

#### 知识点1【函数对象】（了解）

---

#### 知识点2【谓词】（了解）

---

1、一元谓词

---

2、二元谓词

---

#### 知识点3【内建函数对象】（了解）

---

#### 知识点4【适配器】（了解）

---

1、函数对象 适配器

---

2、函数指针 适配器 `ptr_fun`

---

3、成员函数 作为适配器 `mem_fun_ref`

---

4、取反适配器

---

#### 知识点5【常见遍历算法】（了解）

---

1、`for_each`遍历算法

---

2、`transform`算法

---

#### 知识点6【常见查找算法】（了解）

---

1、`find`算法 查找元素

---

案例1：查找基本类型数据

---

案例2：查找自定义类型数据

---

2、`find_if`算法 条件查找

---

案例：

---

3、`adjacent_find`算法 查找相邻重复元素

---

4、`binary_search`算法 二分查找法

---

5、`count`算法 统计元素出现次数

---

6、count\_if算法 统计元素出现次数

#### 知识点7【常用排序算法】

1、merge算法 容器元素合并

2、sort算法 容器元素排序

3、random\_shuffle算法 对指定范围内的元素随机调整次序

4、reverse算法 反转指定范围的元素

#### 知识点8【常见拷贝替换算法】

1、copy算法

2、replace算法

3、replace\_if算法

4、swap算法

#### 知识点9【常用算数生成算法】

1、accumulate算法 计算容器元素累计总和

2、fill算法 向容器中添加元素

#### 知识点10【常用集合算法】

1、set\_intersection求两个set集合的交集

2、set\_union算法 求两个set集合的并集

3、set\_difference算法 求两个set集合的差集

#### 知识点11【综合案例--竞技比赛】

## 知识点1【函数对象】（了解）

重载了函数调用运算符()的类 实例化的对象 就叫：函数对象

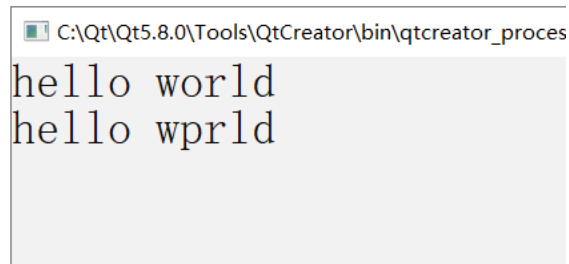
函数对象+ () 触发 重载函数调用运算符 执行 ==>类似函数调用 （仿函数）

```

class Print
{
public:
    void operator()(char *str)
    {
        cout<<str<<endl;
    }
};

void test01()
{
    Print ob;
    ob("hello world");
    Print()("hello wprld");
}

```



```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator_proces
hello world
hello wprld

```

如果函数对象 有一个参数 叫：一元函数对象

如果函数对象 有二个参数 叫：二元函数对象

如果函数对象 有三个参数 叫：多元函数对象

## 知识点2【谓词】（了解）

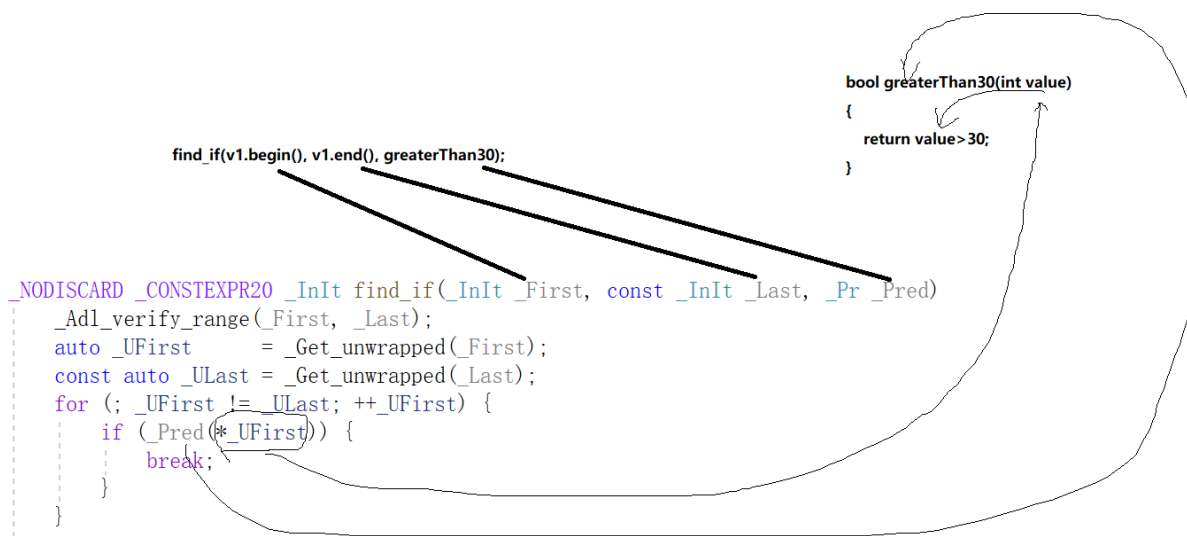
返回值为**bool**类型的普通函数或仿函数 都叫**谓词**。

如果谓词 有一个参数 叫：一元谓词

如果谓词 有二个参数 叫：二元谓词

### 1、一元谓词

拥有查找



```


1 #include<vector>
2 #include<algorithm>
3 bool greaterThan30(int value)

