OTUS

C++ Professional

Реализация умных указателей

otus.ru

Проверить, идет ли запись

Меня хорошо видно && слышно?

Ставим “+”, если все хорошо
“-”, если есть проблемы

Тема вебинара

Реализация умных указателей

Карина Дорожкина

Research Development Team Lead

10 лет опыта C/C++ разработки как в коммерческих, так и research проектах

Спикер конференций C++ Russia, escar Europe

dorozhkinak@gmail.com

Правила вебинара

Активно
участвуем

Off-topic обсуждаем
в telegram Otus-C++-08-2023

Задаем вопрос
в чат

Вопросы вижу в чате,
могу ответить не сразу

Маршрут вебинара

- Время жизни переменных
- Проблемы управления ресурсами
- Глупый умный указатель
- std::unique_ptr
- std::shared_ptr
- std::weak_ptr

Время жизни переменных

Проблемы управления ресурсами

Недостатки сырых указателей

- Память управляется вручную
- По объявлению не понятно, указывает на одиночный объект или на массив
- Объявление не содержит информации о владении ресурсом
- Нет информации о том, как правильно освобождать ресурс (delete, delete[] или иначе?)
- Нет информации о том, был ли уже освобожден ресурс (повторный delete, memory leak)

RAII

Resource Acquisition is Initialisation

- Инкапсуляция каждого ресурса в классе, который
  - Получает ресурс в конструкторе или выбрасывает исключение
  - Деструктор освобождает ресурс, не выбрасывая исключений
- Ресурс всегда используется через RAII класс, который
  - Имеет автоматическое время жизни или является временным объектом, либо
  - Его время жизни ограничено временем жизни автоматического или временного объекта

Выводы:

- Динамически аллоцированная память может стать причиной утечек или UB
- Для управления динамической памятью используется концепция RAII

Глупый умный указатель

std::auto_ptr C++98

Defined in header <memory>

`template< class T > class auto_ptr;`	(1)	(deprecated in C++11) (removed in C++17)
`template<> class auto_ptr<void>;`	(2)	(deprecated in C++11) (removed in C++17)

Недостатки std::auto_ptr

- Конструктор копирования и оператор присваивания модифицируют исходный объект -> нельзя использовать в STL контейнерах

Недостатки std::auto_ptr

- Нет возможности управлять массивами

```
template<class T>
class auto_ptr
{
public:
    ~auto_ptr() throw() {delete ptr;}
    ...
};
```

```
int* ptr = new int[6];
delete[] ptr; //correct delete

std::auto_ptr<int> smart_ptr(new int[6]); //UB
```

Выводы:

- Не стоит использовать std::auto_ptr, начиная со стандарта C++11

std::unique_ptr

std::unique_ptr C++11

std::auto_ptr deprecated C++11, removed C++17

Defined in header <memory>

```
template<
    class T,
    class Deleter = std::default_delete<T>
> class unique_ptr;                                  (1)   (since C++11)
```

```
template <
    class T,
    class Deleter
> class unique_ptr<T[], Deleter>;                    (2)   (since C++11)
```

- Используется для эксклюзивного владения объектом
- Копирование запрещено
- Move-only тип
- Можно определить пользовательскую функцию для освобождения ресурса

std::default_delete

```
template <class T>
struct default_delete {
void operator()(T* ptr) const noexcept {
    static_assert(sizeof(T) > 0,
                  "default_delete can not delete incomplete type" );
    static_assert(!std::is_void<T>::value,
                  "default_delete can not delete incomplete type" );
    delete ptr;
  }
};
```

Определение пользовательского deleter

Пример загрузки плагина

```
static constexpr int dlopenFlags = RTLD_NOW;
std::string pluginName = "./my_plugin.so";

auto dlDeleter = [&pluginName](void *ptr) {
    if (ptr && dlclose(ptr) != 0)
        std::cerr << "Can't close plugin " << pluginName << ": " << dlerror() << std::endl;

    std::cout << "Plugin " << pluginName << " successfully closed" << std::endl;
};


try {
    std::unique_ptr<void, decltype(dlDeleter)> ptr(dlopen(pluginName.c_str(), dlopenFlags),
dlDeleter);
    …
}
catch (std::exception& ex)
{
}
```

Определение пользовательского deleter

Пример использования с библиотекой leptonica

```
struct LeptonicaPixDeleter
{
    void operator()(Pix* p)
    {
        pixDestroy(&p);
    }
};

unsigned char* data;
size_t imageDataSize;

std::unique_ptr<Pix, LeptonicaPixDeleter> image(pixReadMem(data, imageDataSize));
if (!image)
{
    //...
}
else
{
    Process(std::move(image));
}
```

