* 没有**退路可言**、不要给自己找接口、以及**顾虑太多**
* 缺乏目的、目标、方法、效率再勤奋也只是一种时间的牺牲

@[toc]

# 常用算法

**概述**:

* 算法主要是由头文件<algorithm> <functional> <numeric>组成。
* <algorithm>是所有STL头文件中最大的一个，范围涉及到比较、 交换、查找、遍历操作、复制、修改等等
* <numeric>体积很小，只包括几个在序列上面进行简单数学运算的模板函数
* <functional>定义了一些模板类,用以声明函数对象。

## 遍历算法

**学习目标：**

* 掌握常用的遍历算法

**算法简介：**

* for\_each //遍历容器
* transform //搬运容器到另一个容器中

### for\_each

**功能描述：**

* 实现遍历容器

**函数原型：**

* for\_each(iterator beg, iterator end, \_func);
* // 遍历算法 遍历容器元素
* // beg 开始迭代器
* // end 结束迭代器
* // \_func 函数或者函数对象

**示例：**

#include <algorithm>  
#include <vector>  
  
//普通函数  
void print01(int val)   
{  
 cout << val << " ";  
}  
//函数对象  
class print02   
{  
 public:  
 void operator()(int val)   
 {  
 cout << val << " ";  
 }  
};  
  
//for\_each算法基本用法  
void test01() {  
  
 vector<int> v;  
 for (int i = 0; i < 10; i++)   
 {  
 v.push\_back(i);  
 }  
  
 //遍历算法  
 for\_each(v.begin(), v.end(), print01);  
 cout << endl;  
  
 for\_each(v.begin(), v.end(), print02());  
 cout << endl;  
}  
  
int main() {  
  
 test01();  
  
 system("pause");  
  
 return 0;  
}

**总结**：for\_each在实际开发中是最常用遍历算法，需要熟练掌握

### transform

**功能描述：**

* 搬运容器到另一个容器中

**函数原型：**

* transform(iterator beg1, iterator end1, iterator beg2, \_func);

