華中科技大學

智能指针 smart pointer

许向阳

xuxy@hust.edu.cn

路线图

- 指针用法中存在什么问题？
- 解决指针使用缺陷的思路？
- 什么是智能指针？ smart pointer
- 智能指针有哪几种？

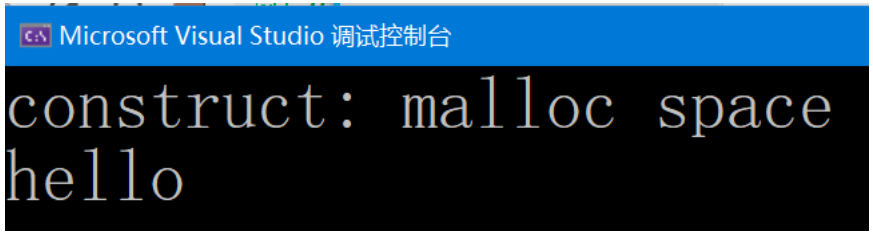
  unique_ptr、shared_ptr、weak_ptr
- 怎么使用智能指针？
- 实现智能指针的原理是什么？

指针使用中的问题


```
#include <iostream>
using namespace std;
class A {
private:      char* s;      int   len;
public:
   A(int n) {
      cout << "construct: malloc space" << endl;
      s = new char[n];          len = n;
   }
   ~A() {    cout <<  “deconstruct: free space”  << endl;
             delete s;
             s = 0;
             len=0;
         }
   void setString(const char* src) {   strcpy_s(s, len, src);    }
   void getString( ) {   cout<<s<<endl;  }
};
```

指针使用中的问题

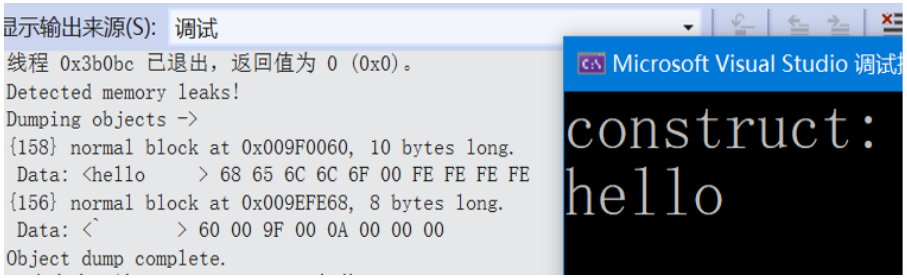

```
void f()
{   A* pa = new A(10);
    pa->setString("hello");
    pa->getString();
}

int main()
{    f();
    _CrtDumpMemoryLeaks();
}
```



Q: 程序有什么缺陷？

Memory leaks!

有哪两处泄露?



A对象本身 ： 8字节
A对象体外空间：10字节

注意正确的测试方法

指针使用中的问题


```
void f()
{   A* pa = new A(10);
    pa->setString("hello");
    pa->getString();
    delete pa;
}
```

解决问题的方法：
加释放空间的语句 delete pa;

delete pa 的执行过程

- 先调用 A 类的析构函数，释放成员s指向的（对象体外）空间 { 10个字节 }
- 再调用 free ， 释放 对象本身（由new） 分配的空间 {8个字节，2个数据成员}

new 的执行过程：先分配空间，再调用构造函数初始化！

指针使用中的问题


```
void pointer_problem()
{
    int* p_leak;
    p_leak = new int[10];  // 空间泄露
                           //  指针移动，未释放原有空间
    int* p1 = new int[20];
    int* p2 = new int[10];
    p1 = p2;
    delete []p2;
    p2 = nullptr;
          //  两个指针，指向同一空间后，释放空间，
          //  导致用另一个指针操作出问题
    //   delete []p1;
    //   p1 = nullptr;
}
```

