

# STL 库部分内容介绍

雷凯翔

重庆市巴蜀中学

2013 年 11 月 13 日（水）午前

## 1 简介

- 何谓 STL
- STL 的组成

## 2 在介绍 STL 之前

- 模板
- 指针
- 命名空间

## 3 算法

- 序列操作
- 排序
- 二分搜索及其他算法

## 4 简单容器和适配器容器

- pair
- bitset
- priority\_queue

## 5 串行容器与关系容器

- vector
- set, multiset
- map, multimap

## 6 终

- 更高级的容器
- 小广告

## 何谓 STL

标准模板库 (Standard Template Library)

C++ 软件库，也是 C++ 标准程序库的一部分。

下文仅涉及 C++98 标准。

不涉及 C++0x 标准新增的特性。

# 组成

算法 就是算法 ^w^

容器 数据结构 ( "▽" )

迭代器 泛化指针。可以辅助容器（数据结构）

函数 重载了 ( ) 的 **struct/class**。用于扩展算法。

适配器 带有数据类型参数的类模板，用来扩展其他组件。  
还有一部分数据结构也在其中。

# 算法的组成

序列操作 reverse, rotate, unique, random\_shuffle, etc.

分组操作 partition, etc.

排序 sort, stable\_sort, etc.

二分搜索 lower\_bound, upper\_bound, etc.

归并 merge, etc.

堆化 push\_heap, pop\_heap, make\_heap, etc.

(建议直接使用容器适配器 priority\_queue。)

最值 max\_element, etc.

其他 next\_permutation, etc.

## 容器的组成

细分为

串行容器 vector, deque 和 list

关系容器 set, multiset, map 和 multimap

其他 例: pair, bitset, valarray, etc.

## 迭代器的组成

输入迭代器 ⊃ 单向迭代器 ⊃ 双向迭代器 ⊃ 随机迭代器 (1)

输入输出迭代器相当于指向输入输出流的指针。

单向迭代器相当于指向链表的指针。

双向迭代器相当于指向双向链表的指针。

随机迭代器相当于指向数组的指针。

传统意义的指针就是随机迭代器。

指针的概念等会儿会简要介绍。

## 适配器的组成

细分为

容器适配器 `stack`, `queue`, `priority_queue`, etc. (从容器 `vector`, `deque`, `list` 适配。)

迭代器适配器 逆向迭代器和插入迭代器。(从迭代器适配。)

函数适配器 否定器、绑定器、函数指针适配器等。(从函数适配。)

# 模板

将具有多种不同数据类型，但操作（运算）相同的多个函数/类合并为一个。

分为模板函数和模板类。

声明的方法均为在函数和类前加 **template** 关键字。

```

1 template<class type_1, class type_2, ...>
2 return_type function_name( ...) {
3     // blahblah
4 }
5 template<class type_1, class type_2, ...>
6 struct/class name {
7     // blahblah
8 };
  
```

# 模板的应用

例如求最大值

```

1 int max(int x, int y) {
2     return x > y ? x : y;
3 }
4 long long max(long long x, long long y) {
5     return x > y ? x : y;
6 }

```

用模板书写

```

1 template<class T>
2 T max(T x, T y) {
3     return x > y ? x : y;
4 }

```

## 模板的应用 con't

还可用于一些常用的函数，如交换

```
1  template<class T_1>
2  void swap(T_1& x, T_1& y) {
3      T_1 z = x;
4      x = y;
5      y = z;
6  }
```

还有读入优化（对于浮点数，下代码无法读出小数点及其之后的内容）

```
1  template<class T1>
2  void Get(T1& T) {
3      char C; bool F = 0;
4      for (; C = getchar(), C < '0' || C > '9'; ) if (C == '-') F = 1;
5      for (T = C - '0'; C = getchar(), C >= '0' && C <= '9'; T = T * 10 + C - '0');
6      F && (T = -T);
7  }
```

# 模板的应用 con't

杨至轩学长以前还给我看过一个模板的搞笑应用——二维线段树。

现在找不到当时的源代码了，映象中大致是

```

1  #include <cstdio>
2
3  template<class T>
4  struct seg {
5      T pool[105];
6      int top;
7      int root;
8
9      int build(int l, int r) {}
10     // ...
11 };
12
13 seg<seg<int>> > seg2d;
```

我们甚至可以轻松炮制出一个高维线段树

```

1  seg<seg<seg<int>>> > seg3d;
```

## 指针

程序中定义的变量，会被存放至内存。内存区的每一个字节都有编号，这个编号就是地址，即指针。

存放地址的变量就是指针变量。

打个比方，如果内存是个巨大的数组 `memory`，那么指针就是各个元素的下标。图中 `memory[1008]` 就是变量 `b`，而存储地址 1008 的 `memory[2]` 就是指向 `b` 的指针变量。

指针变量也有数据类型，且与其所指向的变量的类型相同。

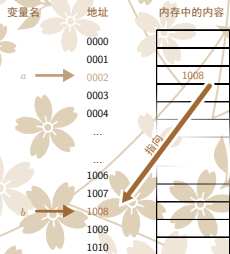


图 1: `a` 为指针变量，指向变量 `b`，并与 `b` 有相同的类型

# 指针的声明与运算

声明指向 `type` 类型的指针变量 `name1, name2, ...` 的语法为

```
1 type *name_1, *name_2, ...;
```

如上图有一个指向整型的指针变量 `a`。

```
1 int *a;
```

## 指针的声明与运算 con't

C++ 提供了两种指针运算  $\&$ （取地址）， $*$ （取值）。

前者表示已知变量取地址，后者表示已知地址取变量。两者互为逆运算。

如上图， $\&b$  与指针类型 1008（即  $a$ ）等价； $*a$  与  $b$  等价。

前面定义了指针变量  $a$ ，但未对其赋初值（地址）。这样就可以使其指向  $b$

```
1  int *a;  
2  a = &b;
```

## 指针的声明与运算 con't

除此之外，C++ 还提供了（a, b 为 `int*`；x, n 为 `int`。）

初始化和赋值运算 eg. `int *a = b; a = b;`

间接引用 对指向的元素赋值。eg. `*a = x`

加减 将指针指向前/后第  $n$  个元素；地址做差

eg. `a ++, a + n, a - n, a += n, a -= n, b - a`

比较 比较地址 eg. `a < b, a <= b, a > b, a >= b,`

`a == b, a != b`

偏移取值 `a[n]` 等价于 `*(a + n)`

## 指针的例子

```
1  int *p;  
2  int a = 520;  
3  p = &a;  
4  printf( "%p->%d\n", p, *p );  
5  *p = 1314;  
6  printf( "%p->%d\n", p, *p );  
7  printf( "a=%d", a );  
8  /*  
9  0022FF78 -> 520  
10 0022FF78 -> 1314  
11 a = 1314  
12 */
```

程序输出如注释所示。当执行了 `p=&a` 后，存取 `*p` 就相当于存取变量 `a` 了。

## 指针与数组

数组在内存中是连续存储的。C++ 规定，数组名表示数组第一个元素的地址。

例如定义了

```
1 int a[5];
```

由于指针可以进行加减运算。 $p + x$  表示指针  $p$  往后移动  $x$  个元素后的指针。

因此  $a + x$  与  $\&a[x]$  是等价的。

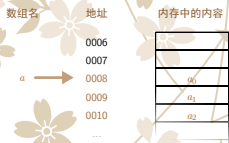


图 2: 数组  $a$

## 指针与数组 con't

这也就是一些人书写

```
1 for(int i = 1; i <= n; i ++)  
2     scanf("%d", a + i);
```

的原因，其实质就是

```
1 for(int i = 1; i <= n; i ++)  
2     scanf("%d", &a[i]);
```

## 指针与数组 con't

```

1  #include <cstdio>
2  #include <cstring>
3  void init( int a[5], int b ) {
4      // memset(a, 0, sizeof a); // wrong
5      memset(a, 0, sizeof (int[5])); // ok
6      for (int i=0; i<5; i=i+1 ) a[i]=i;
7      b = 5;
8  }
9  int main() {
10     int i, a[5], b=0;
11     init( a, b );
12     for ( i=0; i<5; i=i+1 ) printf("%d_",a[i]);
13     printf( "%d\n", b );
14     return 0;
15 }
```

函数 *init* 能改变数组 *a* 原因就在于，参数中的 *a* 是指针而非数组，并且指向 *main* 中的 *a<sub>0</sub>*，即原始的数据。

第 4 行 *memset* 函数不能正确清零的原因是 *sizeof a* 是在计算整型指针的大小，而非数组的大小。

换成第 5 行的 *sizeof (int[5])* 即可正常工作。

## 指针的应用

在读入字符串时，我习惯以下标 1 开始而非以 0 开始。只需对字符串首指针加一即可实现。

```
1  #include <stdio>
2  #include <cstring>
3  char s[1005];
4  int n;
5  int main() {
6      scanf("%s", s + 1);
7      n = strlen(s + 1);
8      for(int i = 1; i <= n; i++)
9          printf("%c\n", s[i]);
10     return 0;
11 }
```

