移动语义与智能指针

移动语义

为什么要用移动语义?

我们回顾一下之前模拟的String.cc

```
class String
{
public:
    String(): _str(new char[1]()) {}

    String(const char* pstr):_str(new char[strlen(pstr) + 1]())
    {strcpy(_str, pstr);}

    String(const String& rhs) :_str(new char[strlen(rhs.c_str()) +
1]())
    { strcpy(_str, rhs.c_str()); }

    String &operator=(const String &rhs){
        if (this != &rhs){
            delete [] _str;
            _str = new char[strlen(rhs.c_str()) + 1];
            strcpy(_str, rhs.c_str());
        }
        return *this;
    }

    ~String(){
        if (_str){
            delete [] _str;
            _str = nullptr;
        }
    }
private:
    char* _str;
};

void test0(){
    String str("hello");
    //拷贝构造
```

```
    String s2 = s1;
    //先构造，再拷贝构造
    //利用"hello"这个字符串创建了一个临时对象
    //并复制给了s3
    //这一步实际上new了两次
    String s3 = "hello";

}
```

创建s3的过程中实际创建了一个临时对象，也会在堆空间上申请一片空间，然后把字符串内容复制给s3的pstr，这一行结束时临时对象的生命周期结束，它申请的那片空间被回收。这片空间申请了，又马上被回收，实际上可以视作一种不必要的开销。我们希望能够少new一次，可以直接将s3能够复用临时对象申请的空间。

左值与右值

左值和右值是针对表达式而言的，**左值**是指表达式执行结束后依然存在的持久对象，**右值**是指表达式执行结束后就不再存在的临时对象。那如何进行区分呢？其实也简单，能对表达式取地址的，称为左值；不能取地址的，称为右值。

在实际使用过程中，字面值常量、匿名对象（临时对象）、匿名变量（临时变量），都称为右值。右值又被称为即将被销毁的对象。

字面值常量，也就是10， 20这样的数字，属于右值，不能取地址。

字符串常量，“world"，是属于左值的，位于内存中的文字常量区

试试看下面这些取址操作和引用绑定操作是否可行：

```
void test1() {
    int a = 1, b = 2;
    &a;
    &b;
    &(a + b);
    &10;
    &String("hello");

    //非const引用尝试绑定
    int & r1 = a;
    int & r2 = 1;

    //const引用尝试绑定
    const int & r3 = 1;
    const int & r4 = a;

```

```
    String s1("hello");
    String s2("wangdao");
    &s1;
    &s2;
    &(s1 + s2);
}
```

如上定义的`int & r1`和`const int & r3`叫作左值引用与const左值引用

非const左值引用只能绑定到左值，不能绑定到右值，也就是非const左值引用只能识别出左值。

const左值引用既可以绑定到左值，也可以绑定到右值，也就是表明const左值引用不能区分是左值还是右值。

——希望能够区分出右值，并且还要进行绑定

就是为了实现String s3 = "hello"的空间复用需求。

右值引用

C++11提出了新特性**右值引用**

右值引用不能绑定到左值，但是**可以绑定到右值**，也就是右值引用可以**识别出右值**

```
    //非const引用不能绑定右值
    int & r1 = a;
    int & r2 = 1; //error

//const引用既可以绑定左值，又可以绑定右值
    const int & r3 = 1;
    const int & r4 = a;

    //右值引用只能绑定右值
    int && r_ref = 10;
    int && r_ref2 = a; //error
```

右值引用本身是左值还是右值?

—— 对r_ref取地址是可行的，r_ref本身是一个左值。但这并不代表右值引用本身一定是左值。

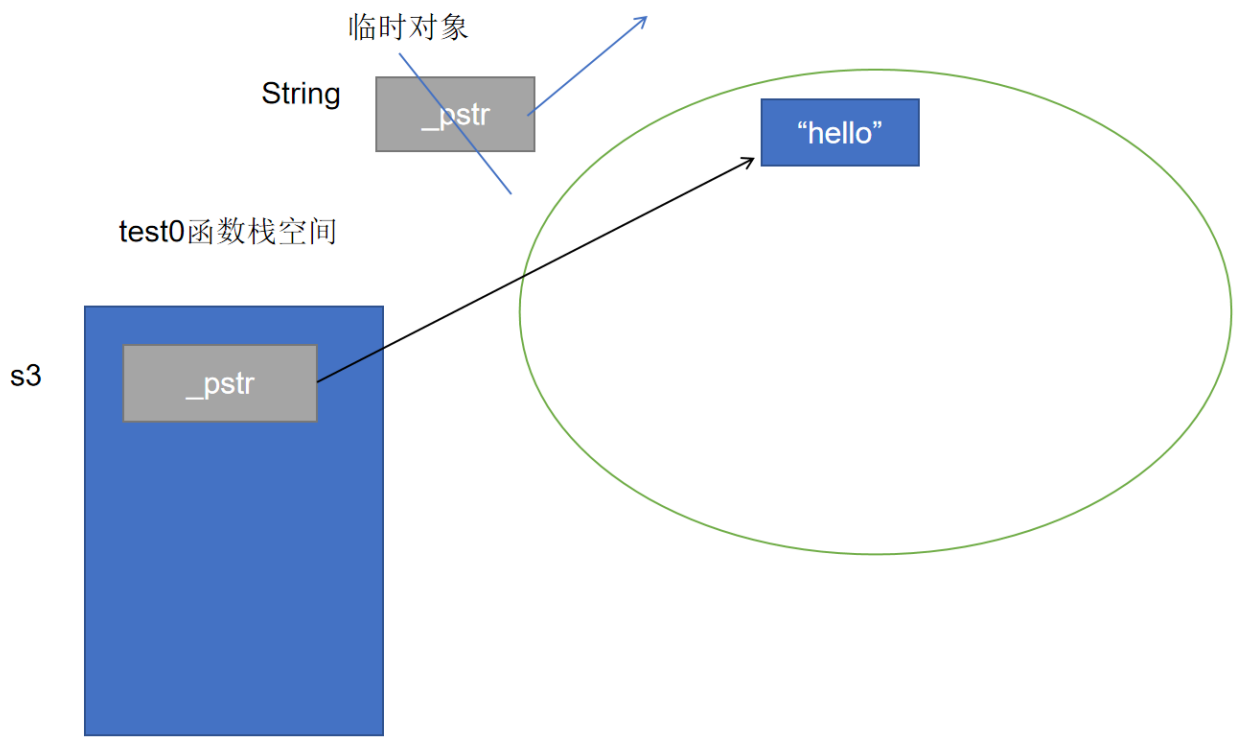
实际上，右值引用既可以是左值（比如：作为函数的参数、有名字的变量），也可以是右值（函数的返回类型）

这个问题，我们留到1.1.6章节再做讨论。

移动构造函数（重要）

有了右值引用后，实际上再接收临时对象作为参数时就可以分辨出来。

之前String str1 = String("hello");这种操作调用的是拷贝构造函数，形参为const String &类型，既能绑定右值又能绑定左值。为了确保右值的复制不出错，拷贝构造的参数设为const引用；为了确保进行左值的复制时不出错，一律采用重新开辟空间的方式。有了能够分辨出右值的右值引用之后，我们就可以定义一个新的构造函数了 —— **移动构造函数**。



给String类加上移动构造函数，在初始化列表中完成浅拷贝，使s3的pstr指向临时对象的pstr所指向的空间（复用），还不能忘记要将右操作数（临时对象）的pstr设为空指针，因为这个临时对象会马上销毁（要避免临时对象调用析构函数回收掉这片堆空间）

```
    String(String && rhs)
    : _pstr(rhs._pstr)
    {
        cout << "String(String&&)" << endl;
        rhs._pstr = nullptr;
    }
```

