

acm c++常用函数 (stl) - CSDN博客

```
#include  
fill(dp,dp+1010,1);/*概率通通赋值为1*/
```

| 全排列函数next_permutation

STL 中专门用于排列的函数（可以处理存在重复数据集的排列问题）

头文件: #include

using namespace std;

调用: next_permutation(start, end);

注意: 函数要求输入的是一个升序排列的序列的头指针和尾指针。

用法:

// 数组

```
int a[N];
```

```
sort(a, a+N);
```

```
next_permutation(a, a+N);
```

// 向量

```
vector ivec;
```

```
sort(ivec.begin(), ivec.end());
```

```
next_permutation(ivec.begin(), ivec.end());
```

例子:

```
vector myVec;
```

// 初始化代码

```
sort(myVec.begin(),myVec.end());
```

```
do{
```

```
for (i = 0 ;i < size;i ++ ) cout <<< " /t " ;
```

```
cout << endl;
```

```
}while (next_permutation(myVec.begin(), myVec.end()));
```

ACM/ICPC 竞赛之STL简介

一、关于STL

STL(Standard Template Library, 标准模板库)是C++语言标准中的重

要组成部分。STL 以模板类和模板函数的形式为程序员提供了各种数据结构和

算法的精巧实现, 程序员如果能够充分地利用STL, 可以在代码空间、执行时

间和编码效率上获得极大的好处。

STL 大致可以分为三大类: 算法(algorithm)、容器(container)、迭代器

(iterator)。

STL 容器是一些模板类, 提供了多种组织数据的常用方法, 例如vector(向量, 类似于数组)、list(列表, 类似于链表)、deque(双向队列)、set(集合)、map(映象)、stack(栈)、queue(队列)、priority_queue(优先队列)等, 通过模板的参数我们可以指定容器中的元素类型。

STL 算法是一些模板函数, 提供了相当多的有用算法和操作, 从简单如for_each(遍历)到复杂如stable_sort(稳定排序)。

STL 迭代器是对C 中的指针的一般化, 用来将算法和容器联系起来。几乎所有的STL算法都是通过迭代器来存取元素序列进行工作的, 而STL中的每一个容器也都定义了其本身所专有的迭代器, 用以存取容器中的元素。有趣的是, 普通的指针也可以像迭代器一样工作。

熟悉了STL 后, 你会发现, 很多功能只需要用短短的几行就可以实现了。通过STL, 我们可以构造出优雅而且高效的代码, 甚至比你自己手工实现的代码效果还要好。

STL 的另外一个特点是, 它是以源码方式免费提供的, 程序员不仅可以自由地使用这些代码, 也可以学习其源码, 甚至按照自己的需要去修改它。

下面是用STL 写的题Ugly Numbers 的代码:

```
#include
```

```
#include
```

```
using namespace std;
```

```

typedef pair<int,int> pair_type;
int main(){
    unsigned long result[1500];
    priority_queue< pair_type,vector,priority_queue::greater > Q;
    Q.push( make_pair(1, 2) );
    for (int i=0; i<1500; i++){
        pair_type node = Q.top(); Q.pop();
        switch(node.second){
            case 2: Q.push( make_pair(node.first*2, 2) );
            case 3: Q.push( make_pair(node.first*3, 3) );
            case 5: Q.push( make_pair(node.first*5, 5) );
        }
        result[i] = node.first;
    }
    int n;
    cin >> n;
    while (n>0){
        cout << result[n-1] << endl;
        cin >> n;
    }
    return 0;
}

```

在ACM 竞赛中，熟练掌握和运用STL 对快速编写实现代码会有极大的帮助。

二、使用STL

在C++标准中，STL 被组织为以下的一组头文件（注意，是没有.h后缀的！）：

algorithm / deque / functional / iterator /list / map
memory / numeric / queue / set / stack /utility / vector

当我们需要使用STL 的某个功能时，需要嵌入相应的头文件。但要注意的是，

在C++标准中，STL 是被定义在std 命名空间中的。如下例所示：

```

#include
int main(){
    std::stack s;
    s.push(0);
    ...
    return 0;
}

```

如果希望在程序中直接引用STL，也可以在嵌入头文件后，用using

namespace 语句将std 命名空间导入。如下例所示：

```

#include
using namespace std;
int main(){
    stack s;
    s.push(0);
    ...
    return 1;
}

```

STL 是C++语言机制运用的一个典范，通过学习STL可以更深刻地理解C++

语言思想和方法。在本系列的文章中不打算对STL 做深入的剖析，而只是想介绍一些STL的基本应用。

有兴趣的同学，建议可以在有了一些STL 的使用经验后，认真阅读一下《C++STL》这本书（电力出版社有该书的中文版）。

ACM/ICPC 竞赛之STL--pair

STL 的头文件中描述了一个看上去非常简单的模板类pair，用来表示一个二元组或元素对，并提供了按照字典序对元素对进行大小比较的比较运算符模板函数。

例如，想要定义一个对象表示一个平面坐标点，则可以：__

```
pair p1;
```

```
cin >> p1.first >> p1.second;
```

pair 模板类需要两个参数：首元素的数据类型和尾元素的数据类型。**pair** 模板类对象有两个成员：**first** 和**second**，分别表示首元素和尾元素。

