# C++记录

## 1添加algorithm编辑

### 说明

算分头文件 #include <algorithm>提供大量的算法

VIS2013智能搜索完后很麻烦要按，上下来选择，其实可以直接按tab就可以选取光标显示的那位置

在C里面，输出字符串时，如果想输出一个换行，那就要加上'\n'这个标志，类似的，输出一个TAB，就加上' \t'，

//解释一些东西.\ ..\ ..\..\

.\表示当前目录文件所在目录之下的目录

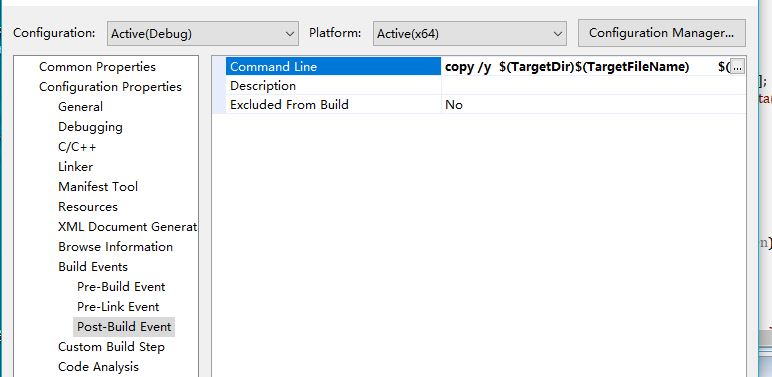
..\表示当前目录文件所在目录向上一级目录下的目录

..\..\目录所在向上二级目录之下的目录

./当前目录

../父级目录

/是根目录



copy /y $(TargetDir)$(TargetFileName) $(ProjectDir)..\..\..\Robot\execute\$(TargetFileName)

//上面是拷贝路径

void GameDesk::CleanHandsData()

{

std::map<int ,CardVector\*> ::iterator it=m\_mapCardVector.begin();

for(;it!=m\_mapCardVector.end();it++)

{

if(it->second)

{

delete it->second;

it->second=NULL;

}

}

m\_mapCardVector.clear();

}

问：类的前置声明

//类的前置声明 指针是用到对象不能用到

问：容器中删除某段区域

vec.erase(vec.begin()+i,vec.end()+j); //删除区间[i,j-1];区间从0开始

问：产找的某个内容

%AppData%

问：lua中理解二维数组一维数组的差别

Lua 兩個的差别-------------- infoNumber【1】

["infoNumber"] = {

"开口1番",

"赖杠5番",

},

["infoNumber"] = {

{"开口1番"},

{"赖杠5番"},

},

infoNumber【1】【1】

问：

（对象）CardVector m\_DanGangVector; //单杠牌列表

for( CardVectorIterator it = m\_DanGangVector.begin(); it != m\_DanGangVector.end(); ++it )

{

if( \*it == DanGangCard )

{

return TRUE;

}

}

（指针 ）CardVector\* m\_pDanGangVector; //癞子皮列表

for( CardVectorIterator it = m\_pDanGangVector->begin(); it != m\_pDanGangVector->end(); ++it )

{

if( m\_DanGangCardVector[i] == \*it )

{

w++;

}

-----------------------------------------------------------

CardVector OutCard;//后来有被引用赋值

if (std::find(OutCard.begin(), OutCard.end(), pMsg->byCard) == OutCard.end())

//迭代器的等于

for (CardVectorIterator it = m\_HandCardVector.begin(); it != m\_HandCardVector.end(); ++it)

{

if (HandCard == \*it)

{

m\_HandCardVector.erase(it);

break;

}

}

////////迭代器在的使用

typedef std::vector<Card> CardVector;

typedef CardVector::iterator CardVectorIterator;

CardVector DanGangVector;

for( CardVectorIterator it = m\_DanGangVector.begin(); it != m\_DanGangVector.end(); ++ it )

{

}

// void CardManager::DelPiZiAtAllCard(Card pizi)

