函数对象和智能指针

函数对象

什么是函数指针

函数本质上存在于代码段，因此，每个函数在代码段中，也有着自己的入口地址。

函数指针是一个指针类型的变量。**它指向代码段中函数入口的地址。**

函数指针的声明

声明格式如下：

```
ret (*func)(args,...)
```

其中，ret是函数指针指向的函数的返回类型，func是该指针的名字，而args...是该指针指向的函数的参数列表。

也就是说，这个名为func的指针只能指向返回值类型、参数类型都相同的**那些函数**。

函数指针由于返回值、参数不同而有很多种类型。每种类型只能指向特定的**一些函数**。

由于函数的类型比较难写，常常用auto关键字**推断函数指针类型**，直接令函数指针=某一函数名。注意如果用auto，在对函数指针声明的同时必须对其初始化，否则编译器无法自动推导指针类型。例如：

```
#include <iostream>
int arr[5] = { 5, 2, 3, 1, 7 };
void increase(int &x){x++;}
void decrease(int &x){x--;}
int main()
{
    int flag;
  std::cin >> flag;
  //void (*func)(int &); //手工指定类型，可以先声明函数指针，再赋值：
  //if(flag==1) {func=increase; } else {func=decrease;}

    auto func = flag==1?increase:decrease; //auto推导函数指针类型，和上两行效果相同
  for (int &x:arr) { func(x); cout << x;}
  return 0;

}
```

函数指针的初始化

给你一个函数，怎么让一个指针指向它呢?

与数组类似，在数组中，**数组名就是**该数组的**首地址**，函数也是一样，函数名就是**该函数的入口地址**，因此，函数名就是该函数的函数指针。也就是说，函数名A是个指针，该指针指向的内存空间和储存该指针的内存空间是同一块，就是储存该函数的内存空间的头部位置。但是如果用别的指针B指向了函数，那么储存B的内存空间和B指向的元素的内存空间不是同一块。

```
#include <iostream>
using namespace std;
void func(){}
int main() {
int arrow[10];
auto p=func;
cout<<arrow<<endl;
cout<<&arrow<<endl;
cout<<func<<endl;
cout<<&func<<endl;
cout<<p<<endl;
cout<<&p<<endl;
}
output:
0x7fff7a33d6b0
0x7fff7a33d6b0
1
1
1
0x7fff7a33d6a8
```

理清楚了指针指向的内存空间与储存指针的内存空间的关系后，我们可以采用如下的两种等价初始化方式:

```
函数指针变量 =  函数名;
函数指针变量 =  &函数名;//取地址运算符&不是必需的，因为一个函数标识符就表示了该函数入口的地址。
//btw，对一个指针取地址会发生什么？数组名不是头指针吗？数组名这个指针储存着首地址？

//这里就解释了，为什么函数调用，必须包含一个圆括号括起来的参数表。
//也就是说，以前我们经常会犯obj.reset这种错误，但实际上该写为obj.reset()
//理由如果func是没有参数的函数，那func是func函数入口的地址，而func()则是调用该函数。
```

函数指针的用途

函数指针主要有两个用途：做函数的参数、调用函数。

(1)函数指针调用函数

用函数指针调用函数有两种方法，都不要忘记参数列表的圆括号()。

```
x = (*fun)();//这种写法更能体现出fun是一个指针
x = fun();
```

这个x和fun，谁是函数，谁是指针？这里体现的不是函数指针初始化吗？

(2)函数指针做函数参数

回忆来自的sort函数。它有两种调用方法：

```
sort(arr,arr+5);//默认从小到大sort，arr数组的类型需要支持比大小操作。如果arr的类型是自定义的struct或者class，没有默认的比大小操作，此时需要重载<运算符。
sort(arr,arr+5,comp);//用自定义的comp函数来sort。这可以很方便地支持逆序sort，只需要把comp函数改为逆序即可。
```

