# Лекция 2

# Smart pointers

Created by drewxa@

# Содержание


идиома RAII
scoped_ptr
unique_ptr
shared_ptr

# Мотивация

Низкоуровневые языки программирования требуют ручного управления оперативной памятью.

Необходимо корректно выделять и освобождать.

# Leak memory

```
char* array = nullptr;
size_t sz = 1 * 1024 /*kb*/ *
              1024 /*Mb*/ *
              1024 /*Gb*/;
for (int i = 0; i < 5; ++i) {
  array = new char[sz];
  // использование array
  // и утечка памяти...
}
delete[] array;
```

# Leak memory

```
char* array = nullptr;
size_t sz = 1 * 1024 /*kb*/ *
              1024 /*Mb*/ *
              1024 /*Gb*/;
for (int i = 0; i < 5; ++i) {
  array = new char[sz];
  // использование array
  // и утечка памяти...
}
delete[] array;
```

Неправильное освобождение памяти

# Double memory free

```
Unit* warrior = new Knight();
Unit* ptr = warrior;
// использование указателей
delete warrior;
delete ptr;
```

# Double memory free

```
Unit* warrior = new Knight();
Unit* ptr = warrior;
// использование указателей
delete warrior;
delete ptr;
```

Повторное освобождение памяти

# Идиома RAII

*Resource Acquisition Is Initialization* - идиома объектно-ориентированного программирования, смысл которой заключается в том, что получение некоторого ресурса неразрывно совмещается с инициализацией объекта, а освобождение — с уничтожением.

# RAII

Другими словами, выделяем память (или любой другой ресурс) в конструкторе некого объекта, а освобождаем - в деструкторе.

# Пример RAII

```
template <class T>
struct ScopedPtr {
  T* ptr_;

  ScopedPtr(T* ptr) {
    ptr_ = ptr;
  }

  ~ScopedPtr() {
    delete ptr_;
  }
};


ScopedPtr<Unit> guard(new Knight);
```

# Пример RAII

```
template <class T>
struct ScopedPtr {
  T* ptr_;

  ScopedPtr(T* ptr) {
    ptr_ = ptr;
  }

  ~ScopedPtr() {
    delete ptr_;
  }
};


ScopedPtr<Unit> guard(new Knight);
```

# Smart pointers

- `boost::scoped_ptr`
- `std::unique_ptr`
- `std::shared_ptr`
- `std::weak_ptr`
- ~~`std::auto_ptr`~~ удален в C++17

# boost::scoped_ptr

Пожалуй, самый простой среди “умных” указателей.

Является **некопируемым** и **неперемещаемым** объектом.

# boost::scoped_ptr

Благодаря `scoped_ptr` разработчику не требуется вызывать delete.

```
// При разрушении объекта |guard| вызовется дестру
// класса boost::scoped_ptr, в котором
// и вызовется delete для класска Image.

auto* ptr = new Image("~/photo1.png");
boost::scoped_ptr<Image> guard(ptr);
// Нет явного вызова delete для |ptr|.
```

# std::unique_ptr

- единалично владеет объектом
- **некопируемый**, но **перемещаемый**
- минимум дополнительных затрат
- совместим с STL-контейнерами
- поддерживает custom deleter

## Единалично владение

`std::unique_ptr` - владеет обектом, на който указывает, т.е. отвечает за уничтожение обекта и освобождение памяти.

## Move only

`std::unique_ptr` - является некопируемый, но перемещаемым объектом.

Его нельзя копировать!

Но можно перемещать.

# Move only

При попытке копировать `std::unique_ptr` получим ошибку компиляции.

```
std::unique_ptr ptr(new Image("~/photo.png"));
// Вызов конструктора копирования.
std::unique_ptr another_ptr = ptr;
```

---

```
In function 'int main()':
10:36: error: use of deleted function 'std::unique
_Dp>::unique_ptr(const std::unique_ptr<_Tp, _Dp>&)
[with _Tp = int; _Dp = std::default_delete<int>]'

In file included from /usr/include/c++/4.9/memory:
/usr/include/c++/4.9/bits/unique_ptr.h:356:7: note
unique_ptr(const unique_ptr&) = delete;
```

# Move only

Но `std::unique_ptr` можно перемещать.

```
std::unique_ptr ptr(new Image("~/photo.png"));
// Вызов конструктора перемещения.
std::unique_ptr another_ptr = std::move(ptr);

// |ptr| никуда не указывает.
assert(ptr == nullptr);

// |another_ptr| владеет объектом Image.
assert(another_ptr != nullptr);
```

