Neusoft

C++标准模板库使用

目录

一、模板简单介绍：

1. 函数模板

请看看下面这个题目：
实现一个函数，输入 2 个变量，输出这两个变量中值比较大的元素。
要求：此函数可以接受 int、char 以及 double 类型的参数。

对于这个问题，如果是 C 语言的话，估计实现会是这个样子：

```
// 用于比较 char 的函数
char MaxOfChar( char cNum1, char cNum2 )
{
    return ( cNum1 > cNum2 ) ? cNum1 : cNum2;
}
// 用于比较 int 的函数
int MaxOfInt( int iNum1, int iNum2 )
{
    return ( iNum1 > iNum2 ) ? iNum1 : iNum2;
}
// 用于比较 double 的函数
double MaxOfDouble( double dNum1, double dNum2 )
{
    return ( dNum1 > dNum2 ) ? dNum1 : dNum2;
}
```

但是到了 C++时代，由于存在重载的概念，所以实现起来应该是这个样子：

```
// 用于比较 char 的函数
char Max( char cNum1, char cNum2 )
{
    return ( cNum1 > cNum2 ) ? cNum1 : cNum2;
}
// 用于比较 int 的函数
int Max( int iNum1, int iNum2 )
{
    return ( iNum1 > iNum2 ) ? iNum1 : iNum2;
}
// 用于比较 double 的函数
double Max( double dNum1, double dNum2 )
{
    return ( dNum1 > dNum2 ) ? dNum1 : dNum2;
}
```

对比上面两个例子，对于函数的实现来说，代码量没有什么变化，只不过函数的名字由 3 个变成了 1 个。这样并非没有意义，对于使用这个函数的用户来讲，他的工作将会减少，对于 C++的实现方式，完全没有必要去记住哪个函数对应哪种数据类型，因为不管是针对哪种数据类型的比较，只需要简单的调用 Max()就可

以了。

那么还有没有更好的解决方式呢？答案是肯定的，那就是模板。请看下面的代码：

```
template < class T >
T Max( const T &Input1, const T &Input2 )
{
    return ( Input1 > Input2 ) ? Input1 : Input2;
}
```

OK，就这么简单，一个比较函数完成了，它可以接受任何类型的参数，包括上边提到的 int、char、double，甚至任何自定义类型(只要你实现了它的比较运算符 operator >)。

这**就是一个模板（函数模板）的例子**，从这个例子可以看出运用模板的好处，更重要的是，具体选用什么样的实际函数是在编译时刻就决定好的，丝毫不会影响运行时的效率。

PS：对于这个特定的例子，还有更简单的方式，就是用宏：

```
#define Max( Input1, Input2 ) ( ( Input1 > Input2 ) ? Input1 : Input2 )
```

当然我们不建议这么做，因为宏没有类型检查。

上面是一个函数模板的实现，我们在后面还要提到类模板。现在先让我们来看看模板的格式。

```
template < class T >————〉①
T Max( const T &Input1, const T &Input2 )
{
    return ( Input1 > Input2 ) ? Input1 : Input2;
}
```

①这个是模板头，此项是必须的，其中 T 是一个替代符，它可以是除去固有符号(int 、long 等)的任何字符，用来表示一个类型（可以是原生类型，也可以是自定义类型）。

除去模板头，其它地方的书写和正常的函数定义是完全一样的。

请注意，在上面的 Input 参数中，我用的是 const&的，这样做的好处是减少函数调用时候的开销，因为我们不知道 T 类型到底有多大。

2. 类模板

在很多情况下，你可能会需要设计一个方法类或者容器类，通常对于这种用于实际应用的类会被要求支持多种数据类型。如果用传统的类设计方法来设计的话，可能就需要设计多个类或者在类里面定义多个不同的方法来支持不同的数据类型（就像前面那个比较函数一样），这时候就需要用到模板类的设计方法。请看下面的题目：

设计一个容器类，要求可以支持多种数据类型，支持添加元素、删除元素等功能。

该如何设计这个类呢？是不是还需要很多个支持各种类型的成员函数？比如说 AddInt(),AddChar(),DelInt(),DelChar()……。完全不需要，在这里我们用到的应该是类模板，看下面的例子：

```
#define MAX_NUM     100
#define SUCCESS      0
#define FAILURE     -1
```

```
template < class T >
class CVector
{
public:
    CVector() : m_Size(0)
    {

    }
    unsigned int Size()
    {
        return m_Size;
    }
    int push_back( const T& Element )
    {
        if( m_Size > MAX_NUM - 1 )
        {
            return FAILURE;
        }
        DataList[ m_Size ] = Element;
        m_Size++;
        return SUCCESS;
    }
    int pop_back( T &Element )
    {
        if( 0 == m_Size )
        {
            return FAILURE;
        }
        Element = DataList[ m_Size - 1 ];
        m_Size --;
        return SUCCESS;
    }
private:
    unsigned int m_Size;
    T DataList[ MAX_NUM ];
};
```

这是一个简单的用模板实现的容器类，他可以支持各种数据类型，支持尾部添加和尾部删除操作。使用的时候应该是这种方式：

```
CVector< TypeName > TypeList;
TypeList.push_back( Element );
TypeList.pop_back( Element );
.....
```

类型名，可以是任何类型

可以看到类模板的格式与函数模板类似，只是使用的时候存在区别，类模板在使用的时候需要进行特化，就是告诉编译器我需要什么类型的实现(实际上函数模板也需要特化，只不过这个过程是隐式的)。

```
CVector< TypeName > TypeList;
```

实际上，我们刚刚实现了一个标准模板库(STL)中的 Vector，只不过真正的 Vector 比这复杂的多，因为它需要考虑到各种性能和空间的因素，行为也很复杂。不过 STL 设计的思想正是基于模板。

二、STL 概论

STL，虽然是一套程序库(Library)，但不是一般印象中的程序库，而是一个有着划时代意义，背后拥有着先进技术与深厚理论的产品。说他是产品也可以，说他是规格也可以，说是软件组件技术发展史上一个大突破点，它也当之无愧。

长久以来，软件界一直希望建立一种可重复运用的东西，以及一种可以制造出"可重复运用的东西"的方法，让工程师/程序员的心血不至于随时间的推移、人事异动而烟消云散。从副程式(subroutines)、程序(procedures)、函数(functions)、类别(classes)，到函数库(function libraries)、类别库(class libraries)、各种组件(components)，从结构化设计、模组化设计、物件导向设计，到样式(patterns)的归纳整理，无一不是软件工程的漫漫奋斗史，为的就是复用性(reusebility)的提升。

