

# Программирование на современном C++

Семантика перемещения. Умные указатели.



# Отметьтесь на портале!

- Посещение необязательное, но тем, кто пришёл, следует отмечаться на портале в начале каждого занятия
- Это позволяет нам анализировать, какие занятия были более или менее интересны студентам, и менять курс в лучшую сторону
- Также это даст возможность вам оставить обратную связь по занятию после его завершения


пн, 1 ноября	вт, 2 ноября	ср, 3 ноября	чт, 4 ноября	пт, 5 ноября	сб, 6 ноября
Нет занятий	<div>18:00 Углубленный C/C++ (w... <span>п</span></div> <div>Обработка исключительных ситуаций. Шаблоны классов и методов. Обобщенное и безопасное программирование</div> <div>А. Халайджи</div> <div>18:00 Углубленный C/C++ (ML) <span>п</span></div> <div>Обработка исключительных ситуаций. Шаблоны классов и методов. Обобщенное и безопасное программирование</div> <div>А. Халайджи</div>	Нет занятий	Нет занятий	Нет занятий	Нет занятий

### Ссылка на Zoom РК сегодня в 18:00

Безопасность интернет-приложений (третий семестр)

Подключиться к конференции Zoom  
<https://maillru.zoom.us/j/98586081818?pwd=eWxwSDdCbIhQNnNwQU5iblFvU2dTZz09>

Идентификатор конференции: 985 8608 1818  
Код доступа: 785885

 Алексей Набережный 55 минут назад

★ 0 💬 0 ↓ 0 ↑

### Запись прошлой лекции (+ что почитать перед РК)

Безопасность интернет-приложений (третий семестр)

## Углублённый C/C++

Лекция 5

📍 Онлайн - ML

Отметьтесь, что вы пришли на занятие. Так вы улучшите свою посещаемость и вас увидит преподаватель в своём "Журнале посещений".

Отметиться

Оставьте отзыв о занятии и мы сможем улучшить учебный процесс.

Оставить отзыв

# План лекции

- Проблемы C++98
- Категории значений
- Семантика перемещения
- Перерыв
- `unique_ptr`
- `shared_ptr`
- Передача умных указателей

# Проблемы C++98

- Главной проблемой остаются утечки памяти

# Проблемы C++98

- Главной проблемой остаются утечки памяти
- Некопируемые объекты приходится передавать по указателю

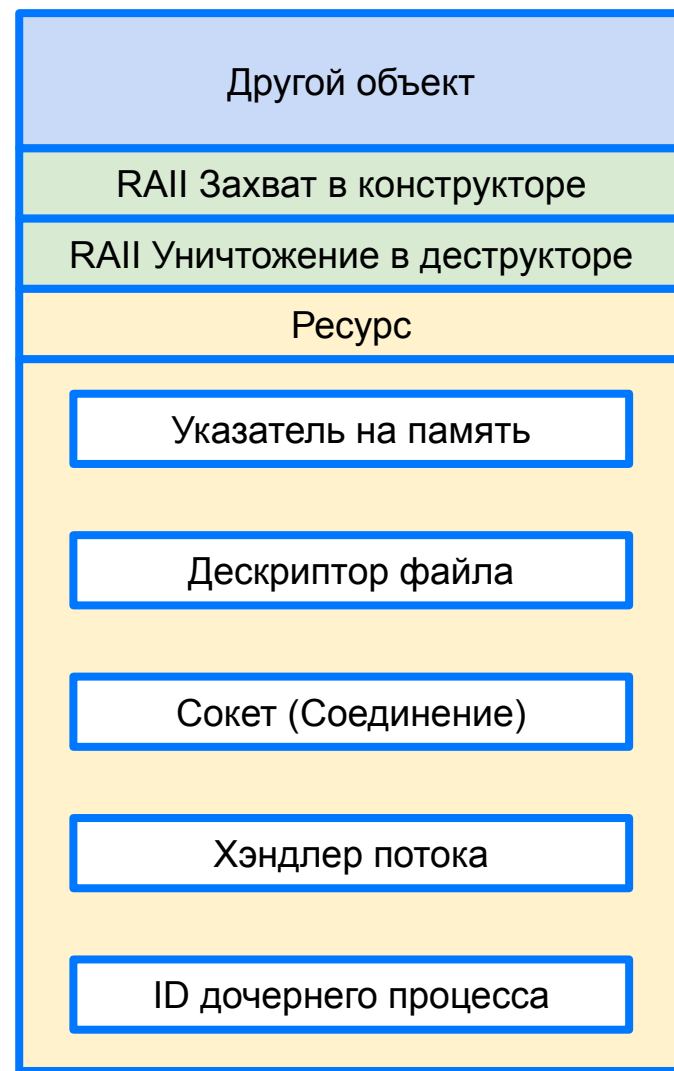
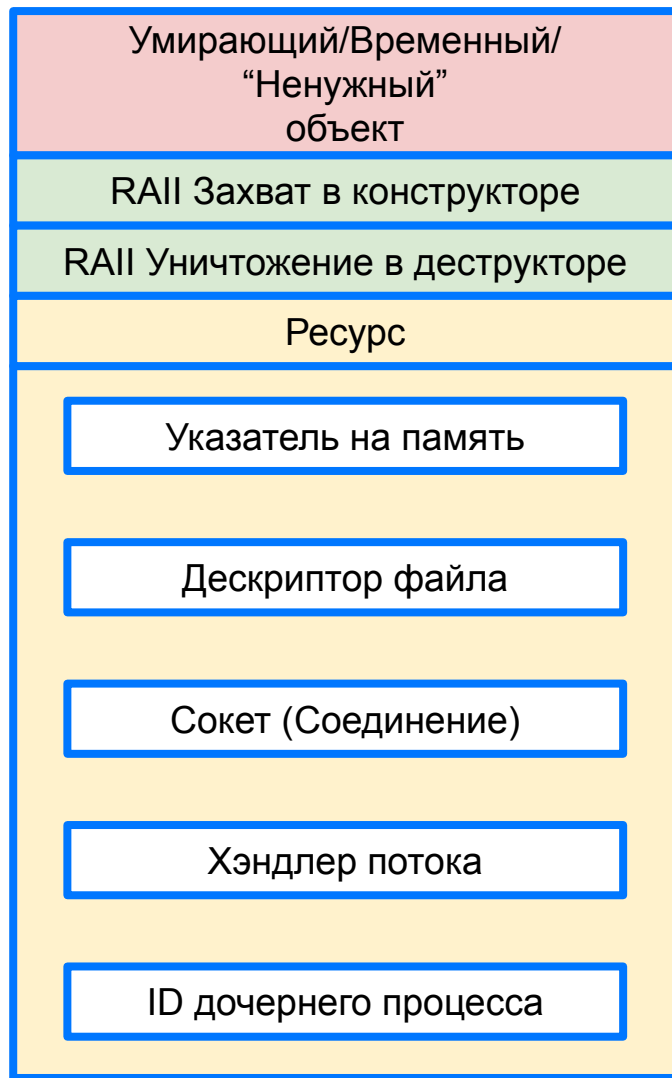
# Проблемы C++98

- Главной проблемой остаются утечки памяти
- Некопируемые объекты приходится передавать по указателю
- Полиморфные объекты приходится передавать по указателю

# Проблемы C++98

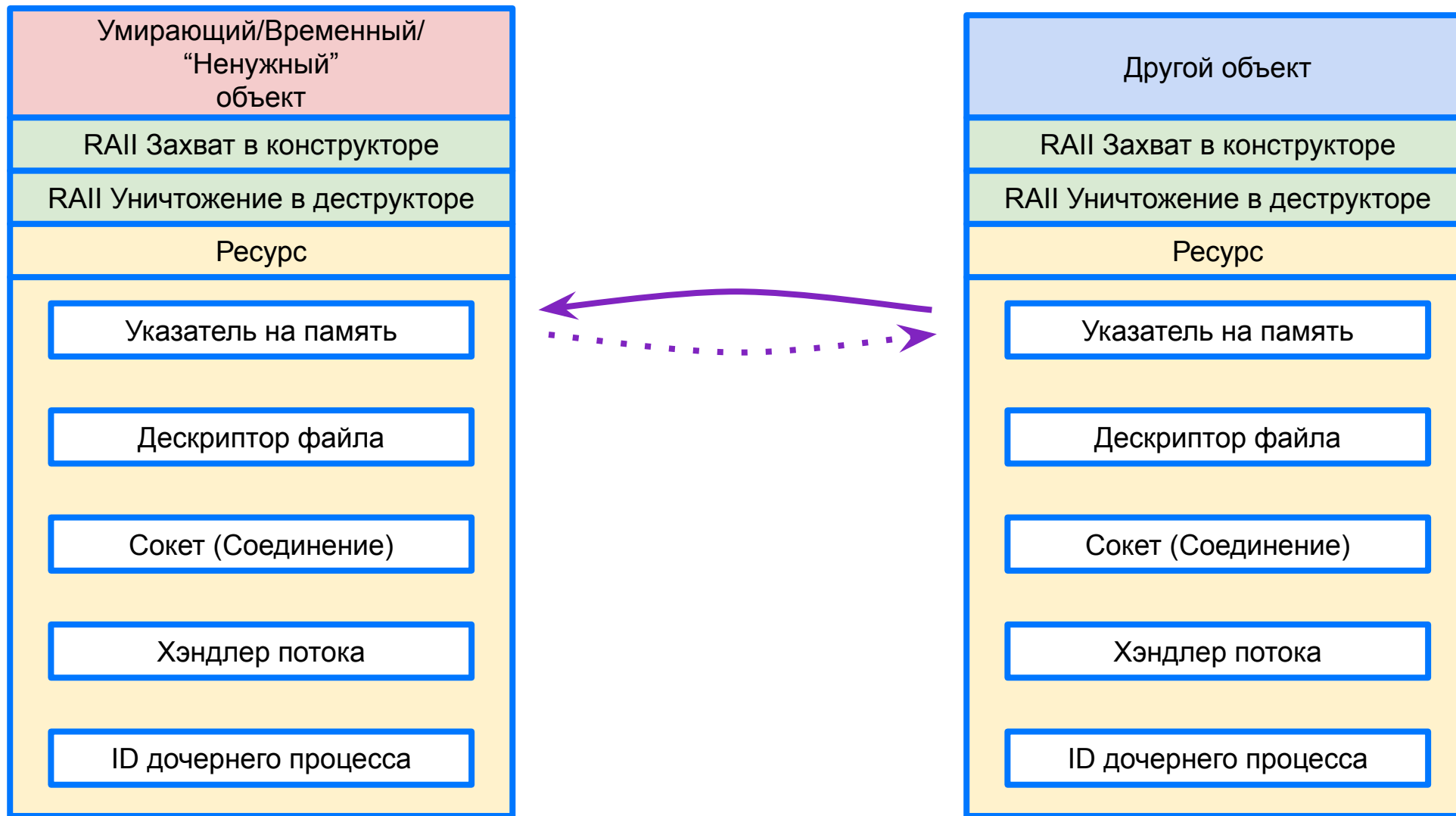
- Главной проблемой остаются утечки памяти
- Некопируемые объекты приходится передавать по указателю
- Полиморфные объекты приходится передавать по указателю
- Лишние копирования временных объектов