Передача std::unique_ptr как аргумент функции

- Передавайте std::unique_ptr<T> по значению, если функция должна получить владение указателем

```
struct Widget{
    Widget(int){}
};

void sink(std::unique_ptr<Widget> uniqPtr){
    // do something with uniqPtr
}

int main(){
    auto uniqPtr = std::make_unique<Widget>(1998);

    sink(std::move(uniqPtr));       // (1)
    sink(uniqPtr);                  // (2) ERROR
}
```

Передача std::unique_ptr как аргумент функции

- Передавайте std::unique_ptr<T> по ссылке, если функция должна заново присвоить значение умного указателя

```
struct Widget{
    Widget(int){}
};

void reseat(std::unique_ptr<Widget>& uniqPtr){
    uniqPtr.reset(new Widget(2003));
    // do something with uniqPtr
}
```

Выводы:

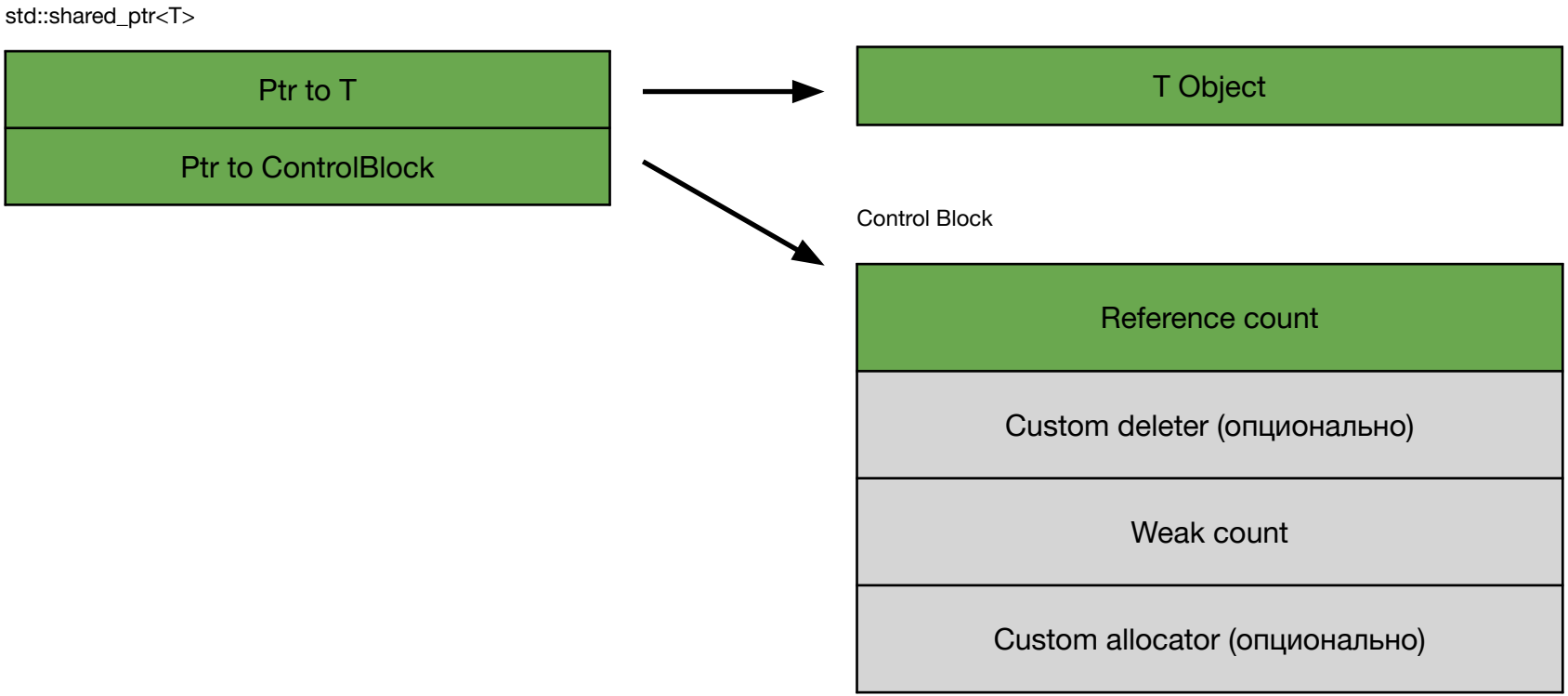
- std::unique_ptr компактный, быстрый, move-only умный указатель для управления ресурсами с эксклюзивным владением
- По умолчанию освобождение ресурсов происходит при помощи delete, но можно определить пользовательский deleter
- Stateful deleter и указатель на функцию увеличивают размер std::unique_ptr
- Для exception safety корректнее использовать std::make_unique (C++14)