————摘自《STL 源码分析》

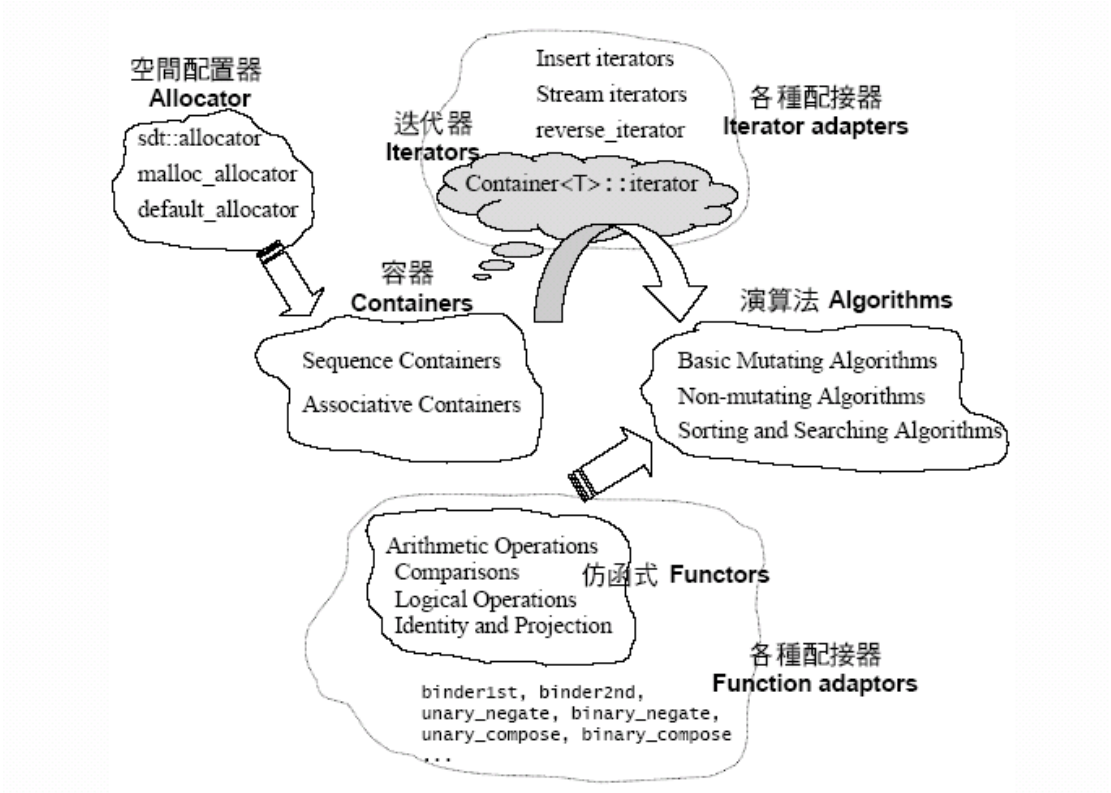
STL 就是在这个背景下诞生的。STL 的价值在两方面。低层次而言，STL 带给我们一套急具价值的零组件，这种价值就像 MFC 对于 Windows 开发过程所带来的价值一样，直接而明朗。除此之外 STL 还带给我们一个高层次的以泛型思维为基础的设计理念。在这个教程里面，我们只会提到如何使用 STL，对于深层次的东西，需要读者自己去慢慢体会。

三、STL 的组件以及关系

STL 提供 6 大组件，彼此可以组合套用：

1. 容器(containers):各种资料结构，如 Vector、List、Map 等，用来存储各种数据。
2. 演算法(algorithms):各种常用的算法，如 sort、search、copy 等，它的作用是为提供各种常用的操作。
3. 迭代器(iterators):一个非常重要的组件，用来将容器和演算法联系起来。也就是通常所说的泛型指针。
4. 仿函数(functors):行为类似函数，可作为演算法的某种策略(policy)。
5. 配接器(adapters):一种用来修饰容器或仿函数界面的东西。
6. 配置器(allocators):负责空间配置与管理，用来保证容器空间的正确分配。

STL 六大组件的关系如下图所示：Container 通过 Allocator 取得数据存储空间，Algorithm 通过 Iterator 存取 Container 内容，Functor 可以协助 Algorithm 完成不同的策略变化，Adapter 可以修饰或者套接 Funtor。



在本文中，只会简单介绍容器的使用方法，可能会涉及一些迭代器或者演算法。更深一步的东西需要自行学习。

四、常用容器介绍

容器基本上可以分为两大类，序列式容器和关联式容器。

1. 序列式容器

所谓序列式容器，就是说容器中的元素都可序(Ordered)，但未排序(Sorted)。

序列式容器包括：Vector、List、Queue、Stack 等。

1.1 Vector

1.1.1 Vector 基本结构

Vector 的行为方式完全是一个数组，它的内存空间完全是连续的，只不过这个数组的大小是可变的，就是说你完全可以不必关心你到底向这个数组里面添加了多少个元素，只管继续添加就对了（当然内存耗尽的情况除外）。

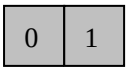
实际上，Vector 的实现就像文章开头所举的例子一样，在初始化的时候，会申请一定的空间用来存储数据，一旦所申请的空间不够用了，它会在另外的地方开辟一块新的内存空间，然后将原空间中的元素全部拷贝到新的空间中，如果不特殊指定，每次新申请的通常是原空间的两倍。

Vector 的空间操作看起来像下面这个样子：

```
// 申请一个 Vector，向其加入 1 个元素，默认值是 0
vector<int> iv(1)
```

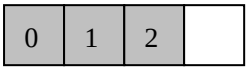
此时 iv.size() == 1; //元素个数
iv. capacity() == 1 //空间大小

```
//向 Vector 尾部加入一个元素“1”
iv.push_back(1);
```




```
iv.size() == 2
iv.capacity() == 2

//向 Vector 尾部加入一个元素“2”
iv.push_back(2);
```



在这个操作之前，Vector 空间数为 2，向 Vector 中加入元素的时候发现空间不够，则以原空间的 2 倍申请一块新的内存。

```
iv.size() == 3;
iv.capacity() == 4;
```

可以观察到，Vector 申请的空间要比其元素个数多，剩余的空间作为预留空间，用于以后的元素 Push 操作。这时 Vector 的一个策略，前面提到过，Vector 的空间是连续的，而连续空间的申请和释放操作都相当耗费时间，所以为了避免频繁申请和释放操作，就采用了“预留空间”这个策略。但这样的做的坏处也非常明显，那就是空间浪费。

1.1.2 Vector 的常用成员函数

构造函数：

vector（）

说明：默认构造函数

```
Example：vector< int > iv;
```

vector（ SizeType count ）

说明：构造一个 vector，初始元素个数为 count，初始值均为 0

```
Example: vector< int > iv(3);
```

vector（ SizeType count， ConstType &Val ）

说明：构造一个 vector，初始元素个数为 count，初始值均为 Val

```
Example: vector< int > iv( 3, 2 );
```

vector（ const _vector& SourceVector ）

说明：构造一个 Vector，并将 SouceVector 中的元素完全 Copy 到构建的 Vector 中

```
Example：  vector< int > iv1;
           vector< int > iv2( iv1 );
```

其他成员函数：

reference at（ size_type Pos ）

说明：取得 vector 在 Pos 位置的元素的引用，此函数和数组下标用法类似。

```
Example：  vector < int > iv;
           iv.push_back( 1 );
           iv.push_back( 2 );
           iv.push_back( 3 );
           int i = iv.at(2); //此时 i 的值应该是 3，
```

reference back（）

说明：返回 vector 最尾部元素的引用，相当于 at(LastPos)。

```
Example：  vector < int > iv;
           iv.push_back( 1 );
           iv.push_back( 2 );
           iv.push_back( 3 );
           int i = iv.back();//此时 i 的值应该是 3，
```

iterator begin()

说明：返回 Vector 初始位置的迭代器(iterator)。————〉②

```
Example：  vector < int > iv;
           iv.push_back( 1 );
           iv.push_back( 2 );
           iv.push_back( 3 );
           vector<int>::iterator  It = iv.begin();
           int i = *It; //此时 i 的值应该是 1，
```

