Умные указатели

МГТУ им. Н.Э. Баумана

April 16, 2016

Умные указатели

Специальные классы, служащие "оберткой" простых указателей на объекты и обеспечивающие автоматическое разрушение объекта при прекращении его использования.

Заголовок:

```
1 #include <memory>
```

Типы указателей:

- std::unique_ptr<T>
- std::shared_ptr<T>
- std::weak_ptr<T>

Какую проблему решают

```
1  void f() // this function may throw exception
2  {
3    throw std::runtime_error("Bad stuff");
4  }
5
6  void g() {
7    int* intPtr = new int;
8    // ...
9    f(); // f() has thrown exception
10   // ...
11   delete intPtr; // intPtr won't be deleted
12 }
13
14 int main() {
15
16   // ...
17   try {
18     g();
19   } catch(const std::exception& e) {
20     std::cout << e.what();
21     return -1;
22   }
23   return 0;
```

std::unique_ptr

Определение

Умный указатель, который:

- единолично владеет объектом через его указатель;
- разрушает объект при выходе unique_ptr из области видимости.

```
1  struct C {
2    int i;
3  }
4  void f() // this function may throw exception
5  {
6    throw std::runtime_error("Bad stuff");
7  }
8
9  void g() {
10   unique_ptr<C> smartCPtr(new C);
11   C otherC = *smartCPtr;
12   int j = smartCPtr->i;
13
14   // ...
15   f(); // f() has thrown exception, unique_ptr calls delete
16   // ...
17   // no need to call delete
18 }
```

std::unique_ptr

Использование

- конструирование

```
unique_ptr<C> cptr(new C(1,2,3)); // will be deallocated with delete (...)
unique_ptr<C[]> cptr(new C[10]); // will be deallocated with delete [](...)
```

- конструирование – специальный деаллокатор

```
void customDeleter(C* pc) { delete pc; }
//...
unique_ptr<C> cptr(new C(1,2,3), customDeleter); // will be deallocated with customDeleter()
```

- освобождение от собственности

```
C* pc = cptr.release();
```

- замена объекта в собственности – предыдущий уничтожается

```
unique_ptr<C> cptr(new C(1,2,3));
cptr.reset(new C); // called delete for C(1,2,3)
```

- получение указателя на объект

```
unique_ptr<C> cptr(new C(1,2,3));
C* pc = cptr.get(); // cptr still owns object
```

- операторы operator->(), operator*()

Какую проблему решают

```
1  struct C {
2    int* mI;
3    C(int* i) : mI(iptr) {}
4    ~C() { delete iptr; }
5  };
6  void f()
7  {
8    int* iptr = new int;
9    {
10     C c1(iptr);
11   } // iptr is deleted here
12
13   C c2(iptr); // iptr is used-after-free
14   // ...
15   // iptr is double-freed
16 }
```

std::shared_ptr

Определение

Умный указатель, который:

- разделяет владение объектом через его указатель.

Разрушение объекта происходит при разрушении последнего shared_ptr, указывающего на него.

Примерное представление shared_ptr:

```
1  class shared_ptr<T> {
2    ReferenceCount* rc;
3    T* objectPtr;
4  public:
5    shared_ptr(T* ptr) {
6      rc = refCountFor(ptr);
7      rc.addReference();
8      objectPtr = ptr;
9    }
10   ~shared_ptr {
11     rc.removeReference();
12     if(rc.refCount() == 0)
13       delete objectPtr;
14   }
15 }
```

Fixing problem

```
1  struct C {
2    shared_ptr<int> mI;
3    C(shared_ptr<int> iptr) : mI(iptr) {}
4    ~C() { } // shared_ptr gets destructed as a member of C
5  };
6  void f()
7  {
8    shared_ptr<int> iptr(new int); // refCount is 1
9    {
10     C c1(iptr); // refCount is 2
11   } // c1 is destructed here, so refCount is 1
12
13   C c2(iptr); // refCount is 2
14 } // c2 is destructed here, so refCount is 1
15 // iptr is destructed here, so refCount is 0, so new int is destructed
```

std::shared_ptr

Использование

- конструирование

```
shared_ptr<C> cptr(new C(1,2,3)); // will be deallocated with delete (...)
```

- конструирование – специальный деаллокатор

```
void customDeleter(C* pc) { delete pc; }
//...
shared_ptr<C> cptr(new C(1,2,3), customDeleter); // will be deallocated with customDeleter()
shared_ptr<C> cptr1(new C[10], std::array_deleter<C>()); // will be deallocated with delete[](...)
```

- замена объекта в собственности – предыдущий уничтожается

```
shared_ptr<C> cptr(new C(1,2,3));
cptr.reset(new C); // called delete for C(1,2,3)
```