std::shared_ptr

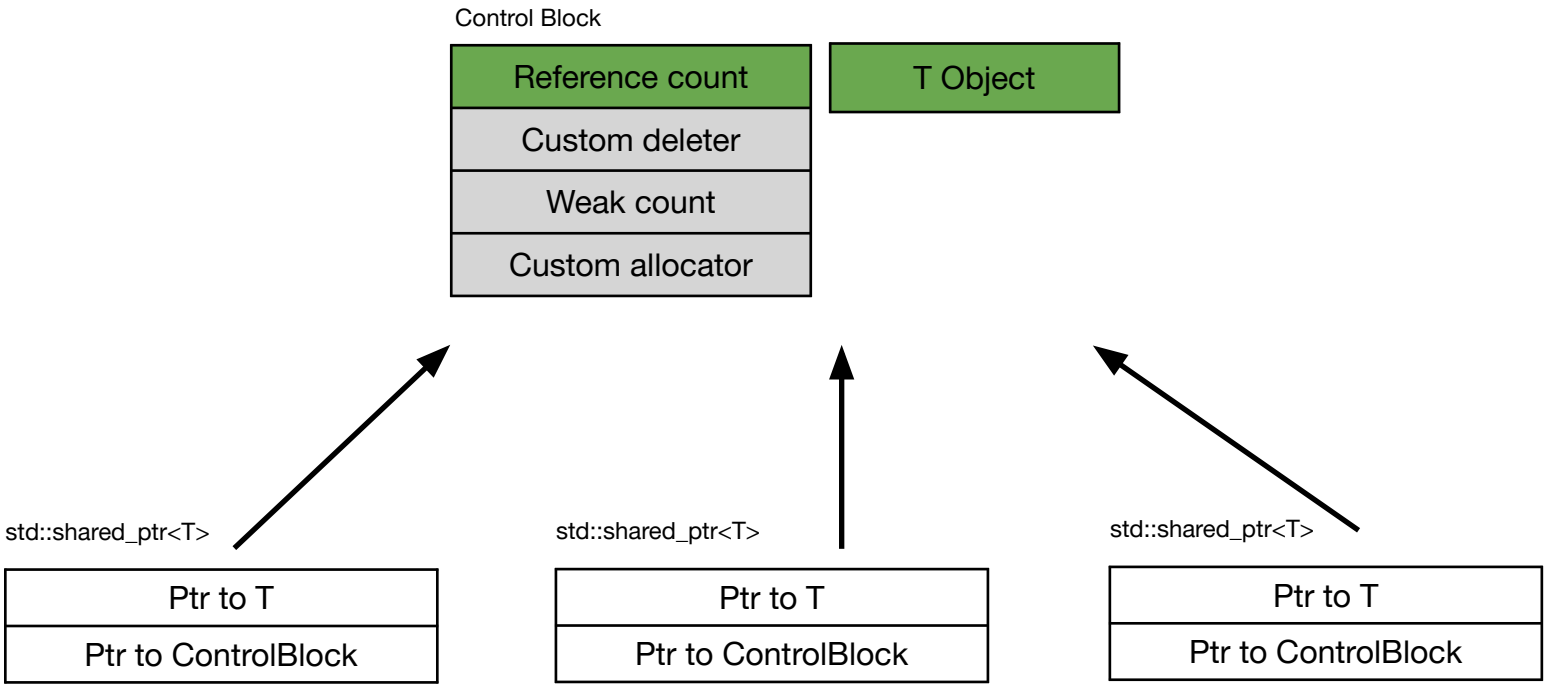
std::shared_ptr C++11

- Позволяет разделять владение сырым указателем
- Ресурс разрушается и память освобождается, когда
  - Последний оставшийся std::shared_ptr разрушается
  - Последний оставшийся std::shared_ptr получает владение над другим указателем через operator = или reset()