# Основная идея семантики перемещения

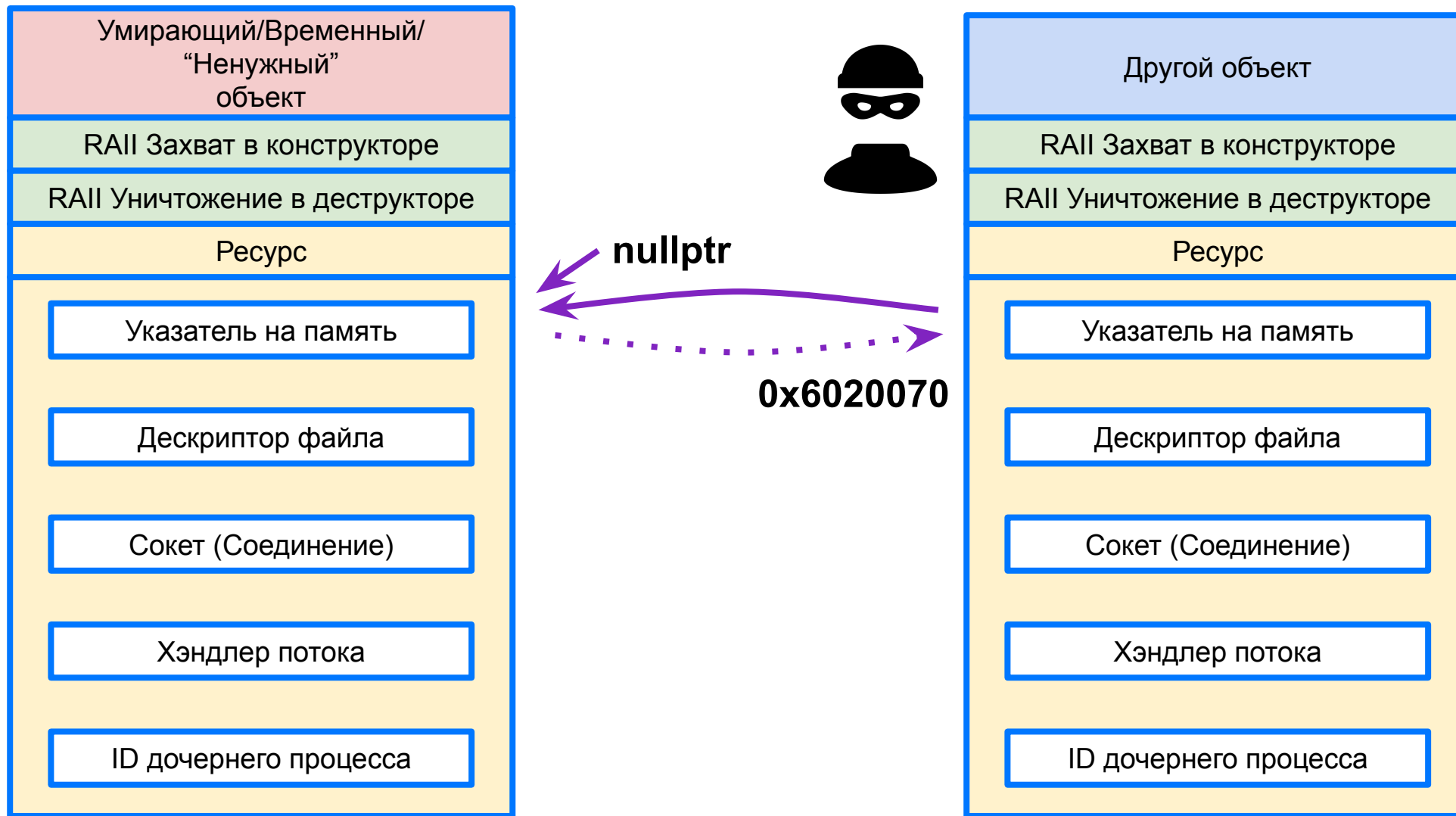




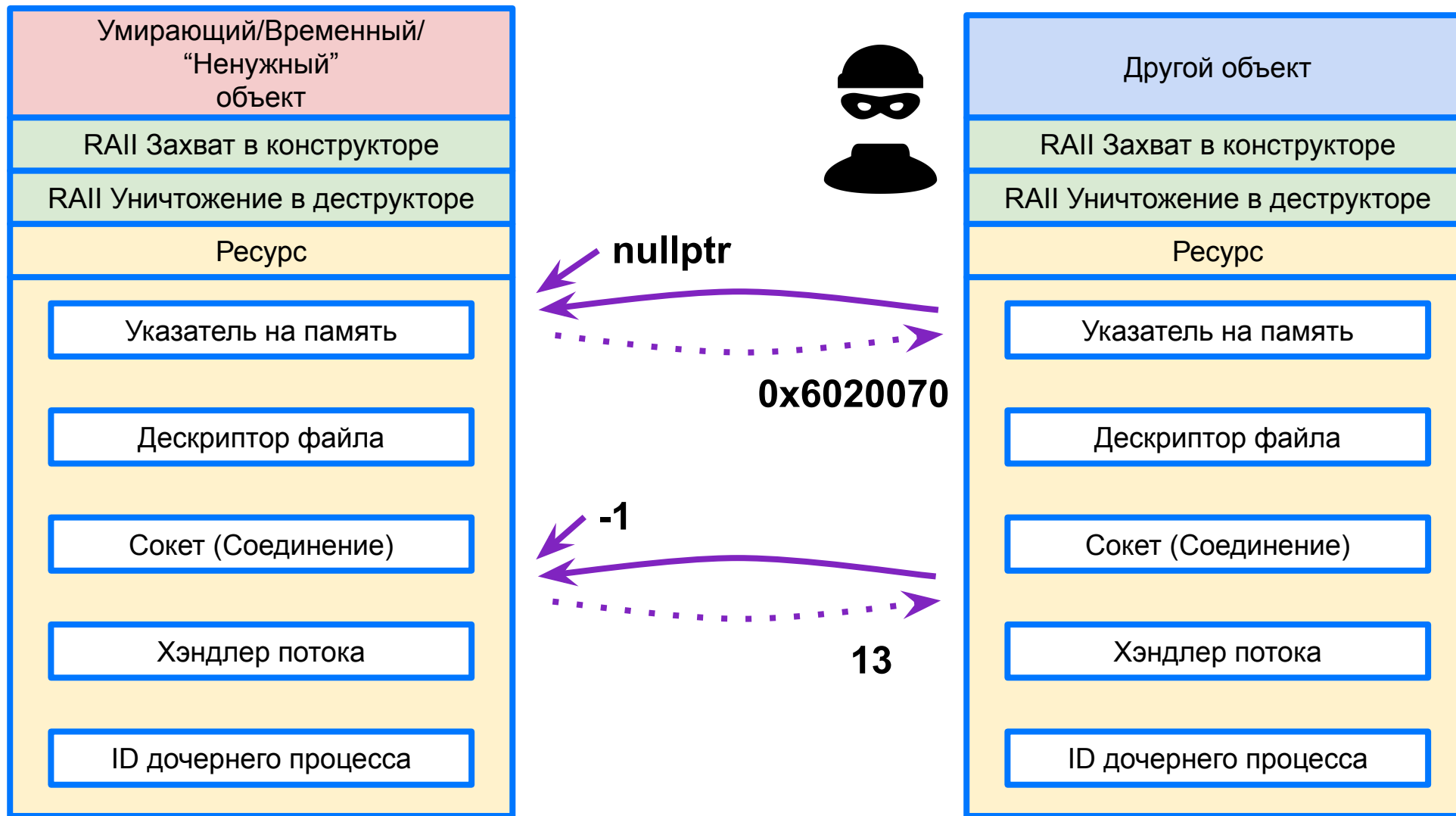
# Основная идея семантики перемещения



# Основная идея семантики перемещения



# Основная идея семантики перемещения



ЧТОБЫ ВСЕ РАБОТАЛО КОРРЕКТНО И УДОБНО, НЕОБХОДИМО НА УРОВНЕ ЯЗЫКА  
ОБОЗНАЧИТЬ, КАКИЕ ТИПЫ ОБЪЕКТОВ ЯВЛЯЮТСЯ ВРЕМЕННЫМИ/УМИРАЮЩИМИ

# Категории значений C++98

```
int foo();
```

```
int g;
```

← lvalue (имеет адрес)

```
g = foo();
```

```
12 = g;
```

```
foo() = g;
```

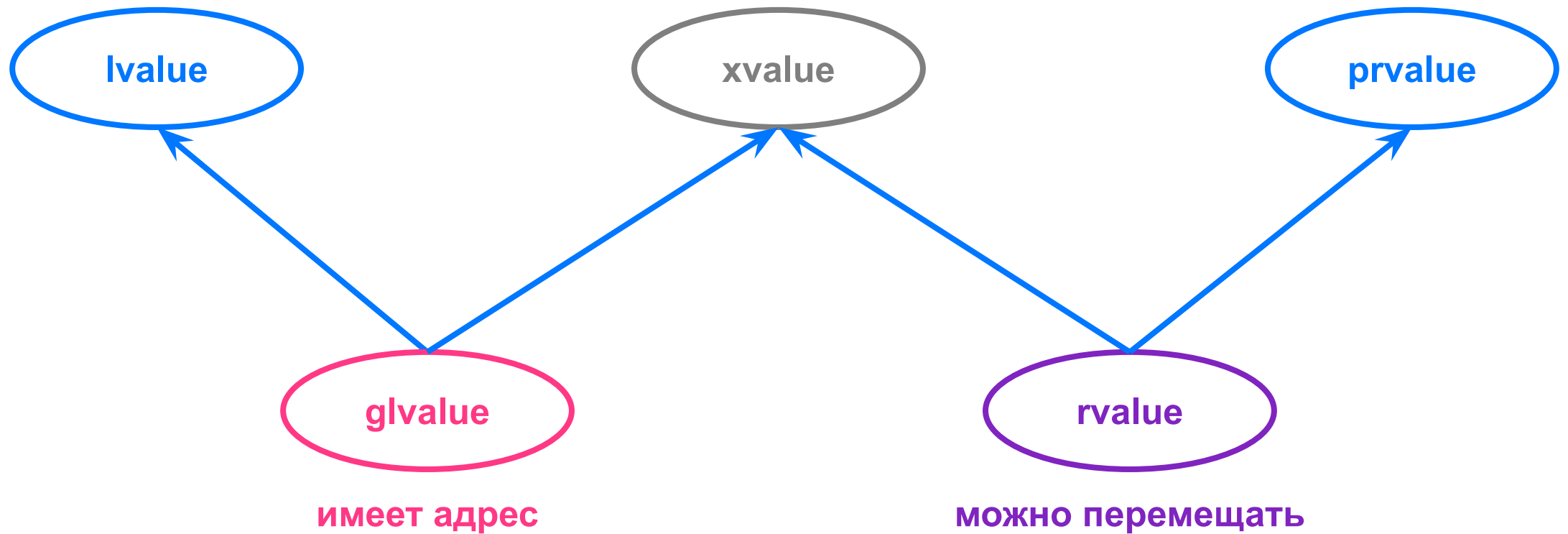
```
foo() = 0;
```

← rvalue

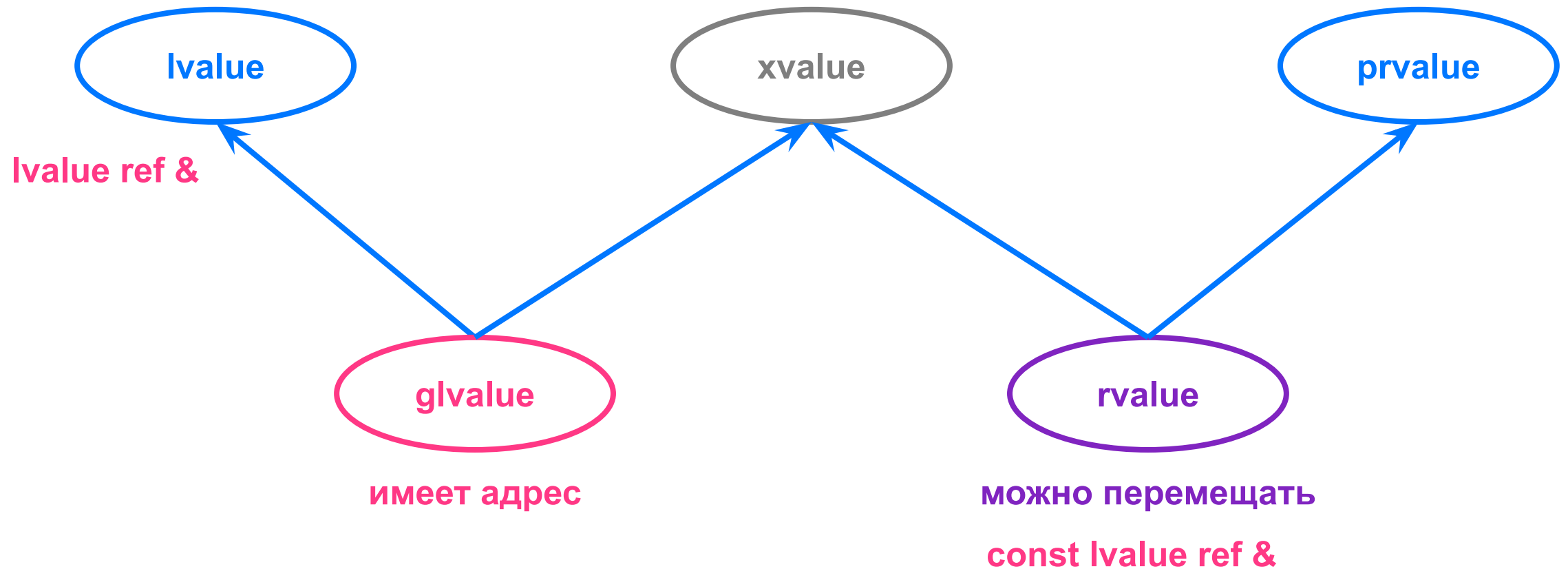
```
int& i = foo();
```

```
const int& i = foo();
```