## 指针的应用 con't

倍增法求后缀数组的时候，需要交换两个数组。一个元素一个元素地换浪费时间，故交换首元素指针即可。

```

1  int wa[maxn],wb[maxn],wv[maxn],ws[maxn];
2  int cmp(int *r,int a,int b,int l)
3  {return r[a]==r[b]&& r[a+1]==r[b+1];}
4  void da(int *r,int *sa,int n,int m) {
5      int i,j,p,*x=wa,*y=wb,*t;
6      for(i=0;i<m;i++) ws[i]=0;
7      for(i=0;i<n;i++) ws[x[i]=r[i]]++;
8      for(i=1;i<m;i++) ws[i]+=ws[i-1];
9      for(i=n-1;i>=0;i--) sa[--ws[x[i]]]=i;
10     for(j=1,p=1;p<n;j*=2,m=p) {
11         for(p=0,i=n-j;i<n;i++) y[p++]=i;
12         for(i=0;i<n;i++) if(sa[i]>=j) y[p++]=sa[i]-j;
13         for(i=0;i<n;i++) wv[i]=x[y[i]];
14         for(i=0;i<m;i++) ws[i]=0;
15         for(i=0;i<n;i++) ws[wv[i]]++;
16         for(i=1;i<m;i++) ws[i]+=ws[i-1];
17         for(i=n-1;i>=0;i--) sa[--ws[wv[i]]]=y[i];
18         for(t=x,x=y,y=t,p=1,x[sa[0]]=0,i=1;i<n;i++)
19             x[sa[i]]=cmp(y,sa[i-1],sa[i],j)?p-1:p++;
20     }
21     return;
22 }
```

## 指针与迭代器

STL 中很多函数的参数就是迭代器。指针就可以充当迭代器。

### 如排序

```
1 sort(a + 1, a + n + 1);
```

可完成  $a_1, a_2, \dots, a_n$  的排序。此处  $a + 1, a + n + 1$  就是两个指针，充当迭代器。

# 命名空间及定义

将全局域分成若干用户命名的域，防止重名冲突。

定义格式

```
1 namespace identifier
2 {
3     //blah blah
4 }
```

例如

```
1 namespace myNamespace
2 {
3     int a, b;
4 }
```

# 命名空间的使用

访问命名空间中的变量时，使用域操作符::

```
1 myNamespace::a  
2 myNamespace::b
```

# 如何防重名

```

1  // namespaces
2  #include <iostream>
3  using namespace std;
4
5  namespace first
6  {
7      int var = 5;
8  }
9
10 namespace second
11 {
12     double var = 3.1416;
13 }
14
15 int main () {
16     cout << first::var << endl;
17     cout << second::var << endl;
18     return 0;
19 }
20 /*
21 5
22 3.1416
23 */

```

# using 关键字

将命名空间中的某一个变量推到当前作用域中

```

1 // using
2 #include <iostream>
3 using namespace std;
4 namespace first {
5     int x = 5;
6     int y = 10;
7 }
8 namespace second {
9     double x = 3.1416;
10    double y = 2.7183;
11 }
12 int main () {
13     using first::x;
14     using second::y;
15     cout << x << endl;
16     cout << y << endl;
17     cout << first::y << endl;
18     cout << second::x << endl;
19     return 0;
20 }
21 /*
22 5
23 2.7183
24 10
25 3.1416
26 */

```

## using 关键字 con't

```

1  // using
2  #include <iostream>
3  using namespace std;
4  namespace first {
5      int x = 5;
6      int y = 10;
7  }
8  namespace second {
9      double x = 3.1416;
10     double y = 2.7183;
11 }
12 int main () {
13     using namespace first;
14     cout << x << endl;
15     cout << y << endl;
16     cout << second::x << endl;
17     cout << second::y << endl;
18     return 0;
19 }
20 /*
21 5
22 10
23 3.1416
24 2.7183
25 */

```

# using 关键字 con't

```

1 // using namespace example
2 #include <iostream>
3 using namespace std;
4 namespace first {
5     int x = 5;
6 }
7 namespace second {
8     double x = 3.1416;
9 }
10 int main () {
11     {
12         using namespace first;
13         cout << x << endl;
14     }
15     {
16         using namespace second;
17         cout << x << endl;
18     }
19     return 0;
20 }
21 /*
22 5
23 3.1416
24 */

```

# 命名空间 std

几乎所有 STL 库，输入输出流等都存放在 std 中。

这即是大家写

```
1 using namespace std;
```

的原因。

# 命名空间的应用

SDOI2012 棋盘覆盖 (BSOJ 3225, BZOJ 2706) 一题中，要求在一个文件里书写三个截然不同的算法。

重名问题难免。

使用命名空间即可避免重名冲突。

# FBI Warning

## 使用本节的函数可能需

```
1 #include <algorithm>  
2 using namespace std;
```

## remove

原型 `template <class Iter, class T>``Iter remove ( Iter first,  
Iter last, const T& val)`

作用 将位于  $[first, last)$  中的, 值为  $val$  的元素删除。返回区间结束的位置。

注意今后所有的区间都是  $begin$  闭  $end$  开。

时间复杂度 关于  $last - first$  线性。

## remove con't

```

1 // remove algorithm example
2 #include <iostream> // std::cout
3 #include <algorithm> // std::remove
4
5 int main () {
6     int myints[] = {10,20,30,30,20,10,10,20}; // 10 20 30 30 20 10 10 20
7
8     // bounds of range:
9     int* pbegin = myints; // ~
10    int* pend = myints+sizeof(myints)/sizeof(int); // ~
11
12    pend = std::remove (pbegin, pend, 20); // 10 30 30 10 10 ? ? ?
13    // ~
14    std::cout << "range contains: ";
15    for (int* p=pbegin; p!=pend; ++p)
16        std::cout << ' ' << *p;
17    std::cout << '\n';
18
19    return 0;
20 }
21
22 // output range contains: 10 30 30 10 10

```

# unique

原型 `template <class Iter>`

`Iter unique (Iter first, Iter last);`

`template <class Iter, class Pred>`

`ForwardIterator unique (Iter first,  
Iter last, Pred pred);`

作用 删除区间  $[first, last)$  相邻重复元素，返回结束位置。

$pred$  是二元谓词函数（有两个参数的 `bool` 型函数。）返回 `true` 当且仅当两者等价。

时间复杂度 关于比  $last - first$  更低的值线性。

## unique con't

```

1 // unique algorithm example
2 #include <iostream> // std::cout
3 #include <algorithm> // std::unique, std::distance
4 #include <vector> // std::vector
5 bool myfunction (int i, int j) {
6     return (i==j);
7 }
8 int main () {
9     int myints[] = {10,20,20,20,30,30,20,20,10}; // 10 20 20 20 30 30 20 20 10
10    std::vector<int> myvector (myints,myints+9);
11
12    // using default comparison:
13    std::vector<int>::iterator it;
14    it = std::unique (myvector.begin(), myvector.end()); // 10 20 30 20 10 ? ? ? ?
15    //
16
17    myvector.resize( std::distance(myvector.begin(),it) ); // 10 20 30 20 10
18
19    // using predicate comparison:
20    std::unique (myvector.begin(), myvector.end(), myfunction); // (no changes)
21
22    // print out content:
23    std::cout << "myvector contains:";
24    for (it=myvector.begin(); it!=myvector.end(); ++it)
25        std::cout << ' ' << *it;
26    std::cout << '\n';
27    return 0;
28 }
29 // output range contains: 10 20 30 20 10

```

## unique 的应用

有时题目会要求从小到大不重复地输出答案。

若所有答案均已求出，但可能有重复时，可以先排序（使用后面的 sort）再使用 unique。

离散化时也可能用到此函数。

## reverse

原型 `template <class Iter>``void reverse (Iter first, Iter last);`作用 区间  $[first, last)$  反转。时间复杂度 关于  $last - first$  的一半线性。

## reverse con't

```

1 // reverse algorithm example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::reverse
4 #include <vector>             // std::vector
5
6 int main () {
7     std::vector<int> myvector;
8
9     // set some values:
10    for (int i=1; i<10; ++i) myvector.push_back(i); // 1 2 3 4 5 6 7 8 9
11
12    std::reverse(myvector.begin(),myvector.end()); // 9 8 7 6 5 4 3 2 1
13
14    // print out content:
15    std::cout << "myvector contains: ";
16    for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
17        std::cout << ' ' << *it;
18    std::cout << '\n';
19
20    return 0;
21 }
22
23 // output range contains: 9 8 7 6 5 4 3 2 1

```

## reverse 的应用

一些计算几何算法要求保证多边形是逆时针的（右手系）。  
如果叉起来面积为负，那么直接执行 `reverse` 将其化为逆时针。

# rotate

原型 `template <class Iter>`

```
void rotate (Iter first, Iter middle,
             Iter last);
```

作用 区间  $[first, last)$  中  $[first, middle)$  滚动到  $[middle, last)$  后。

时间复杂度 关于  $last - first$  线性。

## rotate con't

```

1 // rotate algorithm example
2 #include <iostream>      // std::cout
3 #include <algorithm>      // std::rotate
4 #include <vector>         // std::vector
5
6 int main () {
7     std::vector<int> myvector;
8
9     // set some values:
10    for (int i=1; i<10; ++i) myvector.push_back(i); // 1 2 3 4 5 6 7 8 9
11
12    std::rotate(myvector.begin(),myvector.begin()+3,myvector.end());
13                                     // 4 5 6 7 8 9 1 2 3
14    // print out content:
15    std::cout << "myvector contains: ";
16    for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
17        std::cout << ' ' << *it;
18    std::cout << '\n';
19
20    return 0;
21 }
22
23 // output range contains: 4 5 6 7 8 9 1 2 3

