Lesson04--智能指针

【本节目标】

- 1.为什么需要智能指针?
- 2.智能指针的使用及原理
- 3.C++11和boost中智能指针的关系
- 4.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。

```
#include <vector>
void _MergeSort(int* a, int left, int right, int* tmp)
{
    if (left >= right) return;

    int mid = left + ((right - left) >> 1);
    // [left, mid]
    // [mid+1, right]
    _MergeSort(a, left, mid, tmp);
    _MergeSort(a, mid + 1, right, tmp);

    int begin1 = left, end1 = mid;
    int begin2 = mid + 1, end2 = right;
    int index = left;
    while (begin1 <= end1 && begin2 <= end2)
    {
        if (a[begin1] < a[begin2])
            tmp[index++] = a[begin1++];
        else
            tmp[index++] = a[begin2++];
    }

    while (begin1 <= end1)
        tmp[index++] = a[begin1++];

    while (begin2 <= end2)
        tmp[index++] = a[begin2++];

    memcpy(a + left, tmp + left, sizeof(int)*(right - left + 1));
}
```

比特科技

```
void MergeSort(int* a, int n)
{
    int* tmp = (int*)malloc(sizeof(int)*n);
    _MergeSort(a, 0, n - 1, tmp);

    // 这里假设处理了一些其他逻辑
    vector<int> v(1000000000, 10);
    // ...

    // free(tmp);
}

int main()
{
    int a[5] = { 4, 5, 2, 3, 1 };
    MergeSort(a, 5);

    return 0;
}
```

问题分析：上面的问题分析出来我们发现有以下两个问题?

1. malloc出来的空间，没有进行释放，存在内存泄漏的问题。
2. 异常安全问题。如果在malloc和free之间如果存在抛异常，那么还是有内存泄漏。这种问题就叫异常安全。

2.智能指针的使用及原理

2.1 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;
    }
```

比特科技

```
private:
    T* _ptr;
};

void MergeSort(int* a, int n)
{
    int* tmp = (int*)malloc(sizeof(int)*n);
    // 讲tmp指针委托给了sp对象，用时老师的话说给tmp指针找了一个可怕的女朋友！天天管着你，直到你go
die^^
    SmartPtr<int> sp(tmp);
    // _MergeSort(a, 0, n - 1, tmp);


    // 这里假设处理了一些其他逻辑
    vector<int> v(1000000000, 10);
    // ...
}

int main()
{
    try {
        int a[5] = { 4, 5, 2, 3, 1 };
        MergeSort(a, 5);
    }
    catch(const exception& e)
    {
        cout<<e.what()<<endl;
    }

    return 0;
}
```

2.2 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:
```

比特科技

```
    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。

2.3 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。

```
// C++库中的智能指针都定义在memory这个头文件中
#include <memory>

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

    int _year;
    int _month;
    int _day;
};

int main()
{

    auto_ptr<Date> ap(new Date);
```

比特科技

```
    auto_ptr<Date> copy(ap);

    // auto_ptr的问题：当对象拷贝或者赋值后，前面的对象就悬空了
    // C++98中设计的auto_ptr问题是非常明显的，所以实际中很多公司明确规定了不能使用auto_ptr
    ap->_year = 2018;

    return 0;
}
```

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AutoPtr来了解它的原理

```
// 模拟实现一份简答的AutoPtr,了解原理
template<class T>
class AutoPtr
{
public:
    AutoPtr(T* ptr = NULL)
        : _ptr(ptr)
    {}

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

    // 一旦发生拷贝，就将ap中资源转移到当前对象中，然后另ap与其所管理资源断开联系，
    // 这样就解决了一块空间被多个对象使用而造成程序奔溃问题
    AutoPtr(AutoPtr<T>& ap)
        : _ptr(ap._ptr)
    {
         ap._ptr = NULL;
    }

    AutoPtr<T>& operator=(AutoPtr<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;
    }


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

比特科技

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

int main()
{
    AutoPtr<Date> ap(new Date);

    // 现在再从实现原理层来分析会发现，这里拷贝后把ap对象的指针赋空了，导致ap对象悬空
    // 通过ap对象访问资源时就会出现问题。
    AutoPtr<Date> copy(ap);
    ap->_year = 2018;

    return 0;
}
```

2.4 std::unique_ptr

> C++11中开始提供更靠谱的unique_ptr

[unique_ptr文档]

