模板与STL初步

一、命名空间

A. 定义

为了避免在大规模程序的设计中，以及在程序员使用各种各样的C++库时，标识符的命名发生冲突，标准C++引入了关键字namespace（命名空间），可以更好地控制标识符的作用域。
标准C++库（不包括标准C库）中所包含的所有内容（包括常量、变量、结构、类和函数等）都被定义在命名空间std（standard标准）中。

例如：cout、cin、vector、set、map

也就是说，namespace本身是一种关键字，这就是所谓的命名空间，而std仅仅是已经被定义的一个命名空间。

B. 例子

B. a自定义命名空间

```
namespace A {
        int x, y;
    }
```

B.b 使用命名空间

```
A::x = 3;
A::y = 6;
//对人为定义的命名空间A内的x，y进行赋值
```

C. using申明

使用using声明简化命名空间使用

```
  //使用整个命名空间：所有成员都直接可用
    using namespace A;
    x = 3; y = 6;
```

```
//使用部分成员：所选成员可直接使用
    using A::x;
    x = 3; A::y = 6;
```

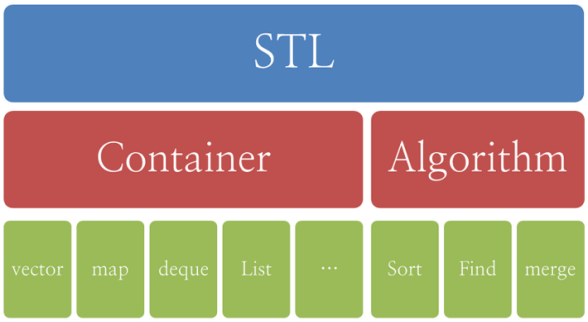
任何情况下，都不应出现命名冲突

二、 STL初步

A. 定义

标准模板库（英文：Standard Template Library，缩写：STL），是一个高效的C++软件库，它被容纳于C++ 标准程序库C++ Standard Library中。其中包含4个组件，分别为**算法、容器、函数、迭代器**。这些都是基于模板编写而成。

关键理念：将“在数据上执行的操作”与“要执行操作的数据”分离。



STL的命名空间是std，一般使用std::name来使用STL的函数或对象，也可以使用using namespace std来引入STL的命名空间

```
 using std::string
```

（不推荐在大型工程中使用，容易污染命名空间）

关于STL的文档和例子可以在以下网址查询 http://www.cplusplus.com/

多写多查多用，是学习STL库的最好方法

B. STL容器

容器是包含、放置数据的工具。通常为数据结构。包括：
1.简单容器（simple container）　2.序列容器（sequence container）　3. 关系容器（associative container）

pair容器

```
//最简单的容器，由两个单独数据组成。
template<class T1, class T2> struct pair{
    T1 first;
    T2 second;
//若干其它函数
};
//通过first、second两个成员变量获取数据。
std::pair<int, int> t;
t.first = 4; t.second = 5;
```

```
//创建：使用函数make_pair。
auto t = std::make_pair(“abc”, 7.8);
//自动推导成员类型。
//此处推导出的t的类型就是Pair类型
```

```
//支持小于、等于等比较运算符。
//先比较first，后比较second。
//要求成员类型支持比较(实现比较运算符重载)。
std::make_pair(1, 4) < std::make_pair(2, 3);
std::make_pair(1, 4) > std::make_pair(1, 2);
```

```
#include <string>
int main(){
  std::pair<std::string, double> p1("Alice", 90.5);//手工指定类型初始化
  std::pair<std::string, double> p2;
  p2.first = "Bob";
  p2.second = 85.0;//先定义，再赋值
  auto p3 = std::make_pair("David", "95.0");//自动推导类型初始化
  return 0;
}
```

tuple容器

C++11新增，**pair的扩展**，由若干成员组成的元组类型。

```
 template< class Types.... > class tuple;
```