解决问题的思路

- 对普通指针加了一层封装机制

  定义一个类，类中含原始指针

  定义类的成员函数，实现 →

- 利用 C++ 的对象的生命期管理机制

解决问题的思路

Q：有无方法，防止漏写 释放空间的语句？

即，能够“自动”释放空间？

- **一个对象，在生命周期结束时，会自动调用析构函数！**
- **将指针封装成一个对象！　解决问题的思路**

```
class PointerEncup {
private:
   A* p;
public:
   PointerEncup(A* ptr) {     p = ptr;
   }
   ~PointerEncup() {
      delete p;          p = 0;
   }
};
```

```
int f( )
{
   A* pa = new A(10);
   pa->setString("hello");
   pa->getString();
   PointerEncup smartp(pa);
}
```

```
construct: malloc space
hello
deconstruct: free space
```

指针封装成对象

Q：怎样通过指针对象 像原来的指针一样使用？

（p 是私有成员）

方法1：加一个返回指针的public函数

```
class PointerEncup {
  private:
     A* p;
  public:   ……
   A* getPointer() {
         return p;
   }
};
```

```
PointerEncup smartp = { pa };
smartp.getPointer()->setString("good");
smartp.getPointer()->getString( );
```

不够简便！

指针封装成对象

Q：怎样通过指针对象 像原来的指针一样使用？

方法2：定义 -> 运算符

```
class PointerEncup {
 private:
    A* p;
 public:   ……
 A* operator->() {
      return p;
 }
};
```

```
PointerEncup smartp = { pa };
smartp -> setString("good");
smartp -> getString( );
```

类似的 * 运算符

```
A& operator *()
{   return *p;      }
```

```
(*smartp) . getString( );
```

指针封装成对象


```
A* pa = new A(10);
PointerEncup smartp = { pa };
```

Q：额外定义了指针 pa, 能否省掉？

```
PointerEncup smartp =  new A(10);

smartp -> setString("good");
smartp -> getString( );

PointerEncup smartp ( new A(10) );
```

指针封装成对象

- 一个对象，在生命周期结束时，会自动调用析构函数！
- 对指针（也称 raw 指针）进行封装
- 定义一个 类，包含

  指针成员，接管 raw 指针

  构造函数，初始化成员

  析构函数，摧毁指针成员指向的对象

  运算符->、*， 像raw指针一样使用

  增加其他函数成员，实现更完备的功能，这就需要明确指针能做哪些操作！

指针封装成对象

Q：前面的做法还有什么问题？

```
class PointerEncup {
  private:
     A* p;
  public:   ……
  A* operator->() {
        return p; }
};
```

对 A类指针，定义了一个类；

对于 B 类指针，怎么办？

```
template <class T>
class PointerEncup {
private:      T* p;
public:     PointerEncup(T* ptr) {       p = ptr;      }
    ~PointerEncup() { if (p) delete p;   p = NULL;  }
    T* operator->() {          return p;      }
    T& operator *() {          return *p;    }
};
```

```
PointerEncup<A> smartp = new A(10);
smartp->setString("good");
smartp->getString();
```

类模板

C++ 的智能指针

unique_ptr 独占指针

shared_ptr 共享指针 （计数指针）

weak_ptr 弱指针

unique_ptr 独享所有权的智能指针

- 无法进行复制构造、赋值操作；
- 只能进行移动操作；
- 无法使两个 unique_ptr 指向同一个对象，只能指向一个对象；
- 指向其他对象时，之前指向的对象会被摧毁；
- unique_ptr 对象会在它们自身被销毁时，自动删除它们管理的对象；
- unique_ptr支持创建数组对象方法。

unique_ptr

- 三种创建方式

- 通过已有裸指针创建

  设有类型 A，以及指向 A 的指针 pa ;　即 A *pa = …

  unique_ptr<A>　p1(pa);

- 通过new创建

  unique_ptr<A>　p1(new A(…));

- 通过make_unique 创建 (推荐)

  unique_ptr<A>　p1= make_unique <A>(…);

unique_ptr


```
int main()
{   unique_ptr<A> p2(new A(10));
    p2->setString("good");
    p2->getString();
}
```

```
construct: malloc space
good
deconstruct: free space
```

```
int main()
{   unique_ptr<A> p3=make_unique<A>(10);
    p3->setString("very good");
    p3->getString();
}
```

```
construct: malloc space
very good
deconstruct: free space
```