②在这里顺便简单介绍一下迭代器(iterator)。为了将容器和演算法联系起来，需要一种指向容器中元素的指针，它的使用方法必须类似于原生指针，也就是说我们可以利用*Iter 取得元素本身，也可以利用 Iter->调用元素的成员函数，甚至我们可能利用 Iter++或者 Iter--方式来取得元素序列中前一个或者后一个元素（当然某些容器的迭代器不支持这样的操作）。为了更好的封装迭代器的内部实现以及提高性能，每一个容器都有其独立的迭代器，这些特定的迭代器都是根据容器本身的特性来进行设计的，充分的考虑到了性能和空间上的问题。对于 Vector 来讲，它的迭代器在使用的时候完全可以等同于一般的指针。

Size_type capacity（）

说明：返回 vector 当前空间大小，以元素个数为单位。

```
Example：  vector < int > iv;
           iv.push_back( 1 );
           iv.push_back( 2 );
           iv.push_back( 3 );
           int iSapce = vi.capacity(); //此时 iSPace 的值应该是 4，
```

void clear()

说明：清空 vector 中所有元素，但不释放空间

```
Example：  vector < int > iv;
```

```
iv.push_back( 1 );
iv.push_back( 2 );
iv.push_back( 3 );
iv.clear();
int iSize = iv.size(); //此时 iSize = 0
int iSapce = iv.capacity(); //此时 iSapce 的值应该是 4，
```

bool empty（）const

说明：判断 vector 是否为空。

```
Example：  vector < int > iv;
           iv.push_back( 1 );
           iv.empty();//结果应该为假
           iv.clear();
           iv.empty();//结果应该为真
```

iterator end（）

说明：返回指向 vector（最后一个元素+1）的迭代器，通常用来判断循环是否结束。

```
Example：  vector < int > iv;
           iv.push_back( 1 );
           iv.push_back( 2 );

          //循环整个 Vector
          for( vector<int>::iterator it = iv.begin(); it != iv.end(); ++it )
          {
           Do Something….
          }
```

iterator erase（ iterator Where ）

iterator erase（ iterator First， iterator Last ）

说明：删除一个位置为“Where”的元素，或者删除从“First”到“Last -1 ”的元素

```
Example：  vector< int > iv;
           iv.push_back( 1 );
           iv.push_back( 2 );
          iv.push_back( 3 );
           iv.erase( iv.begin() + 1 ); //删除元素“2”
           iv.erase( iv.begin(), iv.begin() + 2 ); //删除元素“1”和“3”
```

erase 的参数 Where 和 First 、Last 必须在[begin(),end()]范围之内，否则会发生非法访问的错误。

reference front()

说明：返回 vector 最前面元素的引用，相当于 at(FirstPos)。

```
Example：  vector< int > iv;
           iv.push_back( 1 );
           iv.push_back( 2 );
```

```
iv.push_back( 3 );
int i= iv.front(); //此时 i = 1;
```

iterator insert（iterator Where， const Type & Val ）

说明：在“Where”位置插入指定元素“Val”。

```
Example： vector< int > iv;
          iv.push_back( 1 );
          iv.push_back( 3 ); //此时 Vector 中的元素为 1，3
          iv.insert（ iv.begin() + 1, 2 ）; //此时 Vector 中的元素为 1,2,3
```

insert 的位置必须在[begin(),end()]范围之内，否则会发生非法访问的错误。

void push_back(const Type &Val)

说明：在 Vector 尾部加入一个元素

void pop_back()

说明：删除 Vector 尾部的元素

```
Example：vector< int > iv;
         iv.push_back( 1 );
         iv.push_back( 2 );
         iv.push_back( 3 );//此时元素为 1，2，3
             iv.pop_back();//此时元素为 1，2
```

size_type size() const

说明：返回 Vector 中元素的个数。

reference operator[](size_type Pos)

说明：[]操作符，使 vector 可以像数组一样访问

```
Example：vector< int > iv;
         iv.push_back( 1 );
         iv.push_back( 2 );
         iv.push_back( 3 );
             //iv[0] = 1， iv[1] = 2, iv[2] = 3
```

请注意：

1、operator[]只能用来取得已经存在的元素，而不能向 vector 中添加元素，例如下面的代码是错误的：

```
vector< int > iv;
iv[ 0 ] = 1; //错误，vector 的空间还没有被分配
```

2、operator[]和正常数组的使用方法一样，也同样没有越界检查，比如你通过 iv[10] 的方式访问一个只有 1 个元素的 vector，不会被提示出错，但这样做可能会有不可预计的事情发生。

1.1.3 Vector 综合示例：

题目：实现一个存储正整数的类，此类中的数据用随机数的方式进行填充。提供的方法包括：打印所有的整数、打印数据中所有的素数、判断某随机位置上的数字是否为素数、删除所有的素数

分析：考虑到此数据列表需要支持随机操作，并且不用关心元素所处的位置，所以用 Vector 来实现是个不错的选择

实现：

```
#include <stdlib.h>
#include <time.h>
#include <iostream>
#include <vector>

using namespace std;

class CDataSet
{
public:
  static enum ePrintRange
  {
      ALLDATA = 0,  PRIMEDATA = ALLDATA + 1
  };

  // 用随机数构造一个数据列表
  // iInitDataCnt: 随机数的个数; iMaxData: 随机数中的最大数字
  CDataSet( unsigned int iInitDataCnt, unsigned int iMaxData )
  : m_DataCnt( iInitDataCnt )
  {
      srand( ( unsigned )time( NULL ) );
      for( int iLoop = 0; iLoop < m_DataCnt; ++iLoop )
      {
          m_DataList.push_back( (unsigned int)rand() % iMaxData );
      }
  }

  // 删除数据列表中的所有素数
  int DeletePrimeData()
  {
      int iDeleteCnt = 0;
      vector< int >::iterator it;
      for( it = m_DataList.begin(); it != m_DataList.end(); ++it )
      {
          if( m_bIsPrimeData( *it ) )
          {
              m_DataList.erase( it );
              m_DataCnt--;
              iDeleteCnt++;
          }
      }
```

```
    // 返回删除元素的个数
    return iDeleteCnt;
}

// 按照传入的打印范围进行数据打印
void Print( ePrintRange iPrintRange ) const
{
    switch ( iPrintRange )
    {
        case ALLDATA:
        {
            m_PrintAllData();
            break;
        }
        case PRIMEDATA:
        {
            m_PrintPrimeData();
            break;
        }
        default:
        {
            break;
        }
    }
}

// 判断 iDataIndex 位置的数字是否为素数
bool IsPrimeData( unsigned int iDataIndex )
{
    if( iDataIndex >= m_DataCnt )
    {
        return false;
    }
    return m_bIsPrimeData( m_DataList[ iDataIndex ] );
}

private:
// 判断一个数是否是素数
bool m_bIsPrimeData( int iInputData ) const
{
    for( int iLoop = 2; iLoop < iInputData; ++iLoop )
    {
        if( 0 == iInputData % iLoop )
```

```
            {
                break;
            }
        }
        return ( ( iLoop == iInputData ) && ( iLoop != 1 ) );
    }

    //打印所有整数
    void m_PrintAllData() const
    {
        cout << " All Data List: " << endl;
        for( int iLoop = 0; iLoop < m_DataCnt; ++iLoop )
        {
            cout << m_DataList[ iLoop ] << "\t";
        }

        cout << endl;
    }
    //打印所有素数
    void m_PrintPrimeData() const
    {
        cout << "Prime Data List: " << endl;
        for( int iLoop = 0; iLoop < m_DataCnt; ++iLoop )
        {
            if( m_bIsPrimeData( m_DataList[ iLoop ] ) )
            {
                cout << m_DataList[ iLoop ] << "\t";
            }
        }
        cout << endl;

    }
    vector< int > m_DataList;
    int           m_DataCnt;
};

//客户端实现
void main()
{
    //构造一个数据列表，里面存储 10 个数字，每个数字是小于 100 的随机数字
    CDataSet DataList( 10, 100 );
    DataList.Print( CDataSet::ALLDATA );
    DataList.DeletePrimeData();
    DataList.Print( CDataSet::ALLDATA );
```