通过std::get函数获取数据。

```
 v0 = std::get<0>(tuple1);
 v1 = std::get<1>(tuple2);
```

其下标需要在编译时确定：不能设定运行时可变的长度，不能当做动态数组.

```
int i = 0;
v = std::get<i>(tuple);
//编译错误
```

```
#include<iostream>
using std::tuple;
using std::string;
tuple<int,double,string> t3 = {1, 2.0, "3"};
std::get<0>(t3) = 4;
std::cout << get<1>(t3) <<std::endl;
int i; double d; string s;
std::tie(i, d, s) = t3;
std::cout << i << " " << d << " " << s <<std::endl;
//这串代码存在非常多的错误，首先，没有#include<tuple>
//第二，第6行也需要std::get<0>(t3)
//第三，最重要的，这玩意儿没有main函数
```

修改后的代码：

```
#include<iostream>
#include<tuple>
//using std::tuple;单独使用第三行没有用，必须要有第二行，且只需要有第二行
using std::string;
int main(){
std::tuple<int,double,string> t3 = {1, 2.0, "3"};
std::get<0>(t3) = 4;
std::cout << std::get<1>(t3) <<std::endl;
std::cout << std::get<0>(t3) <<std::endl;
int i; double d; string s;
```

```
std::tie(i, d, s) = t3;
i=10;
std::cout << i <<std::endl;
std::cout << std::get<0>(t3) <<std::endl;
}

output:
2
4
10
4
```

从上述代码可以看出，实际上get函数返回的是tuple的某个元素的左值引用，可以赋值，也可以被输出。而tie函数实际上没有返回值，仅仅是完成了对应的简单赋值，并不是生成左值引用，改变i并不能改变tuple。

tuple的意义，用于函数多返回值传递：

```
#include <tuple>
#include<iostream>
std::tuple<int, double> f(int x){
    return std::make_tuple(x, double(x)/2);
}
int main() {
    int xval;
    double half_x;
    std::tie(xval, half_x) = f(7);
    std::cout<<xval<<std::endl;
    return 0;
}

output:
7
```

作为tuple的特例，pair可用于两个返回值的传递。除此之外，pair在map中大量使用。

Vector容器

会自动扩展容量的数组，以循序(Sequential)的方式维护变量集合。

```
template<class T, class Allocator = std::allocator<T>>
   class vector;
```

STL中最基本的序列容器，提供有效、安全的数组以替代C语言中原生数组。
允许直接以下标访问。（高速）

```
include <vector>;
using namespace std;
int main(){
    std::vector<int> x;//创建
    x.size();//获取当前数组长度
    x.clear();//清空;
    x.push_back(1);//在末尾添加1
    x.pop_back();//删除
    x.insert(x.begin()+1, 5);//在中间添加（低速）
    x.erase(x.begin()+1);//在中间删除（低速）
```

迭代器

一种检查容器内元素并遍历元素的数据类型。

提供一种方法顺序访问一个聚合对象中各个元素, 而又不需暴露该对象的内部表示。
为遍历不同的聚合结构（需拥有相同的基类）提供一个统一的接口。
对遍历元素使用类似指针。

以vector为例

```
vector<int>::iterator iter;
```

定义了一个名为iter的变量，它的数据类型是由vector定义的iterator类型。

```
x.begin();
x.end();
```

begin返回vector中初始位置的迭代器（指针），即x[0]。

end函数返回x[x.size()]，注意到x.size()对应的不是最后一个元素，而是最后一个元素之后。

故而结合begin()和end()能够实现遍历所有元素，即遍历一个 $[0, size())$ 左闭右开区间。

```
++iter;
--iter;
iter+=n;
iter-=n;//迭代器移动
int dist = iter1 - iter2;//返回元素位置差;
*iter = 5;//*是解引用运算符，返回的是左值引用
```

利用iterator遍历vector

因为end()不是最后一个元素，而是最后一个元素之后的位置，该代码可以遍历vec中所有元素

```
for(vector<int>::iterator it = vec.begin();
    it != vec.end(); ++it)  //use *it
  //也可以用auto简化代码，编译器自动推导it是vec的iterator
  for(auto it = vec.begin(); it != vec.end(); ++it)
                        //use *it
  //还可以直接这样遍历
  for (auto x : vec)
```

补充：反向遍历器迭代整个vector

```
for (auto zex = vec.rbegin(); zex != vec.rend(); zex++)
{std::cout << *zex << std::endl;}
```