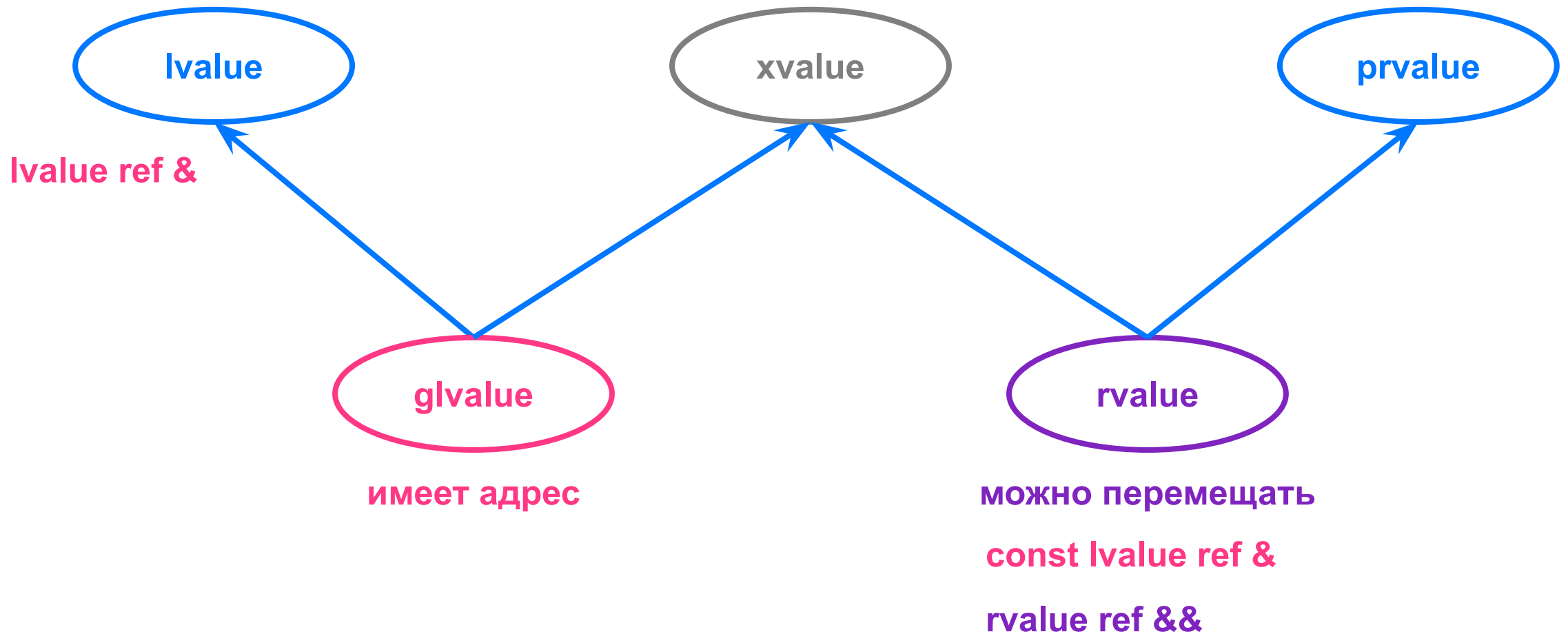
# Категории значений C++17



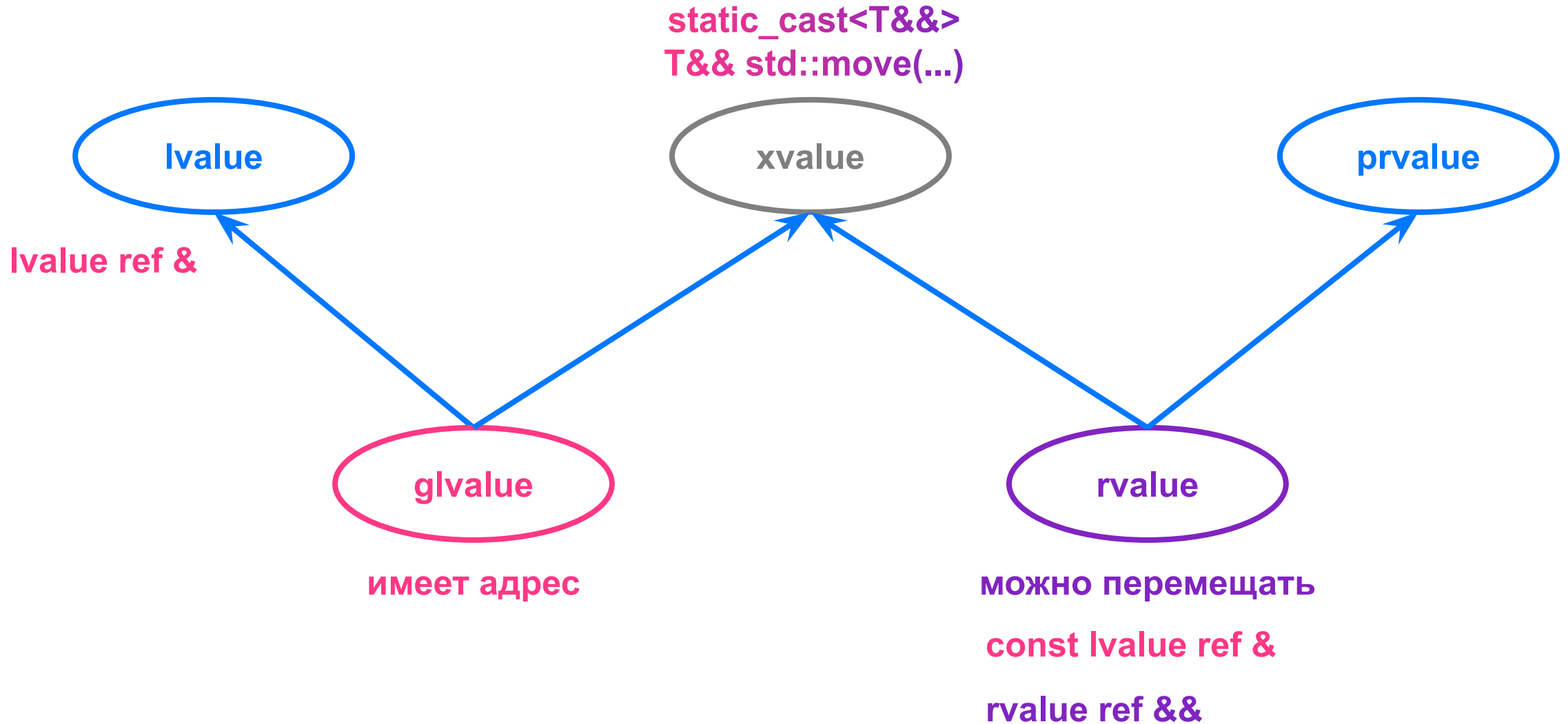
# Категории значений C++17



# Категории значений C++17



# Категории значений C++17





# Семантика перемещения

ЯВНЫЙ КАСТ К RVALUE

Можно отнимать ресурсы у type - объект умирает или больше не нужен.

```
class Type {  
    Type(Type && type) noexcept;  
    Type& operator=(Type && type) noexcept;  
  
    void set_name(std::string && name);  
};  
  
void some_function(Type && type);
```

# Семантика перемещения

rvalue reference заставляет клиента передавать временные объекты или делать `std::move` существующих. После передачи объект потеряет ресурсы.

```
void some_function(Type && type);
```

```
some_function(Object{10, 20, "Hello"});
```

```
some_function(GetNewObject());
```

```
Type type;
```

```
type.setData(...);
```

```
some_function(std::move(type));
```



ТРЕБОВАНИЕ КОМПИЛЯТОРА ДЛЯ СУЩЕСТВУЮЩИХ ОБЪЕКТОВ, для LVALUE

# Пример



# Умные указатели

# Умные указатели

RAII для указателей?

# std::unique\_ptr

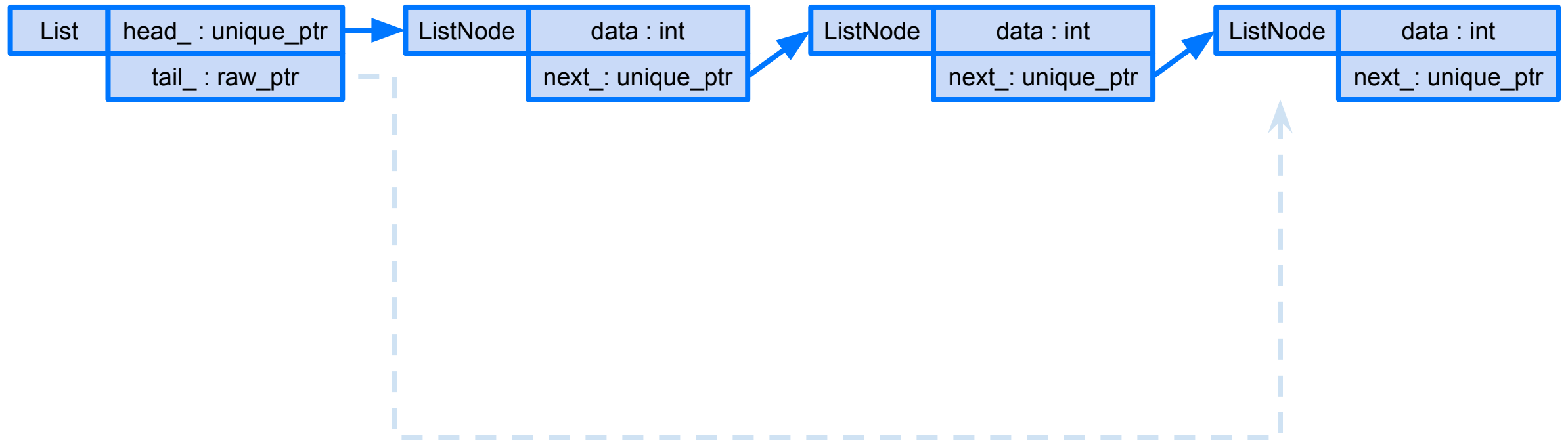
Единолично владеющая RAII обертка над указателем.

```
class std::unique_ptr {  
public:  
    unique_ptr(T* t) : ptr_{t} {} delete  запрещаем копирование: чтобы не делать double  
    unique_ptr(const unique_ptr&) = delete;  теперь пользуемся конструктором перемещения: забираем T* ptr у обертки,  
    unique_ptr(unique_ptr&& other) { ptr_ = exchange(other.ptr_, nullptr); }  
    ~unique_ptr() { delete ptr_; } ГЛАВНОЕ: ОСВОБОЖДАЕМ РЕСУРСЫ В  
    T& operator*() { return *ptr_; } ДЕСТРУКТОРЕ  
    T* operator->() { return ptr_; }  
    explicit operator bool() { return ptr_ != nullptr; }  
private:  
    T* ptr_;  
}
```