在中已经定义了**pair** 上的六个比较运算符：<、>、<=、>=、==、!=，

其规则是先比较**first**，**first** 相等时再比较**second**，这符合大多数应用的逻辑。当然，也可以通过重载这几个运算符来重新指定自己的比较逻辑。

除了直接定义一个**pair** 对象外，如果需要即时生成一个**pair** 对象，也可以调用在中定义的一个模板函数：**make_pair**。**make_pair**需要两个参数，分别为元素对的首元素和尾元素。

在题1067--Ugly Numbers 中，就可以用**pair** 来表示推演树上的结点，用**first** 表示结点的值，用**second** 表示结点是由父结点乘以哪一个因子得到的。

```
#include
```

```
#include
```

```
using namespace std;
```

```
typedef pair<node_type, vector<int>> Q;
```

```
int main(){
```

```
    unsigned long result[1500];
```

```
    priority_queue<Q, vector<Q>, greater<Q>> Q;
```

```
    Q.push( make_pair(1, 2) );
```

```
    for (int i=0; i<1500; i++){
```

```
        node_type node = Q.top(); Q.pop();
```

```
        switch(node.second){
```

```
            case 2: Q.push( make_pair(node.first*2, 2) );
```

```
            case 3: Q.push( make_pair(node.first*3, 3) );
```

```
            case 5: Q.push( make_pair(node.first*5, 5) );
```

```
        }
```

```
        result[i] = node.first;
```

```
    }
```

```
    int n;
```

```
    cin >> n;
```

```
    while (n>0){
```

```
        cout << result[n-1] << endl;
```

```
        cin >> n;
```

```
    }
```

```
    return 0;
```

```
}
```

看上去是很简单的一个头文件，但是的设计中却浓缩

反映了STL 设计的基本思想。有意深入了解和研究STL 的同学，仔细阅读和体会这个简单的头文件，不失为一种入门的途径。

ACM/ICPC 竞赛之STL--vector

在STL 的头文件中定义了**vector**（向量容器模板类），**vector**

容器以连续数组的方式存储元素序列，可以将**vector** 看作是以顺序结构实现的线性表。当我们在程序中需要使用动态数组时，**vector**将会是理想的选择，**vector**可以在使用过程中动态地增长存储空间。

vector 模板类需要两个模板参数，第一个参数是存储元素的数据类型，第二个参数是存储分配器的类型，其中第二个参数是可选的，如果不给出第二个参数，将使用默认的分配器。

下面给出几个常用的定义**vector** 向量对象的方法示例：

vector s;定义一个空的**vector** 对象，存储的是**int**类型的元素。

vector s(n);定义一个含有**n** 个**int**元素的**vector**对象。

vector s(first, last);定义一个**vector** 对象，并从由迭代器**first** 和**last** 定义的序列[**first,last**)中复制初值。

vector 的基本操作有：

s[i]直接以下标方式访问容器中的元素。

s.front() 返回首元素。

s.back() 返回尾元素。

s.push_back(x) 向表尾插入元素**x**。

s.size() 返回表长。

s.empty() 当表空时，返回真，否则返回假。

s.pop_back() 删除表尾元素。

s.begin() 返回指向首元素的随机存取迭代器。

s.end() 返回指向尾元素的下一个位置的随机存取迭代器。

s.insert(it, x) 向迭代器**it** 指向的元素前插入新元素**val**。

s.insert(it, n, x) 向迭代器**it** 指向的元素前插入**n**个**x**。

s.insert(it, first, last) 将由迭代器**first** 和**last** 所指定的序列[**first, last**)插入到迭代器**it**指向的元素前面。

s.erase(it)删除由迭代器**it** 所指向的元素。

s.erase(first, last) 删除由迭代器**first** 和**last** 所_____指定的序列[**first, last**)。

s.reserve(n)

预分配缓冲空间，使存储空间至少可容纳**n** 个元素。

s.resize(n)

改变序列的长度，超出的元素将会被删除，如果序列需要扩展（原空间小于**n**），元素默认值将填满扩展出的空间。

s.resize(n, val)

改变序列的长度，超出的元素将会被删除，如果序列需要扩展（原空间小于**n**），将用**val** 填满扩展出的空间。

s.clear()

删除容器中的所有的元素。

s.swap(v)

将**s** 与另一个**vector** 对象**v** 进行交换。

s.assign(first, last)

将序列替换成由迭代器**first** 和**last** 所指定的序列[**first, last**)。

[**first, last**)不能是原序列中的一部分。

要注意的是，**resize** 操作和**clear** 操作都是对表的有效元素进行的操作，但并不一定会改变缓冲空间的大小。

另外，**vector** 还有其他一些操作如反转、取反等，不再一下列举。

vector 上还定义了序列之间的比较操作运算符(>, <, >=, <=, ==, !=),

可以按照字典序比较两个序列。还是来看一些示例代码。输入个数不定的一组整数，再将这组整数按倒序输出，如下所示：

```
#include
#include
using namespace std;
int main(){
    vector L;
    int x;
    while (cin>>x) L.push_back(x);
    for (int i=L.size()-1; i>=0; i--) cout<< L[i] << " ";
    cout << endl;
    return 0;
}
```

