# Lesson08--智能指针

## 【本节目标】

- 1.为什么需要智能指针?
- 2. 内存泄漏
- 3.智能指针的使用及原理
- 4.C++11和boost中智能指针的关系
- 5.RAII扩展学习

### 1. 为什么需要智能指针?

> 下面我们先分析一下下面这段程序有没有什么**内存方面**的问题？提示一下：注意分析MergeSort函数中的问题。

```
int div()
{
    int a, b;
    cin >> a >> b;
    if (b == 0)
        throw invalid_argument("除0错误");

    return a / b;
}

void Func()
{
 // 1、如果p1这里new 抛异常会如何？
 // 2、如果p2这里new 抛异常会如何？
 // 3、如果div调用这里又会抛异常会如何？
    int* p1 = new int;
 int* p2 = new int;

    cout << div() << endl;

    delete p1;
 delete p2;
}

int main()
{
    try
    {
        Func();
    }
    catch (exception& e)
    {
        cout << e.what() << endl;
```

```
    }

    return 0;
}
```

**问题分析：上面的问题分析出来我们发现有什么问题?**

## 2. 内存泄漏

### 2.1 什么是内存泄漏，内存泄漏的危害

什么是内存泄漏：内存泄漏指因为疏忽或错误造成程序未能释放已经不再使用的内存的情况。内存泄漏并不是指内存在物理上的消失，而是应用程序分配某段内存后，因为设计错误，失去了对该段内存的控制，因而造成了内存的浪费。

内存泄漏的危害：长期运行的程序出现内存泄漏，影响很大，如操作系统、后台服务等等，出现内存泄漏会导致响应越来越慢，最终卡死。

```
void MemoryLeaks()
{
   // 1.内存申请了忘记释放
  int* p1 = (int*)malloc(sizeof(int));
  int* p2 = new int;

  // 2.异常安全问题
  int* p3 = new int[10];

  Func(); // 这里Func函数抛异常导致 delete[] p3未执行，p3没被释放.

  delete[] p3;
}
```

### 2.2 内存泄漏分类（了解）

C/C++程序中一般我们关心两种方面的内存泄漏：

- **堆内存泄漏(Heap leak)**

  堆内存指的是程序执行中依据须要分配通过malloc / calloc / realloc / new等从堆中分配的一块内存，用完后必须通过调用相应的 free或者delete 删掉。假设程序的设计错误导致这部分内存没有被释放，那么以后这部分空间将无法再被使用，就会产生Heap Leak。

- **系统资源泄漏**

  指程序使用系统分配的资源，比方套接字、文件描述符、管道等没有使用对应的函数释放掉，导致系统资源的浪费，严重可导致系统效能减少，系统执行不稳定。

### 2.3 如何检测内存泄漏（了解）

- 在linux下内存泄漏检测：linux下几款内存泄漏检测工具
- 在windows下使用第三方工具：VLD工具说明
- 其他工具：内存泄漏工具比较

### 2.4如何避免内存泄漏

1. 工程前期良好的设计规范，养成良好的编码规范，申请的内存空间记着匹配的去释放。ps：这个理想状态。但是如果碰上异常时，就算注意释放了，还是可能会出问题。需要下一条智能指针来管理才有保证。
2. 采用RAII思想或者智能指针来管理资源。

3. 有些公司内部规范使用内部实现的私有内存管理库。这套库自带内存泄漏检测的功能选项。
4. 出问题了使用内存泄漏工具检测。ps：不过很多工具都不够靠谱，或者收费昂贵。

总结一下:

内存泄漏非常常见，解决方案分为两种：1、事前预防型。如智能指针等。2、事后查错型。如泄漏检测工具。

## 3.智能指针的使用及原理

### 3.1 RAII

RAII（Resource Acquisition Is Initialization）是一种**利用对象生命周期来控制程序资源**（如内存、文件句柄、网络连接、互斥量等等）的简单技术。

