移动语义

1、几个基本概念的理解

左值、右值、const左值引用、右值引用?

可以取地址的是左值，不能取地址的就是右值。**右值可能存在寄存器，也可能存在于栈上(短暂存在栈上)**。

右值包括：临时对象、匿名对象、字面值常量

const左值引用可以绑定到左值与右值上面。正因如此，也就无法区分传进来的参数是左值还是右值。

右值引用只能绑定到右值，不能绑定到左值。所以可以区分出传进来的参数到底是左值还是右值，进而可以区分。

右值引用到底是左值还是右值?

这个与右值引用本身有没有名字有关，如果是int &&rref = 10,右值引用本身就是左值，因为有名字。如果右值引用本身没有名字，那右值引用就是右值，如右值引用作为函数返回值。。

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

2、移动构造函数:

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

3、移动赋值运算符函数

```
String &operator=(String &&rhs)
{
    cout << "String &operator=(String &&)" << endl;
    if(this != &rhs)//1、自移动
    {
        delete [] _pstr;//2、释放左操作数
        _pstr = nullptr;
        _pstr = rhs._pstr;//3、浅拷贝
        rhs._pstr = nullptr;
    }

    return *this;//4、返回*this
}
```

移动函数(移动构造函数和移动赋值运算符函数)**优先于**复制函数(拷贝构造函数和赋值运算符函数)的执行

具有移动语义的函数(移动构造函数和移动赋值运算符函数)**优先于**具有复制控制语义函数(拷贝构造函数和赋值运算符函数)的执行

4、std::move函数

将左值转换为右值，在内部其实上是做了一个强制转换，static_cast<T &&>(lvaule)。将左值转换为右值后，左值就不能直接使用了，如果还想继续使用，必须重新赋值。。。std::move()作用于内置类型没有任何作用，内置类型本身是左值还是右值，经过std::move()后不会改变。

```
//面试题：关于实现String
class String
{
public:
    //.......
    //(当传递右值的时候)具有移动语义的函数优先于具有复制控制语义的函数
    //移动构造函数、移动赋值运算符函数称为具有移动语义的函数
    //移动构造函数(只针对右值)
    String(String &&rhs)
    : _pstr(rhs._pstr)
    {
        cout << "String(String &&)" << endl;
        rhs._pstr = nullptr;
    }

    //移动赋值运算符函数
    String &operator=(String &&rhs)
    {
        cout << "String &operator=(String &&)" << endl;
        if(this != &rhs)//考虑自移动
        {
            delete [] _pstr;//释放左操作数
            _pstr = nullptr;
            _pstr = rhs._pstr;//转移右操作数的资源
            rhs._pstr = nullptr;//右操作数置位空
        }

        return *this;//返回*this
    }

    //.........
private:
    char *_pstr;
};

void test2()
{
    String s1("hello");
    cout << "s1 = " << s1 << endl;

    cout << endl << "执行String s2(s1) " << endl;
    String s2(s1);//传递的是一个左值，拷贝构造函数
    cout << "s2 = " << s2 << endl;

    cout << endl << "执行Stirng s3 = String(world) " << endl;
    String s3 = String("world");//传递的是一个右值，拷贝构造函数
    cout << "s3 = " << s3 << endl;
}
```

```
49
50  void test3()
51  {
52      String s1("hello");
53      String s2("world");//传递的是一个左值，拷贝构造函数
54
55      s2 = String("world");//赋值
56      /* String("World") = String("World"); */
57
58      s1 = std::move(s1);//将左值转换为右值，在内部其实上是做了一个强转转换
59                         //static_cast<T &&>(lvaule)
60                         //如果把左值转换为右值其实就是不想在使用这个左值了
61      cout << "s1 = " << s1 << endl;
62      //s1是左值，现在讲s1转换成右值，右值s1 = s1
63      s1 = std::move(s2);
64      cout << "s1 = " << s1 << endl;
65
66      //如果对左值执行move函数执行，其就变成一个右值，就不能使用了
67      //假如还想继续使用，必须重新复制
68      s2 = "welcome";
69      cout << "s2 = " << s2 << endl;
70  }
```