## Минимум дополнительных затрат

- по умолчанию, `std::unique_ptr` имеет тот же размер, что и обычные указатели
- для большинства операций выполняются точно такие же команды
- значит, что `std::unique_ptr` можно использовать, когда важны расход памяти и времени

# Совместим с STL-контейнерами

```
std::vector<std::unique_ptr> array;
array.push_back(new Image("~/photo.png"));
```

```
// А при использовании boost::scoped_ptr
// возникнет ошибка компиляции.
std::vector<boost::scoped_ptr> array;
array.push_back(new Image("~/photo.png"));
```

## Поддерживает custom deleter

- по умолчанию, `std::unique_ptr` освобождает ресурс через операетор `delete`
- но данное поведение можно настроить
- при создании `std::unique_ptr` можно указать произвольную функцию, которая будет вызываться для освобождения ресурса вместо `delete`

# `std::shared_ptr`

- совместное управление объектом
- **копируемый** и **перемещаемый**
- можно работать в условиях многопоточности

## Совместное управелние объектом

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

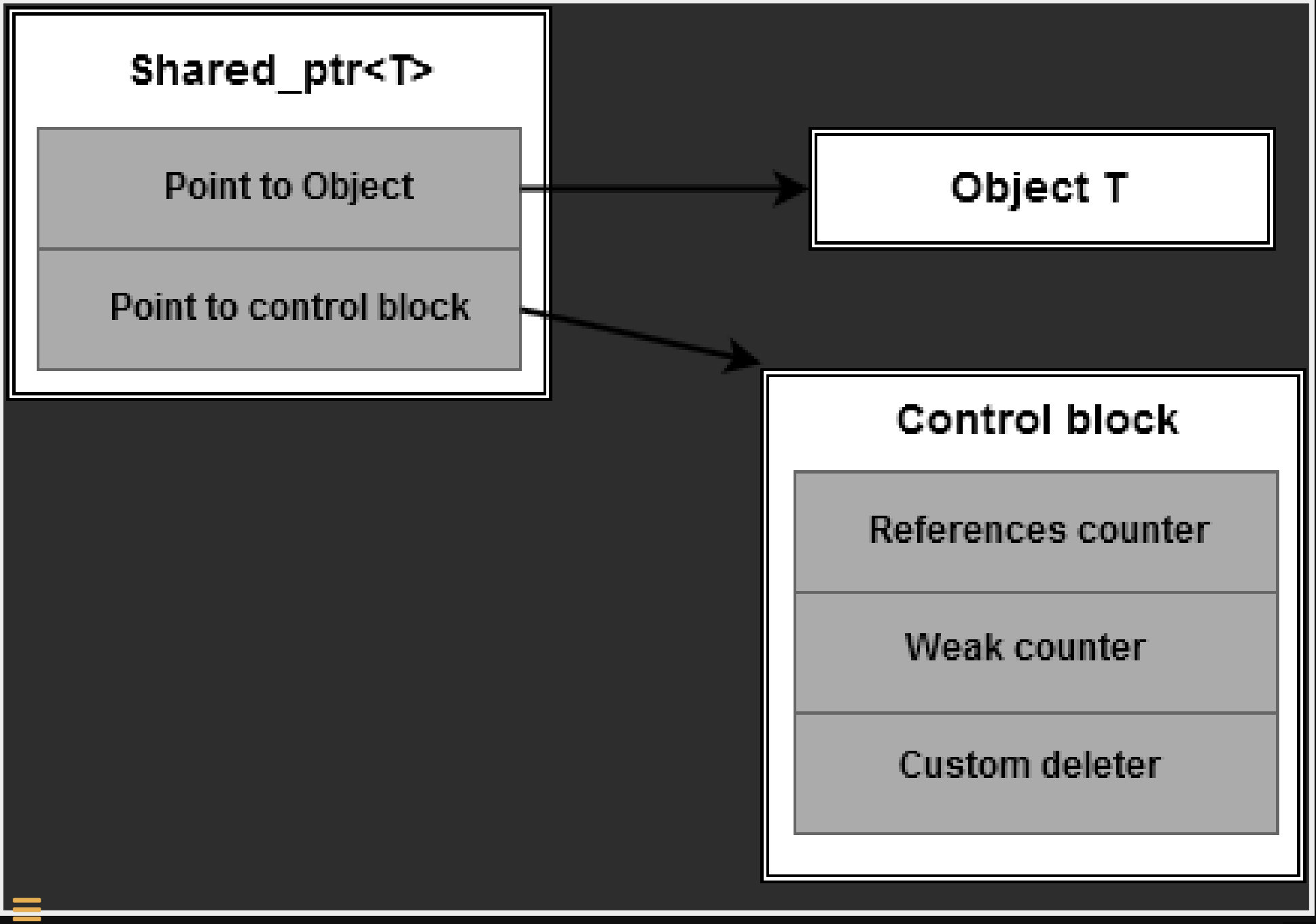
# Копируемый и перемещаемый

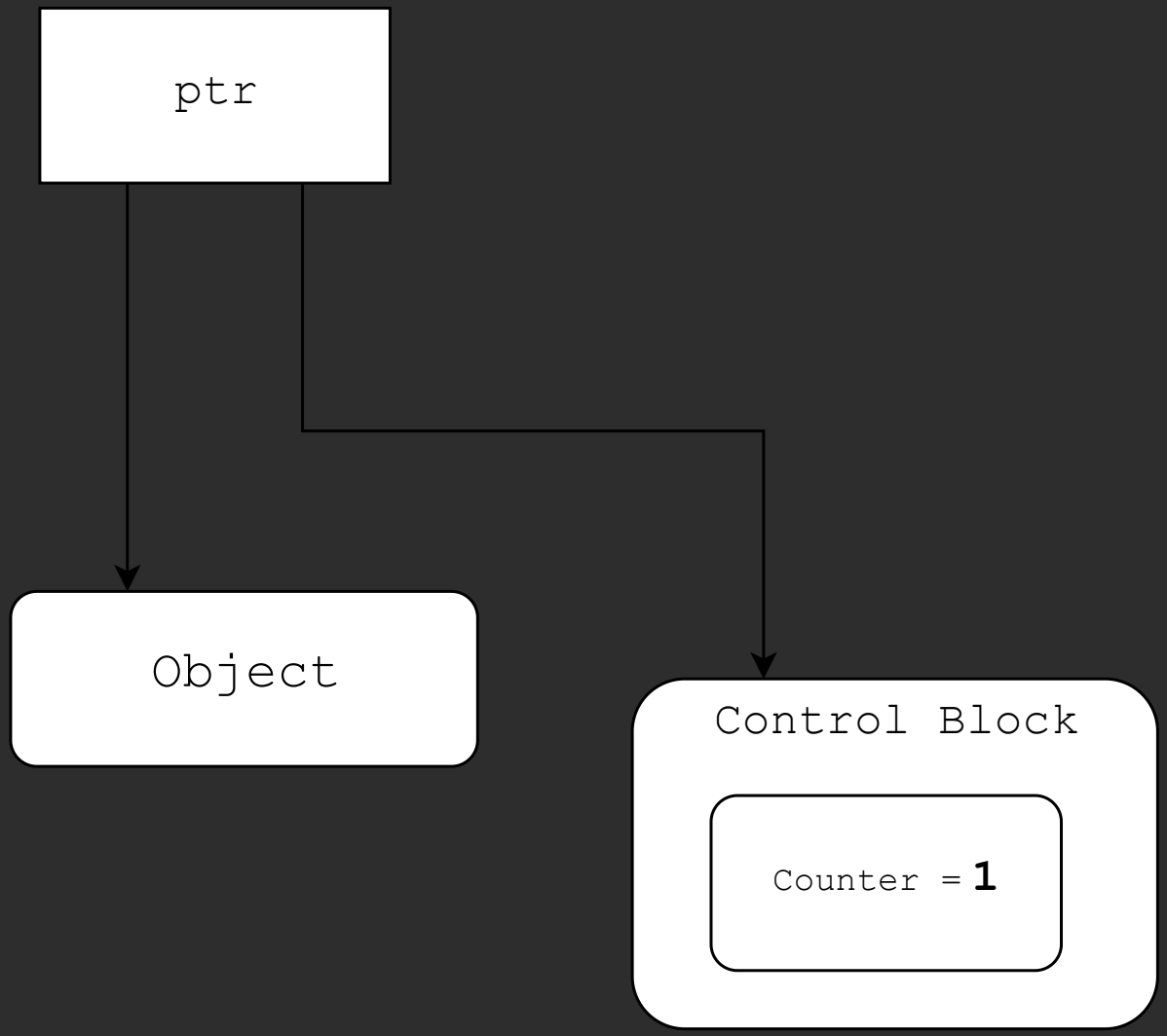
```
std::shared_ptr ptr(new Image("~/photo.png"));

// Copy:
std::shared_ptr another_ptr = ptr;
assert(ptr != nullptr);
assert(another_ptr != nullptr);

// Move:
std::shared_ptr yet_another_ptr = std::move(ptr);
assert(ptr == nullptr);
assert(yet_another_ptr != nullptr);
assert(another_ptr != nullptr);
```

