标准模板库(STL)

STL(Standard Template Library)标准模板库是标准C++库中的一部分，标准模板库为C++提供了完善的数据结构及算法。标准模板库包括三部分：容器、算法和迭代器。

容器是对象的集合，STL的容器有：vector、stack、queue、deque、list、mulitset, set和multimap, map等。STL算法是对容器进行处理，比如排序、合并等操作。迭代器则是访问容器的一种机制。

在C++定义数组时，我们必须提前知晓数组的大小，然后为了避免因为新数据的加入而导致越界，因而需要不断地进行越界检测，这样一来效率就大大降低了。而使用STL有一个非常大的优点就是不需要提前知道数组的大小，当有新元素加入时，容器会自动增大，删除元素时，容器会自动减小。

除此之外，STL还提供了大量的算法用于操作容器。STL具有可扩展性，也就是说可以增加新的容器和算法。

基本序列式容器

STL基本容器可以分为：

- 序列式容器，主要有list、vector、deque
- 关联式容器，主要有set、multiset、map、multimap

序列式容器可以像数组一样通过下标进行访问。关联式容器则是需要通过键值进行访问，关联式容器可以将任何类型的数据作为键值。

类型	容器	描述
序列式容器	vector	按照需要改变长度的数组
序列式容器	list	双向链表
序列式容器	deque	可以操作两端的数组
关联式容器	set	集合
关联式容器	multiset	允许重复的集合
关联式容器	map	图表
关联式容器	multimap	允许重复的图表

基本序列式容器效率比较

在前面介绍序列式的三种容器时，我们简单介绍了在容器各部位插入或删除元素时的处理效率，在此节我们做一个总结。

根据STL公布的容器各种操作效率，我们可以根据不同的需求来选择合适的容器。例如，我们需要频繁的在容器的任意位置插入或删除元素，则我们可以选择list，而非vector和deque。具体的总结见下表。

基本操作	vector	deque	list
在容器头部插入或删除元素	线性	恒定	恒定
在容器尾部插入或删除元素	恒定	恒定	恒定
在容器中部插入或删除元素	线性	线性	恒定
访问容器头部的元素	恒定	恒定	恒定
访问容器尾部的元素	恒定	恒定	恒定
访问容器中部的元素	恒定	恒定	线性

对于vector而言，它只是一个可以伸缩长度的数组，因此除了在尾部插入、删除数据外，在其它任何部位插入、删除数据都是线性的复杂度，容器长度越大，完成相应的操作也就越多。而访问元素则可以根据下标直接访问到，因此访问任何位置的元素，其效率都是恒定的。

对于deque而言，它是一个可以操作头部和尾部的并且可以伸缩长度的数组，因此它在头部和尾部插入、删除数据，效率是恒定的，但是在容器的中间插入元素，则它跟vector一样，同样是要移动其它元素的，因此在中部插入或删除元素效率是线性的。对于访问容器中的元素，它同样可以通过下标进行直接访问，因此效率也是恒定的。

对于list而言，它是一个双向链表，因此在任何位置插入或删除元素都不用移动其它元素，其效率始终是恒定的。对于访问元素，双向链表访问头尾元素都很方便，但是访问中间元素则需要逐一从头部或尾部一一遍历过去，因此访问容器中间的元素其效率是线性的。

在今后的程序设计过程中，如果需要使用容器，应该按照需求选择合适的容器，否则会大大降低程序的效率。如果我们只需要在容器尾部插入删除元素，则vector就够用了，如果还需要在头部也频繁的插入删除元素，则需要选择deque。

我们以vector、deque和list为例介绍基本序列式容器。我们先来看一个关于vector的例子。

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

int main(){
    vector<int> num;
    // 每个容器都有自己的访问方式：迭代器
    typedef vector<int>::iterator iter;
    num.push_back(50);
    num.insert(num.begin(), 10);
    num.insert(num.end(), 20);
    num.push_back(60);
    num.push_back(40);
    cout << num.size() << endl;
    for(int i = 0; i < num.size(); i++)
        cout << num[i] << " ";
    cout << endl;
    num.erase(num.begin());
    cout << num.size() << endl;