std::shared_ptr C++11



std::shared_ptr C++11



Особенности std::shared_ptr из-за ControlBlock

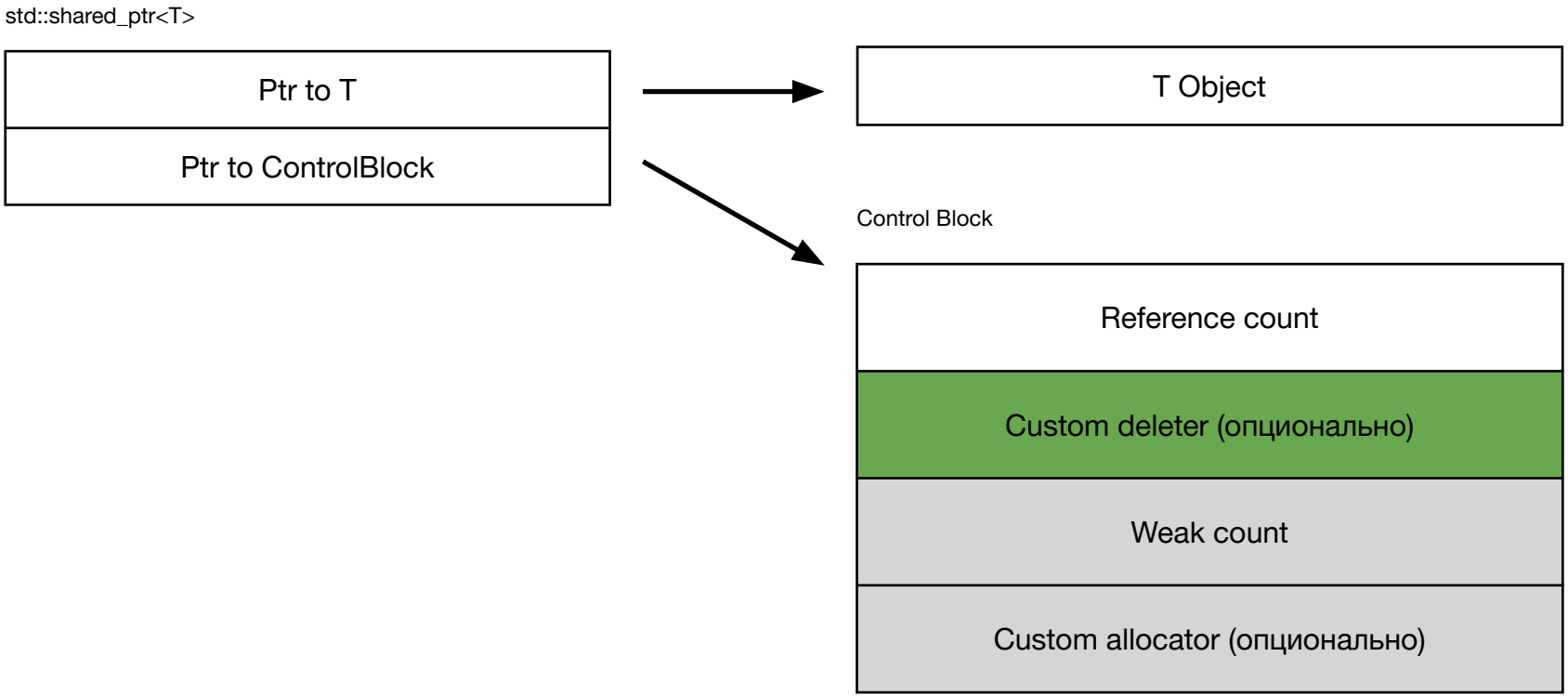
- Размер std::shared_ptr равен размеру двух указателей: указатель на объект и на ControlBlock
- Память для ControlBlock должна быть выделена динамически
- Операции увеличения и уменьшения счетчика должны быть атомарными, так как доступ к разным std::shared_ptr, разделяющим один ресурс, может производиться из разных потоков

Особенности ControlBlock

Создается вместе с первым std::shared_ptr, владеющим ресурсом:

- При вызове std::make_shared
- При конструировании std::shared_ptr из std::unique_ptr
- При конструировании std::shared_ptr из сырого указателя

std::shared_ptr C++11 custom deleter



std::shared_ptr C++11 custom deleter

```
auto customDeleter1 = [](SomeClass *pw) { ... };
auto customDeleter2 = [](SomeClass *pw) { ... };

std::shared_ptr<SomeClass> pw1(new SomeClass, customDeleter1);
std::shared_ptr<SomeClass> pw2(new SomeClass, customDeleter2);
```

std::shared_ptr C++11 custom deleter

В отличие от std::unique_ptr:

- Можно использовать в контейнерах с разными custom deleter

```
std::vector<std::shared_ptr<SomeClass>>  vpw{ pw1, pw2 };
```