第二种定义方式里，我们把comp这个函数名作为参数传了进去。我们知道函数名实际上就是函数地址。所以这实际上就是传进了函数指针做参数。

用自定义的comp函数实现逆序sort：

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

bool comp(int a, int b)
    {   return a > b;  }
//如果是顺序，应该return a<b;
//注意，这里的参数就是int a和int b，不是取引用
int main(){
    int arr[5] = { 5, 2, 3, 1, 7 };
    std::sort(arr, arr + 5, comp);
    for (int x : arr) {
        cout << x << " ";
    }
    return 0;
}
//output:7 5 3 2 1
```

对sort函数来说，它的第三个参数是函数指针，因为我们传入的是comp()函数的头指针，即comp，而不是comp()。

这个函数指针的类型是：

```
bool (*)(T,T)
```

这是一种叫Compare类型的特殊函数指针类型。

是指sort函数默认第三个参数为Compare类型的对象吗？某个函数指针属于某个类型是什么意思？

函数对象

除了自定义的comp，我们也可以用STL提供的预定义的比较函数(需要#include)

```
sort(arr, arr+5, less<int>());//从小到大
sort(arr, arr+5, greater<int>());//从大到小
```

less()，greater()是什么东西呢？和comp一样，它们有Compare类型的函数指针的功能，但它其实是**一个对象**！

以greater()为例，其内部实现机制如下。

注意到，greater的实现已经内置在了中，即以下代码在实际应用时不必写出。此处为了避免关键字冲突，将greater写为Greater。这就好比std里的swap和自己手写的swap一样。

```
#include <iostream>
using namespace std;
template<class T>
class Greater {//大G避免关键字冲突
public:
  Greater(){};//默认构造函数;
    bool operator()(const T &a, const T &b) const   {//重载了()
        return a > b;
    }//参数是引用类型，是为了防止重复拷贝；参数是const，则是为了防止修改a和b
};
```

通过在public中完成对operator()的重载，当该模版被实例化为一个对象后，可以通过()调用该对象，看起来就像一个函数。这样的**重载了()**的对象称为“函数对象”。

```
//Greater是一个模板类
//Greater<int> 用int实例化的类
//Greater<int>() 调用了Greater<int> 的构造函数，构造出的一个Greater<int>类型对象
```

调用函数对象：

```
#include <iostream>
using namespace std;
template<class T>
class Greater {
public:
    bool operator()(const T &a, const T &b) const   {
        return a > b;
    }
};
int main(){
    auto func = Greater<int>();
    cout << func(2, 1) << endl; //True
    cout << func(1, 1) << endl; //False
```

```
    cout << func(1, 2) << endl; //False
    return 0;
}
```

注意，这里我们可以很**优雅**地写func(1,2),是因为之前已经把名叫Greater的类模版实例化为了一个对象。实际上，也可以不实例化而直接调用。但这时一定不要忘记()，第一个()代表构造函数，第二个()才代表对括号的重载。

```
#include <algorithm>
#include <iostream>
#include <functional>
using namespace std;
int main()
{
  cout<<greater<int>()(1,2);//正确
//cout<<greater<int>(1,2);//错误，这是在以1，2为参数调用构造函数
  return 0;
}
//output:0
```

以greater()为参数调用sort：

```
#include <functional>
#include <iostream>
#include <algorithm>
using namespace std;
int main() {
    auto func = greater<int>();//this is an object
    int arr[5]={1,4,2,8,3};
    sort(arr,arr+5,greater<int>());
    for (int i:arr){cout<<i<<' ';}
    return 0;
}
//output:8 4 3 2 1
```

所以，std::sort既可以接受函数指针，又可接受函数对象作为第三个参数。实际上，sort是一个函数模板，其模版参数是待sort的数组的类型和一个Compare类型的函数指针。调用sort的格式如下：