Q：能否 p2 = p3 ?

unique_ptr& operator=(const unique_ptr&) = delete;

Q: 能否用一个智能指针构造另一个指针？

如 unique_ptr<A> p4(p2);

unique_ptr(const unique_ptr&) = delete;

unique_ptr

头文件：memory

```
template <class _Ty, class _Dx /* = default_delete<_Ty> */>
class unique_ptr { // non-copyable pointer to an object
public:
    using pointer      = typename _Get_deleter_pointer_type<_Ty,
remove_reference_t<_Dx>>::type;
    using element_type = _Ty;
    using deleter_type = _Dx;

    template <class _Dx2 = _Dx, _Unique_ptr_enable_default_t<_Dx2> = 0>
    constexpr unique_ptr() noexcept : _Mypair(_Zero_then_variadic_args_t{}) {}

    template <class _Dx2 = _Dx, _Unique_ptr_enable_default_t<_Dx2> = 0>
    constexpr unique_ptr(nullptr_t) noexcept :
_Mypair(_Zero_then_variadic_args_t{}) {}
    ……
```

unique_ptr

- 三种创建方式

  通过已有裸指针创建

  通过new创建

  通过make_unique 创建 (推荐)

- 通过 get( ) 获取被指向对象的地址

- 实现了 -> 和 *

  通过-> 调用成员函数

  通过 * 调用 dereferencing 解引用

unique_ptr


unique_ptr<A> up = make_unique<A>(10);
up.
★ get
★ reset
★ release
get
get_deleter
operator bool
operator*
operator->
operator=
public : A *std::unique_ptr<A>::get() const
文件: memory
基于此上下文的 ★ IntelliCode 建议

shared_ptr 指针


```
void smart_point2object()
{
    shared_ptr<Person> ptr(new Person("xuxy", 50));
    cout <<"Person Size :"<< sizeof(Person) << endl;
    cout << "shared_ptr<Person> Size :" ;
    cout << sizeof(shared_ptr<Person>) << endl;
    ptr->displayInfo();
    ptr->modifyName("xuxiangyang");
    ptr->displayInfo();
}
```

```
Person Size :24
shared_ptr<Person> Size :8
name : xuxy  age: 50
name : xuxiangyang  age: 50
```

shared_ptr智能指针

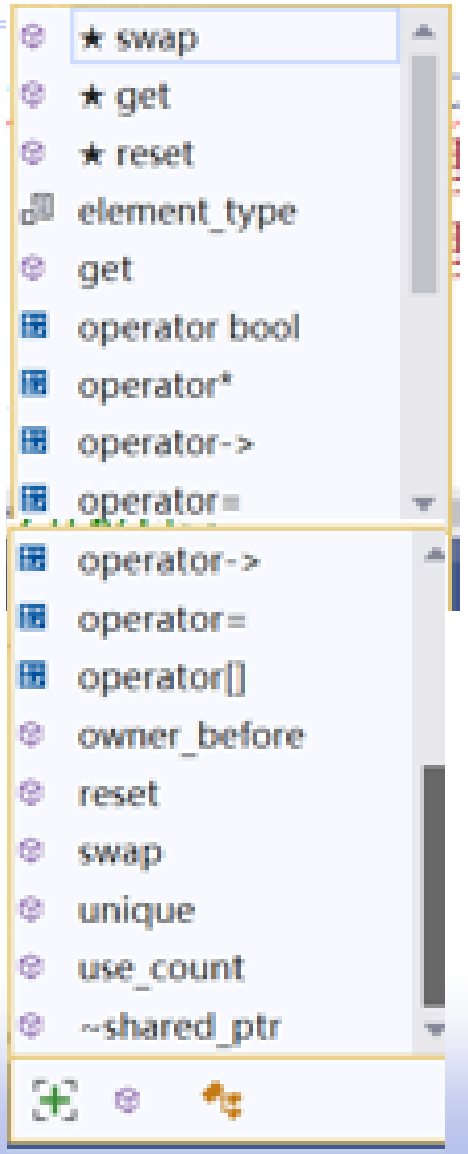
memory

```
shared_ptr& operator=(……)
T* operator->() const noexcept {
        return get();     // 返回原始指针
}

T& operator*() const noexcept {
        return *get();
}

E& operator[](ptrdiff_t _Idx) const noexcept {
        return get()[_Idx];
}
```

