一门学科我们叫做算法（Algorithms）。广义而言，我们所编写的每个程序都是一个算法，其中的每个函数也都是一个算法，毕竟它们都是用来解决或大或小的逻辑问题或数学问题。STL 收录的算法经过了数学上的效能分析与证明，是极具复用价值的，包括常用的排序，查找等等。特定的算法往往搭配特定的数据结构，算法与数据结构相辅相成。

算法分为：质变算法和非质变算法。

质变算法：是指运算过程中会更改区间内的元素的内容。例如拷贝，替换，删除等等

非质变算法：是指运算过程中不会更改区间内的元素内容，例如查找、计数、遍历、寻找极值等等

2.3 迭代器

迭代器（iterator）是一种抽象的设计概念，现实程序语言中并没有直接对应于这个概念的实物。在《Design Patterns》一书中提供了 23 中设计模式的完整描述，其中 iterator 模式定义如下：提供一种方法，使之能够依序寻访某个容器所含的各个元素，而又无需暴露该容器的内部表示方式。

迭代器的设计思维是 STL 的关键所在，STL 的中心思想在于将容器（container）和算法（algorithms）分开，彼此独立设计，最后再一贴胶着剂将他们撮合在一起。从技术角度来看，容器和算法的泛型化并不困难，C++ 的 class template 和 function template 可分别达到目标，如果设计出两这个之间的良好的胶着剂，才是大难题。

迭代器的种类:

输入迭代器	提供对数据的只读访问	只读，支持++、==、！=
输出迭代器	提供对数据的只写访问	只写，支持++
前向迭代器	提供读写操作，并能向前推进迭代器	读写，支持++、==、！=
双向迭代器	提供读写操作，并能向前和向后操作	读写，支持++、--，
随机访问迭代器	提供读写操作，并能以跳跃的方式访问容器的任意数据，是功能最强的迭代器	读写，支持++、--、[n]、-n、<、<=、>、>=

3. 常用容器

3.1 string 容器

3.1.1 string 容器基本概念

C 风格字符串(以空字符结尾的字符数组)太过复杂难于掌握，不适合大程序的开发，所以 C++ 标准库定义了一种 string 类，定义在头文件<string>。

string 和 C 风格字符串对比:

◆ char*是一个指针，string 是一个类

string 封装了 char*，管理这个字符串，是一个 char* 型的容器

◆ string 封装了很多实用的成员方法

查找 find，拷贝 copy，删除 delete 替换 replace，插入 insert

◆ 不用考虑内存释放和越界

string 管理 char* 所分配的内存。每一次 string 的复制，取值都由 string 类负责维护，不用担心复制越界和取值越界等

3.1.2 string 容器常用操作

3.1.2.1 string 构造函数

```
string();//创建一个空的字符串 例如: string str;
string(const string& str);//使用一个 string 对象初始化另一个 string 对象
string(const char* s);//使用字符串 s 初始化
string(int n, char c);//使用 n 个字符 c 初始化
```

3.1.2.2 string 基本赋值操作

```
string& operator=(const char* s);//char*类型字符串 赋值给当前的字符串
string& operator=(const string &s);//把字符串 s 赋给当前的字符串
string& operator=(char c);//字符赋值给当前的字符串
string& assign(const char *s);//把字符串 s 赋给当前的字符串
string& assign(const char *s, int n);//把字符串 s 的前 n 个字符赋给当前的字符串
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 赋给当前字符串
string& assign(const string &s, int start, int n);//将 s 从 start 开始 n 个字符赋值给字符串
```

3.1.2.3 string 存取字符操作

```
char& operator[](int n);//通过[]方式取字符
char& at(int n);//通过 at 方法获取字符
```

3.1.2.4 string 拼接操作

```
string& operator+=(const string& str);//重载+=操作符
string& operator+=(const char* str);//重载+=操作符
string& operator+=(const char c);//重载+=操作符
string& append(const char *s);//把字符串 s 连接到当前字符串结尾
string& append(const char *s, int n);//把字符串 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
当前字符串结尾
string& append(int n, char c);//在当前字符串结尾添加 n 个字符 c
```

3.1.2.5 string 查找和替换

```
int find(const string& str, int pos = 0) const; //查找 str 第一次出现位置,从 pos 开始查找
int find(const char* s, int pos = 0) const;  //查找 s 第一次出现位置,从 pos 开始查找
int find(const char* s, int pos, int n) const;  //从 pos 位置查找 s 的前 n 个字符第一次位置
int find(const char c, int pos = 0) const;  //查找字符 c 第一次出现位置
int rfind(const string& str, int pos = npos) const;//查找 str 最后一次位置,从 pos 开始查找
int rfind(const char* s, int pos = npos) const;//查找 s 最后一次出现位置,从 pos 开始查找
int rfind(const char* s, int pos, int n) const;//从 pos 查找 s 的前 n 个字符最后一次位置
int rfind(const char c, int pos = 0) const; //查找字符 c 最后一次出现位置
string& replace(int pos, int n, const string& str); //替换从 pos 开始 n 个字符为字符串 str
string& replace(int pos, int n, const char* s); //替换从 pos 开始的 n 个字符为字符串 s
```

3.1.2.6 string 比较操作

```
/*
compare 函数在>时返回 1，<时返回 -1，==时返回 0。
比较区分大小写，比较时参考字典顺序，排越前面的越小。
大写的 A 比小写的 a 小。
*/
int compare(const string &s) const;//与字符串 s 比较
int compare(const char *s) const;//与字符串 s 比较
```

3.1.2.7 string 子串

```
string substr(int pos = 0, int n = npos) const;//返回由 pos 开始的 n 个字符组成的字符串
```

3.1.2.8 string 插入和删除操作

```
string& insert(int pos, const char* s); //插入字符串
string& insert(int pos, const string& str); //插入字符串
string& insert(int pos, int n, char c);//在指定位置插入 n 个字符 c
string& erase(int pos, int n = npos);//删除从 Pos 开始的 n 个字符
```

3.1.2.9 string 和 c-style 字符串转换

```
//string 转 char*
string str = "nowcoder";
const char* cstr = str.c_str();
//char* 转 string
char* s = "nowcoder";
string str(s);
```

提示:

在 C++ 中存在一个从 const char* 到 string 的隐式类型转换，却不存在从一个 string 对象到 C_string 的自动类型转换。对于 string 类型的字符串，可以通过 c_str()函数返回 string 对象对应的 C_string。

通常，程序员在整个程序中应坚持使用 string 类对象，直到必须将内容转化为 char* 时才将其转换为 C_string。

提示:

为了修改 string 字符串的内容，下标操作符 [] 和 at 都会返回字符的引用。但当字符串的内存被重新分配之后，可能发生错误。

```
string s = "abcdefg";
char& a = s[2];
char& b = s[3];

a = '1';
b = '2';

cout << s << endl;
cout << (int*)s.c_str() << endl;

s = "ppppppppppppppppppppppppp";

//a = '1';
//b = '2';

cout << s << endl;
cout << (int*)s.c_str() << endl;
```

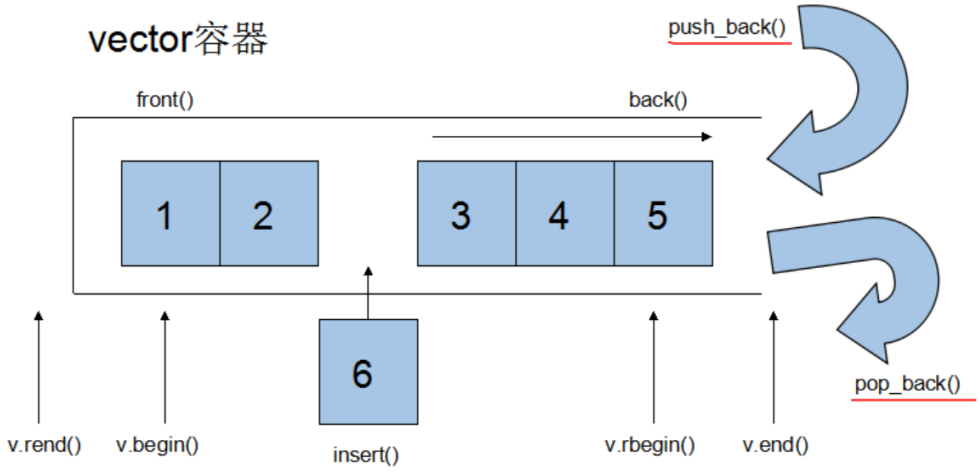
3.2 vector 容器

3.2.1 vector 容器基本概念

vector 的数据安排以及操作方式，与 array 非常相似，两者的唯一差别在于空间的运

用的灵活性。array 是静态空间，一旦配置了就不能改变，要换大一点或者小一点的空间，可以，一切琐碎得由自己来，首先配置一块新的空间，然后将旧空间的数据搬往新空间，再释放原来的空间。vector 是动态空间，随着元素的加入，它的内部机制会自动扩充空间以容纳新元素。因此 vector 的运用对于内存的合理利用与运用的灵活性有很大的帮助，我们再也不必害怕空间不足而一开始就要求一个大块头的 array 了。

vector 的实现技术，关键在于其对大小的控制以及重新配置时的数据移动效率，一旦 vector 旧空间满了，如果客户每新增一个元素，vector 内部只是扩充一个元素的空间，实为不智，因为所谓的扩充空间（不论多大），一如刚所说，是“配置新空间-数据移动-释放旧空间”的大工程，时间成本很高，应该加入某种未雨绸缪的考虑，稍后我们便可以看到 vector 的空间配置策略。