```

## rotate 的应用

旋转卡壳等要求从某一指定点开始卡一周时，可用 `rotate` 将指定点转到第一个点。

以上两个函数对于计算几何中多边形的预处理等可以使用。

# sort

原型 `template <class Iter>`

`void sort (Iter first, Iter last);`

`template <class Iter, class Compare>`

`void sort (Iter first, Iter last,  
Compare comp);`

作用 区间  $[first, last)$  排序。

*comp* 是二元谓词函数，表明严格弱序（strict weak ordering）中，前者是否比后者的位置靠前（<）。

时间复杂度 关于  $last - first$  线性对数。

## sort con't

```

1 // sort algorithm example
2 #include <iostream> // std::cout
3 #include <algorithm> // std::sort
4 #include <vector> // std::vector
5 bool myfunction (int i,int j) { return (i<j); }
6 struct myclass {
7     bool operator() (int i,int j) { return (i<j);}
8 } myobject; // <<< 仿函数
9 int main () {
10     int myints[] = {32,71,12,45,26,80,53,33};
11     std::vector<int> myvector (myints, myints+8); // 32 71 12 45 26 80 53 33
12     // using default comparison (operator <):
13     std::sort (myvector.begin(), myvector.begin()+4); // (12 32 45 71) 26 80 53 33
14     // using function as comp
15     std::sort (myvector.begin()+4, myvector.end(), myfunction); // 12 32 45 71 (26 33 53 80)
16     // using object as comp
17     std::sort (myvector.begin(), myvector.end(), myobject); // (12 26 32 33 45 53 71 80)
18     // print out content:
19     std::cout << "myvector contains: ";
20     for (std::vector<int>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
21         std::cout << ' ' << *it;
22     std::cout << '\n';
23     return 0;
24 }
25
26 // output myvector contains: 12 26 32 33 45 53 71 80
    
```

# sort 的应用

QwQ 你想得到的地方都能用……

## stable\_sort

原型 `template <class Iter>`  
`void stable_sort ( Iter first,`  
`Iter last );`  
`template <class Iter, class Compare>`  
`void stable_sort ( Iter first,`  
`Iter last, Compare comp );`

作用 区间  $[first, last)$  稳定排序。*comp* 的意义同 `sort` 中的。

时间复杂度 内存足够时，关于  $last - first$  线性对数；否则关于  $last - first$  线性多项式对数 ( $O(N \log^2 N)$ )

# stable\_sort con't

```

1 // stable_sort example
2 #include <iostream>      // std::cout
3 #include <algorithm>     // std::stable_sort
4 #include <vector>        // std::vector
5 bool compare_as_ints (double i, double j)
6 {
7     return (int(i)<int(j));
8 }
9 int main () {
10     double mydoubles[] = {3.14, 1.41, 2.72, 4.67, 1.73, 1.32, 1.62, 2.58};
11     std::vector<double> myvector;
12     myvector.assign(mydoubles, mydoubles+8);
13     std::cout << "using default comparison:";
14     std::stable_sort (myvector.begin(), myvector.end());
15     for (std::vector<double>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
16         std::cout << ' ' << *it;
17     std::cout << '\n';
18     myvector.assign(mydoubles, mydoubles+8);
19     std::cout << "using 'compare_as_ints'";
20     std::stable_sort (myvector.begin(), myvector.end(), compare_as_ints);
21     for (std::vector<double>::iterator it=myvector.begin(); it!=myvector.end(); ++it)
22         std::cout << ' ' << *it;
23     std::cout << '\n';
24     return 0;
25 }
26
27 // using default comparison: 1.32 1.41 1.62 1.73 2.58 2.72 3.14 4.67
28 // using 'compare_as_ints' : 1.41 1.73 1.32 1.62 2.72 2.58 3.14 4.67

```

## stable\_sort 的应用

2012 年 THUSC 的集训中有一道交互题。一部分测试点要求对序列排序，并尽可能少的询问两者的大小关系。

按照询问系统的次数评分。

事实证明，stable\_sort 访问比较器的次数比 sort 要少。

然后……然后就没有然后了……TwT

# lower\_bound, upper\_bound

## 原型

```
template <class Iter, class T>
    Iter lower_bound (Iter first, Iter last, const T& val);
template <class Iter, class T, class Comp>
    Iter lower_bound (Iter first, Iter last, const T& val, Comp comp);
```

## 作用

对已经排序的区间  $[first, last)$ ，返回第一个不低于  
 $val$  的元素的迭代器（指针）。  
 $comp$  是二元谓词函数，表面前者的位置是否比后者靠前。

## 时间复杂度

只有  $Iter$  为随机迭代器时，才关于  $last - first$  对数；否则关于  $last - first$  线性。

# lower\_bound, upper\_bound con't

## 原型

```
template <class Iter, class T>
    Iter upper_bound (Iter first, Iter last, const T& val);
template <class Iter, class T, class Comp>
    Iter upper_bound (Iter first, Iter last, const T& val, Comp comp);
```

**作用** 对已经排序的区间  $[first, last)$ ，返回第一个大于  $val$  的元素的迭代器（指针）。

$comp$  的意义同 `lower_bound` 中的。

**时间复杂度** 只有  $Iter$  为随机迭代器时，才关于  $last - first$  对数；否则关于  $last - first$  线性。

# lower\_bound, upper\_bound con't

```

1 // lower_bound/upper_bound example
2 #include <iostream>           // std::cout
3 #include <algorithm>          // std::lower_bound, std::upper_bound, std::sort
4 #include <vector>             // std::vector
5
6 int main () {
7     int myints[] = {10,20,30,30,20,10,10,20};
8     std::vector<int> v(myints,myints+8);           // 10 20 30 30 20 10 10 20
9
10    std::sort (v.begin(), v.end());                 // 10 10 10 20 20 20 30 30
11
12    std::vector<int>::iterator low,up;
13    low=std::lower_bound (v.begin(), v.end(), 20); // 1
14    up= std::upper_bound (v.begin(), v.end(), 20); // 2
15
16    std::cout << "lower_bound at position " << (low - v.begin()) << '\n';
17    std::cout << "upper_bound at position " << (up - v.begin()) << '\n';
18
19    return 0;
20 }
21
22 // lower_bound at position 3
23 // upper_bound at position 6

```

# lower\_bound, upper\_bound can't

## 放弃治疗型二分答案

```

1  #include<cstdio>
2  #include<algorithm>
3  #include<iterator>
4  using namespace std;
5  bool gao(int x)
6  {
7      return x < 9;
8  }
9
10 struct FunIt : public iterator<random_access_iterator_tag, int>
11 {
12     int n;
13     FunIt(int x) : n(x) {}
14     int operator -(const FunIt& another) {return n - another.n;}
15     int operator ++() {n++; return n;}
16     int operator +=(int d) {n += d; return n;}
17     bool operator *() {return ! gao(n);}
18 };
19
20 int main()
21 {
22     printf("The minimal x where x >= 9 is x = %d.\n",
23         std::lower_bound((FunIt)0, (FunIt)0x2f2f2f2f, 0.5).n);
24 }
25 // The minimal x where x >= 9 is x = 9.

```

# next\_permutation

原型

```
template <class Iter>
    bool next_permutation (Iter first, Iter last);
template <class Iter, class Comp>
    bool next_permutation (Iter first, Iter last, Comp comp);
```

作用 求区间  $[first, last)$  下一个排列。返回是否成功。  
 $comp$  的意义同 `sort` 中的。

时间复杂度 关于  $last - first$  的一半线性。

## next\_permutation can't

```

1 // lower_bound/upper_bound example
2 // next_permutation example
3 #include <iostream> // std::cout
4 #include <algorithm> // std::next_permutation, std::sort
5
6 int main () {
7     int myints[] = {1,2,3};
8
9     std::sort (myints,myints+3);
10
11     std::cout << "The 3! possible permutations with 3 elements:\n";
12     do {
13         std::cout << myints[0] << ' ' << myints[1] << ' ' << myints[2] << '\n';
14     } while ( std::next_permutation(myints,myints+3) );
15
16     std::cout << "After loop:" << myints[0] << ' ' << myints[1] << ' ' << myints[2] << '\n';
17
18     return 0;
19 }
20 /*
21 The 3! possible permutations with 3 elements:
22 1 2 3
23 1 3 2
24 2 1 3
25 2 3 1
26 3 1 2
27 3 2 1
28 After loop: 1 2 3
29 */

```

# next\_permutation 的应用

NOIp 2004 火星人。

POJ 1256 Anagram。

.....