//beg1 源容器开始迭代器

//end1 源容器结束迭代器

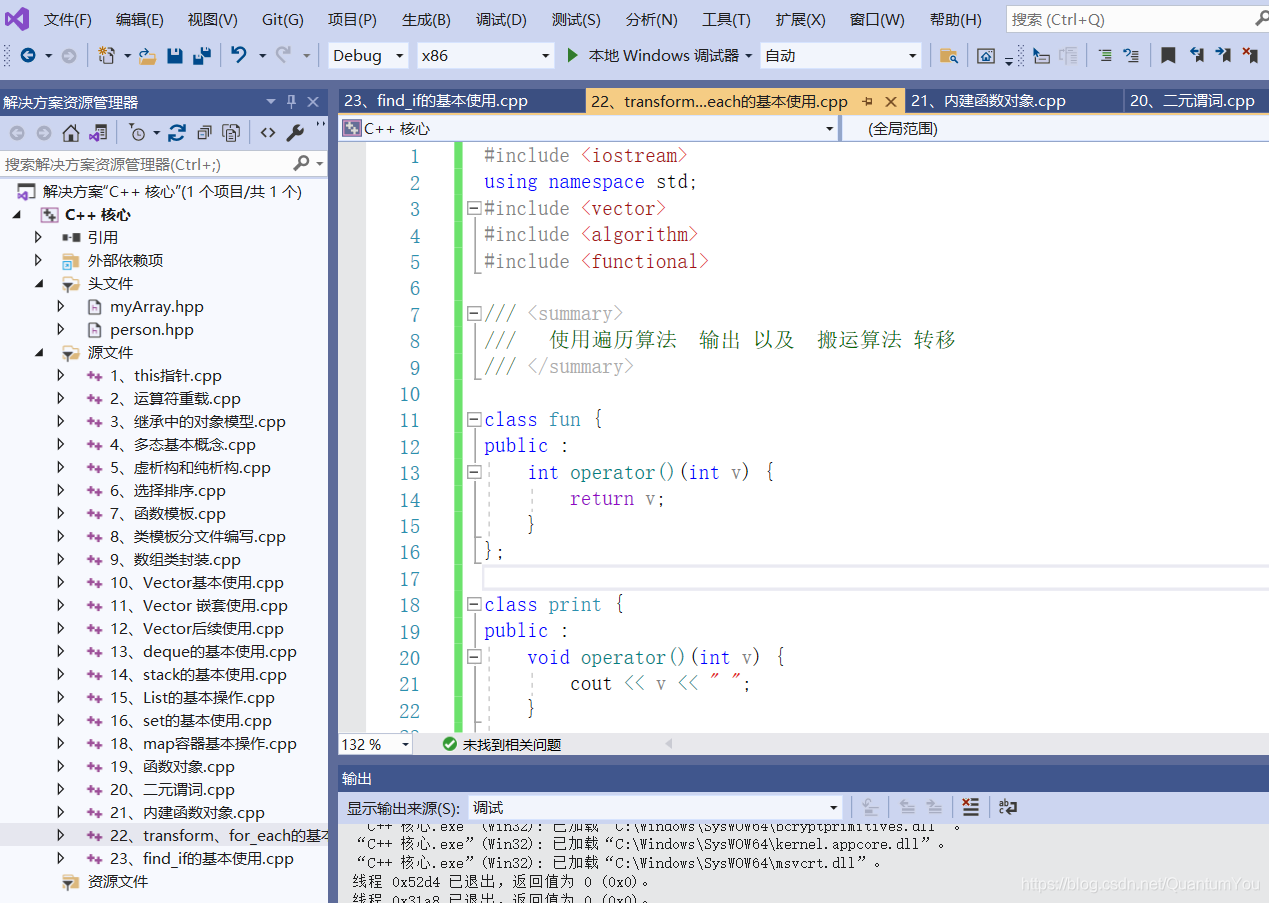
//beg2 目标容器开始迭代器

//\_func 函数或者函数对象

**示例：**

#include<vector>  
#include<algorithm>  
  
//常用遍历算法 搬运 transform  
  
class TransForm  
{  
public:  
 int operator()(int val)  
 {  
 return val;  
 }  
  
};  
  
class MyPrint  
{  
public:  
 void operator()(int val)  
 {  
 cout << val << " ";  
 }  
};  
  
void test01()  
{  
 vector<int>v;  
 for (int i = 0; i < 10; i++)  
 {  
 v.push\_back(i);  
 }  
  
 vector<int>vTarget; //目标容器  
  
 vTarget.resize(v.size()); // 目标容器需要提前开辟空间  
  
 transform(v.begin(), v.end(), vTarget.begin(), TransForm());  
  
 for\_each(vTarget.begin(), vTarget.end(), MyPrint());  
}  
  
int main() {  
  
 test01();  
  
 system("pause");  
  
 return 0;  
}

**总结：** 搬运的目标容器必须要提前开辟空间，否则无法正常搬运



## 常用查找算法

学习目标：

* 掌握常用的查找算法

**算法简介：**

* find //查找元素
* find\_if //按条件查找元素
* adjacent\_find //查找相邻重复元素
* binary\_search //二分查找法
* count //统计元素个数
* count\_if //按条件统计元素个数

### find

**功能描述：**

* 查找指定元素，找到返回指定元素的迭代器，找不到返回结束迭代器end()

**函数原型：**

* find(iterator beg, iterator end, value);
* // 按值查找元素，找到返回指定位置迭代器，找不到返回结束迭代器位置
* // beg 开始迭代器
* // end 结束迭代器
* // value 查找的元素

**示例：**

#include <algorithm>  
#include <vector>  
#include <string>  
void test01() {  
  
 vector<int> v;  
 for (int i = 0; i < 10; i++) {  
 v.push\_back(i + 1);  
 }  
 //查找容器中是否有 5 这个元素  
 vector<int>::iterator it = find(v.begin(), v.end(), 5);  
 if (it == v.end())   
 {  
 cout << "没有找到!" << endl;  
 }  
 else   
 {  
 cout << "找到:" << \*it << endl;  
 }  
}  
  
class Person {  
public:  
 Person(string name, int age)   
 {  
 this->m\_Name = name;  
 this->m\_Age = age;  
 }  
 //重载==  
 bool operator==(const Person& p)   
 {  
 if (this->m\_Name == p.m\_Name && this->m\_Age == p.m\_Age)   
 {  
 return true;  
 }  
 return false;  
 }  
  
public:  
 string m\_Name;  
 int m\_Age;  
};  
  
void test02() {  
  
 vector<Person> v;  
  