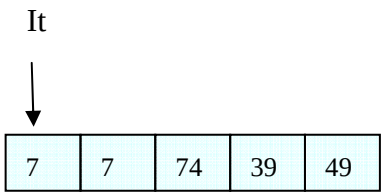
```
}
```

大家可以试一试刚刚实现的这个程序，它可以在大部分的情况下工作的很好，但有的时候可能输出和我们预想的并不一致，甚至可能造成程序的崩溃！这是为什么呢？问题处在 DeletePrimeData（）这个函数，看一看我们的实现：

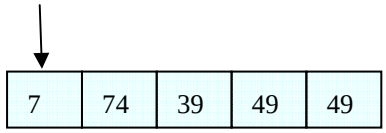
```
for( it = m_DataList.begin(); it != m_DataList.end(); ++it )
{
  if( m_bIsPrimeData( *it ) )
  {
      m_DataList.erase( it );
      m_DataCnt--;
      iDeleteCnt++;
  }
}
```

如果随机产生的数据系列是：7，7，74，39，49.....，让我们看一看函数是怎么执行的。

开始的时候，内存以及 Iterator 如下所示

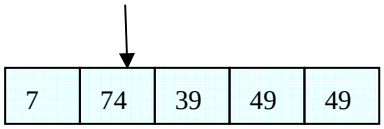


在经过一次素数判断及删除之后，内存变为：



第一个元素被删除，所有元素向前移位。

当第二次循环开始的时候，Iterator++，这时内存及 Iterator 为：



没错，这时候再判断元素的话，判断的将是 74 而不是第二个 7，也就是说，我们漏掉了第二个元素。如果这种问题发生在 Vector 的尾部的话，就可能造成内存的非法访问而引起程序的崩溃。那这个问题该怎么解决呢？看下面的方法：

```
void DeletePrimeData()
{
    vector< int > TempList;
    vector< int >::iterator it;
    for( it = m_DataList.begin(); it != m_DataList.end(); ++it )
    {
        if( !m_bIsPrimeData( *it ) )
        {
```

```
                TempList.push_back( *it );
            }
        }
        m_DataList.assign( TempList.begin(), TempList.end() );

        m_DataCnt = m_DataList.size();
    }
```

在这个函数中，申请了一个新的 Vector，将原 Vector 中的非素数全部 Copy 到新的 Vector 中，然后将原 Vector 中的元素清空并用新 Vector 的元素进行重新填充(m_DataList.assign(TempList.begin(), TempList.end());)。这样做既可以避免在循环中删除元素带来的 Iterator 混乱，有可以在一定程度上提高效率（对于 Vector 中间元素进行操作将有很大的性能问题）

注意：绝对应该禁止在循环体中对 Vector 进行插入或者删除操作

1.2 List

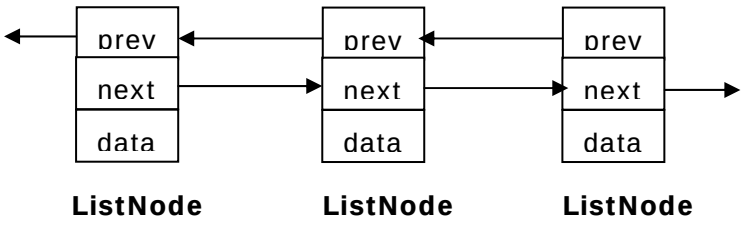
1.2.1 List 的基本结构

相对于 Vector 的线性存储空间，List 就复杂的多，它每次添加或者删除一个元素，就会申请或者释放一个元素的空间，然后用指针将他们联系起来。这样的好处就是精确配置内存，绝对没有一点的浪费。而且对于元素的插入和删除，List 都是常数时间。

List 的基本构成是节点。一下是节点的结构：

```
template < class T >
struct _list_node
{
  Typedef void     *void_pointer;
  void_pointer     prev;    //指向前面元素的指针
  void_pointer     next;    //指向后面元素的指针
  T                data;    //节点实体，用来存储数据
}
```

从上面的结构可以看出，List 实际上是一个双向链表(更确切的说是环状)

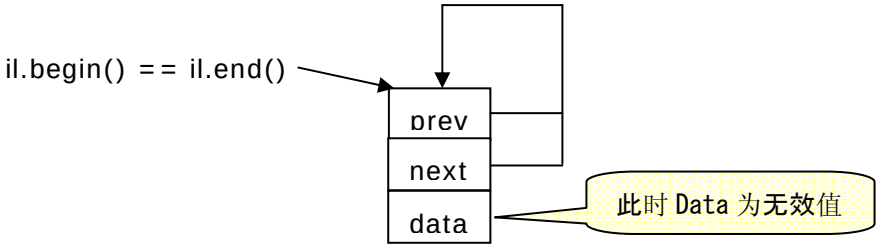


下面根据一段程序来分析 List 的行为方式：

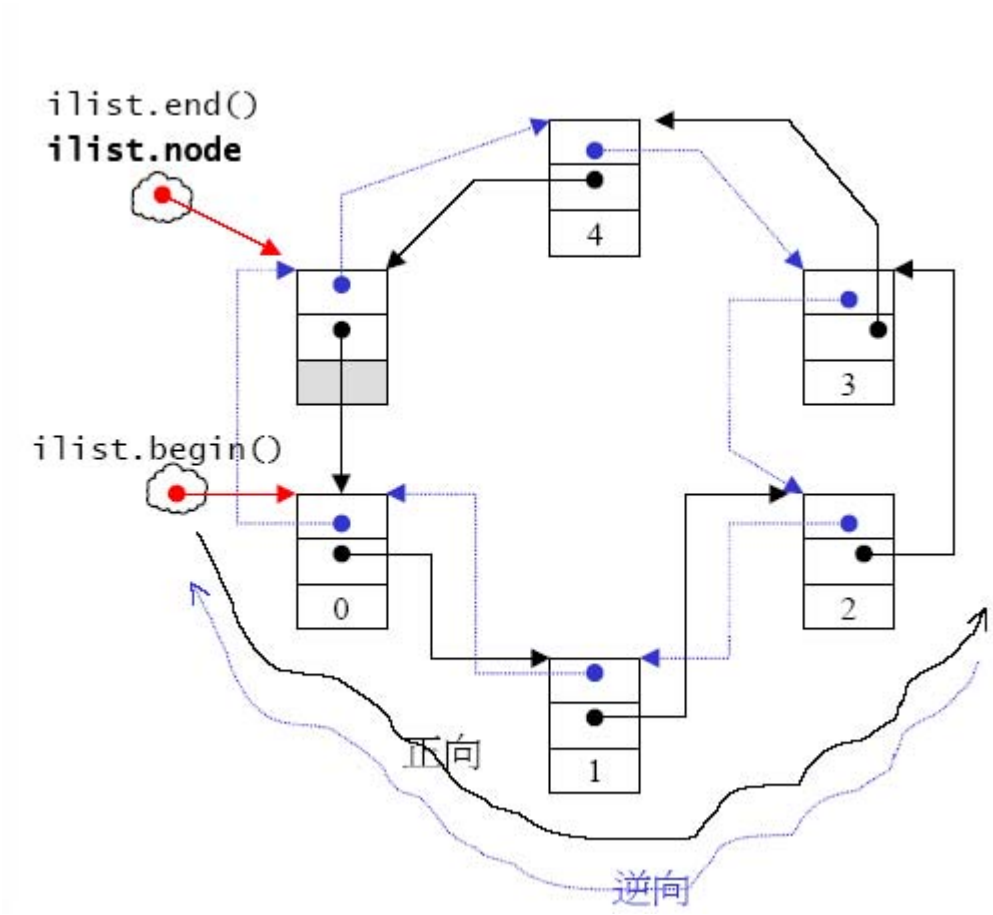
```
Example：
list< int > il;
for( int iLoop = 0; iLoop < 5; ++iLoop )
{
   il.push_back( iLoop );
```

```
}
```