## FBI Warning

### 使用本节的函数/类可能需

```
1 #include <utility>
2 using std::pair;
3 using std::make_pair;
```

# pair

原型 `template <class T1, class T2>`  
`struct pair;`

实质 两个元素合成的一个结构。前者类型 T1 后者 T2。

周边 成员 first, second 两个元素。

比较运算符 `>, ≥, <, ≤, =, ≠`: 参照字典序。

`make_pair` 函数 构建 pair

## pair 的构造函数

```
1 // pair::pair example
2 #include <utility> // std::pair, std::make_pair
3 #include <string> // std::string
4 #include <iostream> // std::cout
5
6 int main () {
7     std::pair <std::string,double> product1; // default constructor
8     std::pair <std::string,double> product2 ("tomatoes",2.30); // value init
9     std::pair <std::string,double> product3 (product2); // copy constructor
10
11     product1 = std::make_pair(std::string("lightbulbs"),0.99); // using make_pair (move)
12
13     product2.first = "shoes"; // the type of first is string
14     product2.second = 39.90; // the type of second is double
15
16     std::cout << "The price of " << product1.first << " is $" << product1.second << '\n';
17     std::cout << "The price of " << product2.first << " is $" << product2.second << '\n';
18     std::cout << "The price of " << product3.first << " is $" << product3.second << '\n';
19     return 0;
20 }
21 // The price of lightbulbs is $0.99
22 // The price of shoes is $39.9
23 // The price of tomatoes is $2.3
```

# 比较运算符

## 原型

```
template <class T1, class T2>
    bool operator== (const pair<T1,T2>& lhs, const pair<T1,T2>& rhs);
template <class T1, class T2>
    bool operator!= (const pair<T1,T2>& lhs, const pair<T1,T2>& rhs);
template <class T1, class T2>
    bool operator< (const pair<T1,T2>& lhs, const pair<T1,T2>& rhs);
template <class T1, class T2>
    bool operator<= (const pair<T1,T2>& lhs, const pair<T1,T2>& rhs);
template <class T1, class T2>
    bool operator> (const pair<T1,T2>& lhs, const pair<T1,T2>& rhs);
template <class T1, class T2>
    bool operator>= (const pair<T1,T2>& lhs, const pair<T1,T2>& rhs);
```

## 作用 比较运算符。

# 比较运算符 con't

```

1 // pair relational operators
2 #include <utility>           // std::pair
3 #include <iostream>         // std::cout
4
5 int main ()
6 {
7     std::pair<int, char> foo (10, 'z');
8     std::pair<int, char> bar (90, 'a');
9
10    if (foo==bar) std::cout << "foo and bar are equal\n";
11    if (foo!=bar) std::cout << "foo and bar are not equal\n";
12    if (foo< bar) std::cout << "foo is less than bar\n";
13    if (foo> bar) std::cout << "foo is greater than bar\n";
14    if (foo<=bar) std::cout << "foo is less than or equal to bar\n";
15    if (foo>=bar) std::cout << "foo is greater than or equal to bar\n";
16
17    return 0;
18 }
19 // foo and bar are not equal
20 // foo is less than bar
21 // foo is less than or equal to bar

```

# make\_pair

原型 `template <class T1, class T2>`  
`pair<T1,T2>`  
`make_pair (T1 x, T2 y);`

作用 生成一个 pair。

# make\_pair can't

```

1 // make_pair example
2 #include <utility>           // std::pair
3 #include <iostream>         // std::cout
4
5 int main () {
6     std::pair <int,int> foo;
7     std::pair <int,int> bar;
8
9     foo = std::make_pair (10,20);
10    bar = std::make_pair (10.5,'A'); // ok: implicit conversion from pair<double,char>
11
12    std::cout << "foo: " << foo.first << ", " << foo.second << '\n';
13    std::cout << "bar: " << bar.first << ", " << bar.second << '\n';
14
15    return 0;
16 }
17 // foo: 10, 20
18 // bar: 10, 65

```

## 应用

按字典序排序。

或者先保证某一个值最大再保证第二个值最大。

```
1 pair<int, int> ans;
2 ans = max(ans, make_pair(a, b));
```

先保证某一个值最大再保证第二个值最大再保证第三个值最大。

```
1 pair<int, pair<int, int>> ans;
2 ans = max(ans, make_pair(a,
3               make_pair(b, c)));
```

# FBI Warning

## 使用本节的函数/类可能需

```
1 #include <bitset>
2 using std::bitset;
```

# bitset

原型 `template <size_t N> class bitset;`

实质 储存并运算任意长度的二进制数。（位运算高精度）  
 $N$  为二进制数长度。

周边 成员运算符 `[]` 访问元素

成员函数 `test` 访问元素（有边界检测）

成员函数 `count` 返回 1 的数量。

位运算重载

成员函数 `to_string` 转为 `string` 类型。

# bitset 构造函数

```

1 // constructing bitsets
2 #include <iostream>           // std::cout
3 #include <string>             // std::string
4 #include <bitset>             // std::bitset
5
6 int main ()
7 {
8     std::bitset<16> foo;
9     std::bitset<16> bar (0xfa2);
10    std::bitset<16> baz (std::string("0101111001"));
11
12    std::cout << "foo: " << foo << '\n';
13    std::cout << "bar: " << bar << '\n';
14    std::cout << "baz: " << baz << '\n';
15
16    return 0;
17 }
18 // foo: 0000000000000000
19 // bar: 0000111110100010
20 // baz: 0000000101111001

```

[]

原型 `bool operator[] (size_t pos) const;`  
`reference operator[] (size_t pos);`

作用 设置元素。

[] con't

```

1 // bitset::operator[]
2 #include <iostream>           // std::cout
3 #include <bitset>             // std::bitset
4
5 int main ()
6 {
7     std::bitset<4> foo;
8
9     foo[1]=1;                  // 0010
10    foo[2]=foo[1];             // 0110
11
12    std::cout << "foo: " << foo << '\n';
13
14    return 0;
15 }
16 /*
17 foo: 0110
18 */

```

test

原型 `bool test (size_t pos) const;`

作用 访问元素。

## test con't

```

1 //bitset::test
2 #include <iostream>           // std::cout
3 #include <string>              // std::string
4 #include <cstdint>             // std::size_t
5 #include <bitset>              // std::bitset
6
7 int main ()
8 {
9     std::bitset<5> foo (std::string("01011"));
10
11     std::cout << "foo contains:\n";
12     std::cout << std::boolalpha;
13     for (std::size_t i=0; i<foo.size(); ++i)
14         std::cout << foo.test(i) << '\n';
15
16     return 0;
17 }
18 /*
19 foo contains:
20 true
21 true
22 false
23 true
24 false
25 */

```

count

原型 `size_t count() const;`

作用 统计 1(true) 的数量

## count con't

```

1 //bitset::count
2 #include <iostream> //std::cout
3 #include <string> //std::string
4 #include <bitset> //std::bitset
5
6 int main ()
7 {
8     std::bitset<8> foo (std::string("10110011"));
9
10    std::cout << foo << "has ";
11    std::cout << foo.count() << "ones and ";
12    std::cout << (foo.size()-foo.count()) << "zeros.\n";
13
14    return 0;
15 }
16 /*
17 10110011 has 5 ones and 3 zeros.
18 */

```

# 位运算重载

## 原型

```

bitset& operator&= (const bitset& rhs);
bitset& operator|= (const bitset& rhs);
bitset& operator^= (const bitset& rhs);
bitset& operator<<= (size_t pos);
bitset& operator>>= (size_t pos);
bitset operator~() const;
bitset operator<<(size_t pos) const;
bitset operator>>(size_t pos) const;
bool operator== (const bitset& rhs) const;
bool operator!= (const bitset& rhs) const;
template<size_t N>
    bitset<N> operator& (const bitset<N>& lhs, const bitset<N>& rhs);
template<size_t N>
    bitset<N> operator| (const bitset<N>& lhs, const bitset<N>& rhs);
template<size_t N>
    bitset<N> operator^ (const bitset<N>& lhs, const bitset<N>& rhs);
// 输入输出流
template<class charT, class traits, size_t N>
    basic_istream<charT, traits>&
        operator>> (basic_istream<charT, traits>& is, bitset<N>& rhs);
template<class charT, class traits, size_t N>
    basic_ostream<charT, traits>&
        operator<< (basic_ostream<charT, traits>& os, const bitset<N>& rhs);

```

## 作用

○  
○○○○○

○○○○  
○○○○○○○○○○  
○○○○○○○○

○○○○○○○○○○○  
○○○○○  
○○○○○○○

○○○○○○○  
○○○○○○○○○○●○○○  
○○○○○○○○○○○○○

○○○○○○○○○○○○○○○  
○○○○○○○○○○○○○○○  
○○○○○○○○○○○○○○○

○  
○

bitset

## 位运算重载 con't

```

1 // bitset operators
2 #include <iostream>           // std::cout
3 #include <string>             // std::string
4 #include <bitset>             // std::bitset
5 int main ()
6 {
7     std::bitset<4> foo (std::string("1001"));
8     std::bitset<4> bar (std::string("0011"));
9
10    std::cout << (foo^=bar) << '\n';           // 1010 (XOR,assign)
11    std::cout << (foo&=bar) << '\n';           // 0010 (AND,assign)
12    std::cout << (foo|=bar) << '\n';           // 0011 (OR,assign)
13    std::cout << (foo<=2) << '\n';             // 1100 (SHL,assign)
14    std::cout << (foo>=1) << '\n';             // 0110 (SHR,assign)
15    std::cout << (~bar) << '\n';               // 1100 (NOT)
16    std::cout << (bar<<1) << '\n';             // 0110 (SHL)
17    std::cout << (bar>>1) << '\n';             // 0001 (SHR)
18    std::cout << (foo==bar) << '\n';           // false (0110==0011)
19    std::cout << (foo!=bar) << '\n';           // true  (0110!=0011)
20    std::cout << (foo&bar) << '\n';           // 0010
21    std::cout << (foo|bar) << '\n';           // 0111
22    std::cout << (foo^bar) << '\n';           // 0101
23    return 0;
24 }
25 /*
26 1010\n 0010\n 0011\n 1100\n 0110\n 1100\n 0110\n 0001\n 0\n 1\n 0010\n 0111\n 0101\n
27 */