```
int main()
{
    unique_ptr<Date> up(new Date);

    // unique_ptr的设计思路非常的粗暴-防拷贝，也就是不让拷贝和赋值。
    unique_ptr<Date> copy(ap);

    return 0;
}
```

> 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

```
// 模拟实现一份简答的UniquePtr,了解原理
template<class T>
class UniquePtr
{
public:
    UniquePtr(T * ptr = nullptr)
        : _ptr(ptr)
    {}

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

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

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

比特科技

```
private:
    // C++98防拷贝的方式：只声明不实现+声明成私有
    UniquePtr(UniquePtr<T> const &);
    UniquePtr & operator=(UniquePtr<T> const &);

     // C++11防拷贝的方式：delete
    UniquePtr(UniquePtr<T> const &) = delete;
    UniquePtr & operator=(UniquePtr<T> const &) = delete;

private:
    T * _ptr;
};
```

2.5 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

```
int main()
{
    // shared_ptr通过引用计数支持智能指针对象的拷贝
    shared_ptr<Date> sp(new Date);
    shared_ptr<Date> copy(sp);

    cout << "ref count:" << sp.use_count() << endl;
    cout << "ref count:" << copy.use_count() << endl;

    return 0;
}
```

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。

```
// 模拟实现一份简答的SharedPtr,了解原理
#include <thread>
#include <mutex>

template <class T>
class SharedPtr
{
public:
    SharedPtr(T* ptr = nullptr)
```

比特科技

```
        : _ptr(ptr)
        , _pRefCount(new int(1))
        , _pMutex(new mutex)
    {}

    ~SharedPtr() {Release();}

    SharedPtr(const SharedPtr<T>& sp)
        : _ptr(sp._ptr)
        , _pRefCount(sp._pRefCount)
        , _pMutex(sp._pMutex)
    {
            AddRefCount();
    }

    // sp1 = sp2
    SharedPtr<T>& operator=(const SharedPtr<T>& sp)
    {
        //if (this != &sp)
        if (_ptr != sp._ptr)
        {
            // 释放管理的旧资源
            Release();

            // 共享管理新对象的资源，并增加引用计数
            _ptr = sp._ptr;
            _pRefCount = sp._pRefCount;
            _pMutex = sp._pMutex;


            AddRefCount();
        }

        return *this;
    }

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

    int UseCount() {return *_pRefCount;}
    T* Get() { return _ptr; }

    void AddRefCount()
    {
        // 加锁或者使用加1的原子操作
        _pMutex->lock();
        ++(*_pRefCount);
        _pMutex->unlock();
    }
private:
    void Release()
    {

        bool deleteflag = false;
```

比特科技

```
        // 引用计数减1，如果减到0，则释放资源
        _pMutex.lock();
        if (--(*_pRefCount) == 0)
        {
            delete _ptr;
            delete _pRefCount;
             deleteflag = true;
        }
        _pMutex.unlock();

        if(deleteflag == true)
            delete _pMutex;
    }
private:
    int* _pRefCount; // 引用计数
    T*   _ptr;       // 指向管理资源的指针
    mutex* _pMutex;      // 互斥锁
};

int main()
{
    SharedPtr<int> sp1(new int(10));
    SharedPtr<int> sp2(sp1);
    *sp2 = 20;
    cout << sp1.UseCount() << endl;
    cout << sp2.UseCount() << endl;

    SharedPtr<int> sp3(new int(10));
    sp2 = sp3;
    cout << sp1.UseCount() << endl;
    cout << sp2.UseCount() << endl;
    cout << sp3.UseCount() << endl;

    sp1 = sp3;
    cout << sp1.UseCount() << endl;
    cout << sp2.UseCount() << endl;
    cout << sp3.UseCount() << endl;

    return 0;
}
```

比特科技

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。
void SharePtrFunc(SharedPtr<Date>& sp, size_t n)
{
    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。
        SharedPtr<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
        copy->_year++;
        copy->_month++;
        copy->_day++;
    }
}

int main()
{
    SharedPtr<Date> p(new Date);
    cout << p.Get() << endl;

    const size_t n = 100;
    thread t1(SharePtrFunc, p, n);
    thread t2(SharePtrFunc, p, n);