```

```

4 {
5     return value>30;
6 }
7 class GreaterThan30
8 {
9 public:
10     bool operator()(int value)
11     {
12         return value>30;
13     }
14 };
15
16 void test02()
17 {
18     vector<int> v1;
19     v1.push_back(10);
20     v1.push_back(30);
21     v1.push_back(50);
22     v1.push_back(70);
23     v1.push_back(90);
24
25     //find_if条件查找
26     vector<int>::iterator ret;
27     //普通函数提供策略 函数名
28     //ret = find_if(v1.begin(), v1.end(), greaterThan30);
29     //仿函数提供策略 类名称+ ()
30     ret = find_if(v1.begin(), v1.end(), GreaterThan30());
31     if(ret != v1.end())
32     {
33         cout<<"寻找的结果:"<<*ret<<endl;
34     }
35 }

```

 C:\Qt\Qt5.8.0\Tools\QtCreator\

寻找的结果:50

## 2、二元谓词

用户排序

```

1 bool myGreaterInt(int v1, int v2)
2 {
3     return v1>v2;
4 }
5 class MyGreaterInt
6 {
7 public:
8     bool operator()(int v1, int v2)
9     {
10         return v1>v2;
11     }
12 };
13 void test03()
14 {
15     vector<int> v1;
16     v1.push_back(10);
17     v1.push_back(50);
18     v1.push_back(30);
19     v1.push_back(90);
20     v1.push_back(70);
21
22     printVectorAll(v1);
23     //sort(v1.begin(), v1.end(), myGreaterInt);
24     sort(v1.begin(), v1.end(), MyGreaterInt());
25
26     printVectorAll(v1);
27 }

```

```

10 50 30 90 70
90 70 50 30 10

```

### 知识点3 【内建函数对象】（了解）

系统提供好的 函数对象

```

1 6个算数类函数对象,除了negate是一元运算,其他都是二元运算。
2 template<class T> T plus<T>//加法仿函数
3 template<class T> T minus<T>//减法仿函数
4 template<class T> T multiplies<T>//乘法仿函数

```

```

5  template<class T> T divides<T>//除法仿函数
6  template<class T> T modulus<T>//取模仿函数
7  template<class T> T negate<T>//取反仿函数
8  6个关系运算类函数对象,每一种都是二元运算。
9  template<class T> bool equal_to<T>//等于
10 template<class T> bool not_equal_to<T>//不等于
11 template<class T> bool greater<T>//大于
12 template<class T> bool greater_equal<T>//大于等于
13 template<class T> bool less<T>//小于
14 template<class T> bool less_equal<T>//小于等于
15 逻辑运算类运算函数,not为一元运算,其余为二元运算。
16 template<class T> bool logical_and<T>//逻辑与
17 template<class T> bool logical_or<T>//逻辑或
18 template<class T> bool logical_not<T>//逻辑非

```

```

void test03()
{
    vector<int> v1;
    v1.push_back(10);
    v1.push_back(50);
    v1.push_back(30);
    v1.push_back(90);
    v1.push_back(70);

    printVectorAll(v1);

    sort(v1.begin(), v1.end(), greater<int>() );

    printVectorAll(v1);
}

```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtc

```

10 50 30 90 70
90 70 50 30 10

```

```

void test04()
{
    vector<int> v1;
    v1.push_back(10);
    v1.push_back(30);
    v1.push_back(50);
    v1.push_back(70);
    v1.push_back(90);

    //find_if条件查找
    vector<int>::iterator ret;

    ret = find_if(v1.begin(), v1.end(), bind2nd(greater<int>(), 30) );
    if(ret != v1.end())
    {
        cout<<"寻找的结果:"<<*ret<<endl;
    }
}

```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator\_process\_stub.exe

寻找的结果:50

## 知识点4 【适配器】（了解）

适配器 为算法 提供接口。

### 1、函数对象 适配器

```

1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  using namespace std;
5  //第二步：公共继承binary_function 参数萃取
6  class printInt:public binary_function<int,int,void>
7  {
8  public:
9      //第三步：整个函数加const修饰
10     void operator()(int value, int tmp) const
11     {
12         cout<<"value="<<value<<" tmp="<< tmp<<endl;
13     }
14 };
15
16 void test01()
17 {
18     vector<int> v1;
19     v1.push_back(10);
20     v1.push_back(30);
21     v1.push_back(50);
22     v1.push_back(70);
23     v1.push_back(90);