ACM/ICPC 竞赛之STL--iterator简介

iterator(迭代器)是用于访问容器中元素的指示器，从这个意义上说，

iterator(迭代器)相当于数据结构中所说的“遍历指针”，也可以把

iterator(迭代器)看作是一种泛化的指针。

STL 中关于**iterator**(迭代器)的实现是相当复杂的，这里我们暂时不去详细讨论关于**iterator**(迭代器)的实现和使用，而只对**iterator**(迭代器)做一点简单的介绍。简单地说，STL中有以下几类**iterator**(迭代器)：

输入**iterator**(迭代器)，在容器的连续区间内向前移动，可以读取容器内任意值；

输出**iterator**(迭代器)，把值写进它所指向的容器中；

前向**iterator**(迭代器)，读取队列中的值，并可以向前移动到下一位置

(++p,p++)；

双向**iterator**(迭代器)，读取队列中的值，并可以向前向后遍历容器；
随机访问**iterator**(迭代器)，可以直接以下标方式对容器进行访问，
vector 的**iterator**(迭代器)就是这种**iterator**(迭代器)；
流**iterator**(迭代器)，可以直接输出、输入流中的值；
每种STL 容器都有自己的**iterator**(迭代器)子类，下面先来看一段简单的示例代码：

```
#include
#include
using namespace std;
main()
{
    vector s;
    for (int i=0; i<10; i++) s.push_back(i);
    for (vector::iterator it=s.begin(); it!=s.end();
        it++)
        cout << *it << " ";
    cout << endl;
    return 1;
}
```

vector 的**begin()**和**end()**方法都会返回一个**vector::iterator**对象，
分别指向**vector** 的首元素位置和尾元素的下一个位置（我们可以称之为结束标志位置）。
对一个**iterator**(迭代器)对象的使用与一个指针变量的使用极为相似，或者可以这样说，指针就是一个非常标准的**iterator**(迭代器)。
再来看一段稍微特别一点的代码：

```
#include
#include
main()
{
    vector s;
    s.push_back(1);
    s.push_back(2);
    s.push_back(3);
    copy(s.begin(), s.end(), ostream_iterator(cout, " "));
    cout <
    return 1;
}
```

这段代码中的**copy** 就是STL 中定义的一个模板函数，**copy(s.begin(), s.end(), ostream_iterator(cout, " "));**的意思是将由**s.begin()**至**s.end()**(不含**s.end()**)所指定的序列复制到标准输出流**cout**中，用" "作为每个元素的间隔。也就是说，这句话的作用其实就是将表中的所有内容依次输出。**iterator**(迭代器)是STL容器和算法之间的“胶合剂”，几乎所有的STL算法都是通过容器的**iterator**(迭代器)来访问容器内容的。只有通过有效地运用**iterator**(迭代器)，才能够有效地运用STL强大的算法功能。

ACM/ICPC 竞赛之STL--string

字符串是程序中经常要表达和处理的数据，我们通常是采用字符数组或字符指针来表示字符串。**STL** 为我们提供了另一种使用起来更为便捷的字符串的表达

方式：**string**。**string** 类的定义在头文件中。

string 类其实可以看作是一个字符的**vector**，**vector**上的各种操作都可以适用于**string**，另外，**string**类对象还支持字符串的拼合、转换等操作。

下面先来看一个简单的例子：

```
#include
#include
using namespace std;
int main(){
    string s = "Hello! ", name;
```

```

cin >> name;
s += name;
s += '!';
cout << s << endl;
return 0;
}

```

再以题**1064--Parenencoding**为例，看一段用**string**作为容器，实现由**P**代码还原括号字符串的示例代码片段：

```

int m;
cin >> m; // P 编码的长度
string str; // 用来存放还原出来的括号字符串
int leftpa = 0; // 记录已出现的左括号的总数
for (int j=0; j
int p;
cin >> p;
for (int k=0; k
str += ')';
leftpa = p;
}

```

ACM/ICPC 竞赛之STL--stack/queue

stack(栈)和**queue**(队列)也是在程序设计中经常会用到的数据容器，**STL**

为我们提供了方便的**stack**(栈)的**queue**(队列)的实现。

准确地说，**STL**中的**stack**和**queue**不同于**vector**、**list**等容器，而是对这些容器的重新包装。这里我们不去深入讨论**STL**的**stack**和**queue**的实现细节，而是来了解一些他们的基本使用。

1、stack

stack模板类的定义在头文件中。

stack模板类需要两个模板参数，一个是元素类型，一个容器类型，但只有元素类型是必要的，在不指定容器类型时，默认的容器类型为**deque**。

定义**stack**对象的示例代码如下：

```

stack s1;
stack s2;

```

stack的基本操作有：

入栈，如例：**s.push(x)**;