如何区分出左值，右值，左值引用，右值引用，const左值引用。

```
1   void test()
2   {
3       int a = 10;
4       int b = 20;
5       int *pflag = &a;
6       vector<int> vec;
7       vec.push_back(1);
8       string s1 = "hello";
9       string s2 = "world";
10
11      const int &ref = a;//const左值引用可以绑定到左值
12      const int &ref1 = 10;//const左值引用可以绑定到右值
13      &ref;
14      &ref1;
15
16      //右值引用
17      const int &&rref = 100;//右值引用可以绑定到右值
18      /* const int &&rref2 = b;//右值引用不能绑定到左值 */
19      &rref;//此时右值引用是一个左值
20
21      &a;//左值
22      &b;
23      &pflag;
24      &(*pflag);
25      &vec[0];
26      &s1;
27      &s2;
28
29
30      /* &(a + b);//error,右值 */
31      /* &(s1 + s2);//error,右值 */
32      /* &(a++);//error; */
```

```
    &(++a);//ok
    /* &100;//error,右值，字面值常量 */
    /* &string("hello");//error,右值 */
}

int &&func2()
{
    return 10;
}

void test2()
{
    int number1 = 10;
    int number2 = 20;
    std::move(number1);
    cout << "number1 = " << number1 << endl;
    cout << "number2 = " << number2 << endl;
}

```

资源管理与智能指针

一、C语言中的问题

C语言在对资源管理的时候，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，所以可以使用C++的方式管理文件指针。。。

```
class SafeFile
{
public:
    //在构造时候托管资源(fp)
    SafeFile(FILE *fp)
    : _fp(fp)
    {
        cout << "SafeFile(FILE *)" << endl;
        if(nullptr == _fp)
        {
            cout << "nullptr == _fp " << endl;
        }
    }

    //提供若干访问资源的方法
    void write(const string &msg)
    {
        fwrite(msg.c_str(), 1, msg.size(), _fp);//调用C语言的函数
    }

    //在销毁(析构)时候释放资源(fp)
    ~SafeFile()
    {
        cout << "~SafeFile()" << endl;
        if(_fp)
        {
```

```
            fclose(_fp);
            cout << "fclose(_fp)" << endl;
        }
    }

private:
    FILE *_fp;
};

void test()
{
    string s1 = "hello,world\n";
    SafeFile sf(fopen("test.txt", "a+"));//其实就是利用栈对象sf的生命周期管理文件指针的资源
    sf.write(s1);
}
```

二、C++的解决办法(RAII技术):