这个语法利用rbegin与rend实现了从后往前遍历整个vec函数，注意对于zex仍然是采用++运算符。

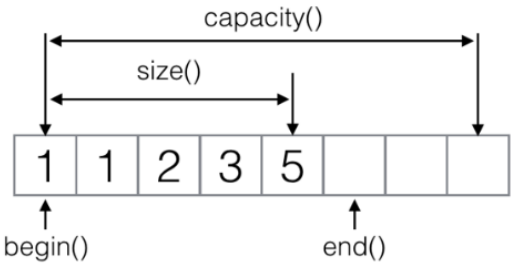
(2).迭代器失效

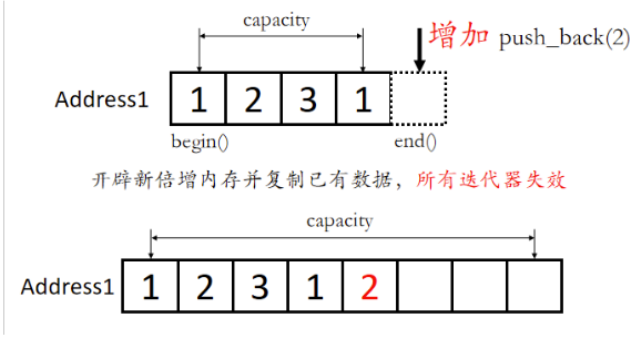
当迭代器不再指向本应指向的元素时，称此迭代器失效。这并不是指指针本身出了问题，而是指向不再正确。

调用insert/erase后，所**修改位置**的迭代器、以及**该位置之后**的所有迭代器失效。（原先的内存空间存储的元素被改变）
调用push_back等修改vector大小的方法时，如果vector恰好达到了capacity上限，编译器会给voctor扩容。扩容后整个vector会发生空间整体迁移，但是迭代器仍然指向之前的内存，从而会使**所有**迭代器失效

具体而言：

> vector是会自动扩展容量的数组，除了size，另保存capacity：最大容量限制。如果size达到了capacity，则另申请一片capacity*2的空间，并整体迁移vector内容。这一过程时间复杂度为均摊O(1)，并且整体迁移过程使所有迭代器失效。
>
> 每次以两倍为单位扩容实际上是满足了堆优化，是一个很优的扩容选择，将在数据结构中具体学习。





因为扩容导致的迭代器失效

在遍历的时候增加元素，可能会导致迭代器失效

```
#include <iostream>
#include <vector>
using namespace std;
int main(){
vector<int> vec = {1,2,3,4,5};
   for(auto it = vec.begin(); it != vec.end(); ++it)
      vec.push_back(*it);
  ////Error，这个循环不会终止，一直进行，直到迭代器失效出现error
   return 0;
}
//这段代码会使vec变成：123451234512345……直到vec达到capacity上限，此时it失效，出现error
```

其实这种情况蛮好理解的，因为每一次循环都会判定it是否等于$vec.end()$，而$vec.end()$不断在变化。故而导致循环不停止，直到迭代器因为容量的迁移而失效。

因为删除导致的迭代器失效

```
#include <iostream>
using namespace std;
#include<vector>

int main() {
vector<int> vec = {1,2,3,4,5};
for(auto it=vec.begin();it!=vec.end();it++)
    {
        cout<<*it<<endl;
    }
    cout<<"====================="<<endl;
   auto first = vec.begin();
   auto second = vec.begin() + 1;
   auto find = vec.begin() + 6;
   vec.erase(second);
   auto ret = vec.erase(second);
   cout<<*second<<endl;
   cout<<*ret<<endl;
   cout<<"my size is "<<vec.size()<<endl;
   cout<<"how could I find "<<*find<<endl;
    for(auto it=vec.begin();it!=vec.end();it++)
```

```
    {
        cout<<*it<<endl;
    }
    return 0;
}
output;
1
2
3
4
5
=====================
4
4
my size is 3
how could I find 4113
1
4
5
```