出栈，如例：**s.pop()**;注意，出栈操作只是删除栈顶元素，并不返回该元素。

访问栈顶，如例：**s.top()**

判断栈空，如例：**s.empty()**，当栈空时，返回**true**。

访问栈中的元素个数，如例：**s.size()**

下面是用**string**和**stack**写的解题**1064--Parenencoding**的程序。

```

#include
#include
#include
using namespace std;
int main(){
int n;
cin >> n;
for (int i=0; i
int m;
cin >> m;
string str;
int leftpa = 0;
for (int j=0; j读入P 编码，构造括号字符串
{

```

```

int p;
cin >> p;
for (int k=0; k
str += ');
leftpa = p;
}

stack s;
for (string::iterator it=str.begin(); it!=str.end(); it++) { //构造M编码
if (*it=='(') s.push(1);
else{
int p = s.top(); s.pop();
cout << p << " ";
if (!s.empty()) s.top() += p;
}
}
cout << endl;
}
return 0;
}

```

2、queue

queue 模板类的定义在头文件中。

与**stack** 模板类很相似，**queue** 模板类也需要两个模板参数，一个是元素类型，一个容器类型，元素类型是必要的，容器类型是可选的，默认为**deque**类型。

定义**queue** 对象的示例代码如下：

```
queue q1;
```

```
queue q2;
```

queue 的基本操作有：

入队，如例：**q.push(x)**; 将**x** 接到队列的末端。

出队，如例：**q.pop()**; 弹出队列的第一个元素，注意，并不会返回被弹出元素的值。

访问队首元素，如例：**q.front()**，即最早被压入队列的元素。

访问队尾元素，如例：**q.back()**，即最后被压入队列的元素。

判断队列空，如例：**q.empty()**，当队列空时，返回**true**。

访问队列中的元素个数，如例：**q.size()**

3、priority_queue

在头文件中，还定义了另一个非常有用的模板类

priority_queue(优先队列)。优先队列与队列的差别在于优先队列不是按照入队的顺序出队，而是按照队列中元素的优先权顺序出队（默认为大者优先，也可以通过指定算子来指定自己的优先顺序）。

priority_queue 模板类有三个模板参数，第一个是元素类型，第二个容器类型，第三个是比较算子。其中后两个都可以省略，默认容器为**vector**，默认算子为**less**，即小的往前排，大的往后排（**出队时序列尾的元素出队**）。

定义**priority_queue** 对象的示例代码如下：

```
priority_queue q1;
```

```
priority_queue<pair>q2; // 注意在两个尖括号之间一定要留空格。
```

```
priority_queue,greater> q3; // 定义小的先出队
```

priority_queue 的基本操作与**queue** 相同。

初学者在使用**priority_queue** 时，最困难的可能就是如何定义比较算子了。

如果是基本数据类型，或已定义了比较运算符的类，可以直接用**STL** 的**less**算子和**greater**算子——默认为使用**less**算子，即小的往前排，大的先出队。如果要定义自己的比较算子，方法有多种，这里介绍其中的一种：重载比较运算符。优先队列试图将两个元素**x**和**y**代入比较运算符(对**less** 算子，调用**x**，对**greater**算子，调用**x>y**)，若结果为真，则**x**排在**y** 前面，**y** 将先于**x** 出队，反之，则将**y**排在**x**前面，**x** 将先出队。

看下面这个简单的示例：

```
#include
```

```

#include
using namespace std;
class T{
public:
int x, y, z;
T(int a, int b, int c):x(a), y(b), z(c){}
};
bool operator < (const T &t1, const T&t2){
return t1.z < t2.z; // 按照z 的顺序来决定
}
int main(){
priority_queue q;
q.push(T(4,4,3));
q.push(T(2,2,5));
q.push(T(1,5,4));
q.push(T(3,3,6));
while (!q.empty()){
T t = q.top(); q.pop();
cout << t.x << " "<< t.y << " " << t.z << endl;
}
return 0; }

```

输出结果为(注意是按照z 的顺序从大到小出队的):

3 3 6

2 2 5

1 5 4

4 4 3

再看一个按照z 的顺序从小到大出队的例子:

```

#include
#include
using namespace std;
class T{
public:
int x, y, z;
T(int a, int b, int c):x(a), y(b), z(c){}
};
bool operator > (const T &t1, const T&t2){
return t1.z > t2.z;
}
int main(){
priority_queue,greater > q;
q.push(T(4,4,3));
q.push(T(2,2,5));
q.push(T(1,5,4));
q.push(T(3,3,6));
while (!q.empty()){
T t = q.top(); q.pop();
cout << t.x << " "<< t.y << " " << t.z << endl;
}
return 0;
}

```

输出结果为:

4 4 3

1 5 4

2 2 5

3 3 6

如果我们把第一个例子中的比较运算符重载为：

```
bool operator < (const T &t1, const T&t2){
return t1.z > t2.z; // 按照z 的顺序来决定t1和t2的顺序
}
```