```

```

24
25 //for_each 提取容器的每个元素
26 //第一步bind2nd 或bind1st
27 //bind2nd将100绑定到第二个参数tmp行 容器的元素在value上
28 for_each(v1.begin(), v1.end(), bind2nd(printInt(), 100) );
29 cout<<endl;
30 }
31
32 int main(int argc, char *argv[])
33 {
34     test01();
35     return 0;
36 }
37

```

```

value=10 tmp=100
value=30 tmp=100
value=50 tmp=100
value=70 tmp=100
value=90 tmp=100

```

```

//for_each 提取容器的每个元素
//第一步bind2nd 或bind1st
//bind2nd将100绑定到第二个参数tmp行 容器的元素在value上
//bind1st将100绑定到第一个参数value行 容器的元素在tmp上
for_each(v1.begin(), v1.end(), bind1st(printInt(), 100) );
cout<<endl;

```

```

value=100 tmp=10
value=100 tmp=30
value=100 tmp=50
value=100 tmp=70
value=100 tmp=90

```

## 2、函数指针 适配器 ptr\_fun

普通函数名 作为适配器



```

void myPrintInt(int value, int tmp)
{
    cout<<"value="<<value<<" tmp="<< tmp<<endl;
}
void test02()
{
    vector<int> v1;
    v1.push_back(10);
    v1.push_back(30);
    v1.push_back(50);
    v1.push_back(70);
    v1.push_back(90);

    //for_each 提取容器的每个元素
    for_each(v1.begin(), v1.end(), bind2nd(ptr_fun(myPrintInt),100) );
    cout<<endl;
}

```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\c

```

value=10 tmp=100
value=30 tmp=100
value=50 tmp=100
value=70 tmp=100
value=90 tmp=100

```

### 3、成员函数 作为适配器 mem\_fun\_ref

```

1  class Data
2  {
3  public:
4      int data;
5  public:
6      Data(){}
7      Data(int d){
8          data = d;
9      }
10     void printInt(int tmp)
11     {
12         cout<<"value="<<data+tmp<<endl;
13     }
14 };
15
16 void test03()
17 {
18
19     vector<Data> v1;
20     v1.push_back(Data(10));
21     v1.push_back(Data(30));
22     v1.push_back(Data(50));
23     v1.push_back(Data(70));
24     v1.push_back(Data(90));
25
26     //for_each 提取容器的每个元素
27     for_each(v1.begin(), v1.end(),
28         bind2nd(mem_fun_ref(&Data::printInt),100) );

```

```
28  cout<<endl;
29  }
```

C:\Qt\Qt5.8.0\Tools\QtCreator\k

```
value=110
value=130
value=150
value=170
value=190
```

## 4、取反适配器

### not1 一元取反

```
void test04()
{
    vector<int> v1;
    v1.push_back(10);
    v1.push_back(30);
    v1.push_back(50);
    v1.push_back(70);
    v1.push_back(90);

    vector<int>::iterator ret;
    ret = find_if(v1.begin(), v1.end(), not1(bind2nd(greater<int>(), 30)));
    if(ret != v1.end())
    {
        cout<<"找到相关数据:"<<*ret<<endl;
    }
}
```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator\_process\_stub.exe

找到相关数据:10

### not2 二元取反

```
1 void test05()
2 {
3     vector<int> v1;
4     v1.push_back(10);
5     v1.push_back(40);
6     v1.push_back(50);
7     v1.push_back(20);
8     v1.push_back(30);
9
10    //lambda 表达式 c++11才支持
11    //[ ]里面啥都不写 lambda不能识别 外部数据
12    //[=] lambda能对 外部数据 读操作
13    //[&] lambda能对 外部数据 读写操作
```

```

14  for_each(v1.begin(), v1.end(), [&](int val){
15      cout<<val<<" ";
16  } );
17  cout<<endl;
18
19  sort(v1.begin(), v1.end(), not2(greater<int>()));
20
21  for_each(v1.begin(), v1.end(), [&](int val){
22      cout<<val<<" ";
23  } );
24  cout<<endl;
25  }

```

```

10 40 50 20 30
10 20 30 40 50

```

## 知识点5 【常见遍历算法】（了解）

### 1、for\_each遍历算法

```

1  /*
2      遍历算法 遍历容器元素
3      @param beg 开始迭代器
4      @param end 结束迭代器
5      @param _callback 函数回调或者函数对象
6      @return 函数对象
7  */
8  for_each(iterator beg, iterator end, _callback);