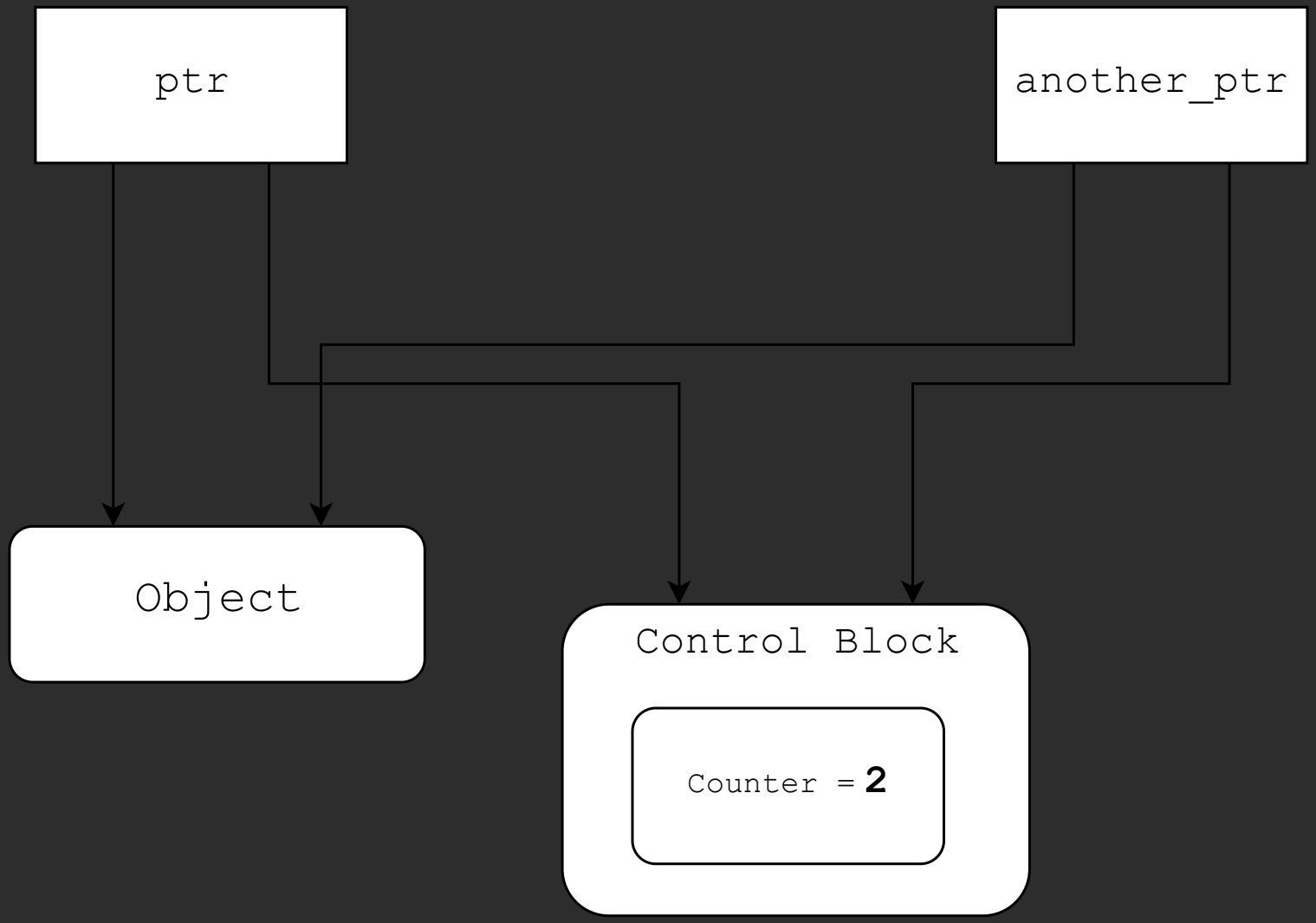
## Shared и многопоточность

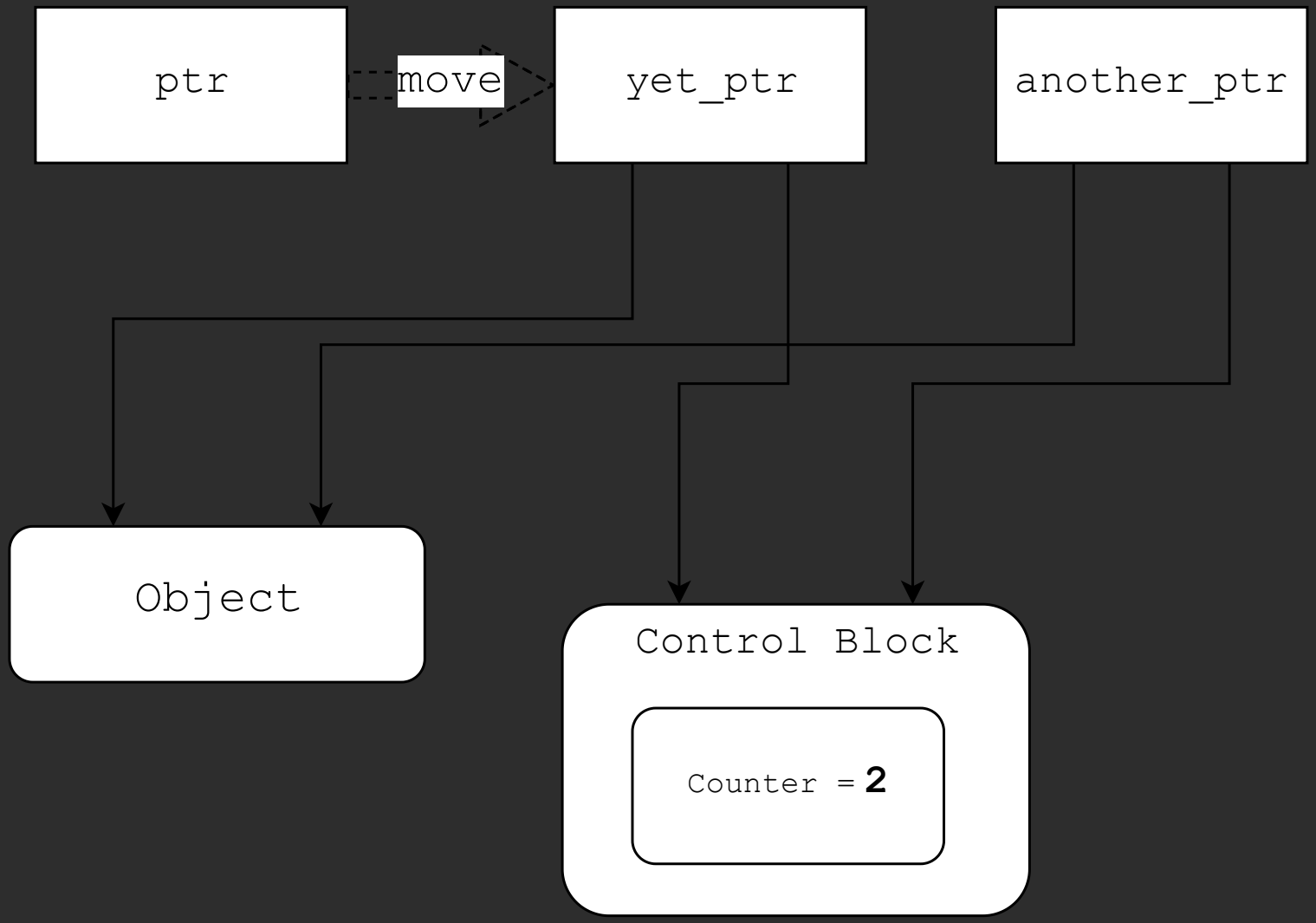
Подсчет ссылок в `shared_ptr` построен с использованием **атомарного** счетчика. Можно безопасно использовать указатели на один и тот же объект из разных потоков. Во всяком случае, не стоит беспокоиться о подсчете ссылок. Потокобезопасность самого объекта – другая проблема, и о ней надо заботится отдельно.