- Можно присваивать std::shared_ptr’ы с разными custom deleter
- Можно передавать как аргумент функции std::shared_ptr с разными custom deleter
- Определение custom deleter в std::shared_ptr не меняет его размер

Передача std::shared_ptr как аргумент функции

- Передавайте по значению для разделения владения ресурсом

```
void share(std::shared_ptr<SomeClass> c);
```

- Передавайте по ссылке, если функция может присвоить новый ресурс

```
void reseat(std::shared_ptr<SomeClass>& c);
```

- Передавайте по const ссылке, если нужен доступ к ресурсу (аналогично SomeClass*)

```
void mayShare(const std::shared_ptr<SomeClass>& c);
```

Выводы:

- std::shared_ptr реализует подсчет ссылок, позволяя разделяемое владение произвольным ресурсом
- std::shared_ptr, как правило по размеру в 2 раза больше std::unique_ptr, включает в себя реализацию ControlBlock с необходимыми атомарными операциями над счетчиком ссылок
- Поддерживаются custom deleter, не влияют на размер std::shared_ptr

std::weak_ptr

std::weak_ptr C++11

- Умный указатель, позволяющий получить доступ к разделяемому ресурсу без фактического владения им
- Связан с std::shared_ptr, от которого был создан
- Не освобождает ресурс
- Реализует модель временного владения ресурсом
- Не реализует операторы разыменования указателя

std::weak_ptr C++11

- Может быть создан либо от std::shared_ptr, либо от другого std::weak_ptr
- Нет конструктора, принимающего сырой указатель

```
template< class Y >
weak_ptr( const std::shared_ptr<Y>& r ) noexcept;

template< class Y >
weak_ptr( const weak_ptr<Y>& r ) noexcept;
```

Пример создания:

```
auto ptr = std::make_shared<SomeClass>();
std::weak_ptr<SomeClass> weakPtr(ptr);
```

Observed методы std::weak_ptr

- Возвращает число std::shared_ptr, разделяющих владение ресурсом

```
long use_count() const noexcept;
```

- Эквивалентна use_count() == 0

```
bool expired() const noexcept;
```

std::weak_ptr временное владение ресурсом

```
std::shared_ptr<T> lock() const noexcept;
```

- Если expired() == true, shared_ptr = null

```
auto spw1 = weakPtr.lock();
std::shared_ptr<SomeClass> spw2 = weakPtr.lock();
```

- Если expired() == true, throw std::bad_weak_ptr

```
std::shared_ptr<SomeClass> spw3(weakPtr);
```

Примеры использования std::shared_ptr и std::weak_ptr

Реализация cache

```
std::shared_ptr<Data> loadData(std::size_t id); //грузит данные из базы

std::shared_ptr<Data> loadDataCached(std::size_t id)
{
    static std::unordered_map<std::size_t, std::weak_ptr<Data>> cache;
    auto objPtr = cache[id].lock();
    if (!objPtr) { //отсутствует в cache, либо expired
       objPtr = loadData(id);
       cache[id] = objPtr;
    }
    return objPtr;
}
```

Выводы:

- std::weak_ptr не может существовать без std::shared_ptr
- std::weak_ptr позволяет устранить кольцевую зависимость, например, может быть использован для реализации cache

std::make_shared

std::shared_ptr C++11

```
auto ptr = new int;

std::shared_ptr<int> shared1(ptr);
std::shared_ptr<int> shared2(ptr); // Undefined behaviour
```

Почему UB?

```
std::shared_ptr<int> shared1(new int);
std::shared_ptr<int> shared2(shared1); // GOOD
```

std::make_shared

- Позволяет избежать дублирование типа

```
auto spw1(std::make_shared<SomeClass>());

std::shared_ptr<SomeClass> spw2(new SomeClass);
```

std::make_shared exception safety
(принципиально до C++17)

