# 智能指针

## 1.智能指针的作用

C++程序设计中使用堆内存是非常频繁的操作，堆内存的申请和释放都由程序员自己管理。程序员自己管理堆内存可以提高了程序的效率，但是整体来说堆内存的管理是麻烦的，C++11 中引入了智能指针的概念，方便管理堆内存。使用普通指针，容易造成堆内存泄露（忘记释放），二次释放，程序发生异常时内存泄露等问题等，使用智能指针能更好的管理堆内存。

理解智能指针需要从下面三个层次：

1. 从较浅的层面看，智能指针是利用了一种叫做 RAII（资源获取即初始化）的技术对普通的指针进行封装，这使得智能指针实质是一个对象，行为表现的却像一个指针。
2. 智能指针的作用是防止忘记调用 delete 释放内存和程序异常的进入 catch 块忘记释放内存。另外指针的释放时机也是非常有考究的，多次释放同一个指针会造成程序崩溃，这些都可以通过智能指针来解决。
3. 智能指针还有一个作用是把值语义转换成引用语义。C++和 Java 有一处最大的区别在于语义不同，在 Java 里面下列代码：

```
Animal a = new Animal();

Animal b = a;
```

你当然知道，这里其实只生成了一个对象，a 和 b 仅仅是把持对象的引用而已。但在 C++中不是这样，

```
Animal a;

Animal b = a;
```

这里却是就是生成了两个对象。

## 2.智能指针的使用

智能指针在 C++11 版本之后提供，包含在头文件<memory>中，shared_ptr、unique_ptr、weak_ptr

### 2.1 shared_ptr 的使用

shared_ptr 多个指针指向相同的对象。shared_ptr 使用引用计数，每一个 shared_ptr 的拷贝都指向相同的内存。每使用他一次，内部的引用计数加 1，每析构一次，内部的引用计数减 1，减为 0 时，自动删除所指向的堆内存。shared_ptr 内部的引用计数是线程安全的，但是对象的读取需要加锁。

- 初始化。智能指针是个模板类，可以指定类型，传入指针通过构造函数初始化。也可以使用 make_shared 函数初始化。不能将指针直接赋值给一个智能指针，一个是类，一个是指针。例如 std::shared_ptr<int> p4 = new int(1);的写法是错误的
- 拷贝和赋值。拷贝使得对象的引用计数增加 1，赋值使得原对象引用计数减 1，当计数为 0 时，自动释放内存。后来指向的对象引用计数加 1，指向后来的对象。
- get 函数获取原始指针
- 注意不要用一个原始指针初始化多个 shared_ptr，否则会造成二次释放同一内存
- 注意避免循环引用，shared_ptr 的一个最大的陷阱是循环引用，循环，循环引用会导致堆内存无法正确释放，导致内存泄漏。循环引用在 weak_ptr 中介绍。

```
#include <iostream>
#include <memory>
int main() {
    {
        int a = 10;
        std::shared_ptr<int> ptra = std::make_shared<int>(a);
        std::shared_ptr<int> ptra2(ptra); //copy
        std::cout << ptra.use_count() << std::endl;
        int b = 20;
        int *pb = &a;
        //std::shared_ptr<int> ptrb = pb;  //error
        std::shared_ptr<int> ptrb = std::make_shared<int>(b);
        ptra2 = ptrb; //assign
        pb = ptrb.get(); //获取原始指针
        std::cout << ptra.use_count() << std::endl;
        std::cout << ptrb.use_count() << std::endl;
    }
}
```

## 2.2 unique_ptr 的使用

unique_ptr“唯一”拥有其所指对象，同一时刻只能有一个 unique_ptr 指向给定对象（通过禁止拷贝语义、只有移动语义来实现）。相比与原始指针 unique_ptr 用于其 RAII 的特性，使得在出现异常的情况下，动态资源能得到释放。unique_ptr 指针本身的生命周期：从 unique_ptr 指针创建时开始，直到离开作用域。离开作用域时，若其指向对象，则将其所指对象销毁(默认使用 delete 操作符，用户可指定其他操作)。unique_ptr 指针与其所指对象的关系：在智能指针生命周期内，可以改变智能指针所指对象，如创建智能指针时通过构造函数指定、通过 reset 方法重新指定、通过 release 方法释放所有权、通过移动语义转移所有权。

```
#include <iostream>
#include <memory>

int main() {
    {
        std::unique_ptr<int> uptr(new int(10));  //绑定动态对象
        //std::unique_ptr<int> uptr2 = uptr;  //不能赋值
```

```
        //std::unique_ptr<int> uptr2(uptr);  //不能拷貝
        std::unique_ptr<int> uptr2 = std::move(uptr); //轉換所有權
        uptr2.release(); //释放所有权
    }
    //超過 uptr 的作用域，内存釋放
}
```