shared_ptr智能指针


```
void smart_pointer_example()
{
    shared_ptr<int> ptr1(new int[3]);
    cout << "ptr1 管理的内存引用计数：  " << ptr1.use_count() << endl;
     shared_ptr<int> ptr2 = move(ptr1);    // 移动构造
    cout << "ptr1 管理的内存引用计数：  " << ptr1.use_count() << endl;
    cout << "ptr2 管理的内存引用计数：  " << ptr2.use_count() << endl;
    shared_ptr<int> ptr3 = ptr2;    // 拷贝构造
    cout << "ptr2 管理的内存引用计数：  " << ptr2.use_count() << endl;
    cout << "ptr3 管理的内存引用计数：  " << ptr3.use_count() << endl;
    // 通过 make_shared 初始化
    shared_ptr<int> ptr4 = make_shared<int>(8);
    ptr4.reset();
    ptr4.reset(new int[10]);
    int* p = ptr4.get();
}
```

shared_ptr智能指针




```
ptr1 管理的内存引用计数:  1
ptr1 管理的内存引用计数:  0
ptr2 管理的内存引用计数:  1
ptr2 管理的内存引用计数:  2
ptr3 管理的内存引用计数:  2
```

shared_ptr 总结

- 智能指针的一种通用实现技术是使用引用计数；
- 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的指针指向同一对象；
- 对普通指针加了一层封装机制，这样的一层封装机制的目的是为了使得智能指针可以方便的管理一个对象的生命期。

不同智能指针的比较

華中科技大學

特性	unique_ptr	shared_ptr	weak_ptr
所有权	独占	共享	无所有权
拷贝语义	禁止拷贝，只能移动	允许拷贝	允许拷贝
性能	零开销（与裸指针相当）	有引用计数开销	有额外开销
使用场景	单一所有者场景	多所有者场景	解决循环引用

实验中的应用


```
Executor * NewExecutor(const Pose &pose)
                { return new ExecutorImpl(pose); }

unique_ptr<Executor> executor(Executor::NewExecutor());

executor->Execute("FFM");
```

总结

智能指针是C++中用于自动管理动态分配内存的工具。

- 通过RAII（资源获取即初始化，Resource Acquisition Is Initialization）原则来确保内存被正确释放；
- 自动内存管理，防止内存泄漏；
- 异常安全：当代码中发生异常时，智能指针能够确保资源被释放；
- 避免悬空指针和重复释放；
- 明确所有权语义：智能指针明确指出了对动态分配内存的所有权。unique_ptr 独占所有权，shared_ptr 共享所有权，weak_ptr 非拥有引用，使代码更易理解和维护。

unique_ptr 总结

- unique_ptr实现独占所有权，即同一时间只能有一个unique_ptr拥有所指对象。当unique_ptr被销毁时，它会自动删除其所拥有的对象。
- 在unique_ptr类内部，保存一个原始指针，用于指向动态分配的对象。
- 禁止拷贝和赋值（通过将拷贝构造函数和赋值运算符设置为delete），但允许移动语义（移动构造函数和移动赋值运算符）。
- 在析构函数中，调用删除器来释放资源。默认情况下，使用delete操作符。

shared_ptr 总结

- 多个shared_ptr可以共享同一个对象。
- 每个shared_ptr内部包含两个指针：一个指向所管理的对象，另一个指向控制块（control block）。控制块包含引用计数（use count）和弱引用计数（weak count），以及分配器（allocator）和删除器（deleter）等信息。
- 当创建一个shared_ptr时，引用计数初始化为1。
- 拷贝构造shared_ptr时，引用计数加1。
- 赋值操作时，左边的shared_ptr原来的引用计数减1，右边的引用计数加1。
- 当shared_ptr析构时，引用计数减1。如果引用计数变为0，则删除所管理的对象，并释放控制块内存。

Game Over