再运行代码，发现String s3 = "hello";

加上编译器的去优化参数 -fno-elide-constructors

发现没有再调用拷贝构造函数，而是调用了移动构造函数。

移动构造函数的特点：

1.移动构造函数优于拷贝构造函数执行（实际上绑定左值也会经历这个过程，但是移动构造函数中的右值引用不能绑定左值，所以采用了拷贝构造函数）

2.移动构造函数如果不显式写出，编译器不会自动生成。

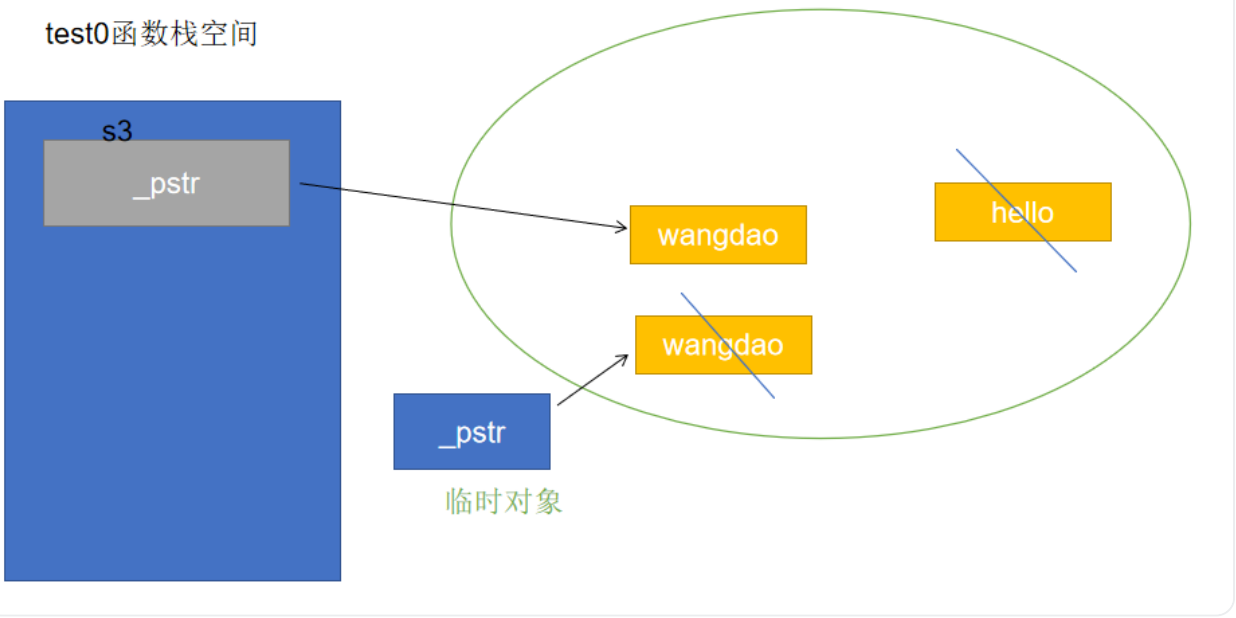
移动赋值函数（重要）

有了移动构造函数的成功经验，很容易想到原本的赋值运算符函数。

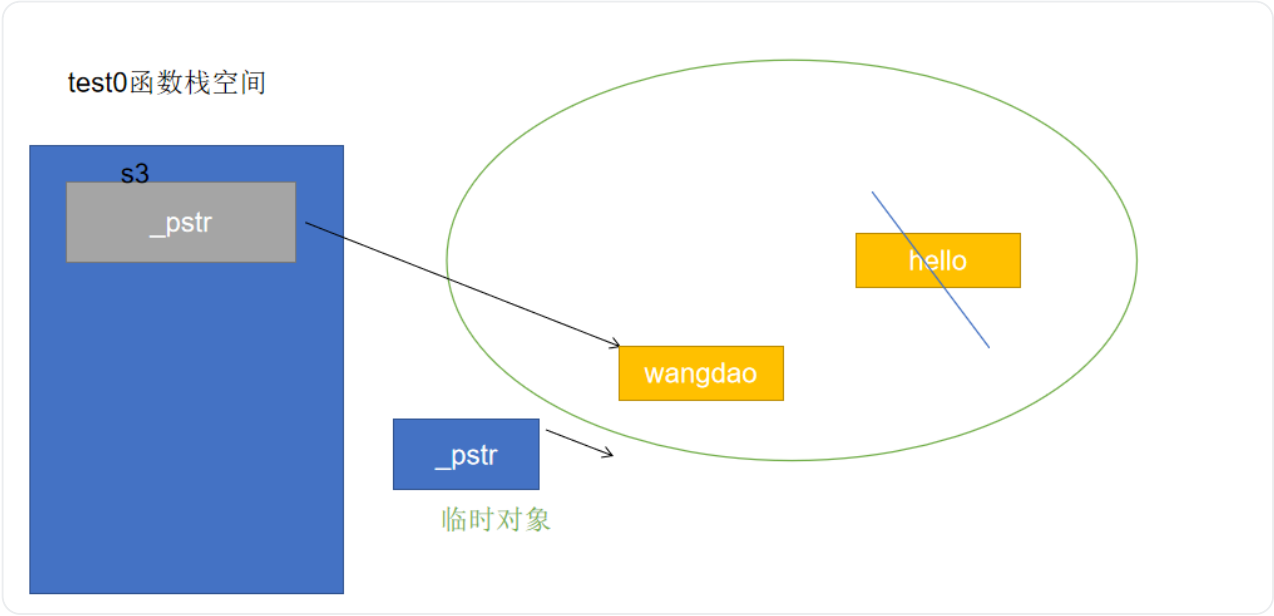
比如，我们进行如下操作时

```
String s3("hello");
s3 = String("wangdao");
```

原本赋值运算符函数的做法



我们希望复用临时对象申请的空间，那么也同样需要赋值运算符函数能够分辨出接收的参数是左值还是右值，同样可以利用右值引用



再写出移动赋值函数（移动赋值运算符函数），优先级也是高于赋值运算符函数

```
    String & operator=(String && rhs){
        if(this != &rhs){
            delete [] _pstr;
            //浅拷贝
            _pstr = rhs._pstr;
            rhs._pstr = nullptr;
            cout << "String& operator=(String&&)" << endl;
        }
        return *this;
    }
```

总结：

将拷贝构造函数和赋值运算符函数称为具有复制控制语义的函数；

将移动构造函数和移动赋值函数称为具有移动语义的函数(移交控制权)

具有移动语义的函数优于具有复制控制语义的函数执行；

具有移动语义的函数如果不显式写出，编译器不会自动生成，必须手写。

思考：移动赋值函数中的自复制判断是否还有必要？

```
    String s1("hello");
    //右值复制给左值，肯定不是同一个对象
    s1 = String("world");
    //创建了两个内容相同的临时对象，也不是同一对象
    String("wangdao") = String("wangdao");
```

似乎去掉自复制判断不会造成问题，但是c++11提出了一种方式，将左值转为右值，就是std::move函数

std::move函数

在一些使用移动语义的场景下，有时需要将左值转为右值。std::move函数的作用是显式的将一个左值转换为右值，**其实现本质上就是一个强制转换**。当将一个左值显式转换为右值后，原来的左值对象就无法正常工作了，必须要重新赋值才可以继续使用。

```
void test() {
    int a = 1;
    &(std::move(a)); //error，左值转成了右值

    String s1("hello");
    cout << "s1:" << s1 << endl;
    String s2 = std::move(s1);
    cout << "s1:" << s1 << endl;
    cout << "s2:" << s2 << endl;
}
```