资源管理RAII技术，利用对象的生命周期管理程序资源(包括内存、文件句柄、锁等)的技术,因为对象在离开作用域的时候，会自动调用析构函数。

关键：要保证资源的释放顺序与获取顺序严格相反。。正好是析构函数与构造函数的作用。。。

RAII常见特征

1、在构造时初始化资源，或者托管资源。
2、析构时释放资源。
3、一般不允许复制或者赋值(值语义-对象语义)
4、提供若干访问资源的方法。

区分：值语义：可以进行复制与赋值。

对象语义：不能进行复制与赋值，一般使用两种方法达到要求：

(1)、将拷贝构造函数和赋值运算符函数设置为私有的就ok。
(2)、将拷贝构造函数和赋值运算符函数使用=delete.

```
template <typename T>
class RAII
{
public:
    //通过构造函数托管资源
    RAII(T *data)
    : _data(data)
    {
        cout << "RAII(T *)" << endl;
    }
    //访问资源的方法
    T *operator->()
    {
        return _data;
    }

    T &operator*()
    {
```

```
        return  *_data;
    }

    T *get() const
    {
        return _data;
    }

    void reset(T *data)
    {
        if(_data)
        {
            delete _data;
            _data = nullptr;
        }
        _data = data;
    }
    RAII(const RAII &rhs) = delete;
    RAII &operator=(const RAII &rhs) = delete;

    //通过析构函数释放资源
    ~RAII()
    {
        cout << "~RAII()" << endl;
        if(_data)
        {
            delete _data;
            _data = nullptr;
        }
    }

private:
    T *_data;
};

void test()
{
    /* Point *pt = new Point(1, 2); */
    /* delete pt; */
    //vector<Point>
    //ppt本身是一个RAII的栈对象
    //ppt他的使用类似于一个指针
    RAII<Point> ppt(new Point(1, 2));
    cout << "ppt = ";
    ppt->print();
    //ppt.operator->().print();

    cout << endl;
    /* RAII<Point> ppt2 = ppt; */
}
```

三、四种智能指针

RAII的对象ppt就有智能指针的雏形。

1、auto_ptr.cc

最简单的智能指针，使用上存在缺陷，所以被弃用。。。(C++17已经将其删除了)

```
void test()
{
    int *pt = new int(10);
    auto_ptr<int> ap(pt);
    cout << "*ap = " << *ap << endl;
    cout << "ap.get() = " << ap.get() << endl;
    cout << "pt = " << pt  << endl;

    cout << endl << endl;
    auto_ptr<int> ap2(ap);//表面上执行拷贝构造函数,但是在底层已经发生了所有权(资源的)的转移
                          //该智能指针存在缺陷
    cout << "*ap2 = " << *ap2 << endl;
    cout << "*ap = " << *ap << endl;//core dump
}
```

2、unique_ptr

比auto_ptr安全多了，明确表明是独享所有权的智能指针，所以不能进行复制与赋值。

```
void test()
{
    unique_ptr<int> up(new int(10));
    cout << "*up = " << *up << endl;
    cout << "up.get() = " << up.get() << endl;//获取托管的指针的值

    cout << endl << endl;
    /* unique_ptr<int> up2(up);//error,独享资源的所有权,不能进行复制 */

    cout << endl << endl;
    unique_ptr<int> up3(new int(10));
    /* up3 = up;//error,不能进行赋值操作 */

    cout << endl << endl;
    unique_ptr<int> up4(std::move(up));//通过移动语义转移up的所有权
    cout << "*up4 = " << *up4 << endl;
    cout << "up4.get() = " << up4.get() << endl;


    cout << endl << endl;
    unique_ptr<Point> up5(new Point(3, 4));//通过移动语义转移up的所有权
    vector<unique_ptr<Point>> numbers;
    numbers.push_back(unique_ptr<Point>(new Point(1, 2)));
    numbers.push_back(std::move(up5));
}
```

3、shared_ptr

虽然unique_ptr解决了auto_ptr的缺陷，但是使用上还是有限制，因为其只能独享所有权，于是就有共享所有权的需求，即是shared_ptr采用的思想与之前的写时复制技术类型，浅拷贝 + 引用计数，use_count函数记录指向一块内存的指针的数目。

```
void test()
{
    shared_ptr<int> sp(new int(10));
    cout << "*sp = " << *sp << endl;
    cout << "sp.get() = " << sp.get() << endl;
    cout << "sp.use_count() = " << sp.use_count() << endl;

    cout << endl << endl;
    {
        shared_ptr<int> sp2(sp);//共享所有权,使用浅拷贝
        cout << "*sp = " << *sp << endl;
        cout << "sp.get() = " << sp.get() << endl;
        cout << "sp.use_count() = " << sp.use_count() << endl;
        cout << "*sp2 = " << *sp2 << endl;
        cout << "sp2.get() = " << sp2.get() << endl;
        cout << "sp2.use_count() = " << sp2.use_count() << endl;
    }

    cout << endl << endl;
    shared_ptr<int> sp3(new int(10));
    sp3 = sp;//赋值
    cout << "*sp = " << *sp << endl;
    cout << "sp.get() = " << sp.get() << endl;
    cout << "sp.use_count() = " << sp.use_count() << endl;
    cout << "*sp3 = " << *sp3 << endl;
    cout << "sp3.get() = " << sp3.get() << endl;
    cout << "sp3.use_count() = " << sp3.use_count() << endl;

    cout << endl << endl;
    shared_ptr<Point> sp4(new Point(3, 4));//通过移动语义转移sp的所有权
    vector<shared_ptr<Point>> numbers;
    numbers.push_back(shared_ptr<Point>(new Point(1, 2)));
    numbers.push_back(sp4);
    numbers[0]->print();
    numbers[1]->print();
}
```

