实验三 智能指针

张昱-编译原理教学团队

中国科学技术大学 计算机科学与技术学院

智能指针

- 定义：C++的智能指针其实就是对普通指针的封装（即封装成一个类），通过**重载** * 和 -> 两个运算符，使得智能指针表现的就像普通指针一样

- 作用：使用普通指针，容易造成堆内存泄漏（忘记释放）、二次释放、程序执行异常时的内存泄漏等问题，使用智能指针能更好地管理堆内存

智能指针

- 实现原理：智能指针的通用实现技术是使用**引用计数**(reference count)来管理共享的对象，当引用计数值减为0时，即可释放对象资源。

- 在头文件<memory>中定义有常用的四种智能指针：std::auto_ptr（已弃用）、std::unique_ptr、std::shared_ptr、std::weak_ptr

目录

- auto_ptr
- unique_ptr
- shared_ptr
- weak_ptr

auto_ptr

❑ auto_ptr

- **基本功能**
  - ❖ 预处理该模板类对象提供对指针的受限垃圾收集能力，允许在auto_ptr对象被销毁时，指针指向的元素被**自动销毁**
- **auto_ptr对象具有对分配给它们的指针拥有所有权的特性：**
  - ❖ 对一个元素拥有所有权的auto_ptr对象负责销毁它指向的元素，并在销毁它时销毁分配给它的内存。析构函数通过调用算子delete来自动执行此操作。
- **不会有两个auto_ptr对象拥有相同的元素。**
  - ❖ 当在两个auto_ptr对象之间发生复制操作时，**所有权被转移**，也即失去所有权的auto_ptr对象不再指向该元素，即被设置为空指针。

auto_ptr

- **风险**
  - std::auto_ptr 通过**复制**构造函数和赋值运算符实现移动语义
    - ✓ 如果将std::auto_ptr按值传递给函数将导致您的资源转移至函数参数（并在函数末尾(超出函数参数的作用域)被销毁）。这之后，当需要从调用方访问std::auto_ptr 参数时（没有意识到它已被转移和删除），这时已是空引用指针！
  - std::auto_ptr始终使用**非数组的delete**来删除其指向的元素。
    - ✓ 这意味着auto_ptr无法正确使用动态分配的数组，因为它使用了错误的释放类型。更糟的是，它不会阻止你将其传递给动态数组，否则动态数组将导致管理不善，从而导致内存泄漏。
  - auto_ptr在标准库的许多其他类（包括大多数容器和算法）上不能很好地发挥作用。这是因为那些标准库类往往假定它们在复制元素项时实际上是在复制而不是移动。

代码示例

```
class Test
{
public:
    Test(string s)
    {
        str = s;
       cout<<"Test creat\n";
    }
    ~Test()
    {
        cout<<"Test delete:"<<str<<endl;
    }
    string& getStr()
    {
        return str;
    }
    void setStr(string s)
    {
        str = s;
    }
    void print()
    {
        cout<<str<<endl;
    }
private:
    string str;
};
```

智能指针可以像类的原始指针一样访问类的public成员，成员函数get()返回一个原始的指针，成员函数reset()重新绑定指向的对象，而原来的对象则会被释放。访问auto_ptr的成员函数时用的是“.”，访问指向对象的成员时用的是“->”。

```
int main()
{
    auto_ptr<Test> ptest(new Test("123"));     //调用构造函数输出Test creat
    ptest->setStr("hello ");                  //修改成员变量的值
    ptest->print();                             //输出hello
    ptest.get()->print();                      //输出hello
    ptest->getStr() += "world !";
    (*ptest).print();                          //输出hello world
    ptest.reset(new Test("123"));//成员函数reset()重新绑定指向的对象，而原来的对象则会被释放，
                                 所以这里会调用一次构造函数，还有调用一次析构函数释放掉之前的对象
    ptest->print();                             //输出123
    func(ptest);
    ptest->print();
    assert(ptest.get()  && "it is NULL now");
    return 0;                                  //此时还剩下一个对象，调用一次析构函数释放该对象
}
```

