vector 的基本使用

```
1.
元素的添加
 iVec.push_back(2);

    /// 在指定位置的前面 插入一个
    iVec.insert(iVec.begin(), -1);
    iVec.insert(iVec.end(), 10);

    //插入指定范围的对象
    int nArray[3] = {11,12,13};
    iVec.insert(iVec.begin() + 1,&nArray[0], &nArray[3]);
    printVec(iVec);                                 // -1 11 12 13 1 2 10

    vector<int> tmp{1,2,3};
    iVec.insert(iVec.end(),tmp.begin(),tmp.end());
    printVec(iVec);                                 // -1 11 12 13 1 2 10 1 2 3

    // 排序
    sort(iVec.begin(),iVec.end());
    printVec(iVec);                                 // -1 1 1 2 2 3 10 11 12 13

    // 删除一个元素 iVec[1]
    iVec.erase(iVec.begin() + 1);
    printVec(iVec);                                 // -1 1 2 2 3 10 11 12 13


    // 去掉重复的元素
    iVec.push_back(1);
    iVec.push_back(1);
    vector<int>::iterator new_end = unique(iVec.begin(),iVec.end());
    iVec.erase(new_end,iVec.end());                 // //删除（真正的删除）重复的元
素
    copy(iVec.begin(),iVec.end(),ostream_iterator<int>(cout," "));  // -1 1
2 3 10 11 12 13 1

2. 如果需要重复给vector 添加元素，而空间已知的情况，用reserve
v.reserve(1000);

/// 字符串的处理
string s;

...
if (s.size() < s.capacity()) {
    s.push_back('x');
```

```
}

3. 强行释放容器的所有内存。有东西也被删除了。
vector<int>().swap(nums);  // 强行释放没有用到的内存

vector<int>(iVec).swap(iVec);        // 如果原来的iVec.capacity() 是16,  但是真实只有9个元素，处理完后,  iVec.capacity() = 9
```

1.文件包含:

首先在程序开头处加上#include以包含所需要的类文件vector

还有一定要加上using namespace std;

2.变量声明:

2.1 例:声明一个int向量以替代一维的数组:vector a;(等于声明了一个int数组a[],大小没有指定,可以动态的向里面添加删除)。

2.2 例:用vector代替二维数组.其实只要声明一个一维数组向量即可,而一个数组的名字其实代表的是它的首地址,所以只要声明一个地址的向量即可,即:vector a.同理想用向量代替三维数组也是一样,vector a;再往上面依此类推.

3- 方法总览：

1.push_back 在数组的最后添加一个数据 2.pop_back 去掉数组的最后一个数据 3.at 得到编号位置的数据 4.begin 得到数组头的指针 5.end 得到数组的最后一个单元+1的指针 6. front 得到数组头的引用 7.back 得到数组的最后一个单元的引用 8.max_size 得到vector最大可以是多大 9.capacity 当前vector分配的大小 10.size 当前使用数据的大小 11.resize 改变当前使用数据的大小，如果它比当前使用的大，者填充默认值 12.reserve 改变当前vecotr所分配空间的大小 13.erase 删除指针指向的数据项 14.clear 清空当前的vector 15.rbegin 将vector反转后的开始指针返回(其实就是原来的end-1) 16.rend 将vector反转构的结束指针返回(其实就是原来的begin-1) 17.empty 判断vector是否为空 18.swap 与另一个vector交换数据

4- 详细的函数实现功能

> c.clear() 移除容器中所有数据。 c.empty() 判断容器是否为空。 c.erase(pos) 删除pos位置的数据 c.erase(beg,end) 删除[beg,end)区间的数据 c.front() 传回第一个数据。 c.insert(pos,elem) 在pos位置插入一个elem拷贝 c.pop_back() 删除最后一个数据。 c.push_back(elem) 在尾部加入一个数据。 c.resize(num) 重新设置该容器的大小 c.size() 回容器中实际数据的个数。 c.begin() 返回指向容器第一个元素的迭代器 c.end() 返回指向容器最后一个元素的迭代器

5- reserve

例如，假定你想建立一个容纳1-1000值的vector。没有使用reserve，你可以像这样来做：

> vector v; for (int i = 1; i <= 1000; ++i) v.push_back(i);

在大多数STL实现中，这段代码在循环过程中将会导致2到10次重新分配。（10这个数没什么奇怪的。记住vector在重新分配发生时一般把容量翻倍，而1000约等于210。）

把代码改为使用reserve，我们得到这个：

> vector v; v.reserve(1000); /// 可是事先分配大小 for (int i = 1; i <= 1000; ++i) v.push_back(i);