如果将移动赋值函数的自复制判断去除，如下情况依然会调用移动赋值函数，但是s1的pstr所指向的空间被回收，且被设为了空指针，会出错

```
String s1("hello");
s1 = std::move(s1);
s1.print();
```

验证：将移动赋值函数中的浅拷贝去掉，让左操作数s1 的 `_pstr` 重新指向一片空间，后面对右操作数的 `_pstr` 设为空指针，但是依然造成了程序的中断，所以说明对std::move(s1)的内容进行修改，会导致s1的内容也被修改。

std::move的本质是在底层做了强制转换（并不是像名字表面的意思一样做了移动）

```
    String & operator=(String && rhs){
            delete [] _pstr;
            _pstr = new char[1]();
            rhs._pstr = nullptr;
            cout << "String& operator=(String&&)" << endl;
        return *this;
    }
```

—— 所以移动赋值函数的自复制判断不应该省略。

右值引用本身的性质

我们来定义一个返回值是右值引用的函数

```
int && func(){
    return 10;
}

void test1(){
    // &func();  //无法取址，说明返回的右值引用本身也是一个右值
    int && ref = func();
    &ref;  //可以取址，此时ref是一个右值引用，其本身是左值
}
```

右值引用本身是左值还是右值，取决于是否有名字，有名字就是左值，没名字就是右值。

```
String func2(){
    String str1("wangdao");
    str1.print();
    return str1;
}

void test2(){
    func2();
    //&func2(); //error,右值
    String && ref = func2();
    &ref;  //右值引用本身为左值
}
```

这里func2的调用按以前的理解会调用拷贝构造函数，但是发现结果是调用了移动构造函数。

当返回的对象其生命周期即将结束，就不再调用拷贝构造函数，而是调用移动构造函数。

如果返回的对象生命周期大于func3函数，执行return语句时还是调用拷贝构造函数

```
String s10("beijing");
String func3(){
    s10.print();
    return s10;
}

void test3(){
    func3();    //调用拷贝构造函数
}
```

总结：当类中同时定义移动构造函数和拷贝构造函数，需要对以前的规则进行补充，**调用哪个函数还需要取决于返回的对象的生命周期**。

资源管理

C语言在进行资源管理的时候，比如文件指针，由于分支较多，或者由于写代码的人与维护的人不一致，导致分支没有写的那么完善，从而导致文件指针没有释放。

```
void UseFile(char const* fn) {
    FILE* f = fopen(fn, “r”); //1. 获取资源
    //…… //2.使用资源
    //回收资源有很多分支
    if (!g()) { fclose(f); return; }
    // ...
    if (!h()) { fclose(f); return; }
    // ...
    fclose(f); // 释放资源
}
```

根据之前单例对象自动释放的经验，我们可以想到利用对象的生命周期去管理资源。那么就可以尝试实现一个安全回收文件的程序了。

```
class SafeFile
{
public:
```

```
    //在构造函数中初始化资源（托管资源）
    SafeFile(FILE * fp)
    : _fp(fp)
    {
        cout << "SafeFile(FILE*) " << endl;
    }
    //提供方法访问资源
    void write(const string & msg){
        fwrite(msg.c_str(),1,msg.size(),_fp);
    }
    //利用析构函数释放资源
    ~SafeFile(){
        cout << "~SafeFile()" << endl;
        if(_fp){
            fclose(_fp);
            cout << "fclose(_fp)" << endl;
        }
    }
private:
    FILE * _fp;
};

void test0(){
    string msg = "hello,world";
    SafeFile sf(fopen("wd.txt","a+"));
    sf.write(msg);
}
```

RAII技术

以上例子其实已经用到了RAII的技术。所谓RAII，是C++提出的资源管理的技术，全称为Resource Acquisition Is Initialization，由C++之父Bjarne Stroustrup提出。其本质是利用对象的生命周期来管理资源（内存资源、文件描述符、文件、锁等），因为当对象的生命周期结束时，会自动调用析构函数。

RAII类的常见特征

RAII技术，具备以下基本特征：

- 在构造函数中初始化资源，或托管资源；
- 在析构函数中释放资源；
- 一般不允许进行复制或者赋值（对象语义）；
- 提供若干访问资源的方法（如：读写文件）。

与对象语义相反的就是值语义。

值语义：可以进行复制或赋值（两个变量的值可以相同）

```
int a = 10; int b = a;  int c = 20;

c = a; //赋值

int d = c; //复制
```

对象语义：不允许复制或者赋值

（全世界不会有两个完全一样的人，程序世界中也不会有两个完全一样的对象）

常用手段：

1. 将拷贝构造函数与赋值运算符函数设置为私有的
2. 将拷贝构造函数与赋值运算符函数=delete
3. 使用继承的思想，将基类的拷贝构造函数与赋值运算符函数删除（或设为私有），让派生类继承基类。

RAII类的模拟

我们可以实现以下的一个类，模拟RAII的思想

```
template <class T>
class RAII
{
public:
    //1.在构造函数中初始化资源（托管资源）
```

```
    RAII(T * data)
    : _data(data)
    {
        cout << "RAII(T*)" << endl;
    }

    //2.在析构函数中释放资源
    ~RAII(){
        cout << "~RAII()" << endl;
        if(_data){
            delete _data;
            _data = nullptr;
        }
    }

    //3.提供若干访问资源的方法
    T * operator->(){
        return _data;
    }

    T & operator*(){
        return *_data;
    }

    T * get() const{
        return _data;
    }

    void set(T * data){
        if(_data){
            delete _data;
            _data = nullptr;
        }
        _data = data;
    }

    //4.不允许复制或赋值
    RAII(const RAII & rhs) = delete;
    RAII& operator=(const RAII & rhs) = delete;
private:
    T * _data;
};
```

如下，pt不是一个指针，而是一个对象，但是它的使用已经和指针完全一致了。这个对象可以托管堆上的Point对象，而且不用考虑delete.

```
void test0() {
    Point * pt = new Point(1, 2);
    //智能指针的雏形
    RAII<Point> raii(pt);
    raii->print();
    (*raii).print();
}
```

RAII技术的本质：利用**栈对象**的生命周期管理资源，因为栈对象在离开作用域时候，会执行析构函数。

智能指针

c++11提供了以下几种智能指针，位于头文件<memory>，它们都是类模板。

```
//std::auto_ptr          c++0x

//std::unique_ptr    c++11

//std::shared_ptr     c++11

//std::weak_ptr        c++11
```

auto_ptr的使用

auto_ptr是最简单的智能指针，使用上存在缺陷，已经被C++17弃用了。

auto_ptr是有复制、赋值函数的。

```
void test0(){
    int * pInt = new int(10);
    //创建auto_ptr对象接管资源
    auto_ptr<int> ap(pInt);
    cout << "*pInt:" << *pInt << endl;
    cout << "*ap:" << *ap << endl;
}
```

尽管会有warning提示，代码仍可通过。发现不用对pInt进行delete，也没有内存泄露。

auto_ptr可以进行复制，但是存在隐患

```
    auto_ptr<int> ap2(ap);
    cout << "*ap2:" << *ap2 << endl; //ok
    cout << "*ap:" << *ap << endl;
```