```
sort(Iterator first,Iterator last,comp);
```

调用时，系统会自行推导函数模版参数，因此我们不必手工指定。

最后，关于sort函数的一个细节：sort函数不仅可以用来sort数组，还可以sort任何**有序的**容器。比如Vector。无论是数组、list还是vector，sort的第一个参数都应该指向第一个元素，而第二个参数则指向**最后一个元素之后**的那个位置，**而不是最后一个元素。**

```cpp
int arr[5]={1,3,2,6,4};
vector<int>vec={1,3,2,6,4};
sort(arr,arr+5);//arr+5=arr[5],这是最后一个元素之后的位置
sort(vec.begin(),vec.end());//vec.end()也是最后一个元素之后的位置
```

std::function类

由对sort的讨论，我们发现函数指针和函数对象有高度的一致性。但另一方面，它们的形式又不统一。

在下面这个例子中，从屏幕、文件读取这两个函数不能被同一个数组统一。

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

string readFromScreen()//从屏幕读取
{
    string input; getline(cin, input);
    return input;
}

class ReadFromFile//从文件读取
{
public:
    string operator()(){
        string input;
        getline(ifstream("input.txt"), input);
        return input;
    }
};
//string operator()()的意思是返回值类型为string，第一个()表示重载(),第二个()表示没有参数

int main(){
  auto readArr[] = {readFromScreen, ReadFromFile()};//我们试图用数组管理这两个类似读取函数
  //auto类型推导失败，因为readFromScreen是string(*)(void)类型的函数指针，ReadFromFile()则是ReadFromFile类型的对象
  return 0;
}
```

解决方法：std::function类，来自头文件。

function类是一种特殊的类模版

function类是一种特殊的**类模板**，模板参数是函数返回值类型和参数类型。但是特殊的是，function只有成员函数，无数据成员。

function类的实例化

function类为函数指针与对象提供了统一的接口。实际上，function的实例可以存储，复制和调用的不仅仅是函数指针与对象，还包括lambda表达式，绑定表达式和指向成员函数和指向数据成员的指针。（我并不知道这些是什么。）不管采用哪种方式，只要调用形式一样（返回值类型、实参类型），我们就可以用function类型来统一。

function类的实例化格式如下：

```
function<ret(args...)>Func;
```

其中ret是函数返回值类型，()中是参数类型列表，Func是该function类对象的名字。

function类的应用

上面的两个读取函数可以统一在function类中：

```
#include <iostream>
#include <fstream>
#include <functional>
#include <cstring>
using namespace std;
string readFromScreen()//从屏幕读取
{
    string input; getline(cin, input);
    return input;
}

class ReadFromFile//从文件读取
{
public:
    string operator()(){
        string input;
        getline(ifstream("input.txt"), input);
        return input;
    }
};

int main()
{
    function<string()> readArr[] =
            {readFromScreen, ReadFromFile()};//function类的数组readArr可以统一这两种读取函数；
    function<string()> readFunc;////readFunc是个function类变量，既可以被函数对象，也可以被函数指针赋值
    readFunc = readFromScreen;
    readFunc = ReadFromFile();

    string (*readFunc2)();//和function类不同，对于传统的函数指针来说
    readFunc2 = readFromScreen;//它只能被函数赋值
    //readFunc2 = ReadFromFile(); //不能被函数对象赋值
    return 0;
}
```

用function实现多态

正如sort函数所实现的那样，通过将function类型作为函数参数，可以统一函数对象和函数指针。以function类型作为函数参数的函数可以仅仅改变参数，获得不同的功能。