则第一个例子的程序会得到和第二个例子的程序相同的输出结果。

再回顾一下用优先队列实现的题1067--Ugly Numbers 的代码：

```
#include
#include
using namespace std;
typedef pair<int,int> node_type;
int main( int argc, char *argv[] ){
unsigned long int result[1500];
priority_queue< node_type,vector,node_type,greater > Q;
Q.push( make_pair(1, 3) );
for (int i=0; i<1500; i++){
node_type node = Q.top();
Q.pop();
switch(node.second){
case 3: Q.push( make_pair(node.first*2, 3) );
case 2: Q.push( make_pair(node.first*3, 2) );
case 1: Q.push( make_pair(node.first*5, 1) );
}
result[i] = node.first;
}
int n;
cin >> n;
while (n>0){
cout << result[n-1] << endl;
cin >> n;
}
return 1;
}
```

ACM/ICPC 竞赛之STL--map

在STL 的头文件中定义了模板类**map**和**multimap**，用有序二叉树来存贮类型为**pair**的元素对序列。序列中的元素以**const Key**部分作为标识，**map**中所有元素的**Key**值都必须是唯一的，**multimap** 则允许有重复的**Key**值。

可以将**map** 看作是由**Key** 标识元素的元素集合，这类容器也被称为关联容器”，可以通过一个**Key**值来快速确定一个元素，因此非常适合于需要按照**Key**值查找元素的容器。**map**模板类需要四个模板参数，第一个是键值类型，第二个是元素类型，第三个是比较算子，第四个是分配器类型。其中键值类型和元素类型是必要的。

map 的基本操作有：

1、定义**map** 对象，例如：

```
map m;
```

2、向**map** 中插入元素对，有多种方法，例如：

```
m[key] = value;
```

[key]操作是**map** 很有特色的操作，如果在**map**中存在键值为key的元素对，则返回该元素对的值域部分，否则将会创建一个键值为key 的元素对，值域为默认值。所以可以用该操作向**map**中插入元素对或修改已经存在的元素对的值域部分。

```
m.insert( make_pair(key, value) );
```

也可以直接调用**insert** 方法插入元素对，**insert** 操作会返回一个**pair**，当**map**中没有与key相匹配的键值时，其**first** 是指向插入元素对的迭代器，其**second**为true；若**map**中已经存在与key 相等的键值时，其**first** 是指向该元素对的迭代器，**second**为false。

3、查找元素对，例如：

```
int i = m[key];
```

要注意的是，当与该键值相匹配的元素对不存在时，会创建键值为`key`的元素对。

```
map::iterator it =m.find(key);
```

如果`map` 中存在与`key` 相匹配的键值时，`find`操作将返回指向该元素对的迭代器，否则，返回的迭代器等于`map` 的`end()`（参见`vector`中提到的`begin`和`end` 操作）。

4、删除元素对，例如：

```
m.erase(key);
```

删除与指定`key` 键值相匹配的元素对，并返回被删除的元素的个数。

```
m.erase(it);
```

删除由迭代器`it` 所指定的元素对，并返回指向下一个元素对的迭代器。

看一段简单的示例代码：

```
#include
#include

using namespace std;

typedef map< M_TYPE> M_TYPE;
typedef M_TYPE::iterator M_IT;
typedef M_TYPE::const_iterator M_CIT;

int main(){
    M_TYPE MyTestMap;
    MyTestMap[3] = "No.3";
    MyTestMap[5] = "No.5";
    MyTestMap[1] = "No.1";
    MyTestMap[2] = "No.2";
    MyTestMap[4] = "No.4";
    M_IT it_stop = MyTestMap.find(2);
    cout << "MyTestMap[2] = "<< it_stop->second << endl;
    it_stop->second = "No.2 Aftermodification";
    cout << "MyTestMap[2] = "<< it_stop->second << endl;
    cout << "Map contents : "<< endl;
    for(M_CIT it = MyTestMap.begin(); it !=MyTestMap.end();
    it++){
        cout << it->second << endl;
    }
    return 0;
}
```

程序执行的输出结果为：

```
MyTestMap[2] = No.2
MyTestMap[2] = No.2 After modification
Map contents :
No.1
No.2 After modification
No.3
No.4
No.5
```

再看一段简单的示例代码：

```
#include
#include

using namespace std;

int main(){
    map m;
```

```

m["one"] = 1;
m["two"] = 2;
// 几种不同的insert 调用方法
m.insert(make_pair("three", 3));
m.insert(map::value_type("four", 4));
m.insert(pair("five", 5));
string key;
while (cin>>key){
map::iterator it =m.find(key);
if (it==m.end()){
cout << "No such key!"<< endl;
}
else{
cout << key << " is " << it->second << endl;
cout << "Erased " <<< endl;
}
}
return 0;
}

```

ACM/ICPC 竞赛之STL--algorithm

无疑是STL 中最大的一个头文件，它是由一大堆模板函数组成的。

下面列举出中的模板函数：

```

adjacent_find / binary_search / copy /copy_backward / count
/ count_if / equal / equal_range / fill / fill_n/ find /
find_end / find_first_of / find_if / for_each/ generate /
generate_n / includes / inplace_merge /iter_swap /
lexicographical_compare / lower_bound /make_heap / max /
max_element / merge / min / min_element /mismatch /
next_permutation / nth_element / partial_sort/
partial_sort_copy / partition / pop_heap /prev_permutation
/ push_heap / random_shuffle / remove /remove_copy /
remove_copy_if / remove_if / replace /replace_copy /
replace_copy_if / replace_if / reverse /reverse_copy /
rotate / rotate_copy / search / search_n /set_difference /
set_intersection / set_symmetric_difference /set_union /
sort / sort_heap / stable_partition /stable_sort / swap /
swap_ranges / transform / unique /unique_copy / upper_bound

