仿函数

函数对象 / 仿函数

1.重载**函数调用操作符**的类，其**对象**常称为函数对象(function object)，也叫仿函数(functor)，使得类对象可以像函数那样调用

2.STL提供的算法往往有两个版本，一种是按照我们常规默认的运算来执行，另一种允许用户自己定义一些运算或操作，通常通过回调函数或模版参数的方式来实现，此时functor便派上了用场，特别是**作为模版参数的时候，只能传类型**

3.函数对象超出了普通函数的概念，其内部可以拥有自己的状态(其实也就相当于函数内的static变量)，可以通过成员变量的方式被记录下来

4.函数对象可以作为**函数的参数**传递

5.函数对象通常不定义构造和析构函数，所以在构造和析构时不会发生任何问题，避免了函数调用时的运行时问题

> 析构函数是一个成员函数，在对象超出范围或通过调用 delete 显式销毁对象时，会自动调用析构函数。析构函数具有与类相同的名称，前面是波形符 (~)。例如，声明 String 类的析构函数：
> ~String ()

6.模版函数对象使函数对象具有**通用性**，这也是它的优势之一

7.STL需要我们提供的functor通常只有一元和二元两种

8.**lambda 表达式**的内部实现其实也是仿函数

9.谓词：返回值为bool的普通函数或者函数对象，也就是我们离散数学中学习的predictor，比较常用的是一元谓词和二元谓词

内建函数对象

使用时需要包含头文件 `<functional>`
STL 内建了一些函数对象，分为：
算术类函数对象

```
template<class T> T plus<T>; // 加法仿函数 +
template<class T> T minus<T>; // 减法仿函数 -
template<class T> T multiplies<T>; // 乘法仿函数 *
template<class T> T divides<T>; // 除法仿函数 /
template<class T> T modulus<T>; // 取模仿函数 %
template<class T> T negate<T>; // 取反函数 -
// negate 是一元运算，其他都是二元运算。
```

关系运算类函数对象

```
template<class T> bool equal_to<T>; // 等于 =
template<class T> bool not_equal_to<T>; // 不等于 !=
template<class T> bool greater<T>; // 大于 >
template<class T> bool greater_equal<T>; // 大于等于 >=
template<class T> bool less<T>; // 小于 <
template<class T> bool less_equal<T>; // 小于等于 <=
```

逻辑运算类函数对象

```
template<class T> bool logical_and<T>; // 逻辑与 &
template<class T> bool logical_or<T>; // 逻辑或 |
template<class T> bool logical_not<T>; // 逻辑非 Not
```

适配器

函数对象适配器
函数对象适配器是一种用于修改函数对象的适配器，可以将一个函数对象转换为另一个函数对象。12

在 C++ 中，有三种常见的函数对象适配器：bind1st、bind2nd 和 not1。其中，bind1st 和 bind2nd 可以将一个二元函数对象转换为一个一元函数对象，not1 可以将一个一元函数对象转换为一个取反的一元函数对象。2

```cpp
#include <iostream>
#include <functional>

using namespace std;

int main() {
    // 定义一个二元函数对象
    auto f = [](int a, int b) -> int {
        return a + b;
    };

    // 将第一个参数绑定为 1，生成一个一元函数对象
    auto g = bind1st(f, 1);

    // 输出 g(2) 的结果
    cout << g(2) << endl; // 输出 3

    return 0;
}
```

例子

函数对象适配器
假设有原程序如下：

```cpp
#include <iostream>
#inlcude <algorithm>
#include <vector>
using namespace std;

class myPrint
{
public:
    void operator()(int val) { cout << val << endl; }
}

int main()
{
    vector<int> v;
    for (int i = 0; i < 10; i++) v.push_back(i);
    for_each(v.begin(), v.end(), myPrint());
    return 0;
}
```

