Morden cpp 30 lectures — smart pointer

Monday, December 14, 2020　5:17 PM

智能指针就是RAII资源管理功能的自然展现.

回顾:

```
class shape_wrapper{
public:
  explicit shape_wrapper(
     shape* ptr = nullptr)
   : ptr_(ptr) {}
  ~shape_wrapper()
   {
     delete ptr_;
   }
  shape* get() const {return ptr_;}

private:
   shape* ptr_;
};
```

缺点,
- 这个类只适用于shape类
- 该类对象的行为不够像指针.
- 拷贝该类对象会引发程序行为异常.

模板化和易用性.

要让这个类能够包装任意类型的指针,需要把它变成一个类模板:

```
template <typename T>
class smart_ptr{
public:
  explicit smart_ptr(T* ptr = nullptr)
   :ptr_(ptr) {}
  ~smart_ptr()
  {
    delete ptr_;
  }
  T* get() const {return ptr_;}
private:
  T* ptr_;
};
```

和 shape_wrapper 比较,就是在开头增加模板声明

template <typename T>,

然后把代码中的 shape 替换成模板参数 T.

使用模板,只要把 shape_wrapper 改成 smart_ptr<shape>

当前的 smart_ptr 和指针的差异:
- 不能用 * 运算符
- 不能用 -> 运算符指向对象成员
- 不能像指针一样用在布尔表达式里.

要解决上述问题,需要添加:

```
template <typename T>
class smart_ptr{
public:
  ```
  T& operator*() const {reture *ptr_;}
  T* operator->() const {return ptr_;}
  operator bool() const {return ptr_;}
}
```

拷贝构造和赋值.

Eg.

```
smart_ptr<shape> ptr1{create_shape(shape_type::circle)};
smart_ptr<shape> ptr2{ptr1}; ← 运行时会对同一内存释放两次,会导致程序崩溃.
```

最简单的情况是禁止拷贝:

```
template <typename T>
class smart_ptr {
  ```
  smart_ptr(const smart_ptr&) = delete;
  smart_ptr& operator =(const smart_ptr&) = delete;
  ```
};
```

还可在拷贝时转移指针所有权:

```
template <typename T>
class smart_ptr {
 ```
 smart_ptr(smart_ptr& other)
 {
   ptr_ = other.release();   ← 在拷贝构造函数中,调用other的release方法释放它
 }   对指针的所有权.
 smart_ptr& operator =(smart_ptr& rhs)
 {   ← 在赋值函数中,拷贝构造产生一个临时对象,
  smart_ptr(rhs).swap(*this); 再调用swap来交换对指针的所有权.
  return *this;
 }
 ```
 T* release()
  {
   T* ptr = ptr_;
   ptr_ = nullptr;
   return ptr;
  }
  void swap(smart_ptr& rhs)
  {
   using std::swap;
   swap(ptr_, rhs.ptr_);
  }
 ```
};
```

这种方法保证了强异常安全,赋值分为拷贝构造和交换两步,异常只可能在第一步发生,this对象不受影响.无论拷贝构造成功与否,结果只有赋值成功和赋值没有效果,不会因赋值破坏当前对象.

上面实现的最大问题,是容易犯错,如果不小心把它传递给另一个smart_ptr,就不再拥有这个对象了.

移动指针.

```
template <typename T>
class smart_ptr {
 ```
 smart_ptr(smart_ptr&& other)
 {
  ptr_ = other.release();
 }
 smart_ptr& operator =(smart_ptr rhs)
 {
  rhs.swap(*this);
  return *this;
 }
 ```
};
```

修改的地方:
- 把拷贝构造函数中的参数类型 smart_ptr& 改为 smart_ptr&& 现在它成了移动构造函数.
- 把赋值函数中的参数类型 smart_ptr& 改为了 smart_ptr, 在构造参数时直接生成新的智能指针. 从而不需要在函数体中构造临时对象. 现在赋值函数的行为是移动还是拷贝完全依赖于构造参数时用的是移动构造还是拷贝构造.