3.2.2 vector 迭代器

vector 维护一个线性空间，所以不论元素的型别如何，普通指针都可以作为 vector 的迭代器，因为 vector 迭代器所需要的操作行为，如 operator*、 operator->、operator++、 operator--、operator+、operator-、operator+=、operator-=，普通指针天生具备。vector 支持随机存取，而普通指针正有着这样的能力。所以 vector 提供

的是随机访问迭代器（Random Access Iterators）。

根据上述描述，如果写如下的代码：

vector<int>::iterator it1;

vector<Teacher>::iterator it2;

it1 的型别其实就是 int*，it2 的型别其实就是 Teacher*。

3.2.3 vector 的数据结构

vector 所采用的数据结构非常简单，线性连续空间，它以两个迭代器 _Myfirst 和 _Mylast 分别指向配置得来的连续空间中目前已被使用的范围，并以迭代器 _Myend 指向整块连续内存空间的尾端。

为了降低空间配置时的速度成本，vector 实际配置的大小可能比客户端需求大一些，以备将来可能的扩充，这边是容量的概念。换句话说，一个 vector 的容量永远大于或等于其大小，一旦容量等于大小，便是满载，下次再有新增元素，整个 vector 容器就得另觅居所。

注意：

所谓动态增加大小，并不是在原空间之后续接新空间（因为无法保证原空间之后尚有可配置的空间），而是一块更大的内存空间，然后将原数据拷贝新空间，并释放原空间。因此，对 vector 的任何操作，一旦引起空间的重新配置，指向原 vector 的所有迭代器就都失效了。这是程序员容易犯的一个错误，务必小心。

3.2.4 vector 常用 API 操作

3.2.4.1 vector 构造函数

```
vector<T> v; //采用模板实现类实现，默认构造函数
vector(v.begin(), v.end());//将 v[begin(), end())区间中的元素拷贝给本身。
vector(n, elem);//构造函数将 n 个 elem 拷贝给本身。
vector(const vector &vec);//拷贝构造函数。
```

3.2.4.2 vector 常用赋值操作

```
assign(beg, end);//将[beg, end)区间中的数据拷贝赋值给本身。
assign(n, elem);//将 n 个 elem 拷贝赋值给本身。
vector& operator=(const vector  &vec);//重载等号操作符
swap(vec);// 将 vec 与本身的元素互换。
```

3.2.4.3 vector 大小操作

```
size();//返回容器中元素的个数
empty();//判断容器是否为空
resize(int num);//重新指定容器的长度为 num，若容器变长，则以默认值填充新位置。如果容器变短，则末尾超出容器长度的元素被删除。
resize(int num, elem);//重新指定容器的长度为 num，若容器变长，则以 elem 值填充新位置。如果容器变短，则末尾超出容器长>度的元素被删除。
capacity();//容器的容量
reserve(int len);//容器预留 len 个元素长度，预留位置不初始化，元素不可访问。
```

3.2.4.4 vector 数据存取操作

```
at(int idx); //返回索引 idx 所指的数据，如果 idx 越界，抛出 out_of_range 异常
operator[];//返回索引 idx 所指的数据，越界时，运行直接报错
front();//返回容器中第一个数据元素
back();//返回容器中最后一个数据元素
```

3.2.4.5 vector 插入和删除操作

```
insert(const_iterator pos, int count,ele);//迭代器指向位置 pos 插入 count 个元素 ele
push_back(ele); //尾部插入元素 ele
pop_back();//删除最后一个元素
erase(const_iterator start, const_iterator end);//删除迭代器从 start 到 end 之间的元素
erase(const_iterator pos);//删除迭代器指向的元素
clear();//删除容器中所有元素
```

3.2.5 vector 小案例

3.2.5.1 巧用 swap，收缩内存空间

```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
#include<vector>
using namespace std;

int main(){

    vector<int> v;
    for (int i = 0; i < 100000;i ++){
        v.push_back(i);
    }

    cout << "capacity:" << v.capacity() << endl;
    cout << "size:" << v.size() << endl;

    //此时 通过 resize 改变容器大小
    v.resize(10);

    cout << "capacity:" << v.capacity() << endl;
    cout << "size:" << v.size() << endl;

    //容量没有改变
    vector<int>(v).swap(v);

    cout << "capacity:" << v.capacity() << endl;
    cout << "size:" << v.size() << endl;

    system("pause");
    return EXIT_SUCCESS;
}
```

3.2.5.2 reserve 预留空间

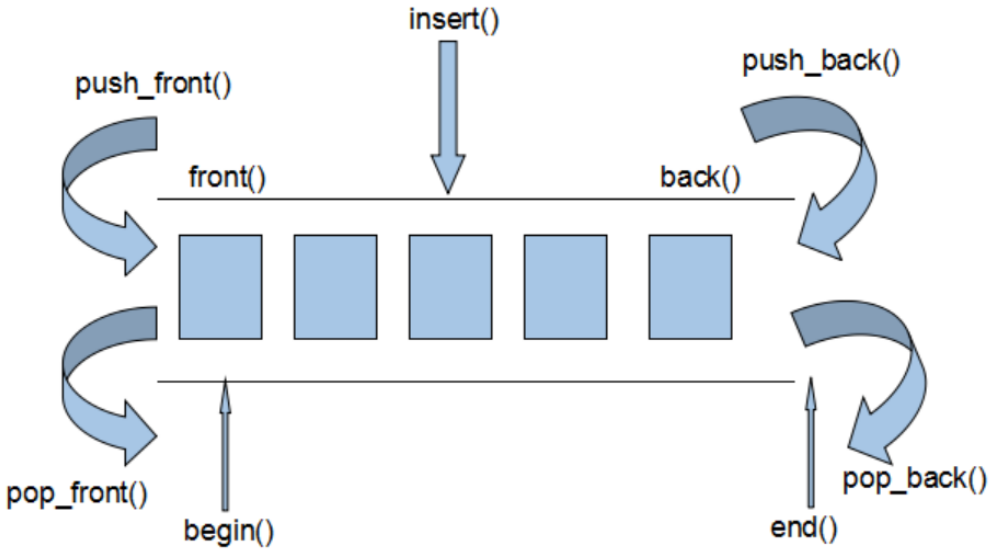
```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
#include<vector>
using namespace std;

int main(){
```

```
    vector<int> v;

    //预先开辟空间
    v.reserve(100000);

    int* pStart = NULL;
    int count = 0;
    for (int i = 0; i < 100000;i ++){
        v.push_back(i);
        if (pStart != &v[0]){
            pStart = &v[0];
            count++;
        }
    }

    cout << "count:" << count << endl;

    system("pause");
    return EXIT_SUCCESS;
}
```

3.3 deque 容器

3.3.1 deque 容器基本概念

vector 容器是单向开口的连续内存空间，deque 则是一种双向开口的连续线性空间。所谓的双向开口，意思是可以在头尾两端分别做元素的插入和删除操作，当然，vector 容器也可以在头尾两端插入元素，但是在其头部操作效率奇差，无法被接受。



deque 容器和 vector 容器最大的差异，一在于 deque 允许使用常数项时间对头端进行元素的插入和删除操作。二在于 deque 没有容量的概念，因为它是动态的以分段连续空间组合而成，随时可以增加一段新的空间并链接起来，换句话说，像 vector 那样，“旧空间不足而重新配置一块更大空间，然后复制元素，再释放旧空间”这样的事情在 deque 身上是不会发生的。也因此，deque 没有必须要提供所谓的空间保留（reserve）功能。

虽然 deque 容器也提供了 Random Access Iterator，但是它的迭代器并不是普通的指针，其复杂度和 vector 不是一个量级，这当然影响各个运算的层面。因此，除非有必要，我们应该尽可能的使用 vector ，而不是 deque 。对 deque 进行的排序操作，为了最高效率，可将 deque 先完整的复制到一个 vector 中，对 vector 容器进行排序，再复制回 deque。