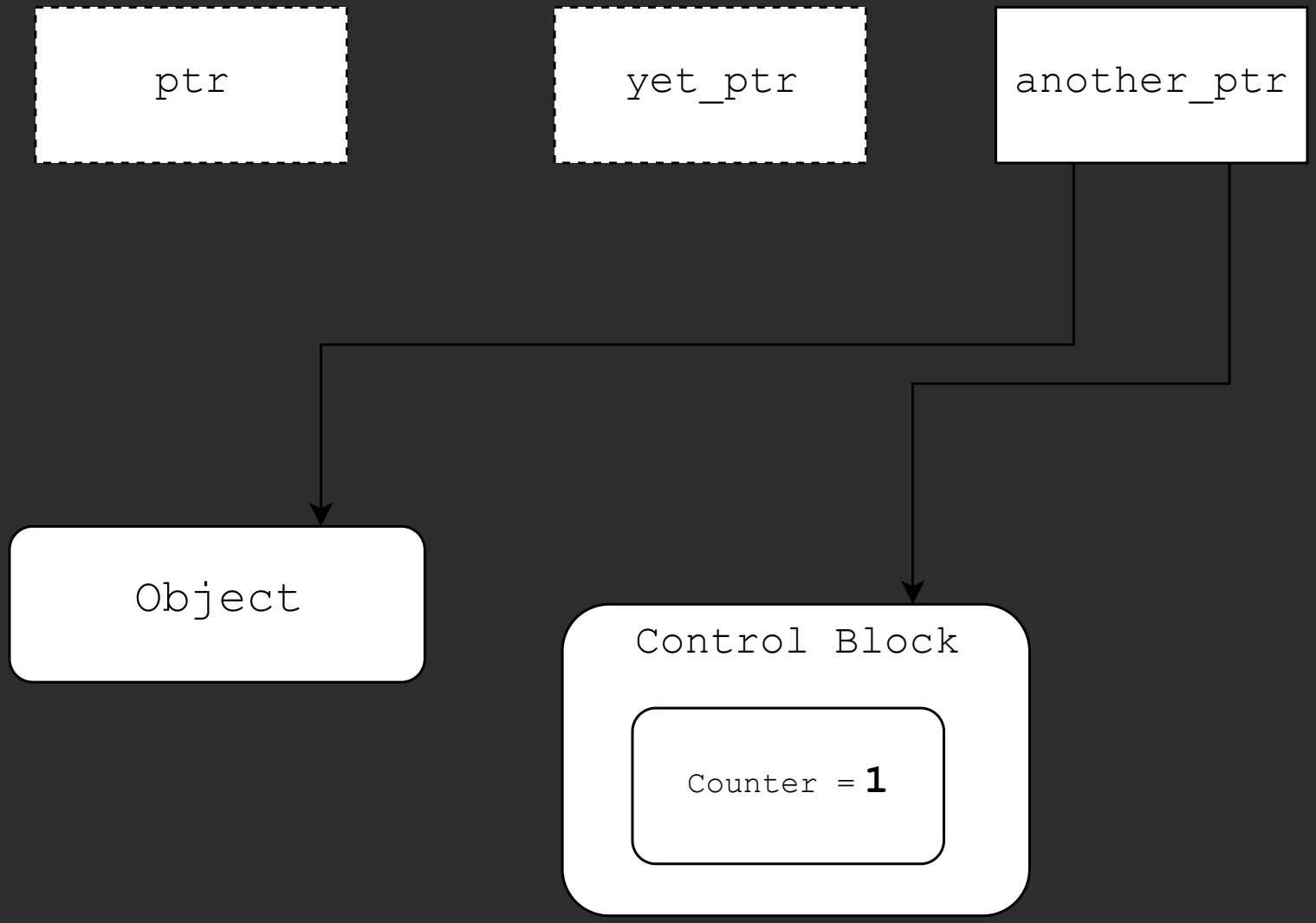
# Устройство `std::shared_ptr`

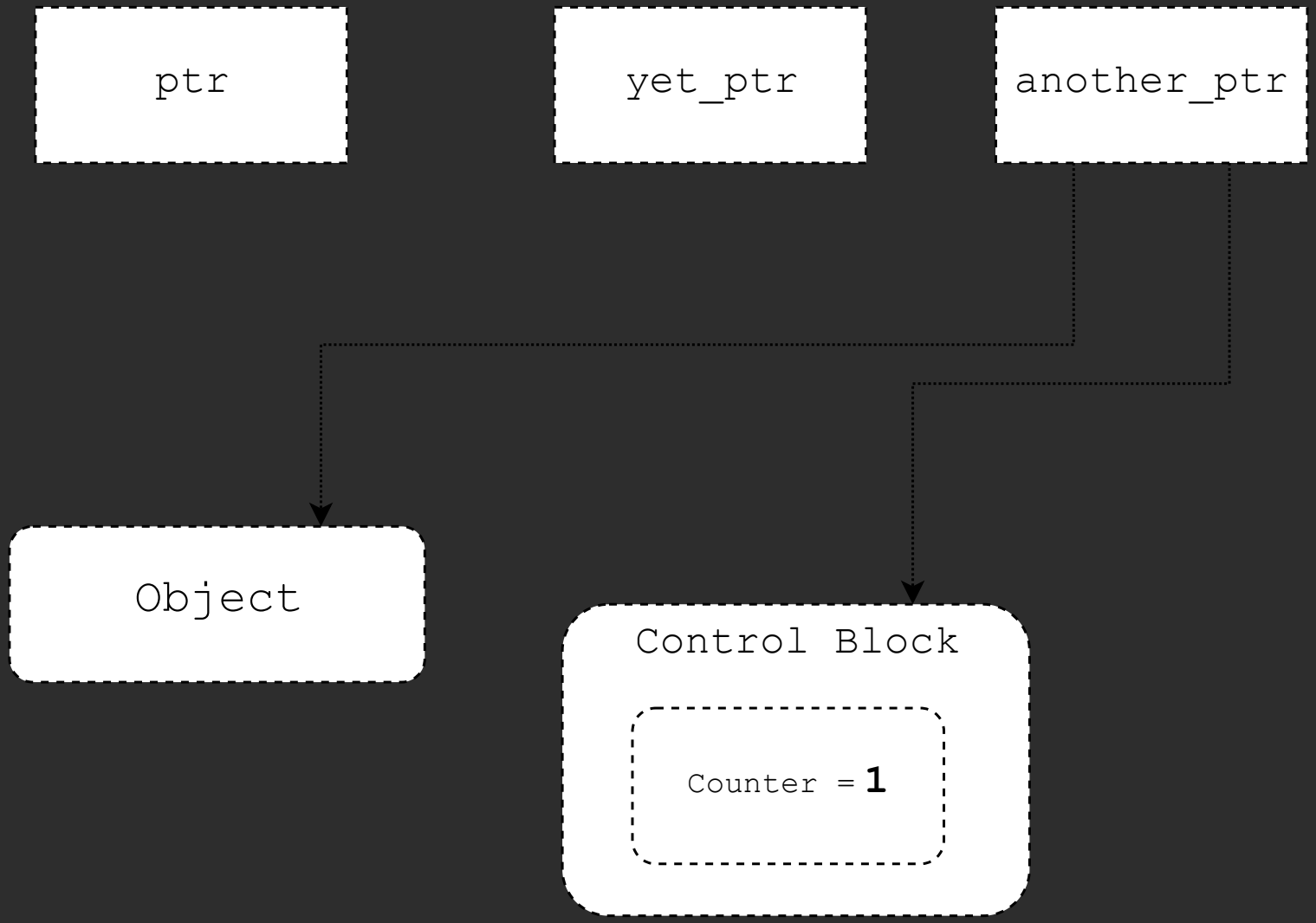

Shared_ptr<T>
Point to Object
Point to control block
Object T
Control block
References counter
Weak counter
Custom deleter