**在对象构造时获取资源**，接着控制对资源的访问使之在对象的生命周期内始终保持有效，**最后在对象析构的时候释放资源**。借此，我们实际上把管理一份资源的责任托管给了一个对象。这种做法有两大好处：

- 不需要显式地释放资源。
- 采用这种方式，对象所需的资源在其生命期内始终保持有效。

```
// 使用RAII思想设计的SmartPtr类
template<class T>
class SmartPtr {
public:
    SmartPtr(T* ptr = nullptr)
        : _ptr(ptr)
    {}

    ~SmartPtr()
    {
        if(_ptr)
            delete _ptr;
    }

private:
    T* _ptr;
};

int div()
{
    int a, b;
    cin >> a >> b;
    if (b == 0)
        throw invalid_argument("除0错误");

    return a / b;
}

void Func()
{
    ShardPtr<int> sp1(new int);
    ShardPtr<int> sp2(new int);

    cout << div() << endl;
}
```

```
int main()
{
    try {
        Func();
    }
    catch(const exception& e)
    {
        cout<<e.what()<<endl;
    }

    return 0;
}
```

### 3.2 智能指针的原理

上述的SmartPtr还不能将其称为智能指针，因为它还不具有指针的行为。指针可以解引用，也可以通过->去访问所指空间中的内容，因此：**AutoPtr模板类中还得需要将* 、->重载下，才可让其像指针一样去使用。**

```
template<class T>
class SmartPtr {
public:
 SmartPtr(T* ptr = nullptr)
     : _ptr(ptr)
 {}

 ~SmartPtr()
 {
     if(_ptr)
         delete _ptr;
 }

 T& operator*() {return *_ptr;}
 T* operator->() {return _ptr;}
private:
 T* _ptr;
};

struct Date
{
    int _year;
    int _month;
    int _day;
};

int main()
{
 SmartPtr<int> sp1(new int);
 *sp1 = 10
 cout<<*sp1<<endl;

 SmartPtr<int> sparray(new Date);
 // 需要注意的是这里应该是sparray.operator->()->_year = 2018;
 // 本来应该是sparray->->_year这里语法上为了可读性，省略了一个->
 sparray->_year = 2018;
 sparray->_month = 1;
 sparray->_day = 1;
```

```
}
```

总结一下智能指针的原理：

1. RAII特性
2. 重载operator*和opertaor->，具有像指针一样的行为。

### 3.3 std::auto_ptr

std::auto_ptr文档

**C++98版本的库中就提供了auto_ptr的智能指针。下面演示的auto_ptr的使用及问题。**

**auto_ptr的实现原理：管理权转移的思想，下面简化模拟实现了一份bit::auto_ptr来了解它的原理**

```
// C++98  管理权转移  auto_ptr
namespace bit
{
    template<class T>
    class auto_ptr
    {
    public:
        auto_ptr(T* ptr)
            :_ptr(ptr)
        {}

        auto_ptr(auto_ptr<T>& sp)
            :_ptr(sp._ptr)
        {
            // 管理权转移
            sp._ptr = nullptr;
        }

        auto_ptr<T>& operator=(auto_ptr<T>& ap)
        {
            // 检测是否为自己给自己赋值
            if (this != &ap)
            {
                // 释放当前对象中资源
                if (_ptr)
                    delete _ptr;

                // 转移ap中资源到当前对象中
                _ptr = ap._ptr;
                ap._ptr = NULL;
            }

            return *this;
        }

        ~auto_ptr()
        {
            if (_ptr)
            {
                cout << "delete:" << _ptr << endl;
                delete _ptr;
```

```
            }
        }

        // 像指针一样使用
        T& operator*()
        {
            return *_ptr;
        }

        T* operator->()
        {
            return _ptr;
        }
    private:
        T* _ptr;
    };
}

// 结论：auto_ptr是一个失败设计，很多公司明确要求不能使用auto_ptr
//int main()
//{
//  std::auto_ptr<int> sp1(new int);
//  std::auto_ptr<int> sp2(sp1); // 管理权转移
//
//  // sp1悬空
//  *sp2 = 10;
//  cout << *sp2 << endl;
//  cout << *sp1 << endl;
//  return 0;
//}
```