3.3.2 deque 容器实现原理

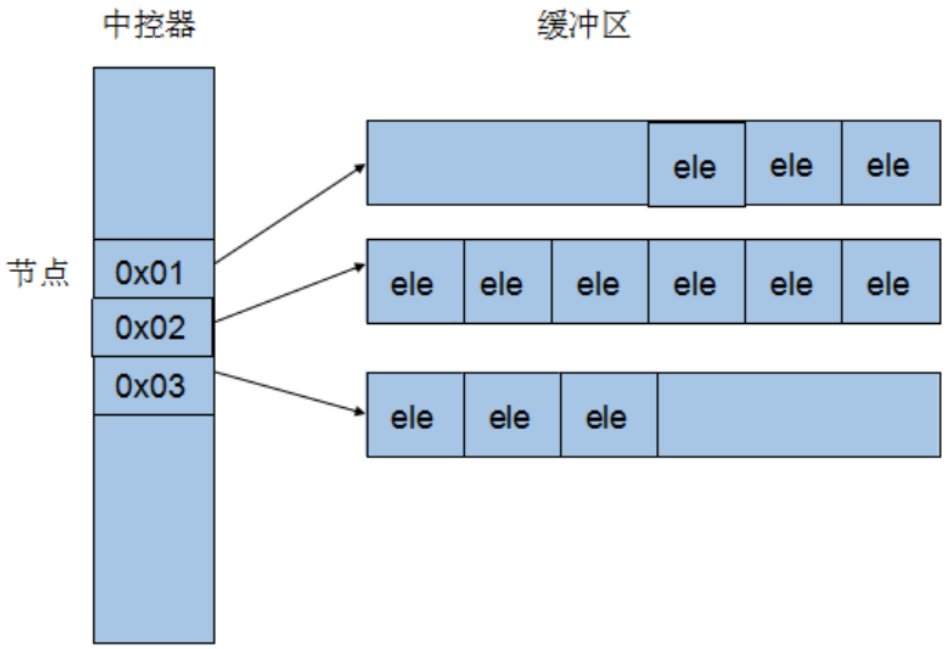
deque 容器是连续的空间，至少逻辑上看来如此，连续现行空间总是令我们联想到 array 和 vector，array 无法成长，vector 虽可成长，却只能向尾端成长，而且其成长其实是一个假象，事实上(1) 申请更大空间 (2)原数据复制新空间 (3)释放原空间 三步骤，如

果不是 vector 每次配置新的空间时都留有余裕，其成长假象所带来的代价是非常昂贵的。

deque 是由一段一段的定量的连续空间构成。一旦有必要在 deque 前端或者尾端增加新的空间，便配置一段连续定量的空间，串接在 deque 的头端或者尾端。deque 最大的工作就是维护这些分段连续的内存空间的整体性的假象，并提供随机存取的接口，避开了重新配置空间，复制，释放的轮回，代价就是复杂的迭代器架构。

既然 deque 是分段连续内存空间，那么就必须有中央控制，维持整体连续的假象，数据结构的设计及迭代器的前进后退操作颇为繁琐。deque 代码的实现远比 vector 或 list 都多得多。

deque 采取一块所谓的 map（注意，不是 STL 的 map 容器）作为主控，这里所谓的 map 是一小块连续的内存空间，其中每一个元素（此处成为一个结点）都是一个指针，指向另一段连续性内存空间，称作缓冲区。缓冲区才是 deque 的存储空间的主体。



3.3.3 deque 常用 API

3.3.3.1 deque 构造函数

```
deque<T> deqT;//默认构造形式
deque(beg, end);//构造函数将[beg, end)区间中的元素拷贝给本身。
deque(n, elem);//构造函数将 n 个 elem 拷贝给本身。
deque(const deque &deq);//拷贝构造函数。
```

3.3.3.2 deque 赋值操作

```
assign(beg, end);//将[beg, end)区间中的数据拷贝赋值给本身。
assign(n, elem);//将 n 个 elem 拷贝赋值给本身。
deque& operator=(const deque &deq); //重载等号操作符
swap(deq);// 将 deq 与本身的元素互换
```

3.3.3.3 deque 大小操作

```
deque.size();//返回容器中元素的个数
deque.empty();//判断容器是否为空
deque.resize(num);//重新指定容器的长度为 num,若容器变长，则以默认值填充新位置。如果容器变短，则末尾超出容器长度的元素被删除。
deque.resize(num, elem); //重新指定容器的长度为 num,若容器变长，则以 elem 值填充新位置,如果容器变短，则末尾超出容器长度的元素被删除。
```

3.3.3.4 deque 双端插入和删除操作

```
push_back(elem);//在容器尾部添加一个数据
push_front(elem);//在容器头部插入一个数据
pop_back();//删除容器最后一个数据
pop_front();//删除容器第一个数据
```

3.3.3.5 deque 数据存取

```
at(idx);//返回索引 idx 所指的数据，如果 idx 越界，抛出 out_of_range。
operator[];//返回索引 idx 所指的数据，如果 idx 越界，不抛出异常，直接出错。
front();//返回第一个数据。
back();//返回最后一个数据
```

3.3.3.6 deque 插入操作

```
insert(pos,elem);//在 pos 位置插入一个 elem 元素的拷贝，返回新数据的位置。
insert(pos,n,elem);//在 pos 位置插入 n 个 elem 数据，无返回值。
insert(pos,beg,end);//在 pos 位置插入[beg,end)区间的数据，无返回值。
```

3.3.3.7 deque 删除操作

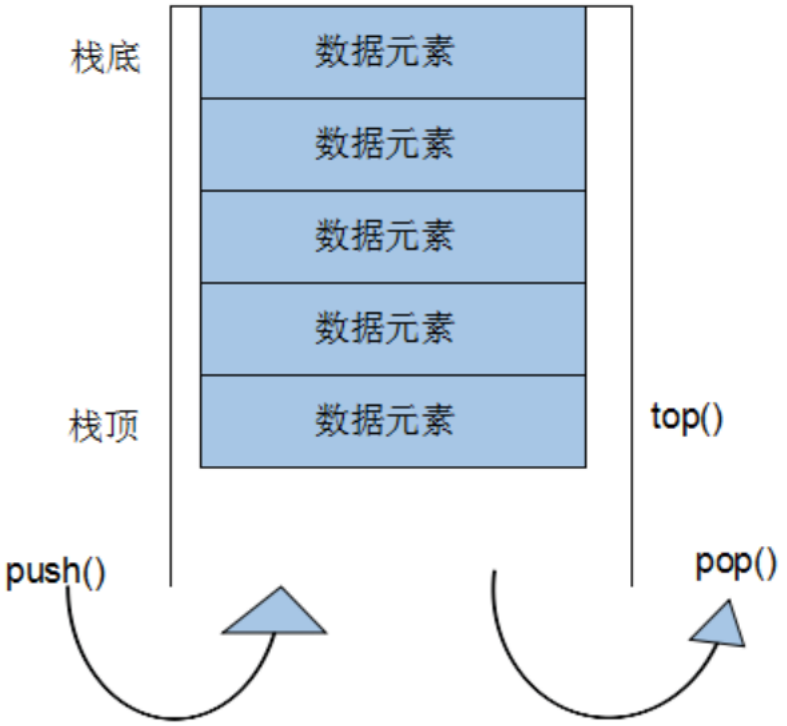
```
clear();//移除容器的所有数据
erase(beg,end);//删除[beg,end)区间的数据，返回下一个数据的位置。
erase(pos);//删除 pos 位置的数据，返回下一个数据的位置。
```

3.4 stack 容器

3.4.1 stack 容器基本概念

stack 是一种先进后出（First In Last Out，FILO）的数据结构，它只有一个出口，形式如图所示。stack 容器允许新增元素，移除元素，取得栈顶元素，但是除了最顶端外，没有任何其他方法可以存取 stack 的其他元素。换言之，stack 不允许有遍历行为。

有元素推入栈的操作称为 push，将元素推出 stack 的操作称为 pop。



3.4.2 stack 没有迭代器

stack 所有元素的进出都必须符合“先进后出”的条件，只有 stack 顶端的元素，才有机会被外界取用。stack 不提供遍历功能，也不提供迭代器。

3.4.3 stack 常用 API

3.4.3.1 stack 构造函数

```
stack<T> stkT;//stack 采用模板类实现， stack 对象的默认构造形式：
stack(const stack &stk);//拷贝构造函数
```

3.4.3.2 stack 赋值操作

```
stack& operator=(const stack &stk);//重载等号操作符
```

3.4.3.3 stack 数据存取操作

```
push(elem);//向栈顶添加元素
pop();//从栈顶移除第一个元素
top();//返回栈顶元素
```