在定义 List 之后(list< int > il)，会为这个 List 申请一个元素的空间，il.begin()和 il.end()都是指向这块空间，而这个内存中的实体（data）是一个无效的数值。List 的结构看起来是下面的样子：



在连续向 List 中添加 5 个元素之后，List 的结构变为下面的样子：



1.2.2 List 的常用成员函数：

List 的构造函数和成员函数比较多，在这里只会介绍一些常用的。并且，有一些函数对于 STL 中的很多容器都是通用的，在这里也不一一介绍了。

常用构造函数：

list()

说明：默认构造函数

Example: list < int > il;

list(size_type Count)

说明：构造一个 list，初始元素个数为 Count，初始值均为 0

```
Example: list < int > il( 3 );
```

list(size_type count, const Type &_Val)

说明：构造一个 List，初始元素个数为 count，初始值均为_Val

```
Example: list< int > il( 3, 5 )
```

list(const list &SourceList)

说明：构造一个 List，将 SourceList 中的所有元素 Copy 到新构造的 List 中。

```
Example: list< int > SourceList( 3, 5 )
         list< int > il( SourceList );
```

其它常用成员函数：

在 List 中，有很多成员函数的使用方法是等同于 Vector 的，这些方法包括：back()、begin（）、clear（）、empty（）、end（）等等（具体可参考 MSDN）。在这些方法里面值得一提的是一些删除元素的方法(clear()、erase（）等)，在 List 中，元素删除的同时会清除其原先所占用的空间，这一点和 Vector 是有区别的。下面介绍一些不同于 Vector 的方法：

void merge（ list< Type, Allocator> &List2 ）

说明：对于两个完全按照递增顺序排序的 List，这个函数将把 List2 中的元素 Copy 到 List1 种，并且按照递增的方式排序，Copy 之后，将 List2 中的所有元素清空。

```
Example:
    list< int > List1;
    list< int > List2;
    List1.push_back( 1 );
    List1.push_back( 3 );
    List1.push_back( 5 );
     List2.push_back( 2 );
     List2.push_back( 4 );
     List2.push_back( 6 );
     List1.merge( List2 );
```

调用 Merge 之前，List1 : 1, 3, 5	List2: 2, 4, 6
调用 Merge 之后，List1 : 1, 2, 3, 4, 5, 6	List2: 空

请注意，调用 Merge 之前，必须保证两个 List 都是完全排序的，不然会产生一个比较混乱的结果，而一般情况下，这种情况并不是用户的本意（产生这种结果的原因可参考 Merge 的实现方法）。如果是针对用户自定义类型的 List，在使用这个函数之前，还要保证实现了 operator < 。

void pop_front()

说明：删除 List 中的第一个元素，在 Vector 中没有实现这个成员函数，因为在 Vector 中如果需要删除首元素的话，可能会带来效率上的很大损失。

Example:

```
list< int > il;
il.push_back( 1 );
il.push_back( 2 );//List 中元素为：1，2
il.pop_front();   //List 中元素为:2
```

void push_front(const Type &_Val)

说明：向 List 的首位插入一个元素。

Example：

```
list< int > il;
il.push_front( 1 );
```

void sort()

说明：对 List 按照从小到大的顺序进行排序，对于自定义类型必须实现 operator <

Example:

```
list< int > il;
il.push_back( 3 );
il.push_back( 1 );
il.push_back( 2 );//List 中的元素： 3，1，2
il.sort(); //List 中的元素：1，2，3
```

1.2.3 List 的综合示例：

题目：有一个书架，上面的图书按种类分类（种类自定），现在有一批新的图书需要按照各自的种类插入到此类图书的末尾位置。

分析：每种图书都有自己的种类，并且每类图书按照次序存放，说明原有的顺序不应该被打乱，并且这个书架可能需要频繁的中间插入操作。考虑到 **List** 在中间插入和删除的常数效率，这个问题的容器选择 List 是很好的。

实现：

```
#pragma warning(disable: 4786)

#include <algorithm>
#include <iostream>
#include <list>
using namespace std;

#define NOINDEX        -1

enum eBookKind
{
    COMPUTER    = 0,
    ENGLISH     = COMPUTER    + 1,
    CHEMISTRY   = ENGLISH  + 1,
```

```
    PHYSICS       = CHEMISTRY + 1,
    ALLBOOKKIND = PHYSICS + 1,
};

struct Book_t
{
    bool operator == ( const Book_t &_Right ) const
    {
      return ( iBookKind == _Right.iBookKind );
    }

    int                 iBookKind;  // Book 的种类
    int                 iBookIndex; // Book 在一类图书中的位置

    static int      KindCnt[ ALLBOOKKIND ]; // 静态成员，表示每种图书的数量
};

int Book_t::KindCnt[ ALLBOOKKIND ] = { 2, 1, 2, 1 };

void Print( list< Book_t> StudentList )
{
    list< Book_t >::iterator it;
    for( it = StudentList.begin(); it != StudentList.end(); ++it )
    {
      cout << it->iBookKind << "\t" << it->iBookIndex << endl;
    }
}

list< Book_t >::iterator Seek( list< Book_t >::iterator it, int iSeekCnt )
{
    for( int iLoop = 0; iLoop < iSeekCnt; ++iLoop )
    {
      it++;
    }

    return it;
}

void main()
{
    // 现有图书
    const Book_t BookList[ ] =
    {
      COMPUTER, 0,
```

```
    COMPUTER, 1,
    ENGLISH,  0,
    CHEMISTRY,0,
    CHEMISTRY,  1,
    PHYSICS,  0
  };

  // 将现有图书插入书架
  list< Book_t >  BookCase( BookList, BookList + sizeof( BookList ) / sizeof( Book_t ) );

  // 一批新书,NOINDEX 为表示还没有插入到书架中
  const Book_t NewBookList[ ] =
  {
    COMPUTER, NOINDEX,
    CHEMISTRY,NOINDEX,
    ENGLISH,  NOINDEX,
    PHYSICS,  NOINDEX,
    COMPUTER, NOINDEX,
    PHYSICS,  NOINDEX
  };

  int iNewBookCnt = sizeof( NewBookList ) / sizeof( Book_t );

  list< Book_t >::iterator it;
  Book_t   stTempBook;

  for( int iLoop =0; iLoop < iNewBookCnt; ++iLoop )
  {
    stTempBook = NewBookList[ iLoop ];

    // 找到相同种类图书的第一个位置，在这里默认为每种图书都存在至少一本书
    it = find( BookCase.begin(), BookCase.end(), stTempBook );

    // 更新新插入的书在此类书中的位置
    stTempBook.iBookIndex = Book_t::KindCnt[ it->iBookKind ];

    // 指针偏移到下一类书的开头并将新书插入( 因为插入操作是前插入 )
    it = Seek( it, Book_t::KindCnt[ it->iBookKind ]++ );
    BookCase.insert( it, stTempBook );

  }

}
```