# std::unique\_ptr

Пример

# list std::unique\_ptr





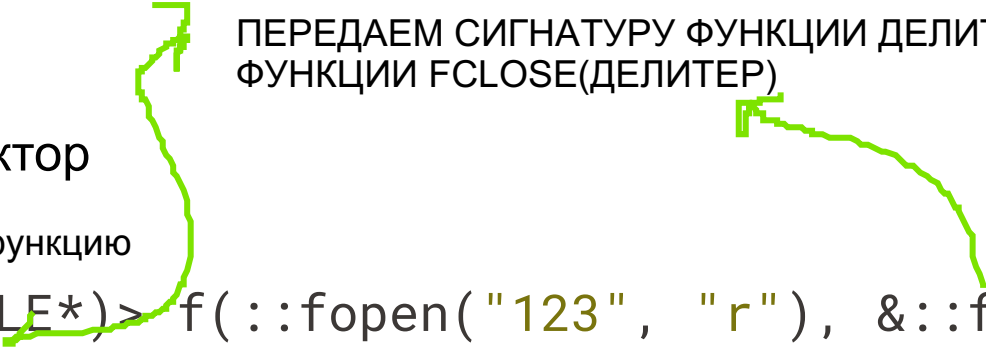
# std::unique\_ptr

Дополнительные моменты

У unique\_ptr можно указать свой деструктор

указатель на функцию

```
std::unique_ptr<FILE, int(*)(FILE*)> f(::fopen("123", "r"), &::fclose);  
std::unique_ptr<sqlite3, decltype(&sqlite3_close)> f(sq3_bd, &sqlite3_close);  
std::unique_ptr<zip_t, decltype(&zip_close)> f(zip_open(...), &zip_close);
```



ПЕРЕДАЕМ СИГНАТУРУ ФУНКЦИИ ДЕЛИТЕРА, ЭТО СИГНАТУРА ФУНКЦИИ FCLOSE(ДЕЛИТЕР)

# std::unique\_ptr

Дополнительные моменты

Умные указатели обычно инициализируются через функции make\_???

Для shared\_ptr std::make\_shared иногда эффективнее.

Для unique\_ptr std::make\_unique **была** безопаснее.

Традиционно все равно используются

```
auto file_uptr = std::make_unique<File>("file.txt");  
some_function(std::unique_ptr<MyClass>(new MyClass(param)), func_throws());  
some_function(std::make_unique<MyClass>(param), func_throws());  
auto file_sptr = std::make_shared<File>("file.txt");
```

# std::shared\_ptr

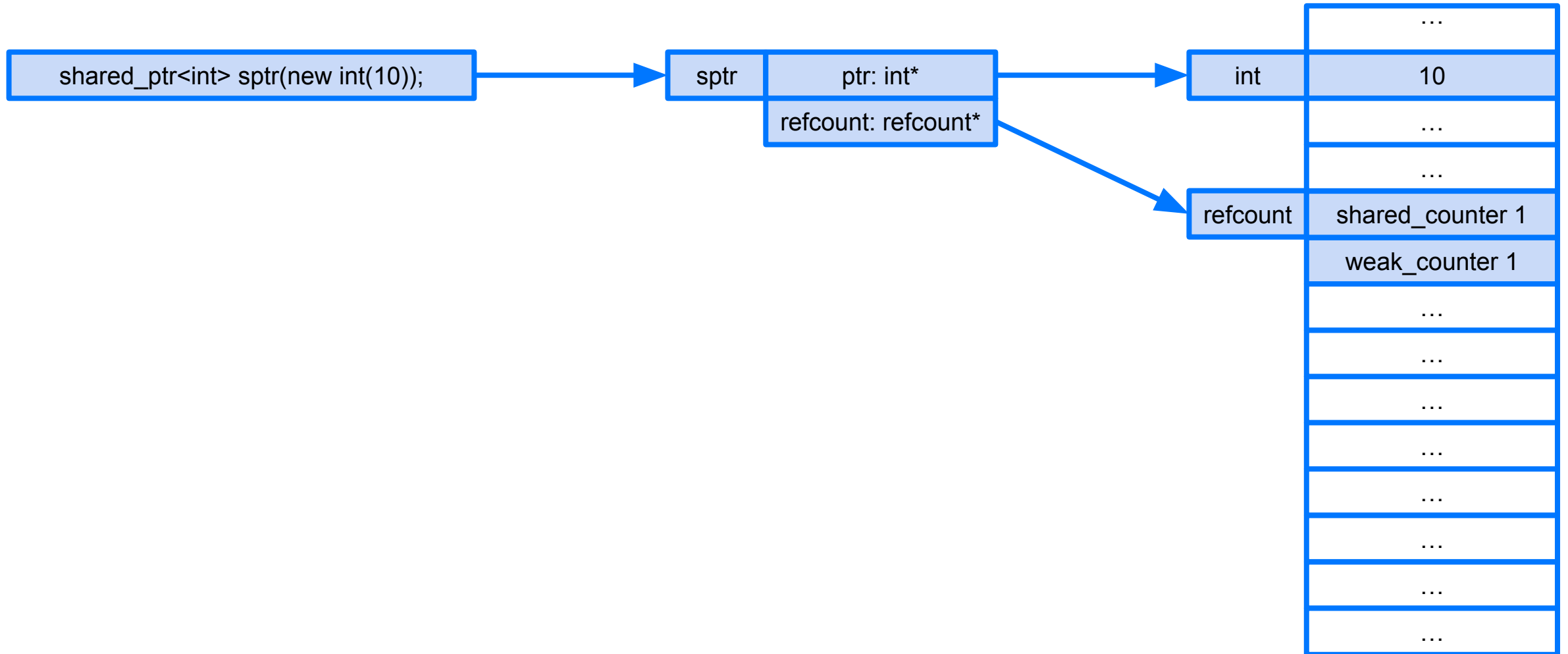
Разделяющая владение RAII обертка над указателем с подсчетом ссылок.

# std::shared\_ptr

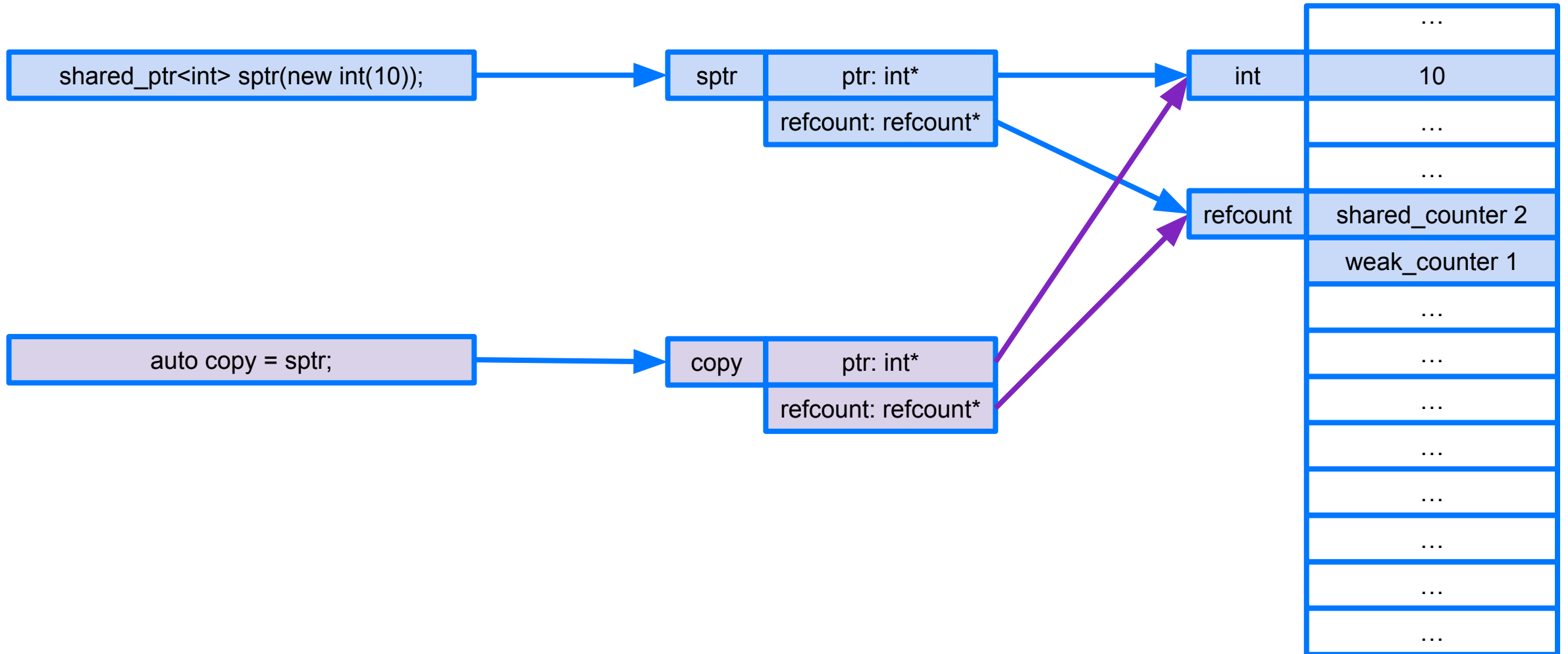
Разделяющая владение RAII обертка над указателем с подсчетом ссылок.

Для каждого объекта считается число shared\_ptr, которые на него ссылаются, когда число доходит до 0 - объект уничтожается