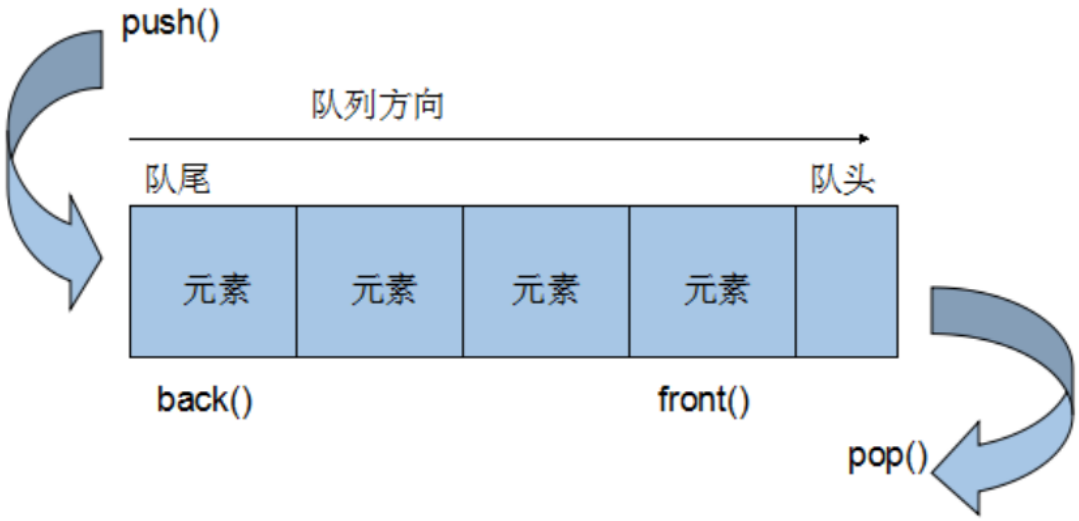
3.4.3.4 stack 大小操作

```
empty();//判断堆栈是否为空
size();//返回堆栈的大小
```

3.5 queue 容器

3.5.1 queue 容器基本概念

queue 是一种先进先出(First In First Out，FIFO)的数据结构，它有两个出口，queue 容器允许从一端新增元素，从另一端移除元素。



3.5.2 queue 没有迭代器

queue 所有元素的进出都必须符合“先进先出”的条件，只有 queue 的顶端元素，才有机会被外界取用。queue 不提供遍历功能，也不提供迭代器。

3.5.3 queue 常用 API

3.5.3.1 queue 构造函数

```
queue<T> queT;//queue 采用模板类实现，queue 对象的默认构造形式
queue(const queue &que);//拷贝构造函数
```

3.5.3.2 queue 存取、插入和删除操作

```
push(elem);//往队尾添加元素
pop();//从队头移除第一个元素
back();//返回最后一个元素
front();//返回第一个元素
```

3.5.3.3 queue 赋值操作

```
queue& operator=(const queue &que);//重载等号操作符
```

3.5.3.4 queue 大小操作

```
empty();//判断队列是否为空
size();//返回队列的大小
```

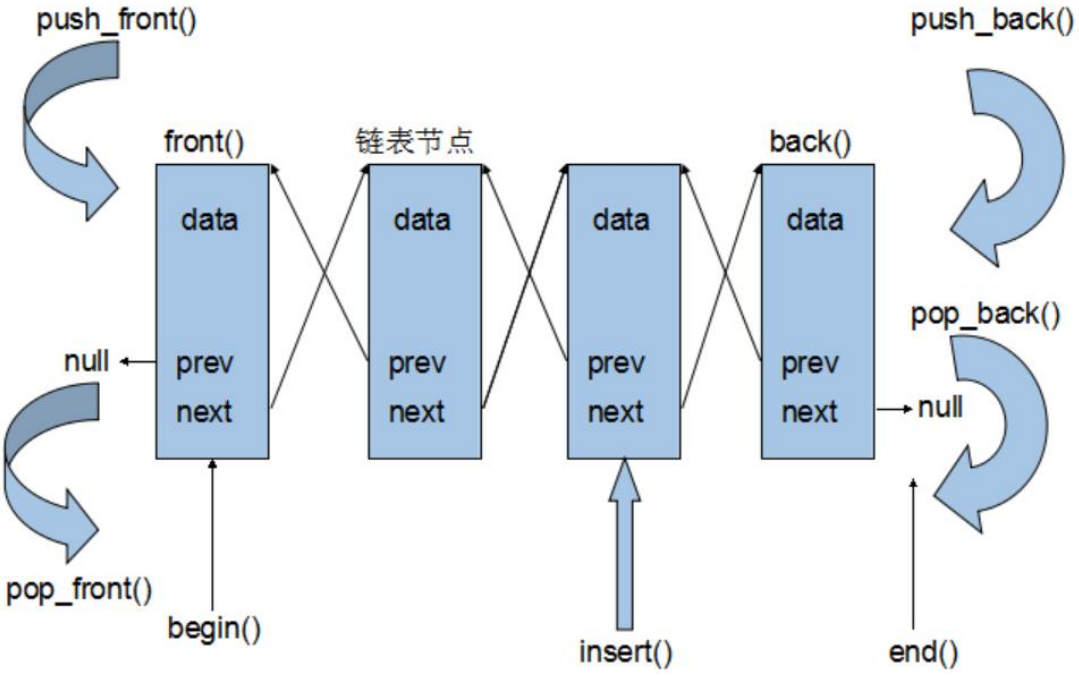
3.6 list 容器

3.6.1 list 容器基本概念

链表是一种物理存储单元上非连续、非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的。链表由一系列结点（链表中每一个元素称为结点）组成，结点可以在运行时动态生成。每个结点包括两个部分：一个是存储数据元素的数据域，另一个是存储下一个结点地址的指针域。

相较于 vector 的连续线性空间，list 就显得负责许多，它的好处是每次插入或者删除一个元素，就是配置或者释放一个元素的空间。因此，list 对于空间的运用有绝对的精准，一点也不浪费。而且，对于任何位置的元素插入或元素的移除，list 永远是常数时间。

list 和 vector 是两个最常被使用的容器。list 容器是一个双向循环链表。



- 采用动态存储分配，不会造成内存浪费和溢出
- 链表执行插入和删除操作十分方便，修改指针即可，不需要移动大量元素
- 链表灵活，但是空间和时间额外耗费较大

3.6.2 list 容器的迭代器

list 容器不能像 vector 一样以普通指针作为迭代器，因为其节点不能保证在同一块连续的内存空间上。list 迭代器必须有能力指向 list 的节点，并有能力进行正确的递增、递减、取值、成员存取操作。所谓“list 正确的递增，递减、取值、成员取用”是指，递增时指向下一个节点，递减时指向上一个节点，取值时取的是节点的数据值，成员取用时取的是节点的成员。

由于 list 是一个双向链表，迭代器必须能够具备前移、后移的能力，所以 list 容器提供的是 Bidirectional Iterators（双向迭代器）。

list 有一个重要的性质，插入操作和删除操作都不会造成原有 list 迭代器的失效。这在 vector 是不成立的，因为 vector 的插入操作可能造成记忆体重新配置，导致原有的迭代器全部失效，甚至 list 元素的删除，也只有被删除的那个元素的迭代器失效，其他迭代器不受任何影响。

3.6.3 list 容器的数据结构

list 容器不仅是一个双向链表，而且还是一个循环的双向链表。

```
#define _CRT_SECURE_NO_WARNINGS
#include<iostream>
#include<list>
using namespace std;

int main(){

    list<int> myList;
    for (int i = 0; i < 10; i ++){
        myList.push_back(i);
    }

    list<int>::_Nodeptr node =  myList._Myhead->_Next;

    for (int i = 0; i < myList._Mysize * 2;i++){
        cout << "Node:" << node->_Myval << endl;
        node = node->_Next;
        if (node == myList._Myhead){
            node = node->_Next;
        }
    }

    system("pause");
    return EXIT_SUCCESS;
}
```

3.6.4 list 常用 API

3.6.4.1 list 构造函数

```
list<T> lstT;//list 采用采用模板类实现,对象的默认构造形式:
list(beg,end);//构造函数将[beg, end)区间中的元素拷贝给本身。
list(n,elem);//构造函数将 n 个 elem 拷贝给本身。
list(const list &lst);//拷贝构造函数。
```

3.6.4.2 list 数据元素插入和删除操作

```
push_back(elem);//在容器尾部加入一个元素
pop_back();//删除容器中最后一个元素
push_front(elem);//在容器开头插入一个元素
pop_front();//从容器开头移除第一个元素
```

```
insert(pos,elem);//在 pos 位置插 elem 元素的拷贝，返回新数据的位置。
insert(pos,n,elem);//在 pos 位置插入 n 个 elem 数据，无返回值。
insert(pos,beg,end);//在 pos 位置插入[beg,end)区间的数据，无返回值。
clear();//移除容器的所有数据
erase(beg,end);//删除[beg,end)区间的数据，返回下一个数据的位置。
erase(pos);//删除 pos 位置的数据，返回下一个数据的位置。
remove(elem);//删除容器中所有与 elem 值匹配的元素。
```