```

to\_string

原型 `template <class charT,  
class traits, class Alloc>  
basic_string<charT,traits,Alloc>  
to_string() const;`

作用 转 string。

## to\_string con't

```

1 // bitset::to_string
2 #include <iostream>           // std::cout
3 #include <string>             // std::string
4 #include <bitset>             // std::bitset
5
6 int main ()
7 {
8     std::bitset<4> mybits;      // mybits: 0000
9     mybits.set();              // mybits: 1111
10
11     std::string mystring =
12         mybits.to_string<char,std::string::traits_type,std::string::allocator_type>();
13
14     std::cout << "mystring: " << mystring << '\n';
15
16     return 0;
17 }
18 /*
19 1111
20 */

```

## 应用

超过 64 bits 的位运算。搜索减枝。常数优化。

NOI 2013 D1T1

## FBI Warning

### 使用本节的函数/类可能需

```
1 #include <queue>
2 using std::priority_queue;
```

# priority\_queue

原型

```
template <class T, class Container = vector<T>,
          class Compare = less<typename Container::value_type>>
class priority_queue;
```

实质

大根堆。*Container* 用 `vector<T>` 就好。*Compare* 表明严格弱序（strict weak ordering）中，前者是否比后者的位置靠前（<），仿函数类型推奖。

周边

成员函数 `empty` 是否为空。

成员函数 `size` 大小

成员函数 `top` 堆顶元素

成员函数 `push` 将……压入堆。

成员函数 `pop` 将堆顶弹出。

## 关于仿函数

仿函数不是函数，是重载了 () 的 **struct/class**。

但调用起来类似函数。

```
1 struct myclass {
2     int operator()(int a, int b) {return a + b;}
3 } myobject;
4 int x = myobject (0, 9);           // function-like syntax with object myobject
```

实践表明，仿函数比函数指针要快一些。

C++ 提供了一些预先定义好的仿函数类型，如大于 (>)

```
1 template <class T> struct greater;
```

存放于

```
1 #include <functional>
```

○  
○○○○○

○○○○○  
○○○○○○○○○○○  
○○○○○○○○○

○○○○○○○○○○○  
○○○○○○○○○  
○○○○○○○

○○○○○○○○○  
○○○○○○○○○○○○○  
○○○●○○○○○○○○○

○○○○○○○○○○○○○○○  
○○○○○○○○○○○○○○○  
○○○○○○○○○○○○○○○○○

○  
○

priority\_queue

## priority\_queue 构造函数

```

1 // constructing priority queues
2 #include <iostream>           // std::cout
3 #include <queue>               // std::priority_queue
4 #include <vector>              // std::vector
5 #include <functional>          // std::greater
6 class mycomparison
7 {
8     bool operator() (const int& lhs, const int&rhs) const
9     {
10         return (lhs<rhs);
11     }
12 };
13 int main ()
14 {
15     int myints[] = {10,60,50,20};
16     std::priority_queue<int> first;
17     std::priority_queue<int> second (myints,myints+4);
18     std::priority_queue<int, std::vector<int>, std::greater<int> >
19         third (myints,myints+4);
20     // using mycomparison:
21     typedef std::priority_queue<int,std::vector<int>,mycomparison> mypq_type;
22     mypq_type fourth;           // less-than comparison
23     return 0;
24 }
```

empty

原型 `bool empty() const;`

作用 是否为空

时间复杂度 常数

empty can't

```

1 // priority_queue::empty
2 #include <iostream>           // std::cout
3 #include <queue>               // std::priority_queue
4
5 int main ()
6 {
7     std::priority_queue<int> mypq;
8     int sum (0);
9
10    for (int i=1;i<=10;i++) mypq.push(i);
11
12    while (!mypq.empty())
13    {
14        sum += mypq.top();
15        mypq.pop();
16    }
17
18    std::cout << "total:␣" << sum << '\n';
19
20    return 0;
21 }
22 // total: 55

```

size

原型 `size_type size() const;`

作用 大小

时间复杂度 常数

○  
○○○○○

○○○○  
○○○○○○○○○○  
○○○○○○○○

○○○○○○○○○○○  
○○○○○○○  
○○○○○○○

○○○○○○○○○  
○○○○○○○○○○○○○  
○○○○○○○○●○○○○○

○○○○○○○○○○○○○○○○○  
○○○○○○○○○○○○○○○○○  
○○○○○○○○○○○○○○○○○○○

○  
○

## priority\_queue

size con't

```

1 // priority_queue::size
2 #include <iostream>           // std::cout
3 #include <queue>               // std::priority_queue
4
5 int main ()
6 {
7     std::priority_queue<int> myints;
8     std::cout << "0. size: " << myints.size() << '\n';
9
10    for (int i=0; i<5; i++) myints.push(i);
11    std::cout << "1. size: " << myints.size() << '\n';
12
13    myints.pop();
14    std::cout << "2. size: " << myints.size() << '\n';
15
16    return 0;
17 }
18 /*
19 0. size: 0
20 1. size: 5
21 2. size: 4
22 */

```

top

原型 `const value_type& top() const;`

作用 堆顶。

时间复杂度 常数

# top con't

```

1 // priority_queue::top
2 #include <iostream>           // std::cout
3 #include <queue>               // std::priority_queue
4
5 int main ()
6 {
7     std::priority_queue<int> mypq;
8
9     mypq.push(10);
10    mypq.push(20);
11    mypq.push(15);
12
13    std::cout << "mypq.top() is now " << mypq.top() << '\n';
14
15    return 0;
16 }
17 /*
18 mypq.top() is now 20
19 */

```

push, pop

原型 `void push (const value_type& val);`  
`void pop();`

作用 压堆/弹堆顶。

时间复杂度 均为  $O(\log N)$ 。  $N$  表示堆的大小

## push, pop con't

```

1 // priority_queue::push/pop
2 #include <iostream>           // std::cout
3 #include <queue>               // std::priority_queue
4
5 int main ()
6 {
7     std::priority_queue<int> mypq;
8
9     mypq.push(30);
10    mypq.push(100);
11    mypq.push(25);
12    mypq.push(40);
13
14    std::cout << "Popping out elements...";
15    while (!mypq.empty())
16    {
17        std::cout << ' ' << mypq.top();
18        mypq.pop();
19    }
20    std::cout << '\n';
21
22    return 0;
23 }
24 /*
25 Popping out elements... 100 40 30 25
26 */

```

# priority\_queue 的应用

配合懒惰删除，可以实现堆化 Dijkstra。

```

1  std::vector<std::pair<int,int> > g[N]; // 邻接表 vector 等会介绍
2  int dis[N];
3  std::priority_queue<std::pair<int,int> > heap;
4  memset(dis, 0x2f, sizeof dis);
5
6  dis[s]=0;
7  heap.push(std::make_pair(-dis[s], s));
8  for(int x, distx; heap.empty() == 0; ) {
9      x = heap.top().second;
10     distx = - heap.top().first;
11     heap.pop();
12     if(distx != dis[x]) continue;
13     for(typeof(g[x].begin()) i = g[x].begin(); i != g[x].end(); i++)
14         if(dis[i -> first] > dis[x] + (i -> second)) {
15             dis[i -> first] = dis[x] + (i -> second);
16             heap.push(std::make_pair(-dis[i -> first], i -> first));
17         }
18 }

```

# FBI Warning

使用本节的函数/类可能需

```
1 #include <vector>
2 using std::vector;
```

## vector

原型

```
template < class T, class Alloc = allocator<T> > class vector;
```

实质

类似于数组，但是可以更改大小。元素类型为 T

周边

成员运算 [] 定位。

成员类型

iterator 随机迭代器。

成员函数

begin, end 头尾迭代器。

头闭尾开。下略

成员函数

size, resize 返回（更改）大小。

成员函数

push\_back, pop\_back

末端插入，末端删除。

成员函数

clear 清空。

## 关于随机迭代器

随机迭代器能实现指针的所有功能。包括在常数时间内

初始化和赋值运算 eg.  $X \ a = b; \ a = b;$

间接引用 对指向的元素赋值。eg.  $*a = x$

加减 将指针指向前/后第  $n$  个元素；地址做差

eg.  $a ++, a + n, a - n, a += n, a -= n, b - a$

比较 比较地址 eg.  $a < b, a \leq b, a > b, a \geq b,$

$a == b, a != b$

偏移取值  $a[n]$  等价于  $*(a + n)$

## vector 构造函数

```

1 // constructing vectors
2 #include <iostream>
3 #include <vector>
4
5 int main ()
6 {
7     unsigned int i;
8
9     // constructors used in the same order as described above:
10    std::vector<int> first; // empty vector of ints
11    std::vector<int> second (4,100); // four ints with value 100
12    std::vector<int> third (second.begin(),second.end()); // iterating through second
13    std::vector<int> fourth (third); // a copy of third
14
15    // the iterator constructor can also be used to construct from arrays:
16    int myints[] = {16,2,77,29};
17    std::vector<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );
18
19    std::cout << "The contents of fifth are:";
20    for (std::vector<int>::iterator it = fifth.begin(); it != fifth.end(); ++it)
21        std::cout << ' ' << *it;
22    std::cout << '\n';
23
24    return 0;
25 }
26 // The contents of fifth are: 16 2 77 29

