编程风格

2015年2月6日 11:02

对于作为函数界定符的花括号，习惯上将其单独一行。
如果对象只需读而无须写的话，习惯要把其设为const

标识符命名规则：
1.不能出现连续两个下画线。
2.不能以下画线紧连大写字母开头。
3.定义在函数体外的标识符不能以下画线开头。
4.自定义类名最好以大写字母开头。
5.变量名一般消协
6.多个单词时，以驼峰状区分： StudentLoans

边缘知识

2015年2月6日　　10:52

注释不能嵌套

初始化：在创建时获得一个特定值
默认初始化： 如果定义变量时，没有指定初值，则执行默认初始化
　　默认初始化规则：
　　1.定义于任何函数之外的变量被初始化为 0 。
　　2.定义于函数内部的内置类型变量将不被初始化。
　　3.每个类自己决定其初始化对象的方式。（绝大多数类都支持无须显式初始化定义对象，类为其提供了一个合适的默认值）

定义与声明：
声明：一个文件如果想使用别处定义的名字则必须包含对那个名与字的声明。
定义：负责创建与名字关联的实体

声明的用法： 在变量前添加extern
　　例如： extern int i ;
定义只能一次，但是声明可以有多次。

左值与右值：

左值：可以放在赋值符号左边的变量，左值表示存储在计算机内存的对象，即具有对应的可以由用户访问的存储单元。左值相当于地址值。
右值：当一个符号或者常量放在操作符右边的时候，计算机就读取他们的“右值”，也就是其代表的真实值，右值相当于数据值。右值指的是引用了一个存储在某个内存地址里的数据。

头文件不应该包括using 声明！

优先级：

　　1.cout << i << ++i << endl; //这是未定义行为！！！ 因为编译器没有规定运算

顺序，可能先算i，也可能先算++i
2.f()+g()+h();
假如上述函数影响同一对象，则是错误的，会产生未定义行为！
3.i=i++；是未定义的！！！

i++；对运算对象+1，但求值结果是对象改变前的那个值的副本。
建议无关系时，采用前置增版本！

运算符优先级

2015年2月11日　　10:45

::	. / −>	[] / ()	前缀++/--	cast类型转换	后缀++/--	~	!	正负号
*解引用	&取地址	()类型转换	sizeof	new	乘除	%取模	加减	移位
关系运算符	&/|/^	&& ||	? :	=	,			

迭代器

2015年2月8日 15:14

所有标准库容器都有迭代器

auto beg = v.begin() ; 返回第一个元素的迭代器
auto end = v.end() ; 即尾后迭代器。指向容器尾元素的下一位置
若容器为空，则两者返回的是同一个迭代器

迭代器的运算符：
iter -> mem 等价于 (*iter).mem
-> 运算符能把 * 和 . 两个操作结合在一起

检查迭代器it指向的对象是否为空，要用iter -> empty() 或者 (*iter).empty()
iter.empty错误！！

两个迭代器不能相加！！！

迭代器类型：
vector<T>::iterator it ;
vector<T>::const_iterator ;

计算最接近中间元素的迭代器的算法：
1. auto mid = v.begin()+v.size()/2;
2. auto mid = v.begin()+ (v.end()-v.begin()) / 2 ;

forward_list 不支持递减运算符(--)

迭代器范围：迭代器范围内的元素包括：［begin , end) 即左闭合区间

插入迭代器：是一种迭代器适配器
接受一个容器，生成一个能实现向给定容器添加元素的迭代器
通过对一个插入迭代器赋值，该迭代器调用容器操作来给容器指定位置插入一个元素

对插入迭代器++/* 没有任何意义，仍然会返回迭代器本身

插入迭代器有三种类型：

back_inserter(C)	创建一个使用push_back的迭代器
front_inserter(C)	创建一个使用push_front的迭代器
inserter(C， p)	创建一个使用insert的迭代器，p为指向容器C的一个迭代器 赋值后元素被插入到p所表示元素之前

iostream 迭代器：要include〈iterator〉
istream_iterator： 读取输入流
ostream_iterator ：向一个输出流写数据
这些迭代器将它们对应的流当作一个特定类型的元素序列来处理
通过使用这些迭代器，我们可以使用泛型算法从流的对象读取数据和写入数据！！！
当创建一个流迭代器时，必须指定迭代器将要读写的对象类型，还可以将它绑定到一个流:如：
istream_iterator<int> int_it (cin) ;
还可以用默认初始化创建一个可以当作尾后迭代器值使用的迭代器

例子：从标准输入读取数据，存入一个vector
istream_iterator<int> in_iter(cin) ; //从cin读取int
istream_iterator<int> eof ; //istream 尾后迭代器

while(in_iter!=eof)
    vec.push_back(*in_iter++)
这个循环每次从cin读取int值，再保存在vec中，eof被定义成空的isream_iterator，从而可以当作尾后迭代器来使用，一旦流读取时遇到文件尾或者IO错误，则迭代器的值与尾后迭代器相同！！！

更加有用的例子：
istream_iterator<int> in_iter(cin) , eof ;
vector<int> vec(int_iter,eof) ;
这个语句会从cin中一直读取数据，直到遇到文件尾或者一个不是int的数据为止！

当将istream_iterator绑定到一个流时，标准库并不保证迭代器立即从流中读取数据。
但是总是会正常读取的，例如下面这个程序，

```
istream_iterator<int> in_iter(cin), eof;
vector<int> vec(in_iter, eof);
for(auto i : vec){
   cout << i ;
}
return 0;
```

假如输入1 2 3 4 ^z，则输出仍然就是1234 ， 不会吞掉最开始输入的数据！！

istream_iterator操作：

istream_iterator<T> in(is)	in从is流中读取类型尾T的值
istream_iterator<T> end	读取类型为T的值的istream_iterator迭代器，表示尾后位置
in1==in2	in1和in2必须读取相同类型，如果它们都是尾后迭代器，或绑定到相同的输入，则相等
*in	返回从流中读取的值
in->mem	
++in , in++	读取流的下一值，前置版本返回一个指向递增后的迭代器的引用 后知版本返回旧值

ostream_iterator:

ostream_iterator的操作

ostream_iterator<T> out(os) ;	out将类型T的值写到输出流os中
ostream_iterator<T> out(os,d);	out将类型为T的值写到输出流os中，每个值后面都输出一个d。 d是一个指向空字符结尾的C风格字符串 （格式化输出时很有用！！！）
out = val	用<<运算符将val写入到out所绑定的ostream中，val的类型必须与out可写的类型兼容
*out , ++out , out++	没有意义，仍然返回out

```
ostream_iterator<int> out_iter(cout, " ");
for (auto e : vec)
   out_iter = e;          // 把e写入cout
                          //相当于*out_iter++ = e， *和++对于ostream_iterator可以省略！
   cout << endl;
```

还可以调用copy()来打印vec中的元素

```
copy(vec.begin(),vec.end(),out_iter);
cout << endl;
```

反向迭代器：

递增（++）一个反向迭代器会移动到前一个元素
有：四种类型
C.rbegin();
C.rendl();
C.crbegin();
C.crend;

可以通过sort一对反向迭代器来使容器整理为递减序！！！
若riter是反向迭代器，riter.base() 返回一个其对应的普通迭代器

类型

2015年2月6日　10:53

main 函数的返回值类型必须是int
内置类型：语言自身定义的类型

float ：6位有效数字
double ： 10位有效数字
int :　　16位　　5位数（max=32768）
long :　　32位　　10位数
long long :　64位　　19位数　　（C++11）

无符号型，类型前缀加 “unsigned”

选择类型的经验准则：
1.当确定数不为负时，用无符号
2.超过int 时就用 long long
3.浮点数选用double ， 因为double与float计算代价差不多，甚至double还更快

类型转换：
unsigned char = -1 ;　　对于无符号char，-1就是255

把浮点数赋值给整数型时，将只保留小数点前面的整数部分

当赋给无符号类型一个超出它所表示范围的值时，结果是初始值对无符号类型表示数值总数取模后的余数。
例如：unsigned char ch = 256;
256对255取模后的余数为-1，−1赋值给ch 就转换为了 255.