当ap2复制了ap后，对ap2管理的资源进行访问没有问题，但是对ap解引用会导致段错误。

通过阅读源码的实现，ap的指针被置为了空指针。

```
template <class _Tp>
class auto_ptr {
public:
   auto_ptr(auto_ptr& __a) __STL_NOTHROW
   : _M_ptr(__a.release())
   {}

   _Tp* release() __STL_NOTHROW
   {
    _Tp* __tmp = _M_ptr;
    _M_ptr = nullptr;
    return __tmp;
  }

private:
  _Tp* _M_ptr;
};
```

也就是说，`auto_ptr<int> ap2(ap);`这一步表面上执行了拷贝操作，但是底层已经将右操作数ap所托管的堆空间的控制权交给了左操作数ap2，并且将ap底层的指针数据成员置空，该拷贝操作存在隐患，所以auto_ptr被弃用了。

```
int * pInt = new int(10);
auto_ptr<int> ap(pInt);
/* auto_ptr<int> ap2(pInt); */

auto_ptr<int> ap2(ap);
cout << "*ap2:" << *ap2 << endl;
cout << "*pInt:" << *pInt << endl;
//猜测：通过auto_ptr的拷贝构造
//从ap拷贝出ap2对象
//实际是一个控制权移交的过程
//ap的指针被置空了
cout << "*ap:" << *ap << endl;
```

unique_ptr的使用（重要）

unique_ptr对auto_ptr进行了改进。

特点1：不允许复制或者赋值

具备对象语义。

特点2：独享所有权的智能指针

```
void test0(){
    unique_ptr<int> up(new int(10));
    cout << "*up:" << *up << endl;
    cout << "up.get(): " << up.get() << endl;

    cout << endl;
    //独享所有权的智能指针，对托管的空间独立拥有
    //拷贝构造已经被删除
    unique_ptr<int> up2 = up;//复制操作 error

    //赋值运算符函数也被删除
    unique_ptr<int> up3(new int(20));
    up3 = up;//赋值操作 error
}
```

将auto_ptr的缺陷摒弃了，具有对象语义，语法层面不允许复制、赋值。

```
unique_ptr<int> up(new int(10));
cout << "*up:" << *up << endl;
cout << "up.get():" << up.get() << endl;

cout << endl;

//unique_ptr不允许复制或赋值
/* unique_ptr<int> up2 = up; */
/* unique_ptr<int> up3(new int(20)); */
/* up = up3; */
```

特点3：作为容器元素

要利用**移动语义**的特点，可以直接传递unique_ptr的右值作为容器的元素。如果传入左值形态的unique_ptr，会进行复制操作，而unique_ptr是不能复制的。

构建右值的方式有

1、std::move的方式

2、可以直接使用unique_ptr的构造函数，创建匿名对象（临时对象），构建右值

```
    vector<unique_ptr<Point>> vec;
    unique_ptr<Point> up4(new Point(10,20));
    //up4是一个左值
    //将up4这个对象作为参数传给了push_back函数，会调用拷贝构造
    //但是unique_ptr的拷贝构造已经删除了
    //所以这样写会报错
    vec.push_back(up4);  //error

    vec.push_back(std::move(up4));  //ok
    vec.push_back(unique_ptr<Point>(new Point(1,3))); //ok

```

```
void test1(){
    unique_ptr<Point> up(new Point(4,9));
    vector<unique_ptr<Point>> vec;
    //往vector当中存左值，会发生复制
    //但是unique_ptr的拷贝构造已经删除
    /* vec.push_back(up); */
    vec.push_back(std::move(up));
    vec.push_back(unique_ptr<Point>(new Point(2,8)));
}
```

shared_ptr的使用（重要）

智能指针独享资源的控制权固然是一种需求，但有些场景下也需要允许共享控制权。

shared_ptr就是共享所有权的智能指针，可以进行复制或赋值，但复制或赋值时，并不是真正拷贝对象，而只是将引用计数加1了。即shared_ptr引入了引用计数，其思想与COW技术类似，又称为是强引用的智能指针。

特征1：共享所有权的智能指针

可以使用**引用计数**记录对象的个数。

特征2：可以进行复制或者赋值

表明具备值语义。

特征3：也可以作为容器的元素

作为容器元素的时候，即可以传递左值，也可以传递右值。（区别于unique_ptr只能传右值）

特征4：也具备移动语义

表明也有移动构造函数与移动赋值函数。

```
shared_ptr<int> sp(new int(10));
    cout << "sp.use_count(): " << sp.use_count() << endl;

    cout << endl;
    cout << "执行复制操作" << endl;
    shared_ptr<int> sp2 = sp;
    cout << "sp.use_count(): " << sp.use_count() << endl;
    cout << "sp2.use_count(): " << sp2.use_count() << endl;

    cout << endl;
    cout << "再创建一个对象sp3" << endl;
    shared_ptr<int> sp3(new int(30));
    cout << "sp.use_count(): " << sp.use_count() << endl;
    cout << "sp2.use_count(): " << sp2.use_count() << endl;
    cout << "sp3.use_count(): " << sp3.use_count() << endl;

    cout << endl;
    cout << "执行赋值操作" << endl;
    sp3 = sp;
    cout << "sp.use_count(): " << sp.use_count() << endl;
    cout << "sp2.use_count(): " << sp2.use_count() << endl;
    cout << "sp3.use_count(): " << sp3.use_count() << endl;
    cout << "*sp:" << *sp << endl;
    cout << "*sp2:" << *sp2 << endl;
    cout << "*sp3:" << *sp3 << endl;
    cout << "sp.get():" << sp.get() << endl;
    cout << "sp2.get():" << sp2.get() << endl;
    cout << "sp3.get():" << sp3.get() << endl;

```

```
shared_ptr<int> sp(new int(10));
cout << "*sp:" << *sp << endl;
cout << sp.get() << endl;

//复制操作
cout << endl;
shared_ptr<int> sp2 = sp;
cout << "*sp:" << *sp << endl;
cout << sp.get() << endl;
cout << "*sp2:" << *sp2 << endl;
cout << sp2.get() << endl;

cout << sp.use_count() << endl;
cout << sp2.use_count() << endl;

//赋值操作
cout << endl;
shared_ptr<int> sp3(new int(20));
sp2 = sp3;
cout << sp.use_count() << endl;
cout << sp2.use_count() << endl;
cout << sp3.use_count() << endl;

//作为容器元素
vector<shared_ptr<int>> vec;
vec.push_back(sp);
vec.push_back(std::move(sp2));
```