    // 通过迭代器访问容器
    for(iter it=num.begin(); it!=num.end(); it++){
        cout << num[i] << " ";
    }
```

```
    for(int i = 0; i < num.size(); i++)
        cout << num[i] << " ";
    cout << endl;
    return 0;
}
```

例2：

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

int main(){
    deque< int > num;
    num.push_back(50);
    num.insert(num.begin(), 10);
    num.insert(num.end(), 20);
    num.push_back(60);
    num.push_back(40);
    cout<<num.size()<<endl;
    for(int i=0; i < num.size(); i++)
        cout<<num[i]<<" ";
    cout<<endl;
    num.erase(num.begin());
    cout<<num.size()<<endl;
    for(int i=0; i < num.size(); i++)
        cout<<num[i]<<" ";
    cout<<endl;
    return 0;
}
```

我们将例1中的vector换成deque，运行程序我们发现两个程序的运行结果完全相同。是不是vector和deque相同呢？其实不是的，vector说到底是个数组，在非尾部插入元素都需要移动其它元素，而deque则不同，它是一个可以操作数组头部和尾部的数组，因此在头部或尾部插入或删除数据，其处理效率都是一样的。当我们需要频繁在头部和尾部插入或删除数据，则deque优于vector。

例3：

```
#include <iostream>
#include <string>
#include <list>
using namespace std;

int main(){
    list<string> str;
    str.insert( str.begin(), "A" );
    str.insert( str.begin(), "B" );
    str.insert( str.end(), "C" );
    str.insert( str.end(), "D" );
    str.insert( str.begin(), "E" );
    str.insert( str.begin(), "F" );
    // list<string>::iterator iter;
```

```
    // typedef list<string>::const_iterator it;    // 迭代器只能访问容器，不能修改容器内的元素
    typedef list<string>::iterator it;             // 迭代器不仅能访问容器，还能修改容器内的元素
    it iter;
    for(iter = str.begin(); iter != str.end(); iter++)
        cout << *iter << endl;
    str.reverse();
    for(iter = str.begin(); iter != str.end(); iter++)
        cout<< * iter << endl;
    return 0;
}
```

在本例中我们定义了一个list容器string类型的实例str，之后我们先在容器中添加了6个string类型元素，为了遍历str容器，我们定义了一个迭代器iter。**通常每定义一个容器，就会有一个与容器数据类型相关的迭代器**，本例中定义了容器str，则它的对应的容器有：

```
list<string>::iterator
list<string>::const_iterator
```

如果我们不需要修改容器中的元素，仅仅只是进行访问的话，则可以定义为const_iterator。

为了从头到尾遍历容器，我们先将迭代器指向str.begin()，for循环的结束条件是str.end()，每次运行一遍循环体中的内容，迭代器自增一次，相当于指向下一个元素，我们之所以能够直接使用自增运算符，那是因为在容器的类中系统已经重载过了自增操作符。当我们需要获得当前迭代器所指的元素时，我们可以用取值操作符“*”来操作迭代器，“iter”就为迭代器所指向的元素。在此程序中我们之所以没有按照vector和deque的方式，以下标进行访问容器中的元素，那是因为list并没有重载下标操作符，因而不能根据下标进行直接访问。

在主函数中我们调用了reverse函数，对容器中的元素进行翻转，然后再次打印容器中的元素。**list容器是一个双向链表**，因此在容器中的任何位置插入元素，都不需要移动其它元素，因此执行效率是稳定的。

基本关联容器

基本的关联式容器主要有：set、multiset、map和multimap，这四种容器可以分为两组：map和set。

set可以理解为我们数学中的集合，它可以包含0个或多个不重复的数据，这些数据被称为键值。map也是一种集合，它同样可以包含0个或多个不排序的元素对，每一个元素对有一个键值和一个与键值相关联的值，在map中键值是不允许重复的。而multiset则是允许重复的集合，multimap则是允许有重复键值的map，因为multiset和multimap可以看做是set和map的扩展，因此我们将主要介绍set和map。

例1：

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

int main(){
   set<int> s;
   s.insert(s.begin(), 9);
   s.insert(1);
   s.insert(s.end(),4);
```