2. 关联式容器：

所谓关联式容器，观念上类似关联式数据库（实际上简单的多）：每个元素都有一个键值（Key）和一个实值（Value）。当元素被插入到关联式容器中时，容器内部结构便依照其键值的大小，以某种特定规则将这个元素放置于特定的位置。关联式容器没有所谓的头尾，所以不会有 push_back()、push_front()、pop_back()、pop_front()这样的操作。

关联式容器分为两大类：Map 和 Set 以及他们的衍生容器 MultiMap 和 MultiSet。他们的底层实现都是红黑树(RB-Tree)，因为红黑树提供了很好的搜索效率。

由于关联式底容器层实现也就是红黑树的实现稍稍有些复杂，并且为了保持良好的平衡性还需要进行各种旋转，在本教程里就不深入探讨容器的底层结构问题，只是简单列举几种常用的关联式容器的用法。、

2.1 Set

Set 的特性是，所有元素都会根据元素的键值自动被排序；Set 不能同时拥有实值（Value）和键值（Key），Set 元素中的实值就是键值，键值就是实值；Set 不允许两个元素拥有相同的键值。

2.1.1 常用构造函数：

```
set();
```
说明：默认构造函数
```
set< int > is;
```

```
set( const _set &_Right )
```
说明：拷贝构造函数
```
set< int > is1;
is1.insert( 0 );
is1.insert( 1 );
set< int > is2( is1 );
```

2.1.2 其他成员函数：

Size_type count(const key& _Key) const

说明：此函数返回 Key 值为_Key 的元素个数，实际上，由于在 set 中，相同 Key 值的元素只能存在一个，所以此返回值只有 1 和 0 两种情况。

```
Example：
        set< int > is;
                is.insert( 1 );
          is.insert( 1 );
        int iCnt = is.count( 1 );//iCnt = 1
        iCnt = is.count( 2 ); //iCnt = 0
```

pair< iterator, iterator > equal_range(const Key &_Key)

说明：返回比 Key 值大于或者等于_Key 的一组迭代子。对于返回值的 Pair 中，first 为 Key 值大于或者等于_Key 的迭代子，second 为 Key 值大于_Key 的迭代子。

```
Example:
        set< int > is;
        pair< set< int >::iterator, set< int >::iterator > Ret;
```

```
is.insert( 0 );
is.insert( 1 );
  is.insert( 2 );
  Ret = is.equal_range( 1 );//*Ret.first=1;*Ret.second=2;
  Ret = is.equal_range( 2 );//*Ret.first=2;Ret.second= is.end()
```

iterator find(const Key &_Key)

说明：查找一个 Key 为_Key 的元素，返回所查找元素的迭代子，如果查找失败，返回值为 end().

虽然通用的运算法中也有 find（）这个方法，但是对于自带 find（）方法的容器最好使用自带的方法，因为通用的 find（）没有针对容器做过优化，在效率方面劣于自带的方法。

Example:

```
set< int > is;
is.insert( 0 );
is.insert( 1 );
is.insert( 2 );
if( is.find( 1 ) != is.end() )
{
    Do something...
  }
```

pair< iterator, bool >insert(const value_type& Value)

说明：插入一个元素，如果要插入的元素的键值(key)已经存在，则返回值中的 bool 变量为 false，迭代子变量指向原元素存在的位置；如果插入元素的 key 不存在，则 bool 值为 true,返回新插入元素的迭代子。

Example：

```
set< int > is;
pair< set< int >::iterator, bool > Ret;
        Ret = is.insert( 1 ); //ret.second = true
        Ret = is.insert( 2 ); //ret.second = true
        Ret = is.insert( 1 ); //ret.second = false
```

key_compare key_comp() const;

说明：返回 set 的 key 比较函数，可以用来比较两个 Key 在 Set 中的相对位置关系。

Example：

```
set< int, less< int > > is1;//由小到大排列
set< int, less< int > >::key_compare kcl1 = is1.key_comp();
bool b = kcl1( 1, 2 ); // b = true
set< int, greater< int > > is2; //由小到大排列
set< int, greater< int > >::key_compare kcl2 = is2.key_comp();
b = kcl2( 1, 2 );      //b = false
```

value_compare value_comp() const;

说明：返回 set 的值比较函数，用来比较两个元素在 Set 中的相对位置关系，由于 Set 的 Value 和 Key 是同一个概念，所以这个方法实际上等同于 key_comp()

iterator lower_bound(const Key &_Key)

说明：返回大于或者等于_Key 的第一个元素的迭代子

Example：

```
set< int > is;
is.insert( 0 );
is.insert( 1 );
is.insert( 3 );
int i = *( is.lower_bound( 1 ) );// i=1
i = *( is.lower_bound( 2 ) );// i=3
```

iterator upper_bound(const Key &_Key)

说明：返回大于_Key 的第一个元素的迭代子

Example：

```
set< int > is;
is.insert( 0 );
is.insert( 1 );
is.insert( 3 );
int i = *( is.up_bound( 1 ) );// i=3
i = *( is. up_bound ( 3 ) );// i=无效值，因为不存在大于 3 的元素
```

2.2 multiset

multiset 的行为方式和 set 非常相似，set 的所有成员函数都可以在 multiset 中使用。只不过 multiset 允许存在 Key 值相同的元素。下面介绍几个 multiset 常用的方法

size_type count(const Key &_Key) const;

说明：在 set 中也有这个方法，只不过它在 multiset 中才能起到它的作用。返回 Key 值为_Key 的元素个数

Example:

```
multiset< int > is;
is.insert( 0 );
is.insert( 1 );
is.insert( 1 );
is.insert( 2 );
int iCnt = is.count( 1 ); //iCnt = 2
iCnt = is.count( 2 ); //iCnt = 1
```

Set 和 MultiSet 的综合示例：

set 和 multiset 的区别就是 multiset 允许 Key 值相同的元素存在而 set 不可以，这为我们的选择提供了一个依据

题目：制造一个彩票发生器，用户可以从中购买彩票。每个彩票的号码只有一个，号码的范围 0—MaxCnt。当用户选择的号码不存在的时候返回相应的信息。用户可以随时按照号码从小到大的顺序查看所有彩票的购买情况。