shared_ptr的循环引用

shared_ptr还存在一个问题 —— 循环引用问题。

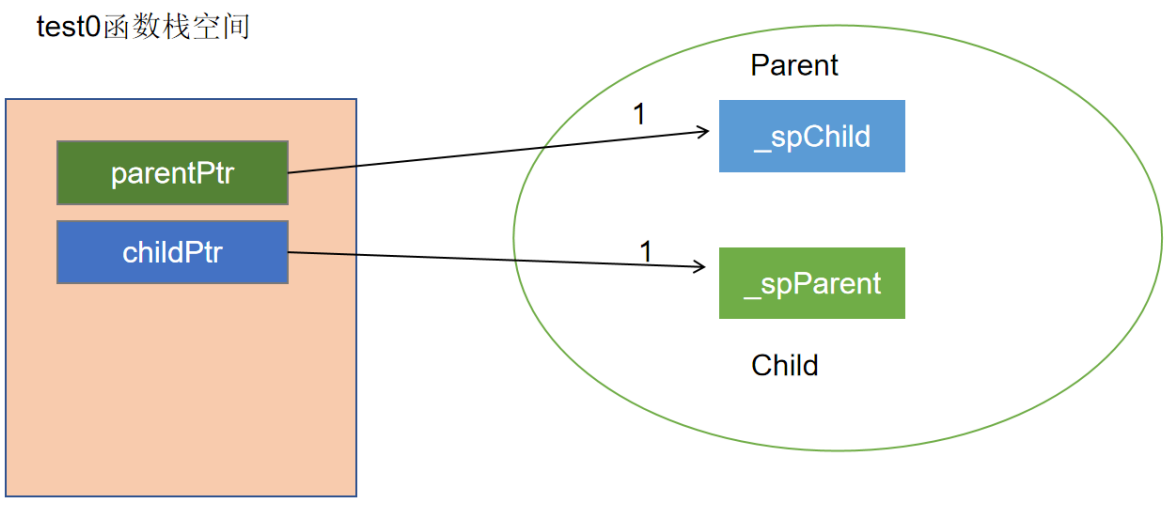
我们建立一个Parent和Child类的一个结构

```
1   class Child;
2
3   class Parent
4   {
5   public:
6    Parent()
7    { cout << "Parent()" << endl; }
8    ~Parent()
9    { cout << "~Parent()" << endl; }
10   //只需要Child类型的指针，不需要类的完整定义
11   shared_ptr<Child> spChild;
12  };
13
```

```
class Child
{
public:
 Child()
 { cout << "child()" << endl; }
 ~Child()
 { cout << "~child()" << endl; }
 shared_ptr<Parent> spParent;
};
```

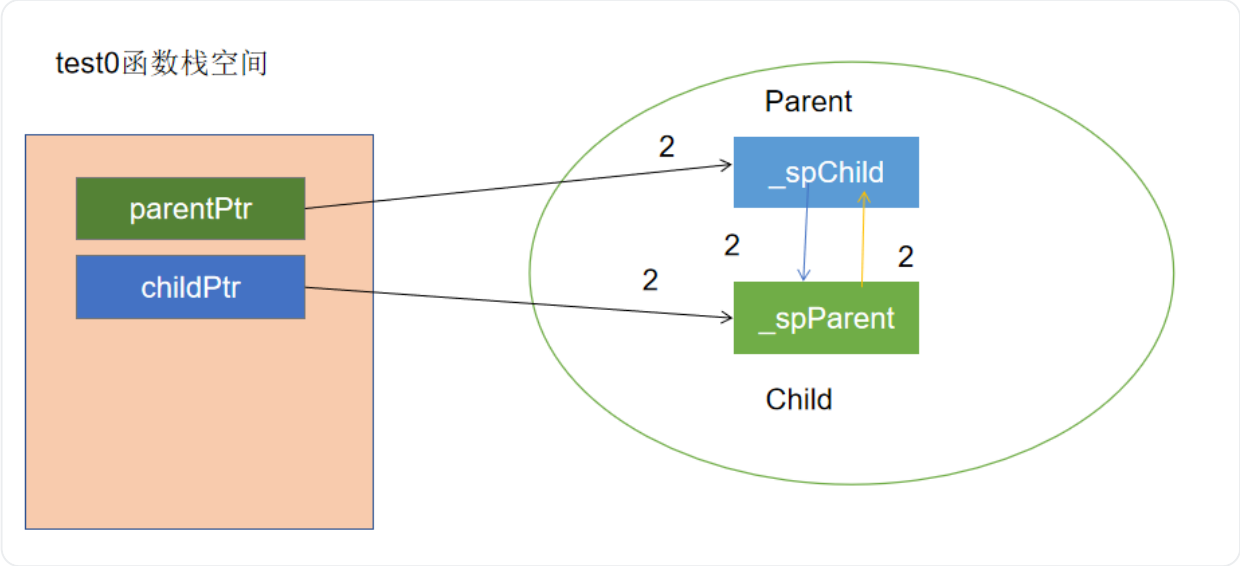
由于shared_ptr的实现使用了引用计数，那么如果进行如下的创建

```
shared_ptr<Parent> parentPtr(new Parent());
 shared_ptr<Child> childPtr(new Child());
 //获取到的引用计数都是1
 cout << "parentPtr.use_count():" << parentPtr.use_count() <<
endl;
 cout << "childPtr.use_count():" << childPtr.use_count() << endl;
```

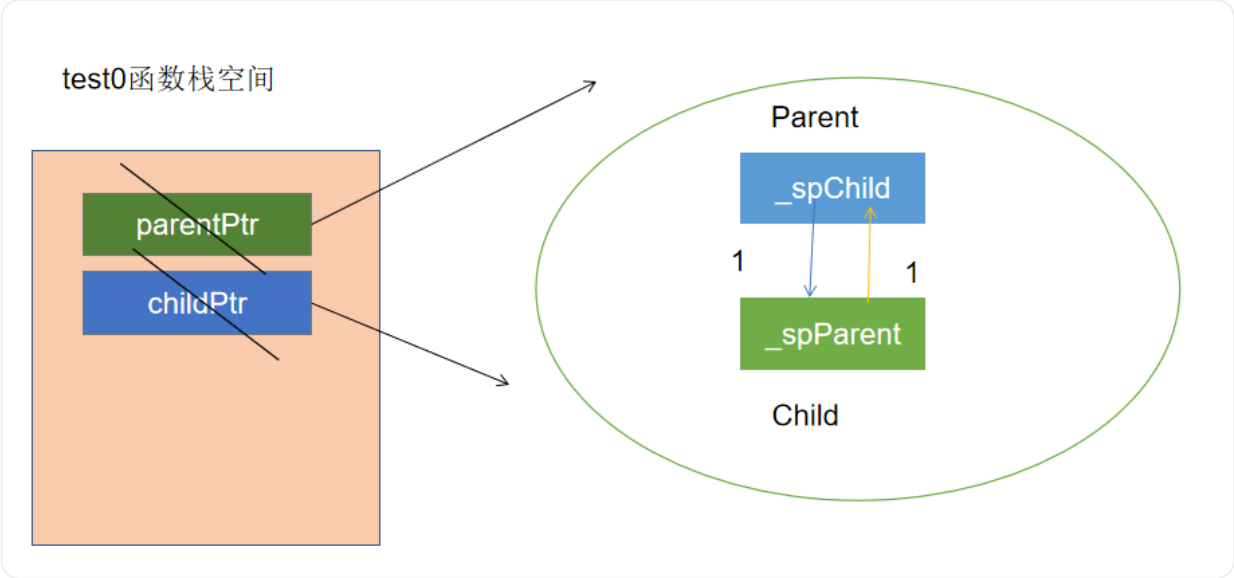


```
parentPtr->spChild = childPtr;
 childPtr->spParent = parentPtr;
 //获取到的引用计数都是2
 cout << "parentPtr.use_count():" << parentPtr.use_count() <<
endl;
 cout << "childPtr.use_count():" << childPtr.use_count() << endl;
```

——程序结束时，发现Parent和child的析构函数都没有被调用



childPtr和parentPtr会先后销毁，但是堆上的Parent对象和Child对象的引用计数都变成了1，而不会减到0，所以没有回收



解决思路：

——希望某一个指针指向一片空间，能够指向，但是不会使引用计数加1，那么堆上的Parent对象和Child对象必然有一个的引用计数是1，栈对象再销毁的时候，就可以使引用计数减为0

