函数对象和智能指针

directory

函数对象

函数指针

- `void (*func)(int&);` 返回值//指针符号//声明的变量名//参数列表
- 和数组名类似：
  - 数组名 = 指向数组的第一个元素的指针
  - 函数名 = 指向函数的指针
- 使用auto自动推断函数的类型
  - auto必须要初始化
  - 例如 自动推断func的类型为`void(*)(int&)`

```
#include <iostream>
using namespace std;
void increase(int &x){x++;}  void decrease(int &x){x--;}
int main() {
    int flag; cin >> flag;
    int arr[] = {1,2,3,4,5};
    // mod1
    void (*func)(int&); //声明函数指针
    if(flag == 1) {func = increase; } else {func = decrease;}
    // mod2
    auto func = flag == 1 ? increase : decrease; //和上两行效果相同

    for (int &x : arr) { func(x); cout << x;}
}
```

`std::sort`STL函数

- `#include <algorithm>`

```
template<class Iterator>
void sort(Iterator first, Iterator last);
```

```
template<class Iterator, class Compare>
void sort(Iterator first, Iterator last, Compare cmp);
// 比较函数
bool cmp(int a, int b){return a > b;}
```

- 实际上 Compare的类型是`bool(*)(int, int)` STL提供了预定义的比较函数
- `#include <functional>`
- 从小到大: `sort(arr, arr+5, less<int>())`
- 从大到小: `sort(arr, arr+5, greater<int>())`

函数对象

实际上，`greater<int>()`是一个对象

- `greater`是一个模板类
- `greater<int>` 用int实例化的类
- `greater<int>()` 该类的一个对象
- 它表现的像一个函数，因此称为函数对象

```
#include <functional>
#include <iostream>
using namespace std;
int main() {
    auto func = greater<int>();
    cout << func(2, 1) << endl; //True
}
```

如何实现函数对象

```
#include <iostream>
#include <string>
using namespace std;
template<class T>
class Greater {
public:
    bool operator()(const T &a, const T &b) const {
    // 注意都是const，因为排序中，comp不能修改数据，一般情况下comp也不能修改自身
        return a > b;
    }
};
int main(){
    auto func = Greater<int>();
    cout << func(2, 1) << endl; //True
    cout << func(1, 1) << endl; //False

    auto func1 = Greater<string>();
    cout << func1("a", "b") << endl;
}
```

需要重载operator()运算符 并且该函数需要是public访问权

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

class A {
public:
    template <typename T>
    T operator() (T a, T b) {
        cout << a << '+' << b << '=' << a + b << endl;
        return a;
    }
};

template <typename T>
T addFunc(T a, T b, A& func) {
    func(a, b);
    return a;
}
int main() {
    A func;
    addFunc(1, 4, func);
    addFunc(1.4, 3.8, func);
}
```

实现自己的sort

sort的第三个参数是什么类型?

- `template <class Iterator, class Compare>`
- 模板类型, 可以接受**函数指针/函数对象**

```
#include <algorithm>
#include <functional>
using namespace std;

bool comp(int a, int b) {
    return a > b;
}
template<class Iterator, class Compare>
void mysort(Iterator first, Iterator last, Compare comp){
//mysort的时间复杂度为O(n^2), std::sort的时间复杂度为O(nlogn)
    for (auto i = first; i != last; i++)
        for (auto j = i; j != last; j++)
            if (!comp(*i, *j)) swap(*i, *j);
}
int main() {
    int arr[5] = { 5, 2, 3, 1, 7 };
    mysort(arr, arr + 5, comp);
    mysort(arr, arr + 5, greater<int>());
    //既可接受函数指针, 又可接受函数对象
}
```

自定义类型的排序

假设有一个`class People {public: int age, weight; };`如何按照年龄从小到大排序?

- 方法一：重载小于运算符

```
class People {
public:
    int age, weight;
    bool operator<(const People &b) const { return age < b.age; }
};
int main() {
    vector<People> vec = {{18, 50}, {16, 40}};
    sort(vec.begin(), vec.end());
}
```

- 方法二：定义比较函数

```
class People {
    public: int age, weight;
};

bool compByAge(const People &a, const People &b) {return a.age < b.age; }

int main() {
    vector<People> vec = {{18, 50}, {16, 40}};
    sort(vec.begin(), vec.end(), compByAge);
}
```

- 方法三：定义比较函数对象