在C++中,如果提供了移动构造函数而没有手动提供拷贝构造函数,后者将被禁用.

```
smart_ptr<shape> ptr1{create_shape(shape_type::circle)};
smart_ptr<shape> ptr2{ptr1}; 编译出错.
smart_ptr<shape> ptr3;
ptr3 = ptr1; 编译出错.
ptr3 = std::move(ptr1);                     } 这是可以的.
smart_ptr<shape> ptr4{std::move(ptr3)};     }
```

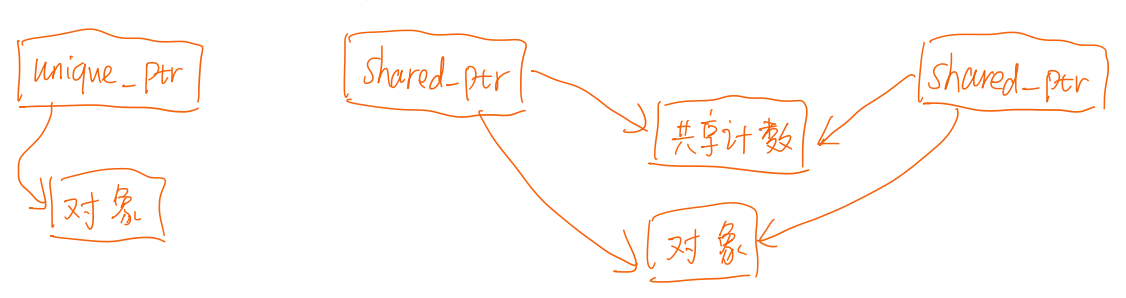
这是C++11中unique_ptr的基本行为.

子类指向基类的指针转换.

让 circle* 隐式转换为 shape*:

```
template <typename U>
smart_ptr(smart_ptr<U>&& other)
{
  ptr_ = other.release();
}
```

引用计数
- 一个对象只能被单个unique_pointer拥有,
- 多个智能指针同时拥有一个对象,当它们全部失效时,这个对象也会被删除.这就是shared_ptr.



多个不同的shared_ptr,在共享对象时也要共享同一个计数.当最后一个指向对象和共享计数的shared_ptr析构时,需要删除对象和共享计数.

共享计数的接口:

```
class shared_count{
public:
  shared_count();
  void add_count();
  long reduce_count();
  long get_count() const;
};
```

shared_count类除构造函数外有三个方法:
- 增加计数,不需返回计数值
- 减少计数,需返回计数值,以判断是否是最后一个指向共享计数的shared_ptr.
- 获取计数

```
class shared_count{
public:
  shared_count() : count_(1) {}

  void add_count()
  {
    ++count_;
  }

  long reduce_count()
  {
   return --count_;
  }

   long get_count() const
    {
     return count_;
    }
private:
    long count_;
};
```

引用计数智能指针
- 构造函数,构造 shared_count.
- 析构函数,看到 ptr_ 非空时,对引用数减1.并在引用数降到零时彻底删除对象和共享计数.
- 私有成员变量.

```
template <typename T>
class smart_ptr{
public:
  explicit smart_ptr(T* ptr = nullptr)
   : ptr_(ptr)
  {
    if(ptr) {
      shared_count_ = new shared_count();
    }
  }

  ~smart_ptr()
  {
    if (ptr_ && !shared_count_->reduce_count()){
      delete ptr_;
      delete shared_count_;
    }
  }
private:
  T* ptr_;
  shared_count* shared_count_;
};
```

为了方便实现赋值,需要一个新的swap成员函数:

```
void swap(smart_ptr& rhs)
{
  using std::swap;
  swap(ptr_, rhs.ptr_);
  swap(shared_count_, rhs.shared_count_);
}
```

拷贝函数和移动构造函数需要更新.

```
template <typename U>
smart_ptr(const smart_ptr<U>& other)
{
  ptr_ = other.ptr_; 在拷贝构造时.
  if(ptr_) {      ← 在指针非空时,引用数+1.
    other.shared_count_->add_count();   ↙ 复制共享计数的指针
    shared_count_ = other.shared_count_;
  }
}


template <typename U>
smart_ptr(smart_ptr<U>&& other)
{
  ptr_ = other.ptr_;
  if(ptr_) {
     shared_count_ = other.shared_count_;
     other.ptr_ = nullptr;
  }  ↑ 移动构造中,不需要调整引用计数,
}      直接把 other.ptr_ 置为空,认为 other
       不再指向该共享对象.
```

然而,上述代码有编译错误:

fatal error: 'ptr_' is a private member of 'smart_ptr<circle>'.