当赋给带符号类型一个超出它所表示范围的值时，是未定义的
ps：余数的定义：
a=qd+r　　// r为余数，a,d,q为整数，d!=0

切勿混用带符号类型和无符号类型：
例：unsigned u = 10 ;
int i =-42 ;
cout << i + i << endl;　//输出-84
cout << u + i << endl;　//输出4294967264
因为 u + i 相加时先把i变成无符号数，带符号数转换成无符号数会等

于这个负数加上无符号数的模

隐式类型转化：

1. 本质：被设计得尽可能避免损失精度
2. 小的类型被提升为大的类型
3. 右侧运算对象被转化成左侧运算对象
4. 函数调用时也会发生类型转化
5. 数组只有在decltype、& 、sizeof、typeid 等运算符时，才会被视为数组，其余时刻被隐式转化为指针。

显式转化：

强制类型转化： cast-name<type>(expression)

type是要转化的目标类型

若type时引用类型，则结果是左值

cast-name指定执行哪种类型转换

cast-name：

1. static_cast ：普遍适用。
2. const_cast ： 将常量对象转换成非常量对象，去const性质！

旧式的强制类型转换：

1. type (expr)
2. (type) expr

const：

const 对象一旦创见后值就不能改变，所以const对象必须初始化
例如： const int i = 12 ;
编译器在编译时，会把用到i的地方全部用12替代！！
且，const对象仅在当前文件内有效

类型别名 typedef

两种常规方法：
typedef double wages ; //wages是double的同义词
using SI = Sales_item ; //SI 是Sales_item的同义词（C++11）

auto

auto让编译器通过初始值来推算变量类型 （C++11）
所以auto的定义必须有初始值！

auto语句也能在一条语句中声明多个变量，但是因为一条声明语句中只能有一个基本的数据类型，所以改语句中所有变量的初始基本数据类型必须一样。

decltype
作用：返回操作数的数据类型
在此过程中，编译器分析表达式并得到它的类型，但不实际计算表达式的值！
例如：decltype(f()) sum =0；

与引用：
例如： int i =42 ;
int &r = i ;
decltype(r) a ; //错误，这时a的类型是 int & 引用类型，必须要初始化
decltype(r+0) b ； // 正确，这时r+0返回一个具体值而不是一个左值！

如果decltype() 中，使用一个不加括号的变量，则得到的结果是该变量类型
使用一个加了一层或多层括号的变量，编译器就会把它当成一个表达式，这时会变成引用！！
例如：int i =42 ;
decltype(i) e ; //正确， e 是int
decltype((i)) e ; //错误，e 是 int & ，必须初始化

——> 切记！！！ decltype(()) 的结果永远是引用！

布尔类型字面值： true false
指针字面值　　： nullptr
布尔值不应该参与运算！

符号

2015年2月6日　　10:55

操作符：

endl：结束当前行，并将缓冲区刷新

. 点运算符只能用于类类型对象

%只能用于两个整数之间操作

位运算符：

~	位求反	~i	0变1，1变0
<<	左移	a<<b	在右侧插入0，前者为处理对象，后者为移动位数
>>	右移	a>>b	在左侧插入0，前者为处理对象，后者为移动位数
&	位与	a&b	如果a,b对应位都是1，则为1，否则为0
^	位异或	a^b	如果a,b对应位有且只有一个为1，则为1，否则为0
|	位或	a|b	如果a,b对应位至少有一个为1，则为1，否则为0

虽然运算对象可以为正或负，但是负的时候符号位如何处理没有规定，所以未定义
建议仅将位运算用于无符号类型

使用移位运算符：

```
1UL << 27          //生成一个值，该值只有第27位为1
                  //注意！二进制数也是以整型保存的！
```

例如：统计学生成绩：

```
    unsigned long quiz1 = 0;     //int只有16位。long有32位
    quiz1 |= 1UL << 27 ;         //学生27通过测试，其余置0
    //注意！！可以用|=  ！！
    // quiz1 &= ~(1UL << 27);      //后来发现学生27没有通过，置为0
    bool status = quiz1 & (1UL << 27);//检测第27个学生的通过情况
    //注意！！千万不能cout << queze1 ，这样会输出10进制的数，而且C++也没
有二进制的格式控制符
    只能用下面的方法来输出二进制！！！
    while (quiz1 != 0)    //不是 quiz1 /2 != 0
    {
```

```
        cout << quiz1 % 2;
        quiz1 /= 2;
    }
```

//且这样输出以后会有28位，因为第一位默认为0，因为我们是移动n位代表第n个学生，而本来就是存在1位的！

sizeof 运算符：

1. sizeof 并不实际计算其运算对象的值
2. 对指针执行sizeof只会对指针计算大小，不会对其指向对象计算大小
3. 对解引用指针执行sizeof 运算得到指针所指向的对象所占空间的大小
4. 对数组执行sizeof得到整个数组所占空间大小（sizeof不会把数组当成指针）
5. 对string / vector 对象执行sizeof 只返回该类型固定部分大小，不计算元素所占空间

例子： sizeof(array)/sizeof(*array) // 返回数组array 的元素数量

字符以及字符串

2015年2月6日　12:30

字面值常量：形如 42 这样的就是字面值常量，每个字面值常量对应一种数据类型。

字面值的类型是对应数据类型最小的那个。
如果一个字面值连与之关联的最大数据类型都不能容纳下，则产生错误。

字符字面值：　　' a '
字符串字面值：　"Hello Wold!"