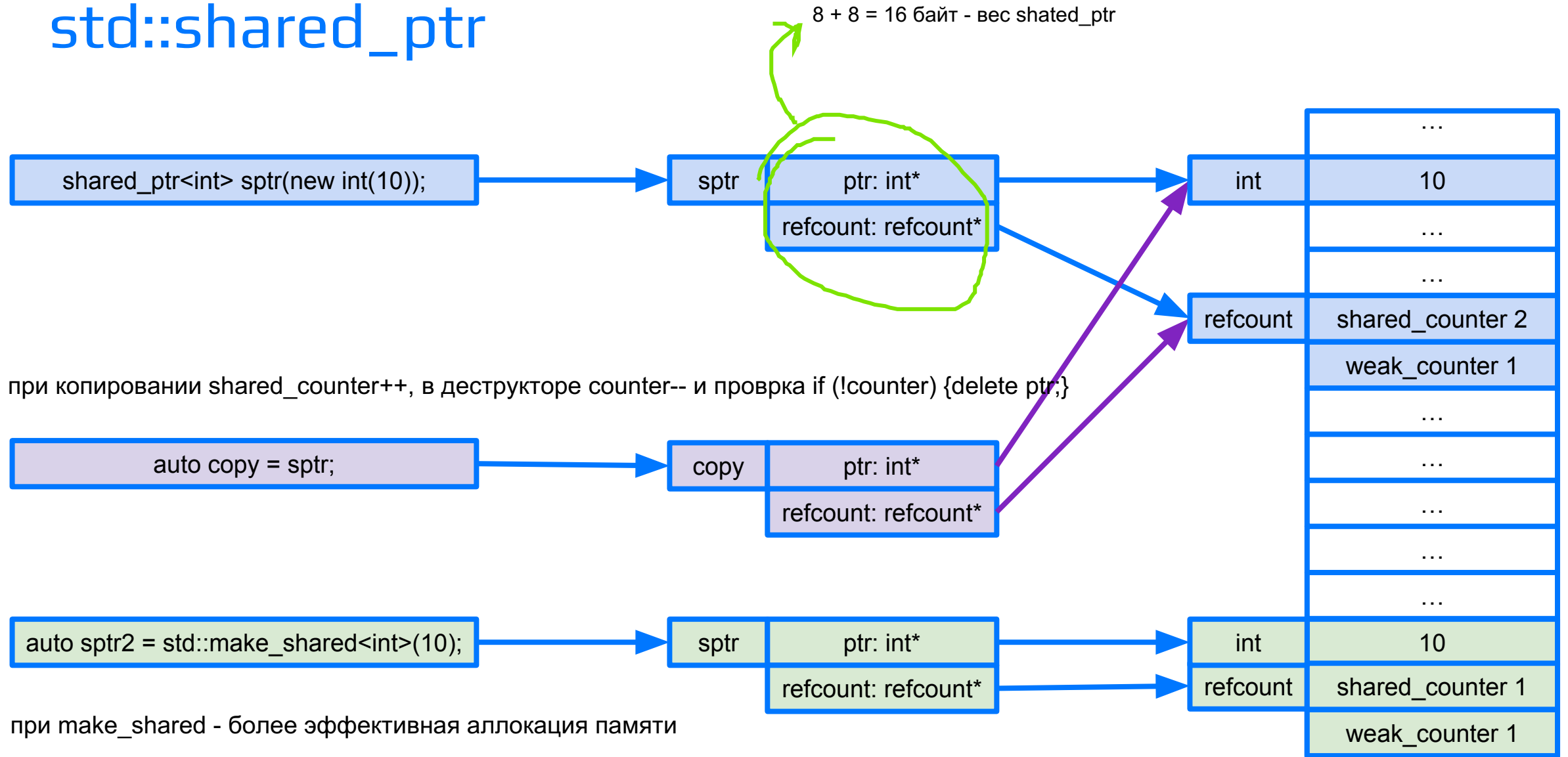
# std::shared\_ptr



# std::shared\_ptr



# std::shared\_ptr

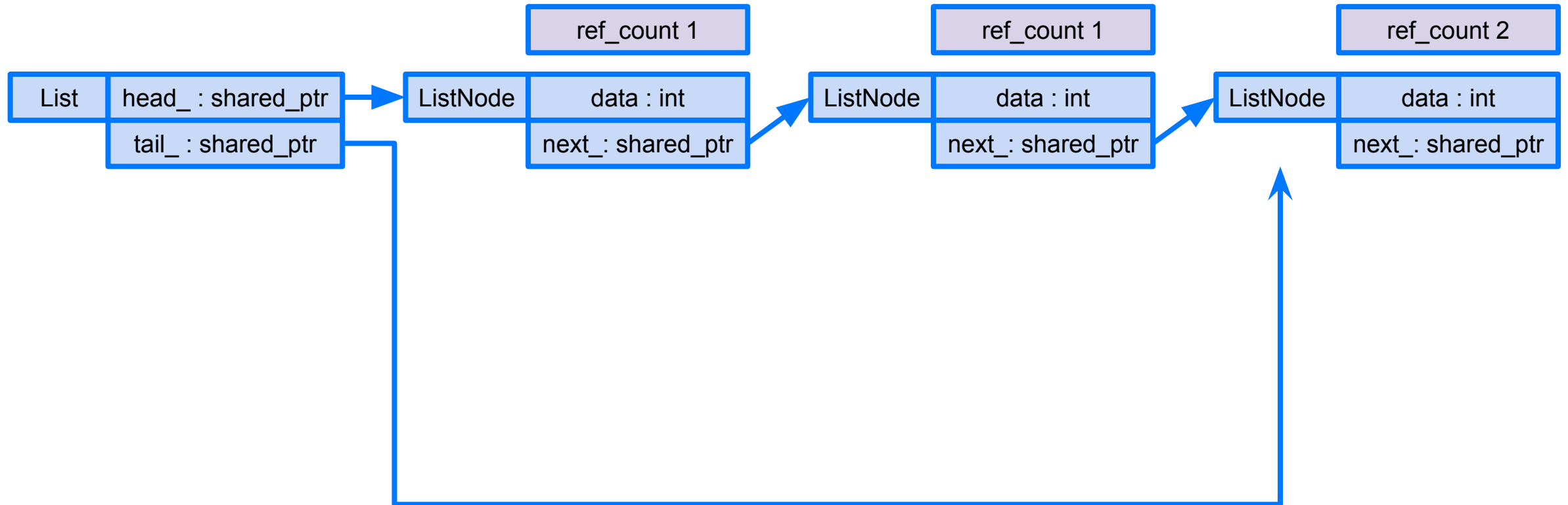


# std::shared\_ptr

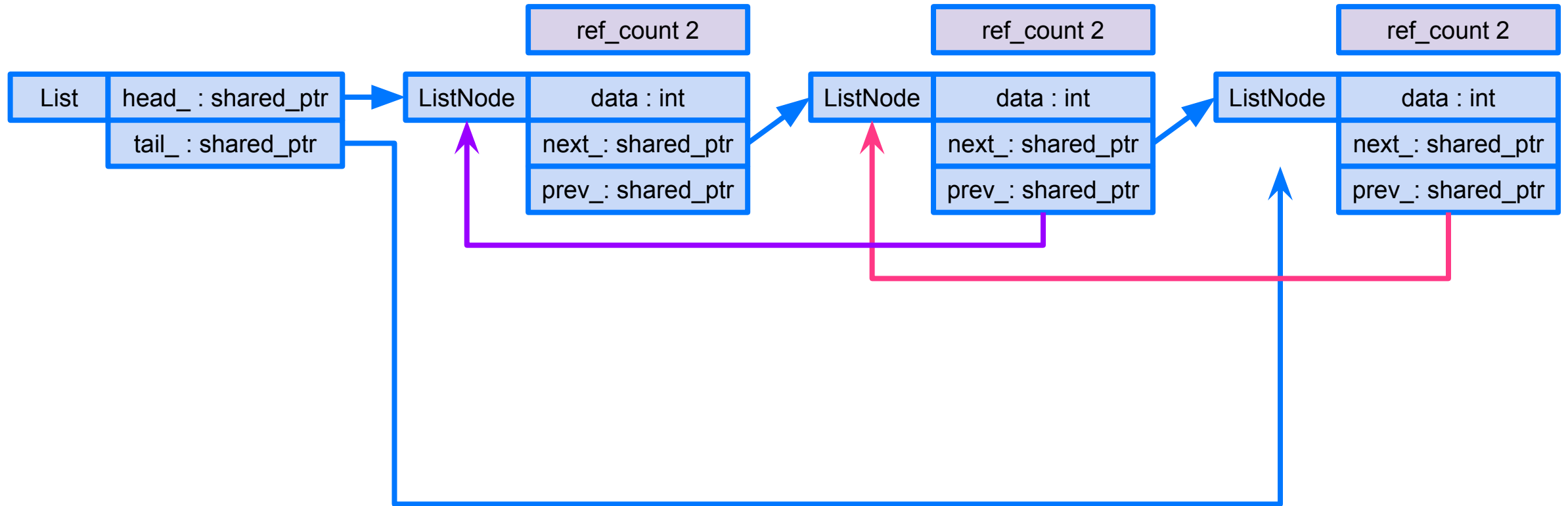
Пример



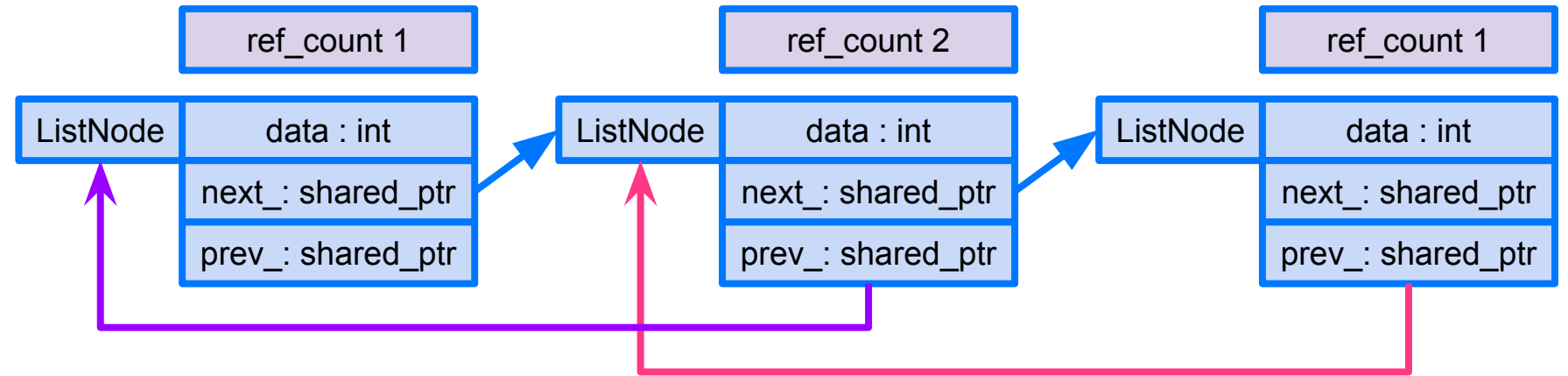
# std::shared\_ptr



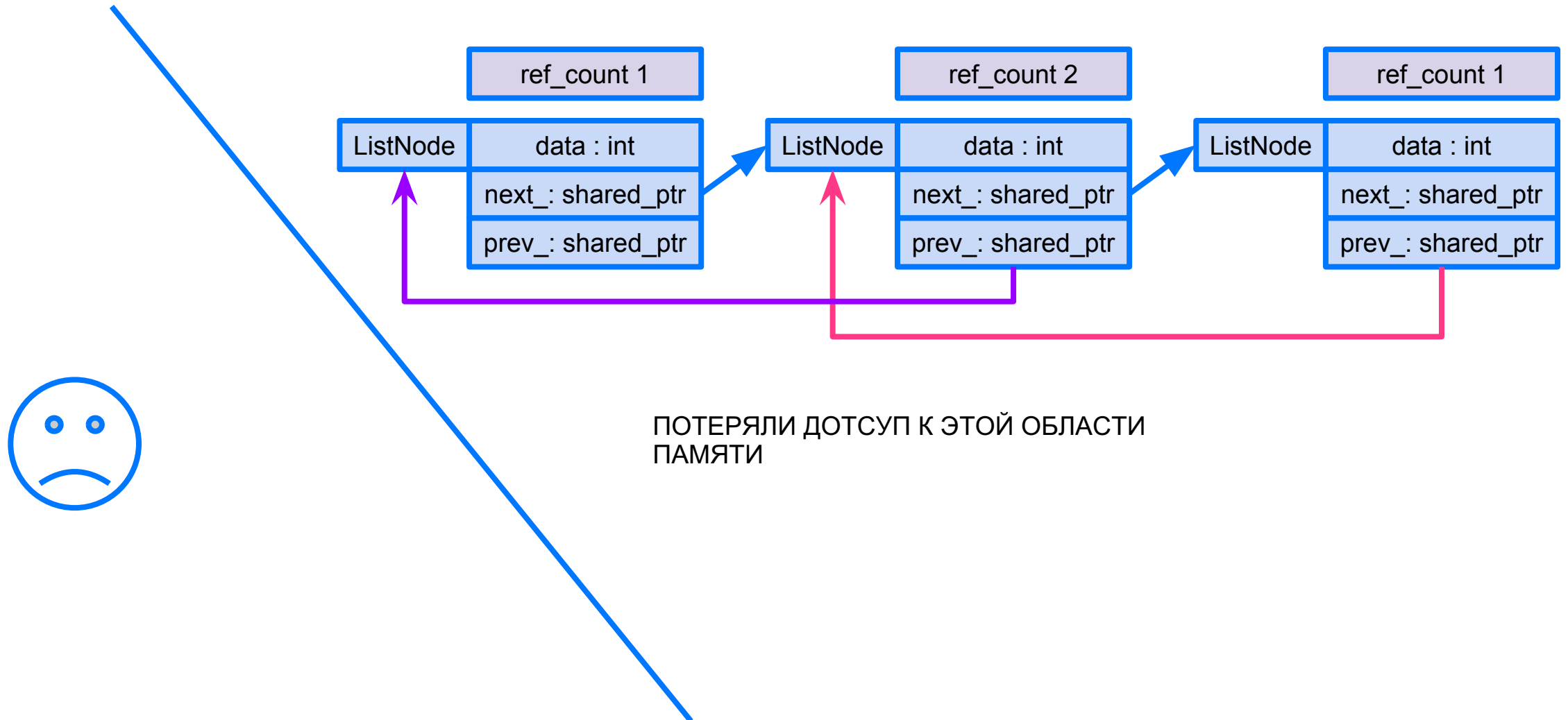
# std::shared\_ptr



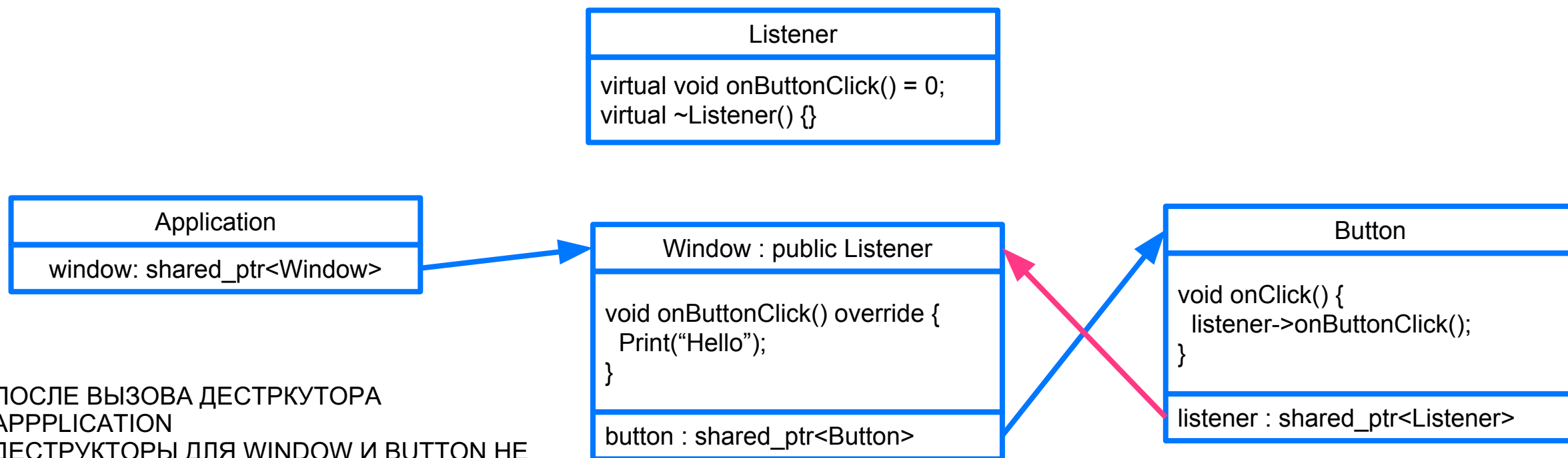
# Проблема циклических ссылок



# std::shared\_ptr



# Примеры циклических ссылок

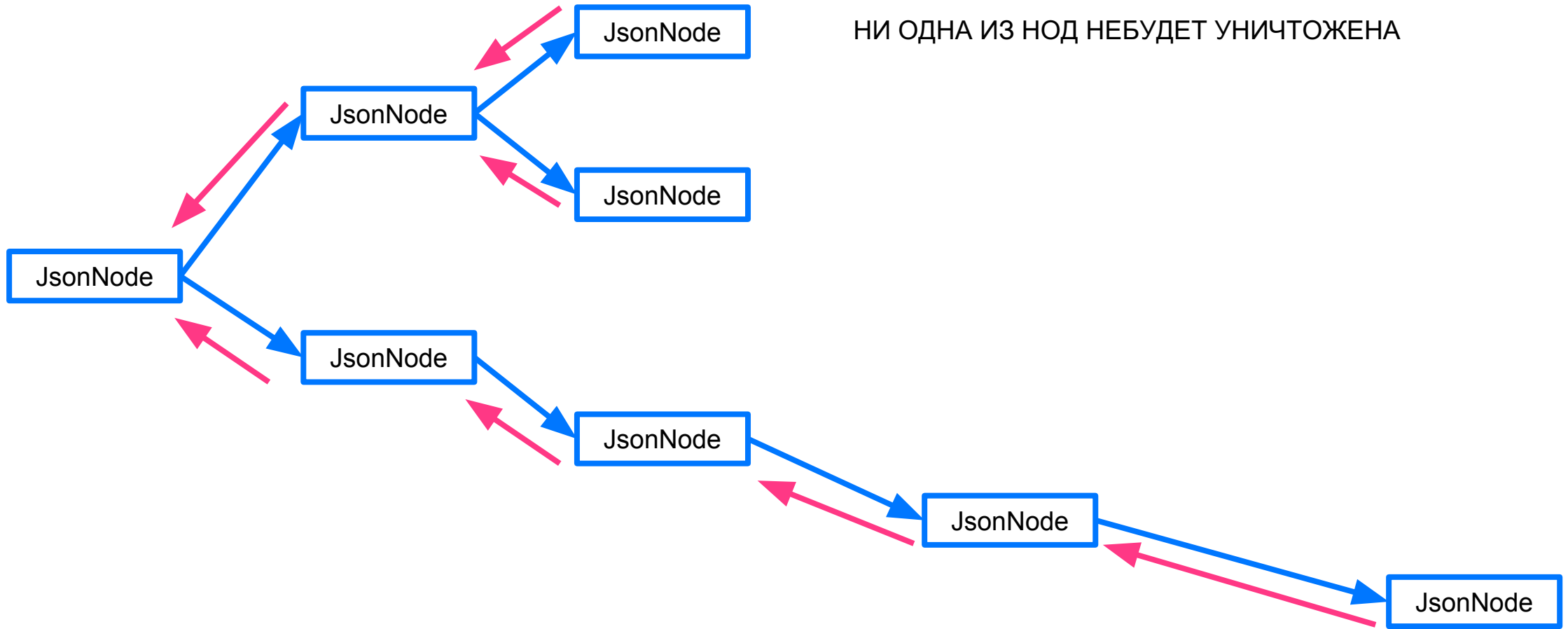


ПОСЛЕ ВЫЗОВА ДЕСТРКУТОРА APPLICATION  