3.6.4.3 list 大小操作

```
size();//返回容器中元素的个数
empty();//判断容器是否为空
resize(num);//重新指定容器的长度为 num，
若容器变长，则以默认值填充新位置。
如果容器变短，则末尾超出容器长度的元素被删除。
resize(num, elem);//重新指定容器的长度为 num，
若容器变长，则以 elem 值填充新位置。
如果容器变短，则末尾超出容器长度的元素被删除。
```

3.6.4.4 list 赋值操作

```
assign(beg, end);//将[beg, end)区间中的数据拷贝赋值给本身。
assign(n, elem);//将 n 个 elem 拷贝赋值给本身。
list& operator=(const list &lst);//重载等号操作符
swap(lst);//将 lst 与本身的元素互换。
```

3.6.4.5 list 数据的存取

```
front();//返回第一个元素。
back();//返回最后一个元素。
```

3.6.4.6 list 反转排序

```
reverse();//反转链表，比如 lst 包含 1,3,5 元素，运行此方法后，lst 就包含 5,3,1 元素。
sort(); //list 排序
```

3.7 set/multiset 容器

3.7.1 set/multiset 容器基本概念

3.7.1.1 set 容器基本概念

set 的特性是，所有元素都会根据元素的键值自动被排序。set 的元素不像 map 那样可以同时拥有实值和键值，set 的元素即是键值又是实值。set 不允许两个元素有相同的键值。

可以通过 set 的迭代器改变 set 元素的值吗？不行，因为 set 元素值就是其键值，关系到 set 元素的排序规则。如果任意改变 set 元素值，会严重破坏 set 组织。换句话说，set 的 iterator 是一种 const_iterator。

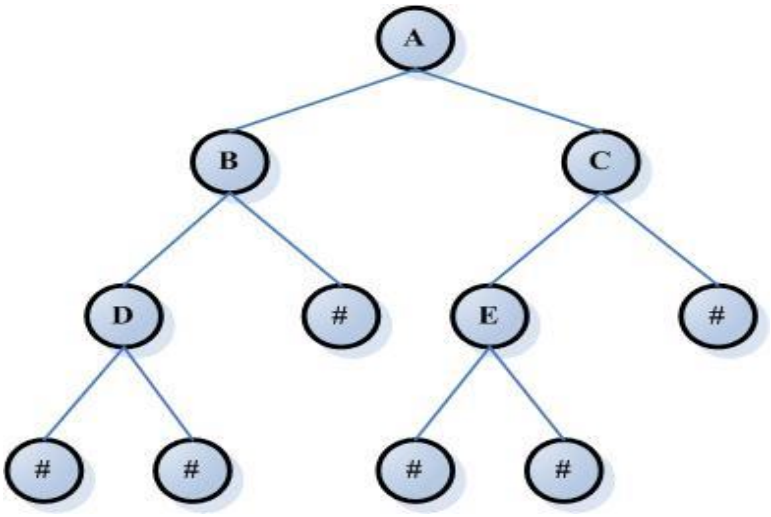
set 拥有和 list 某些相同的性质，当对容器中的元素进行插入操作或者删除操作的时候，操作之前所有的迭代器，在操作完成之后依然有效，被删除的那个元素的迭代器必然是一个例外。

3.7.1.2 multiset 容器基本概念

multiset 特性及用法和 set 完全相同，唯一的差别在于它允许键值重复。set 和 multiset 的底层实现是红黑树，红黑树为平衡二叉树的一种。

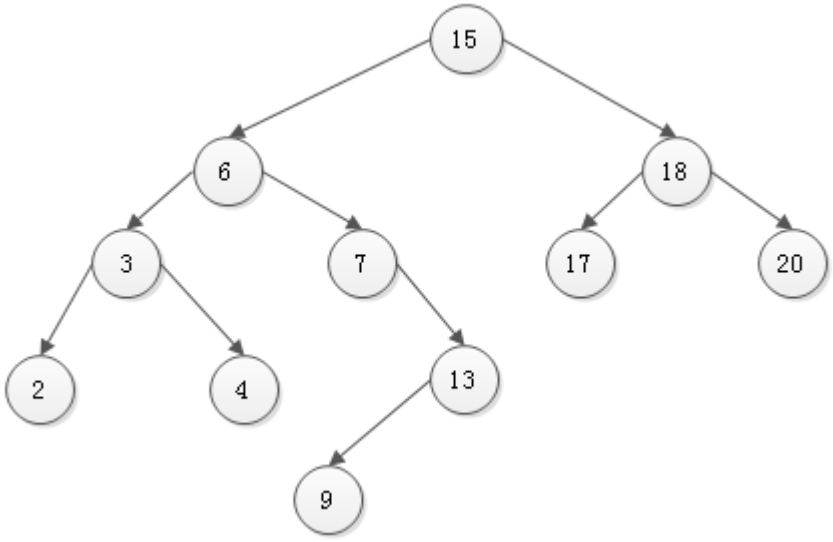
树的简单知识：

二叉树就是任何节点最多只允许有两个字节点。分别是左子结点和右子节点。



二叉树示意图

二叉搜索树，是指二叉树中的节点按照一定的规则进行排序，使得对二叉树中元素访问更加高效。二叉搜索树的放置规则是：任何节点的元素值一定大于其左子树中的每一个节点的元素值，并且小于其右子树的值。因此从根节点一直向左走，一直到无路可走，即得到最小值，一直向右走，直至无路可走，可得到最大值。那么在儿茶搜索树中找到最大元素和最小元素是非常简单的事情。下图为二叉搜索树：



上面我们介绍了二叉搜索树，那么当一个二叉搜索树的左子树和右子树不平衡的时候，那么搜索依据上图表示，搜索 9 所花费的时间要比搜索 17 所花费的时间要多，由于我们的输入或者经过我们插入或者删除操作，二叉树失去平衡，造成搜索效率降低。

所以我们有了一个平衡二叉树的概念，所谓的平衡不是指的完全平衡。

3.7.2 set 常用 API

3.7.2.1 set 构造函数

```
set<T> st;//set 默认构造函数：
mulitset<T> mst; //multiset 默认构造函数:
set(const set &st);//拷贝构造函数
```

3.7.2.2 set 赋值操作

```
set& operator=(const set &st);//重载等号操作符
swap(st);//交换两个集合容器
```

3.7.2.3 set 大小操作

```
size();//返回容器中元素的数目
empty();//判断容器是否为空
```

3.7.2.4 set 插入和删除操作

```
insert(elem);//在容器中插入元素。
clear();//清除所有元素
erase(pos);//删除 pos 迭代器所指的元素，返回下一个元素的迭代器。
erase(beg, end);//删除区间[beg,end)的所有元素 ，返回下一个元素的迭代器。
erase(elem);//删除容器中值为 elem 的元素。
```

3.7.2.5 set 查找操作

```
find(key);//查找键 key 是否存在,若存在，返回该键的元素的迭代器；若不存在，返回 set.end();
count(key);//查找键 key 的元素个数
```

```
lower_bound(keyElem);//返回第一个 key>=keyElem 元素的迭代器。
upper_bound(keyElem);//返回第一个 key>keyElem 元素的迭代器。
equal_range(keyElem);//返回容器中 key 与 keyElem 相等的上下限的两个迭代器。
```

set 的返回值　　指定 set 排序规则:

```
//插入操作返回值
void test01(){

    set<int> s;
    pair<set<int>::iterator,bool> ret = s.insert(10);
    if (ret.second){
        cout << "插入成功:" << *ret.first << endl;
    }
    else{
        cout << "插入失败:" << *ret.first << endl;
    }

    ret = s.insert(10);
    if(ret.second){
        cout << "插入成功:" << *ret.first << endl;
    }
    else{
        cout << "插入失败:" << *ret.first << endl;
    }

}

struct MyCompare02{
    bool operator()(int v1,int v2){
        return v1 > v2;
    }
};

//set 从大到小
void test02(){

    srand((unsigned int)time(NULL));
    //我们发现 set 容器的第二个模板参数可以设置排序规则，默认规则是 less<_Kty>
    set<int, MyCompare02> s;
    for (int i = 0; i < 10;i++){
        s.insert(rand() % 100);
    }
```

```
    for (set<int, MyCompare02>::iterator it = s.begin(); it != s.end(); it ++){
        cout << *it << " ";
    }
    cout << endl;
}

//set 容器中存放对象
class Person{
public:
    Person(string name,int age){
        this->mName = name;
        this->mAge = age;
    }
public:
    string mName;
    int mAge;
};


struct MyCompare03{
    bool operator()(const Person& p1,const Person& p2){
        return p1.mAge > p2.mAge;
    }
};

void test03(){

    set<Person, MyCompare03> s;

    Person p1("aaa", 20);
    Person p2("bbb", 30);
    Person p3("ccc", 40);
    Person p4("ddd", 50);

    s.insert(p1);
    s.insert(p2);
    s.insert(p3);
    s.insert(p4);

    for (set<Person, MyCompare03>::iterator it = s.begin(); it != s.end(); it++){
        cout << "Name:" << it->mName << " Age:" << it->mAge << endl;
    }

}
```