例：

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

//省略readFromScreen\ReadFromFile\calculateAdd\writeToScreen
void process(
        function<string()> read,
        function<string(string)> calculate,
        function<void(string)>  write)
{
    string data = read();
    string output = calculate(data);
    write(output);
}
int main()
{
    process(readFromScreen, calculateAdd, writeToScreen);
    process(ReadFromFile(), calculateAdd, writeToScreen);//多态
    return 0;
}

```

例2:

当process的参数类型是function<string()>时，可以同时接受func1,func2,func3作为函数参数。

如果process的参数类型是string (*func)(),就无法接受f2,f3;

如果process的参数类型是Func2,就无法接受f1,f3;

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

string func1();//func1是个全局函数，参数为void，返回值为string
class Func2
{public: string operator()();
};
Func2 func2;//func2是一个函数对象，参数为void，返回值为string
function<string()> func3;//func3是一个function类，参数为void，返回值为string

void process(function<string()> func){
  string str=func();
  cout<<str;
}
/*也可以写成：
template<class T>
void process(T func){...}
```

```
即用函数模版实现多态。比function写起来还简单
/*

int main()
{
    process(func1);
    process(func2);
    process(func3);
    return 0;
}

```

STL中大量函数用到了函数对象(#include)。以下这些函数都调用了函数指针或者函数对象作为参数。

for_each 对序列进行指定操作
find_if 找到满足条件的对象
count_if 对满足条件的对象计数
binary_search 二分查找满足条件的对象
并且也有许多预置的函数对象(#include)
less 比较a<b
equal_to 比较a==b
greater 比较a>b
plus 返回a+b
.....
熟练使用函数对象有助于实现复杂的功能

以for_each为例:

```
// for_each example
#include <iostream>     // std::cout
#include <algorithm>    // std::for_each
#include <vector>       // std::vector

void myfunction (int& i) {  //普通的函数
  i++;
  std::cout<<i<<' ';
}

struct myclass {           //函数对象类型的函数
  void operator() (int& i) {
    i+=2;
    std::cout<< i <<' ';}
} myobject;

int main () {
  std::vector<int> myvector;
  myvector.push_back(10);
  myvector.push_back(20);
  myvector.push_back(30);

  std::cout << "after myfunction,now myvector contains:";
  for_each (myvector.begin(), myvector.end(), myfunction);//传进函数指针作为参数
  std::cout <<std::endl;
```

```
26
27    std::cout << "after myobject,now myvector contains:";
28    for_each (myvector.begin(), myvector.end(), myobject);//传进函数对象作为参数
29
30    return 0;
31  }
32  //output:
33  //after myfunction,now myvector contains:11 21 31
34  //after myobject,now myvector contains:13 23 33
```

两次调用了for_each函数，分别对vector里面的每一个元素执行了myfunction和myobject。因为这两个函数的函数参数都是引用，主函数中的变量也会被修改。

另一个count_if的例子：

```
1  // count_if example
2  #include <iostream>      // std::cout
3  #include <algorithm>     // std::count_if
4  #include <vector>        // std::vector
5
6  bool IsOdd (int i) { return ((i%2)==1); }
7
8  int main () {
9    std::vector<int> myvector;
10   for (int i=1; i<10; i++) myvector.push_back(i); // myvector: 1 2 3 4 5 6 7
   8 9
11
12   int mycount = count_if (myvector.begin(), myvector.end(), IsOdd);
13   std::cout << "myvector contains " << mycount  << " odd values.\n";
14
15   return 0;
16 }
17 //output:
18 //myvector contains 5 odd values.
```

count_if函数需要传进一个返回值为bool类型的函数作为参数。

与其他多态实现方式的对比

（1）使用虚函数实现：

对基类的指针或者引用，在运行时通过虚函数表确认该指针或引用的实际类型，并调用实际类型的重写覆盖后的函数，以实现多态。

是晚绑定（运行时绑定）

（2）使用模板实现：

上文中的greater()就是模版实现多态的例子。通过传入不同的模版参数，自动实现重载，可以实现函数对象和函数指针的多态。

是早绑定（编译期绑定）

(3)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）
是晚绑定（运行时绑定）

意义

将函数也对象化。函数可以作为参数传递，函数也可以作为变量储存，并且只要函数的参数和返回值相同，就可以被视为同一种类型的变量，不再需要模板来调用不同的函数。

智能指针

当两个指针A，B同时指向一个变量C的时候，我们希望只有A，B均被析构的时候，C才被析构。

如何做到？引入智能指针！（包含在头文件中）

一篇CSDN上的说明的链接：https://blog.csdn.net/flowing_wind/article/details/81301001

构造智能指针

智能指针的创建：

```
shared_ptr<int> p1(new int(1));//指向内容为1的一块内存
shared_ptr<int> p2; //空指针（不初始化）
auto p3=make_shared<int>(3);//make_shared<T>(agrs)是一个函数，返回值是一个share_ptr,指向一个动态分配的类型为T的对象，args是用来初始化T的参数。
int a;shared_ptr<int>p4(a);//用已有对象创建智能指针。
```