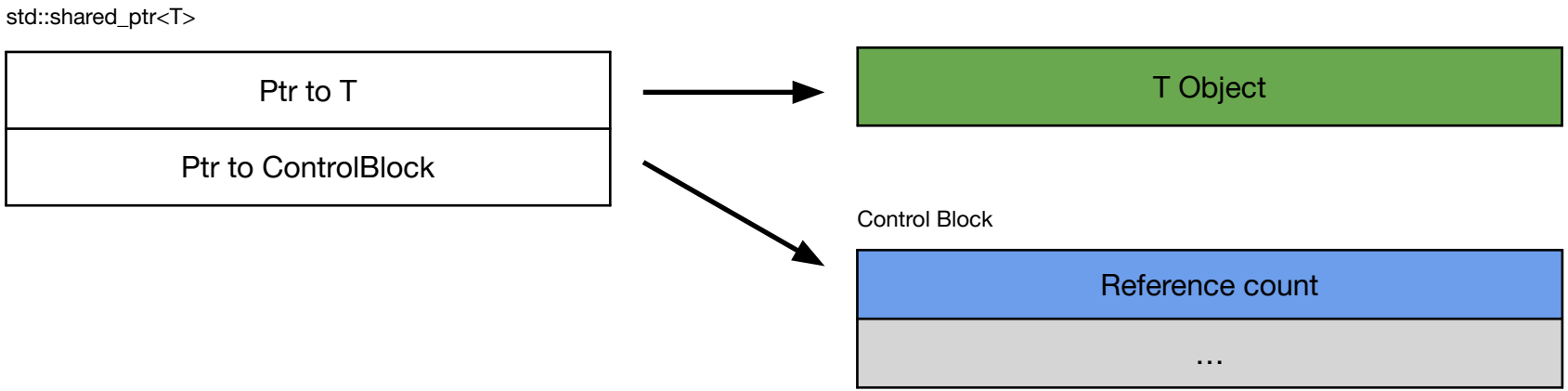
```
processData(std::shared_ptr<Data>(new Data), ComputePriority());

processData(std::make_shared<Data>(), ComputePriority());
```

std::make_shared

- Оптимизирует выделение памяти

```
std::shared_ptr<SomeClass> spw2(new SomeClass);
```

1 выделение памяти для SomeClass
1 выделение памяти для ControlBlock

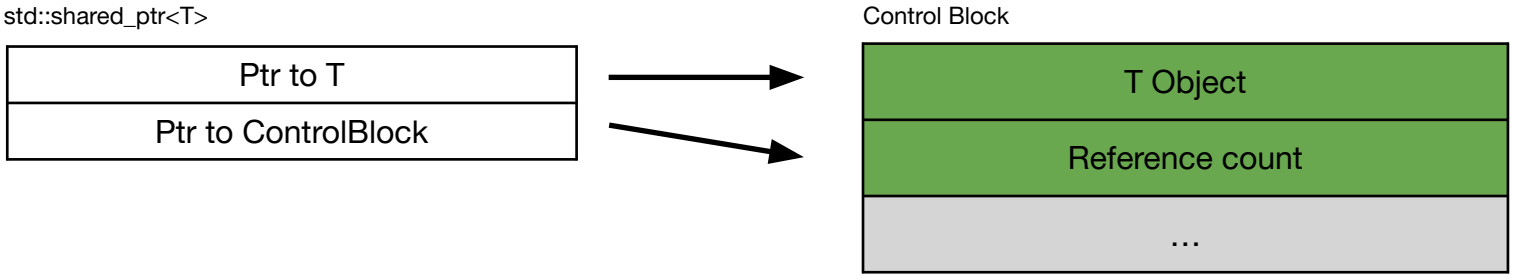


std::make_shared

```
auto spw = std::make_shared<SomeClass>();
```

1 выделение памяти

Уменьшает статический размер программы и ускоряет создание std::shared_ptr



Ограничения std::make_shared

Нет возможности использовать с custom deleter

```
auto customDeleter = [ ](SomeClass*) {...};
std::shared_ptr<SomeClass> ptr(new SomeClass, customDeleter);
```

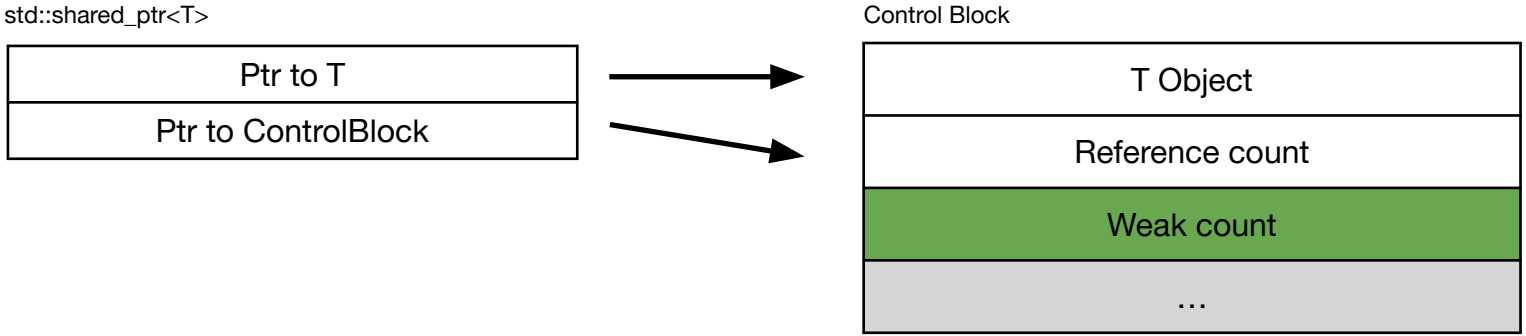
Конструктор std::shared_ptr:

```
template< class Y, class Deleter >
shared_ptr( Y* ptr, Deleter d );
```