错误原因是模板的各个实例间没有 friend 关系,不能互访私有成员 ptr_ 和 shared_count_.

需要在 smart_ptr 定义中显式声明:

```
template <typename U>
friend class smart_ptr;
```

还要加一个返回引用计数的函数:

```
long use_count() const
{
  if(ptr_) {
    return shared_count_->get_count();
  }
  else
  {
   return 0;
  }
}
```

可以用下面的代码验证一下所有功能:

```
class shape{
public:
  virtual ~shape() {}
};
```

virtual function ↓

A virtual function is a member function which is declared within a base class and is re-defined (overriden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.
- virtual function ensures that correct function is called for an object, regardless of the type of reference (or pointer) used for function call
- they are mainly used to achieve runtime polymorphism.
- functions are declared with a virtual keyword in the base class.
- The resolving of a function call is done at run-time.

Rules for virtual functions:
- virtual functions cannot be static and also cannot be a friend function of another class
- virtual function should be accessed using pointer or reference of base class type to achieve runtime polymorphism.
- The prototype of virtual functions should be same in base as well as derived class.
- They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override or re-define the virtual function, in that case the base class version of function is used.
- A class may have virtual destructor but it cannot have virtual constructor.

Eg.

```
#include <iostream>
using namespace std;

class base {
public:
  virtual void print()
  {
    cout << "print base class" << endl;
  }

  void show()
  {
    cout << "show base class" << endl;
  }
};

class derived: public base{
public:        if we have created a virtual function in the base class
  void print() ← and it is being overriden in the derived class, then we don't need
  {                                   virtual keyword in the derived class,
    cout << "print derived class" << endl;  functions are automatically considered as
  }                                  virtual functions in the derived class.
  void show()
  {
   cout << "show derived class" << endl;
  }
};

int main()
{
  base* bptr;
  derived d;   base class pointer 'bptr' contains the
  bptr = &d; ← address of object 'd' of derived class.

  // virtual function, binded at runtime
  bptr -> print(); ← since print() function is declared with virtual keyword so it will be
                     bound at run-time, pointer is pointing to object of derived class.
  // non-virtual function, binded at compile time
  bptr -> show(); ↖ show() is non-virtual so it will be bound during compile time
}                    pointer is of base type.
Output:
print derived class
show base class
```

- Runtime polymorphism is achieved only through a pointer or reference of base class type.
- A base class pointer can point to the objects of base class as well as to the objects of derived class.

virtual function ↑

```
class Circle: public shape{
public:
  ~circle() { puts("~circle()"); }
};


int main()
{
  smart_ptr<circle> ptr1(new circle());
  printf("use count of ptr1 is %ld\n", ptr1.use_count());

  smart_ptr<shape> ptr2;
  printf("use count of ptr2 was %ld\n", ptr2.use_count());

  ptr2 = ptr1;
  print("use count of ptr2 is now %ld\n", ptr2.use_count());

  if(ptr1) {
    puts("ptr1 is not empty");
  }
}
Output:
use count of ptr1 is 1          }
use count of ptr2 was 0         } 引用计数的变化.
use count of ptr2 is now 2      }
ptr1 is not empty
~circle().  对象被成功删除.
```

指针类型转换

C++里的不同类型强制转换:

- static_cast
- reinterpret_cast
- const_cast
- dynamic_cast

智能指针需要实现类似的转换,需要添加构造函数,允许在对智能指针内部的指针对象赋值时,使用一个现有的智能指针的共享计数.