```

begin, end

原型 `iterator begin();`  
`const_iterator begin() const;`  
`iterator end();`  
`const_iterator end() const;`

作用 首尾迭代器

时间复杂度 常数

## begin, end can't

```

1 // vector::begin/end
2 #include <iostream>
3 #include <vector>
4
5 int main ()
6 {
7     std::vector<int> myvector;
8     for (int i=1; i<=5; i++) myvector.push_back(i);
9
10    std::cout << "myvector contains: ";
11    for (std::vector<int>::iterator it = myvector.begin() ; it != myvector.end(); ++it)
12        std::cout << ' ' << *it;
13    std::cout << '\n';
14
15    return 0;
16 }
17
18 // myvector contains: 1 2 3 4 5

```

## size, resize

原型 `size_type size() const;`  
`void resize (size_type n,`  
`value_type val = value_type());`

作用 询问大小

修改大小为  $n$ 。如果长度增加，多于的用  $val$  填充。

$val$  默认为构造函数（整数为 0），可以不指定。

时间复杂度 前者常数。后者与插入/删除的元素数成正比。

## size, resize con't

```

1 // vector::size
2 #include <iostream>
3 #include <vector>
4
5 int main ()
6 {
7     std::vector<int> myints;
8     std::cout << "0.size:" << myints.size() << '\n';
9
10    for (int i=0; i<10; i++) myints.push_back(i);
11    std::cout << "1.size:" << myints.size() << '\n';
12
13    myints.insert (myints.end(),10,100);
14    std::cout << "2.size:" << myints.size() << '\n';
15
16    myints.pop_back();
17    std::cout << "3.size:" << myints.size() << '\n';
18
19    return 0;
20 }
21 // 0. size: 0
22 // 1. size: 10
23 // 2. size: 20
24 // 3. size: 19

```

## size, resize con't

```

1 // resizing vector
2 #include <iostream>
3 #include <vector>
4
5 int main ()
6 {
7     std::vector<int> myvector;
8
9     // set some initial content:
10    for (int i=1;i<10;i++) myvector.push_back(i);
11
12    myvector.resize(5);
13    myvector.resize(8,100);
14    myvector.resize(12);
15
16    std::cout << "myvector contains:";
17    for (int i=0;i<myvector.size();i++)
18        std::cout << ' ' << myvector[i];
19    std::cout << '\n';
20
21    return 0;
22 }
23 // myvector contains: 1 2 3 4 5 100 100 100 0 0 0 0

```

## push\_back, pop\_back

原型 `void push_back (const value_type& val);`  
`void pop_back();`

作用 插入/删除尾部元素，长度自动 +1, -1。

时间复杂度 均摊常数。

# push\_back, pop\_back con't

```

1 // vector::push_back
2 #include <iostream>
3 #include <vector>
4
5 int main ()
6 {
7     std::vector<int> myvector;
8     int myint;
9
10    std::cout << "Please enter some integers (enter 0 to end):\n";
11
12    do {
13        std::cin >> myint;
14        myvector.push_back (myint);
15    } while (myint);
16
17    std::cout << "myvector stores " << int(myvector.size()) << " numbers.\n";
18
19    return 0;
20 }

```

# push\_back, pop\_back con't

```

1 // vector::pop_back
2 #include <iostream>
3 #include <vector>
4
5 int main ()
6 {
7     std::vector<int> myvector;
8     int sum (0);
9     myvector.push_back (100);
10    myvector.push_back (200);
11    myvector.push_back (300);
12
13    while (!myvector.empty())
14    {
15        sum+=myvector.back();
16        myvector.pop_back();
17    }
18
19    std::cout << "The elements of myvector add up to " << sum << '\n';
20
21    return 0;
22 }
23
24 // The elements of myvector add up to 600

```

clear

原型 `void clear();`

作用 清空（重置大小为 0）

时间复杂度 关于长度线性。

## clear con't

```

1 // clearing vectors
2 #include <iostream>
3 #include <vector>
4 int main ()
5 {
6     std::vector<int> myvector;
7     myvector.push_back (100);
8     myvector.push_back (200);
9     myvector.push_back (300);
10
11     std::cout << "myvector contains: ";
12     for (unsigned i=0; i<myvector.size(); i++)
13         std::cout << ' ' << myvector[i];
14     std::cout << '\n';
15
16     myvector.clear();
17     myvector.push_back (1101);
18     myvector.push_back (2202);
19
20     std::cout << "myvector contains: ";
21     for (unsigned i=0; i<myvector.size(); i++)
22         std::cout << ' ' << myvector[i];
23     std::cout << '\n';
24
25     return 0;
26 }
27 // myvector contains: 100 200 300
28 // myvector contains: 1101 2202

```

## 应用

## 邻接表

```

1  #include <vector>
2  #include <iostream>
3  int main()
4  {
5      std::vector<std::vector<std::size_t>> graph(5);
6      graph[0].push_back(1); // V0->V1.
7      graph[1].push_back(4); // V1->V4.
8      graph[1].push_back(0); // V1->V0.
9      graph[2].push_back(1); // V2->V1.
10     graph[2].push_back(3); // V2->V3.
11     graph[3].push_back(0); // V3->V0.
12     graph[4].push_back(3); // V4->V3.
13     // 假定访问点 1.
14     for (std::vector<int>::iterator it = graph[1].begin(); it != graph[1].end(); ++it)
15         std::cout << '1' << *it;
16     std::cout << std::endl;
17     // 程序运行后输出 4 0, 表示点 1 连接了 4 点和 0 点。
18     return 0;
19 }

```

# FBI Warning

## 使用本节的函数/类可能需

```
1 #include <set>
2 using std::set;
3 using std::multiset;
```

# set

**原型** `template < class T, class Compare = less<T>,  
class Alloc = allocator<T> > class set;`

**实质** 平衡树。重复元素储存一次。 T 为元素类型。  
Compare 的意义同 priority\_queue 中的。

**周边** 成员类型 iterator 双向迭代器

成员函数 begin, end 头尾迭代器。

成员函数 insert, erase 插入删除

成员函数 clear, empty, size 清空/是否为空/大小。

成员函数 lower\_(upper\_)bound 二分查找元素。

## 关于双向迭代器

相对于随机迭代器，双向迭代器不能在常数时间内任意跳转，比较大小（可以判等），只能在常数时间内跳过一个元素。

以下语句是非法的（ $a, b$  为双向迭代器， $n$  为整型变量）

- $a + n, a += n, a - n, a -= n, b - a$
- $a < b, a > b, a \leq b, a \geq b$
- $a[n]$

○  
○○○○○

○○○○  
○○○○○○○○○○  
○○○○○○○○

○○○○○○○○○○○  
○○○○○○○○○  
○○○○○○○

○○○○○○○○○  
○○○○○○○○○○○○  
○○○○○○○○○○○○

○○○○○○○○○○○○○○  
○○●○○○○○○○○○○○○  
○○○○○○○○○○○○○○○○

○  
○

set, multiset

## set 构造函数

```

1 // constructing sets
2 #include <iostream>
3 #include <set>
4 bool fncomp (int lhs, int rhs) {return lhs<rhs;}
5 struct classcomp {
6     bool operator() (const int& lhs, const int& rhs) const
7     {return lhs<rhs;}
8 };
9 int main ()
10 {
11     std::set<int> first; // empty set of ints
12
13     int myints[] = {10,20,30,40,50};
14     std::set<int> second (myints,myints+5); // range
15
16     std::set<int> third (second); // a copy of second
17
18     std::set<int> fourth (second.begin(), second.end()); // iterator ctor.
19
20     std::set<int,classcomp> fifth; // class as Compare
21
22     bool(*fn_pt)(int,int) = fncomp;
23     std::set<int,bool(*)(int,int)> sixth (fn_pt); // function pointer as Compare
24
25     return 0;
26 }
```

begin, end

原型      iterator begin();  
const\_iterator begin() const;  
            iterator end();  
const\_iterator end() const;

作用 首尾迭代器

时间复杂度 常数

begin, end can't

```

1 // set::begin/end
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     int myints[] = {75,23,65,42,13};
8     std::set<int> myset (myints,myints+5);
9
10    std::cout << "myset contains: ";
11    for (std::set<int>::iterator it=myset.begin(); it!=myset.end(); ++it)
12        std::cout << ' ' << *it;
13
14    std::cout << '\n';
15
16    return 0;
17 }
18 // myset contains: 13 23 42 65 75

```

# insert

## 原型

```
pair<iterator, bool> insert (const value_type& val);
iterator insert (iterator position, const value_type& val);
template <class InputIterator>
void insert (InputIterator first, InputIterator last);
```

## 作用

插入元素。第一个函数表示直接插入 *val*。第二个函数表示已知将被插入的位置 *position*，插入 *val*。第三个函数表示将  $[first, last)$  的元素全部插入 *set*。第一个函数返回插入后元素的位置，以及是否插入成功的 *pair*。第二个函数返回插入后元素的位置。