```
class People {
    public: int age, weight;
};

class AgeComp {
public:
    bool operator()(const People &a, const People &b) const {return a.age <
b.age;}
};

int main() {
    vector<People> vec = {{18, 50}, {16, 40}};
    sort(vec.begin(), vec.end(), AgeComp());
}
```

std::function类

基于虚函数模板设计模式

```
class CalculatorBase {
public:
    virtual string read();
    virtual string calculate(string);
    virtual void calculate(string);
    void process() {
        string data = read();
        string output = calculate(data);
        write(output);
    }
};
```

使用函数对象:

```
void process(ReadFunc read, CalFunc calculate, WriteFunc write) {
    string data = read();
    string output = calculate(data);
    write(output);
}
process(readFromScreen, calculateAdd, writeToFile);
```

- ReadFunc, CalFunc, WriteFunc分别是什么？ 参数是函数指针

```
ReadFunc = string(*)(void)
CalFunc = string(*)(string)
WriteFunc = void(*)(string)
```

参数同时包含函数指针和函数对象 想用数组储存选项

```
auto readArr[] = {readFromScreen, ReadFromFile()}
process(readArr[0], calculate, write);
process(readArr[1], calculate, write);
```

- auto是什么类型?
    - 无法推导
    - 函数指针和函数对象不是同一种类型
- 需要一个类型能够统一两者

统一函数指针和函数对象

- std::function类

- #include <functional>头文件
- 尖括号里是**返回值(参数列表)**

```
function<string()> readArr[] =
        {readFromScreen, ReadFromFile()};
function<string(string)> calculateArr =
        {calculateAdd, CalculateMul()};
function<void(string)> writeArr[] =
        {writeToScreen, WriteToFile()};
```

function为函数指针与对象提供了统一的接口

```
#include <iostream>
#include <fstream>
#include <functional>
using namespace std;

class ReadFromFile {
public:
    string operator()(){
       string input;
       getline(ifstream("input.txt"), input);
       return input;
    }
};

string readFromScreen(){
    string input;
    getline(cin, input);
    return input;
}

int main() {
    function<string()> readArr[] =
            {readFromScreen, ReadFromFile()};
    function<string()> readFunc;
    readFunc = readFromScreen; // 允许函数的赋值
    readFunc = ReadFromFile();
    string (*readFunc2)(); // 函数指针
    readFunc2 = readFromScreen;
    //readFunc2 = ReadFromFile(); // 错误，类型不一致
    return 0;
}
```

对比实现方式

- 使用虚函数实现
  - 需要构造基类和子类
  - 运行时确定调用函数的地址

- 使用模板实现
  - 可以支持函数指针和函数对象（通过模板，自动重载实现）
  - 编译期确定调用函数的地址（当T不为std::function时）
- 使用`std::function`实现
  - 也可以支持函数指针和函数对象（通过function的多态）
  - 运行时确定调用函数的地址

智能指针与引用计数

智能指针有什么用?

- AB对象共享一个C对象,C对象需由AB内部销毁,应该在AB都销毁的时C才能销毁,那么谁来销毁C?

智能指针

`shared_ptr`

- `#include <memory>`
- 构造方法

```
shared_ptr<int> p1(new int(1));
shared_ptr<MyClass> p2 = make_shared<MyClass>(2);
shared_ptr<MyClass> p3 = p2;
shared_ptr<int> p4; //空指针
```

访问对象

```
int x = *p1;        //从指针访问对象
int y = p2->val;    //访问成员变量
```

销毁对象

- p2和p3指向同一对象，当两者均出作用域才会被销毁

引用计数

引用计数归0时,销毁对象

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

int main()
{
    shared_ptr<int> p1(new int(4));
    cout << p1.use_count() << ' '; // 1
    {  // begin 作用域
```

```
        shared_ptr<int> p2 = p1;
        cout << p1.use_count() << ' '; // 2
        cout << p2.use_count() << ' '; // 2
    }   //p2出作用域
    cout << p1.use_count() << ' '; // 1
    cout << *p1 << endl; // 4
    //调用delete, 销毁int*
}
```