ДЕСТРУКТОРЫ ДЛЯ WINDOW И BUTTON НЕ  
БУДУТ ВЫЗВАНЫ: НА КАЖДЫЙ ССЫЛАЕТСЯ  
МИНИМУМ 1 SHARED\_PTR

(WINDOW НЕ УНИЧТОЖИТСЯ, ПОТОМУ ЧТО НА НЕГО ЕЩЕ БУДЕТ ССЫЛАТЬСЯ BUTTON ЧЕРЕЗ ССЫЛКУ НА ИНТЕРФЕЙСНЫЙ КЛАСС LISTENER, BUTTON  
ЖЕ ОСТАНЕТСЯ В ПАМЯТИ ИЗ-ЗА ТОГО, ЧТО НЕ УНИЧТОЖИТСЯ LISTENER, ССЫЛАЮЩИЙСЯ НА НЕГО -- ЗАМКНУТЫЙ КРУГ)

# Примеры циклических ссылок

НИ ОДНА ИЗ НОД НЕБУДЕТ УНИЧТОЖЕНА

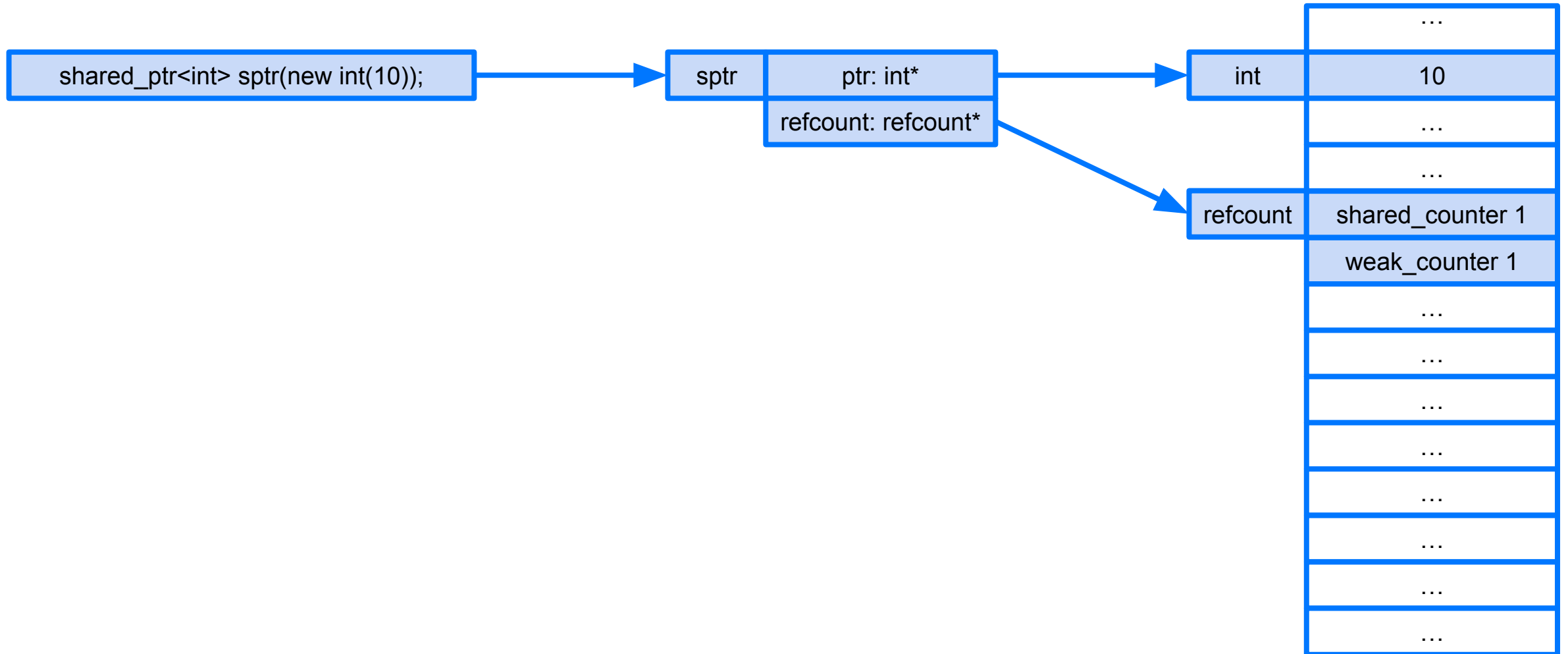


# std::weak\_ptr

Не владеющая обертка над указателем на объект, которым владеет shared\_ptr.