分析：因为每个彩票号码只允许存在一份，并且所有的彩票需要进行排序，所以我们选择的容器应该是 set 。

实现：

```
#pragma warning(disable: 4786)

#include <algorithm>
#include <iostream>
#include <set>
#include <stdlib.h>
#include <time.h>
using namespace std;

#define RANDOM     0

enum eReturnCode
{
    SUCCESS = 0,
    WRONG_NUMBER = SUCCESS + 1,
    NUMBER_EXIST = WRONG_NUMBER + 1,
    ALLNUMBER_EXIST = NUMBER_EXIST + 1
};

struct Lottery_t
{
    friend bool operator < ( const Lottery_t & _Left, const Lottery_t &_Right )
    {
        return ( _Left.iLotteryNum < _Right.iLotteryNum );
    }

    // 彩票号码
    int iLotteryNum;
    // 买了多少注
    int iLotteryCnt;
};

class CLotteryProduction
{
public:
    CLotteryProduction() : m_MaxNum( 10 )
    {
        srand( ( unsigned ) time( NULL ) );
    }
    eReturnCode BuyLottery( Lottery_t *pLottery,
```

```
                    int iLotteryNum = RANDOM,
                    int iBuyCnt = 1 )
{
    // 所有号码都已售出
    if( m_LotteryList.size() == m_MaxNum + 1 )
    {
        return ALLNUMBER_EXIST;
    }
    // 输入的号码不正确
    if( ( iLotteryNum > m_MaxNum ) || ( iLotteryNum < 0 ) )
    {
        return WRONG_NUMBER;
    }

    Lottery_t stLotteryForEntry;
    if( RANDOM == iLotteryNum )
    {
        // 产生一个随机号码,生成彩票
        for( ; ; )
        {
            stLotteryForEntry.iLotteryNum = GetRandomNum();
            stLotteryForEntry.iLotteryCnt = iBuyCnt;
            pair< set< Lottery_t >::iterator, bool > Ret =
                        m_LotteryList.insert( stLotteryForEntry );
            if( Ret.second )
            {
                break;
            }
        }
    }
    else
    {
        // 按照用户选择的号码生成彩票
        stLotteryForEntry.iLotteryNum = iLotteryNum;
        stLotteryForEntry.iLotteryCnt = iBuyCnt;
        pair< set< Lottery_t >::iterator, bool > Ret =
                        m_LotteryList.insert( stLotteryForEntry );
        if( Ret.second )
        {
            return NUMBER_EXIST;
        }
    }

    *pLottery = stLotteryForEntry;
```

```
        return SUCCESS;

    }

    void Display() const
    {
        set< Lottery_t >::const_iterator it;
        cout << "LotteryNo\t" << "LotteryCnt" << endl;
        for( it = m_LotteryList.begin(); it != m_LotteryList.end(); ++it )
        {
            cout << it->iLotteryNum << "\t" << it->iLotteryCnt << endl;
        }
    }
private:
    int GetRandomNum() const
    {
        return ( rand() % ( m_MaxNum + 1 ) );
    }
    set< Lottery_t > m_LotteryList;
    const int             m_MaxNum;
};
void main()
{

    CLotteryProduction LotteryFactory;
    eReturnCode Ret;
    for( ; ; )
    {
        Lottery_t stLottery;
        // 购买一张彩票
        Ret = LotteryFactory.BuyLottery( &stLottery );
        if( ALLNUMBER_EXIST == Ret )
            {
                break;
            }
        }

        //显示彩票的购买信息
        LotteryFactory.Display();
    }
```

2.3 map

Map 的特性是，所有元素都会根据元素的键值自动被排序。Map 的所有元素都是 pair，同时拥有实值

（Value）和键值（Key）。Pair 的第一元素被视为键值，第二元素被视为实值。Map 不允许元素拥有相同的键值。

Map 和 Set 的最大区别就是它的实值和键值分开。

常用构造函数：

map()

说明：默认构造函数

Example：

```
map< int, const char* > MemberList;
```

map(const map& _Right)

说明：拷贝构造函数

Example:

```
map< int, const char* > MemberList1;
MemberList1.insert( make_pair( 0, “Mike” ) );
map< int, const char* > MemberList2( MemberList1 );
```

其它常用成员函数：

iterator find(const Key &_Key);

说明：在 map 中寻找 Key 为_Key 的元素，返回找到元素的迭代子。

Example：

```
map< int, const char* > MemberList1;
MemberList1.insert( make_pair( 0, “Mike” ) );
MemberList1.insert( make_pair( 1, “Tom” ) );
MemberList1.find( 1 );//找到的是(0,”Mike”)这个元素
```

pair< iterator, bool > insert(const value_type &_Val)

说明：插入一个元素，如果此元素不存在，返回值 pair 中的 bool 值为 true,否则为 false，pair 中的 first 表示插入元素的迭代子。

Example：

```
map< int, string > MemberList1;
typedef map< int, string >::iterator MapIt;
pair< MapIt, bool > Ret;
Ret = MemberList1.insert( make_pair( 0, string(“Mike” ) ) );
//此时*Ret.first = (0,“Mike”), Ret.second = true;
Ret = MemberList1.insert( make_pair( 1, string (“Tom” )) );
//此时*Ret.first = (1,“Tom”), Ret.second = true;
Ret = MemberList1.insert( make_pair( 1, string(“Mary” )) );
//此时*Ret.first = (1,“Tom”), Ret.second = false;
```

key_compare key_comp() const;

说明：返回 Map 的 Key 值比较函数，用来判断两个 Key 值在 Map 中的相对位置。

Example：

```
map< int, string > MemberList;
map< int, string >::key_compare kcl;
kcl = MemberList.key_comp();
bool b = kcl( 1, 2 );// b = true
b = kcl( 2, 1 ); //b = false;
```

value_compare value_comp() const;

说明：返回 Map 的元素比较函数，用来判断两个元素在 Map 中的相对位置。

此函数实际上还是通过比较两个元素的 Key 值来确定它们的位置关系。

Example：

```
map< int, string > MemberList;
typedef map< int, string >::value_compare VALUE_COMP;
VALUE_COMP  vcl=MemberList. value_comp();
bool b = vcl ( make_pair( 0, string( “Mike” ) ),
               make_pair( 1, string( “Tom” ) ) );
//此时 b = true
b = vcl (make_pair( 1, string( “Tom” ) ),
         make_pair( 0, string( “Mike” ) ) );
//此时 b = false;
```

Type &operator [] （ const Key &_Key ）

说明：在 map 中也提供了 operator[]，此重载操作符用于改变下标为_Key 的元素的实值。

Example：

```
map< int, string > MemberList;
MemberList.insert( make_pair( 0, string(“Mike”) ) );
MemberList.insert( make_pair( 1, string(“Tom”) ) );
//此时 Key 为 1 的元素实值为”Tom”
map< int, string >::iterator it;
it = MemberList.find( 1 );
MemberList[ it->first ] = “Mary”;
//此时 Key 为 1 的元素实值为”Mary”
```

2.4 multimap

multimap 的行为方式与 map 几乎一致，只是 multimap 允许多于一个的元素拥有相同的键值。对于 multimap 在这里就不进行介绍了，除了 operator[]，map 的所有方法均可应用于 multimap。

multimap 和 map 的示例：

multimap 和 map 的区别就是 multimap 允许相同 Key 值得元素而 map 不允许，这给他们的选择制定了一个标准

题目：实现一个学生队列，用于给学生进行排队。排队的方式分为按身高排序，按体重排序和按年龄排序。要求可以随时改变此队列的排序方式，并能够进行打印

分析：这个题目中要求进行排序，并且能够随时改变排序的方式，所以应该选择 map 或者 multimap，这样改变排序方式应该更容易一些。同时，考虑到学生的属性（身高、体重和年龄）有可能一致，所以我们选择的容器应该是 multimap。