从上述代码可以见得：

1. erase函数传入的参数和返回值类型都是iterator，但是你可以选择不用新的iterator接受返回值，比如第15行。
2. erase返回的迭代器指向的必定是被erase掉的所有元素之后的第一个元素。
3. 我们已经知道，对于vector使用erase，会使得被erase的元素与随后的元素的迭代器全部失效，故而这些访问都没有意义，比如上文对于find的访问，由于find已经是野指针了，这个访问没有分析的意义。

此处纠正对于指针的长期错误理解

指针/迭代器指向的一定是一个变量（左值，在内存中有对应空间即有地址），而不能只是一个值（右值，如0，1等）。故而指针指向地址这个说法本身就不准确，因为储存某个变量的内存空间的地址只是一个右值，而不是变量。变量A构造之后，储存A的内存空间的地址就固定了，此时如果指针P指向A，等价于指向了这块地址，然而我们仍然能够让P指针指向新的变量B，却无法改变储存A的内存空间。

回到vec的erase函数，我们已经明确了erase函数返回的新迭代器将会指向被erase掉的元素之后的第一个元素。
erase操作有两种可能，第一种不会发生shrink，被清除的数据之后的元素往前移动，填补之前被清除的元素。比如我传入vec[2]的迭代器，只erase掉vec[2]，那么原本位于vec[3]的元素就移动到之前vec[2]所在的内存空间，而我返回的迭代器一定指向vec[3]，所以看上去像是返回的迭代器和传入的迭代器都指向了同一个内存空间，但是这是显然错误的。不过是指向的元素碰巧先后占用了同一块内存空间。
情况二，erase导致了shrink的发生。也就是说vec容量减少的时候可能会发生shrink，可能整个vector被搬到另外一个较小的内存空间。这个时候，erase返回的指针仍然指向被erase掉的所有元素之后的第一个元素，储存这个元素的内存空间地址可能和之前的完全没有关系。

一个绝对安全的准则：
理解清楚迭代器所指的对象，在修改过容器后，换用新的迭代器，不使用之前的迭代器。

在c++ reference中，容器的修改操作有一项Iterator validity，表示该操作是否会引发迭代器失效。若修改容器后仍要使用原来的迭代器，查文档确定迭代器是否有效。

http://www.cplusplus.com/reference/algorithm/find/?kw=find

List

链表容器（底层实现是双向链表）

```
include <list>
include <algorithm>
std::list<int> l;//定义
l.push_front(1);//插入前端
l.push_back(2); //插入末端
std::find(l.begin(), l.end(), 2); //返回迭代器
l.insert(it, 4); //it为迭代器，将4插入到it指向的元素前面
```

注意，对list使用的find函数，需要包含头文件。如果find函数没有搜索到想要的对象，会返回指向end的指针。

```cpp
#include <iostream>
using namespace std;
#include <list>
#include <algorithm>
int main() {
std::list<int> l;//定义
l.push_front(1);//插入前端
l.push_back(2); //插入末端
l.push_back(5); //插入末端
l.push_front(100);//插入前端
l.push_back(100); //插入末端
l.push_back(2); //插入末端
auto it=std::find(l.begin(), l.end(), 2); //返回迭代器
l.insert(it, 4); //it为迭代器，将4插入到it指向的元素前面
cout<<*it<<endl;
cout<<"=========================="<<endl;
for(auto it=l.begin();it!=l.end();it++)
{
    cout<<*it<<endl;
}
    return 0;
}
output:
2
==========================
100
1
4
2
5
100
2
```

上述例子再次验证了之前对于迭代器的理解。我的it迭代器指向的永远是第一个2这个元素。虽然我在it前面插入了4，然后2这个元素往后移走了，但是it仍然指向2。另外，我们也可以发现，list的find功能遇见多个相同元素时，会默认返回从前到后第一个符合条件的元素的迭代器。

如果list的成员是自定义的class或struct，可以通过重载operator==和自己定义构造函数来构造find。

注意，==运算符重载的参数必须是常量左值引用

```cpp
class myint{
  int i;
  bool operator==(cosnt myint& other)
  {return (i==other.i);}
  myint(int t):i(t){};
};

int main(){
  list<myint>test;
  ...
  std::find(test.begin(), test.end(), myint(2));//查找test里面有没有成员i为2的myint
}
```