```

### 2、transform算法

```

1  transform算法 将指定容器区间元素搬运到另一容器中
2      注意：transform 不会给目标容器分配内存，所以需要我们提前分配好内存
3      @param beg1 源容器开始迭代器
4      @param end1 源容器结束迭代器
5      @param beg2 目标容器开始迭代器
6      @param _callback 回调函数或者函数对象
7      @return 返回目标容器迭代器
8  */
9  transform(iterator beg1, iterator end1, iterator beg2, _callback);

```

```

1 int myTransInt01(int val)
2 {
3     return val;
4 }
5
6 void test01()
7 {
8     vector<int> v1;
9     v1.push_back(10);
10    v1.push_back(70);
11    v1.push_back(30);
12    v1.push_back(50);
13    v1.push_back(90);
14
15
16    vector<int> v2;
17    v2.resize(v1.size());
18    transform(v1.begin(), v1.end(), v2.begin(), myTransInt01);
19    printVectorInt(v2);
20 }

```

10 70 30 50 90

## 知识点6 【常见查找算法】（了解）

### 1、find算法 查找元素

```

1  /*
2      find算法 查找元素
3      @param beg 容器开始迭代器
4      @param end 容器结束迭代器
5      @param value 查找的元素
6      @return 返回查找元素的位置
7  */
8  find(iterator beg, iterator end, value)

```

#### 案例1：查找基本类型数据

```

void test02()
{
    vector<int> v1;
    v1.push_back(10);
    v1.push_back(70);
    v1.push_back(30);
    v1.push_back(50);
    v1.push_back(90);

    vector<int>::iterator ret;
    ret = find(v1.begin(), v1.end(), 30);
    if(ret != v1.end())
    {
        cout<<"查找的数据:"<<*ret<<endl;
    }
}

```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcr

查找的数据:30

## 案例2：查找自定义类型数据

```

1  #include<string>
2  class Person
3  {
4      friend void test03();
5  private:
6      int num;
7      string name;
8  public:
9      Person(){}
10     Person(int num, string name){
11         this->num = num;
12         this->name = name;
13     }
14     bool operator==(const Person &ob)
15     {
16         return ((this->num == ob.num) && (this->name == ob.name));
17     }
18 };
19
20 void test03()
21 {
22     vector<Person> v1;
23     v1.push_back(Person(100, "lucy"));
24     v1.push_back(Person(101, "bob"));
25     v1.push_back(Person(102, "tom"));

```

```

26
27 vector<Person>::iterator ret;
28 //find 查找自定义数据类型 需要重载==
29 ret = find(v1.begin(), v1.end(), Person(101,"bob"));
30 if(ret != v1.end())
31 {
32     cout<<"查找的数据:"<<(*ret).num<<" "<<(*ret).name<<endl;
33 }
34 }

```

查找的数据:101 bob

## 2、find\_if算法 条件查找

```

1  /*
2      @param beg 容器开始迭代器
3      @param end 容器结束迭代器
4      @param callback 回调函数或者谓词(返回bool类型的函数对象)
5      @return bool 查找返回true 否则false
6  */
7  find_if(iterator beg, iterator end, _callback);

```

案例:

```

1  void test04()
2  {
3      vector<int> v1;
4      v1.push_back(10);
5      v1.push_back(70);
6      v1.push_back(30);
7      v1.push_back(50);
8      v1.push_back(90);
9
10     vector<int>::iterator ret;
11     ret = find_if(v1.begin(), v1.end(), bind2nd(greater<int>(),30) );
12     if(ret != v1.end())
13     {
14         cout<<"查找的数据:"<<*ret<<endl;
15     }
16 }

```

查找的数据:70

### 3、adjacent\_find算法 查找相邻重复元素

```
1  /*
2      @param beg 容器开始迭代器
3      @param end 容器结束迭代器
4      @param _callback 回调函数或者谓词(返回bool类型的函数对象)
5      @return 返回相邻元素的第一个位置的迭代器
6  */
7  adjacent_find(iterator beg, iterator end, _callback);
```

```
void test05()
{
    vector<int> v1;
    v1.push_back(10);
    v1.push_back(20);
    v1.push_back(20);
    v1.push_back(30);
    v1.push_back(30);
    vector<int>::iterator ret;

    ret = adjacent_find(v1.begin(),v1.end());
    if(ret != v1.end())
    {
        cout<<"第一个相邻重复的元素:"<<*ret<<endl;
    }
}
```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator\_process\_stu  
第一个相邻重复的元素:20

### 4、binary\_search算法 二分查找法

```
1  /*
2      注意：在无序序列中不可用
3      @param beg 容器开始迭代器
4      @param end 容器结束迭代器
5      @param value 查找的元素
6      @return bool 查找返回true 否则false
7  */
8  bool binary_search(iterator beg, iterator end, value);
```

```

void test06()
{
    vector<int> v1;
    v1.push_back(10);
    v1.push_back(70);
    v1.push_back(30);
    v1.push_back(50);
    v1.push_back(90);
    sort(v1.begin(), v1.end());

    bool ret;
    ret = binary_search(v1.begin(), v1.end(), 30);
    if(ret)
    {
        cout<<"存在该数据"<<endl;
    }
}