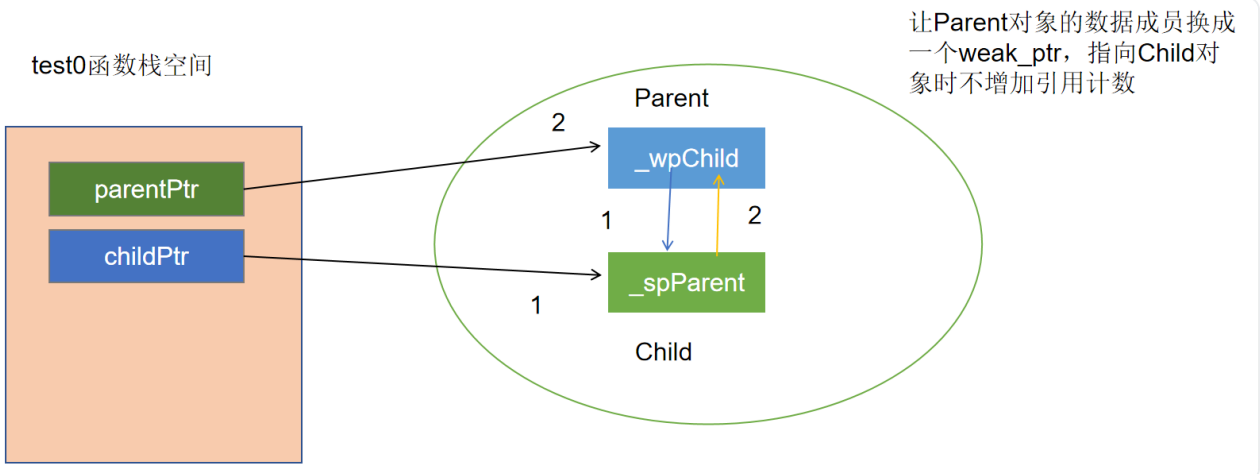
shared_ptr无法实现这一效果，所以引入了weak_ptr.

> weak_ptr是一个弱引用的智能指针，不会增加引用计数。
>
> shared_ptr是一个强引用的智能指针。
>
> 强引用，指向一定会增加引用计数，只要有一个引用存在，对象就不能释放；
>
> 弱引用并不增加对象的引用计数，但是它知道所托管的对象是否还存活。

——循环引用的解法，将Parent类或Child类中的任意一个shared_ptr换成weak_ptr类型的智能指针

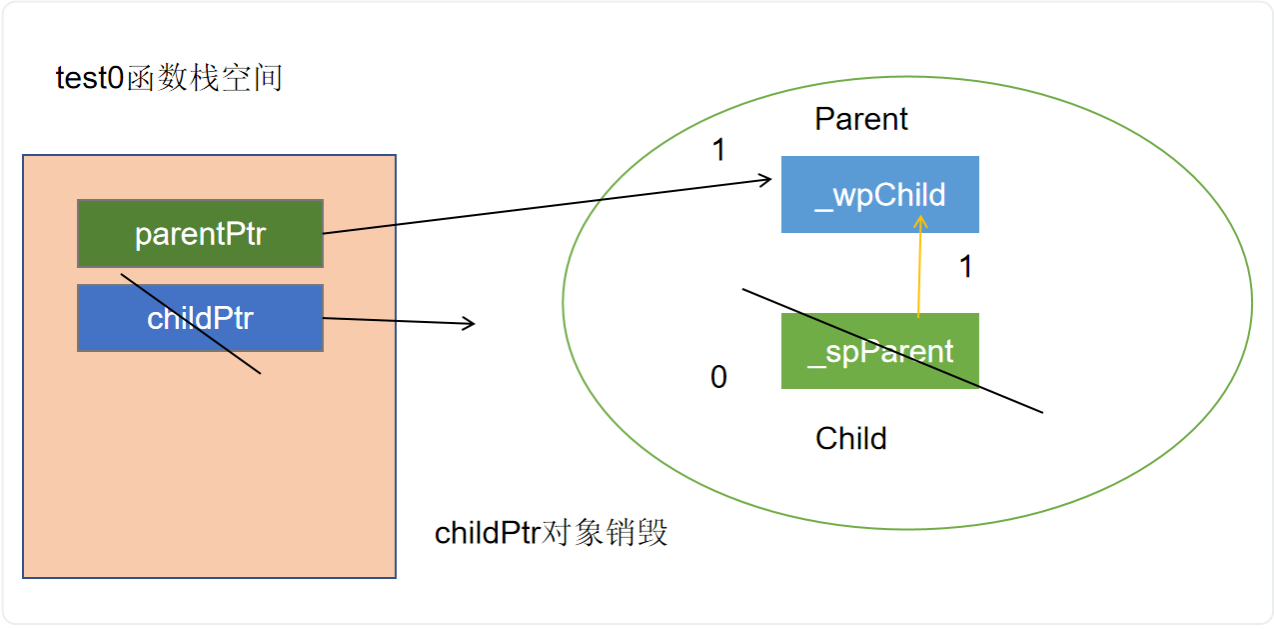
比如：将Parent类中的shared_ptr类型指针换成weak_ptr

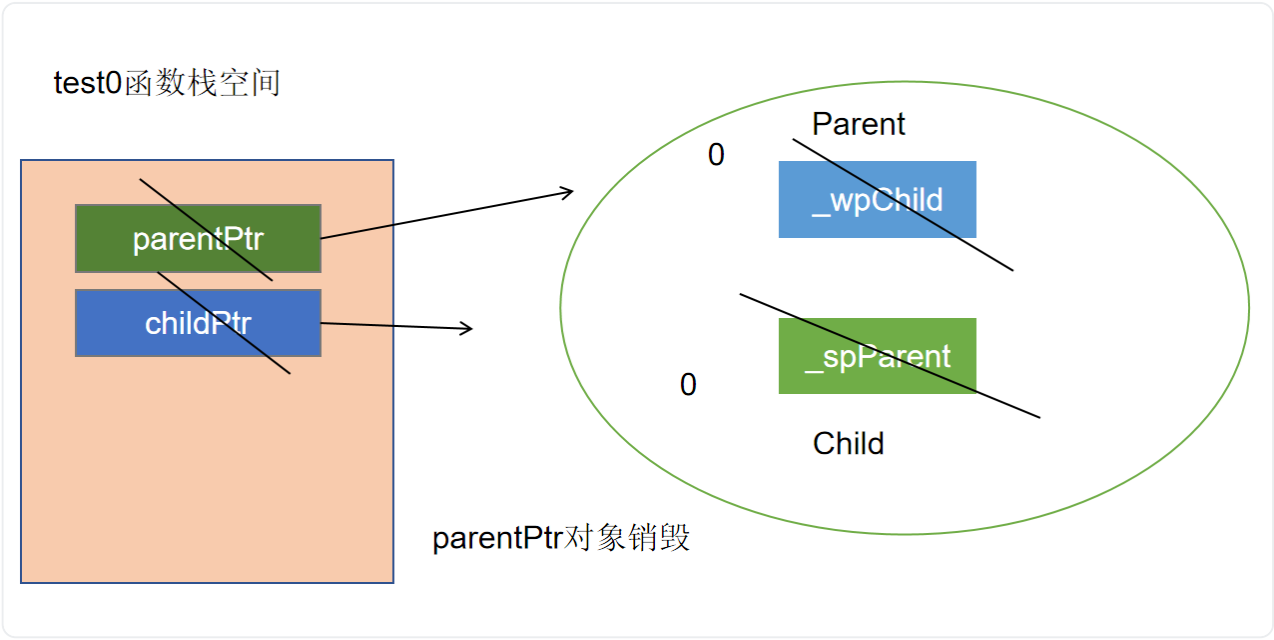
```
//parentPtr是一个管理Parent对象的智能指针
//可以利用箭头运算符访问它所管理的Parent对象的成员
//_spChild就是Parent对象的成员
//同时也是一个能够管理Child对象的智能指针
//因为shared_ptr类型的智能指针可以进行赋值操作
//所以可以使_spChild也能管理childPtr所管理的对象
parentPtr->_wpChild = childPtr;
childPtr->_spParent = parentPtr;
//获取到的引用计数都是2
cout << "parentPtr.use_count():" << parentPtr.use_count() << endl;
cout << "childPtr.use_count():" << childPtr.use_count() << endl;
```



