Morden cpp notes — smart pointer

Friday, December 4, 2020 2:40 PM

In morden C++, never use naked pointer, use smart pointer instead. There are two main types:
- unique_ptr
- shared_ptr

* Unique_ptr.

need to specify the type during declaration.

```
unique_ptr<int> ptr1(new int(10)); // int smart pointer.
assert(*ptr1 = 10); //可以使用*取内容.
assert(ptr1 != nullptr); //可以判断是否为空指针

unique_ptr<string> ptr2(new string("hello")); //string智能指针.
assert(*ptr2 == "hello"); //可以用*取内容.
assert(ptr2->size() == 5); //可以用->调用成员函数.
```

unique_ptr虽然名字叫指针,但实际并不是,而是一个对象.不要企图调用delete,它会自动管理初始化的指针,在离开作用域的时候析构释放内存.

另外,它也没有定义加减运算,不能随意移动指针地址,完全避免了越界,让代码更安全。

```
ptr1++;      }
ptr2 += 2;   } 编译错误.
```

除了调用delete,加减运算,另个错误是当作普通对象,不初始化,而是声明后直接用:

```
unique_ptr<int> ptr3; //未初始化智能指针.
*ptr3 = 42; //错误!操作了空指针.
```

未初始化的unique_ptr表示空指针,操作空指针会导致core dump错误.

为了避免这种低级错误,可以调用工厂函数make_unique(),强制创建智能指针时必须初始化.同时可以用auto,少写一些代码.

```
auto ptr3 = make_unique<int>(42); //工厂函数创建智能指针.
assert(ptr3 && *ptr3 == 42);
auto ptr4 = make_unique<string>("god of war); //工厂函数创建智能指针.
assert(!ptr4->empty());
```

不过, make_unique要求C++14.也可以用C++11实现一个简化版的:

```
template<class T, class... Args>   //可变参数指针.
std::unique_ptr<T>     //返回智能指针.
my_make_unique(Args&&... args) //可变参数模板的入口参数.
{
  return std::unique_ptr<T>(     //构造智能指针
      new T(std::forward<Args>(args)...)); //完美转发.
}.
```

使用unique_ptr,要特别注意所有权的问题.指针的所有权是唯一的,不允许共享,任何时候只能一个人持有它.为了实现这个目的,unique_ptr应用了C++的转移move语义,同时禁止了拷贝赋值,所以在向另一个unique_ptr赋值的时候,必须用std::move()函数显式地声明所有权的转移.赋值操作后,指针的所有权被转走了,原来的unique_ptr变成了空指针,新的unique_ptr接替了管理权,保证了所有权的唯一性。

```
auto ptr1 = make_unique<int>(42); //工厂函数创建智能指针.
assert(ptr1 && *ptr1 == 42); //此时智能指针有效.
auto ptr2 = std::move(ptr1); //使用move转移所有权.
```

* Shared_ptr.

```
shared_ptr<int> ptr1(new int(10)); //int智能指针.
assert(*ptr1 = 10); //可以使用*取内容.
shared_ptr<string> ptr2(new string("hello")); // string智能指针.
assert(*ptr2 == "hello"); //可以用*取内容.
auto ptr3 = make_shared<int>(42); //工厂函数创建智能指针.
assert(ptr3 && *ptr3 == 42); //可以判断是否为空指针.
auto ptr4 = make_shared<string>("zelda"); //工厂函数创建智能指针.
assert(!ptr4->empty()); //可以使用->调用成员函数.
```

shared_ptr与unique_ptr最大不同:它的所有权是可以被安全共享,支持拷贝赋值,允许被多人同时持有,就像原始指针一样.

```
auto ptr1 = make_shared<int>(42); //工厂函数创建智能指针.
assert(ptr1 && ptr1.unique()); //此时智能指针有效且唯一.
auto ptr2 = ptr1; //直接拷贝赋值,不需要使用move().
assert(ptr1 && ptr2); //此时两个智能指针均有效.
assert(ptr1 == ptr2); //shared_ptr可以直接比较,
//两个智能指针均不唯一,且引用计数为2.
assert(!ptr1.unique() && ptr1.use_count == 2);
assert(!ptr2.unique() && ptr2.use_count == 2);
```

shared_ptr支持安全共享的秘密在于内部使用了"引用计数".
引用计数最开始的时候是1,表示只有一个持有者.如果发生拷贝赋值,也就是共享的时候,引用计数就增加,而发生析构销毁的时候,引用计数就减少.只有当引用计数减少到0,也就是没人使用这个指针的时候,它才会真正调用delete释放内存.