3.1、循环引用

该智能指针在使用的时候，会使得引用计数增加，从而会出现**循环引用**的问题？？？两个shared_ptr智能指针互指，导致引用计数增加，不能靠对象的销毁使得引用计数变为0，从而导致内存泄漏。。。

```
class Child;

class Parent
{
public:
    Parent()
    {
        cout << "Parent()" << endl;
    }
```

```
    ~Parent()
    {
        cout << "~Parent()" << endl;
    }

    shared_ptr<Child> pParent;
};

class Child
{
public:
    Child()
    {
        cout << "Child()" << endl;
    }

    ~Child()
    {
        cout << "~Child()" << endl;
    }

    shared_ptr<Parent> pChild;
};


void test()
{
    //循环引用可能导致内存泄漏
    shared_ptr<Parent> parentPtr(new Parent());
    shared_ptr<Child> childPtr(new Child());
    cout << "parentPtr.use_count() = " << parentPtr.use_count() << endl;
    cout << "childPtr.use_count() = " << childPtr.use_count() << endl;

    cout << endl << endl;
    parentPtr->pParent = childPtr;//sp = sp
    childPtr->pChild = parentPtr;
    cout << "parentPtr.use_count() = " << parentPtr.use_count() << endl;
    cout << "childPtr.use_count() = " << childPtr.use_count() << endl;
}
```

解决循环引用的办法是使得其中一个改为weak_ptr，不会增加引用计数，这样可以使用对象的销毁而打破引用计数减为0的问题。。

修改： shared_ptr pChild;改为 weak_ptr pChild;即可解决循环引用的问题。。。

```
parentPtr->pParent = childPtr;//sp = sp
childPtr->pChild = parentPtr;//wp = sp,weak_ptr不会导致引用计数加1
```

4、weak_ptr

与shared_ptr相比，称为弱引用的智能指针，shared_ptr是强引用的智能指针。weak_ptr不会导致引用计数增加，但是它不能直接获取资源，必须通过lock函数从wp提升为sp，从而判断共享的资源是否已经销毁。

```
void test()
{
```

```
    /* weak_ptr<Point> wp(new Point(1, 2));//error,不能直接获取资源 */
    weak_ptr<Point> wp;
    {
        shared_ptr<Point> sp(new Point(1, 2));
        wp = sp;
        cout << "wp.use_count = " << wp.use_count() << endl;
        cout << "sp.use_count = " << sp.use_count() << endl;

        //weak_ptr不能直接获取资源
        cout << "wp.expired = " << wp.expired() << endl;//此方法等同于use_count()=0
        shared_ptr<Point> sp2 = wp.lock();//判断共享的资源是否已经销毁的方式就是从wp提升为sp
        if(sp2)
        {
            cout << "提升成功" << endl;
        }
        else
        {
            cout << "提升失败" << endl;
        }
    }

    cout << endl << "当块语句执行结束之后 : "  << endl;
    cout << "wp.expired = " << wp.expired() << endl;
    //weak_ptr不能直接获取资源
    shared_ptr<Point> sp2 = wp.lock();
    if(sp2)
    {
        cout << "提升成功" << endl;
    }
    else
    {
        cout << "提升失败" << endl;
    }
}
```

以上就是四种智能指针的使用，使用起来还比较简单。接下来看看unique_ptr(存在于模板的第二个参数)与shared_ptr(存在于构造函数之中)相关的删除器。