 //创建数据  
 Person p1("aaa", 10);  
 Person p2("bbb", 20);  
 Person p3("ccc", 30);  
 Person p4("ddd", 40);  
  
 v.push\_back(p1);  
 v.push\_back(p2);  
 v.push\_back(p3);  
 v.push\_back(p4);  
  
 vector<Person>::iterator it = find(v.begin(), v.end(), p2);  
 if (it == v.end())   
 {  
 cout << "没有找到!" << endl;  
 }  
 else   
 {  
 cout << "找到姓名:" << it->m\_Name << " 年龄: " << it->m\_Age << endl;  
 }  
}

总结： 利用find可以在容器中找指定的元素，返回值是**迭代器**

### find\_if

**功能描述：**

* 按条件查找元素

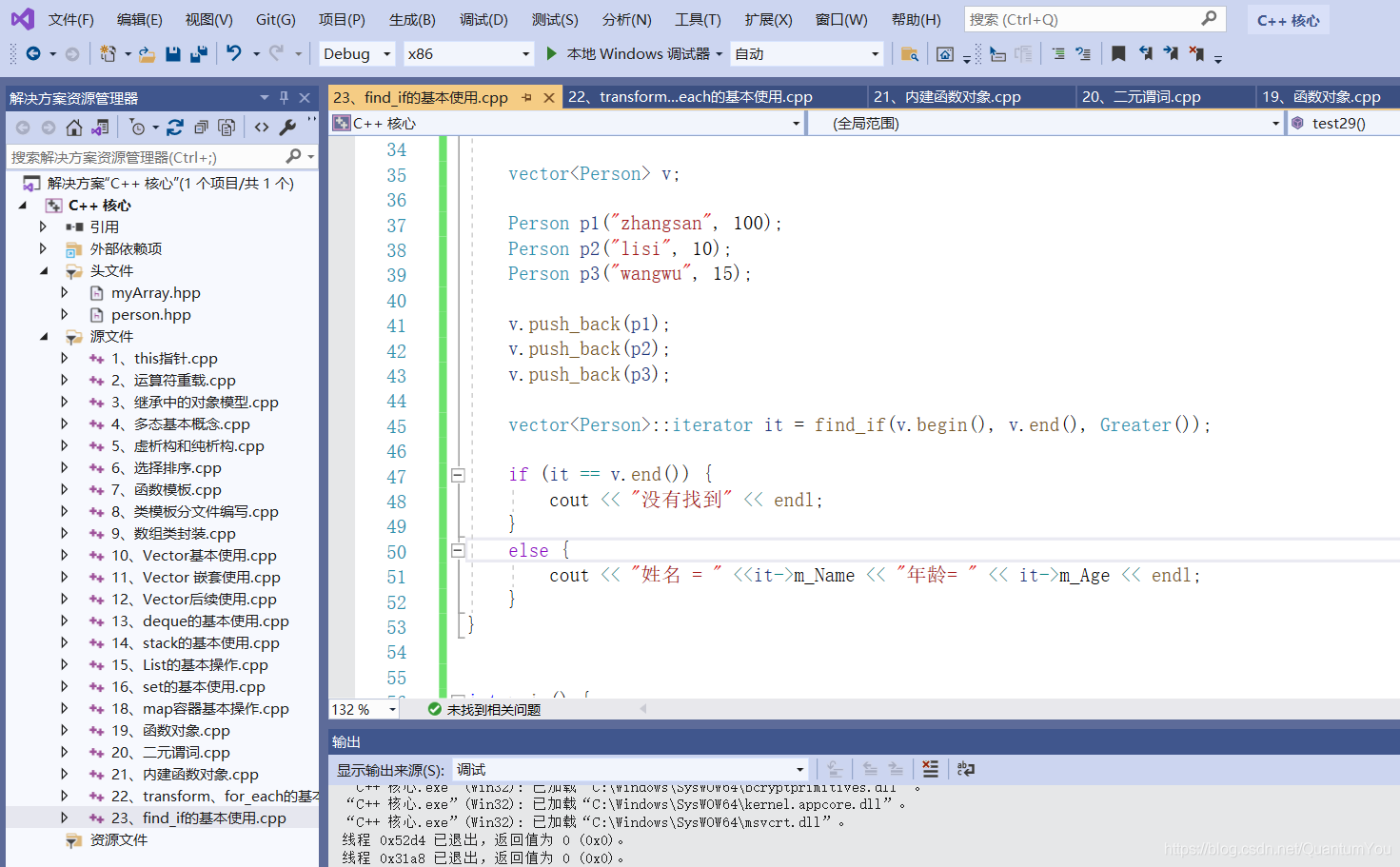
**函数原型：**

* find\_if(iterator beg, iterator end, \_Pred);
* // 按值查找元素，找到返回指定位置迭代器，找不到返回结束迭代器位置
* // beg 开始迭代器
* // end 结束迭代器
* // \_Pred 函数或者谓词（返回bool类型的仿函数）

**示例：**

#include <algorithm>  
#include <vector>  
#include <string>  
  
//内置数据类型  
class GreaterFive  
{  
public:  
 bool operator()(int val)  
 {  
 return val > 5;  
 }  
};  
  
void test01() {  
  
 vector<int> v;  
 for (int i = 0; i < 10; i++) {  
 v.push\_back(i + 1);  
 }  
  
 vector<int>::iterator it = find\_if(v.begin(), v.end(), GreaterFive());  
 if (it == v.end()) {  
 cout << "没有找到!" << endl;  
 }  
 else {  
 cout << "找到大于5的数字:" << \*it << endl;  
 }  
}  
  
//自定义数据类型  
class Person {  
public:  
 Person(string name, int age)  
 {  
 this->m\_Name = name;  
 this->m\_Age = age;  
 }  
public:  
 string m\_Name;  
 int m\_Age;  
};  
  
class Greater20  
{  
public:  
 bool operator()(Person &p)  
 {  
 return p.m\_Age > 20;  
 }  
  
};  
  