```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator

存在该数据

## 5、count算法 统计元素出现次数

```

1  /*
2      @param beg 容器开始迭代器
3      @param end 容器结束迭代器
4      @param value
5      @return int返回元素个数
6  */
7  count(iterator beg, iterator end, value);

```

```

void test07()
{
    vector<int> v1;
    v1.push_back(10);
    v1.push_back(30);
    v1.push_back(70);
    v1.push_back(30);
    v1.push_back(90);

    cout<<count(v1.begin(), v1.end(), 30)<<endl;
}

```

C:\Qt\Qt5.8.0\Tools\C

2

## 6、count\_if算法 统计元素出现次数

```

1  /*
2  count_if算法 统计元素出现次数

```



```

3     @param beg 容器开始迭代器
4     @param end 容器结束迭代器
5     @param callback 回调函数或者谓词(返回bool类型的函数对象)
6     @return int返回元素个数
7  */
8  count_if(iterator beg, iterator end, _callback);

```

```

void test08()
{
    vector<int> v1;
    v1.push_back(10);
    v1.push_back(30);
    v1.push_back(70);
    v1.push_back(30);
    v1.push_back(90);

    cout<<count_if(v1.begin(), v1.end(), bind2nd(greater<int>(), 10))<<endl;
}

int main(int argc, char* argv[])

```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator\_process\_stub.exe

4

## 知识点7 【常用排序算法】

### 1、merge算法 容器元素合并

```

1  /*
2     merge算法 容器元素合并，并存储到另一容器中
3     注意:两个容器必须是有序的
4     @param beg1 容器1开始迭代器
5     @param end1 容器1结束迭代器
6     @param beg2 容器2开始迭代器
7     @param end2 容器2结束迭代器
8     @param dest 目标容器开始迭代器
9  */
10 merge(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest)

```

```

void test10()
{
    vector<int> v1;
    v1.push_back(10);
    v1.push_back(30);
    v1.push_back(50);

    vector<int> v2;
    v2.push_back(20);
    v2.push_back(40);
    v2.push_back(60);
    v2.push_back(70);
    v2.push_back(80);

    vector<int> v3;
    v3.resize(v1.size() + v2.size());
    merge(v1.begin(), v1.end(), v2.begin(), v2.end(), v3.begin());
    printVectorInt(v3);
}

```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator\_process\_stub.exe

10 20 30 40 50 60 70 80

## 2、sort算法 容器元素排序

```

1  /*
2      sort算法 容器元素排序
3      @param beg 容器1开始迭代器
4      @param end 容器1结束迭代器
5      @param _callback 回调函数或者谓词(返回bool类型的函数对象)
6  */
7  sort(iterator beg, iterator end, _callback)

```

## 3、random\_shuffle算法 对指定范围内的元素随机调整次序

```

1  /*
2      random_shuffle算法 对指定范围内的元素随机调整次序
3      @param beg 容器开始迭代器
4      @param end 容器结束迭代器
5  */
6  random_shuffle(iterator beg, iterator end)

```

```

#include<stdlib.h>
#include<time.h>
void test11()
{
    vector<int> v1;
    v1.push_back(10);
    v1.push_back(30);
    v1.push_back(50);
    v1.push_back(70);
    v1.push_back(90);
    printVectorInt(v1);
    //设置随机数种子
    srand(time(NULL));
    //洗牌
    random_shuffle(v1.begin(), v1.end());
    printVectorInt(v1);
}

```

```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcr
10 30 50 70 90
90 10 70 30 50

```

#### 4、reverse算法 反转指定范围的元素

```

1  /*
2      reverse算法 反转指定范围的元素
3      @param beg 容器开始迭代器
4      @param end 容器结束迭代器
5  */
6  reverse(iterator beg, iterator end)

```

```

void test12()
{
    vector<int> v1;
    v1.push_back(10);
    v1.push_back(30);
    v1.push_back(50);
    v1.push_back(70);
    v1.push_back(90);
    printVectorInt(v1);
    //洗牌
    reverse(v1.begin(), v1.end());
    printVectorInt(v1);
}

```

```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator_pro
10 30 50 70 90
90 70 50 30 10

```

## 知识点8 【常见拷贝替换算法】

### 1、copy算法

```

1  /*

```

```

2    copy算法 将容器内指定范围的元素拷贝到另一容器中
3    @param beg 容器开始迭代器
4    @param end 容器结束迭代器
5    @param dest 目标起始迭代器
6    */
7    copy(iterator beg, iterator end, iterator dest)

```

```

void test13()
{
    vector<int> v1;
    v1.push_back(10);
    v1.push_back(30);
    v1.push_back(50);
    v1.push_back(70);
    v1.push_back(90);
    vector<int> v2;
    v2.resize(v1.size());

    copy(v1.begin(), v1.end(), v2.begin());
    printVectorInt(v2);
}

