

## Лекция 3. "Умные" указатели

ИУ8

October 10, 2017

На текущей лекции рассмотрим:

- RAII
- "умные" указатели
- `scoped_ptr`
- `unique_ptr`
- `shared_ptr`

## Вечный кошмар разработчика C++

```
1 int foo() {  
2     Object obj = new Object();  
3     // use obj  
4     // ...  
5     // but forget to free memory.  
6 }
```

## Только цифры

На 08 октября 2017 года в проекте Chromium

- ключевое слово `delete` встречается 15,800 раз
- `unique_ptr` - 64,100
- `regex:\w+_ptr` - 66,100

## Утечка ресурсов

```
1 try {  
2     Image * img = new Image("~/Puss_in_Boots.png");  
3     // ...  
4     throw std::logic_error();  
5     delete img;  
6 }  
7 catch(...) {  
8 }
```

## RAII

Идиома объектно-ориентированного программирования, смысл которой заключается в том, что получение некоторого ресурса неразрывно совмещается с инициализацией объекта, а освобождение — с уничтожением.

## Использование RAII

```
1 template<class T>
2 struct ScopedPtr{
3     T * ptr;
4
5     ScopedPtr(T * p) {
6         ptr = p;
7     }
8
9     ~ScopedPtr() {
10         delete ptr;
11     }
12 };
13 //...
14 try {
15     ScopedPtr<Image> img(new Image("~/Puss_in_Boots.png"));
16     // ...
17     throw std::logic_error();
18 }
19 catch(...) {
20 }
```

## "Умные" указатели

**"Умные" указатели** - объект, работать с которым можно как с обычным указателем, но в отличие от последнего, он реализует идиому RAII и предоставляет некоторый дополнительный функционал.

В стандарте C++11 появились следующие "умные" указатели: **unique\_ptr**, **shared\_ptr** и **weak\_ptr**. Все они объявлены в заголовочном файле `<memory>`.

Библиотека **boost** предоставляет дополнительные 4 класса умных указателей – **boost::scoped\_ptr**, **boost::scoped\_array**, **boost::shared\_array** и **boost::intrusive\_ptr**.

## auto\_ptr

Еще один тип умного указателя, который достался в наследство от C++98, - это **auto\_ptr**. Если вы разрабатываете на C++11, то никогда и нигде **не используйте** **auto\_ptr**. На замену ему пришел более совершенный **unique\_ptr**.

## `boost::scoped_ptr`

Пожалуй, самый простой среди "умных" указателей. Является не копируемым и неперемещаемым объектом.

## Основные методы

- конструктор
- деструктор
- `operator *`
- `operator ->`
- `get`
- `reset`
- `swap`

## Пример

```
1 try {  
2     boost::scoped_ptr<Image> img(new Image("~/Puss_in_Boots.png"));  
3     // ...  
4     throw std::logic_error();  
5 }  
6 catch(...) {  
7 }
```



## `std::unique_ptr`

Использование `unique_ptr` позволяет не рассматривать вопросы освобождения ресурса, тем более не беспокоиться о том, что это уничтожение выполнялось в точности один раз.

Класс `std::unique_ptr` воплощает в себе семантику исключительного владения.

По умолчанию, `std::unique_ptr` имеет тот же размер, что и обычные указатели, и для большинства операций выполняются точно такие же команды. Это означает, что такие указатели можно использовать даже в ситуациях, когда важны расход памяти и времени.

`std::unique_ptr` всегда владеет тем, на что указывает. Перемещение `std::unique_ptr` передает владение от исходного объекта целевому. Копирование `std::unique_ptr` **запрещено**.

## Освобождение ресурса

При разрушении объекта ненулевой `std::unique_ptr` освобождает ресурс, которым владеет. По умолчанию освобождение ресурса выполняется через оператор `delete`. Но данное поведение можно настроить при создании `std::unique_ptr`.

## Custom deleters

При конструировании `std::unique_ptr` можно указать произвольную функцию (или функциональный объект, он же функтор), которая будет вызываться для освобождения ресурса.

Все функции пользовательских удалителей принимают обычный указатель на удаляемый объект и затем выполняют все необходимые действия по его удалению.