1.字符串字面值的类型实际上是由常量字符构成的数组。
2.编译器在每个字符串的结尾添加一个空字符'\0' ，因此字符串字面值实际长度多1
3.如果两个字符串字面值位置紧邻且仅由空格、缩进、换行符分隔，则它们实际上是一个整体。
例如："haha"　"ni hao"
　　" ni hao "
　　算作一个字符

定义和初始化string

```
string s;
string s2=s1;
string s2( s1 ) ;
string s = " haha ";
string s ( "haha" ) ;
string s( 10, 'c' ) ;      //得到ccccccccc
                |----->不能是字符串！
```

————————〉string 的最后一位不是'\0'！！！

如果直接使用 ‘=’是拷贝初始化
如果不使用 ‘=’是直接初始化

string 中的最后一个元素是str[str. size()-1]

string 的操作：

is >> s　　若is是cin ，则遇到空白符就停止！

getline (is , s) ;
1.换行符会被is读进来，但是不会存入s中，一开始读入换行符则s为空，若要保留输入时的分行格式，则必须手动添加。
2.执行读取操作时，string 会自动忽略开头的空白（即空格，换行，制表等），并从一个真正的字符开始读起，直到遇到下一处空白

s.empty() ; 是否为空

s.size() ;　返回字符个数（string::size_type类型）——ps:如果一个表达式中已经有了size函数，就不要用int了，避免混用数据类型可能带来的问题。

s[n] ;

< , > , <= , >= ,== ,!=　比较两者第一对相异字符的字典序（a>A）

+　当string对象和字面值混在一条语句中时，必须确保每一个+号两侧的运算对象至少有一个是string。如：string s= "aaa"+"bbb"+ str ; 是错误的！

处理string中的字符：

一下表格中的函数都在头文件cctype中：

isalnum(c)	当c是字谜或数字时为真
isalpha(c)	当c是字母时为真
iscntrl(c)	当c为控制字符时为真
isdigit(c)	当c是数字时为真
isgraph(c)	当c不是空格但可打印时为真
islower(c)	当c是小写字母时为真
isprint(c)	当c是可打印字符时为真
ispunct（c）	当c是标点符号时为真
isspace(c)	当c是空白时为真（空格，制表符，回车，换行，进纸符）

isupper（c）	当c是大写字母时为真
tolower	大写变小写，小写不变
toupper	小写变大写，大写不变

C风格字符串：
（尽量不要使用）

本质：字符数组且以'\0'结尾
C风格字符串函数：

strlen(p)	返回长度
strcmp(p1,p2)	比较p1，p2的相等性，若==，返回0，若p1>p2,返回一个正值,若p1<p2则返回一个负值
strcat(p1,p2)	将p2附加到p1之后，返回p1
strcpy(p1,p2)	将p2拷贝给p1，返回p1

注意，C风格字符串比较的时候，只能用strcmp，不能用> < ，因为对于编译器数组是指针，实际上比较的变成了两个指针，地址之间是不能比较的！

混用string 与 C风格字符串：
允许以C风格字符串给string赋值或者初始化
在string的加法中允许使用C风格字符串作为其中一个运算对象

string中有 c_str() 成员函数， 把string变成C风格字符串
s.c_str() ， 返回s的C风格字符串

多维数组：

严格来说，C++没有多维数组，多维数组其实是数组的数组。
当一个数组的元素仍然为数组时，前一个维度表示数组本身大小，后一个维度表示其元素（也是数组）的大小。
例如：int a[3][4] ； //大小为3的数组，每个元素是含有4个整数的数组

输入输出

2015年2月6日　10:59

当遇到文件结束符（end-of-file）, 或者一个无效输入时，istream对象的状态会变为无效，即为假

end of file 在 Mac中是 ctrl + d

一个流一旦发生错误，后续的IO操作都会失败。
确定一个流对象状态最简单的方法是：

```
while(cin >> word ){
    。。。
}
```

导致缓冲刷新的原因有：
1.程序正常结束。
2.缓冲区满
3.endl　　换行并刷新缓冲区
4.flush　　仅仅刷新缓冲区
5. unitbuf　在每个输出操作之后刷新缓冲区
6. nounitbuf　回到正常缓冲方式
...
（PS：如果程序崩溃，缓冲区不会被自动刷新，调试时，要确保已经刷新了缓冲区，否则会导致错误信息残留）

文件输入输出：

头文件：fsream 定义了三个类型：

ifstream	从给定文件读取数据
◆ ofstream	从给定文件写入数据
fstream	可读写给定文件

fstream特有的操作

fstream　　fstrm ;	创建一个未绑定的文件流
fstream　fstrm(s) ;	创建一个fstream，并打开并绑定名为s的文件，模式默认
fstream　fstrm (s , mode);	创建一个fstream，并按指定模式打开名为s的文件
fstrm.open (s)	打开名为s的文件，并将文件与fstrm绑定。模式默认

fstrm.close()	关闭与fstrm绑定的文件，返回void
fstrm.is_open()	返回一个bool值，是否成功打开与fstrm关联的文件

mode 文件模式：

in	以读方式打开
out	以写方式打开
app	每次写操作前均定位到文件末尾
ate	打开文件后立即定位到文件末尾
trunc	截断文件（即清空文件）
binary	以二进制方式进行IO

指定文件模式限制：

1. 只有当out也被设定时，才可以设定trunc模式
2. 只要trunc没被设定，就可以设定app

没有指定模式时以默认模式打开：

1. 与ifstream关联的文件默认以in模式打开
2. 与ofstream关联的文件默认以out模式打开
3. 与fstream关联的文件默认以in和out模式打开

默认情况下，以out模式打开文件会丢失已有数据
阻止文件被清空的方法是：
1. 同时指定app模式
如： ofstream app("file", ofstream::app)； //隐含打开文件
ofstream app("file", ofstream::out | ofstream::app)；
2.或者以fstream打开（因为包含了in模式，所以不会被清空）

每次调用open时都可以改变其文件模式：

```
ofstream out;
out.open("file"); //模式隐含输出和截断
out.open("file", ofstream::app); //模式为输出和追加
```

通常情况下，out模式意味着同时使用trunc模式

头文件sstream定义了三个类型来支持内存 IO，这些类型可以向string写/读入数据，就像string是一个IO流一样

istringstream 从string读取数据
ostringstream 从string写入数据
stringstream 既可以读也可以写

stringstream 特有的操作：

sstream strm;	strm是一个未绑定的stringstream对象，sstream是头文件sstream中定义的一个类型。
sstream strm(s);	strm是一个sstream对象，保存一个s的拷贝
strm.str()	返回strm所保存的string拷贝
strm.str(s)	将s拷贝到strm中，返回void

一个使用istringstream存入人电话号码的例子：
istringstream 主要用于一些工作对整行进行处理，一些工作对行内单个单词进行处理时。

输入示例： lee 602432 3423526 34325346
draw 178281919

```
struct PersonInfo {
   string name;
   vector<string> phones;
};

string line, word;
// will hold all the records from the input

vector<PersonInfo> people;
// read the input a line at a time until end-of-file (or other error)

while (getline(is, line)) {
   PersonInfo info;            // object to hold this record's data
   istringstream record(line); // bind record to the line we just read
   record >> info.name;        // read the name不会读取空格以及空格后的东西，相当于cin
   while (record >> word)      // read the phone numbers 从空格后开始读取
      info.phones.push_back(word);  // and store them
   people.push_back(info); // append this record to people
}
```