3.7.3 对组（pair）

对组（pair）将一对值组合成一个值，这一对值可以具有不同的数据类型，两个值可以分别用 pair 的两个公有属性 first 和 second 访问。

类模板：template <class T1, class T2> struct pair

如何创建对组?

```
//第一种方法创建一个对组
pair<string, int> pair1(string("name"), 20);
cout << pair1.first << endl; //访问 pair 第一个值
cout << pair1.second << endl;//访问 pair 第二个值
//第二种
pair<string, int> pair2 = make_pair("name", 30);
cout << pair2.first << endl;
cout << pair2.second << endl;
//pair=赋值
pair<string, int> pair3 = pair2;
cout << pair3.first << endl;
cout << pair3.second << endl;
```

3.8 map/multimap 容器

3.8.1 map/multimap 基本概念

map 的特性是，所有元素都会根据元素的键值自动排序。map 所有的元素都是 pair，同时拥有实值和键值，pair 的第一元素被视为键值，第二元素被视为实值，map 不允许两个元素有相同的键值。

我们可以通过 map 的迭代器改变 map 的键值吗？答案是不行，因为 map 的键值关系到 map 元素的排列规则，任意改变 map 键值将会严重破坏 map 组织。如果想要修改元素的实值，那么是可以的。

map 和 list 拥有相同的某些性质，当对它的容器元素进行新增操作或者删除操作时，

操作之前的所有迭代器，在操作完成之后依然有效，当然被删除的那个元素的迭代器必然是个例外。

multimap 和 map 的操作类似，唯一区别 multimap 键值可重复。

map 和 multimap 都是以红黑树为底层实现机制。

3.8.2 map/multimap 常用 API

3.8.2.1 map 构造函数

```
map<T1, T2> mapTT;//map 默认构造函数:
map(const map &mp);//拷贝构造函数
```

3.8.2.2 map 赋值操作

```
map& operator=(const map &mp);//重载等号操作符
swap(mp);//交换两个集合容器
```

3.8.2.3 map 大小操作

```
size();//返回容器中元素的数目
empty();//判断容器是否为空
```

3.8.2.4 map 插入数据元素操作

```
map.insert(...); //往容器插入元素，返回 pair<iterator,bool>
map<int, string> mapStu;
// 第一种 通过 pair 的方式插入对象
mapStu.insert(pair<int, string>(3, "小张"));
// 第二种 通过 pair 的方式插入对象
mapStu.inset(make_pair(-1, "校长"));
// 第三种 通过 value_type 的方式插入对象
mapStu.insert(map<int, string>::value_type(1, "小李"));
// 第四种 通过数组的方式插入值
mapStu[3] = "小刘";
mapStu[5] = "小王";
```

3.8.2.5 map 删除操作

```
clear();//删除所有元素
erase(pos);//删除 pos 迭代器所指的元素，返回下一个元素的迭代器。
erase(beg,end);//删除区间[beg,end)的所有元素 ，返回下一个元素的迭代器。
erase(keyElem);//删除容器中 key 为 keyElem 的对组。
```

3.8.2.6 map 查找操作

```
find(key);//查找键 key 是否存在,若存在，返回该键的元素的迭代器；/若不存在，返回 map.end();
count(keyElem);//返回容器中 key 为 keyElem 的对组个数。对 map 来说，要么是 0，要么是 1。对
multimap 来说，值可能大于 1。
lower_bound(keyElem);//返回第一个 key>=keyElem 元素的迭代器。
upper_bound(keyElem);//返回第一个 key>keyElem 元素的迭代器。
equal_range(keyElem);//返回容器中 key 与 keyElem 相等的上下限的两个迭代器。
```

3.9 STL 容器使用时机

	vector	deque	list	set	multiset	map	multimap
典型内存结构	单端数组	双端数组	双向链表	二叉树	二叉树	二叉树	二叉树
可随机存取	是	是	否	否	否	对 key 而言：不是	否
元素搜寻速度	慢	慢	非常慢	快	快	对 key 而言：快	对 key 而言：快
元素安插移除	尾端	头尾两端	任何位置	-	-	-	-

- vector 的使用场景：比如软件历史操作记录的存储，我们经常要查看历史记录，比如上一次的记录，上上次的记录，但却不会去删除记录，因为记录是事实的描述。
- deque 的使用场景：比如排队购票系统，对排队者的存储可以采用 deque，支持头端的快速移除，尾端的快速添加。如果采用 vector，则头端移除时，会移动大量的数据，速度慢。

  vector 与 deque 的比较：

一：vector.at()比 deque.at()效率高，比如 vector.at(0)是固定的，deque 的开始位置却是不固定的。

二：如果有大量释放操作的话，vector 花的时间更少，这跟二者的内部实现有关。

三：deque 支持头部的快速插入与快速移除，这是 deque 的优点。

- list 的使用场景：比如公交车乘客的存储，随时可能有乘客下车，支持频繁的不确实位置元素的移除插入。
- set 的使用场景：比如对手机游戏的个人得分记录的存储，存储要求从高分到低分的顺序排列。
- map 的使用场景：比如按 ID 号存储十万个用户，想要快速要通过 ID 查找对应的用户。二叉树的查找效率，这时就体现出来了。如果是 vector 容器，最坏的情况下可能要遍历完整个容器才能找到该用户。

4. 常用算法

4.1 函数对象

重载函数调用操作符的类，其对象常称为函数对象（function object），即它们是行为类似函数的对象，也叫仿函数(functor)，其实就是重载“()”操作符，使得类对象可以像函数那样调用。

注意:

1.函数对象(仿函数)是一个类，不是一个函数。

2.函数对象(仿函数)重载了”()“操作符使得它可以像函数一样调用。

分类:假定某个类有一个重载的 operator()，而且重载的 operator() 要求获取一个参

数，我们就将这个类称为“一元仿函数”（unary functor）；相反，如果重载的 operator() 要求获取两个参数，就将这个类称为“二元仿函数”（binary functor）。

函数对象的作用主要是什么？STL 提供的算法往往都有两个版本，其中一个版本表现出最常用的某种运算，另一版本则允许用户通过 template 参数的形式来指定所要采取的策略。

```
//函数对象是重载了函数调用符号的类
class MyPrint
{
public:
    MyPrint()
    {
        m_Num = 0;
    }
    int m_Num;

public:
    void operator() (int num)
    {
        cout << num << endl;
        m_Num++;
    }
};

//函数对象
//重载了()操作符的类实例化的对象，可以像普通函数那样调用,可以有参数 ，可以有返回值
void test01()
{
    MyPrint myPrint;
    myPrint(20);

}
// 函数对象超出了普通函数的概念，可以保存函数的调用状态
void test02()
{
    MyPrint myPrint;
    myPrint(20);
    myPrint(20);
    myPrint(20);
```

```
    cout << myPrint.m_Num << endl;
}

void doBusiness(MyPrint print,int num)
{
    print(num);
}

//函数对象作为参数
void test03()
{
    //参数1：匿名函数对象
    doBusiness(MyPrint(),30);
}
```

总结：

1、函数对象通常不定义构造函数和析构函数，所以在构造和析构时不会发生任何问题，避免了函数调用的运行时问题。

2、函数对象超出普通函数的概念，函数对象可以有自己的状态

3、函数对象可内联编译，性能好。用函数指针几乎不可能

4、模版函数对象使函数对象具有通用性，这也是它的优势之一

4.2 谓词

谓词是指普通函数或重载的 operator()返回值是 bool 类型的函数对象(仿函数)。如果 operator 接受一个参数，那么叫做一元谓词,如果接受两个参数，那么叫做二元谓词，谓词可作为一个判断式。

```
class GreaterThenFive
{
public:
    bool operator()(int num)
    {
```

```
        return num > 5;
    }

};
//一元谓词
void test01()
{
    vector<int> v;
    for (int i = 0; i < 10;i ++)
    {
        v.push_back(i);
    }

     vector<int>::iterator it =  find_if(v.begin(), v.end(),
GreaterThenFive());
     if (it == v.end())
     {
         cout << "没有找到" << endl;
     }
     else
     {
         cout << "找到了: " << *it << endl;
     }
}

//二元谓词
class MyCompare
{
public:
    bool operator()(int num1, int num2)
    {
        return num1 > num2;
    }
};

void test02()
{
    vector<int> v;
    v.push_back(10);
    v.push_back(40);
    v.push_back(20);
    v.push_back(90);
    v.push_back(60);
```

```
    //默认从小到大
    sort(v.begin(), v.end());
    for (vector<int>::iterator it = v.begin(); it != v.end();it++)
    {
        cout << *it << " ";
    }
    cout << endl;
    cout << "----------------------------" << endl;
    //使用函数对象改变算法策略，排序从大到小
    sort(v.begin(), v.end(),MyCompare());
    for (vector<int>::iterator it = v.begin(); it != v.end(); it++)
    {
        cout << *it << " ";
    }
    cout << endl;
}
```