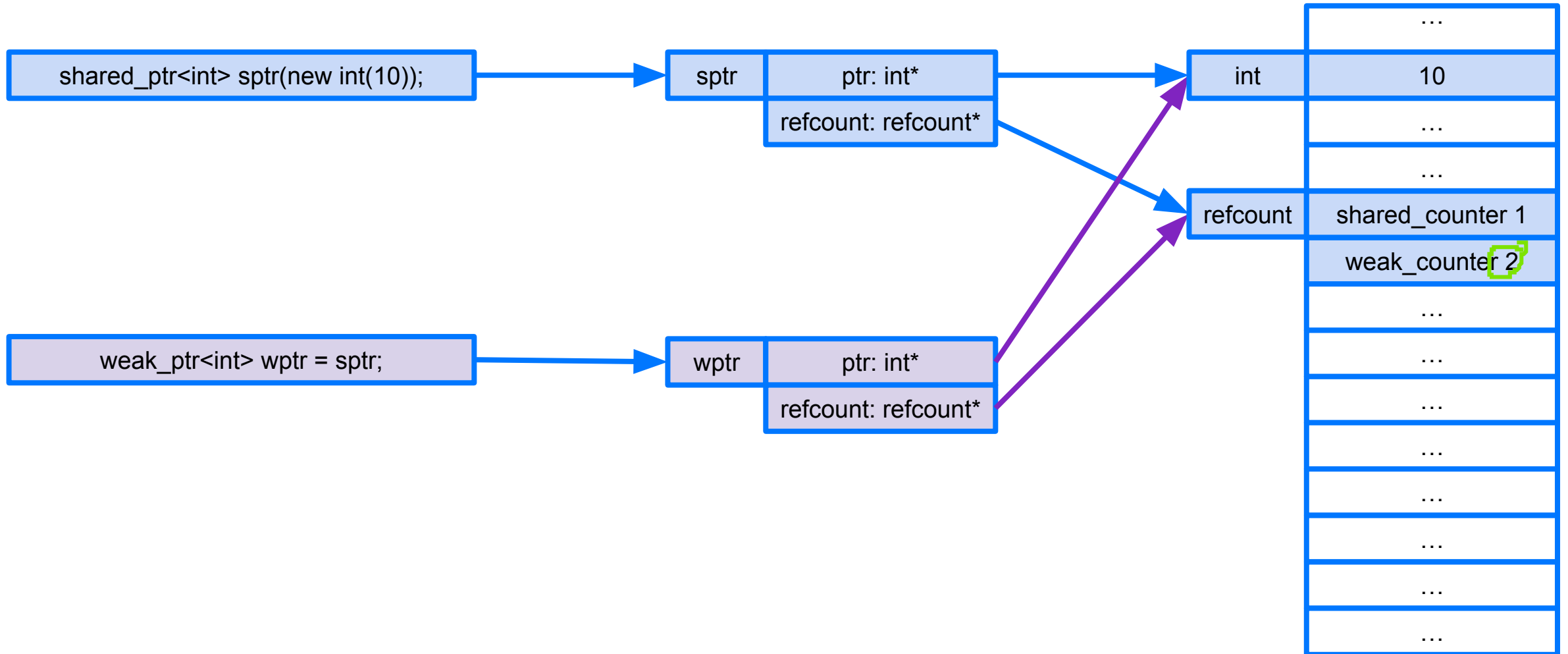
Может проверять что объект еще жив и временно становиться shared\_ptr.

# std::weak\_ptr

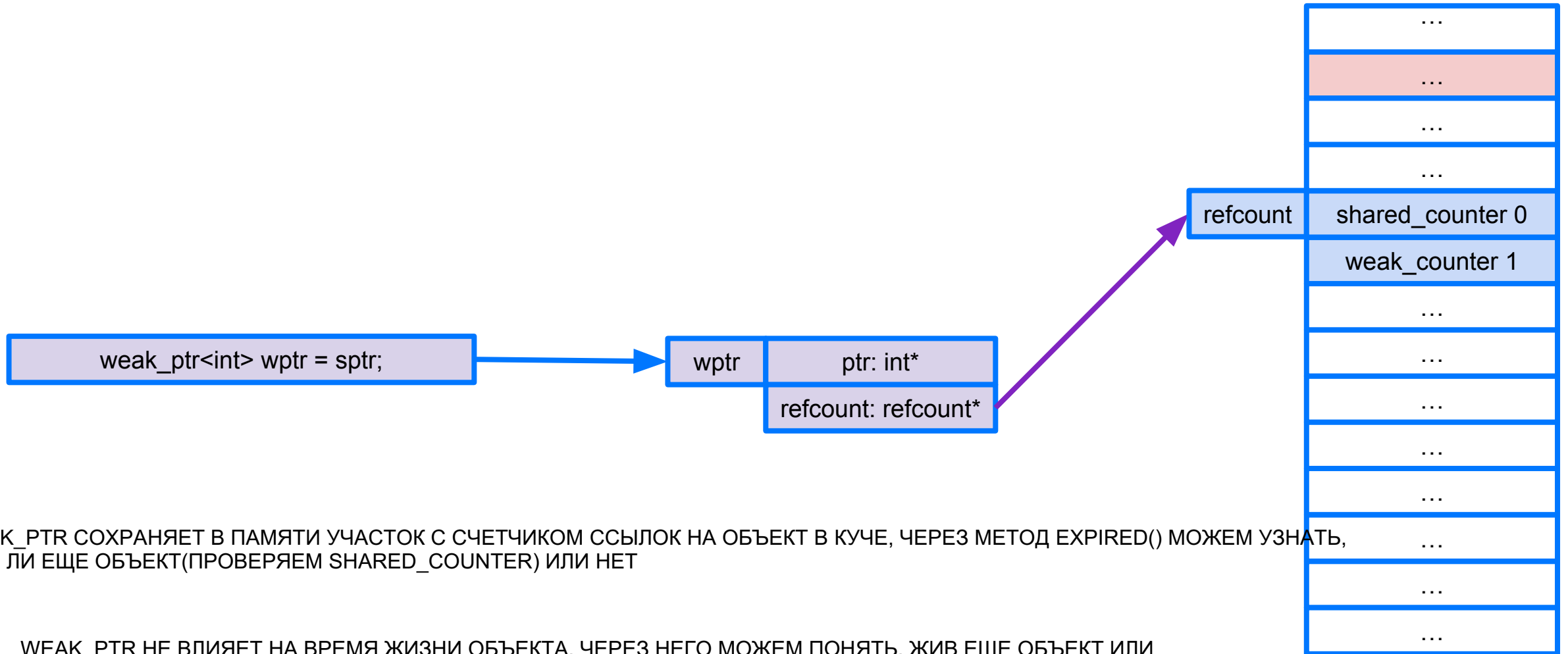




# std::weak\_ptr



# std::weak\_ptr

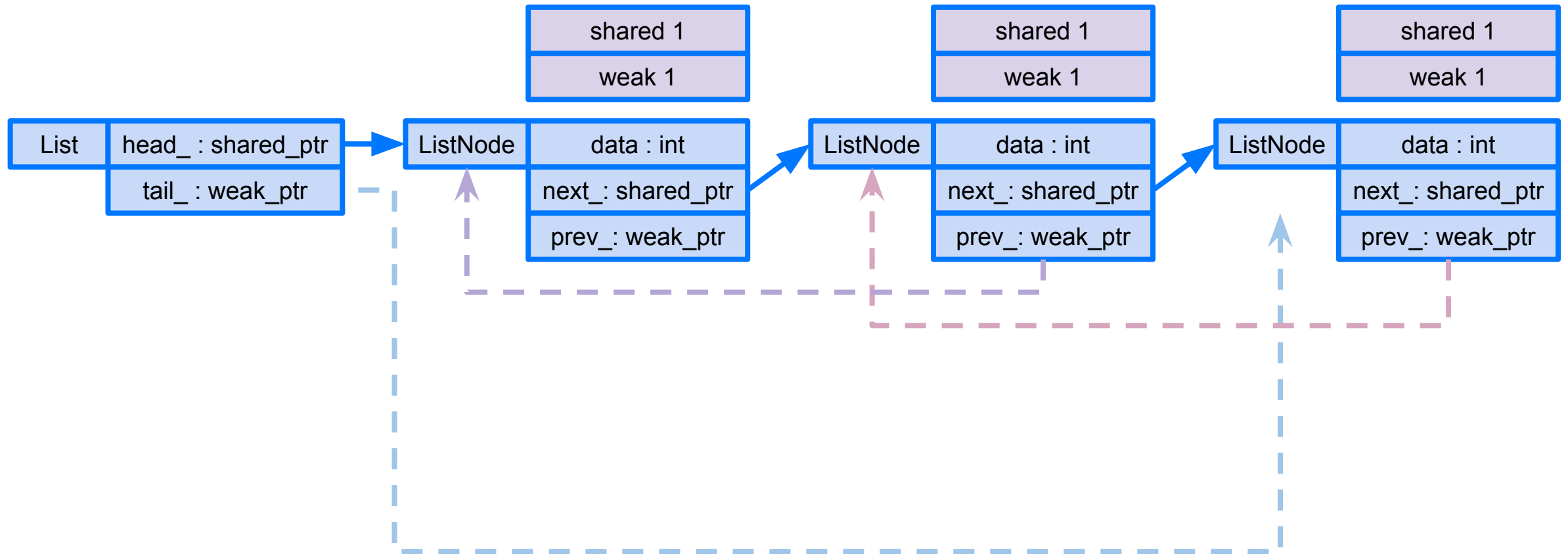


WEAK\_PTR СОХРАНЯЕТ В ПАМЯТИ УЧАСТОК С СЧЕТЧИКОМ ССЫЛОК НА ОБЪЕКТ В КУЧЕ, ЧЕРЕЗ МЕТОД EXPIRED() МОЖЕМ УЗНАТЬ, ЖИВ ЛИ ЕЩЕ ОБЪЕКТ(ПРОВЕРЯЕМ SHARED\_COUNTER) ИЛИ НЕТ

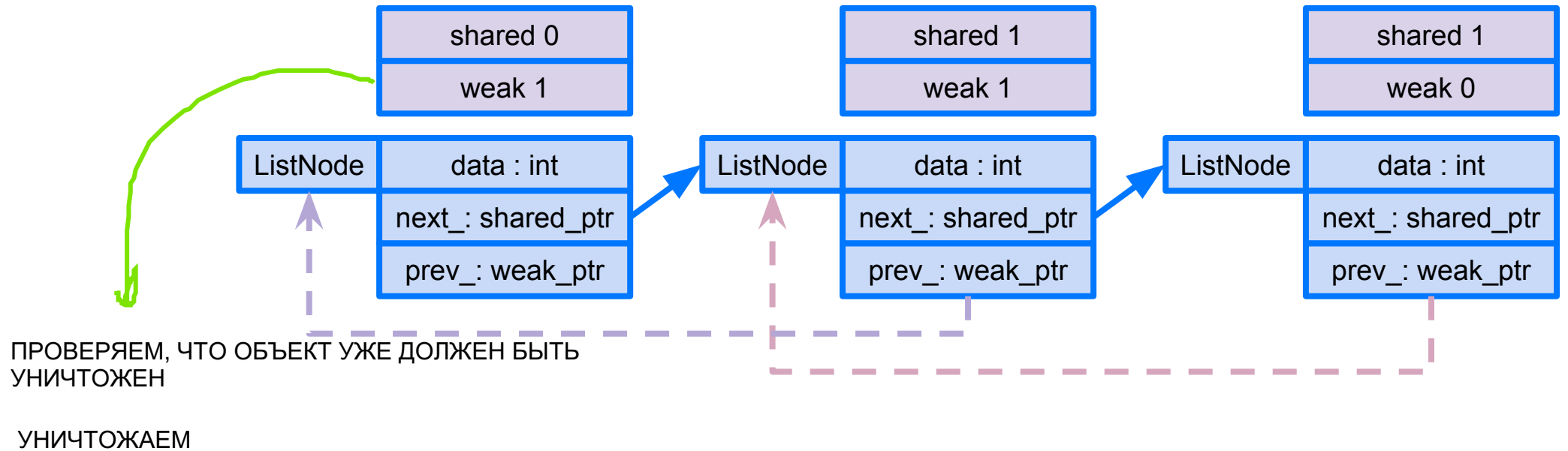
WEAK\_PTR НЕ ВЛИЯЕТ НА ВРЕМЯ ЖИЗНИ ОБЪЕКТА, ЧЕРЕЗ НЕГО МОЖЕМ ПОНЯТЬ, ЖИВ ЕЩЕ ОБЪЕКТ ИЛИ НЕТ