Eg:

```
template <typename U>
smart_ptr(const smart_ptr<U>& other, T* ptr)
{
  ptr_ = ptr;
  if (ptr_) {
     other.shared_count_-> add_count();
     shared_count_ = other.shared_count_;
  }
}
```

实现 dynamic_pointer_cast:

```
template <typename T, typename U>
smart_ptr<T> dynamic_pointer_cast(
  const smart_ptr<U>& other)
{
  T* ptr = dynamic_cast<T*>(other.get());
  return smart_ptr<T>(other, ptr);
}
```

验证代码中加上:

```
smart_ptr<circle> ptr3 = dynamic_pointer_cast<circle>(ptr2);
print("use count of ptr3 is %ld\n", ptr3.use_count());
Output:
use count of ptr3 is 3
```

完整的 smart_ptr 代码:

```
#include <utility> //std::swap.

class shared_count {

public:
  shared_count() noexcept
   : count_(1) {}

  void add_count() noexcept
  {
   ++count_;
  }

  long reduce_count() noexcept
  {
   return --count_;
  }

  long get_count() const noexcept
  {
   return count_;
  }

private:
  long count_;
};

template<typename T>
class smart_ptr{

public:
   template <typename U>
   friend class smart_ptr;

   explicit smart_ptr(T* ptr = nullptr)
    : ptr_(ptr)
   {
     if(ptr) {
       shared_count_ = new shared_count();
     }
   }

   ~smart_ptr()
   {
     printf("~smart_ptr(): %p\n", this);
     if( ptr_ && !shared_count_->reduce_count() ) {
        delete ptr_;
        delete shared_count_;
     }
   }

   template <typename U>
   smart_ptr(const smart_ptr<U>& other) noexcept.
   {
     ptr_ = other.ptr_;
     if(ptr_) {
       other.shared_count_-> add_count();
       shared_count_ = other.shared_count_;
     }
   }

   template <typename U>
   smart_ptr(smart_ptr<U>&& other) noexcept
   {
     ptr_ = other.ptr_;
     if(ptr_) {
      shared_count_ = other.shared_count_;
      other.ptr_ = nullptr;
     }
   }

   template <typename U>
   smart_ptr(const smart_ptr<U>& other, T* ptr) noexcept
   {
     ptr_ = ptr;
     if(ptr_) {
       other.shared_count_-> add_count();
       shared_count_ = other.shared_count_;
     }
   }

   smart_ptr& operator =(smart_ptr rhs) noexcept
   {
     rhs.swap(*this);
     return *this;
   }

   T* get() const noexcept
   {
     return ptr_;
   }

   long use_count() const noexcept
   {
     if(ptr_) {
       return shared_count_-> get_count();
     }
     else
     {
       return 0;
     }
   }

   void swap(smart_ptr& rhs) noexcept
   {
     using std::swap;
     swap(ptr_, rhs.ptr_);
     swap(shared_count_, rhs.shared_count_);
   }

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

   T* operator->() const noexcept
   {
     return ptr_;
   }

   operator bool() const noexcept
   {
     return ptr_;
   }

private:
   T* ptr_;
   shared_count* shared_count_;
};

template <typename T>
void swap(smart_ptr<T>& lhs, smart_ptr<T>& rhs) noexcept
{
  lhs.swap(rhs);
}

template <typename T, typename U>
smart_ptr<T> static_pointer_cast(
   const smart_ptr<U>& other) noexcept
{
  T* ptr = static_cast<T*>(other.get());
  return smart_ptr<T>(other, ptr);
}

template <typename T, typename U>
smart_ptr<T> reinterpret_pointer_cast(
   const smart_ptr<U>& other> noexcept
{
  T* ptr = reinterpret_cast<T*>(other.get());
  return smart_ptr<T>(other, ptr);
}

template <typename T, typename U>
smart_ptr<T> const_pointer_cast(
   const smart_ptr<U>& other) noexcept
{
  T* ptr = const_cast<T*>(other.get());
  return smart_ptr<T>(other, ptr);
}

template <typename T, typename U>
smart_ptr<T> dynamic_pointer_cast(
  const smart_ptr<U>& other) noexcept
{
  T* ptr = dynamic_cast<T*>(other.get());
  return smart_ptr<T>(other, ptr);
}
```