4.3 内建函数对象

STL 内建了一些函数对象。分为：算数类函数对象,关系运算类函数对象，逻辑运算类仿函数。这些仿函数所产生的对象，用法和一般函数完全相同，当然我们还可以产生无名的临时对象来履行函数功能。使用内建函数对象，需要引入头文件 #include<functional>。

- 6 个算数类函数对象,除了 negate 是一元运算，其他都是二元运算。

```
template<class T> T plus<T>//加法仿函数
template<class T> T minus<T>//减法仿函数
template<class T> T multiplies<T>//乘法仿函数
template<class T> T divides<T>//除法仿函数
template<class T> T modulus<T>//取模仿函数
template<class T> T negate<T>//取反仿函数
```

- 6 个关系运算类函数对象,每一种都是二元运算。

```
template<class T> bool equal_to<T>//等于
template<class T> bool not_equal_to<T>//不等于
template<class T> bool greater<T>//大于
template<class T> bool greater_equal<T>//大于等于
template<class T> bool less<T>//小于
template<class T> bool less_equal<T>//小于等于
```

- 逻辑运算类运算函数,not 为一元运算，其余为二元运算。

```
template<class T> bool logical_and<T>//逻辑与
template<class T> bool logical_or<T>//逻辑或
template<class T> bool logical_not<T>//逻辑非
```

内建函数对象举例:

```
//取反仿函数
void test01()
{
    negate<int> n;
    cout << n(50) << endl;
}

//加法仿函数
void test02()
{
    plus<int> p;
    cout << p(10, 20) << endl;
}

//大于仿函数
void test03()
{
    vector<int> v;
    srand((unsigned int)time(NULL));
    for (int i = 0; i < 10; i++){
        v.push_back(rand() % 100);
    }

    for (vector<int>::iterator it = v.begin(); it != v.end(); it++){
        cout << *it << " ";
    }
    cout << endl;
    sort(v.begin(), v.end(), greater<int>());

    for (vector<int>::iterator it = v.begin(); it != v.end(); it++){
        cout << *it << " ";
    }
    cout << endl;
```

```
}
```

3.1.4 函数对象适配器

```
//函数适配器 bind1st bind2nd
//现在我有这个需求 在遍历容器的时候，我希望将容器中的值全部加上 100 之后显示出来，怎么做？
//我们直接给函数对象绑定参数 编译阶段就会报错
//for_each(v.begin(), v.end(), bind2nd(myprint(),100));
//如果我们想使用绑定适配器,需要我们自己的函数对象继承binary_function 或者 unary_function
//根据我们函数对象是一元函数对象 还是二元函数对象
class MyPrint :public binary_function<int,int,void>
{
public:
    void operator()(int v1,int v2) const
    {
        cout << "v1 = : " << v1 << " v2 = :" <<v2  << " v1+v2 = :" << (v1 + v2) << endl;

    }
};
//1、函数适配器
void test01()
{
    vector<int>v;
    for (int i = 0; i < 10; i++)
    {
        v.push_back(i);
    }
    cout << "请输入起始值：" << endl;
    int x;
    cin >> x;

    for_each(v.begin(), v.end(), bind1st(MyPrint(), x));
    //for_each(v.begin(), v.end(), bind2nd( MyPrint(),x ));
}
//总结：  bind1st和bind2nd区别?
//bind1st ：  将参数绑定为函数对象的第一个参数
//bind2nd ：  将参数绑定为函数对象的第二个参数
//bind1st bind2nd将二元函数对象转为一元函数对象
```

```
class GreaterThenFive:public unary_function<int,bool>
{
public:
    bool operator ()(int v) const
    {
        return v > 5;
    }
};

//2、取反适配器
void test02()
{
    vector <int> v;
    for (int i = 0; i < 10;i++)
    {
        v.push_back(i);
    }

//  vector<int>::iterator it =  find_if(v.begin(), v.end(), GreaterThenFive()); //
返回第一个大于5的迭代器
//  vector<int>::iterator it = find_if(v.begin(), v.end(),
not1(GreaterThenFive())); //返回第一个小于5迭代器
    //自定义输入
    vector<int>::iterator it = find_if(v.begin(), v.end(), not1
( bind2nd(greater<int>(),5)));
    if (it == v.end())
    {
        cout << "没找到" << endl;
    }
    else
    {
        cout << "找到" << *it << endl;
    }

    //排序  二元函数对象
    sort(v.begin(), v.end(), not2(less<int>()));
    for_each(v.begin(), v.end(), [](int val){cout << val << " "; });

}
//not1 对一元函数对象取反
//not2 对二元函数对象取反

void MyPrint03(int v,int v2)
```

```
{
    cout << v + v2<< " ";
}

//3、函数指针适配器   ptr_fun
void test03()
{
    vector <int> v;
    for (int i = 0; i < 10; i++)
    {
        v.push_back(i);
    }
    // ptr_fun( )把一个普通的函数指针适配成函数对象
    for_each(v.begin(), v.end(), bind2nd( ptr_fun( MyPrint03 ), 100));
}


//4、成员函数适配器
class Person
{
public:
    Person(string name, int age)
    {
        m_Name = name;
        m_Age = age;
    }

    //打印函数
    void ShowPerson(){
        cout << "成员函数:" << "Name:" << m_Name << " Age:" << m_Age << endl;
    }
    void Plus100()
    {
        m_Age += 100;
    }
public:
    string m_Name;
    int m_Age;
};

void MyPrint04(Person &p)
{
    cout << "姓名：" <<  p.m_Name << " 年龄：" << p.m_Age << endl;
```

```
};

void test04()
{
    vector <Person>v;
    Person p1("aaa", 10);
    Person p2("bbb", 20);
    Person p3("ccc", 30);
    Person p4("ddd", 40);
    v.push_back(p1);
    v.push_back(p2);
    v.push_back(p3);
    v.push_back(p4);

    //for_each(v.begin(), v.end(), MyPrint04);
    //利用 mem_fun_ref 将Person内部成员函数适配
    for_each(v.begin(), v.end(), mem_fun_ref(&Person::ShowPerson));
//  for_each(v.begin(), v.end(), mem_fun_ref(&Person::Plus100));
//  for_each(v.begin(), v.end(), mem_fun_ref(&Person::ShowPerson));
}

void test05(){

    vector<Person*> v1;
    //创建数据
    Person p1("aaa", 10);
    Person p2("bbb", 20);
    Person p3("ccc", 30);
    Person p4("ddd", 40);

    v1.push_back(&p1);
    v1.push_back(&p2);
    v1.push_back(&p3);
    v1.push_back(&p4);

    for_each(v1.begin(), v1.end(), mem_fun(&Person::ShowPerson));
}

//如果容器存放的是对象指针,   那么用mem_fun
//如果容器中存放的是对象实体，那么用 mem_fun_ref
```

4.2 算法概述

算法主要是由头文件<algorithm> <functional> <numeric>组成。

<algorithm>是所有 STL 头文件中最大的一个,其中常用的功能涉及到比较，交换，查找,遍历，复制，修改，反转，排序，合并等...

<numeric>体积很小，只包括在几个序列容器上进行的简单运算的模板函数.