ЕСЛИ WEAK\_COUNTER == 0, ОБЛАСТЬ ПАМЯТИ REFCOUNT  
ОСВОБОЖДАЕТСЯ

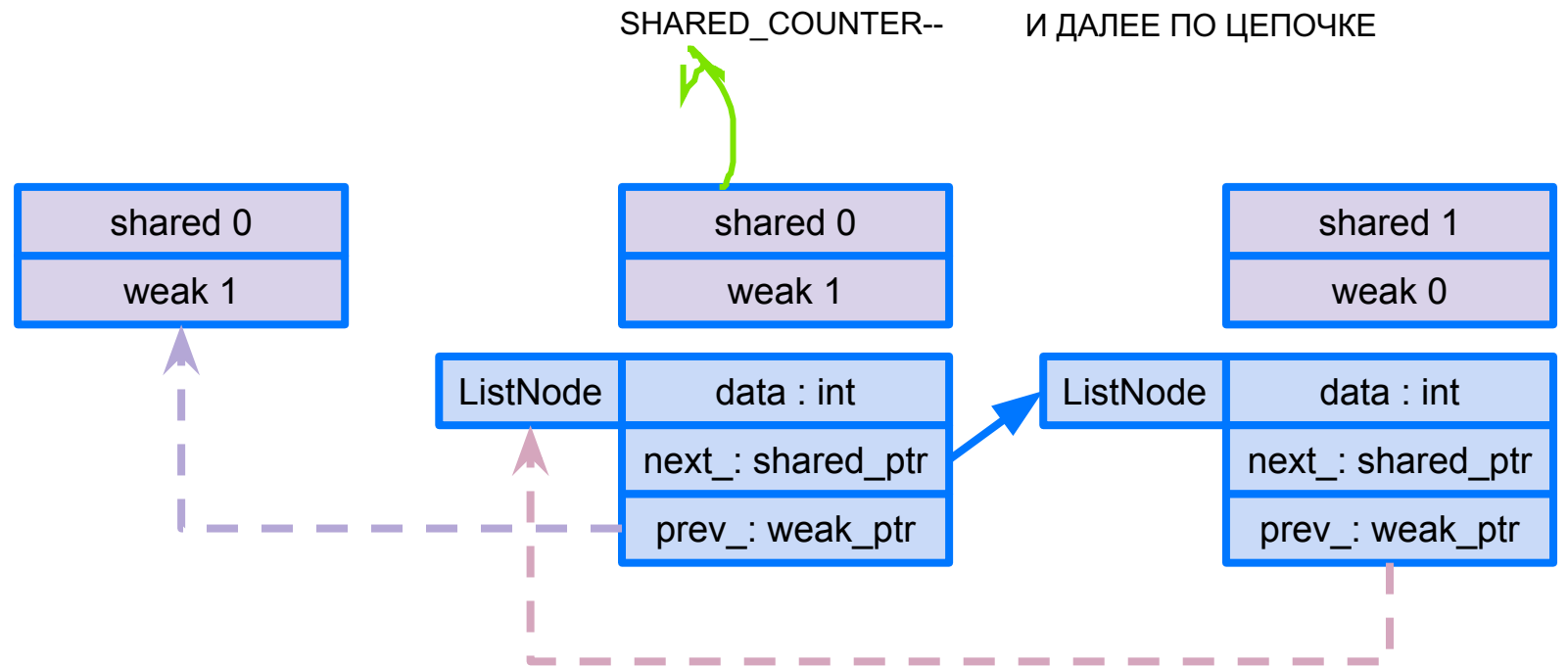
# std::weak\_ptr



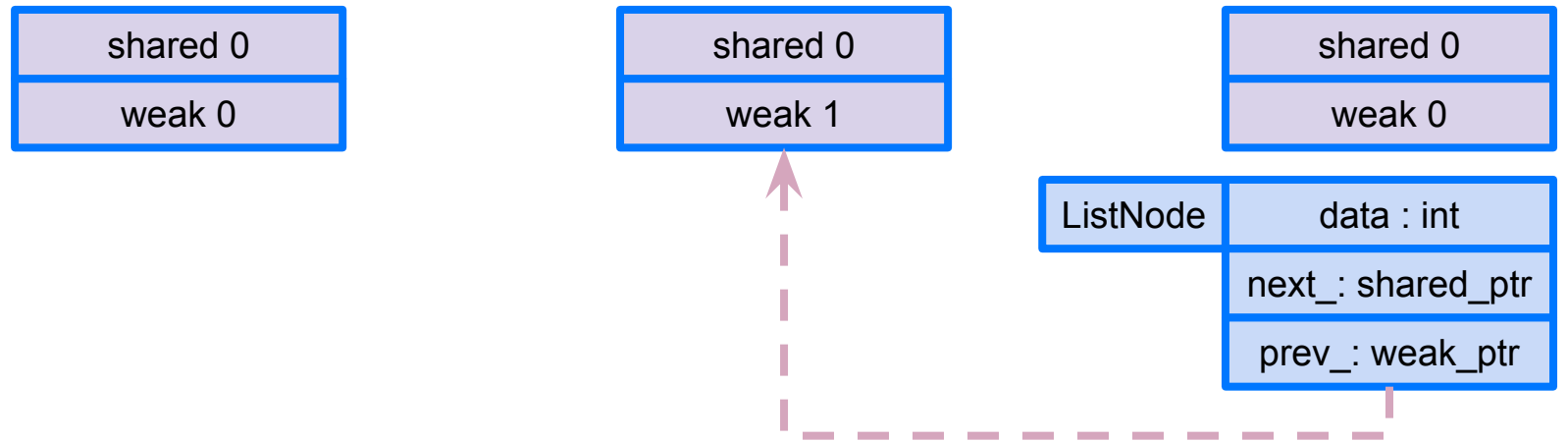
# std::weak\_ptr



# std::weak\_ptr



# std::weak\_ptr

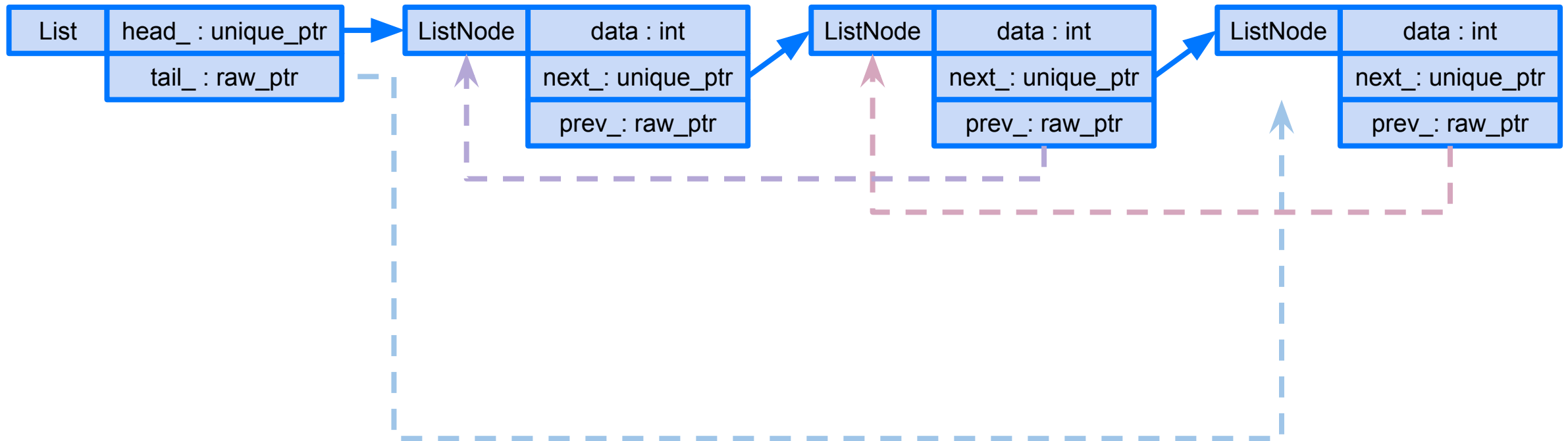


# std::weak\_ptr

shared 0
weak 0

shared 0
weak 0

# std::unique\_ptr + raw\_ptr





# shared\_ptr/weak\_ptr vs unique\_ptr

- unique\_ptr полная аналогия с сырым указателем SIZE: 8 БАЙТ
- shared\_ptr накладнее по памяти SIZE: 16 БАЙТ
- shared\_ptr накладнее по времени работы
- unique\_ptr можно отдать в shared\_ptr ( s\_ptr.reset(std::move(u\_ptr)));
- Один раз став shared\_ptr никогда не может перестать им быть.



Т.К. НИ ПРОГРАММИСТ, НИ SHARED\_PTR НЕ ЗНАЕТ, КТО ЕЩЕ НА ДАННЫЙ МОМЕНТ ВЛАДЕЕТ РЕСУРСОМ, А ПОТОМУ ПРЕВРАЩЕНИЕ SHARED\_PTR В UNIQUE\_PTR, ПРИМЕНЕНИЕ `STD::MOVE(SHARED_PTR)` НЕДОПУСТИМО

# Рекомендации

- Создавать умные указатели лучше через `make_XXX`
- Используйте `shared_ptr`, если владение объектом равноправно **разделяется** между другими объектами (ТО ЕСТЬ `SHARED_COUNTER` НА ПРОТЯЖЕНИИ ЖИЗНИ ОБЪЕКТА МИНИМУМ <sup>2)</sup>
- В противном случае <sup>2)</sup> используйте `unique_ptr`
- В вашей программе **нет утечек памяти** до тех пор, пока в ней нет **`new`** и **`delete`**.

(НЕ СЧИТАЯ ЦИКЛИЧЕСКИЕ ССЫЛКИ В СЛУЧАЕ `SHARED_PTR`)

В СЛУЧАЕ СО СВЯЗКОЙ `UNIQUE_PTR` + `RAW_PTR` - ОПРЕДЕЛЯЕМСЯ, КТО ЕДИНОЛИЧНО ВЛАДЕЕТ ОБЪЕКТОМ И ОПРЕДЕЛЯЕТ ВРЕМЯ ЕГО ЖЕЗНИ, ГЛАВНОЕ: НЕ ПЫТАТЬСЯ УДАЛЯТЬ ОБЪЕКТ ЧЕРЕЗ `RAW_PTR` В СЛУЧАЕ, ЕСЛИ ДЕСТУРКТОР `UNIQUE_PTR` УЖЕ ОТРАБОТАЛ

# Передача умных указателей

Осталось ли место сырым указателям в C++?

# Передача умных указателей

Осталось ли место сырым указателям в C++?

В современном C++ сырой указатель трактуется как невладеющий

`Object* foo();` -- C++98 - НЕПОНЯТНО, ВЛАДЕЮЩИЙ УКАЗАТЕЛЬ ИЛИ НЕТ, ОБЯЗАНЫ ЛИ УДАЛИТЬ ОБЪЕКТ ПО УКАЗАТЕЛЮ ПОСЛЕ ИСПОЛЬЗОВАНИЯ В ФУНКЦИИ ИЛИ НЕТ  
MODERN C++: ЕСЛИ СЫРОЙ УКАЗАТЕЛЬ, ТО ОБЪЕКТОМ ВЛАДЕЕМ НЕ МЫ, ОН БУДЕТ УДАЛЕН ДАЛЕЕ ПО ХОДУ РАБОТЫ ПРОГРАММЫ  
`void foo(Object* obj);` -- АНАЛОГИЧНО: ПРИНЯТО, ЧТО ПРИ ТАКОЙ СИГНАТУРЕ FOO НЕ БУДЕТ ЗАНИМАТЬСЯ УДАЛЕНИЕМ ОБЪЕКТА, МЫ ПО-ПРЕЖНЕМУ ЯВЛЯЕМСЯ ВЛАДЕЛЬЦАМИ ПЕРЕДАННОГО В ФУНКЦИЮ ОБЪЕКТА

# GotW #91 Smart Pointer Parameters

Разбор

# Практика

# Спасибо за внимание!