让两个指针指向同一个位置：

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

class Myclass{
  public:
  int myint;
  Myclass(int i):myint(i){};
};

int main(){
  shared_ptr<Myclass> p2=make_shared<Myclass>(2);
  shared_ptr<Myclass> p3=p2;//p2和p3指向同一块内存
  //auto p3(p2);和上面的句子等价
  shared_ptr<Myclass> p4;
  p4=p3;//对已有的智能指针，也可以直接进行赋值。这会使p4指针的count-1，p3指针的count+1.
  cout<<p2<<endl;
  cout<<p3<<endl;
  cout<<p4<<endl;
}//output:
//0x14b0e80
//0x14b0e80
```

```
//0x14b0e80
```

引用计数use_count()

用obj.use_count()函数，可以得到智能指针obj此刻指向的物体（包括obj自己在内）共有几个智能指针指向它。

例：

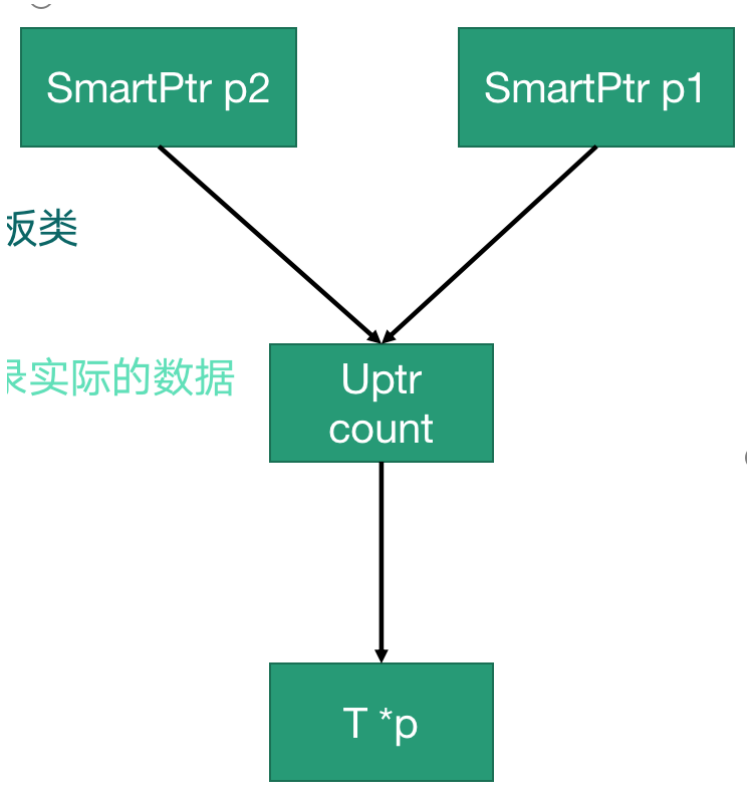
```
#include <memory>
#include <iostream>
using namespace std;
int main()
{
    shared_ptr<int> p1(new int(4));
    cout << p1.use_count() << ' '; // 1
    {
        shared_ptr<int> p2 = p1;
        cout << p1.use_count() << ' '; // 2
        cout << p2.use_count() << ' '; // 2
    }   //p2出作用域
    cout << p1.use_count() << ' '; // 1
}

```

如果函数参数是智能指针类型，则在调用函数时，由于调用了拷贝构造构造智能指针作为参数，新构造出的智能指针指向同一对象，会发生count增加。如果想要避免，可以将函数参数类型改为智能指针的引用。