栈上的childPtr对象先销毁，会使堆上的Child对象的引用计数减1，因为这个Child对象的引用计数本来就是1，所以减为了0，回收这个Child对象，造成堆上的Parent对象的引用计数也减1,;

再当parentPtr销毁时，会再让堆上的Parent对象的引用计数减1，所以也能够回收。





weak_ptr的使用

weak_ptr是弱引用的智能指针，它是shared_ptr的一个补充，使用它进行复制或者赋值时，并不会导致引用计数加1，是为了解决shared_ptr的问题而诞生的。

weak_ptr知道所托管的对象是否还存活，如果存活，必须要提升为shared_ptr才能对资源进行访问，不能直接访问。

初始化

```
 weak_ptr<int> wp;//无参的方式创建weak_ptr

//也可以利用shared_ptr创建weak_ptr
weak_ptr<int> wp2(sp);
```

判断关联的空间是否还在

1.可以直接使用use_count函数

如果use_count的返回值大于0，表明关联的空间还在

2.将weak_ptr提升为shared_ptr

```
shared_ptr<int> sp(new int(10));
 weak_ptr<int> wp;//无参的方式创建weak_ptr
 wp = sp;//赋值
```

这种赋值操作可以让wp也能够托管这片空间，但是它作为一个weak_ptr仍不能够去管理，甚至连访问都不允许（weak_ptr不支持直接解引用）

想要真正地去进行管理需要使用lock函数将weak_ptr提升为shared_ptr

```
shared_ptr<int> sp2 = wp.lock();
 if(sp2){
     cout << "提升成功" << endl;
     cout << *sp2 << endl;
 }else{
     cout << "提升失败，托管的空间已经被销毁" << endl;
 }
```

```
shared_ptr<int> sp3 = wp.lock();
if(sp3){
    cout << "提升成功" << endl;
    cout << *sp3 << endl;
}else{
    cout << "提升失败，没有托管的空间" << endl;
}
```

如果托管的资源没有被销毁，就可以成功提升为shared_ptr，否则就会返回一个空的shared_ptr（空指针）

——查看lock函数的说明

```
std::shared_ptr<T> lock() const noexcept;
//将weak_ptr提升成一个shared_ptr，然后再来判断shared_ptr,进而知道weak_ptr指向的空间还在不在
```

3.**可以使用expired函数**

```
bool expired() const noexcept;
//weak_ptr去判断托管的资源有没有被回收
```

该函数返回true等价于use_count() == 0.

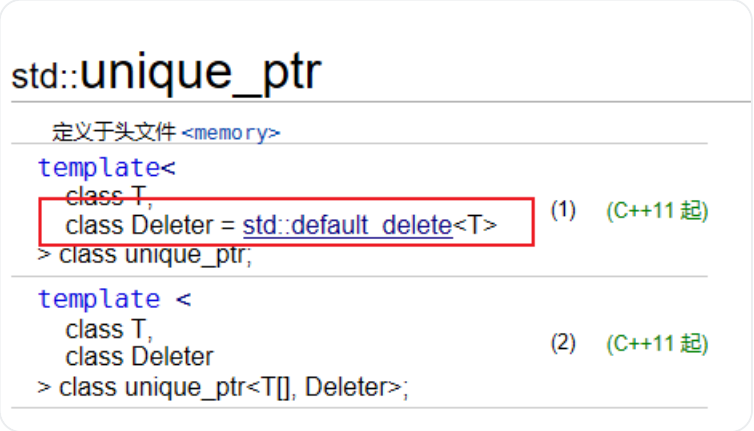
```
bool flag = wp.expired();
if(flag){
     cout << "托管的空间已经被销毁" << endl;
 }else{
     cout << "托管的空间还在" << endl;
 }
```

```
cout << endl;
//expired()返回true代表没有空间被托管
//返回false代表有空间被托管
if(!wp.expired()){
    cout << "有空间" << endl;
}else{
    cout << "没有空间" << endl;
}
```

删除器

很多时候我们都用new来申请空间，用delete来释放。库中实现的各种智能指针，默认也都是用delete来释放空间。但是若我们采用malloc申请的空间或是用fopen打开的文件，这时智能指针的默认处理方式就不能解决了，必须为智能指针定制删除器，这样，我们的智能指针就可以定制化释放资源的方式了。

unique_ptr对应的删除器



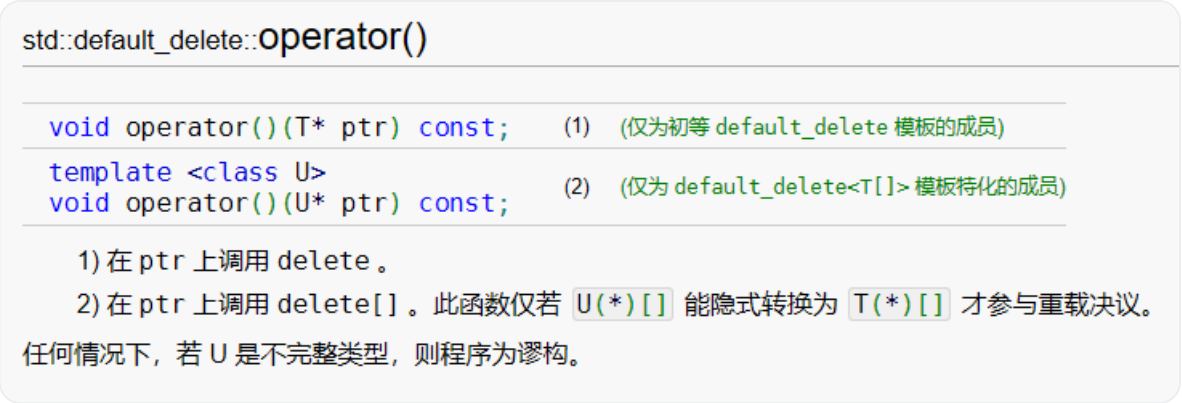
std::unique_ptr

定义于头文件 <memory>

```
template<
    class T,
    class Deleter = std::default_delete<T>
> class unique_ptr;                                  (1)   (C++11 起)

template <
    class T,
    class Deleter
> class unique_ptr<T[], Deleter>;                    (2)   (C++11 起)
```

定义unique_ptr时，如果没有指定删除器参数，就会使用默认的删除器。点开std::default_delete的说明



std::default_delete::operator()

```
void operator()(T* ptr) const;                      (1)   (仅为初等 default_delete 模板的成员)

template <class U>
void operator()(U* ptr) const;                      (2)   (仅为 default_delete<T[]> 模板特化的成员)
```

1) 在 ptr 上调用 delete 。

2) 在 ptr 上调用 delete[] 。此函数仅若 U(*)[] 能隐式转换为 T(*)[] 才参与重载决议。

任何情况下，若 U 是不完整类型，则程序为谬构。

无论接管的是什么类型的资源，回收时都是会执行delete语句或delete [ ]