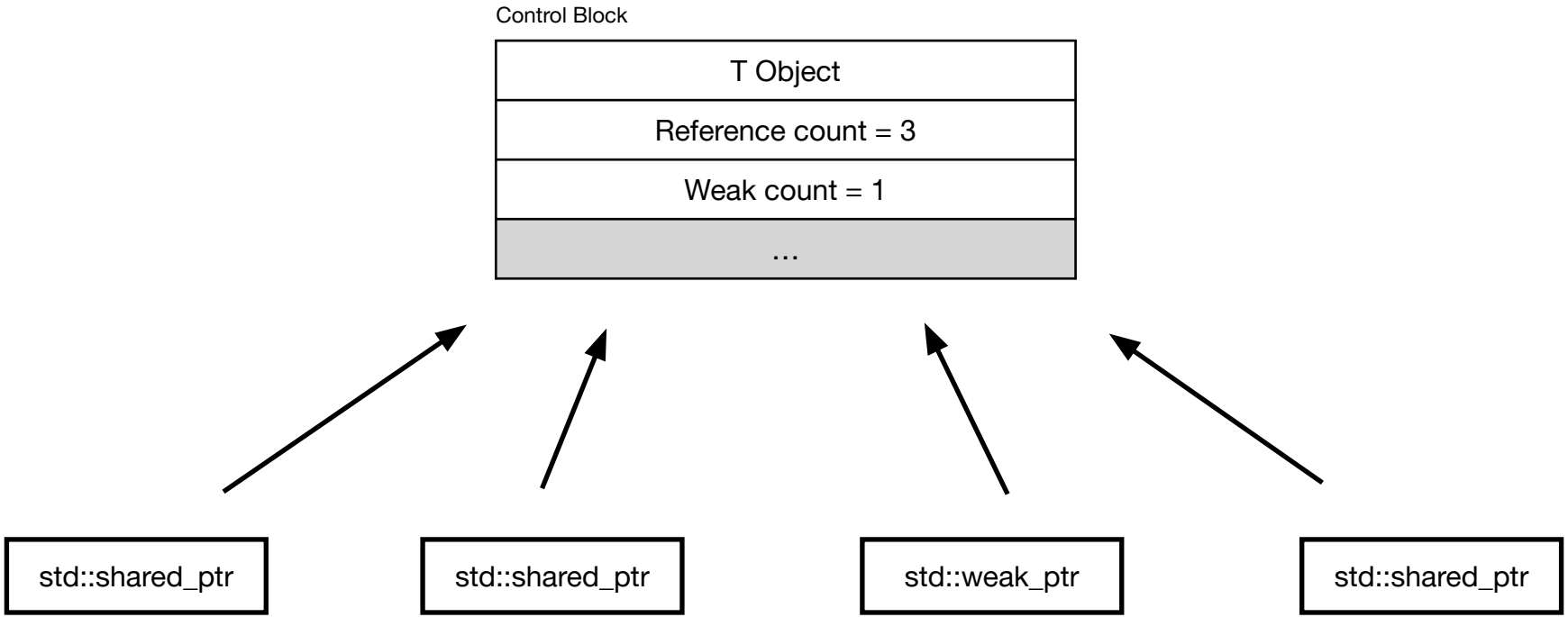
Определение std::make_shared:

```
template< class T, class... Args >
shared_ptr<T> make_shared( Args&&... args );
```