## 时间复杂度

关于大小对数。若提供正确的 *position*，则均摊常数。

○  
○○○○○

○○○○  
○○○○○○○○○○  
○○○○○○○○

○○○○○○○○○○○○  
○○○○○○○○○○○○  
○○○○○○○

○○○○○○○○○  
○○○○○○○○○○○○  
○○○○○○○○○○○○

○○○○○○○○○○○○○○  
○○○○○○○○●○○○○○○○○  
○○○○○○○○○○○○○○○○○○

○  
○

set, multiset

## insert con't

```

1 // set::insert (C++98)
2 #include <iostream>
3 #include <set>
4 int main ()
5 {
6     std::set<int> myset;
7     std::set<int>::iterator it;
8     std::pair<std::set<int>::iterator,bool> ret;
9     // set some initial values:
10    for (int i=1; i<=5; ++i) myset.insert(i*10);    // set: 10 20 30 40 50
11    ret = myset.insert(20);    // no new element inserted
12    if (ret.second==false) it=ret.first;    // "it" now points to element 20
13    myset.insert (it,25);    // max efficiency inserting
14    myset.insert (it,24);    // max efficiency inserting
15    myset.insert (it,26);    // no max efficiency inserting
16    int myints[] = {5,10,15};    // 10 already in set, not inserted
17    myset.insert (myints,myints+3);
18    std::cout << "myset contains:";
19    for (it=myset.begin(); it!=myset.end(); ++it)
20        std::cout << ' ' << *it;
21    std::cout << '\n';
22    return 0;
23 }
24 // myset contains: 5 10 15 20 24 25 26 30 40 50

```

# erase

## 原型

```
void erase (iterator position);
size_type erase (const value_type& val);
void erase (iterator first, iterator last);
```

## 作用

删除元素。第一个函数表示删除 *position* 指向的键值。第二个函数表示直接删除 *val*。第三个函数表示将  $[first, last)$  的元素删除。

第二个函数返回被删除的元素数。

## 时间复杂度

对第一个函数，均摊常数。对第二个函数，关于大小对数。对第三个函数，复杂度为大小的对数加上  $last - first$ 。

○  
○○○○○

○○○○  
○○○○○○○○○○  
○○○○○○○○

○○○○○○○○○○○  
○○○○○○○○○○○  
○○○○○○○

○○○○○○○○○  
○○○○○○○○○○○○  
○○○○○○○○○○○○○

○○○○○○○○○○○○○○○  
○○○○○○○○○○●○○○○○○○  
○○○○○○○○○○○○○○○○○○○

○  
○

set, multiset

## erase con't

```

1 // erasing from set
2 #include <iostream>
3 #include <set>
4 int main ()
5 {
6     std::set<int> myset;
7     std::set<int>::iterator it;
8     // insert some values:
9     for (int i=1; i<10; i++) myset.insert(i*10); // 10 20 30 40 50 60 70 80 90
10    it = myset.begin();
11    // "it" points now to 20
12    myset.erase (it);
13    myset.erase (40);
14    it = myset.find (60);
15    myset.erase (it, myset.end());
16    std::cout << "myset contains:";
17    for (it=myset.begin(); it!=myset.end(); ++it)
18        std::cout << ' ' << *it;
19    std::cout << '\n';
20    return 0;
21 }
22 // myset contains: 10 30 50

```

# clear, empty, size

## 原型

```
void clear();
bool empty() const;
size_type size() const;
```

作用 清空/询问是否为空/询问大小。

时间复杂度 第一个关于大小线性。后两个常数。

# clear, empty, size con't

```

1 // set::clear
2 #include <iostream>
3 #include <set>
4 int main ()
5 {
6     std::set<int> myset;
7     myset.insert (100);
8     myset.insert (200);
9     myset.insert (300);
10    std::cout << "myset contains:";
11    for (std::set<int>::iterator it=myset.begin(); it!=myset.end(); ++it)
12        std::cout << ' ' << *it;
13    std::cout << '\n';
14    myset.clear();
15    myset.insert (1101);
16    myset.insert (2202);
17    std::cout << "myset contains:";
18    for (std::set<int>::iterator it=myset.begin(); it!=myset.end(); ++it)
19        std::cout << ' ' << *it;
20    std::cout << '\n';
21    return 0;
22 }
23 // myset contains: 100 200 300
24 // myset contains: 1101 2202

```

○  
○○○○○

○○○○  
○○○○○○○○○○  
○○○○○○○○

○○○○○○○○○○○○  
○○○○○○○○○○○○  
○○○○○○○○

○○○○○○○○○  
○○○○○○○○○○○○  
○○○○○○○○○○○○

○○○○○○○○○○○○○○  
○○○○○○○○○○○○●○○○○  
○○○○○○○○○○○○○○○○

○  
○

set, multiset

# clear, empty, size con't

```

1 // set::empty
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::set<int> myset;
8
9     myset.insert(20);
10    myset.insert(30);
11    myset.insert(10);
12
13    std::cout << "myset contains:";
14    while (!myset.empty())
15    {
16        std::cout << ' ' << *myset.begin();
17        myset.erase(myset.begin());
18    }
19    std::cout << '\n';
20
21    return 0;
22 }
23 // myset contains: 10 20 30

```

# clear, empty, size con't

```

1 // set::size
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::set<int> myints;
8     std::cout << "0.size:" << myints.size() << '\n';
9
10    for (int i=0; i<10; ++i) myints.insert(i);
11    std::cout << "1.size:" << myints.size() << '\n';
12
13    myints.insert (100);
14    std::cout << "2.size:" << myints.size() << '\n';
15
16    myints.erase(5);
17    std::cout << "3.size:" << myints.size() << '\n';
18
19    return 0;
20 }
21 // 0. size: 0
22 // 1. size: 10
23 // 2. size: 11
24 // 3. size: 10

```

## lower\_bound, upper\_bound

原型 `iterator lower_bound (const value_type& val) const;`  
`iterator upper_bound (const value_type& val) const;`

作用 返回第一个不低于/超过 *val* 的元素的迭代器。

时间复杂度 关于大小对数。

切记不要调用算法库中的 lower\_bound,  
upper\_bound, 否则时间复杂度会退化。

# lower\_bound, upper\_bound can't

```

1 // set::lower_bound/upper_bound
2 #include <iostream>
3 #include <set>
4
5 int main ()
6 {
7     std::set<int> myset;
8     std::set<int>::iterator itlow,itup;
9
10    for (int i=1; i<10; i++) myset.insert(i*10); // 10 20 30 40 50 60 70 80 90
11
12    itlow=myset.lower_bound (30);                //
13    itup=myset.upper_bound (60);                  //
14
15    myset.erase(itlow,itup);                      // 10 20 70 80 90
16
17    std::cout << "myset contains:";
18    for (std::set<int>::iterator it=myset.begin(); it!=myset.end(); ++it)
19        std::cout << ' ' << *it;
20    std::cout << '\n';
21
22    return 0;
23 }
24 // myset contains: 10 20 70 80 90

```

# multiset

multiset 除了能储存多个相同的元素外，其余特性几乎与 set 相同。

两者均由红黑树实现。

## 应用

- 一些比较简单的平衡树题。  
eg. HNOI2002 营业额统计
- 只加点的二维动态凸包。先找插入的三个不公线的点，求其三角形的重心，并以该重心为原点维护插入点的极角序。
- 斜率和插入点横坐标都不单调的斜率优化型动态规划。
- .....

# FBI Warning

## 使用本节的函数/类可能需

```
1 #include <map>
2 using std::map;
3 using std::multimap;
```

# map

原型 `template < class Key, class T, class Compare = less<Key>,  
class Alloc = allocator<pair<const Key,T> > > class map;`

实质 平衡树实现的映射。重复原象储存一次。Key 为原象类型，T 为象类型。Compare 的意义同 set 中的。

## map con't

周边 成员类型 `iterator` 双向迭代器，指向类型

`pair<const Key, T>`

成员函数 `begin, end` 头尾迭代器。

成员运算 `[]` 访问元素。

成员函数 `insert, erase` 插入删除

成员函数 `clear, empty, size` 清空/是否为空/大小。

成员函数 `lower_(upper_)bound` 二分查找元素。

# map 构造函数

```

1 // constructing maps
2 #include <iostream>
3 #include <map>
4 bool fncomp (char lhs, char rhs) {return lhs<rhs;}
5 struct classcomp {
6     bool operator() (const char& lhs, const char& rhs) const
7     {return lhs<rhs;}
8 };
9 int main ()
10 {
11     std::map<char,int> first;
12     first['a']=10;
13     first['b']=30;
14     first['c']=50;
15     first['d']=70;
16     std::map<char,int> second (first.begin(),first.end());
17     std::map<char,int> third (second);
18     std::map<char,int,classcomp> fourth; // class as Compare
19     bool(*fn_pt)(char,char) = fncomp;
20     std::map<char,int,bool(*) (char,char)> fifth (fn_pt); // function pointer as Compare
21     return 0;
22 }
```

begin, end

原型

```
iterator begin();  
const_iterator begin() const;  
iterator end();  
const_iterator end() const;
```