四、为智能指针定制删除器

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，这时我们的智能指针就无法来处理，因此我们需要为智能指针定制删除器，提供一个可以自由选择析构的接口，这样，我们的智能指针就可以处理不同形式开辟的空间以及可以管理文件指针。

自定义智能指针的方式有两种，函数指针与仿函数(函数对象)

函数指针的形式：

```
template<class T>
void Free(T* p)
{
    if (p)
        free(p);
}
```

```cpp
template<class T>
void Del(T* p)
{
    if (p)
        delete p;
}
void FClose(FILE* pf)
{
    if (pf)
        fclose(pf);
}

//定义函数指针的类型
typedef void(*DP)(void*);

template<class T>
class SharedPtr
{
public:
    SharedPtr(T* ptr = NULL ,DP dp=Del)
    :_ptr(ptr)
    , _pCount(NULL)
    , _dp(dp)
    {
        if (_ptr != NULL)
        {
            _pCount = new int(1);
        }
    }
private:
    void Release()
    {
        if (_ptr&&0==--GetRef())
        {
            //delete _ptr;
            _dp(_ptr);
            delete _pCount;
        }
    }
    int& GetRef()
    {
        return *_pCount;
    }
private:
    T* _ptr;
    int* _pCount;
    DP _dp;
};
```

关于删除器的使用

```cpp
struct FILECloser
{
    void operator()(FILE *fp)
    {
        if(fp)
        {
```

```
            fclose(fp);
            cout << "fclose(fp)" << endl;
        }
    }
};
void test()
{
    unique_ptr<FILE, FILECloser> up(fopen("wuhan.txt", "a+"));
    string msg = "hello,world\n";
    fwrite(msg.c_str(), 1, msg.size(), up.get());
    /* fclose(up.get()); */
}
void test1()
{
    shared_ptr<FILE> up(fopen("wuhan1.txt", "a+"), FILECloser());
    string msg = "hello,world\n";
    fwrite(msg.c_str(), 1, msg.size(), up.get());
    /* fclose(up.get()); */
}
```

五、智能指针的误用

1、同一个裸指针被不同的智能指针托管，导致被析构两次。

1.1、直接使用

```
//error，将同一个裸指针用不同的智能指针进行托管
Point *pt = new Point(10, 20);
unique_ptr<Point> up1(pt);
unique_ptr<Point> up2(pt);
```

1.2、间接使用

```
//error，将同一个裸指针用不同的智能指针进行托管
unique_ptr<Point> up1(new Point(1, 2));
unique_ptr<Point> up2(new Point(30, 50));
up1.reset(up2.get());
```

2、还是裸指针被智能指针托管形式，但是比较隐蔽。。。

```
class Point
: public std::enable_shared_from_this<Point>
{
public:
    Point(int ix = 0, int iy = 0)
    : _ix(ix)
    , _iy(iy)
    {
        cout << "Point(int = 0, int = 0)" << endl;
    }

    void print() const
    {
        cout << "(" <<_ix
             << ","  << _iy
             << ")" << endl;
```

```
    }

    /* Point *addPoint(Point *pt) */
    shared_ptr<Point> addPoint(Point *pt)
    {
        _ix += pt->_ix;
        _iy += pt->_iy;

        //this指针是一个裸指针
        /* return shared_ptr<Point>(this); */
        return shared_from_this();
    }
    ~Point()
    {
        cout << "~Point()" << endl;
    }
private:
    int _ix;
    int _iy;
};

void test3()
{
    shared_ptr<Point> sp1(new Point(1, 2));
    cout << "sp1 = ";
    sp1->print();

    cout << endl;
    shared_ptr<Point> sp2(new Point(3, 4));
    cout << "sp2 = ";
    sp2->print();

    cout << endl;
    shared_ptr<Point> sp3(sp1->addPoint(sp2.get()));
    cout << "sp3 = ";
    sp3->print();
}
```