void test02() {  
  
 vector<Person> v;  
  
 //创建数据  
 Person p1("aaa", 10);  
 Person p2("bbb", 20);  
 Person p3("ccc", 30);  
 Person p4("ddd", 40);  
  
 v.push\_back(p1);  
 v.push\_back(p2);  
 v.push\_back(p3);  
 v.push\_back(p4);  
  
 vector<Person>::iterator it = find\_if(v.begin(), v.end(), Greater20());  
 if (it == v.end())  
 {  
 cout << "没有找到!" << endl;  
 }  
 else  
 {  
 cout << "找到姓名:" << it->m\_Name << " 年龄: " << it->m\_Age << endl;  
 }  
}  
  
int main() {  
  
 //test01();  
  
 test02();  
  
 system("pause");  
  
 return 0;  
}

**总结**：find\_if按条件查找使查找更加灵活，提供的仿函数可以改变不同的策略



### adjacent\_find

**功能描述：**

* 查找相邻重复元素

**函数原型：**

* adjacent\_find(iterator beg, iterator end);
* // 查找相邻重复元素,返回相邻元素的第一个位置的迭代器
* // beg 开始迭代器
* // end 结束迭代器

**示例：**

#include <algorithm>  
#include <vector>  
  
void test01()  
{  
 vector<int> v;  
 v.push\_back(1);  
 v.push\_back(2);  
 v.push\_back(5);  
 v.push\_back(2);  
 v.push\_back(4);  
 v.push\_back(4);  
 v.push\_back(3);  
  
 //查找相邻重复元素  
 vector<int>::iterator it = adjacent\_find(v.begin(), v.end());  
 if (it == v.end()) {  
 cout << "找不到!" << endl;  
 }  
 else {  
 cout << "找到相邻重复元素为:" << \*it << endl;  
 }  
}

总结：面试题中如果出现查找相邻重复元素，记得用STL中的adjacent\_find算法

### binary\_search

**功能描述：**

* 查找指定元素是否存在

**函数原型：**

* bool binary\_search(iterator beg, iterator end, value);
* // 查找指定的元素，查到 返回true 否则false
* // 注意: 在**无序序列中不可用**
* // beg 开始迭代器
* // end 结束迭代器
* // value 查找的元素