```
    s.insert(5);
    s.insert(6);
    s.insert(7);
    s.insert(9);
    s.insert(0);
    set<int>::const_iterator iter;
    for(iter = s.begin(); iter != s.end(); iter++)
        cout<< *iter << " ";
    cout<< endl;
    iter = s.find(2);
    if(iter == s.end())
        cout<< "NOT Found 2!" << endl;
    else
        cout<< "Found 2 in set!" << endl;
    iter = s.find(5);
    if(iter == s.end())
        cout<< "NOT Found 5!" << endl;
    else
        cout<< "Found 5 in set!" << endl;
    return 0;
}
```

本例是set的一个示例，在主函数中我们先创建了一个set容器整型的示例s，之后就开始向容器中添加数据。对于set容器，insert被重载过，其调用方式有两种，我们既可以向前面序列式容器那样调用：

```
s.insert(s.begin(), 9);
s.insert(s.end(), 5);
```

也可以不指定插入位置，而直接插入元素：

```
s.insert(6);
```

但是无论以哪种方式进行调用，insert函数都能保证插入元素后set容器中不会出现重复的元素。

在插入一些元素之后，我们就定义了一个迭代器iter用于访问容器中的元素。访问的方法和序列式容器相同。

find函数可以用于查找set容器中是否包含指定的键值，如果存在，则返回指向该键值的迭代器，如果不存在，则返回与end()函数相同的结果。在例1中，我们分别查找了键值2和5是否存在于集合s中，因为s容器没有2元素，因此返回值等于s.end()，存在5元素，因此查找5元素时，函数返回结果不等于s.end()。

例2：

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

int main(){
    map<char, int> m;
    m['a'] = 1;
    m['b'] = 2;
    m['c'] = 3;
```

```
    m['d'] = 4;
    m['e'] = 1;
    m['f'] = 2;
    m['g'] = 3;
    m['h'] = 4;
    m['a'] = 0;
    map<char, int>::iterator iter;
    for(iter = m.begin(); iter != m.end(); iter++)
        cout << iter->first << " -- " << iter->second << endl;
    return 0;
}
```

本例是一个map容器的示例程序，map是一种关联式的列表，即将一个键值与一个值一一对应起来。我们直接来看主函数，主函数一开始定义了一个map容器示例m，“map< char, int > m;”语句中char表示键值的数据类型，int表示与键值对应的值的数据类型。我们用字符来作为键值，键值在map中是不允许出现重复的，但是与键值对应的值value可以出现重复，如本例中 m['a']和m['e']相同。但是如果键值出现相同，则以最后一次出现的作为结果，例如本例中一开始“m['a'] = 1;”，在后面又出现“m['a'] = 0;”，此时不会有语法错误，这两句可以理解为“m['a'] = 1;”是给“m['a']”赋初值，其值为1，而“m['a'] = 0;”则可以理解为我们将“m['a']”的值由1修改为0。

遍历map时同样是使用迭代器，在函数中我们定义了一个迭代器iter，由于map中是由元素对组成的，包含两个元素，因此遍历方法与前面所介绍的容器稍有不同，前面的容器用*iter就可以直接获得所需要的元素，而map容器则需要通过iter->first访问键值，并用iter->second访问与键值对应的值。

例2最终运行结果如下：

```
A -- 0
B -- 2
c -- 3
d -- 4
e -- 1
f -- 2
g -- 3
h -- 4
```

multimap

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

int main(){
    multimap<string, int> mult;
    mult.insert(pair<string, int>("abc", 2));
    mult.insert(pair<string, int>("eq", 5));
    mult.insert(pair<string, int>("abc", 13));
    mult.insert(pair<string, int>("aa", 2));
    mult.insert(pair<string, int>("kk", 20));

    typedef multimap<string, int>::const_iterator iter;
    iter it;
    for(it=mult.begin(); it!=mult.end(); it++)
        cout << it->first << "-----" << it->second << endl;
```

```
    return 0;
}
```

打印结果：(键的值排序)

```
aa------2
abc-----2
abc-----13
eq------5
kk------20
```