## Пример 1

```
1 auto pImg = std::make_unique<Image>("~/Puss_in_Boots.png");
```

## Пример 2

```
1 auto customDeleter = [](Image * p) {  
2     std::cout << "debug mode";  
3     delete p;  
4 };  
5 std::unique_ptr<Image, decltype(customDeleter)> img(nullptr,  
6     customDeleter);  
6 pImg.reset(new Image("~/Puss_in_Boots.png"));
```

## Основные методы

- конструктор
- деструктор
- operator=
- operator \*
- operator ->
- release
- reset
- swap
- get
- get\_deleter
- operator bool

## `std::shared_ptr`

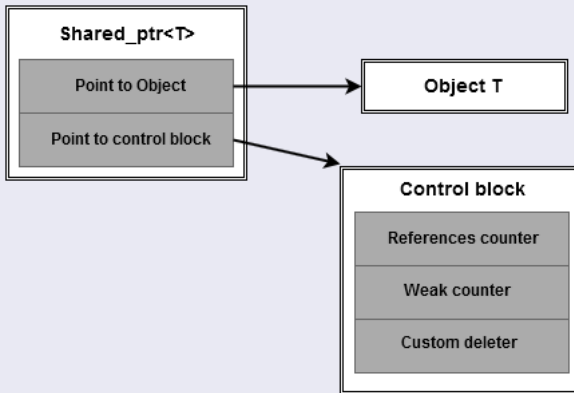
Указатель `std::shared_ptr` используется для управления ресурсами путем **совместного** впадения, т.е. объект, на который указывает `shared_ptr`, уничтожится только после того, как не останется ни одного `shared_ptr`, ссылающегося на него.

Никакой конкретный указатель `std::shared_ptr` не владеет объектом, на который указывает. Все указатели `std::shared_ptr` сотрудничают для гарантированного уничтожения целевого объекта только в той точке, где он станет более ненужным.

`std::shared_ptr` в два раза больше размера обычного указателя.

Класс `std::shared_ptr` имеет API, предназначенное только для работы с указателями на единичные объекты. Не существует `std::shared_ptr<T[]>`.

## Как работает shared\_ptr



Управляющий блок имеется для каждого объекта, управляемого указателями `std::shared_ptr`.

## Счетчик ссылок

Указатель `std::shared_ptr` может сообщить, является ли он последним указывающим на ресурс указателем с помощью **счетчика ссылок**. Он имеет значение, связанное с ресурсом, и отслеживает, какое количество указателей `std::shared_ptr` указывает на него.

Память для счетчика ссылок должна выделяться динамически. Счетчик ссылок связан с объектом, на который указывает `std::shared_ptr`, однако сам целевой объект об этом счетчике ничего не знает.

Конструкторы `std::shared_ptr` увеличивают этот счетчик (в случае конструктора перемещения это не так)  
Деструкторы `std::shared_ptr` уменьшают его  
Операторы копирующего присваивания делают и то, и другое (у копируемого объекта счетчик увеличивается, у получателя уменьшается)

## Производительность

Инкремент и декремент счетчика ссылок должны быть **атомарными**. Атомарные операции обычно медленнее неатомарных, так что несмотря на то, что обычно счетчики ссылок имеют размер в одно слово, следует рассматривать их чтение и запись как относительно дорогостоящие операции.

При передаче `std::shared_ptr` по значению происходит его копирование, и к его внутреннему счетчику прибавляется единица. Операция должна выполняться атомарно, что оказывает влияние на производительность. Передавайте `std::shared_ptr` по ссылке, где это возможно.



## Custom deleter

Подобно `std::unique_ptr`, `std::shared_ptr` в качестве механизма удаления ресурса по умолчанию использует `delete`

Для `std::shared_ptr` тип удалителя НЕ является частью типа интеллектуального указателя. Это делает `std::shared_ptr` более гибким указателем. Например, можно добавить два `std::shared_ptr` с разными деаллокаторами в один вектор, или присвоить значение одного другому.

Другим отличием от `std::unique_ptr` является то, что указание пользовательского удалителя не влияет на размер объекта `std::shared_ptr`. Независимо от удалителя объект `std::shared_ptr` имеет размер, равный размеру двух указателей.

Избегайте создания указателей из обычных встроенных указателей

Создание более одного `std::shared_ptr` из единственного обычного указателя приведет к **неопределенному поведению**.

## Пример

```
1 // BAD CASE
2 auto ptr = new Image("~/my_pic.png");
3
4 std::shared_ptr<Image> spw1(ptr);
5
6 std::shared_ptr<Image> spw2(ptr);
```

Объясните код.