因为shared_ptr具有完整的"值语义"(即可以拷贝赋值),所以可以在任何场合替代原始指针,而不用担心资源回收的问题,比如用于容器存储指针,用于函数安全返回动态创建的对象,

- shared_ptr注意事项.

它的代价是,引用计数的存储和管理都是成本,过度使用会降低运行效率.
另一个要注意的地方是shared_ptr的销毁操作.因为我们把指针交给了shared_ptr去自动管理,但在运行阶段,引用计数的变动是很复杂的,很难知道它真正释放资源的时机,无法像Java, Go那样明确掌控,调整垃圾回收机制.

要特别小心对象的析构函数,不要有非常复杂,严重阻塞的操作.一旦shared_ptr在某个不确定时间点析构释放资源,就会阻塞整个进程或者线程,排查起来很费工夫.

```
class DemoShared  final  //危险的类,不定时的地雷
{
 public:
    DemoShared() = default;
   ~DemoShared()  //复杂的操作会导致shared_ptr析构时世界静止.
   {
      //stop the world ...
   }
};
```

shared_ptr的引用计数,也导致了一个新的问题,就是"循环引用",这在把shared_ptr作为类成员的时候最容易出现:
e.g. 链表节点.

```
class Node final
{
 public:
   using this_type = Node;
   using shared_type = std::shared_ptr<this_type>;

 public:
   shared_type next; //使用智能指针指向下一节点.
};

auto n1 = make_shared<node>(); //工厂函数创建智能指针.
auto n2 = make_shared<node>(); //工厂函数创建智能指针.
assert(n1.use_count() == 1); //引用计数为1.
assert(n2.use_count() == 1);
n1->next = n2; //两个节点互指,形成了循环引用.
n2->next = n1;
assert(n1.use_count() == 2); //引用计数为2.
assert(n2.use_count() == 2); //无法减到0,无法销毁,导致内存泄漏.
```

两个节点指针刚创建时,引用计数是1,但指针互指(拷贝赋值)之后,引用计数都变成了2.这是因为,shared_ptr意识不到循环引用,多算了一次计数,后果就是引用计数无法减到0,无法调用析构函数执行delete,最终导致内存泄漏.

从根本上杜绝循环引用,必须用weak_ptr,专为打破循环引用设计,只观察指针,不会增加引用计数(弱引用),但在需要的时候,可以调用成员函数lock(),获取shared_ptr(强引用).

```
class Node final
{
 public:
   using this_type = Node;
   //注意,别名改为weak_ptr
   using shared_type = std::weak_ptr<this_type>;

 public:
   shared_type next;
};
auto n1 = make_shared<Node>(); //工厂函数创建智能指针
auto n2 = make_shared<Node>(); //工厂函数创建智能指针.
n1->next = n2; //两个节点互指,形成了循环引用.
n2->next = n1;
assert(n1.use_count() == 1); //因为使用了weak_ptr,引用计数为1.
assert(n2.use_count() == 1); //打破循环引用,不会导致内存泄漏.
if (!n1->next.expired()) { //检查指针是否有效.
   auto ptr = n1->next.lock(); // lock()获取shared_ptr
   assert(ptr == n2);
}
```

Tips:
- 如果指针是"独占"使用,就选择unique_ptr,它为裸指针添加了许多限制,更加安全.
- 如果指针是"共享"使用,就用shared_ptr,用法几乎与原始指针一样.
- 应当使用工厂函数make_unique(), make_shared()来创建智能指针,强制初始化,而且还能用auto简化声明.
- shared_ptr有少量的管理成本,不要过度使用.
- 不要再使用裸指针,和new, delet来操作内存了.