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

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

    return 0;
}
```

比特科技

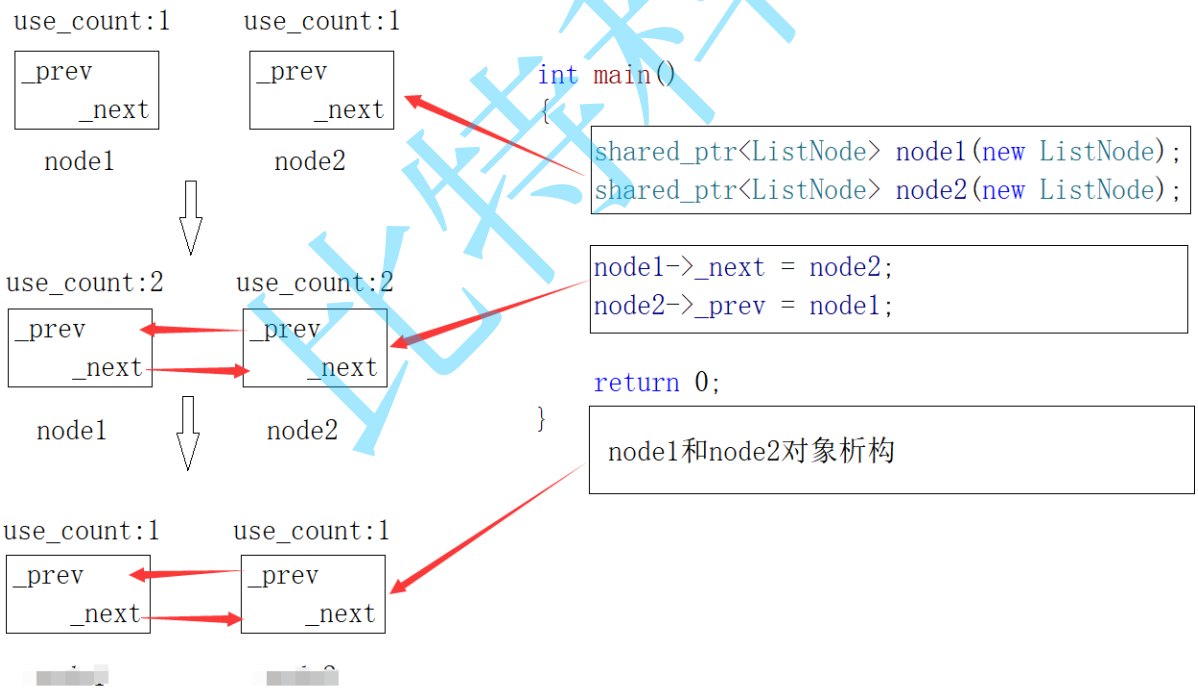
std::shared_ptr的循环引用

```
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;
    shared_ptr<int> sp1((int*)malloc(4), freeFunc);

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

    return 0;

}
```

比特科技

3.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。

4.RAII扩展学习

RAII思想除了可以用来设计智能指针，还可以用来设计守卫锁，防止异常安全导致的死锁问题。

```
#include <thread>
#include <mutex>

// C++11的库中也有一个lock_guard，下面的LockGuard造轮子其实就是为了学习他的原理
template<class Mutex>
class LockGuard
{
public:
    LockGuard(Mutex& mtx)
        :_mutex(mtx)
    {
        _mutex.lock();
    }

    ~LockGuard()
    {
        _mutex.unlock();
    }

    LockGuard(const LockGuard<Mutex>&) = delete;

private:
    // 注意这里必须使用引用，否则锁的就不是一个互斥量对象
    Mutex& _mutex;
};

mutex mtx;
static int n = 0;

void Func()
{
    for (size_t i = 0; i < 1000000; ++i)
    {
        LockGuard<mutex> lock(mtx);
        ++n;
    }

}
```

比特科技

```
int main()
{
    int begin = clock();
    thread  t1(Func);
    thread  t2(Func);

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

    int end = clock();

    cout << n << endl;
    cout <<"cost time:" <<end - begin << endl;

    return 0;
}
```

【作业】

1. 理解为什么需要智能指针？重点注意智能指针是一种预防型的内存泄漏的解决方案。智能指针在C++没有垃圾回收器环境下，可以很好的解决异常安全等带来的内存泄漏问题，
2. 理解RAII.
3. 理解auto_ptr、unique_ptr、shared_ptr等智能指针的使用及原理。

比特科技