当我们逐步构造输出，最后希望一起打印时，优选用 ostringstream

操纵符控制输出输出格式：

定义在iostream中的操纵符	用法如：cout << boolalpha <<bool_val << noboolalpha ;
boolalpha	将ture和false输出为字符串形式
noboolalpha	恢复默认1，0输出
showbase	对整型输出表示进制的前缀： 前缀0x　16进制 前缀0　8进制
noshowbase	
showpoint	对浮点值总是显示小数点
noshowpoint	
showpos	对非负数显示+
noshowpos	
uppercase	在16进制中打印0X，在科学计数法中打印E
nouppercase	在16进制中打印0x，在科学计数法中打印e（默认）
dec	整数显示十进制
hex	整数显示十六进制
oct	整数显示八进制
left	左对齐，值右侧添加填充字符

right	右对齐（默认）
internal	控制负数的符号的位置，它左对齐符号，右对齐值，中间用空格填满
scientific	浮点数显示为科学记数法
hexfloat	浮点值显示为十六进制(c++11)
defaultfloat	重置浮点值显示为十进制(c++11)
unitbuf	每次输出后都刷新缓冲区
nounitbuf	恢复正常的缓冲区刷新方式
skipws	输入运算符，跳过空白符（默认）
noskipws	输入运算符，不跳过空白符（可以使输入时，读取空白符）
flush	刷新ostream缓冲区
ends	插入空字符，然后刷新ostream缓冲区
endl	插入换行，然后刷新ostream缓冲区
定义在iomanip中的操作符	
setw(n)	下一个（不是全部）数字或者字符串的最小空间为n
setprecision(n)	将浮点数精度设置为n
setfill('c')	用指定字符代替默认的空格来补白输出
setbase(b)	将整数输出为b进制（对2进制无效！）

引用

2015年2月6日　　13:47

定义引用时，把引用和它的初始值绑定在一起，而不是把初始值拷贝。

无法令引用重新绑定到另外一个对象，因此引用必须初始化
引用即别名

引用本身不是一个对象，故不能定义引用的引用。

一般情况下，引用都要和与之绑定的对象类型严格匹配。

指针

2015年2月6日　　13:50

指针与引用的不同点
1.指针本身就是个对象
2.允许对指针赋值和拷贝
3.可以更改指针指向
ps：引用只是对象的一个别名，而指针复杂多了

因为引用不是对象，没有实际地址，所以不能定义指向引用的指针
但是可以定义指向指针的引用

获得空指针的三种方法：
int *pr = nullpr;　　(优选，C++11)
int *pr = NULL;
int *pr = 0;

建议：初始化所有的指针。

空指针无地址，即地址=0
所有：while(pr){
　　...
　　}

　　直到真正为空时终止。

语句

2015年2月8日 14:42

范围 for ：

```
for ( declaration : expression ){
    ......
}
```

每次迭代，declaration 部分的变量会被初始化为 expression 的下一个元素值。
若想要在之后的……中改变对象中的值，必须把循环变量定义成引用！

```
for ( auto &c : s ) {
}
```

范围for循环内，不能改变其所遍历序列的大小！

while 语句条件内以及循环体内的变量每次迭代都经历从创建到销毁的过程。

break 语句负责终止离他最近的 while , do-while , for， switch 语句，并从这些语句后的第一条开始继续执行

continue 语句终止最近的循环中的当前迭代并立即开始下一次迭代。
只能出现在 for , while , do while 循环内部。

return 语句的作用是把控制权交到调用该函数的地方
void 函数可以直接 return;

进制

2015年2月8日　　14:47

16进制的数组映射表示方法：
const string hexdigits = "0123456789ABCDEF" ;
这样就导致了hexdigits[11]=A　，　10进制和16进制就对照起来了！

模板

2015年2月8日 14:49

有类模板和函数模板

实例化：编译器根据模板创建类或函数的过程成为实例化

数组

2015年2月8日　　15:38

字符数组的特殊性：
最后一定要有一个空字符'\0' ， 若没有，则系统自动添加'\0'
数组下标类型：size_t

用数组的时候编译器会把它转化为指针。
所有当使用数组作为一个 auto 时，推断得到的是指针而非数组。
但是当使用decltype()时，返回数组类型，
例如：int a[10]={0};
　　decltype(a) b;
　　则b是大小为10的数组！（切记，连同数组大小一同返回！！！）

数组不是类类型，故没有成员函数，但是可用标准库函数
int *beg = begin (array) ; 指向首元素
int *end = end (array) ; 指向array尾元素的下一位置
begin() ， end() 定义在iterator头文件中

内置的下标运算符所用的索引值不是无符号型，这与vector和string不同

函数

2015年2月9日 15:22

args 即 arguments ?

函数在头文件中声明，在源文件中定义。

每次调用函数，会用传入的实参对形参进行创建并初始化。
如果形参是引用类型，则会被绑定到对应实参上去

对于指针形参：当执行指针拷贝操作时，拷贝的是指针的地址，拷贝之后，两个形参的指针和实参的指针是两个指针，但是因为指针使我们可以间接地访问它所指的对象，所以通过指针可以修改对象值。

建议：在C++中优选引用，哪怕不需要改变实参值，也优选引用。
（因为拷贝大的类类型对象或者容器对象比较低效）
如果函数无须改变引用形参的值，最好将其声明为常量引用！
如 const string &str

虽然函数只能返回一个值，但是引用形参给了我们一个方法可以等效地返回多个结果。

当实参有const时，这时它传给常量或者非常量对象都是可以的！

不要return返回局部对象的引用或指针

重载函数：
函数名字相同但是形参数量或者类型不用的函数。
编译器会根据传递的实参类型推断想要的是哪个函数。

默认实参：

1.在函数的多次调用中都被赋予一个相同的值的形参
2.默认形参的初始化可以出现在函数的形参列表中。但是！一旦一个形参有了默认值，则之后的所有形参都必须有默认值
3.如果要使用默认形参，则只要在调用函数的时候省略该实参就行了

重载函数的最佳匹配：

> 基本思想：寻找实参与形参类型最接近的那个
> 若检查之后没有一个重载函数能脱颖而出的，那么调用错误

随机数

2015年2月22日　　15:41

定义在头文件<random>中

```
default_random_engine  e;     //生成随机无符号数
e();              //调用对象来生成下一个随机数
```

随机数引擎操作	
engine e ;	默认生成随机数
engine e(s) ;	使用整型值s作为种子
e.seed(s)	使用种子s重置引擎状态
e.min()	
e.max()	

选择一个种子：

可以使用时间：

```
头文件：<ctime>
default_random_engine el(time(0)) ;
//time()返回从一个特定时刻到现在有多少秒
//只适用于前后间隔时间很长的情况，否则都在一秒内会导致随机数相同
```

顺序容器综述

2015年2月11日 12:54

每一个容器都定义在一个与容器类型名相同的头文件中

顺序容器类型：

vector	可变大小数组	支持快速随机访问	在尾部添删元素很快
deque	双端队列	支持快速随机访问	在头尾添删元素很快
list	双向链表	只支持双向顺序访问	在任何位置添删元素都很快
forward_list	单向链表	只支持单向顺序访问	在任何位置添删元素都很快
array	固定大小数组	支持快速随机访问	不能添删元素
string	与vector相似，专门保存字符	支持快速随机访问	在尾部添删元素很快

string 和 vector 将元素保存在连续的空间中，所以用下标来计算其地址是非常快速的，但是在中间位置操作就会很耗时，因为在一次中间位置插入或删除元素后，需要移动操作位置之后的所有元素。

建议：现代C++程序应该使用标准库容器，而不是更原始的数据结构，如内置数组
通常使用vector是最好的选择

确定使用哪种顺序容器：

1. 一般优选vector
2. 要求在中间插入或删除元素时，用list或者forward_list
3. 只在头尾添删用deque
4. 在输入时需要在中间位置插入元素，其后又要能随机访问时，一种假如是可以通过排序使得要插入位置变成最后一个的话就用vector，另一种倘若不能通过排序解决可以考虑在输入时用list，输入完成后把内容拷贝到vector