### 3.4 std::unique_ptr

**C++11中开始提供更靠谱的unique_ptr**

unique_ptr文档

**unique_ptr的实现原理：简单粗暴的防拷贝，下面简化模拟实现了一份UniquePtr来了解它的原理**

```
// C++11库才更新智能指针实现
// C++11出来之前，boost搞除了更好用的scoped_ptr/shared_ptr/weak_ptr
// C++11将boost库中智能指针精华部分吸收了过来
// C++11->unique_ptr/shared_ptr/weak_ptr

// unique_ptr/scoped_ptr
// 原理：简单粗暴 -- 防拷贝
namespace bit
{
    template<class T>
    class unique_ptr
    {
    public:
        unique_ptr(T* ptr)
            :_ptr(ptr)
        {}

        ~unique_ptr()
```

```
        {
            if (_ptr)
            {
                cout << "delete:" << _ptr << endl;
                delete _ptr;
            }
        }

        // 像指针一样使用
        T& operator*()
        {
            return *_ptr;
        }

        T* operator->()
        {
            return _ptr;
        }

        unique_ptr(const unique_ptr<T>& sp) = delete;
        unique_ptr<T>& operator=(const unique_ptr<T>& sp) = delete;

    private:
        T* _ptr;
    };
}

//int main()
//{
//  /*bit::unique_ptr<int> sp1(new int);
//  bit::unique_ptr<int> sp2(sp1);*/
//
//  std::unique_ptr<int> sp1(new int);
//  //std::unique_ptr<int> sp2(sp1);
//
//  return 0;
//}
```

### 3.5 std::shared_ptr

**C++11中开始提供更靠谱的并且支持拷贝的shared_ptr**

[std::shared_ptr文档]

**shared_ptr的原理：是通过引用计数的方式来实现多个shared_ptr对象之间共享资源。**例如：比特老师晚上在下班之前都会通知，让最后走的学生记得把门锁下。

1. shared_ptr在其内部，**给每个资源都维护了着一份计数，用来记录该份资源被几个对象共享。**
2. 在**对象被销毁时(也就是析构函数调用)**，就说明自己不使用该资源了，对象的引用计数减一。
3. **如果引用计数是0**，就说明自己是最后一个使用该资源的对象，**必须释放该资源**；
4. **如果不是0**，就说明除了自己还有其他对象在使用该份资源，**不能释放该资源**，否则其他对象就成野指针了。

```
// 引用计数支持多个拷贝管理同一个资源，最后一个析构对象释放资源
```

```cpp
namespace bit
{
    template<class T>
    class shared_ptr
    {
    public:
        shared_ptr(T* ptr = nullptr)
            :_ptr(ptr)
            , _pRefCount(new int(1))
            , _pmtx(new mutex)
        {}

        shared_ptr(const shared_ptr<T>& sp)
            :_ptr(sp._ptr)
            , _pRefCount(sp._pRefCount)
            , _pmtx(sp._pmtx)
        {
            AddRef();
        }

        void Release()
        {
            _pmtx->lock();

            bool flag = false;
            if (--(*_pRefCount) == 0 && _ptr)
            {
                cout << "delete:" << _ptr << endl;
                delete _ptr;
                delete _pRefCount;

                flag = true;
            }

            _pmtx->unlock();

            if (flag == true)
            {
                delete _pmtx;
            }
        }

        void AddRef()
        {
            _pmtx->lock();

            ++(*_pRefCount);

            _pmtx->unlock();
        }

        shared_ptr<T>& operator=(const shared_ptr<T>& sp)
        {
            //if (this != &sp)
            if (_ptr != sp._ptr)
            {
                Release();
```

```
                _ptr = sp._ptr;
                _pRefCount = sp._pRefCount;
                _pmtx = sp._pmtx;
                AddRef();
            }

            return *this;
        }

        int use_count()
        {
            return *_pRefCount;
        }

        ~shared_ptr()
        {
            Release();
        }

        // 像指针一样使用
        T& operator*()
        {
            return *_ptr;
        }

        T* operator->()
        {
            return _ptr;
        }

        T* get() const
        {
            return _ptr;
        }
    private:
        T* _ptr;
        int* _pRefCount;
        mutex* _pmtx;
    };

    // 简化版本的weak_ptr实现
    template<class T>
    class weak_ptr
    {
    public:
        weak_ptr()
            :_ptr(nullptr)
        {}

        weak_ptr(const shared_ptr<T>& sp)
            :_ptr(sp.get())
        {}

        weak_ptr<T>& operator=(const shared_ptr<T>& sp)
        {
            _ptr = sp.get();

            return *this;
```

```
        }

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

        T* operator->()
        {
            return _ptr;
        }


    private:
        T* _ptr;
    };
}

// shared_ptr智能指针是线程安全的吗？
// 是的，引用计数的加减是加锁保护的。但是指向资源不是线程安全的

// 指向堆上资源的线程安全问题是访问的人处理的，智能指针不管，也管不了
// 引用计数的线程安全问题，是智能指针要处理的
//int main()
//{
//  bit::shared_ptr<int> sp1(new int);
//  bit::shared_ptr<int> sp2(sp1);
//  bit::shared_ptr<int> sp3(sp1);
//
//  bit::shared_ptr<int> sp4(new int);
//  bit::shared_ptr<int> sp5(sp4);
//
//  //sp1 = sp1;
//  //sp1 = sp2;
//
//  //sp1 = sp4;
//  //sp2 = sp4;
//  //sp3 = sp4;
//
//  *sp1 = 2;
//  *sp2 = 3;
//
//  return 0;
//}
```