```

如果详细叙述每一个模板函数的使用，足够写一本书的了。还是来看几个简单的示例程序吧。

示例程序之一，for_each 遍历容器：

```

#include
#include
#include
using namespace std;
int Visit(int v) // 遍历算子函数
{
cout << v << " ";
return 1;
}
class MultInt // 定义遍历算子类
{
private:

```

```

int factor;
public:
MultInt(int f) : factor(f){}
void operator()(int &elem) const{
elem *= factor;
}
};

int main(){
vector L;
for (int i=0; i<10; i++) L.push_back(i);
for_each(L.begin(), L.end(), Visit);
cout << endl;
for_each(L.begin(), L.end(), MultInt(2));
for_each(L.begin(), L.end(), Visit);
cout << endl;
return 0;
}

```

程序的输出结果为:

```

0 1 2 3 4 5 6 7 8 9
0 2 4 6 8 10 12 14 16 18

```

示例程序之二， `min_element/max_element`，找出容器中的最小/最大值：

```

#include
#include
#include
using namespace std;

int main(){
vector L;
for (int i=0; i<10; i++) L.push_back(i);
vector::iterator min_it =min_element(L.begin(),L.end());
vector::iterator max_it =max_element(L.begin(),L.end());
cout << "Min is " <<*min_it << endl;
cout << "Max is " <<*max_it << endl;
return 1;
}

```

程序的输出结果为:

```

Min is 0
Max is 9

```

示例程序之三， `sort` 对容器进行排序：

```

#include
#include
#include
using namespace std;

void Print(vector &L){
for (vector::iterator it=L.begin(); it!=L.end();it++)
cout << *it << " ";
cout << endl;
}

int main(){
vector L;
for (int i=0; i<5; i++) L.push_back(i);
for (int i=9; i>=5; i--) L.push_back(i);
Print(L);
}

```

```

sort(L.begin(), L.end());
Print(L);
sort(L.begin(), L.end(), greater()); // 按降序排序
Print(L);
return 0;
}

```

程序的输出结果为:

```
0 1 2 3 4 9 8 7 6 5
```

```
0 1 2 3 4 5 6 7 8 9
```

```
9 8 7 6 5 4 3 2 1 0
```

示例程序之四，`copy` 在容器间复制元素：

```

#include
#include
#include
using namespace std;
int main()
{
// 先初始化两个向量v1 和v2
vector v1, v2;
for (int i=0; i<=5; i++)v1.push_back(10*i);
for (int i=0; i<=10; i++) v2.push_back(3*i);
cout << "v1 = ( " ;
for (vector::iterator it=v1.begin(); it!=v1.end(); it++)
cout << *it << " ";
cout << ")" << endl;
cout << "v2 = ( " ;
for (vector::iterator it=v2.begin(); it!=v2.end();
it++)
cout << *it << " ";
cout << ")" << endl;
// 将v1 的前三个元素复制到v2 的中间
copy(v1.begin(), v1.begin()+3, v2.begin()+4);
cout << "v2 with v1 insert = ( " ;
for (vector::iterator it=v2.begin(); it!=v2.end();
it++)
cout << *it << " ";
cout << ")" << endl;
// 在v2 内部进行复制，注意参数2 表示结束位置，结束位置不参与复制
copy(v2.begin()+4, v2.begin()+7, v2.begin()+2);
cout << "v2 with shifted insert = ( " ;
for (vector::iterator it=v2.begin(); it!=v2.end();
it++)
cout << *it << " ";
cout << ")" << endl;
return 1;
}

```

程序的输出结果为:

```
v1 = ( 0 10 20 30 40 50 )
```

```
v2 = ( 0 3 6 9 12 15 18 21 24 27 30 )
```

```
v2 with v1 insert = ( 0 3 6 9 0 10 20 21 24 27 30 )
```

```
v2 with shifted insert = ( 0 3 0 10 20 10 20 21 24 27 30 )
```

[STL in ACM](#)

容器(container):

迭代器(iterator): 指针

内部实现: 数组 // 就是没有固定大小的数组, **vector**直接翻译是向量

vector // **T** 就是数据类型, **Alloc** 是关于内存的一个什么东西, 一般是使用默认参数。

支持操作:

begin(), //取首个元素, 返回一个**iterator**

end(), //取末尾 (最后一个元素的下一个存储空间地址)

size(), //就是数组大小的意思

clear(), //清空

empty(), //判断**vector** 是否为空

[] //很神奇的东东, 可以和数组一样操作

//举例: **vector a;** //定义了一个**vector**

//然后我们就可以用**a[i]**来直接访问**a** 中的第**i + 1** 个元素! 和数组的下标一模一样!