容器操作：

类型别名	
iterator	此容器类型的迭代器类型
const_iterator	只能读取元素但不能修改元素的迭代器类型
size_type	无符号整数类型，足够保存此种容器类型最好可能容器的大小
difference_type	带符号整数类型，足够保存两个迭代器之间的距离
value_type	元素类型
reference	元素的左值类型
const_reference	元素的const左值类型
构造函数	
C c;	
C c1 (c2); C c1=c2 ;	支持一个容器到另一个容器的拷贝，两个容器类型必须相同
C c(b , e);	将迭代器b和e指定的范围内的元素拷贝到c（array不支持）
C c{a, b, c, . . . } ; C c={a, b, c. . . } ;	

C c（n）； C c(n,t)；	n个元素 n个初始化为t的元素
赋值，swap，assign	
c1 = c2	
c1 = {a,b,c...}	(array不支持)
a.swap(b) 成员函数	交换a,b的元素。（两个容器类型相同）
swap (a,b) 非成员函数	等价（swap：元素本身并未交换，只是交换了两个容器内部的数据结构） （sway不进行拷贝删除或者插入操作，故迭代器指针引用都不会失效（除非sway两个array））
seq.assign (b,e)	将seq中的元素替换为迭代器b和e所表示的范围中的元素，b,e不能指向seq （array和关联容器不适用）
seq.assign (il)	将seq中的元素替换为初始化列表il中的元素（array和关联容器不适用）
seq.assign(n,t)	将seq中的元素替换为n个值为t的元素 （array和关联容器不适用）
大小	
c.size()	c中元素的数目（不支持forward_list）
c.max_size()	c可保存的最大元素数目
c.empty()	若c中存储了元素，返回false ，否则返回 true
添加/删除元素 （容器通用）	（不适用array）注：不同容器中接口不同
c.insert(args)	将args拷贝进c
c.emplace(inits)	使用inits构造出c中的一个元素
c.erase(args)	删除args指定元素
c.clear()	删除c中所有元素
向顺序容器添加元素	这些操作会改变容器大小，array不支持这些操作 forward_list有自己专有版本的insert和emplace forward_list不支持push_back和emplace_back vector和string不支持push_front 和 emplace_front
c.push_back(t) c.emplace_back(args)	在c的尾部创建一个值为t或由args创建的元素，返回void
c.push_front(t) c.emplace_front(args)	在c的头部创建一个值为t或由args创建的元素，返回void
c.insert(p,t) c.emplace(p,args)	在迭代器p指向的元素之前创建一个值为t或者由args创建的元素 返回指向新添加元素的迭代器
c.insert(p,n,t)	在迭代器p指向的元素之前插入n个值为t或者由args创建的元素 返回指向新添加的第一个元素的迭代器，若n为0，返回p
c.insert(p,b,e)	在迭代器p指向的元素之前插入b和e指定范围内的元素（两端都可取到，） 返回指向新添加的第一个元素的迭代器，若范围为空，返回p
c.insert(p,il)	il是一个花括号包围的值列表，将这些值插入到p之前，返回新添加第一个元素的迭代器
	PS：向一个vector,string,deque插入元素，会使所有指向容器的迭代器、引用和指针失效
	使用这些操作时，要切记各类容器自身特性避免影响性能
向顺序容器删除元素	不适用与array forward_list有特殊版本的erase forward_list不支持pop_back； vector和string 不支持pop_front
c.pop_back()	删除c中尾元素

c.pop_front()	删除c中首元素
c.erase(p)	删除迭代器p所指定的元素，返回一个指向被删元素之后的元素的迭代器。 若p指向尾元素，返回尾后迭代器，若p是尾后迭代器，则函数行为未定义
c.erase(b,e)	删除迭代器b和e所指定范围内的元素。返回指向最后一个被删元素之后元素的迭代器。 若e本身就是尾后迭代器，则返回尾后迭代器
c.clear()	删除c中所有元素，返回void
c.back()	返回c中尾元素的引用，c为空时未定义
c.front()	返回c中首元素的引用，c为空时未定义 PS:注意，begin()和end()时返回迭代器，而这两个是返回引用！
c[n] / c.at(n)	返回下标为n的元素的引用
改变顺序容器大小	如果当前大小大于所要求大小，则删除后部元素 如果当前大小小于所要求大小，则将新元素添加到后部 不适用与array
c.resize(n)	调整c大小为n个元素
c.resize(n,t)	调整c大小为n个元素，任何新添加的元素都初始化为t
关系运算符	
= !=	所有容器都支持
< , <= , > , >=	无序容器不支持
	比较两个容器实际上是进行元素的逐对比较
获取迭代器	
c.begin() , c.end()	
c.cbegin(),c.cend()	返回const_iterator （当不需要写时，最好用带c版本！）
反向容器的额外成员	不支持forward_list
reverse_iterator	按逆序寻址元素的迭代器
const_reverse_iterator	不能修改元素的逆序迭代器（但是迭代器可++）

容器操作使迭代器失效的几个情况：

添加元素时：

i. 对于vector与string，如果存储空间被重新分配，则指针、引用都会失效
   如果存储空间未重新分配，指向插入位置之前的元素的迭代器、指针、引用仍有效，之后的失效