实现：

```
#pragma warning(disable: 4786)

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

enum eQueueKind
{
    QUEUE_BY_HIGH    = 0,
    QUEUE_BY_WEIGHT = QUEUE_BY_HIGH + 1,
    QUEUE_BY_AGE = QUEUE_BY_WEIGHT + 1,
};

struct Student_t
{
    friend bool operator > ( const Student_t & _Left, const Student_t &_Right )
    {
        return ( _Left.iHigh > _Right.iHigh );
    }
    int iHigh;
    int iWeight;
    int iAge;
  };

  class CStudentQueue
  {
  public:
    // 构造函数,接收的参数表明排序方法
    CStudentQueue(    eQueueKind    QueueKind    =    QUEUE_BY_HIGH    )    :
  m_QueueKind( QueueKind )
    {

    }

    // 向队列中加入学生
    void InsertStudent( const Student_t *pInputList, int iInputCnt )
    {
        switch( m_QueueKind )
```

```
    {
        case QUEUE_BY_HIGH:
            InsertByHigh( pInputList, iInputCnt );
            break;
        case QUEUE_BY_WEIGHT:
            InsertByWeight( pInputList, iInputCnt );
            break;
        case QUEUE_BY_AGE:
            InsertByAge( pInputList, iInputCnt );
            break;
        default:
            break;
    }
}

// 打印所有的学生信息
void Display() const
{
    cout << "High\t" << "Weight\t" << "Age" << endl;
    multimap< int, Student_t, greater< int > >::const_iterator it;
    for( it = m_StudentList.begin(); it != m_StudentList.end(); ++it )
    {
        cout << it->second.iHigh    << "\t"
             << it->second.iWeight  << "\t"
             << it->second.iAge     << endl;
    }

}

// 改变队列排序方式
void ConverQueueKind( eQueueKind QueueKind )
{
    if( QueueKind == m_QueueKind )
    {
        return;
    }
    multimap< int , Student_t, greater< int > > TempList;
    multimap< int , Student_t, greater< int > >::iterator it;

    // 根据传入的排序方法进行重新排序,并放入临时 Map 中
    switch( QueueKind )
    {
        case QUEUE_BY_HIGH:
        {
```

```
                for( it = m_StudentList.begin(); it != m_StudentList.end(); ++it )
                {
                    TempList.insert( make_pair( it->second.iHigh, it->second ) );
                }
                break;
            }
            case QUEUE_BY_WEIGHT:
            {
                for( it = m_StudentList.begin(); it != m_StudentList.end(); ++it )
                {
                    TempList.insert( make_pair( it->second.iWeight, it->second ) );
                }
                break;
            }
            case QUEUE_BY_AGE:
            {
                for( it = m_StudentList.begin(); it != m_StudentList.end(); ++it )
                {
                    TempList.insert( make_pair( it->second.iAge, it->second ) );
                }
                break;
            }
            default:
                break;
        }

        // 交换两个 Map 中的元素
        m_StudentList.swap( TempList );

        // 改变排序类型
        m_QueueKind = QueueKind;
    }
private:
    void InsertByHigh( const Student_t *pInputList, int iInputCnt )
    {
        for( int iLoop = 0; iLoop < iInputCnt; ++iLoop )
        {
            m_StudentList.insert( make_pair( pInputList->iHigh, *pInputList ) );
            pInputList++;
        }
    }
    void InsertByWeight( const Student_t *pInputList, int iInputCnt )
    {
        for( int iLoop = 0; iLoop < iInputCnt; ++iLoop )
```

```
        {
            m_StudentList.insert( make_pair( pInputList->iWeight, *pInputList ) );
            pInputList++;
        }
    }
    void InsertByAge( const Student_t *pInputList, int iInputCnt )
    {
        for( int iLoop = 0; iLoop < iInputCnt; ++iLoop )
        {
            m_StudentList.insert( make_pair( pInputList->iAge, *pInputList ) );
            pInputList++;
        }
    }
    eQueueKind                  m_QueueKind;
    multimap< int, Student_t, greater< int > > m_StudentList;
};

void main()
{
    Student_t StudentList1[] =
    {
        175, 70, 24,
        174, 80, 20,
        182, 90, 21,
        175, 65, 21,
        163, 60, 24,
    };
    Student_t StudentList2[] =
    {
        180, 70, 24,
        166, 50, 12,
        182, 90, 19,
        175, 65, 18,

    };

    // 构造一个队列,默认排序方式为按身高排序
    CStudentQueue StudentQueue;
    // 向队列中插入学生
    StudentQueue.InsertStudent( StudentList1, sizeof( StudentList1 ) / sizeof( Student_t ) );

    StudentQueue.Display();

    // 改变队列排序方式
```

```
    StudentQueue.ConverQueueKind( QUEUE_BY_WEIGHT );

    StudentQueue.InsertStudent( StudentList2, sizeof( StudentList2 ) / sizeof( Student_t ) );
    StudentQueue.Display();

}
```

五、写在后面

本教材只是简单了介绍了模板的概念以及集中常用容器的使用方法，对于 STL 的具体实现方法并没有涉及。STL（更深一步可以说是泛型编程）可以说是一个划时代的产物。最简单的要求是能够在实现程序的时候尽可能的考虑一下是否可以使用已有的 STL 中的方法，力求使用 STL 就像使用 C/C++固有的库函数一样得心应手，这样可以使你所实现的程序更加优美，效率也更高（这里的效率既指程序运行效率也包括编码的效率）。更进一步，如果有精力的话可以深入的了解一下 STL 实现的方法以及思想，这将使你的思路更为开阔，可能在有的时候你会有这样的感叹“原来还可以这么做”！

六、 附录：如何选择容器

以下文字摘自《Effective STL》，其中可能涉及到文章中没有提到的容器

- 你需要“可以在容器的任意位置插入一个新元素”的能力吗？如果是，你需要序列容器，关联容器做不到。
- 你关心元素在容器中的顺序吗？如果不，散列容器就是可行的选择。否则，你要避免使用散列容器。
- 必须使用标准 C++中的容器吗？如果是，就可以除去散列容器、slist 和 rope。
- 你需要哪一类迭代器？如果必须是随机访问迭代器，在技术上你就只能限于 vector、deque 和 string，但你也可能会考虑 rope。如果需要双向迭代器，你就用不了 slist 和散列容器的一般实现。
- 当插入或者删除数据时，是否非常在意容器内现有元素的移动？如果是，你就必须放弃连续内存容器。
- 容器中的数据的内存布局需要兼容 C 吗？如果是，你就只能用 vector。
- 查找速度很重要吗？如果是，你就应该看看散列容器，排序的 vector 和标准的关联容器——大概是这个顺序。
- 你介意如果容器的底层使用了引用计数吗？如果是，你就得避开 string，因为很多 string 的实现是用引用计数。你也不能用 rope，因为权威的 rope 实现是基于引用计数的。于是你得重新审核你的 string，你可以考虑使用 vector<char>。
- 你需要插入和删除的事务性语义吗？也就是说，你需要有可靠地回退插入和删除的能力吗？如果是，你就需要使用基于节点的容器。如果你需要多元素插入的事务性语义，你就应该选择 list，因为 list 是唯一提供多元素插入事务性语义的标准容器。事务性语义对于有兴趣写异常安全代码的程序员来说非常重要。
- 你要把迭代器、指针和引用的失效次数减到最少吗？如果是，你就应该使用基于节点的容器，因为在这些容器上进行插入和删除不会使迭代器、指针和引用失效（除非它们指向你删除的元素）。一般来说，在连续内存容器上插入和删除会使所有指向容器的迭代器、指针和引用失效。
- 你需要具有有以下特性的序列容器吗：1）可以使用随机访问迭代器；2）只要没有删除而且插入只发生在容器结尾，指针和引用的数据就不会失效？这个一个非常特殊的情况，但如果你遇到这种情况，

deque 就是你梦想的容器。（有趣的是，当插入只在容器结尾时，deque 的迭代器也可能会失效，deque 是唯一一个“在迭代器失效时不会使它的指针和引用失效”的标准 STL 容器。）