若我们希望在每个数据输出的时候加上一个基值，并且该基值由用户输入
则可以修改为：

```cpp
#include <iostream>
#inlcude <algorithm>
#include <vector>
#include <functional>
using namespace std;

class myPrint: public binary_function<int, int, void>
// 2.做继承 参数1类型 + 参数2类型 + 返回值类型 binary_function
{
public:
    void operator()(int val, int base) const // 3. 加const，和父类保持一致
    {
        cout << val + base << endl;
    }
}

int main()
{
    vector<int> v;
    for (int i = 0; i < 10; i++) v.push_back(i);
    int n;
    cin >> n;
    for_each(v.begin(), v.end(), bind2nd(myPrint(), n));
    // 1. 将参数进行绑定 bind2nd
    // bind1st 功能类似，不过n会被绑定到第一个参数中
    return 0;
};
```

取反适配器:

原程序:

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

class GreaterThanFive
{
public:
    bool operator()(int val) { return val > 5; }
}

int main()
{
    vector<int> v;
    for (int i = 0; i < 10; i++) v.push_back(i);

    vector<int>::iterator pos = find_if(v.begin(), v.end(), GreaterThanFive());

    if (pos != v.end()) cout << *pos << endl;
    return 0;
}
```

我们希望找第一个不大于5的数，但又不想再写一个LessEqualThanFive
修改为：

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

class GreaterThanFive: public unary_function<int, bool>
// 2. 做继承 参数类型 + 返回值类型 unary_function
{
public:
    bool operator()(int val) const // 3.加 const
    {
        return val > 5;
    }
}

int main()
{
    vector<int> v;
    for (int i = 0; i < 10; i++) v.push_back(i);

    vector<int>::iterator pos = find_if(v.begin(), v.end(), not1(GreaterThanFive()));  //1. 一元

    if (pos != v.end()) cout << *pos << endl;
    return 0;
}
```

函数指针适配器:
沿用函数对象适配器的例子，假设 myPrint 是一个全局函数

```cpp
for_each(v.begin(), v.end(), bind2nd(ptr_fun(myPrint), n));
// 函数指针适配器 ptr_fun 将函数指针适配成仿函数
```

成员函数适配器:
我们假设有一个 Dog类 ， Dog类 内部有一个 bark() 成员方法，有一个 装满了Dog的vector 叫做 v

```cpp
for_each(v.begin(), v.end(), mem_fun_ref(&Dog::bark));
// 成员函数适配器 mem_fun_ref
// 如果容器中存放的不是对象实体，而是对象指针时，则需使用 ptr_fun
```

偏函数

对于一个多参数的函数，在某些应用场景下，它的**一些参数往往取固定值**，可以针对这样的函数，生成一个**新函数**，该新函数不包含原函数中已指定固定值的参数。（partial function application, 偏函数）

> 偏函数可缩小一个函数的适用范围，提高函数的针对性

例如对于 `void print(int n, int base); // 按base进制来输出n`
我们想要固定为十进制输出n，则可以修改其为偏函数如下：

```
#include <functional>
using namespace std;
using namespace std::placeholders;

void print(int n, int base);

function<void(int)> print10 = bind(print, _1, 10);
print10(23); //相当于 print(23, 10)
```

> function类和bind的使用需要c++11标准

算法(Algorithm)

算法主要由头文件 `<algorithm><functional><numeric>` 组成，其中：
`<algorithm>` 是所有STL头文件中最大的一个，其中常用的功能涉及到比较、交换、查找、遍历、复制、修改、反转、排序、合并等

`<numeric>` 体积很小，只包括在几个序列容器上进行简单运算的模版函数

`<functional>` 定义了一些模版类，用以声明函数对象

自定义的类如果想要直接使用算法库，则需补全默认构造函数、拷贝构造函数、析构函数、赋值操作符、小于操作符、等于操作符

常用遍历算法

for_each:

```
/**
  * 遍历算法 遍历容器元素
  * @param beg 开始迭代器
  * @param end 结束迭代器
  * @param _callback 函数回调或者函数对象
  * @return 函数对象
*/
for_each(iterator beg, iterator end, _callback);

# include <iostream>
<p class="mume-header " id="include-iostream"></p>

# include <vector>
<p class="mume-header " id="include-vector"></p>

# include <algorithm>
<p class="mume-header " id="include-algorithm"></p>


using namespace std;

void print(int i) {
    cout << i << " ";
}

int main() {
    vector<int> v = {1, 2, 3, 4, 5};

    // 使用 for_each 算法输出 v 中的每个元素
    for_each(v.begin(), v.end(), print); // 输出 1 2 3 4 5

    return 0;
}
```