```
#include <memory>
#include <iostream>
using namespace std;
void f1(shared_ptr<int> p1) {//拷贝构造了智能指针作为参数
    cout << p1.use_count();//此时count为2
}
void f2(shared_ptr<int> &p1){//参数是智能指针引用
    cout << p1.use_count();//此时count仍为1
}
int main()
{
    shared_ptr<int> p1(new int);//此时count为1
    f1(p1);
    cout << p1.use_count();//出f1函数体时，参数被析构，count回归到1
    f2(p1);
    cout << p1.use_count();
    return 0;
}
//output:2111
```

智能指针的实现原理



如图，智能指针指向的是辅助指针Uptr，辅助指针再指向真正指向数据存放位置的指针p.

辅助指针Uptr有两个功能：一方面，它指向真正指向数据存放位置的指针p；另一方面，它具有成员count，记录有几个智能指针指向自己。每个指向“真正的”数据的指针p都只被一个辅助指针指向，p和Uptr一一对应。

这也就是为什么我们不能直接用智能指针给普通指针赋值

```
#include <memory>
#include <iostream>
using namespace std;
int main()
{
    shared_ptr<int> p1(new int(4));
    cout << p1.use_count() << ' '; // 1
    int *normal=p1;//我们希望normal可以和p1指向相同的内存
    return 0;
}//output:
/*.code.tio.cpp:8:7: error: no viable conversion from 'shared_ptr<int>' to
'int *'
        int *normal=p1;
            ^      ~~*/
```

那如果想让普通指针指向智能指针真正指向的位置该怎么办呢？利用p.get()函数！p.get()是智能指针p真正指向的数据对应的指针（又称为裸指针）（也就是存储真正数据位置的地址）。

```
int *normal=p1.get();
```

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

```
4
5  int main(){
6    int *pi = new int(2);
7    shared_ptr<int> ptr(pi); //用已有对象初始化
8    cout<<"ptr now points to ";
9    cout<<ptr.get()<<endl;
10   int *p =ptr.get();
11   cout<<"p now points to ";
12   cout<<p<<endl;
13   cout<<"But the count of ptr is "<<ptr.use_count();
14   return 0;
15 }
16 //output
17 ptr now points to 0x1557e70
18 p now points to 0x1557e70
19 But the count of ptr is 1
```

上面的例子说明， 虽然理论上可以这么做，但因为普通指针不和辅助指针发生关系，无法增加count的数量。因此一般**不能混合使用普通指针和智能指针**。如果我们希望有一个指向该物体的指针，并且不被记入count，可以使用weak_ptr.（见后）

智能指针的初始化

可以用智能指针初始化另一个智能指针：

```
1 shared_ptr<int> ptr2(new int(3));
2 shared_ptr<int> ptr3(ptr2);
```

但是不能使用同一裸指针初始化多个智能指针（这样会出现多个辅助指针）

下面这个例子是错的。

```
1 int* p = new int();
2 shared_ptr<int> p1(p);
3 shared_ptr<int> p2(p);
```

智能指针的析构

智能指针析构的时候，其内部的成员辅助指针并不一定会被析构，因为还可能有其他智能指针指向该辅助指针。在智能指针析构的时候时，其辅助指针内记录智能指针数量的成员count发生$count--$.只有$count--$后为0的时候，辅助指针才会被同时析构。

智能指针的其他操作

```
1 p.get() //获取裸指针,i.e.获得实际指针的情况
2 p.reset()   //
3 shared_ptr<myType> q=static_pointer_cast<myType>(p)//不做类型检查，直接把p转换成
  myType类型的指针
4 shared_ptr<Base> q=dynamic_pointer_cast<Base>(p)//做类型检查。
```

其中，dynamic_pointer_cast和static_pointer_cast所遵循的规则和我们在L9中学过的dynamic_cast、static_cast相同。但是其参数和返回值都是shared_ptr.