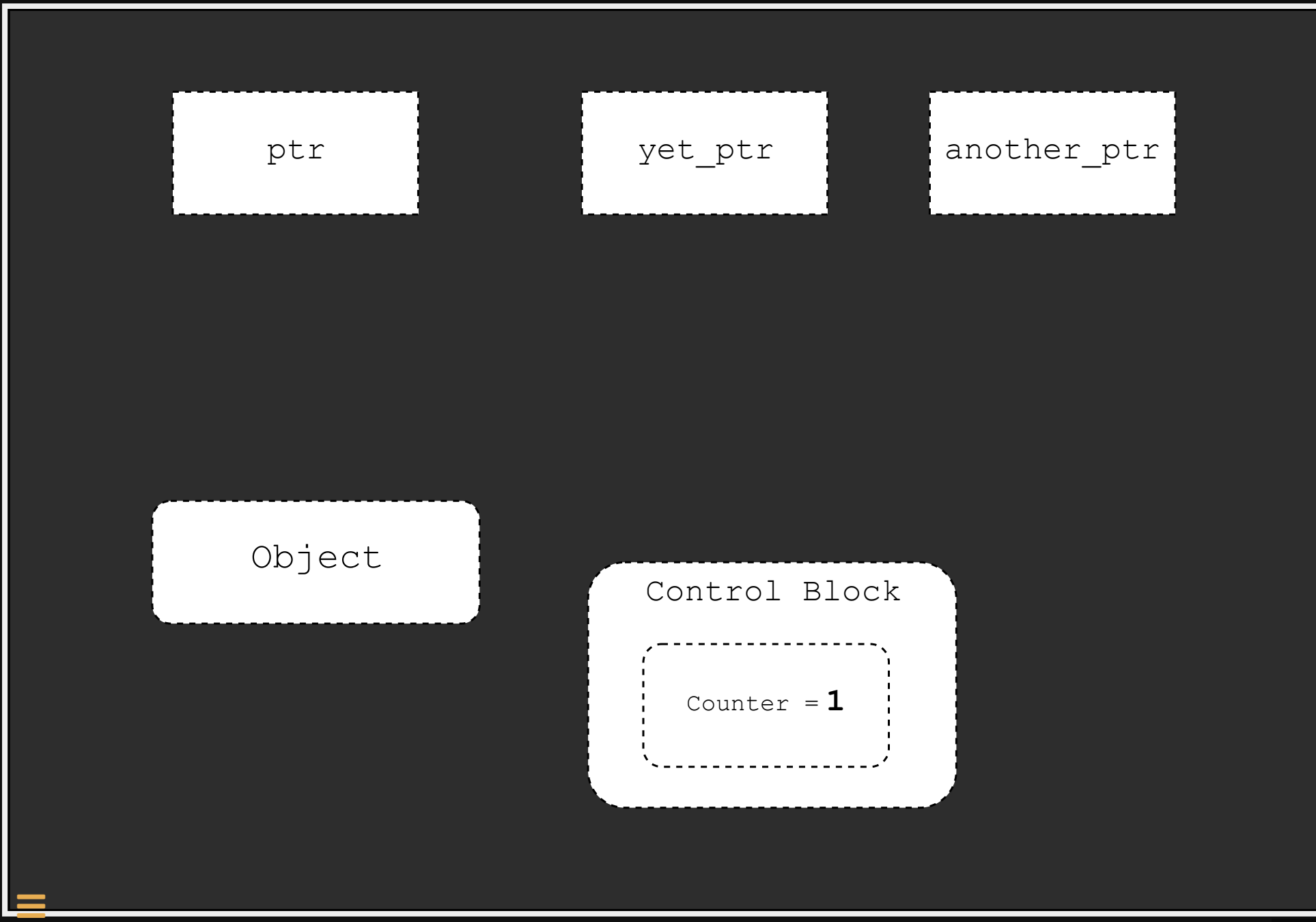

ptr
Object
Control Block
Counter = 1


ptr
another_ptr
Object
Control Block
Counter = 2


ptr
move
yet_ptr
another_ptr
Object
Control Block
Counter = 2


ptr
yet_ptr
another_ptr
Object
Control Block
Counter = 1


ptr
yet_ptr
another_ptr
Object
Control Block
Counter = 1


ptr
yet_ptr
another_ptr
Object
Control Block
Counter = 1

# Как прострелить себе ногу

## Двойное освобождение памяти

```
Image* raw = new Image("~/photo.png");

std::unique_ptr ptr = std::unique_ptr(raw);

std::unique_ptr yet_ptr = std::unique_ptr(raw);
```

# Как прострелить себе ногу

## ... тоже самое и с shared_ptr

```
Image* raw = new Image("~/photo.png");

std::shared_ptr ptr = std::shared_ptr(raw);

std::shared_ptr yet_ptr = std::shared_ptr(raw);
```

# Используйте make-функции

```
// Creating unique:
std::unique_ptr ptr =
    std::make_unique<Image>("~/photo.png");

// Creating shared:
std::shared_ptr another_ptr =
    std::make_shared<Image>("~/photo.png");
```

# Резюме

# Резюме

# Резюме

# Домашнее задание

- `std::weak_ptr`
- реализация `std::unique_ptr`
- shared_ptr и RAII