// {

// std::vector <Card>:: iterator iter=m\_AllCard.begin();

// for ( ;iter!=m\_AllCard.end();)

// {

// if(\*iter==pizi)

// {

// iter=m\_AllCard.erase(iter);

// break;

// }

// else

// {

// iter++;

// }

// }

// }

//复制

CardVector chengBaoUserHandCard( pChengBaoUser->GetHandCardVector().begin() , pChengBaoUser->GetHandCardVector().end() );

//////子类父类之间奇妙的错

CBasehu HuAction//申明基类

//少了括号我就找不都成员下面就对了

(CActionHuPai\*)m\_HuAction->setDesk(this);（错）

( (CActionHuPai\*)m\_HuAction)->setDesk(this);（对了）

CardVector vtemp=vhand;

int arryYJz[13]={1,1,1,1,1,1,1,1,1}

for(int i=0;i<13;i++)

{

CardVectorIterator it=std:find(vtemp.begin(),vtemp.end(),arryYJz[i])//迭代器赖

If(\*it!=vtemp.end)

Vtemp.erase(it)

}

random\_shuffle()有两个参数，第一个参数是指向序列首元素的迭代器，第二个参数则指向序列最后一个元素的下一个位置

/洗牌

void CardManager::Shuffle()

{

std::random\_shuffle(m\_AllCard.begin(), m\_AllCard.begin()+m\_AllCard.size());

random\_shuffle()是个完全通用的算法-适用于内建的数据类型和用户自定义类型。下面的例子创建了一个有7个字符串对象的向量，它包含一周的天数并使用random\_shuffle()打乱他们的排列顺序：

#include <string>  
#include <vector>  
#include <algorithm>  
#include <iostream>  
using namespace std;  
int main()  
{  
　vector<string> vs;  
　vs.push\_back(string ("Sunday"));  
　vs.push\_back (string ("Monday"));  
　...

如何使用random\_shuffle()处理内置数组

在使用容器代替内置数组时，你不要有什么负担。所有STL算法不仅适用于容器，也适用于序列。因此，你也能将random\_shuffle()算法应用于内置数组。只是要注意random\_shuffle()的第二个参数要指向数组上界的下一个元素位置：

char carr[4] = {''a'', ''b'', ''c'', ''d''};  
/\*carr+4 指向数组上界的下一个元素位置\*/  
random\_shuffle(carr, carr+4);　  
for (int i = 0; i < 4; i++)  
　cout<<carr[i]; /\* 显示被打乱顺序的元素 \*/

///

三、顺序容器的赋值操作

|  |  |
| --- | --- |
| c1 = c2 | 删除容器 c1 的所有元素，然后将 c2 的元素复制给 c1。c1 和c2 的类型（包括容器类型和元素类型）必须相同 |
| c1.swap(c2) | 交换内容：调用完该函数后，c1 中存放的是 c2 原来的元素，c2 中存放的则是 c1 原来的元素。c1 和 c2 的类型必须相同。**该函数的执行速度通常要比将 c2 复制到 c1 的操作快** |
| c.assign(b,e) | 重新设置 c 的元素：将迭代器 b 和 e 标记的范围内所有的元素复制到 c 中。b 和 e 必须不是指向 c 中元素的迭代器 |
| c.assign(n,t) | 将容器 c 重新设置为存储 n 个值为 t 的元素 |

四、测试assign

assign 操作首先删除容器中所有的元素，然后将其参数所指定的新元素插入到该容器中。与复制容器元素的构造函数一样，如果两个容器类型相

同，其元素类型也相同，就可以使用赋值操作符（=）将一个容器赋值给另一个容器。如果在不同（或相同）类型的容器内，元素类型不相同但是

相互兼容，则其赋值运算必须使用 assign 函数。例如，可通过 assign 操作实现将 vector 容器中一段 char\* 类型的元素赋给 string 类型 list 容器。

