1. 认识六大部件
2. 复杂度
3. “前闭后开”区间
4. range-base for statement

一、认识六大部件

#include <iostream>

#include <vector>

#include <algorithm>

#include <functional>

int main()

{

int ia[6] = {27,217,12,47,109,83};

std::vector<int,std::allocator<int> > vi(ia,ia+6);

std::cout<<std::count\_if(vi.begin(),vi.end(),std::not1(std::bind2nd(std::less<int>(),40)));

return 0;

}

1. container=vector=容器
2. allocator=allocator=分配器
3. algorithm=count\_if=算法
4. iterator=vi.begin(),vi.end()=迭代器
5. function adapter = not1，bind2nd = 反函数分配器
6. function object = less =
7. 复杂度
8. “前闭后开”区间

Container<T> c;

...

Container<T>::iterator ite = c.begin();

for(;ite!=c.end();++ite)

...

//c.end()作为一个泛型指针，如果你解引用它，是没有意义的

c.begin()就不同了，解引用它，就是它的容器的值

至于为什么这么做呢，是因为这样做我们很方便对容器进行迭代，

你想，如果c.end()是一个有意义的东西，那么问题来了，for(;ite!=c.end();++ite)

中的跳出迭代条件该是什么？

如果仍然是ite!=c.end();那么，c.end()解引用的值不就丢掉了没有利用价值了？

有的人说，我这样遍历迭代不就利用上了吗？

Container<T>::iterator ite = c.begin();

while(1)

{

std::cout<<\*(iter)<<endl;

if(iter==c.end())

{

break;

}

else

{

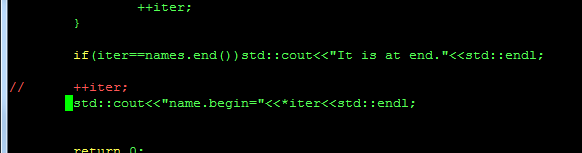
iter++;

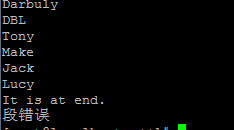
}

}

这样做不行的原因：

1. 迭代太过麻烦，可支持的迭代方式太单一，所以抛弃掉
2. 在尝试把c.end()给解引用之后,直接段错误，直接内存溢出





1. range-base for statement

for(decl:coll)

{

statement

}

for (auto elem:vec)

{

std::cout<<elem<<std::endl;

}

for(auto& elem:vec)

{

elem \*= 3;

}

//这是C++2.0 就是在2011年版本推出的一种快速迭代的方法