```
shared_ptr<Base> q=dynamic_pointer_cast<Base>(p);//该dynamic_cast等价于下面的语句。
Base*q=dynamic_cast<Base*>(p.get());
```

reset函数：将p指向另外一个对象。当()为空的时候，p指向null.当使用reset函数时，原来p指向的对象的count会-1。

```
#include <iostream>
#include <memory>
using namespace std;
int main(){
  shared_ptr<int>sp(new int (1)) ;
  cout<<"the address is"<<sp<<'\n';
  sp.reset(new int (2));
  cout<<"the address is"<<sp<<'\n';
  sp.reset();
  cout<<"the address is"<<sp<<'\n';
  return 0;
}
//ouput:
the address is0x17e1e70
the address is0x17e1eb0
the address is0
```

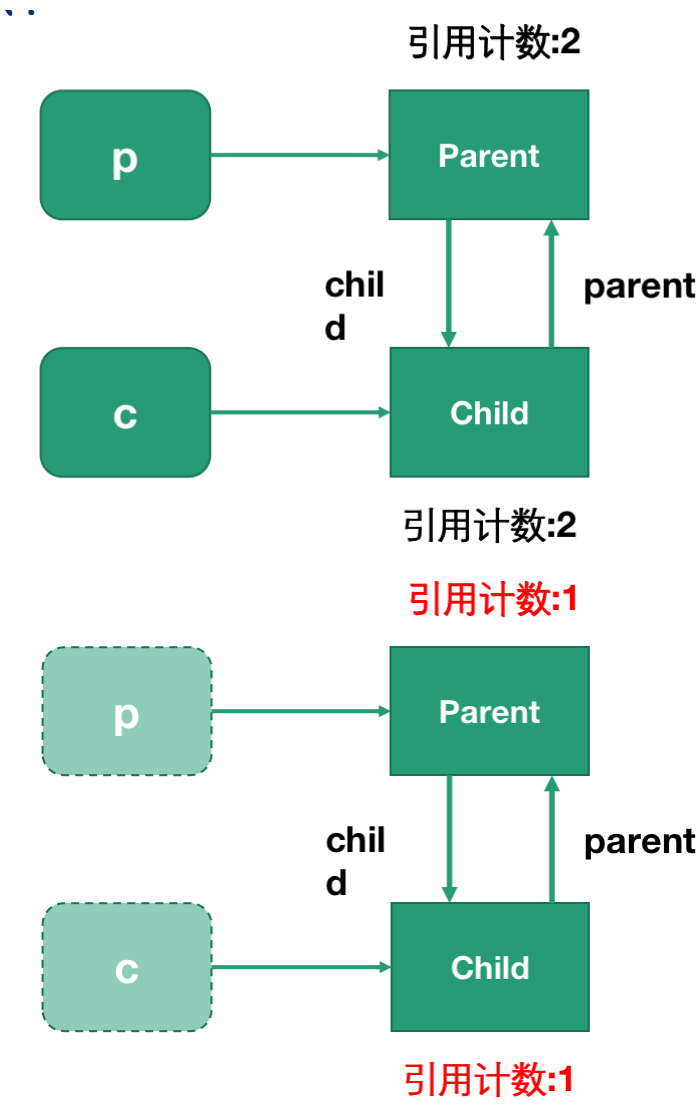
智能指针的问题

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

class Child;
class Parent {
    shared_ptr<Child> child;
public:
    Parent() {cout << "parent constructing" << endl; }
    ~Parent() {cout << "parent destructing" << endl; }
    void setChild(shared_ptr<Child> c) {
        child = c;
    }
};
class Child {
    shared_ptr<Parent> parent;
public:
    Child() {cout << "child constructing" << endl; }
    ~Child() {cout << "child destructing" << endl; }
    void setParent(shared_ptr<Parent> p) {
        parent = p;
    }
};
void test() {
    shared_ptr<Parent> p(new Parent());
    shared_ptr<Child> c(new Child());
    p->setChild(c);
```

```
    c->setParent(p);
    //p和c被销毁
}

int main()
{
    test();
    return 0;
}
output:
parent constructing
child constructing
//p和c没有发生析构!
```



如图，Parent和Child两个对象中的成员分别是指向对方的智能指针，导致p，c被析构之后，Parent和Child所在内存没有被析构。

解决方法：weak_ptr.