push_back(), **pop_back()** //从末尾插入或弹出

insert() **O(N)** //插入元素, **O(n)**的复杂度

erase() **O(N)** //删除某个元素, **O(n)**的复杂度

可以用于数组大小不定且空间紧张的情况

Iterator 用法举例:

```
int main(){
    int n,i;
    vector vi; //类似于我们定义一个数组, 同 int vi[1000];但vector
    的大小是自动调整的
    vector::iterator itr; //两个冒号
    while (scanf("%d",&n) != EOF)vi.push_back(n);
    for (i = 0 ; i < vi.size() ; i++)printf("%d/n",vi[i]);
    for (itr = vi.begin() ; itr != vi.end() ;itr++)
    printf("%d/n",*itr);
    return 0;
}
```

类似: 双端队列, 两头都支持进出

支持**push_front()**和**pop_front()**

是的精简版:) //栈, 只支持从末尾进出

支持**push()**, **pop()**, **top()**

是的精简版 //单端队列, 就是我们平时所说的队列, 一头进, 另一头出

支持**push()**, **pop()**, **front()**, **back()**

内部实现: 双向链表 //作用和**vector** 差不多, 但内部是用链表实现

list

支持操作:

begin(), **end()**, **size()**, **clear()**, **empty()**

push_back(), **pop_back()** //从末尾插入或删除元素

push_front(), **pop_front()**

insert() **O(1)** //链表实现, 所以插入和删除的复杂度的**O(1)**

erase() **O(1)**

sort() **O(nlogn)**, 不同于中的**sort**

//不支持[]操作!

内部实现: 红黑树 //Red-Black Tree, 一种平衡的二叉排序树

set //又是一个**Compare** 函数, 类似于**qsort**函数里的那个**Compare**函数, 作为红黑树在内部实现的比较方式

insert() **O(logn)**

erase() **O(logn)**

find() **O(logn)** 找不到返回**a.end()**

lower_bound() **O(logn)** 查找第一个不小于**k** 的元素

upper_bound() **O(logn)** 查找第一个大于**k** 的元素

equal_range() **O(logn)** 返回**pair**

允许重复元素的

的用法及**Compare** 函数示例:

```
struct SS {int x,y};
```

```
struct Itstr {
```

```
bool operator() (SS a, SS b)
```

```
{return a.x < b.x;} //注意，按C 语言习惯，double型要写成这样:
```

```
return a.x < b.x ? 1 : 0;
```

```
};
```

```
int main() {
```

```
set st;
```

```
...
```

```
}
```

内部实现: **pair** 组成的红黑树 //map 中文意思: 映射!!

map //就是很多**pair** 组成一个红黑树

insert() **O(logn)**

erase() **O(logn)**

find() **O(logn)** 找不到返回**a.end()**

lower_bound() **O(logn)** 查找第一个不小于**k** 的元素

upper_bound() **O(logn)** 查找第一个大于**k** 的元素

equal_range() **O(logn)** 返回**pair**

[**key**]运算符**O(logn)** *** //这个..太猛了，怎么说呢，数组有一个下标，

如**a[i]**,这里**i** 是**int** 型的。数组可以认为是从**int** 印射到另一个类型的印

射，而**map** 是一个任意的印射，所以**i** 可以是任何类型的!

允许重复元素, 没有[]运算符

内部实现: 堆 //优先队列，听**RoBa** 讲得，似乎知道原理了，但不明白干

什么用的

priority_queue

支持操作:

push() **O(n)**

pop() **O(n)**

top() **O(1)**

See also: **push_heap()**, **pop_heap()** ... in

用法举例:

priority_queue maxheap; //int 最大堆

struct Itstr { //又是这么个**Compare** 函数，重载运算符??? 不明

白为什么要这么写...反正这个**Compare** 函数对我来说是相当之神奇。**RoBa**

说了，照着这么写就是了。

```
bool operator()(int a,int b)
```

```
{return a > b;}
```

```
};
```

```
priority_queue minheap; //int 最小堆
```

```
1.sort()
```

```
void sort(RandomAccessIterator first,RandomAccessIterator
```

```
last);
```

```
void sort(RandomAccessIterator first,RandomAccessIterator
```

```
last, StrictWeakOrdering comp);
```

```
区间[first,last)
```

Quicksort,复杂度**O(nlogn)**

(**n=last-first**,平均情况和最坏情况)

用法举例:

1.从小到大排序(int, double, char, string, etc)

```
const int N = 5;
int main()
{
    int a[N] = {4,3,2,6,1};
    string str[N] = {"TJU","ACM","ICPC","abc","kkkkk"};
    sort(a,a+N);
    sort(str,str+N);
    return 0;
}
```

2.从大到小排序 (需要自己写comp 函数)

```
const int N = 5;
int cmp(int a,int b) {return a > b;}
int main()
{
    int a[N] = {4,3,2,6,1};
    sort(a,a+N,cmp);
    return 0;
}
```