## Осторожно!

Если два объекта ссылаются друг на друга с помощью `shared_ptr` и вы хотите освободить объекты и связанные с ними ресурсы при условии, что на них больше никто не ссылается, указатель `shared_ptr` не освободит данные, потому что счетчик ссылок будет равен 1.

## В общем случае отдавайте приоритет make-функции

- make-функции устраняют дублирование кода, повышают безопасность кода по отношению к исключениям и в случае функций `std::make_shared` генерируют меньший по размеру и более быстрый код;
- Ситуации, когда применение make-функций неприемлемо, включают необходимость указания пользовательских удалителей и необходимость передачи инициализаторов в фигурных скобках;
- Для указателей `std::shared_ptr` дополнительными ситуациями, в которых применение make-функций может быть неблагоприятным, являются классы с пользовательским управлением памятью, а также системы, в которых проблемы с объемом памяти накладываются на использование очень больших объектов и наличие указателей `std::weak_ptr`, время жизни которых существенно превышает время жизни указателей `std::shared_ptr`.

## Резюме по `shared_ptr`

- Требуется динамически выделенная память для хранения управляющего блока
- Операции, требующие работы со счетчиком ссылок, из-за своей атомарности дороги
- + Разыменование `std::shared_ptr` не является более дорогостоящим, чем разыменование обычного указателя
- + Операции, требующие работы со счетчиком ссылок, потокобезопасные
- + Автоматическое управление временем жизни динамически выделяемых ресурсов с совместным владением

## Резюме

Если вам достаточно исключительного владения, лучшим выбором является `std::unique_ptr`.

Его производительность близка к производительности обычных указателей, а преобразование в `std::shared_ptr` выполняется очень легко.

В большинстве случаев *применение `std::shared_ptr`* предпочтительнее, чем ручное управление временем жизни объекта с **совместным владением**.

## `std::weak_ptr`

Класс `std::weak_ptr` требует создания совместно используемого указателя. Как только последний совместно используемый указатель, владеющий объектом, потеряет владение, слабый указатель автоматически станет пустым.

Помимо конструктора по умолчанию и копирующего конструктора, класс `std::weak_ptr` содержит только конструктор, получающий аргумент типа `std::shared_ptr`.

Нельзя использовать операторы `*` и `->` для доступа к объектам, на который ссылается указатель `std::weak_ptr`.

Чтобы получить доступ, следует создать соответствующий `std::shared_ptr`.

## weak\_ptr to shared\_ptr

Существует два способа получить `std::shared_ptr` из `std::weak_ptr`

- Использовать функцию `std::weak_ptr::lock`, которая возвращает `std::shared_ptr` в случае, если `std::weak_ptr` не просрочен, иначе `nullptr` (в некоторых компиляторах будет сгенерировано исключение)
- Использовать конструктор `std::shared_ptr`, который принимает в качестве параметра `std::weak_ptr`. Если `std::weak_ptr` просрочен, генерируется исключение



## `weak_ptr::expired()`

Если `std::weak_ptr` ссылается на уничтоженный объект (счетчик ссылок у `shared_ptr` равен 0), такой указатель называют висячим или просроченным

Чтобы узнать, является ли висячим объект `std::weak_ptr`, можно использовать функцию `std::weak_ptr::expired()`.

```
1 using namespace std;
2 auto spw = make_shared<Widget>();
3 weak_ptr<Widget> wpw(spw);
4 cout << boolalpha << wpw.expired() << endl; // cout: false
5 cout << boolalpha << wpw.use_count() == 0 << endl; // cout: false
6 spw.reset();
7 cout << boolalpha << wpw.expired() << endl; // cout: true
8 cout << boolalpha << wpw.use_count() == 0 << endl; // cout: true
```

## Самостоятельное изучение

- `nothrow`
- `enable_shared_from_this<T>`

## Список литературы

- Мейерс. Эффективный и современный C++. 42 рекомендации по использованию C++ 11 и C++ 14
- Джосаттис. Стандартная библиотека C++. Справочное руководство
- <http://archive.kalnitsky.org/2011/11/02/smart-pointers-in-cpp11/>
- <https://herbsutter.com/2013/06/05/gotw-91-solution-smart-pointer-parameters/>