目录

- auto_ptr
- unique_ptr
- shared_ptr
- weak_ptr

unqiue_ptr

❑ unqiue_ptr

■ 基本概念

- ❖ std::unique_ptr是用于取代std::auto_ptr的产物,在C++11中引入
  - ✓ 在C++11之前，C++语言没有区分复制语义和移动语义的机制，重载复制语义来实现移动语义会导致怪异的情况和无意的错误。例如，对于r1 = r2，无法判断 r2 是否会被改变
- ❖ std::unique_ptr正确地实现了**移动语义**
  - ✓ 如果希望转移std::unique_ptr对象r2管理的内容，则需要通过r1 = std::move(r2)，将r2转换成右值，这将触发移动赋值而不是复制赋值
- ❖ std::unique_ptr具有**重载的operator *和operator ->**，可用于返回被管理的资源
  - ✓ Operator *返回对托管资源的引用，operator ->返回指针。在使用这些运算符之前，应检查std::unique_ptr对象r是否确实有管理的资源，这可以简单地通过判断 r 是否为真来做到，因为std::unique_ptr在条件判断中能隐式强制转换为布尔值

- ❖ 注：在C++11中正式定义移动(move)的概念，并增加移动语义(move semantics)，从而将复制和移动区别开来。也因此，std::auto_ptr已经被一系列移动感知的(move-aware)智能指针取代：std::unique_ptr、std::weak_ptr、std::shared_ptr

代码示例

```
class Test
{
public:
    Test(string s)
    {
        str = s;
       cout<<"Test creat\n";
    }
    ~Test()
    {
        cout<<"Test delete:"<<str<<endl;
    }
    string& getStr()
    {
        return str;
    }
    void setStr(string s)
    {
        str = s;
    }
    void print()
    {
        cout<<str<<endl;
    }
private:
    string str;
};
unique_ptr<Test> fun()
{
    return unique_ptr<Test>(new Test("789"));//调用了构造函数，输出Test creat
}
void func(unique_ptr<Test> & ptest){
    return;
}
```

```
int main()
{
    unique_ptr<Test> ptest(new Test("123"));    //调用构造函数，输出Test creat
    func(ptest);
    unique_ptr<Test> ptest2(new Test("456"));    //调用构造函数，输出Test creat
    ptest->print();                                 //输出123
    ptest2 = std::move(ptest);    //不能直接ptest2 = ptest，调用了move后ptest2原本的对象会被释放，
                                    ptest2对象指向原本ptest对象的内存，输出Test delete 456
    if(ptest == NULL)cout<<"ptest = NULL\n";    //因为两个unique_ptr不能指向同一内存地址，
                                    所以经过前面move后ptest会被赋值NULL，输出ptest=NULL
    Test* p = ptest2.release();    //release成员函数把ptest2指针赋为空，但是并没有释放指针指向的内存，
                                        所以此时p指针指向原本ptest2指向的内存
    p->print();                    //输出123
    ptest.reset(p);                //重新绑定对象，原来的对象会被释放掉，但是ptest对象本来就释放过了，
                                        所以这里就不会再调用析构函数了
    ptest->print();                //输出123
    ptest2 = fun();            //这里可以用=，因为使用了移动构造函数，函数返回一个unique_ptr会自动调用移动构造函数
    ptest2->print();           //输出789
    return 0;                   //此时程序中还有两个对象，调用两次析构函数释放对象
}
```