**示例：**

#include <algorithm>  
#include <vector>  
  
void test01()  
{  
 vector<int>v;  
  
 for (int i = 0; i < 10; i++)  
 {  
 v.push\_back(i);  
 }  
 //二分查找  
 bool ret = binary\_search(v.begin(), v.end(),2);  
 if (ret)  
 {  
 cout << "找到了" << endl;  
 }  
 else  
 {  
 cout << "未找到" << endl;  
 }  
}  
  
int main() {  
  
 test01();  
  
 system("pause");  
  
 return 0;  
}

**总结**: 二分查找法查找效率很高，值得注意的是查找的容器中元素必须的有序序列

### count

**功能描述：**

* 统计元素个数

**函数原型：**

* count(iterator beg, iterator end, value);
* // 统计元素出现次数
* // beg 开始迭代器
* // end 结束迭代器
* // value 统计的元素

**示例：**

#include <algorithm>  
#include <vector>  
  
//内置数据类型  
void test01()  
{  
 vector<int> v;  
 v.push\_back(1);  
 v.push\_back(2);  
 v.push\_back(4);  
 v.push\_back(5);  
 v.push\_back(3);  
 v.push\_back(4);  
 v.push\_back(4);  
  
 int num = count(v.begin(), v.end(), 4);  
  
 cout << "4的个数为： " << num << endl;  
}  
  
//自定义数据类型  
class Person  
{  
public:  
 Person(string name, int age)  
 {  
 this->m\_Name = name;  
 this->m\_Age = age;  
 }  
 bool operator==(const Person & p)  
 {  
 if (this->m\_Age == p.m\_Age)  
 {  
 return true;  
 }  
 else  
 {  
 return false;  
 }  
 }  
 string m\_Name;  
 int m\_Age;  
};  
  
void test02()  
{  
 vector<Person> v;  
  
 Person p1("刘备", 35);  
 Person p2("关羽", 35);  
 Person p3("张飞", 35);  
 Person p4("赵云", 30);  
 Person p5("曹操", 25);  
  
 v.push\_back(p1);  
 v.push\_back(p2);  
 v.push\_back(p3);  
 v.push\_back(p4);  
 v.push\_back(p5);  
   
 Person p("诸葛亮",35);  
  
 int num = count(v.begin(), v.end(), p);  
 cout << "num = " << num << endl;  
}  
int main() {  
  
 //test01();  
  
 test02();  
  
 system("pause");  
  
 return 0;  
}

**总结：** 统计自定义数据类型时候，需要配合重载 operator==

### count\_if

**功能描述：**

* 按条件统计元素个数

**函数原型：**

* count\_if(iterator beg, iterator end, \_Pred);
* // 按条件统计元素出现次数
* // beg 开始迭代器
* // end 结束迭代器
* // \_Pred 谓词

**示例：**

#include <algorithm>  
#include <vector>  
  
class Greater4  
{  
public:  
 bool operator()(int val)  
 {  
 return val >= 4;  
 }  
};  
  
//内置数据类型  
void test01()  
{  
 vector<int> v;  
 v.push\_back(1);  
 v.push\_back(2);  
 v.push\_back(4);  
 v.push\_back(5);  
 v.push\_back(3);  
 v.push\_back(4);  
 v.push\_back(4);  
  
 int num = count\_if(v.begin(), v.end(), Greater4());  
  
 cout << "大于4的个数为： " << num << endl;  
}  
  
//自定义数据类型  
class Person  
{  
public:  
 Person(string name, int age)  
 {  
 this->m\_Name = name;  
 this->m\_Age = age;  
 }  
  
 string m\_Name;  
 int m\_Age;  
};  
  
class AgeLess35  
{  
public:  
 bool operator()(const Person &p)  
 {  
 return p.m\_Age < 35;  
 }  
};  
void test02()  
{  
 vector<Person> v;  
  
 Person p1("刘备", 35);  
 Person p2("关羽", 35);  
 Person p3("张飞", 35);  
 Person p4("赵云", 30);  
 Person p5("曹操", 25);  
  
 v.push\_back(p1);  
 v.push\_back(p2);  
 v.push\_back(p3);  
 v.push\_back(p4);  
 v.push\_back(p5);  
  
 int num = count\_if(v.begin(), v.end(), AgeLess35());  
 cout << "小于35岁的个数：" << num << endl;  
}  
  
  
int main() {  
  
 //test01();  
  
 test02();  
  
 system("pause");  
  
 return 0;  
}

**总结** :按值统计用count，按条件统计用count\_if