看下面这个例子，利用unique_ptr管理文件资源，出现问题

```
1  void test0(){
2      string msg = "hello,world\n";
3      FILE * fp = fopen("res1.txt","a+");
4      fwrite(msg.c_str(),1,msg.size(),fp);
5      fclose(fp);
6  }
7
8  void test1(){
9      string msg = "hello,world\n";
10     unique_ptr<FILE> up(fopen("res2.txt","a+"));
11     //get函数可以从智能指针中获取到裸指针
12     fwrite(msg.c_str(),1,msg.size(),up.get());
13     //fclose(up.get());
14 }
```

问题的原因：接管文件资源时，也使用了delete语句，导致错误

——需要自定义删除器

仿照参考文档上默认删除器的示例，创建一个代表删除器的struct，定义operator()函数

```
struct FILECloser{
    void operator()(FILE * fp){
        if(fp){
            fclose(fp);
            cout << "fclose(fp)" << endl;
        }
    }
};
```

创建unique_ptr接管文件资源时，删除器参数使用我们自定义的删除器

```
void test1(){
    string msg = "hello,world\n";
    unique_ptr<FILE, FILECloser> up(fopen("res2.txt","a+"));
    //get函数可以从智能指针中获取到裸指针
    fwrite(msg.c_str(),1,msg.size(),up.get());
}
```

如果管理的是普通的资源，不需要写出删除器，就使用默认的删除器即可，只有针对FILE或者socket这一类创建的资源，才需要改写删除器，使用fclose之类的函数。

shared_ptr对应的删除器

unique_ptr 和 shared_ptr区别：

对于unique_ptr，删除器是模板参数

对于shared_ptr，删除器是构造函数参数

std::shared_ptr<T>::shared_ptr

```
constexpr shared_ptr() noexcept;
constexpr shared_ptr( std::nullptr_t ) noexcept;
template< class Y >
explicit shared_ptr( Y* ptr );
template< class Y, class Deleter >
shared_ptr( Y* ptr, Deleter d );
```

所以传入删除器参数的位置不同

```
void test2(){
    string msg = "hello,world\n";
    FILECloser fc;
    //在shared_ptr的构造函数参数中加入删除器对象
    shared_ptr<FILE> sp(fopen("res3.txt","a+"),fc);
    fwrite(msg.c_str(),1,msg.size(),sp.get());
}
```

```
void test1(){
    string msg = "hello,world\n";
    unique_ptr<FILE,FILECloser> up(fopen("res2.txt","a+"));
    //get函数可以从智能指针中获取到裸指针
    fwrite(msg.c_str(),1,msg.size(),up.get());
    //fclose(up.get());
}

void test2(){
    string msg = "hello,world\n";
    FILECloser fc;
    shared_ptr<FILE> up(fopen("res3.txt","a+"),fc);
    //get函数可以从智能指针中获取到裸指针
    fwrite(msg.c_str(),1,msg.size(),up.get());
    //fclose(up.get());
}
```

智能指针的误用

智能指针被误用的情况，**原因都是将一个原生裸指针交给了不同的智能指针进行托管，而造成一个对象被销毁两次**。

对于shared_ptr与unique_ptr都会产生这个问题。

—— unique_ptr要注意的误用

```
void test0(){
 //需要人为注意避免
 Point * pt = new Point(1,2);
 unique_ptr<Point> up(pt);
 unique_ptr<Point> up2(pt);
}

void test1(){
 unique_ptr<Point> up(new Point(1,2));
 unique_ptr<Point> up2(new Point(1,2));
 //让两个unique_ptr对象托管了同一片空间
 up.reset(up2.get());
}
```

```
void test0(){
    //需要人为注意避免
    Point * pt = new Point(1,2);
    unique_ptr<Point> up(pt);
    unique_ptr<Point> up2(pt);
}

void test1(){
    unique_ptr<Point> up(new Point(1,2));
    unique_ptr<Point> up2(new Point(1,2));
    up.reset(new Point(8,9));
    up.reset(up2.get());
}
```

——shared_ptr要注意的误用

使用不同的智能指针托管同一片堆空间，即使是shared_ptr也是不行的。

之前进行的shared_ptr的复制、赋值的参数都是shared_ptr的对象，不能直接多次把同一个裸指针传给它的构造。

```
void test2(){
 Point * pt = new Point(10,20);
 shared_ptr<Point> sp(pt);
 shared_ptr<Point> sp2(pt);
}

void test3(){
 //使用不同的智能指针托管同一片堆空间
 shared_ptr<Point> sp(new Point(1,2));
 shared_ptr<Point> sp2(new Point(1,2));
 sp.reset(sp2.get());
}
```

—— 还有一种误用

给Point类加入了这样的成员函数

```
Point * addPoint(Point * pt){
 _ix += pt->_ix;
 _iy += pt->_iy;
 return this;
}
```

使用时，这样还是使得sp3和sp同时托管了同一个堆对象

```
shared_ptr<Point> sp(new Point(1,2));
shared_ptr<Point> sp2(new Point(3,4));

//创建sp3的参数实际上是sp所对应的裸指针
 //效果还是多个智能指针托管了同一块空间
 shared_ptr<Point> sp3(sp->addPoint(sp2.get()));
 cout << "sp3 = ";
 sp3->print();
```

——需要给sp3的构造函数传入`shared_ptr<Point>`对象，而不是裸指针

解决思路：**通过this指针获取本对象的shared_ptr**

可以修改Point中的addPoint函数

```
    shared_ptr<Point> addPoint(Point * pt){
        _ix += pt->_ix;
        _iy += pt->_iy;
        return shared_ptr<Point>(this);
    }
```

但是这样写，在addPoint函数中创建的匿名智能指针对象接收的还是sp对应的裸指针，那么这个匿名对象和sp所托管的空间还是同一片空间。匿名对象销毁时会delete一次，sp销毁时又会delete一次。

——使用智能指针辅助类enable_shared_from_this的成员函数shared_from_this

enable_shared_from_this (C++11)	允许对象创建指代自身的 shared_ptr (类模板)

成员函数

(构造函数)	构造 enable_shared_from_this 对象 (受保护成员函数)
(析构函数)	销毁 enable_shared_from_this 对象 (受保护成员函数)
operator=	返回到 `this` 的引用 (受保护成员函数)
shared_from_this	返回共享 `*this` 所有权的 `shared_ptr` (公开成员函数)

在Point的addPoint函数中需要使用shared_from_this函数返回的shared_ptr作为返回值，要想在Point类中调用enable_shared_from_this的成员函数，最佳方案可以让Point类继承enable_shared_from_this类。

这样修改addPoint函数后，问题解决。

```
class Point
: public std::enable_shared_from_this<Point>
{
public:
    //...
    shared_ptr<Point> addPoint(Point & pt) {
        _ix += pt._ix;
        _iy += pt._iy;
        return shared_from_this();
    }
};
```

```
shared_ptr<Point> addPoint(Point * pt){
    _ix += pt->_ix;
    _iy += pt->_iy;
    return shared_from_this();
    /* return shared_ptr<Point>(this); */
    /* return this; */
}
```

```
void test0(){
    shared_ptr<Point> sp(new Point(1,2));
    shared_ptr<Point> sp2(new Point(3,4));

    //创建sp3的参数实际上是sp所对应的裸指针
    //效果还是多个智能指针托管了同一块空间
    shared_ptr<Point> sp3(sp->addPoint(sp2.get()));
    cout << "sp3 = ";
    sp3->print();
}
```

总结：智能指针的误用全都是使用了不同的智能指针托管了同一块堆空间（同一个裸指针）。