3. 对结构体排序

```
struct SS {int first,second;};
int cmp(SS a,SS b) {
    if (a.first != b.first) return a.first
    return a.second < b.second;
}
```

v.s. qsort() in C (平均情况 $O(n\log n)$, 最坏情况 $O(n^2)$) //qsort 中的cmp 函数写起来就麻烦多了!

```
int cmp(const void *a,const void *b) {
    if (((SS*)a)->first != ((SS*)b)->first)
    return ((SS*)a)->first - ((SS*)b)->first;
    return ((SS*)a)->second - ((SS*)b)->second;
}
```

```
qsort(array,n,sizeof(array[0]),cmp);
```

sort()系列:

```
stable_sort(first,last,cmp); //稳定排序
```

```
partial_sort(first,middle,last,cmp); //部分排序
```

将前(middle-first)个元素放在[first,middle)中, 其余元素位置不定

e.g.

```
int A[12] = {7, 2, 6, 11, 9, 3, 12, 10, 8, 4,1, 5};
```

```
partial_sort(A, A + 5, A + 12);__
```

// 结果是"1 2 3 4 5 11 12 10 9 8 7 6".

Detail: Heapsort ,

$O((last-first)*\log(middle-first))$

sort()系列:

```
partial_sort_copy(first, last, result_first,result_last,
cmp);
```

//输入到另一个容器, 不破坏原有序列

```
bool is_sorted(first, last, cmp);
```

//判断是否已经有序

```
nth_element(first, nth, last, cmp);
```

//使[first,nth)的元素不大于[nth,last), $O(N)$

e.g. input: 7, 2, 6, 11, 9, 3, 12, 10, 8, 4,1, 5

```
nth_element(A,A+6,A+12);
```

Output: 5 2 6 1 4 3 7 8 9 10 11 12

2. binary_search()

```
bool binary_search(ForwardIterator first,ForwardIterator  
last, const LessThanComparable& value);
```

```
bool binary_search(ForwardIterator first,ForwardIterator  
last, const T& value, StrictWeakOrderingcomp);
```

在[first,last)中查找value， 如果找到返回True,否则返回False

二分检索， 复杂度 $O(\log(\text{last}-\text{first}))$

v.s. bsearch() in C

Binary_search()系列

```
itr upper_bound(first, last, value, cmp);
```

//itr 指向大于value 的第一个值(或容器末尾)

```
itr lower_bound(first, last, value, cmp);
```

//itr 指向不小于valude 的第一个值(或容器末尾)

```
pair equal_range(first, last, value, cmp);
```

//找出等于value 的值的范围 $O(2*\log(\text{last} - \text{first}))$

```
int A[N] = {1,2,3,3,3,5,8}
```

```
*upper_bound(A,A+N,3) == 5
```

```
*lower_bound(A,A+N,3) == 3
```

```
make_heap(first,last,cmp) O(n)
```

```
push_heap(first,last,cmp) O(logn)
```

```
pop_heap(first,last,cmp) O(logn)
```

```
is_heap(first,last,cmp) O(n)
```

e.g:

```
vector vi;
```

```
while (scanf("%d",&n) !=EOF) {
```

```
vi.push_back(n);
```

```
push_heap(vi.begin(),vi.end());
```

```
}
```

Others interesting:

```
next_permutation(first, last, cmp)
```

```
prev_permutation(first, last, cmp)
```

//both $O(N)$

```
min(a,b);
```

```
max(a,b);
```

```
min_element(first, last, cmp);
```

```
max_element(first, last, cmp);
```

Others interesting:

```
fill(first, last, value)
```

```
reverse(first, last)
```

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

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

//返回指针指向合并后的末尾处

```
random_shuffle(first, last, rand)
```

头文件

```
#include
```

```
#include
```

```
#include
```

```
#include
```

```
#include
```

```

#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
#include
using namespace std;

```

1 保留小数点后两位

```

#include

cout << setiosflags(ios::fixed) << setprecision(2)<< getAdd(num) << endl;

```

2 截取字符串 类似spilt

```

#include

const char * spilt="/";
char *p;
p=strtok(str,spilt);
while(p!=NULL)
{
    //cout << p << endl;
    num[i++]=atoi(p);
    p=strtok(NULL,spilt);
}

```

3 自动排序 sort函数

```

#include
#include
sort(Rs.begin(),Rs.end());
sort(Rs.begin(),Rs.end(),greater());

```

4 开方函数

```

#include
return 2*sqrt(R*R-b*b/4);

```

5断点调试

```

cout << "cout%len " << count%len << ' ' << "num " << num << endl;

```

6基本格式

```

#include
using namespace std;
int main()
{
    int count;
    while(cin >> count)
    {

```

```
    }  
    return 0;  
}
```

7 关于排序

```
bool cmp(int a,int b)  
{  
    return abs(a)>abs(b);  
}  
sort(vec.begin(),vec.end(),cmp);
```

8 求字符串长度

```
strlen(str)
```

9//cin.getline(字符指针,字符个数N,结束符);

//结束符(默认的是以'\n'结束)

```
while(cin.getline(a,100))
```

10 字符串比较

```
//strcmp(字符串1， 字符串2)
```

```
//s1<0 ; s1=s2 0 ;s1>s2 >0
```