**std::shared_ptr的线程安全问题**

通过下面的程序我们来测试shared_ptr的线程安全问题。需要注意的是shared_ptr的线程安全分为两方面：

1. 智能指针对象中引用计数是多个智能指针对象共享的，两个线程中智能指针的引用计数同时++或--，这个操作不是原子的，引用计数原来是1，++了两次，可能还是2.这样引用计数就错乱了。会导致资源未释放或者程序崩溃的问题。所以只能指针中引用计数++、--是需要加锁的，也就是说引用计数的操作是线程安全的。
2. 智能指针管理的对象存放在堆上，两个线程中同时去访问，会导致线程安全问题。

```
// 1.演示引用计数线程安全问题，就把AddRefCount和SubRefCount中的锁去掉
```

```
// 2.演示可能不出现线程安全问题，因为线程安全问题是偶现性问题，main函数的n改大一些概率就变大了，就容易出现了。
// 3.下面代码我们使用SharedPtr演示，是为了方便演示引用计数的线程安全问题，将代码中的SharedPtr换成shared_ptr进行测试，可以验证库的shared_ptr，发现结论是一样的。
struct Date
{
    int _year = 0;
    int _month = 0;
    int _day = 0;
};

void SharePtrFunc(bit::shared_ptr<Date>& sp, size_t n, mutex& mtx)
{
    cout << sp.get() << endl;

    for (size_t i = 0; i < n; ++i)
    {
        // 这里智能指针拷贝会++计数，智能指针析构会--计数，这里是线程安全的。
        bit::shared_ptr<Date> copy(sp);

        // 这里智能指针访问管理的资源，不是线程安全的。所以我们看看这些值两个线程++了2n次，但是最终看到的结果，并一定是加了2n
        {
            unique_lock<mutex> lk(mtx);
            copy->_year++;
            copy->_month++;
            copy->_day++;
        }
    }
}

int main()
{
    bit::shared_ptr<Date> p(new Date);
    cout << p.get() << endl;
    const size_t n = 100000;
    mutex mtx;
    thread t1(SharePtrFunc, std::ref(p), n, std::ref(mtx));
    thread t2(SharePtrFunc, std::ref(p), n, std::ref(mtx));

    t1.join();
    t2.join();

    cout << p->_year << endl;
    cout << p->_month << endl;
    cout << p->_day << endl;

    cout << p.use_count() << endl;

    return 0;
}
```