ii. 对于deque，如果在首尾添加元素，迭代器失效，但是指向存在元素的指针和引用不会失效
   如果在首尾之外添加元素，则指针引用迭代器全部失效

iii. 对于list 和forward_list， 指向容器的迭代器（包括尾后迭代器和首前迭代器）、指针、引用仍有效

删除元素时：

i. 对于vector与string，指向被删除之前的元素的迭代器引用指针仍有效。

ii. 对于deque，如果删除首尾元素，则除了尾后迭代器，别的迭代器、指针、引用仍然有效
   如果删除首尾之外的元素，则全部失效

iii. 对于list 和forward_list，全部仍然有效

string

2015年2月12日 13:45

构造string的方法：	
string s(cp, n)	s是cp指向的数组中前n个字符的拷贝，此数组至少含n个字符
string s(s2,pos2)	s是string s2 从下标 pos2 开始的字符的拷贝，若pos2>s2.size()，则未定义
string s(s2, pos2, len2)	s是string s2 从下标pos2开始len2个字符的拷贝。
子字符串操作	
s.substr(pos,n)	返回一个string，包含s中从Pos开始的n个字符的拷贝。 pos默认值为0，n默认值为s.size()-pos，即拷贝从pos开始的所有字符。
修改string的操作	
s.insert(pos,args)	在pos之前插入，pos可以是一个下标或者迭代器 接受下标版本：返回一个指向s的引用； 接受迭代器版本：返回指向第一个插入字符的迭代器
s.erase(pos,len)	删除pos开始的len个字符，若len省略，则删至末尾为止 返回指向s的引用
s.assign(args)	将s中的字符替换为args指定的字符，返回一个指向s的引用
s.append(args)	将args追加到s后，返回一个指向s的引用
s.replace(range,args)	删除s中范围range内的字符，替换为args指定的字符。 range是一个下标和一个长度，或者是一对指向s的迭代器 返回指向s的引用
args可以是下面形式之一	
str	字符串
str,pos,len	str中从pos开始最多len个字
cp,len	从cp指向的字符数组的前len个字符
cp	cp指向的以空字符结尾的字符数组（cp即c风格字符串）
n,c	n个字符c
b,e	迭代器b和e指定的范围内的字符
初始化列表	
string的搜索操作	每个搜索操作返回一个string::size_type值，表示匹配发生位置下标（是一个unsigned类型，最好用auto去保存） 若搜索失败，则返回一个名为string::npos的stastic成员 搜索是大小写敏感的
s.find(args)	查找第一次出现
s.rfind(args)	查找最后一次出现（逆向搜索，但是单词顺序不变，如"yes"逆向仍然认为是"yes"）
s.find_first_of(args)	在s中查找args中任意一个字符第一次出现的位置
s.find_last_of(args)	在s中查找args中任意一个字符最后一次出现的位置
s.find_first_not_of(args)	在s中查找第一个不在args中的字符
s.find_last_not_of(args)	在s中查找最后一个不在args中的字符
args必须是以下形式之一	
c,pos	从s中位置pos开始查找字符c ， pos默认为0
s2,pos	从s中位置pos开始查找字符串s2，pos默认为0
cp,pos	从s中位置pos开始查找指针cp指向的以空字符结尾的C风格字符串，pos默认为0
cp,pos,n	从s中位置pos开始查找指针cp指向数组的前n个字符，pos和n无默认值

compare函数	s.compare 等于返回0，小于返回负数，大于返回整数
s.compare (s2)	比较s和s2
s.compare (pos1,n1,s2)	将s中从pos1开始的n1个字符与s2进行比较
s.compare (pos1,n1,s2,po2,n2)	
s.compare (cp)	s与c风格字符串cp比较
s.compare (pos1,n1,cp)	
s.compare (pos1, n1, cp, n2)	
数值转换	数值数据与string之间的转换
to_string(val)	一组重载函数，返回数值val的string表示，val可以是任何算术类型
stoi(s,p,b)	返回s的起始子串（表示整数内容）的数值。b表示转换所用的基数，默认为10 p是size_t的指针，用来保存s中第一个非数值字符的下标，p默认为0（即不保存下标） 下面同理。
stol(s,p,b)	
stoul(s,p,b)	
stoll(s,p,b)	
stoull(s,p,b)	
stof(s,p)	
stod(s,p)	
stold(s,p)	

vector

2015年2月8日 14:51

因为引用不是对象，所以vector不能包含引用。
组成vector的元素也可是vector

定义和初始化vector：

vector <T> v	执行默认初始化
vector <T> v （v1）	注意！两个vector的类型必须相同！
vector <T> v = v1	注意！两个vector的类型必须相同！
vector <T> v (n , val)	v 包含了n个重复的元素，元素值都为val
vector <T> v (n)	v 包含了n个重复执行了值初始化的对象
vector <T> v {a,b,c...}	
vector <T> v ={a,b,c...}	

vector 操作：（string也支持下面所有操作）

v.push_back()	在尾端添加元素。（范围for循环中不能添减元素）
v.empty()	
v.size()	返回元素个数，类型：vector<T>::size_type
v[n]	返回第n个位置上的引用
v1 = v2	
v = {a,b,c...}	
v1 == v2	
v1 != v2	
< , <= ,> , >=	字典序
iter + n iter - n	向前移动n个位置 向后移动n个位置

iter1-iter2	它们之间的距离，可正可负

deque

2015年2月11日 14:44

deque是双向开口的连续性存储空间。虽说是连续性存储空间，但这种连续性只是表面上的，实际上它的内存是动态分配的，它在堆上分配了一块一块的动态储存区，每一块动态存储去本身是连续的，deque自身的机制把这一块一块的存储区虚拟地连在一起。

它首次插入一个元素，默认会动态分配512字节空间，当这512字节空间用完后，它会再动态分配自己另外的512字节空间，然后虚拟地连在一起。deque的这种设计使得它具有比vector复杂得多的架构、算法和迭代器设计。它的性能损失比之vector，是几个数量级的差别。所以说，deque要慎用

与vector相比，deque功能上的不同之处在于：

1）两端都能快速插入元素和删除元素（vector只在尾端快速进行此类操作）。

2）存取元素时，deque的内部结构会多一个间接过程，所以元素的存取和迭代器的动作会稍稍慢一些。

3）迭代器需要在不同区块间跳转，所以必须是特殊的智能型指针，非一般指针。

4）在对内存区块有所限制的系统中（例如PC系统），deque可以内含更多元素，因为它使用不止一块内存。因此deque的max_size()可能更大。

5）deque不支持对容量和内存重分配时机的控制。特别要注意的是，除了头尾两端，在任何地方插入或删除元素，都将导致指向deque元素的任何指针、引用、迭代器失效。不过，deque的内存重分配优于vector，因为其内部结构显示，deque不必在内存重分配时复制所有元素。

6）deque的内存区块不再被使用时，会被释放。deque的内存大小是可缩减的。

array

2015年2月12日 10:25

array的大小也是其类型的一部分
所以定义array时，要指定大小：

```
array<int , 42> i ;
```

由于大小是类型的一部分，所以array不支持普通容器的构造函数

与其他容器不同，一个默认构造的array是非空的：它包含了与其大小一样多的元素，元素被默认初始化

如果对array列表初始化，初始值的数目必须小于等于array的大小
但是array不支持列表赋值！

内置数组类型不能拷贝，但是对array可以拷贝，但是要求类型一样，又因为大小是array类型的一部分，所以大小也要求一样

关联容器综述

2015年2月13日 15:28

关联容器都支持普通容器的操作！
不支持顺序容器的位置相关的操作（如push_back，push_front这种）
原因是关联容器是一句关键字存储的！
关联容器的迭代器是双向的

8大类型

按关键字有序保存元素	
map	关联数组；关键字 - 值
set	集合；　　关键字即值
multimap	关键字可重复出现的map
multiset	关键字可重复出现的set
无序集合	
unordered_map	用哈希函数组织的map
unordered_set	用哈希函数组织的set
unordered_multimap	哈希组织的map，关键字可重复出现
unordered_multiset	哈希组织的set，关键字可重复出现

对于有序容器，关键字类型必须定义元素比较的方法，默认情况下，标准库使用<比较

关联容器的额外的类型别名	
key_type	此容器类型的关键字类型（使用方法:set<string>::key_type）
mapped_type	每个关键字关联的类型，只适用于map
value_type	对于set,于key_type相同 对于map，为pair<const key_type ,mapped_type>
关联容器迭代器	记住： 一个map的value_type是一个pair，我们可以改变pair的值，但是不能改变关键字成员的值 一个set中的关键字也是const，不能改变
	当使用一个迭代器遍历一个有序关联容器时，元素按关键字升序遍历
添加元素	（对于map和set插入已存在元素对其没有任何影响） 对于map插入时切记元素是一个pair，如果对于想要插入的数据没有一个现成的pair 就创建一个pair，可用的创建方法： {word,1}　//优选 make_pair(word,1) pair<string,size_t>(word,1) map<string,size_t>::value_type(word,1)
c.insert(v) c.emplace(args)	v是value_type类型的对象；args用来构造一个元素 插入元素，返回一个pair，包含一个指向具有指定关键字的元素，以及一个指示插入是否成功的bool值
c.insert(b,e) c.insert(il)	b,e表示一个c::value_type类型的范围，il是这类值的花括号列表，返回void
c.insert(p,v) c.emplace(p,args)	从迭代器p处开始搜索新元素应该存储的位置，返回一个迭代器，指向具有给定关键词的元素