弱引用weak_ptr

弱引用指针指向对象的时候，不会被count计数。因此，weak_ptr必须由shared_ptr构造，否则其指向的对象无法被析构。

构造弱引用指针：

```
shared_ptr<int> sp(new int(3));
weak_ptr<int> wp1 = sp;//weak_ptr必须由shared_ptr构造
```

虽然弱引用指针不被count计数，但仍然可以获取引用次数count。其count值为指向同一个对象的智能指针的数量。

弱引用指针的操作

```
wp.use_count()  //获取引用计数
wp.reset()          //清除指针
wp.expired()        //当弱引用指针的count==0时，失效，返回true。否则返回false。
sp = wp.lock()  //从弱引用wp获得一个智能指针sp，该智能指针会增加count。
```

独享所有权unique_ptr

unique_ptr:每个对象只能由一个unique_ptr指向它。该对象不能被其他shared_ptr,weak_ptr指向。

某个时刻只能有一个unique_ptr指向一个给定对象，由于一个unique_ptr“拥有”它指向的对象，因此unique_ptr不支持普通的拷贝或赋值操作。

unique_ptr的创建

```
unique_ptr<int>u1;//空指针
unique_ptr<int>u2(new int(3));//新分配一块内存给unique_ptr.
int a=1;unique_ptr<int>u3(a);//用已有对象a创建unique_ptr.
auto u4 = std::make_unique<int>(20);//也是新分配一块内存给unique_ptr.make_unique函数自动返回unique_ptr类型的返回值，供编译器推导u4的类型。
```

和weak_ptr不同，unique_ptr不需要和shared_ptr共同使用。实际上，unique_ptr不能和shared_ptr同时指向一个对象，也不能用shared_ptr构造unique_ptr。

下面这个例子就是错的。

```
shared_ptr<int> sp(new int(1));
unique_ptr<int> up(sp);
```

unique_ptr的操作

```
unique_ptr<int>u1;//创建空unique_ptr
u1=nullptr;//此时会释放u1指向的对象
u1.release();//此时不会释放u1指向的对象，但是u1会放弃对该对象的控制权。这个函数的返回值是一个指向该对象的普通指针，此时u1被置空
//也就是说，虽然我们不能拷贝或者赋值unique_ptr，但是可以通过调用release或reset将指针所有权从一个（非const）unique_ptr转移给另一个unique_ptr或是普通指针
u1.reset();//在改变u1指向的同时，会释放u1指向的对象
```

将一个对象的所有权在两个unique_ptr之间转换的方法

```
//方法1:将p1指向的对象转移给p2
unique_ptr<int> p1(new int (1));
unique_ptr<int> p2(p1.release());//release将p1置为空，并且返回裸指针
//方法2:将p3指向的对象转移给p2
unique_ptr<int>p3(new int(3));
p2.reset(p3.release());//reset释放了p2原来指向的内存
```

release成员返回unique_ptr当前保存的指针并将其置为空。因此，p2被初始化为p1原来保存的指针，而p1被置为空。
reset成员接受一个可选的指针参数，令unique_ptr重新指向给定的指针。
调用release会切断unique_ptr和它原来管理的的对象间的联系。release返回的指针通常被用来初始化另一个智能指针或给另一个智能指针赋值。

例子：

```
#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//up1移动给up2，up1就被销毁了
int* p = up2.release();
            //放弃指针控制权，返回裸指针
delete p;
return 0;
}

```

总结

智能指针可以帮助管理内存，避免内存泄露。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