# Умные указатели

Зацепин Михаил

Московский физико-технический институт

Москва, 2018

# Содержание

# Чем нас не устраивают обычные указатели (raw pointers)

- Нельзя понять, является ли указатель владеющим
- Каким образом нужно уничтожить то, на что указывает указатель
- Нужно уничтожить ровно один раз
- Как понять, является ли указатель висячим, т.е. что он указывает на память, которая больше не хранит объект

# Умные указатели в C++

- `auto_ptr` (deprecated в C++11, removed в C++17)
- `unique_ptr`
- `shared_ptr`
- `weak_ptr`

# unique_ptr

```
template<class T, class Deleter> class unique_ptr;
```

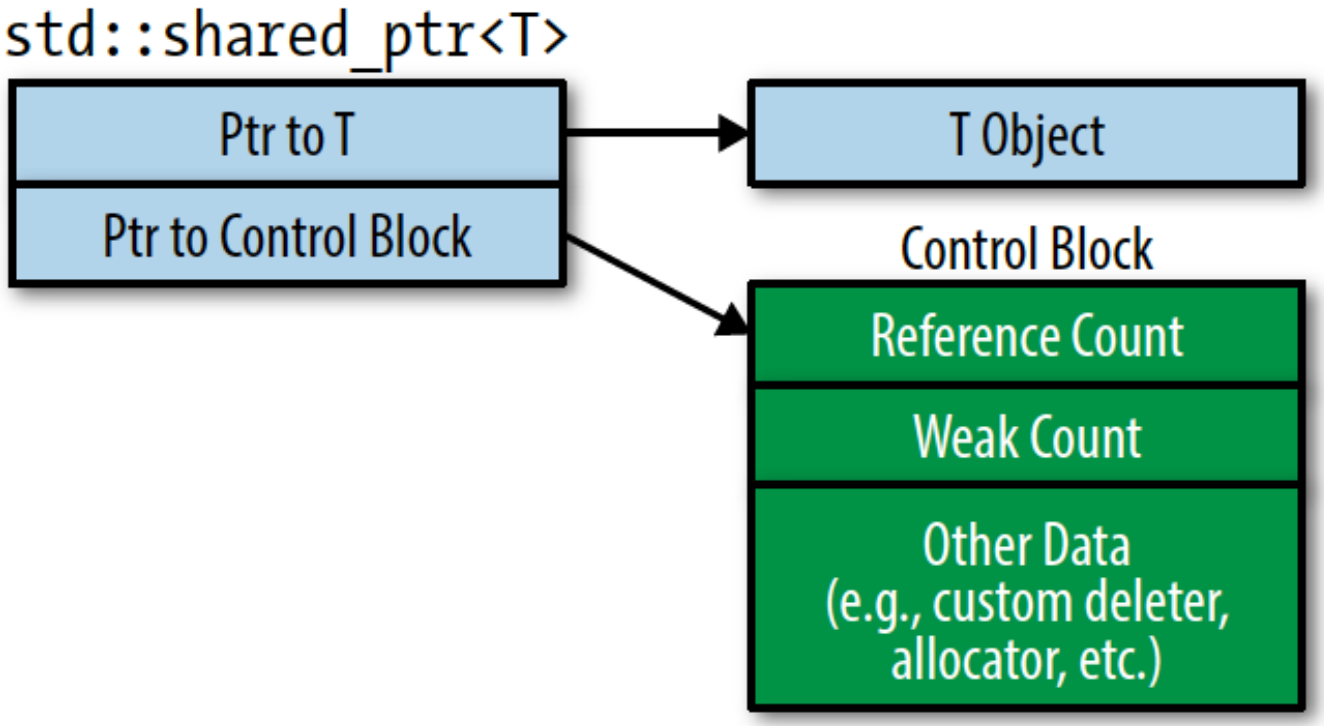
1. Владеет объектом, на который указывает.
2. Запрещает копирование, но разрешает перемещение (move).
3. При уничтожении указателя, уничтожается и объект, которым он владеет.
4. Позволяет возвращать из функции объекты, выделенные на динамической памяти.
5. Кроме того корректно уничтожает такие объекты при раскрутке стека во время проталкивания исключений.
6. Преобразуется в `shared_ptr`

# shared_ptr

```
template<class T> class shared_ptr;
```

1. Совместное владение.
2. Реализация - счетчик ссылок:
   1. Создание `shared_ptr` - увеличение счетчика ссылок на 1
   2. Разрушение `shared_ptr` - уменьшение счетчика ссылок на 1
   3. Счетчик становится равным 0 - разрушение объекта
3. `deleter` также присутствует, но не как часть типа, а как часть управляющего блока

# Управляющий блок shared_ptr



# Особенности использования shared_ptr

```
int* ptr = new int(10);
std::shared_ptr<int> smart_ptr1(ptr);
std::shared_ptr<int> smart_ptr2(ptr);
```

# Особенности использования shared_ptr

```
int* ptr = new int(10);
std::shared_ptr<int> smart_ptr1(ptr);
std::shared_ptr<int> smart_ptr2(ptr);
```

Неправильно: два счетчика ссылок

# Особенности использования shared_ptr

```
int* ptr = new int(10);
std::shared_ptr<int> smart_ptr1(ptr);
std::shared_ptr<int> smart_ptr2(ptr);
```

Неправильно: два счетчика ссылок

Корректно:

```
std::shared_ptr<int> smart_ptr1(new int(10));
std::shared_ptr<int> smart_ptr2(smart_ptr1);
```

# Производительность shared_ptr

1. Размер x2 по сравнению с raw-указателем
2. Динамическое выделение памяти под управляющий блок
3. Атомарный инкремент / декремент

# weak_ptr

```
template<class T> class weak_ptr;
```

1. Невладеющий "напарник" shared_ptr
2. Отслеживание висячих объектов
3. Отсутствует operator*() и операторы сравнения

# Использование weak_ptr

```
std::shared_ptr<int> shared_ptr(new int(10));

// Создание из weak_ptr или shared_ptr
std::weak_ptr<int> weak_ptr(shared_ptr);

// Получение доступа. Два способа:
std::shared_ptr<int> new_shared_ptr = weak_ptr.lock();
std::shared_ptr<int> new_shared_ptr_2(weak_ptr);
```

В чем разница между двумя способами получения доступа?

# Использование weak_ptr

```
std::shared_ptr<int> shared_ptr(new int(10));

// Создание из weak_ptr или shared_ptr
std::weak_ptr<int> weak_ptr(shared_ptr);

// Получение доступа. Два способа:
std::shared_ptr<int> new_shared_ptr = weak_ptr.lock();
std::shared_ptr<int> new_shared_ptr_2(weak_ptr);
```

В чем разница между двумя способами получения доступа?
Поведение в случае если weak_ptr висячий:

1. `lock` - вернет нулевой указатель
2. конструктор бросит исключение `bad_weak_ptr`

# Применение weak_ptr

1. Кеширование объектов
2. Решение проблемы циклических ссылок

# make_unique и make_shared vs new

1. Краткость кода
2. Повышение безопасности кода
3. Повышение производительности

# make_unique и make_shared vs new

1. Краткость кода
2. Повышение безопасности кода
3. Повышение производительности
4. Но не всегда

# Использование make_unique и make_shared

```
// Сравните
std::unique_ptr<Task> ptr(new Task);
auto ptr2(std::make_unique<Task>());

// А вдруг исключение?
processTask(std::unique_ptr<Task>(new Task),
            computeNewTaskPriority());
```

# Литература


1. Скотт Мейерс. Эффективный и современный C++
2. Скотт Мейерс. 55 способов ...