目录

- auto_ptr
- unique_ptr
- shared_ptr
- weak_ptr

shared_ptr

❑ 基本概念

■ std::shared_ptr是一个基于引用计数管理所共享的对象的智能指针

- ❖ 多个std::shared_ptr可以管理同一对象，即共享对象的所有权，并且采用引用计数管理所共享的对象
  - ✓ 每个shared_ptr对象关联有一个共享的**引用计数**。当拷贝一个 shared_ptr ，将其引用计数值加 1
  - ✓ shared_ptr提供**unique()**和**use_count()**两个函数来检查其共享的引用计数值，前者测试该shared_ptr是否是唯一拥有者（即引用计数值为1），后者返回引用计数值
  - ✓ 当shared_ptr共享的引用计数降低到 0 的时候，所管理的对象自动被析构（调用其析构函数释放对象）

shared_ptr

- **std::shared_ptr的创建与转化**
  - std::make_shared 可以用**std::make_shared**创建std::shared_ptr。 std::make_shared在C++11中可用
  - 可以从唯一指针创建共享指针一个std::unique_ptr。可以**通过接受该std::unique_ptr右值的std::shared_ptr构造函数转化为std::shared_ptr**，则该std::unique_ptr的内容将移动到std::shared_ptr
    - ✓ std::shared_ptr不能安全地转化为std::unique_ptr

shared_ptr

- **std::shared_ptr的风险**
  - 如果管理资源的任何std :: shared_ptr未被正确销毁，则资源将无法正确释放

- **std::shared_ptr与数组**
  - 在C++14及更早的版本中，std::shared_ptr并没有管理数组的支持；在C++17中std::shared_ptr提供对数组的支持，但是std::make_shared仍缺乏对数组的支持，不能用于创建共享数组，这个问题将在C++20中解决

代码示例

```
class Test
{
public:
    Test(string s)
    {
        str = s;
       cout<<"Test creat\n";
    }
    ~Test()
    {
        cout<<"Test delete:"<<str<<endl;
    }
    string& getStr()
    {
        return str;
    }
    void setStr(string s)
    {
        str = s;
    }
    void print()
    {
        cout<<str<<endl;
    }
private:
    string str;
};
unique_ptr<Test> fun()
{
    return unique_ptr<Test>(new Test("789"));
}
```

```
int main()
{
    shared_ptr<Test> ptest(new Test("123"));     //调用构造函数输出Test create
    shared_ptr<Test> ptest2(new Test("456"));     //调用构造函数输出 Test creat
    cout<<ptest2->getStr()<<endl;                 //输出456
    cout<<ptest2.use_count()<<endl;                 //显示此时资源被几个指针共享，输出1
    ptest = ptest2;          //"456"引用次数加1，“123”销毁，输出Test delete：123
    ptest->print();          //输出456
    cout<<ptest2.use_count()<<endl;                 //该指针指向的资源被几个指针共享，输出2
    cout<<ptest.use_count()<<endl;                 //2
    ptest.reset();     //重新绑定对象，绑定一个空对象，当时此时指针指向的对象还有其他指针能指向就不会释放该对象的内存空间，
    ptest2.reset();     //此时“456”销毁，此时指针指向的内存空间上的指针为0，就释放了该内存，输出Test delete
    cout<<"done !\n";
    return 0;
}
```

目录

- auto_ptr
- unique_ptr
- shared_ptr
- weak_ptr

weak_ptr

❑ weak_ptr

■ 基本概念

- ❖ std::weak_ptr是用来解决std::shared_ptr循环引用时的**死锁问题**
  - ✓ 如果说两个std::shared_ptr相互引用形成环,那么这两个指针的引用计数永远不可能下降为0,资源永远不会释放
- ❖ std::weak_ptr是对共享对象的一种**弱引用**，不会增加对象的引用计数值
- ❖ std::weak_ptr和std::shared_ptr之间可以**相互转化**，std::shared_ptr可以**直接赋值**给std::weak_ptr，它可以通过调用**lock函数**来获得std::shared_ptr

weak_ptr

- **方法访问**
  - 不能通过weak_ptr直接访问对象的方法，比如B对象中有一个方法print(),不能使用pa->pb_->print() 访问
  - pb_是一个weak_ptr，应该先把它转化为shared_ptr,如：

```
1. shared_ptr<B> p= pa->pb_.lock();
2. p->print();
```