## 2.3 weak_ptr 的使用

weak_ptr 是为了配合 shared_ptr 而引入的一种智能指针，因为它不具有普通指针的行为，没有重载 operator*和->,它的最大作用在于协助 shared_ptr 工作，像旁观者那样观测资源的使用情况。weak_ptr 可以从一个 shared_ptr 或者另一个 weak_ptr 对象构造，获得资源的观测权。但 weak_ptr 没有共享资源，它的构造不会引起指针引用计数的增加。使用 weak_ptr 的成员函数 use_count()可以观测资源的引用计数，另一个成员函数 expired()的功能等价于 use_count()==0,但更快，表示被观测的资源(也就是 shared_ptr 的管理的资源)已经不复存在。weak_ptr 可以使用一个非常重要的成员函数 lock()从被观测的 shared_ptr 获得一个可用的 shared_ptr 对象， 从而操作资源。但当 expired()==true 的时候，lock()函数将返回一个存储空指针的 shared_ptr。

```
#include <iostream>
#include <memory>

int main() {
    {
        std::shared_ptr<int> sh_ptr = std::make_shared<int>(10);
        std::cout << sh_ptr.use_count() << std::endl;

        std::weak_ptr<int> wp(sh_ptr);
        std::cout << wp.use_count() << std::endl;
        if(!wp.expired()){
            std::shared_ptr<int> sh_ptr2 = wp.lock(); //get another shared_ptr
            *sh_ptr = 100;
            std::cout << wp.use_count() << std::endl;
        }
    }
    //delete memory
}
```

## 2.4 循环引用

考虑一个简单的对象建模——家长与子女：a Parent has a Child, a Child knowshis/her Parent。在 Java 里边很好写，不用担心内存泄漏，也不用担心空悬指针，只要正确初始化 myChild 和 myParent，那么 Java 程序员就不用担心出现访问错误。一个 handle 是否有效，只需要判断其是否 non null。

```
public class Parent
{
```

```
    private Child myChild;
}
public class Child
{
    private Parent myParent;
}
```

在 C++ 里边就要为资源管理费一番脑筋。如果使用原始指针作为成员，Child 和 Parent 由谁释放？那么如何保证指针的有效性？如何防止出现空悬指针？这些问题是 C++面向对象编程麻烦的问题，现在可以借助 smart pointer 把对象语义（pointer）转变为值（value）语义，shared_ptr 轻松解决生命周期的问题，不必担心空悬指针。但是这个模型存在循环引用的问题，注意其中一个指针应该为 weak_ptr。

## 原始指针的做法，容易出错

```
#include <iostream>
#include <memory>
class Child;
class Parent;
class Parent {
private:
    Child* myChild;
public:
    void setChild(Child* ch) {
        this->myChild = ch;
    }
    void doSomething() {
        if (this->myChild) {
        }
    }

    ~Parent() {
        delete myChild;
    }
};
class Child {
private:
    Parent* myParent;
public:
    void setPartent(Parent* p) {
        this->myParent = p;
    }
    void doSomething() {
        if (this->myParent) {
        }
    }
    ~Child() {
        delete myParent;
    }
```

```
};
int main() {
    {
        Parent* p = new Parent;
        Child* c =  new Child;
        p->setChild(c);
        c->setPartent(p);
        delete c;  //only delete one
    }
    return 0;
}
```

循环引用内存泄露的问题

```
#include <iostream>
#include <memory>
class Child;
class Parent;
class Parent {
private:
    std::shared_ptr<Child> ChildPtr;
public:
    void setChild(std::shared_ptr<Child> child) {
        this->ChildPtr = child;
    }
    void doSomething() {
        if (this->ChildPtr.use_count()) {
        }
    }
    ~Parent() {
    }
};
class Child {
private:
    std::shared_ptr<Parent> ParentPtr;
public:
    void setPartent(std::shared_ptr<Parent> parent) {
        this->ParentPtr = parent;
    }
    void doSomething() {
        if (this->ParentPtr.use_count()) {
        }
    }
    ~Child() {
    }
};
int main() {
```

```
    std::weak_ptr<Parent> wpp;
    std::weak_ptr<Child> wpc;
    {
        std::shared_ptr<Parent> p(new Parent);
        std::shared_ptr<Child> c(new Child);
        p->setChild(c);
        c->setPartent(p);
        wpp = p;
        wpc = c;
        std::cout << p.use_count() << std::endl; // 2
        std::cout << c.use_count() << std::endl; // 2
    }
    std::cout << wpp.use_count() << std::endl;  // 1
    std::cout << wpc.use_count() << std::endl;  // 1
    return 0;
}
```