删除元素	
c.erase(k)	从c中删除每个关键字为k的元素，返回一个size_type值，指出被删除元素数量
c.erase(p)	从c中删除迭代器p指向的元素（p!=c.end()），返回一个指向p之后元素的迭代器，若p指向c中的尾元素，则返回c.end()
c.erase(b,e)	返回e
访问（查找）元素	lower_bround 和 upper_bound 不适用于无序容器 下标和at操作只适用于非const
c.find(k)	返回指向第一个关键字k的元素的迭代器 若k不在，返回尾后迭代器
c.count(k)	返回关键字=k的元素的数量
c.lower_bound(k)	返回指向第一个关键字不小于k的元素的迭代器
c.upper_bound(k)	返回指向第一个关键字大于k的元素的迭代器
	PS：在multi-中 若k在容器中，则lower_bound()指向第一个具有给定关键词的元素 upper_bound()指向最后一个匹配关键字元素之后的位置(用来得到具有该关键字的元素范围) 若k不在容器中，lower和upper会返回相等的迭代器，指向一个不影响排序的关键字插入位置
c.equal_range(k)	返回一个迭代器pair，有两个成员迭代器 若k存在，第一个迭代器指向第一个匹配的元素，第二个迭代器指向最后一个匹配的元素之后的位置，来表示关键字=k的元素的范围。 若k不存在，pair的两个成员均等于c.end()

pair类型：定义在<utility>中

一个pair保存两个数据成员，是一个用来生成特定类型的模板

创建一个pair时必须提供两个类型名

pair<string,string> anon ;

两个数据成员是public的，分别命名为first,second

pair上的操作	
pair<T1,T2> p	
pair<T1,T2> p (v1,v2)	
pair<T1, T2> p = {v1, v2}	
make_pair(v1,v2)	返回一个用v1,v2初始化的pair，pair的类型从v1，v2上推断出来

pair的first成员是一个迭代器，指向给定关键字的元素。

second成员是一个bool值，表示插入成功还是已经存在于容器中。

map

2015年2月13日 15:33

简单例子

```
map<string, size_t> word_count;
string word;
while (cin >> word)      //如果word还未在 map中，下标运算符会创建一个新元素，关键字是word，值为0
  ++word_count[word];

for (const auto &w : word_count)   //注意是常量引用！
  cout <<  w.first << " occurs " << w.second << " times" << endl;
```

当从map中提取一个元素时，会得到一个pair类型的对象，pair是一个模板类型
保存两个名为first和second的（公有）数据成员（不是成员函数！！）。
map所使用的pair用firtst成员保存关键字，second成员保存对应值！

map和unordered_map提供下标运算符和at函数 （set不支持下标）
但multi-版本的map没有下标操作，因为无法一一映射
map下标运算符接受一个索引（即关键字），获取与关键字相关联的值。
返回类型是：mapped_type
解引用一个map迭代器时返回类型是：value_type
如果关键字不在map中，会为它创建一个元素并插入到map中，关联值将进行值初始化！
故可以用下标来创建新的元素。例如：

```
map <string,size_t> word_count;
word_count["Anna"]=1;  //初始化关键字Anna对应的值为1！
```

也因为下标为给map添加新元素，所以要小心使用

也可以用at函数访问值：
c.at(k) ： 访问关键字为k的元素，若k不在c中，抛出out_of_range异常

如果一个multimap或multiset中有多个元素具有给定关键字，则这些元素在容器中会相邻存储
因此，假如想要打印一个作者的所有著作，可以利用连续存储这一特性：

```
multimap<string, string> authors;
string search_item("Alain de Botton"); // author we'll look for
auto entries = authors.count(search_item); // number of elements
auto iter = authors.find(search_item); // first entry for this author

// loop through the number of entries there are for this author
while(entries) {
  cout << iter->second << endl; // print each title
  ++iter;     // advance to the next title
  --entries;  // keep track of how many we've printed
}
```

set

2015年2月13日　　15:33

无序容器

2015年2月22日 15:28

无序容器不是以比较运算符来组织元素的，而是用一个哈希函数，和关键字类型的==运算符

无序容器提供了与有序容器相同的操作。
有4中类型，都是有序容器前加 unordered_

无序容器在存储上组织为一组桶，用一个哈希函数把元素映射到桶
为了访问一个元素，容器首先计算元素的哈希值，指出应该搜索哪个桶，容器将具有一个特定哈希值的所有元素都保存在相同的桶中，若multi-版本，则所有具有相同关键字的元素也都会在一个桶中。
但是若一个桶中保存了很多元素，那么查找一个特定元素就需要大量比较操作

无序容器的管理操作	
桶接口	
c.bucket_count()	正在使用的桶数量
c.max_bucket_count()	容器能容纳的最大的桶数量
c.bucket_size(n)	第n个桶中有多少元素
c.bucket(k)	关键字为k的元素在哪个桶中
桶迭代	
……	

搜索

2015年2月8日 15:32

二分搜索：（前提：有序序列）

```
//利用迭代器：
auto beg = text.begin() , end = text.end() ;
auto mid = text.begin() + (end - beg) / 2;
while (mid != end && *mid != sought)
{
    if (sought < *mid)
        end = mid ;
    else
        beg = mid + 1;
    mid = beg + (end - beg) / 2;
}
```

泛型算法综述

2015年2月12日 15:38

大多数算法定义在<algorithm>

一般泛型算法并不直接操作容器，而是遍历两个迭代器的范围进行操作
所以算法只会运行在迭代器之上，不会改变底层容器大小，只可能改变容器中所保存元素的值
也可能在容器中移动元素

对于只读算法，用cbegin（）,cend（）版本迭代器

搜索

2015年2月12日 15:53

find(b,e,val);	返回指向第一个等于给定值的元素的迭代器 如果无匹配元素，则find返回第二个参数来表示搜索失败 PS：由于指针就如同内置数组上的迭代器一样，所用可用find来在数组中查找值 如：int *result = find(begin(ia),end(ia),val) ;
find_if （ b, e, cmp ）	find_if算法对输入序列的每个元素调用给定的这个谓词，返回第一个使谓词返回非0的元素，若不存在这样的元素，返回尾迭代器
find_if_not （ b, e, cmp ）	返回第一个使谓词返回为0的元素
count(b,e,val)	返回一个计数器，指出val出现了多少次
count_if(b,e,cmp)	统计有多少个使cmp为非0元素
all_of(b,e,cmp)	返回一个bool值，是否对所有元素都成功
any_of(b,e,cmp)	返回一个bool值，是否对任一个元素都成功
none_of(b,e,cmp)	返回一个bool值，是否对所有元素都不成功
search(b1,e1,b2,e2) search(b1,e1,b2,e2,cmp)	返回第二个输入范围（子序列）在第一个输入范围中第一次出现的位置，若未找到，返回e1
find_first_of(b1,e1,b2,e2) find_first_of(b1,e1,b2,e2,cmp)	返回第二个输入范围内任意元素在第一个范围中首次出现的位置的迭代器，未找到返回e1
find_end(b1,e1,b2,e2) find_end(b1,e1,b2,e2， cmp)	返回第二个输入范围在第一个输入范围内，最后一次出现的位置。未找到返回e1