transform:

```
/**
  * transform算法 将指定容器内的元素搬运到另一个容器中
  * 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
  * @param beg1 源容器开始迭代器
  * @param end1 源容器结束迭代器
  * @param beg2 目标容器开始迭代器
  * @param _callback 回调函数或者函数对象
  * @return 返回目标容器迭代器
*/
iterator transform(iterator beg1, iterator end1, iterator beg2, _callback);
```

> 注意：目标容器一定要提前分配好内存。

常用查找算法

find:

```
/**
  * find 算法 查找元素
  * @param beg 容器开始迭代器
  * @param end 容器结束迭代器
  * @param value 查找的元素
  * @return 返回查找元素的位置
*/
iterator find(iterator beg, iterator end, value);
```

find_if:

```
/**
  * find_if 算法 条件查找
  * @param beg 容器开始迭代器
  * @param end 容器结束迭代器
  * @param _callback 回调函数或者谓词(返回 bool 类型的函数对象)
  * @return 返回查找元素的位置
*/
iterator find_if(iterator beg, iterator end, _callback);

# include <iostream>
<p class="mume-header " id="include-iostream-1"></p>

# include <vector>
<p class="mume-header " id="include-vector-1"></p>

# include <algorithm>
<p class="mume-header " id="include-algorithm-1"></p>


using namespace std;

bool is_odd(int n) {
    return n % 2 == 1;
}

int main() {
    vector<int> v = {1, 2, 3, 4, 5};

    // 使用 find_if 算法查找 v 中第一个奇数
    auto it = find_if(v.begin(), v.end(), is_odd);

    // 输出查找结果
    if (it != v.end()) {
        cout << "Found " << *it << " at position " << it - v.begin() << endl;
    } else {
        cout << "Not found" << endl;
    }

    return 0;
}
```

> 利用find_if实现自定义类的find操作的时候，之前的函数适配器可能会派上用场。

adjacent_find:

```
/**
  * adjacent_find 算法 查找相邻重复元素
  * @param beg 容器开始迭代器
  * @param end 容器结束迭代器
  * @param _callback 回调函数或者谓词(返回 bool 类型的函数对象)
  * @return 返回相邻元素的第一个位置的迭代器
*/
iterator adjacent_find(iterator beg, iterator end, _callback);
```

binary_search:

```
/**
  * binary_search 算法 二分法查找
  * 注意：在无序序列中不可用
  * @param beg 容器开始迭代器
  * @param end 容器结束迭代器
  * @param value 查找的元素
  * @return bool 查找返回true，否则false
*/
bool binary_search(iterator beg, iterator end, value);
```

计数算法

count:

```
/**
  * count 算法 统计元素出现次数
  * @param beg 容器开始迭代器
  * @param end 容器结束迭代器
  * @param value 待计数的元素
  * @return int 返回元素个数
*/
int count(iterator beg, iterator end, value);
```

count_if:

```
/**
  * count_if 算法 统计元素出现次数
  * @param beg 容器开始迭代器
  * @param end 容器结束迭代器
  * @param _callback 回调函数或者谓词
  * @return int 返回元素个数
*/
int count_if(iterator beg, iterator end, _callback);
```