结合链表的原理很容易就能理解以下特点：

不支持下标等随机访问，ie：无法使用类似test[2]的语句。故而想要遍历list，必须用迭代器。
支持高速的在任意位置插入/删除数据（链表常常在需要不断插入删除的情况下使用）。
其访问主要依赖迭代器。
操作不会导致迭代器失效（除指向被删除的元素的迭代器外）。

关于list的更多函数参见：https://blog.csdn.net/yas12345678/article/details/52601578

关联容器Set

不重复元素构成的无序集合；

内部按大小顺序排列，比较器由函数对象Compare完成。
（注意：无序是指不保持插入顺序（不同于vector和list），容器内部排列顺序是根据元素大小排列。）

```
std::set<int> s;//定义
s.insert(1);
s.find(1);//返回迭代器（位置）
s.erase(s.find(1));    //导致被删除元素的迭代器失效（这句话是删除了集合里的元素1）
 s.count(1);    //1的个数，总是0或1（存在性）
```

关联容器Map

A.关联数组(pair)

map的值类型为：pair<Key,val>。

map中的元素key互不相同，需要key存在比较器。

可以通过下标访问（即使key不是整数）。**下标访问时如果元素不存在，则创建对应元素，而不会提示访问错误。**也可使用insert函数进行插入。

所谓的创建对应元素：比如map包含的元素为pair<Key, T>，那么找不到key，就会构造T（），调用默认构造函数。

```
template<class Key,
   class T,
   class Compare = std::less<Key>,
   class Allocator =
      std::allocator<std::pair<const Key, T> >
//这个其实是一个类模板，用Key的类来构造对应的map。
```

每个元素由两个数据项组成，map将一个数据项映射到另一个数据项中。

```
#include <string>
#include <map>
int main() {
    std::map<std::string, int> s;//创建了key为string类型，val为int类型的map
    s["oop"] = 1;//插入方式1
    //如果"oop"有对应的val，那就是赋值，如果没有，可以理解为插入，也可以理解为构造
//看起来像是重载了[]，使得下标不是int也可以访问，这是map类型支持的操作。在[]中用key直接访问val
    s.insert(std::make_pair(std::string("oop"), 1));//插入方式2
    return 0;
}

```

map支持用key直接访问对应迭代器：

```
s.find(key);//查询，仅需要提供key值，返回迭代器。
s.count(key);//统计，仅需要提供key值，返回0或1。
s.erase(it);//it是迭代器，删除it指向的元素，导致it失效
s.erase(s.find(key));
```

map常用作过大的稀疏数组或以字符串为下标的数组。

```
#include <string>
#include <map>

int main() {
    std::map<std::string, std::string> M;
    M["fp"] = "c";
    M["oop"] = M["fp"] + "++"; // M["oop"] = "c++"
//string的神奇操作：“a”+“b”=“ab”
    return 0;
}
```

Set和Map所用到的数据结构都是红黑树（一种二叉平衡树）
其几乎所有操作复杂度均为O(logn)

三、总结

序列容器（有顺序）和关联容器（无顺序）的区别

序列容器：vector、list

关联容器：set、map

序列容器与关联容器的区别：
序列容器中的元素有顺序，可以按顺序访问。
关联容器中的元素无顺序，可以按数值（大小）访问。
vector中插入删除操作会使操作位置之后全部的迭代器失效。
其他容器中只有被删除元素的迭代器失效。

选择依据

在实际应用中，容器的选择可能需要综合考虑多方面因素，包括算法复杂度，功能需求，内存分配策略等，下面提供几个可供参考但不完整的角度（可以进一步阅读《Effective STL》）：
算法复杂度：对于序列容器而言，如果在序列中间存在频繁的插入或删除操作，使用list，否则使用vector（或deque）
元素的顺序：如果需要在容器的任意位置插入新元素，需要选择序列容器而不是关联容器
元素查找速度：如元素的查找速度是关键的考虑因素，可以考虑排序的vector或关联容器set、map等
迭代器、指针或引用失效：如果希望在元素插入和删除操作后,迭代器、指针或引用失效的情况尽可能少出现，可以考虑使用list和关联容器set、map等