作用 首尾迭代器

时间复杂度 常数

## begin, end con't

```

1 // map::begin/end
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::map<char,int> mymap;
8     std::map<char,int>::iterator it;
9
10    mymap['b'] = 100;
11    mymap['a'] = 200;
12    mymap['c'] = 300;
13
14    // show content:
15    for (std::map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)
16        std::cout << it->first << "=>" << it->second << '\n';
17
18    return 0;
19 }
20 // a => 200
21 // b => 100
22 // c => 300

```

[]

原型 `mapped_type& operator[] (const key_type& k);`

作用 访问原象  $k$  的象的引用。若  $k$  在容器中不存在，则先将该原象插入容器，象使用默认构造函数（int 为 0），再返回原象  $k$  的象的引用。

时间复杂度 对数大小

## [] con't

```

1 // accessing mapped values
2 #include <iostream>
3 #include <map>
4 #include <string>
5 int main ()
6 {
7     std::map<char, std::string> mymap;
8     mymap['a']="an element";
9     mymap['b']="another element";
10    mymap['c']=mymap['b'];
11    std::cout << "mymap['a'] is " << mymap['a'] << '\n';
12    std::cout << "mymap['b'] is " << mymap['b'] << '\n';
13    std::cout << "mymap['c'] is " << mymap['c'] << '\n';
14    std::cout << "mymap['d'] is " << mymap['d'] << '\n';
15    std::cout << "mymap now contains " << mymap.size() << " elements.\n";
16    return 0;
17 }
18 /*
19 mymap['a'] is an element
20 mymap['b'] is another element
21 mymap['c'] is another element
22 mymap['d'] is
23 mymap now contains 4 elements.
24 */

```

# insert

## 原型

```
pair<iterator,bool> insert (const value_type& val);
iterator insert (iterator position, const value_type& val);
template <class InputIterator>
void insert (InputIterator first, InputIterator last);
```

## 作用

插入元素。三种函数与 set 中的对应。注意 value\_type 是 pair<const Key, T>, 即插入的应为原象和象的 pair。

## 时间复杂度

关于大小对数。若提供正确的 *position*, 则均摊常数。

# insert con't

```

1 // map::insert (C++98)
2 #include <iostream>
3 #include <map>
4 int main ()
5 {
6     std::map<char,int> mymap;
7     // first insert function version (single parameter):
8     mymap.insert ( std::pair<char,int>('a',100) );
9     mymap.insert ( std::pair<char,int>('z',200) );
10    std::pair<std::map<char,int>::iterator,bool> ret;
11    ret = mymap.insert ( std::pair<char,int>('z',500) );
12    if (ret.second==false) {
13        std::cout << "element 'z' already existed";
14        std::cout << " with a value of " << ret.first->second << '\n';
15    }
16    // second insert function version (with hint position):
17    std::map<char,int>::iterator it = mymap.begin();
18    mymap.insert (it, std::pair<char,int>('b',300)); // max efficiency inserting
19    mymap.insert (it, std::pair<char,int>('c',400)); // no max efficiency inserting
20    // third insert function version (range insertion):
21    std::map<char,int> anothermap;
22    anothermap.insert(mymap.begin(),mymap.find('c'));

```

## insert con't

```

23
24 // showing contents:
25 std::cout << "mymap contains:\n";
26 for (it=mymap.begin(); it!=mymap.end(); ++it)
27     std::cout << it->first << "=>" << it->second << '\n';
28 std::cout << "anothermap contains:\n";
29 for (it=anothermap.begin(); it!=anothermap.end(); ++it)
30     std::cout << it->first << "=>" << it->second << '\n';
31 return 0;
32 }
33 // element 'z' already existed with a value of 200
34 // mymap contains:
35 // a => 100
36 // b => 300
37 // c => 400
38 // z => 200
39 // anothermap contains:
40 // a => 100
41 // b => 300

```

## erase

## 原型

```
void erase (iterator position);
size_type erase (const value_type& val);
void erase (iterator first, iterator last);
```

**作用** 删除元素。三种函数与 set 中的对应。注意 value\_type 是 Key，即删除的应为原象。

**时间复杂度** 对第一个函数，均摊常数。对第二个函数，关于大小对数。对第三个函数，复杂度为大小的对数加上  $last - first$ 。

○  
○○○○○

○○○○○  
○○○○○○○○○○  
○○○○○○○○○

○○○○○○○○○○○  
○○○○○○○  
○○○○○○○

○○○○○○○○○  
○○○○○○○○○○○○  
○○○○○○○○○○○○○

○○○○○○○○○○○○○○○  
○○○○○○○○○○○○○○○  
○○○○○○○○○○○○●○○○○○○○

○  
○

map, multimap

## erase con't

```

1 // erasing from map
2 #include <iostream>
3 #include <map>
4 int main ()
5 {
6     std::map<char,int> mymap;
7     std::map<char,int>::iterator it;
8     // insert some values:
9     mymap['a']=10;
10    mymap['b']=20;
11    mymap['c']=30;
12    mymap['d']=40;
13    mymap['e']=50;
14    mymap['f']=60;
15    it=mymap.find('b');
16    mymap.erase (it);           // erasing by iterator
17    mymap.erase ('c');         // erasing by key
18    it=mymap.find ('e');
19    mymap.erase ( it, mymap.end() ); // erasing by range
20    // show content:
21    for (it=mymap.begin(); it!=mymap.end(); ++it)
22        std::cout << it->first << "□=>□" << it->second << '\n';
23    return 0;
24 }
25 // a => 10
26 // d => 40

```

## clear, empty, size

原型

```
void clear();  
bool empty() const;  
size_type size() const;
```

作用 清空/询问是否为空/询问大小。

时间复杂度 第一个关于大小线性。后两个常数。

# clear, empty, size con't

```

1 // map::clear
2 #include <iostream>
3 #include <map>
4 int main ()
5 {
6     std::map<char,int> mymap;
7     mymap['x']=100;
8     mymap['y']=200;
9     mymap['z']=300;
10    std::cout << "mymap contains:\n";
11    for (std::map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)
12        std::cout << it->first << "=>" << it->second << '\n';
13    mymap.clear();
14    mymap['a']=1101;
15    mymap['b']=2202;
16    std::cout << "mymap contains:\n";
17    for (std::map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)
18        std::cout << it->first << "=>" << it->second << '\n';
19    return 0;
20 }
21 // mymap contains:
22 // x => 100
23 // y => 200
24 // z => 300
25 // mymap contains:
26 // a => 1101
27 // b => 2202

```

# clear, empty, size con't

```

1 // map::empty
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::map<char,int> mymap;
8
9     mymap['a']=10;
10    mymap['b']=20;
11    mymap['c']=30;
12
13    while (!mymap.empty())
14    {
15        std::cout << mymap.begin()->first << "=>" << mymap.begin()->second << '\n';
16        mymap.erase(mymap.begin());
17    }
18
19    return 0;
20 }
21 // a => 10
22 // b => 20
23 // c => 30

```

# clear, empty, size con't

```

1 // map::size
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::map<char,int> mymap;
8     mymap['a']=101;
9     mymap['b']=202;
10    mymap['c']=302;
11
12    std::cout << "mymap.size() is " << mymap.size() << '\n';
13
14    return 0;
15 }
16 // mymap.size() is 3

```

# lower\_bound, upper\_bound

原型 `iterator lower_bound (const value_type& val) const;`  
`iterator upper_bound (const value_type& val) const;`

作用 返回第一个原象不低于/超过 *val* 的元素的迭代器。

时间复杂度 关于大小对数。

切记不要调用算法库中的 lower\_bound,  
upper\_bound, 否则时间复杂度会退化。

## lower\_bound, upper\_bound can't

```

1 // map::lower_bound/upper_bound
2 #include <iostream>
3 #include <map>
4
5 int main ()
6 {
7     std::map<char,int> mymap;
8     std::map<char,int>::iterator itlow,itup;
9     mymap['a']=20;
10    mymap['b']=40;
11    mymap['c']=60;
12    mymap['d']=80;
13    mymap['e']=100;
14    itlow=mymap.lower_bound ('b'); // itlow points to b
15    itup=mymap.upper_bound ('d'); // itup points to e (not d!)
16    mymap.erase(itlow,itup);      // erases [itlow,itup)
17    // print content:
18    for (std::map<char,int>::iterator it=mymap.begin(); it!=mymap.end(); ++it)
19        std::cout << it->first << "=>" << it->second << '\n';
20    return 0;
21 }
22 // a => 20
23 // e => 100

```

# multimap

multimap 除了能储存多个相同的原象外，其余特性几乎与 map 相同。  
两者均由红黑树实现。

## 应用

- 一些图论题用字符串为节点编号。可用之将字符串化为整数
- 诸如连通性状态压缩的动态规划涉及到的状态非常复杂，不便于压缩。可用之将状态映射为值，代替哈希（注意超时的风险，hash\_map 推奖）
- 离散化（注意超时的风险，sort 再 unique 推奖）
- .....

## 更高级的容器

还有一些罕见的容器，如 `hash_set`，`hash_map`（带哈希的平衡树）、`rope`（可持久化的平衡树）等。

感兴趣的请自行探索。

# 私について

QQ 695582089

Mail shyoshyo [at] qq [dot] com

Codeforces bakabakashyoshyo

topcoder lkx

Ask.fm ask.fm/shyoshyoQwQ