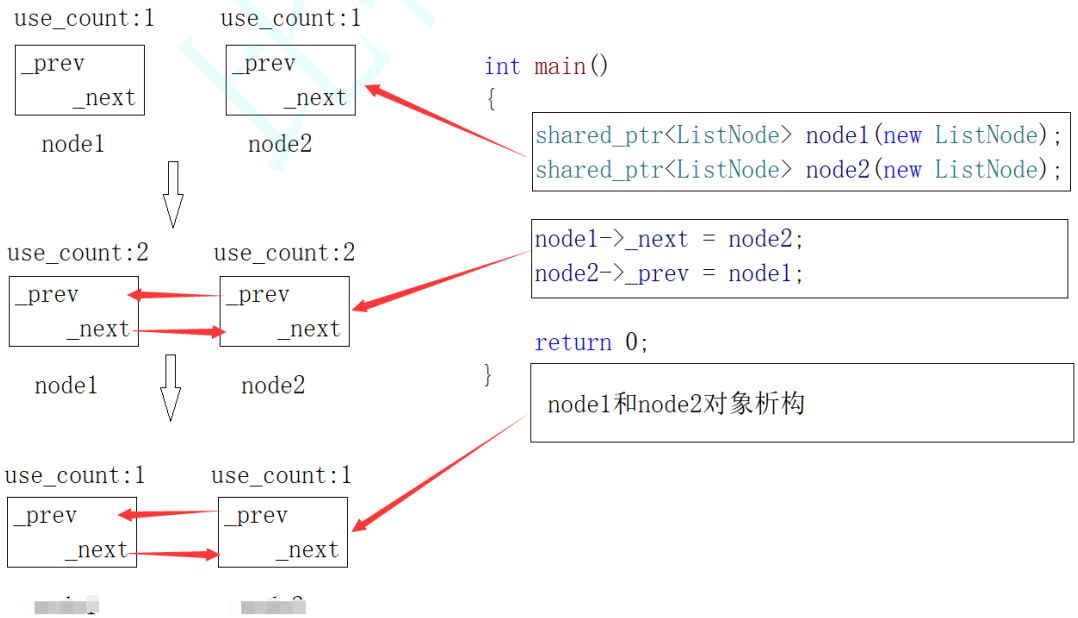
### std::shared_ptr的循环引用

```
struct ListNode
{
    int _data;
```

```
    shared_ptr<ListNode> _prev;
    shared_ptr<ListNode> _next;

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

int main()
{
    shared_ptr<ListNode> node1(new ListNode);
    shared_ptr<ListNode> node2(new ListNode);
    cout << node1.use_count() << endl;
    cout << node2.use_count() << endl;

    node1->_next = node2;
    node2->_prev = node1;

    cout << node1.use_count() << endl;
    cout << node2.use_count() << endl;

    return 0;
}
```

**循环引用分析：**

1. node1和node2两个智能指针对象指向两个节点，引用计数变成1，我们不需要手动delete。
2. node1的_next指向node2，node2的_prev指向node1，引用计数变成2。
3. node1和node2析构，引用计数减到1，但是_next还指向下一个节点。但是_prev还指向上一个节点。
4. 也就是说_next析构了，node2就释放了。
5. 也就是说_prev析构了，node1就释放了。
6. 但是_next属于node的成员，node1释放了，_next才会析构，而node1由_prev管理，_prev属于node2成员，所以这就叫循环引用，谁也不会释放。




```
// 解决方案：在引用计数的场景下，把节点中的_prev和_next改成weak_ptr就可以了
// 原理就是，node1->_next = node2;和node2->_prev = node1;时weak_ptr的_next和
_prev不会增加node1和node2的引用计数。
struct ListNode
```

```
{
    int _data;
    weak_ptr<ListNode> _prev;
    weak_ptr<ListNode> _next;

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

int main()
{
    shared_ptr<ListNode> node1(new ListNode);
    shared_ptr<ListNode> node2(new ListNode);
    cout << node1.use_count() << endl;
    cout << node2.use_count() << endl;

    node1->_next = node2;
    node2->_prev = node1;

    cout << node1.use_count() << endl;
    cout << node2.use_count() << endl;

    return 0;
}
```

**如果不是new出来的对象如何通过智能指针管理呢？其实shared_ptr设计了一个删除器来解决这个问题（ps：删除器这个问题我们了解一下）**

```
// 仿函数的删除器
template<class T>
struct FreeFunc {
    void operator()(T* ptr)
    {
        cout << "free:" << ptr << endl;
        free(ptr);
    }
};

template<class T>
struct DeleteArrayFunc {
    void operator()(T* ptr)
    {
        cout << "delete[]" << ptr << endl;
        delete[] ptr;
    }
};

int main()
{
    FreeFunc<int> freeFunc;
    std::shared_ptr<int> sp1((int*)malloc(4), freeFunc);

    DeleteArrayFunc<int> deleteArrayFunc;
    std::shared_ptr<int> sp2((int*)malloc(4), deleteArrayFunc);


    std::shared_ptr<A> sp4(new A[10], [](A* p){delete[] p; });
```

```
    std::shared_ptr<FILE> sp5(fopen("test.txt", "w"), [](FILE* p)
{fclose(p); });

    return 0;
}
```

## 4.C++11和boost中智能指针的关系

1. C++ 98 中产生了第一个智能指针auto_ptr.
2. C++ boost给出了更实用的scoped_ptr和shared_ptr和weak_ptr.
3. C++ TR1，引入了shared_ptr等。不过注意的是TR1并不是标准版。
4. C++ 11，引入了unique_ptr和shared_ptr和weak_ptr。需要注意的是unique_ptr对应boost的scoped_ptr。并且这些智能指针的实现原理是参考boost中的实现的。