- получение указателя на объект

```
shared_ptr<C> cptr(new C(1,2,3));
C* pc = cptr.get(); // cptr still owns object
```

- операторы operator->(), operator*()
- кто еще владеет?

```
cptr.unique(); // bool, true if sole owner
cptr.use_count(); // size_t, number of owners
```

std::shared_ptr

Полезное

- конструирование: одновременное выделение памяти для блока управления и объекта

```
shared_ptr<T> make_shared<T>(Args... args) // args is T construtor
    args
```

- std::enable_shared_from_this

```
struct C : public std::enable_shared_from_this<C> {
  shared_ptr<C> pointer_to_this() {
    return shared_from_this();
  }
}
```

- преобразование типа

Bad:

```
shared_ptr<Base> bptr(new Derived(1,2,3));
shared_ptr<Derived> dptr(bptr.get());
```

Good:

```
shared_ptr<const Base> bptr_const(new Derived(1,2,3));
shared_ptr<Base> bptr = const_pointer_cast<Base>(bptr);
shared_ptr<Derived> dptr = dynamic_pointer_cast<Derived>(bptr);
shared_ptr<Derived> dptr1 = static_pointer_cast<Derived>(bptr);
```

Какую проблему решают

```
1  struct Parent {
2    vector<shared_ptr<Child>> children;
3    ~Parent() {}
4  };
5  struct Child {
6    shared_ptr<Parent> mom;
7    shared_ptr<Parent> dad;
8    Child(const shared_ptr<Parent>& m, const shared_ptr<Parent>& d)
9      :mom(m), dad(d) {};
10 };
11 void f()
12 {
13   shared_ptr<Parent> Alice(new Parent("Alice"));
14   shared_ptr<Parent> Bob(new Parent("Bob"));
15   shared_ptr<Child> Eve(new Child("Eve", Alice, Bob));
16   Alice.children.push_back(Eve);
17   Bob.children.push_back(Eve);
18 } // objects will not be destructed
19 // Because they have cyclic dependency
20 // refCount(Alice) == 1
21 // refCount(Bob) == 1
22 // refCount(Eve) == 2
```

std::weak_ptr

Определение

Умный указатель, который:

- содержит "слабую" ссылку на объект, которым владеет shared_ptr.

Создание weak_ptr не увеличивает refCount.
Для использования объекта, на который указывает weak_ptr, необходимо его преобразование в shared_ptr.

```
1  class weak_ptr<T> {
2    ReferenceCount* rc;
3    T* objectPtr;
4  public:
5    weak_ptr(T* ptr) {
6      rc = refCountFor(ptr);
7      objectPtr = ptr;
8    }
9    shared_ptr<T> lock() {
10     if(rc.refCount() != 0)
11       return shared_ptr<T>(objectPtr);
12     return nullptr;
13   }
14 }
```

Fixing problem

```
1  struct Parent {
2    vector<weak_ptr<Child>> children;
3    ~Parent() {}
4  };
5  struct Child {
6    shared_ptr<Parent> mom;
7    shared_ptr<Parent> dad;
8    Child(const shared_ptr<Parent>& m, const shared_ptr<Parent>& d)
9      :mom(m), dad(d) {};
10 };
11 void f()
12 {
13   shared_ptr<Parent> Alice(new Parent("Alice"));
14   shared_ptr<Parent> Bob(new Parent("Bob"));
15   shared_ptr<Child> Eve(new Child("Eve", Alice, Bob));
16   Alice.children.push_back(weak_ptr(Eve));
17   Bob.children.push_back(weak_ptr(Eve));
18   shared_ptr<Child> alicesFirstChild(Alice.children[0].lock());
19 } // alicesFirstChild gets destructed: Eve's refCount = 1
20 // Eve gets destructed: Alice's refCount = 1, Bob's refCount = 1, Eve's
       refCount = 0, new Child("Eve") is destructed ()
21 // Bob gets destructed: Bob's refCount = 0, new Parent("Bob") is
       destructed
```

std::weak_ptr

Использование

- конструирование

```
1 shared_ptr<C> cptr(new C(1,2,3));
2 weak_ptr<C> weakCPtr(cptr);
```

- прекращение владения объектом

```
1 shared_ptr<C> cptr(new C(1,2,3));
2 weak_ptr<C> weakCPtr(cptr);
3 weakCPtr.reset();
```

- получение shared_ptr на объект

```
1 shared_ptr<C> cptr(new C(1,2,3));
2 weak_ptr<C> weakCPtr(cptr);
3 shared_ptr<C> cptr1 = weakCPtr.lock(); // will point to nullptr, if
      object is released
```

- существует ли объект?

```
1 weakCPtr.expired(); // bool, true object is already destructed
2 weakCPtr.use_count(); // size_t, number of shared_ptr owners
```