代码示例

```
void fun()
{
    shared_ptr<B<A>> pb(new B<A>());
    shared_ptr<A> pa(new A());
    pb->pa_ = pa;
    pa->pb_ = pb;
    cout<<"fun : "<<pb.use_count()<<endl;
    cout<<"fun : "<<pa.use_count()<<endl;
    // 这个函数执行完会出现相互引用导致的内存泄漏
}
void fun1()
{
    shared_ptr<B<A1>> pb(new B<A1>());
    shared_ptr<A1> pa(new A1());
    pb->pa_ = pa;
    pa->pb_ = pb;
    cout<<"fun1 : "<<pb.use_count()<<endl;
    cout<<"fun1 : "<<pa.use_count()<<endl;
}
int main()
{
    fun();
    fun1();
    return 0;
}
```

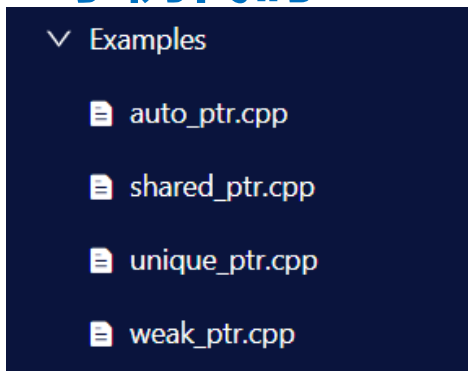
```
template <typename T>
class B;
class A
{
public:
    shared_ptr<B<A>> pb_;
    ~A()
    {
        cout<<"A delete\n";
    }
};
class A1
{
public:
    weak_ptr<B<A1>> pb_;
    ~A1()
    {
        cout<<"A1 delete\n";
    }
};
template <typename T>
class B
{
public:
    shared_ptr<T> pa_;
    ~B()
    {
        cout<<"B delete\n";
    }
};
```

- **fun**
  - 可以看到fun函数中 pa 、pb 之间互相引用，两个资源的引用计数为2
  - 当要跳出函数时，智能指针pa、pb析构时两个资源引用计数会减一，得到计数值1，导致跳出函数时资源没有被释放（A、B的析构函数没有被调用）。
- **fun1**
  - 把类A里面的shared_ptr< B> pb_; 改为 weak_ptr< B> pb_
  - 资源B的引用开始时只有1，当pb析构时，B的计数变为0，B得到释放
  - B释放的同时也会使A的计数减一，同时pa析构时使A的计数减一，那么A的计数为0，A得到释放。

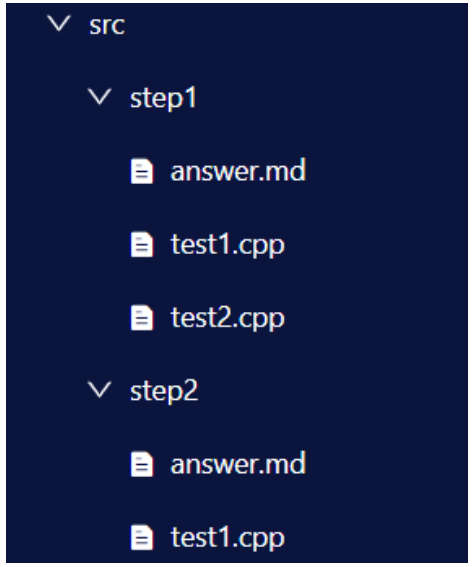
实践任务

实践任务宜配合如图代码仓库使用

示例代码



任务代码



实践任务

- **任务一**
  - 请参考std::auto_ptr存在的风险这一部分构造两个样例程序，分别展示因为auto_ptr使用不当而导致引用空指针、出现内存泄漏这两种情况。要求所编写的程序尽量简单，并且加上必要的注释。请将程序放在src/step1/文件夹下，并且分别命名为test1.cpp、test2.cpp
  - 你认为unique_ptr可以作为实参传递给函数吗，为什么？请将你的思考放在src/step1/文件夹下，并且命名为answer.md
    - ✓ Hint: 从值传递和引用传递两方面考虑