这在循环中不会发生重新分配。

在大小和容量之间的关系让我们可以预言什么时候插入将引起vector或string执行重新分配，而且，可以预言什么时候插入会使指向容器中的迭代器、指针和引用失效。例如，给出这段代码，

```
string s;

...
if (s.size() < s.capacity()) {
    s.push_back('x');
}
```

push_back的调用不会使指向这个string中的迭代器、指针或引用失效，因为string的容量保证大于它的大小。如果不是执行push_back，代码在string的任意位置进行一个insert，我们仍然可以保证在插入期间没有发生重新分配，但是，与伴随string插入时迭代器失效的一般规则一致，所有从插入位置到string结尾的迭代器/指针/引用将失效。

6- 用swap方法强行释放STL Vector所占内存

有一种方法来把它从曾经最大的容量减少到它现在需要的容量。这样减少容量的方法常常被称为“收缩到合适（shrink to fit）”。该方法只需一条语句：vector(ivec).swap(ivec); 表达式vector(ivec)建立一个临时vector，它是ivec的一份拷贝：vector的拷贝构造函数做了这个工作。但是，vector的拷贝构造函数只分配拷贝的元素需要的内存，所以这个临时vector没有多余的容量。然后我们让临时vector和ivec交换数据，这时我们完成了，ivec只有临时变量的修整过的容量，而这个临时变量则持有了曾经在ivec中的没用到的过剩容量。在这里（这个语句结尾），临时vector被销毁，因此释放了以前ivec使用的内存，收缩到合适。

> template < class T> void ClearVector(vector& v) { vectorvtTemp; vtTemp.swap(v); }

如

```
    vector<int> v ;
    nums.push_back(1);
    nums.push_back(3);
    nums.push_back(2);
    nums.push_back(4);
    vector<int>().swap(v);

/* 或者v.swap(vector<int>()); */

/*或者{ std::vector<int> tmp = v;   v.swap(tmp);   }; //加大括号{ }是让tmp退出{
}时自动析构*/
```

7- Vector 内存管理成员函数的行为测试

C++ STL的vector使用非常广泛，但是对其内存的管理模型一直有多种猜测，下面用实例代码测试来了解其内存管理方式，测试代码如下：

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

int main()
{
    vector<int> iVec;
    cout << "容器 大小为: " << iVec.size() << endl;
    cout << "容器 容量为: " << iVec.capacity() << endl; //1个元素，  容器容量为1

    iVec.push_back(1);
    cout << "容器 大小为: " << iVec.size() << endl;
    cout << "容器 容量为: " << iVec.capacity() << endl; //2个元素，  容器容量为2

    iVec.push_back(2);
    cout << "容器 大小为: " << iVec.size() << endl;
    cout << "容器 容量为: " << iVec.capacity() << endl; //3个元素，  容器容量为4

    iVec.push_back(3);
    cout << "容器 大小为: " << iVec.size() << endl;
    cout << "容器 容量为: " << iVec.capacity() << endl; //4个元素，  容器容量为4

    iVec.push_back(4);
    iVec.push_back(5);
    cout << "容器 大小为: " << iVec.size() << endl;
    cout << "容器 容量为: " << iVec.capacity() << endl; //5个元素，  容器容量为8
```

```
    iVec.push_back(6);
    cout << "容器 大小为: " << iVec.size() << endl;
    cout << "容器 容量为: " << iVec.capacity() << endl; //6个元素,  容器容量为8
    iVec.push_back(7);
    cout << "容器 大小为: " << iVec.size() << endl;
    cout << "容器 容量为: " << iVec.capacity() << endl; //7个元素,  容器容量为8

    iVec.push_back(8);
    cout << "容器 大小为: " << iVec.size() << endl;
    cout << "容器 容量为: " << iVec.capacity() << endl; //8个元素,  容器容量为8

    iVec.push_back(9);
    cout << "容器 大小为: " << iVec.size() << endl;
    cout << "容器 容量为: " << iVec.capacity() << endl; //9个元素,  容器容量为16
/* vs2005/8 容量增长不是翻倍的, 如
    9个元素    容量9
    10个元素 容量13 */