```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator\_process\_stub.exe

10 30 50 70 90

copy提升:

```

#include<iterator>
void test14()
{
    vector<int> v1;
    v1.push_back(10);
    v1.push_back(30);
    v1.push_back(50);
    v1.push_back(70);
    v1.push_back(90);

    copy(v1.begin(), v1.end(), ostream_iterator<int>(cout, " "));
    cout<<endl;
}

```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator\_process\_stub.exe

10 30 50 70 90

## 2、replace算法

```

1    /*
2    replace算法 将容器内指定范围的旧元素修改为新元素
3    @param beg 容器开始迭代器
4    @param end 容器结束迭代器
5    @param oldvalue 旧元素
6    @param newvalue 新元素
7    */

```

```
8 replace(iterator beg, iterator end, oldvalue, newvalue)
```

```
void test15()
{
    vector<int> v1;
    v1.push_back(10);
    v1.push_back(30);
    v1.push_back(50);
    v1.push_back(70);
    v1.push_back(90);

    copy(v1.begin(), v1.end(), ostream_iterator<int>(cout, " ") );
    cout<<endl;

    replace(v1.begin(), v1.end(), 30, 3000);

    copy(v1.begin(), v1.end(), ostream_iterator<int>(cout, " ") );
    cout<<endl;
}

int main(int argc, char *argv[])
```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator\_process\_stub.exe

```
10 30 50 70 90
10 3000 50 70 90
```

### 3、replace\_if算法

```
1  /*
2      replace_if算法 将容器内指定范围满足条件的元素替换为新元素
3      @param beg 容器开始迭代器
4      @param end 容器结束迭代器
5      @param callback函数回调或者谓词(返回Bool类型的函数对象)
6      @param oldvalue 新元素
7  */
8  replace_if(iterator beg, iterator end, _callback, newvalue)
```

```
1  class GreaterThan30
2  {
3  public:
4      bool operator()(int value)
5      {
6          return value>30;
7      }
8  };
9
10 void test16()
11 {
12     vector<int> v1;
13     v1.push_back(10);
```

```

14  v1.push_back(30);
15  v1.push_back(50);
16  v1.push_back(70);
17  v1.push_back(90);
18
19  copy(v1.begin(), v1.end(), ostream_iterator<int>(cout, " ")) ;
20  cout<<endl;
21
22  replace_if(v1.begin(), v1.end(), GreaterThan30() , 3000);
23
24  copy(v1.begin(), v1.end(), ostream_iterator<int>(cout, " ")) ;
25  cout<<endl;
26  }

```

```

C:\Qt\Qt5.6\tools\qcreator\bin\qcreator_pro...
10 30 50 70 90
10 30 3000 3000 3000

```

## 4、swap算法

```

1  /*
2      swap算法 互换两个容器的元素
3      @param c1容器1
4      @param c2容器2
5  */
6  swap(container c1, container c2)

```

# 知识点9 【常用算数生成算法】

## 1、accumulate算法 计算容器元素累计总和

```

1  /*
2      accumulate算法 计算容器元素累计总和
3      @param beg 容器开始迭代器
4      @param end 容器结束迭代器
5      @param value累加值 （注意：求和完后 + value）
6  */
7  accumulate(iterator beg, iterator end, value)

```

## 2、fill算法 向容器中添加元素

```

1  /*
2      fill算法 向容器中添加元素

```

```

3    @param beg 容器开始迭代器
4    @param end 容器结束迭代器
5    @param value t填充元素
6    */
7    fill(iterator beg, iterator end, value)

```

```

void test17()
{
    vector<int> v1;
    v1.resize(5);

    fill(v1.begin(), v1.end(), 10);

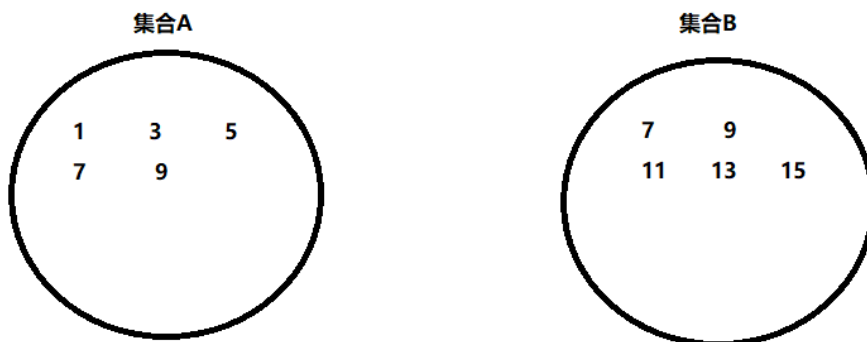
    copy(v1.begin(), v1.end(), ostream_iterator<int>(cout, " "));
    cout<<endl;
}