正确的做法

```
#include <iostream>
#include <memory>
class Child;
class Parent;
class Parent {
private:
    //std::shared_ptr<Child> ChildPtr;
    std::weak_ptr<Child> ChildPtr;
public:
    void setChild(std::shared_ptr<Child> child) {
        this->ChildPtr = child;
    }
    void doSomething() {
        //new shared_ptr
        if (this->ChildPtr.lock()) {
        }
    }
    ~Parent() {
    }
};
class Child {
private:
    std::shared_ptr<Parent> ParentPtr;
public:
    void setPartent(std::shared_ptr<Parent> parent) {
        this->ParentPtr = parent;
    }
    void doSomething() {
```

```
        if (this->ParentPtr.use_count()) {
        }
    }
    ~Child() {
    }
};
int main() {
    std::weak_ptr<Parent> wpp;
    std::weak_ptr<Child> wpc;
    {
        std::shared_ptr<Parent> p(new Parent);
        std::shared_ptr<Child> c(new Child);
        p->setChild(c);
        c->setPartent(p);
        wpp = p;
        wpc = c;
        std::cout << p.use_count() << std::endl; // 2
        std::cout << c.use_count() << std::endl; // 1
    }
    std::cout << wpp.use_count() << std::endl;  // 0
    std::cout << wpc.use_count() << std::endl;  // 0
    return 0;
}
```

# 3. 智能指针的设计和实现

智能指针类将一个计数器与类指向的对象相关联，引用计数跟踪该类有多少个对象共享同一指针。每次创建类的新对象时，初始化指针并将引用计数置为 1；当对象作为另一对象的副本而创建时，拷贝构造函数拷贝指针并增加与之相应的引用计数；对一个对象进行赋值时，赋值操作符减少左操作数所指对象的引用计数（如果引用计数为减至 0，则删除对象），并增加右操作数所指对象的引用计数；调用析构函数时，构造函数减少引用计数（如果引用计数减至 0，则删除基础对象）。智能指针就是模拟指针动作的类。所有的智能指针都会重载 -> 和 * 操作符。智能指针还有许多其他功能，比较有用的是自动销毁。这主要是利用栈对象的有限作用域以及临时对象（有限作用域实现）析构函数释放内存。

```
#include <iostream>
#include <memory>
template<typename T>
class SmartPointer {
 private:
     T* _ptr;
     size_t* _count;
 public:
    SmartPointer(T* ptr = nullptr) :
            _ptr(ptr) {
        if (_ptr) {
            _count = new size_t(1);
```

```
        } else {
            _count = new size_t(0);
        }
    }
    SmartPointer(const SmartPointer& ptr) {
        if (this != &ptr) {
            this->_ptr = ptr._ptr;
            this->_count = ptr._count;
            (*this->_count)++;
        }
    }
    SmartPointer& operator=(const SmartPointer& ptr) {
        if (this->_ptr == ptr._ptr) {
            return *this;
        }
        if (this->_ptr) {
            (*this->_count)--;
            if (this->_count == 0) {
               delete this->_ptr;
                delete this->_count;
            }
        }
       this->_ptr = ptr._ptr;
       this->_count = ptr._count;
        (*this->_count)++;
        return *this;
    }
    T& operator*() {
        assert(this->_ptr == nullptr);
        return *(this->_ptr);
    }
    T* operator->() {
        assert(this->_ptr == nullptr);
        return this->_ptr;
    }
    ~SmartPointer() {
       (*this->_count)--;
        if (*this->_count == 0) {
            delete this->_ptr;
            delete this->_count;
        }
   }
    size_t use_count(){
        return *this->_count;
    }
};
int main() {
```

```
    {
        SmartPointer<int> sp(new int(10));
        SmartPointer<int> sp2(sp);
        SmartPointer<int> sp3(new int(20));
        sp2 = sp3;
        std::cout << sp.use_count() << std::endl;
       std::cout << sp3.use_count() << std::endl;
    }
}
```