实践任务

- **任务二**
  - weak_ptr可以直接通过普通指针引用对象吗?
    - ✓ 补充：“通过普通指针引用对象”是指诸如weak_ptr<int> wptr(new int(0))，这种直接在普通指针上增加引用(reference)的做法。请大家注意和通过重载运算符*和->访问对象的区别
  - 在程序中，如果管理weak_ptr的shared_ptr释放了，那么还可以通过weak_ptr去访问对象吗?
    - ✓ 补充：“管理weak_ptr的shared_ptr”是指给weak_ptr赋值(转化成weak_ptr)的那个shared_ptr
  - 请大家在src/step2/test1.cpp文件中构造样例来找到上述两个问题的答案，请给你的样例加上必要的注释，并将你的答案写在src/step2/answer.md中

实践任务

- **任务三**
  - 请问当程序执行到L17和L19时，程序已经产生的输出为何?
    - ✓ A. L17处已经输出了”H”,L19处也已经输出了”H”。原因是std::unique_ptr<A> uptr2 = std::move(uptr)语句调用了实例化的A对象的析构函数。
    - ✓ B. L17前无输出,L19处已经输出了”H”。原因是uptr2.release()语句调用了实例化的A对象的析构函数。
    - ✓ C. 均无输出，原因是uptr2.release()执行以后，实例化的A对象不再被任何unique指针引用，出现了内存泄漏。
    - ✓ D. L17前无输出,L19处已经输出了”H”。原因是第18行处离开了uptr2的作用域，自动调用了实例化的A对象的析构函数。

```
1. #include <iostream>
2. #include <memory>
3.
4. class A
5. {
6. public:
7.     ~A(){std::cout << "H";}
8. };
9.
10. int main() {
11.     {
12.         std::unique_ptr<A> uptr(new A());
13.         std::unique_ptr<A> uptr2 = std::move(uptr);
14.         uptr2.release();
15.         // L17
16.     }
17.     // L19
18. }
```

实践任务

- **任务三**
  - 请问程序最终的输出是什么？
    - ✓ A. 33
    - ✓ B. 31
    - ✓ C. 21
    - ✓ D. 10

```
1. #include <iostream>
2. #include <memory>
3.
4. int main() {
5.     auto x = new int(0);
6.     std::shared_ptr<int> sptr1(x);
7.     {
8.         std::weak_ptr<int> sptr2 = sptr1;
9.         auto sptr3 = sptr1;
10.        std::cout << sptr1.use_count();
11.    }
12.    std::cout << sptr1.use_count();
13. }
```

实践任务

- **任务三**
  - x是int *类型，如图有三个代码片段，选项中正确的有:
    - ✓ A. Code1可以执行，最终x只被sptr引用
    - ✓ B. Code1可以执行，最终x被sptr和uptr同时引用
    - ✓ C. Code1编译无法通过
    - ✓ D. Code2可以正确执行，最终x只被uptr引用
    - ✓ E. Code2编译无法通过
    - ✓ F. Code3可以正确执行，最终x的被引用数是1
    - ✓ G. Code3编译无法通过

Code1:

```
1. std::unique_ptr<int> uptr(x);
2. std::shared_ptr<int> sptr(x);
```

Code2:

```
1. std::shared_ptr<int> sptr(x);
2. std::unique_ptr<int> uptr = sptr;
```

Code3:

```
1. std::unique_ptr<int> uptr(x);
2. std::weak_ptr<int> wptr(x);
```

下期再见！

Thanks!