二分搜索算法	每个算法两个版本，第一个版本默认用<来检测元素，第二个版本使用给定的比较操作cmp
lower_bound(b,e,val) lower_bound(b,e,val,cmp)	返回一个迭代器，表示第一个小于等于val的元素。不存在返回e
upper_bound(b,e,val) upper_bound(b,e,val,cmp)	返回一个迭代器，表示第一个大于val的元素。不存在返回e
equal_range(b,e,val) equal_range(b,e,val,cmp)	返回一个pair，其first成员是lower_bound返回的迭代器，second成员是upper_bound返回的迭代器
binary_search(b, e, val） binary_search(b, e, val,cmp)	返回一个bool，指出是否包含=val的元素

其他

2015年2月12日 15:58

count（b, e, val）	返回给定val在序列中出现的次数

计算

2015年2月13日 11:31

accumulate（b,e,n）	求b,e返回内元素之和，返回和，n表示和的初始值（对string操作时，相当于连接字符串）
accumulate（b,e,n,cmp）	用cmp定义的算术运算符来计算
inner_product(b1,e1,b2,n)	求两个序列内积，即对应元素乘积之和

比较

2015年2月13日 11:35

equal（b,e,b2）	比较两个序列是否保存相同值， 如果相等返回true，如果不相等返回false
mismatch(b1,e1,b2)	比较两个序列中元素，返回一个pair，表示两个序列中第一个不匹配的元素 若都匹配，返回的pair的第一个迭代器为e1,第二个迭代器为b2中偏移量等于第一个序列长度的位置
最大值最小值	max与min一致
min(val1,val2)	
min(val1,val2,cmp)	
min(list)	
min(list,cmp)	
minmax(val1,val2)	返回一个pair，first成员是最小值，second是最大值
minmax(val1,val2,cmp)	
minmax(list)	
minmax(list,cmp)	
min_element(b,e)	_element版本返回各自前函数所对应元素的迭代器
min_element(b,e,cmp)	
max_element(b,e)	
max_element(b,e,cmp)	
minmax_element(b, e)	
minmax_element(b, e,cmp)	

添加与删除元素

2015年2月13日 11:38

添加元素	
fill(b, e, val)	将val写入序列中的每个元素
fill_n(b,n,val)	将val分别赋值给b及之后的n个元素 （切记，b,e内要有足够的空间能够存放）
copy(b, e, b1) copy_n(b, n, dest)	把b,e范围内的内容拷贝到b1开始的容器中。返回最后一个拷贝元素之后的位置的迭代器
copy_if(b,e,dest,cmp)	将输入范围内满足cmp谓词函数的数拷贝到dest中
move(b, e, dest)	将序列内元素移动到dest中
transform(b,e,dest,cmp)	将b,e中满足谓词函数的元素写到dest中
replace(b, e, key, val) replace_if(b,e,cmp,val)	将序列所有=key的元素替换为val
replace_copy(b,e,dest,key,val)	将序列所有=key的元素替换为val后保存到dest为首的序列中
replace_copy_if(b,e,dest,cmp,val)	将序列所有满足cmp的元素替换为val后保存到dest为首的序列中
merge(b1,e1,b2,e2,dest) merge(b1, e1, b2, e2, dest, cmp)	两个序列必须都是有序的，将合并后的序列写入dest, 第一个版本用<运算符比较，第二个版本用cmp自定义操作比较

保证算法有足够的元素空间来输出数据的方法是使用插入迭代器：

back_inserter 是定义在头文件 iterator 中的一个函数

接受一个指向容器的引用，返回一个与该容器绑定的插入迭代器

当通过此迭代器赋值时，赋值运算符会调用push_back ，将一个具有给定值的元素添加到容器中

如fill_n(back_inserter(vec),10,0) ；添加10个元素到vec中，vec一开始可以为空

“删除”元素	
remove(b,e,val)	从序列中“删除”=val的元素，采用的方法是用保留元素覆盖要删除的元素 返回最后一个删除元素的尾后位置
remove_if(b,e,cmp)	
remove_copy(b,e,dest,val)	
remove_copy_if(b,e,dest,cmp)	

排列

2015年2月13日 12:03

sort(b, e)	按<的顺序重排
sort(b,e,cmp)	
stable_sort(b,e)	
stable_sort(b,e,cmp)	
is_sorted(b, e)	返回一个bool，是否有序
is_sorted(b,e,cmp)	
is_sorted_until(b,e)	在输入序列中查找最长初始有序子序列，并返回子序列的尾后迭代器
is_sorted_until(b,e,cmp)	
unique(b,e) unique(b, e, cmp)	在使用unique之前，要对序列进行排序，确保相同元素相邻出现 unique算法将相邻重复项消除，使得不重复的元素出现在容器的开始部分， 并返回一个指向不重复范围之后第一个元素的迭代器（算法并不能删除重复元素，需要自己手动删除） 这时，可以手动调用C.erase(end_unique,C.end()) ;来删除所有重复元素
unique_copy(b,e,dest)	
unique_copy_if(b,e,dest,cmp)	

list与forward_list的特定容器算法

2015年2月13日 15:05

对于链表类型，使用通用算法代价太高，应该优先使用成员函数版本的算法！
链表版本的算法通过改变元素间链接而不是真正交换它们的值，所有性能更好

list与forward_list成员函数版本的算法	都返回void
lst.merge(lst2) lst.merge(lst2,cmp)	将来自lst2的元素合并入lst，lst和lst2都必须是有序的。 且合并后元素将从lst2中删除，lst2变空。 第一个版本使用 第二个版本自定义
lst.remove(val) lst.remove_if(pred)	调用erase删除掉给定值相等或令pred为真的每个元素
lst.reverse()	反转lst中元素的顺序
lst.sort() lst.sort(cmp)	
lst.unique() lst.unique(pred)	调用erase删除相同元素的连续拷贝 第一个版本用== 第二个版本用给定的二元谓词
lst.splice(args) flst.splice_affer(args)	见下
args代表：	
(p,ls2)	p是一个指向lst中元素的迭代器，或者一个指向flst首前位置的迭代器 函数将lst2的所有元素移动到lst中的p之前的位置或者flst中p之后的位置 将元素从lst2中删除
(p,lst2,p2)	p2是一个指向lst2中位置的有效迭代器， 将p2指向的元素移动到lst中，或将p2之后的元素移动到flst中
(p,lst2,b,e)	将给定范围中的元素从lst2移动到lst或flst

交换

2015年2月22日 20:33

iter_swap(iter1, iter2)	交换两者所表示的元素，返回void
swap_ranges(b1,e1,b2)	将输入范围内元素与b2开头的范围内的元素进行交换，两个范围不能有重叠，返回递增后的b2，指向最后一个交换元素之后的位置