```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator\_process\_stub.exe

10 10 10 10 10

## 知识点10 【常用集合算法】



A和B的并集: 1 3 5 7 9 11 13 15

A和B的交集: 7 9

A 差集 B: 1 3 5

B 差集 A: 11 13 15

### 1、set\_intersection求两个set集合的交集

```

1    /*
2    set_intersection算法 求两个set集合的交集
3    注意:两个集合必须是有序序列
4    @param beg1 容器1开始迭代器
5    @param end1 容器1结束迭代器
6    @param beg2 容器2开始迭代器
7    @param end2 容器2结束迭代器
8    @param dest 目标容器开始迭代器
9    @return 目标容器的最后一个元素的迭代器地址
10   */

```

```

11  set_intersection(iterator beg1, iterator end1, iterator beg2, iterator e
nd2, \
12  iterator dest)

```

```

1  void test18()
2  {
3      vector<int> v1;
4      v1.push_back(1);
5      v1.push_back(3);
6      v1.push_back(5);
7      v1.push_back(7);
8      v1.push_back(9);
9
10     vector<int> v2;
11     v2.push_back(7);
12     v2.push_back(9);
13     v2.push_back(11);
14     v2.push_back(13);
15     v2.push_back(15);
16
17     vector<int> v3;//存放交集
18     v3.resize( min(v1.size(), v2.size()));
19
20     vector<int>::iterator ret;
21     ret = set_intersection(v1.begin(), v1.end(), v2.begin(),v2.end(), v3.be
gin());
22     copy(v3.begin(), ret, ostream_iterator<int>(cout, " ")) );
23     cout<<endl;
24 }
25

```

7 9

## 2、 set\_union算法 求两个set集合的并集

```

1  /*
2      set_union算法 求两个set集合的并集
3      注意:两个集合必须是有序序列
4      @param beg1 容器1开始迭代器

```



```

5      @param end1 容器1结束迭代器
6      @param beg2 容器2开始迭代器
7      @param end2 容器2结束迭代器
8      @param dest 目标容器开始迭代器
9      @return 目标容器的最后一个元素的迭代器地址
10     */
11     set_union(iterator beg1, iterator end1, iterator beg2, iterator end2, it
erator dest)
12

```

```

1 void test18()
2 {
3     vector<int> v1;
4     v1.push_back(1);
5     v1.push_back(3);
6     v1.push_back(5);
7     v1.push_back(7);
8     v1.push_back(9);
9
10    vector<int> v2;
11    v2.push_back(7);
12    v2.push_back(9);
13    v2.push_back(11);
14    v2.push_back(13);
15    v2.push_back(15);
16
17    vector<int> v3;
18    v3.resize( v1.size()+v2.size());
19
20    vector<int>::iterator ret;
21    ret = set_union(v1.begin(), v1.end(), v2.begin(),v2.end(), v3.begin());
22    copy(v3.begin(), ret, ostream_iterator<int>(cout, " ") );
23    cout<<endl;
24 }

```

C:\Qt\Qt5.8.0\Tools\QtCreator\bin\qtcreator\_p

1 3 5 7 9 11 13 15

### 3、set\_difference算法 求两个set集合的差集

```
1  /*
2      set_difference算法 求两个set集合的差集
3      注意:两个集合必须是有序序列
4      @param beg1 容器1开始迭代器
5      @param end1 容器1结束迭代器
6      @param beg2 容器2开始迭代器
7      @param end2 容器2结束迭代器
8      @param dest 目标容器开始迭代器
9      @return 目标容器的最后一个元素的迭代器地址
10 */
11 set_difference(iterator beg1, iterator end1, iterator beg2, iterator end
12 , iterator dest)
```

```
1 void test18()
2 {
3     vector<int> v1;
4     v1.push_back(1);
5     v1.push_back(3);
6     v1.push_back(5);
7     v1.push_back(7);
8     v1.push_back(9);
9
10    vector<int> v2;
11    v2.push_back(7);
12    v2.push_back(9);
13    v2.push_back(11);
14    v2.push_back(13);
15    v2.push_back(15);
16
17    vector<int> v3;
18    v3.resize( v1.size());
19
20    vector<int>::iterator ret;
21    ret = set_difference(v1.begin(), v1.end(), v2.begin(),v2.end(), v3.begi
22 n());
23    copy(v3.begin(), ret, ostream_iterator<int>(cout, " ") );
24    cout<<endl;
25 }
```