<functional> 定义了一些模板类,用以声明函数对象。

4.3 常用遍历算法

```
/*
    遍历算法 遍历容器元素
    @param beg 开始迭代器
    @param end 结束迭代器
    @param _callback  函数回调或者函数对象
    @return 函数对象
*/
for_each(iterator beg, iterator end, _callback);
/*
    transform 算法 将指定容器区间元素搬运到另一容器中
    注意 : transform 不会给目标容器分配内存，所以需要我们提前分配好内存
    @param beg1 源容器开始迭代器
    @param end1 源容器结束迭代器
    @param beg2 目标容器开始迭代器
    @param _cakkback 回调函数或者函数对象
    @return 返回目标容器迭代器
*/
transform(iterator beg1, iterator end1, iterator beg2, _callbakc)
```

for_each:

```
/*

template<class _InIt,class _Fn1> inline
void for_each(_InIt _First, _InIt _Last, _Fn1 _Func)
{
    for (; _First != _Last; ++_First)
        _Func(*_First);
}
```

```
*/

//普通函数
void print01(int val){
    cout << val << " ";
}
//函数对象
struct print001{
    void operator()(int val){
        cout << val << " ";
    }
};

//for_each 算法基本用法
void test01(){

    vector<int> v;
    for (int i = 0; i < 10;i++){
        v.push_back(i);
    }

    //遍历算法
    for_each(v.begin(), v.end(), print01);
    cout << endl;

    for_each(v.begin(), v.end(), print001());
    cout << endl;

}

struct print02{
    print02(){
        mCount = 0;
    }
    void operator()(int val){
        cout << val << " ";
        mCount++;
    }
    int mCount;
};

//for_each 返回值
void test02(){
```

```
    vector<int> v;
    for (int i = 0; i < 10; i++){
        v.push_back(i);
    }

    print02 p = for_each(v.begin(), v.end(), print02());
    cout << endl;
    cout << p.mCount << endl;
}

struct print03 : public binary_function<int, int, void>{
    void operator()(int val,int bindParam) const{
        cout << val + bindParam << " ";
    }
};

//for_each 绑定参数输出
void test03(){

    vector<int> v;
    for (int i = 0; i < 10; i++){
        v.push_back(i);
    }

    for_each(v.begin(), v.end(), bind2nd(print03(),100));
}
```

transform:

```
//transform 将一个容器中的值搬运到另一个容器中
/*
    template<class _InIt, class _OutIt, class _Fn1> inline
    _OutIt _Transform(_InIt _First, _InIt _Last,_OutIt _Dest, _Fn1 _Func)
    {

        for (; _First != _Last; ++_First, ++_Dest)
            *_Dest = _Func(*_First);
        return (_Dest);
    }

    template<class _InIt1,class _InIt2,class _OutIt,class _Fn2> inline
    _OutIt _Transform(_InIt1 _First1, _InIt1 _Last1,_InIt2 _First2, _OutIt _Dest, _Fn2
_Func)
```

```
    {
        for (; _First1 != _Last1; ++_First1, ++_First2, ++_Dest)
            *_Dest = _Func(*_First1, *_First2);
        return (_Dest);
    }
*/

struct transformTest01{
    int operator()(int val){
        return val + 100;
    }
};
struct print01{
    void operator()(int val){
        cout << val << " ";
    }
};
void test01(){

    vector<int> vSource;
    for (int i = 0; i < 10;i ++){
        vSource.push_back(i + 1);
    }

    //目标容器
    vector<int> vTarget;
    //给 vTarget 开辟空间
    vTarget.resize(vSource.size());
    //将 vSource 中的元素搬运到 vTarget
    vector<int>::iterator it = transform(vSource.begin(), vSource.end(),
vTarget.begin(), transformTest01());
    //打印
    for_each(vTarget.begin(), vTarget.end(), print01()); cout << endl;

}

//将容器 1 和容器 2 中的元素相加放入到第三个容器中
struct transformTest02{
    int operator()(int v1,int v2){
        return v1 + v2;
    }
};
void test02(){
```

```
    vector<int> vSource1;
    vector<int> vSource2;
    for (int i = 0; i < 10; i++){
        vSource1.push_back(i + 1);
    }

    //目标容器
    vector<int> vTarget;
    //给 vTarget 开辟空间
    vTarget.resize(vSource1.size());
    transform(vSource1.begin(), vSource1.end(), vSource2.begin(),vTarget.begin(),
transformTest02());
    //打印
    for_each(vTarget.begin(), vTarget.end(), print01()); cout << endl;
}
```

4.4 常用查找算法

```
/*
    find 算法 查找元素
    @param beg 容器开始迭代器
    @param end 容器结束迭代器
    @param value 查找的元素
    @return 返回查找元素的位置
*/
find(iterator beg, iterator end, value)
/*
    find_if 算法 条件查找
    @param beg 容器开始迭代器
    @param end 容器结束迭代器
    @param  callback 回调函数或者谓词(返回 bool 类型的函数对象)
    @return bool 查找返回 true 否则 false
*/
find_if(iterator beg, iterator end, _callback);

/*
    adjacent_find 算法 查找相邻重复元素
    @param beg 容器开始迭代器
    @param end 容器结束迭代器
    @param  _callback 回调函数或者谓词(返回 bool 类型的函数对象)
    @return 返回相邻元素的第一个位置的迭代器
*/
```

```
adjacent_find(iterator beg, iterator end, _callback);
/*
    binary_search 算法 二分查找法
    注意: 在无序序列中不可用
    @param beg 容器开始迭代器
    @param end 容器结束迭代器
    @param value 查找的元素
    @return bool 查找返回 true 否则 false
*/
bool binary_search(iterator beg, iterator end, value);
/*
    count 算法 统计元素出现次数
    @param beg 容器开始迭代器
    @param end 容器结束迭代器
    @param  value 回调函数或者谓词(返回 bool 类型的函数对象)
    @return int 返回元素个数
*/
count(iterator beg, iterator end, value);
/*
    count_if 算法 统计元素出现次数
    @param beg 容器开始迭代器
    @param end 容器结束迭代器
    @param  callback 回调函数或者谓词(返回 bool 类型的函数对象)
    @return int 返回元素个数
*/
count_if(iterator beg, iterator end, _callback);
```

4.5 常用排序算法

```
/*
    merge 算法 容器元素合并，并存储到另一容器中
    注意:两个容器必须是有序的
    @param beg1 容器 1 开始迭代器
    @param end1 容器 1 结束迭代器
    @param beg2 容器 2 开始迭代器
    @param end2 容器 2 结束迭代器
    @param dest  目标容器开始迭代器
*/
merge(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest)
/*
    sort 算法 容器元素排序
    @param beg 容器 1 开始迭代器
```

```
    @param end 容器 1 结束迭代器
    @param _callback 回调函数或者谓词(返回 bool 类型的函数对象)
*/
sort(iterator beg, iterator end, _callback)
/*
    random_shuffle 算法 对指定范围内的元素随机调整次序
    @param beg 容器开始迭代器
    @param end 容器结束迭代器
*/
random_shuffle(iterator beg, iterator end)
/*
    reverse 算法 反转指定范围的元素
    @param beg 容器开始迭代器
    @param end 容器结束迭代器
*/
reverse(iterator beg, iterator end)
```

4.6 常用拷贝和替换算法

```
/*
    copy 算法 将容器内指定范围的元素拷贝到另一容器中
    @param beg 容器开始迭代器
    @param end 容器结束迭代器
    @param dest 目标起始迭代器
*/
copy(iterator beg, iterator end, iterator dest)
/*
    replace 算法 将容器内指定范围的旧元素修改为新元素
    @param beg 容器开始迭代器
    @param end 容器结束迭代器
    @param oldvalue 旧元素
    @param newvalue 新元素
*/
replace(iterator beg, iterator end, oldvalue, newvalue)
/*
    replace_if 算法 将容器内指定范围满足条件的元素替换为新元素
    @param beg 容器开始迭代器
    @param end 容器结束迭代器
    @param callback 函数回调或者谓词(返回 Bool 类型的函数对象)
    @param oldvalue 新元素
*/
replace_if(iterator beg, iterator end, _callback, newvalue)
/*
```

```
    swap 算法 互换两个容器的元素
    @param c1 容器 1
    @param c2 容器 2
*/
swap(container c1, container c2)
```

4.7 常用算数生成算法

```
/*
    accumulate 算法 计算容器元素累计总和
    @param beg 容器开始迭代器
    @param end 容器结束迭代器
    @param value 累加值
*/
accumulate(iterator beg, iterator end, value)
/*
    fill 算法 向容器中添加元素
    @param beg 容器开始迭代器
    @param end 容器结束迭代器
    @param value t 填充元素
*/
fill(iterator beg, iterator end, value)
```

4.8 常用集合算法

```
/*
    set_intersection 算法 求两个 set 集合的交集
    注意:两个集合必须是有序序列
    @param beg1 容器 1 开始迭代器
    @param end1 容器 1 结束迭代器
    @param beg2 容器 2 开始迭代器
    @param end2 容器 2 结束迭代器
    @param dest  目标容器开始迭代器
    @return 目标容器的最后一个元素的迭代器地址
*/
set_intersection(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest)
/*
    set_union 算法 求两个 set 集合的并集
    注意:两个集合必须是有序序列
```

```
    @param beg1 容器1开始迭代器
    @param end1 容器1结束迭代器
    @param beg2 容器2开始迭代器
    @param end2 容器2结束迭代器
    @param dest  目标容器开始迭代器
    @return 目标容器的最后一个元素的迭代器地址
*/
set_union(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest)
/*
    set_difference 算法 求两个set集合的差集
    注意:两个集合必须是有序序列
    @param beg1 容器1开始迭代器
    @param end1 容器1结束迭代器
    @param beg2 容器2开始迭代器
    @param end2 容器2结束迭代器
    @param dest  目标容器开始迭代器
    @return 目标容器的最后一个元素的迭代器地址
*/
set_difference(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest)
```