常用排序算法

sort:

```
/**
  * sort 算法 容器元素排序
  * @param beg 容器开始迭代器
  * @param end 容器结束迭代器
  * @param _callback 回调函数或者谓词
*/
sort(iterator beg, iterator end, _callback);
```

merge:

```
/**
  * merge 算法 容器元素合并，并储存到另一个容器中
  * 注意：两个容器必须是有序的
  * @param beg1 容器1开始迭代器
  * @param end1 容器1结束迭代器
  * @param beg2 容器2开始迭代器
  * @param end2 容器2结束迭代器
  * @param dest 目标容器开始迭代器
*/
merge(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest);
```

random_shuffle:

```
/**
  * random_shuffle 算法 对指定范围内的元素随机调整次序
  * @param beg 容器开始迭代器
  * @param end 容器结束迭代器
*/
random_shuffle(iterator beg, iterator end);
// 如果想要每次打乱不同，需要自己设置随机数种子
```

reverse:

```
/**
  * reverse 算法 反转指定范围的元素
  * @param beg 容器开始迭代器
  * @param end 容器结束迭代器
*/
reverse(iterator beg, iterator end);
```

常用拷贝和替换算法

copy:

```
/**
  * copy算法 将容器内指定范围的元素拷贝到另一容器当中
  * @param beg 容器开始迭代器
  * @param end 容器结束迭代器
  * @param dest 目标容器开始迭代器
*/
copy(iterator beg, iterator end, iterator dest);

vector<int> v = {1, 2, 3, 4, 5};
for_each(v.begin(), v.end(), [](int val){cout << val << " ";});
// 等价于
copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
// 需要#include <iterator>
```

replace:

```
/**
  * replace算法 将容器内指定范围的旧元素修改为新元素
  * @param beg 容器开始迭代器
  * @param end 容器结束迭代器
  * @param oldvalue 旧元素
  * @param newvalue 新元素
*/
replace(inerator beg, iterator end, oldvalue, newvalue);
```

replace_if:

```
/**
  * 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
  * @param beg 容器开始迭代器
  * @param end 容器结束迭代器
  * @param _callback 回调函数或者谓词（返回bool类型的函数对象）
  * @param newvalue 新元素
*/
replace_if(inerator beg, inerator end, _callback, newvalue);
```

swap:

```
/**
  * swap 算法 互换两个容器元素
  * @param c1 容器1
  * @param c2 容器2
*/
swap(container c1, container c2);
```

其它常用算法

accumulate

```
# include <numeric> // 注意头文件不是algorithm了
<p class="mume-header " id="include-numeric--注意头文件不是algorithm了"></p>

/**
  * accumulate 算法 计算容器元素累计总和
  * @param beg 容器开始迭代器
  * @param end 容器结束迭代器
  * @param value 起始累加值
*/
accumulate(iterator beg, iterator end, value);
```

fill

```
/**
  * fill 算法
  * @param beg 容器开始迭代器
  * @param end 容器结束迭代器
  * @param value 填充元素
*/
fill(iterator beg, iterator end, value);
```

set_intersection:

```
/**
  * set_intersection 算法 求两个set集合的交集
  * 注意：两个集合必须是有序序列
  * @param beg1 容器1开始迭代器
  * @param end1 容器1结束迭代器
  * @param beg2 容器2开始迭代器
  * @param end2 容器2结束迭代器
  * @param dest 目标容器开始迭代器
  * @return 目标容器最后一个元素的迭代器地址
*/
set_intersection(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest);
```

set_union:

```
/**
  * set_union 算法 求两个set集合的并集
  * 注意：两个集合必须是有序序列
  * @param beg1 容器1开始迭代器
  * @param end1 容器1结束迭代器
  * @param beg2 容器2开始迭代器
  * @param end2 容器2结束迭代器
  * @param dest 目标容器开始迭代器
  * @return 目标容器最后一个元素的迭代器地址
*/
set_union(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest);
```

set_difference:

```
/**
  * set_difference 算法 求两个set集合的差集
  * 注意：两个集合必须是有序序列
  * @param beg1 容器1开始迭代器
  * @param end1 容器1结束迭代器
  * @param beg2 容器2开始迭代器
  * @param end2 容器2结束迭代器
  * @param dest 目标容器开始迭代器
  * @return 目标容器最后一个元素的迭代器地址
*/
set_difference(iterator beg1, iterator end1, iterator beg2, iterator end2, iterator dest);
```