/* 测试effective stl中的特殊的交换 swap() */
    cout << "当前vector 的大小为: " << iVec.size() << endl;        // 9
    cout << "当前vector 的容量为: " << iVec.capacity() << endl;    // 16
    vector<int>(iVec).swap(iVec);                                  // 9

    cout << "临时的vector<int>对象 的大小为: " << (vector<int>(iVec)).size()
<< endl;
    cout << "临时的vector<int>对象 的容量为: " << (vector<int>
(iVec)).capacity() << endl;
    cout << "交换后, 当前vector 的大小为: " << iVec.size() << endl;
    cout << "交换后, 当前vector 的容量为: " << iVec.capacity() << endl;

    return 0;
}
```

结果为:

```
容器 大小为: 0
容器 容量为: 0
容器 大小为: 1
容器 容量为: 1
容器 大小为: 2
容器 容量为: 2
容器 大小为: 3
容器 容量为: 4
容器 大小为: 5
容器 容量为: 8
容器 大小为: 6
```

```
容器 容量为： 8
容器 大小为： 7
容器 容量为： 8
容器 大小为： 8
容器 容量为： 8
容器 大小为： 9
容器 容量为： 16
当前vector 的大小为： 9
当前vector 的容量为： 16
临时的vector<int>对象 的大小为： 9
临时的vector<int>对象 的容量为： 9
交换后，当前vector 的大小为： 9
交换后，当前vector 的容量为： 9
```

8- vector的其他成员函数

c.assign(beg,end)：将[beg; end)区间中的数据赋值给c。 c.assign(n,elem)：将n个elem的拷贝赋值给c。 c.at(idx)：传回索引idx所指的数据，如果idx越界，抛出out_of_range。 c.back()：传回最后一个数据，不检查这个数据是否存在。 c.front()：传回地一个数据。 get_allocator：使用构造函数返回一个拷贝。 c.rbegin()：传回一个逆向队列的第一个数据。 c.rend()：传回一个逆向队列的最后一个数据的下一个位置。 c.~ vector ()：销毁所有数据，释放内存

9- vector的过程中的一些问题,特此列出讨论:

> 1)
>
> vector a;
>
> int b = 5;
>
> a.push_back(b);
>
> 此时若对b另外赋值时不会影响a[0]的值

2)

```
vector <int*> a;
 int *b;
 b= new int[4];
 b[0]=0;
 b[1]=1;
 b[2]=2;
 a.push_back(b);
 delete b;          //释放b的地址空间
```

```
for(int i=0 ; i <3 ; i++)
{
        cout<<a[0][i]<<endl;
}
```

此时输出的值并不是一开始b数组初始化的值，而是一些无法预计的值.

分析:根据1) 2)的结果,可以想到,在1)中, 往a向量中压入的是b的值,即a[0]=b,此时a[0]和b是存储在两个不同的地址中的.因此改变b的值不会影响a[0];

而在2)中,因为是把一个地址(指针)压入向量a,即a[0]=b,因此释放了b的地址也就释放了a[0]的地址,因此a[0]数组中存放的数值也就不得而知了. 【vector 中的保留未释放的指针，在Java 中会存在垃圾泄漏问题】

10- 其它比较常用的方法

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

using namespace std;

void printVec(vector<int>& vec ){
    for( int val: vec )
        cout << val << " ";

    cout << "\n" ;
}


int main(){
    vector<int> iVec;

    /// 在后面插入元素
    iVec.push_back(1);
    iVec.push_back(2);

    /// 在指定位置的前面 插入一个
    iVec.insert(iVec.begin(), -1);
    iVec.insert(iVec.end(), 10);

    //插入指定范围的对象
    int nArray[3] = {11,12,13};
```

```
    iVec.insert(iVec.begin() + 1,&nArray[0], &nArray[3]);
    printVec(iVec);                                 // -1 11 12 13 1 2 10

    vector<int> tmp{1,2,3};
    iVec.insert(iVec.end(),tmp.begin(),tmp.end());
    printVec(iVec);                                 // -1 11 12 13 1 2 10 1 2 3

    // 排序
    sort(iVec.begin(),iVec.end());
    printVec(iVec);                                 // -1 1 1 2 2 3 10 11 12 13

    // 删除一个元素 iVec[1]
    iVec.erase(iVec.begin() + 1);
    printVec(iVec);                                 // -1 1 2 2 3 10 11 12 13


    // 去掉重复的元素
    iVec.push_back(1);
    iVec.push_back(1);
    vector<int>::iterator new_end = unique(iVec.begin(),iVec.end());
    iVec.erase(new_end,iVec.end());                 // //删除（真正的删除）重复的元
素
    copy(iVec.begin(),iVec.end(),ostream_iterator<int>(cout," "));  // -1 1
2 3 10 11 12 13 1



    return 0;
}
```