由于 assign 操作首先删除容器中原来存储的所有元素，因此，传递给 assign 函数的迭代器不能指向调用该函数的容器内的元素。

**[cpp]** [view plain](http://blog.csdn.net/amber2012/article/details/9199711) [copy](http://blog.csdn.net/amber2012/article/details/9199711)

1. **void** test\_assign()
2. {
3. std::list<std::string> ls;
4. ls.push\_back("aaa");
5. ls.push\_back("bbb");
6. ls.push\_back("ccc");
8. std::deque<std::string> de;
9. de.assign(ls.begin(),ls.end());//把list assign到deque
10. print\_deque(de);//aaa\_bbb\_ccc\_
12. std::vector<std::string> vec;
13. vec.push\_back("111");
14. vec.push\_back("222");
15. vec.push\_back("333");
17. de.assign(vec.begin(),vec.begin()+1);
18. print\_deque(de);//111\_,左闭右开

21. de.assign(5, "000");
22. print\_deque(de);//000\_000\_000\_000\_000\_
23. }

代码1．什么是递归函数(recursive function)

　　递归函数即自调用函数，在函数体内部直接或间接地自己调用自己，即函数的嵌套调用是函数本身。

　　例如，下面的程序为求n!：

　　　　long fact(int n)

　　　　{

　　　　　if(n==1)

　　　　　return 1；

　　　　　return fact(n-1)\*n； ／／出现函数自调用

　　　　}

2．函数调用机制的说明

　　任何函数之间不能嵌套定义， 调用函数与被调用函数之间相互独立(彼此可以调用)。 发生函数调用时，被调函数中保护了调用函数的运行环境和返回地址，使得调用函数的状态可以在被调函数运行返回后完全恢复，而且该状态与被调函数无关。

　　被调函数运行的代码虽是同一个函数的代码体，但由于调用点，调用时状态， 返回点的不同，可以看作是函数的一个副本，与调用函数的代码无关，所以函数的代码是独立的。被调函数运行的栈空间独立于调用函数的栈空间，所以与调用函数之间的数据也是无关的。函数之间靠参数传递和返回值来联系，函数看作为黑盒。

　　这种机制决定了C／C++允许函数递归调用。

3．递归调用的形式

　　递归调用有直接递归调用和间接递归调用两种形式。

　　直接递归即在函数中出现调用函数本身。

　　例如，下面的代码求斐波那契数列第n项。 斐波那契数列的第一和第二项是1，后面每一项是前二项之和，即1，1，2，3，5，8，13,...。 代码中采用直接递归调用：

　　　　long fib(int x)

　　　　{

　　　　　if(x>2)

　　　　　　return(fib(x-1)+fib(x-2))； //直接递归

　　　　　else

　　　　　　return 1；

　　　　}

　　间接递归调用是指函数中调用了其他函数，而该其他函数却又调用了本函数。例如，下面的代码定义两个函数，它们构成了间接递归调用：

　　　　int fnl(int a)

　　　　{

　　　　　int b；

　　　　　b=fn2(a+1)； //间接递归

　　　　　　　　　　　//...

　　　　}

　　　　int fn2(int s)

　　　　{

　　　　　int c；

　　　　　c=fnl(s-1); //间接递归

　　　　　　　　　　　//...

　　　　}

　　上例中，fn1()函数调用了fn2()函数，而fn2()函数又调用了fn1()函数。

4．递归的条件

　　(1)须有完成函数任务的语句。

　　例如，下面的代码定义了一个递归函数：

　　　　#include

　　　　void count(int val) //递归函数可以没有返回值

　　　　{ if(val>1)

　　　　　　　count(val-1)； 、

　　　　　cout<<"ok:" <<<="" 此语句完成函数任务="" />

　　该函数的任务是在输出设备上显示"ok：整数值”。

　　(2)—个确定是否能避免递归调用的测试

　　例如，上例的代码中，语句"if(val>1)"便是—个测试， 如果不满足条件，就不进行递归调用。

　　(3)一个递归调用语句。

该递归调用语句的参数应该逐渐逼近不满足条件，以至最后断绝递归。

　　例如，上面的代码中，语句“if(val>1)” 便是一个递归调用，参数在渐渐变小，这种发展趋势能使测试"if(val>1)”最终不满足。

　　(4)先测试，后递归调用。

在递归函数定义中，必须先测试，后递归调用。也就是说，递归调用是有条件的，满足了条件后，才可以递归。

　　例如，下面的代码无条件调用函数自己，造成无限制递归，终将使栈空间溢出：

　　　　#include

　　　　void count(int val)