## 知识点11 【综合案例--竞技比赛】

某市举行一场跳水比赛，共有24个人参加。比赛共三轮，前两轮为淘汰赛，第三轮为决赛。

比赛方式：分组比赛，每组6个人；选手每次要随机分组，进行比赛；

第一轮分为4个小组，每组6个人。比如编号为: 100-123. 整体进行抽签

(draw) 后顺序演讲。当小组演讲完后，淘汰组内排名最后的三个选手，然后继续下一个小组的比赛。

第二轮分为2个小组，每组6人。比赛完毕，淘汰组内排名最后的三个选手，然后继续下一个小组的比赛。

第三轮只剩下1组6个人，本轮为决赛，选出前三名。 比赛评分：10个评委打分，去除最低、最高分，求平均分每个选手演讲完由10个评委分别打分。该选手的最终得分是去掉一个最高分和一个最低分，求得剩下的8个成绩的平均分。选手的名次按得分降序排列。

需求分析： 1) 产生选手（ABCDEFGHIJKLMNPOQRSTUVWXYZ）姓名、得分；选手编号

```
1 #include <iostream>
2 #include <string>
3 #include <vector>
4 #include <map>
5 #include <algorithm>
6 #include <stdlib.h>
7 #include <time.h>
8 #include <deque>
9 using namespace std;
10 class Player
11 {
12     friend void playGame(int index,vector<int> &v, map<int,Player> &m, vect
or<int> &v1);
13 private:
14     int num;
15     string name;
16     float score[3];
17 public:
18     Player(){}
19     Player(int num, string name)
```

```

20  {
21      this->num = num;
22      this->name = name;
23  }
24  };
25  void createPlayer(vector<int> &v, map<int,Player> &m)
26  {
27      int i=0;
28      string seedName="ABCDEFGHJKLMNOPQRSTUVWXYZ";
29      for(i=0; i<24; i++)
30      {
31          int num = 100;
32          num = num+i;
33          string tmpName = "选手";
34          tmpName += seedName[i];
35
36          v.push_back(num);
37          m.insert(make_pair(num, Player(num,tmpName)));
38      }
39  }
40  void playGame(int index,vector<int> &v, map<int,Player> &m, vector<int>
&v1)
41  {
42      //选手编号随机分组
43      srand(time(NULL));
44      random_shuffle(v.begin(), v.end());
45
46      //每名选手比赛
47      multimap<float, int, greater<float>> mul;//存放每组的分数--编号
48      int count = 0;
49      vector<int>::iterator it=v.begin();
50
51      cout<<"-----第"<<index<<"轮比赛-----"<<endl;
52      for(;it!=v.end(); it++)
53      {
54          count++;
55          //定义deque容器 存放评委打分
56          deque<float> d;
57          int i=0;
58          for(i=0;i<10;i++)

```

```

59  {
60  d.push_back( (float)(rand()%41+60) );
61  }
62  //排序
63  sort(d.begin(),d.end());
64  //去掉最高、最低分
65  d.pop_back();
66  d.pop_front();
67  //求平均分
68  float avg = accumulate(d.begin(),d.end(), 0)/d.size();
69  //将平均分 赋值给m中选手
70  m[*it].score[index-1] = avg;
71  mul.insert(make_pair(avg, *it));
72
73  if(count%6 == 0)//刚好一组
74  {
75  //分析竞技名单
76  cout<<"\t第"<<count/6<<"组的晋级名单:"<<endl;
77  int i=0;
78  multimap<float, int, greater<float>>::iterator mit=mul.begin();
79  for(i=0;i<3;i++,mit++)
80  {
81  v1.push_back( (*mit).second);
82  cout<<"\t\t"<<(*mit).second<<" "<<(*mit).first<<endl;
83  }
84  //打印当前组的得分情况
85  cout<<"\t第"<<count/6<<"组的得分情况:"<<endl;
86  mit=mul.begin();
87  for(i=0;i<6;i++, mit++)
88  {
89  int num = (*mit).second;
90  cout<<"\t\t"<<num<<" "<<m[num].name<<" "<<m[num].score[index-1]<<endl;
91  }
92
93
94  mul.clear();
95  }
96
97  }
98  }

```

```
99
100 int main(int argc, char *argv[])
101 {
102     vector<int> v;//存放选手编号
103     map<int,Player> m;//存放编号--选手信息
104
105     //创建选手
106     createPlayer(v, m);
107
108     //比赛
109     vector<int> v1;//存放晋级的编号
110     playGame(1,v,m, v1);
111
112     vector<int> v2;//存放晋级的编号
113     playGame(2,v1,m, v2);
114
115     vector<int> v3;//存放晋级的编号
116     playGame(3,v2,m, v3);
117     return 0;
118 }
```

-----第1轮比赛-----

第1组的晋级名单:

116 89

102 80

120 80

第1组的得分情况:

116 选手Q 89

102 选手C 80

120 选手U 80

109 选手J 79

104 选手E 71

105 选手F 69

第2组的晋级名单:

111 83

100 80

107 80

第2组的得分情况:

111 选手L 83

100 选手A 80

107 选手H 80

117 选手R 78

122 选手W 77

103 选手D 74

第3组的晋级名单:

106 86

115 83