实现自己的引用计数

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

template <typename T>
class SmartPtr;  //声明智能指针模板类

template <typename T>
class U_Ptr { //辅助指针
private:
    friend class SmartPtr<T>; //SmartPtr是U_Ptr的友元类
    U_Ptr(T *ptr) :p(ptr), count(1) { }
    ~U_Ptr() { delete p; }
    int count;
    T *p; // 实际数据存放
};
template <typename T>
class SmartPtr { //智能指针
    U_Ptr<T> *rp;
public:
    SmartPtr(T *ptr) :rp(new U_Ptr<T>(ptr)) { }
    SmartPtr(const SmartPtr<T> &sp) :rp(sp.rp) {
        ++rp->count;
  }
    SmartPtr& operator=(const SmartPtr<T>& rhs) {
        ++rhs.rp->count;
        if (--rp->count == 0) //减少自身所指rp的引用计数 pA = pB
            delete rp; //删除所指向的辅助指针
        rp = rhs.rp;
        return *this;
    }
    ~SmartPtr() {
        if (--rp->count == 0)
            delete rp;
    }
    T & operator *() { return *(rp->p); }
    T* operator ->() { return rp->p; }
};

int main(int argc, char *argv[]) {
    int *pi = new int(2);
```

```
    SmartPtr<int> ptr1(pi); //构造函数
    SmartPtr<int> ptr2(ptr1); //拷贝构造
    SmartPtr<int> ptr3(new int(3)); //能否ptr3(pi)???
    // 不可以 pi是普通整型指针
    //智能指针的构造函数需要接受动态分配的对象指针
    //进行内存管理和引用计数
    ptr3 =  ptr2; //注意赋值运算
    cout << *ptr1 << endl; //输出2
    *ptr1 = 20;
    cout << *ptr2 << endl; //输出20
}
```

shared_ptr的其他用法

- `p.get()` 获取裸指针
- `p.reset()` 清除指针并减少引用计数
- `static_pointer_cast<int>(p)`
  - 转为int类型指针(和static_cast类似，无类型检查)
- `dynamic_pointer_cast<Base>(p)`
  - 转为int类型指针(和dynamic_cast类似，动态类型检查)
- 注意!
  - 不能使用同一裸指针初始化多个智能指针

```
int* p = new int();
shared_ptr<int> p1(p);
shared_ptr<int> p2(p);
// 会产生多个辅助指针!
```

- 不能直接使用智能指针维护对象数组
  - 数组的delete方式; 所有的实现都是delete p;删除数组需要delete[] p;

`week_ptr<T> p(new T())`

- 指向对象但不计数
- 弱引用的创建
  - `shared_ptr<int> sp(new int(3));`
  - `weak_ptr<int> wp1 = sp;`
- 弱引用的用法
  - `wp.use_count()`: 获取引用计数
  - `wp.reset()`: 清除指针
  - `wp.expired()`: 检查对象是否无效
  - `sp = wp.lock()`: 从弱引用获得一个智能指针

```
#include <memory>
#include <iostream>
using namespace std;
int main()
{
    std::weak_ptr<int> wp;
    {
        auto sp1 = std::make_shared<int>(20);
        wp = sp1;
        cout << wp.use_count() << endl; //1
        auto sp2 = wp.lock();   //从弱引用中获得一个shared_ptr
        cout << wp.use_count() << endl; //2
        sp1.reset();                                //sp1释放指针
        cout << wp.use_count() << endl; //1
    }   //sp2销毁
    cout << wp.use_count() << endl; //0
    cout << wp.expired() << endl;     //检查弱引用是否失效：True
    return 0;
}
```

独享所有权

- shared_ptr涉及引用计数,性能较差
- 保证一个对象只被一个指针引用使用unique_ptr

```
#include <memory>
#include <utility>
using namespace std;
int main() {
auto up1 = std::make_unique<int>(20);
//unique_ptr<int> up2 = up1;
            //错误，不能复制unique指针
unique_ptr<int> up2 = std::move(up1);
            //可以移动unique指针
int* p = up2.release();
            //放弃指针控制权，返回裸指针
delete p;
return 0;
}
```

- 优点
    - 智能指针可以帮助管理内存，避免内存泄露
    - 区分unique_ptr和shared_ptr能够明确语义
    - 在手动维护指针不可行，复制对象开销太大时，智能指针是唯一选择。
- 缺点
    - 引用计数会影响性能

- 智能指针不总是智能，需要了解内部原理
- 需要小心环状结构和数组指针