　　　　{

　　　　　count(val-1)； //无限制递归

　　　　　if(val>1) //该语句无法到达

　　　　　　cout <<"ok： " <<　　　　}

5．消去递归

　　大多数递归函数都能用非递归函数来代替。例如，下面的代码求两个整数a，b的最大公约数，用递归和非递归函数分别定义之：

　　　　long gcdt(int a,int b) //递归版

　　　　{

　　　　　if(a%b==0)

　　　　　　return b；

　　　　　return gcdl(b,a%b)；

　　　　}

　　　　long gcd2(int a,int b) //非递归版

　　　　{

　　　　　　int temp；

　　　　　　while(b!=0)

　　　　　　{

　　　　　　　temp=a%b；

　　　　　　　a=b；

　　　　　　　b=temp；

　　　　　　}

　　　　　　return a；

　　　　}

　　思考：将求n!的递归函数非递归化。

6．递归的评价

　　递归的目的是简化程序设计，使程序易读。

　　但递归增加了系统开销。 时间上， 执行调用与返回的额外工作要占用CPU时间。空间上，随着每递归一次，栈内存就多占用一截。

　　相应的非递归函数虽然效率高，但却比较难编程，而且相对来说可读性差。

　　现代程序设计的目标主要是可读性好。随着计算机硬件性能的不断提高，程序在更多的场合优先考虑可读而不是高效，所以，鼓励用递归函数实现程序思想。

wKaiKou = 2 << (CheckKaiMenCount(pPlayer->GetFuZi()) - 1 );//这个挺好的计算^

//

声明升级版本不能用 DECLSPEC\_DEPRECATED

VOID WriteLog(IN DWORD dwLevel,IN LPCSTR lpFmt, IN ...);

问：在做牌中声明了一个二维数组static Card byTempCard[20][72] =

{。。。 return byTempCard[1]};

//cmd copy/? 查询

xcopy /E /Y E:\git\HaiNanGames\games\hnmj\_ziji E:\git\HaiNanGames\games\hnmj

rem 注释

### STL之set:set的基本用法

### 此篇文章来源

https://blog.csdn.net/ac\_hexin/article/details/52042562

#### Set容器和其他容器差不多，无非就是相同的值不存在，存进去自动排序好了 //第二次插入5，重复元素，不会插入

构造set集合主要的目的是为了快速检索，在使用set前，需要在程序头文件中包含声明

#include<set>

##### 1 创建set对象时，需要指定类型，这一点和其他容器一样，同时主要一下它是在std 如果没用using namespace std

1 正向遍历 从小到大

set<**int**> s;

1. s.insert(5); //第一次插入5，可以插入
2. s.insert(1);
3. s.insert(6);
4. s.insert(3);
5. s.insert(5); //第二次插入5，重复元素，不会插入
6. set<**int**>::iterator it; //定义前向迭代器
7. //中序遍历集合中的所有元素
8. **for**(it = s.begin(); it != s.end(); it++)
9. {
10. cout << \*it << " ";
11. }
12. cout << endl;
13. **return** 0;
14. }
15. //运行结果：1 3 5 6

这个显示清除5 ，并且排序了顺序

反向遍历 从大到小

1. set<**int**>::reverse\_iterator rit; //定义反向迭代器
2. //反向遍历集合中的所有元素
3. **for**(rit = s.rbegin(); rit != s.rend(); rit++)
4. {
5. cout << \*rit << " ";
6. }
7. cout << endl;
8. **return** 0;
9. }
10. //运行结果：6 5 3 1