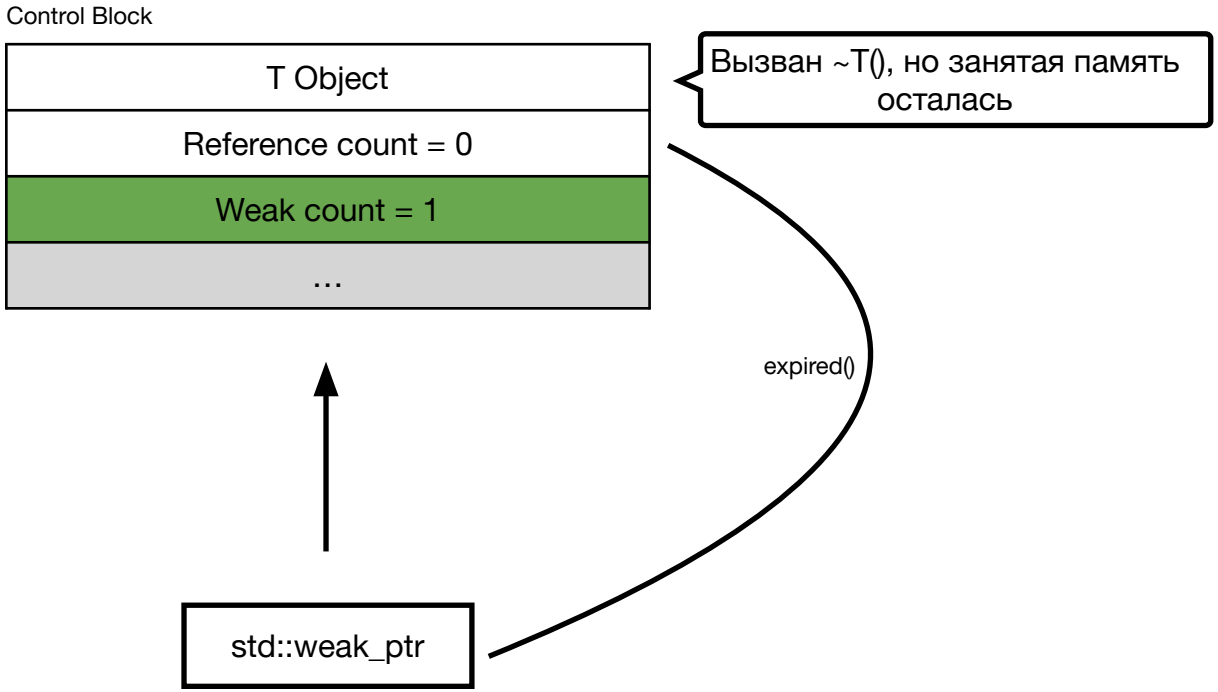
Ограничения std::make_shared



Ограничения std::make_shared



Ограничения std::make_shared



Ограничения std::make_shared

- Память под объект формально будет оставаться выделенной до уничтожения последнего std::weak_ptr

```
class ReallyBigType { ... };
auto pBigObj = std::make_shared<ReallyBigType>();
```

- В случае использования new, память будет освобождена, как только разрушится последний std::shared_ptr, несмотря на наличие std::weak_ptr

```
std::shared_ptr<ReallyBigType> pBigObj(new ReallyBigType);
```

Выводы std::make_shared

Преимущества:

- Уменьшает количества кода
- Exception safe (актуально до C++17)
- Оптимизирует выделение памяти

Недостатки:

- Нельзя использоваться с custom deleter
- Необходимо оценивать объем требуемой памяти под объект и наличие std::weak_ptr

std::enable_shared_from_this

Intro

- Бывают ситуации, когда управляемый через std::shared_ptr ресурс, должен отдать std::shared_ptr на самого себя

```
class Processor
{
public:
    std::shared_ptr<Processor> GetShared()
    {
        return std::shared_ptr<Processor>(this);
    }
};

auto processor = std::make_shared<Processor>();
auto ptr = processor->GetShared();
```

Корректен ли код выше?

std::enable_shared_from_this

Примерная реализация

- Порожденный std::shared_ptr должен ссылаться на уже существующий ControlBlock
- На ControlBlock могут ссылаться и std::shared_ptr, и std::weak_ptr
- Хранение std::shared_ptr в объекте на самого себя делает объект бессмертным

```
struct Immortal {
 //....
 std::shared_ptr<Immortal> self;
};

shared_ptr<Immortal> immortal;
```

Выводы std::enable_shared_from_this

- Позволяет корректно отдать std::shared_ptr на this
- Использует внутренний std::weak_ptr для нахождения исходного ControlBlock

Пара задач на повторение

Корректен ли код?

```
auto i = new int{41};
unique_ptr<int> ptr1(i);
unique_ptr<int> ptr2(i);
```

Корректен ли код?

```
unique_ptr<int> ptr1(new int{41});
unique_ptr<int> ptr2(std::move(ptr1));
```

```
auto ptr1 = std::make_unique<int>(41);
```

Корректен ли код?

```
std::weak_ptr<int> weak(new int(52));
std::shared_ptr<int> shared(weak);
std::cout << *weak;
```

```cpp
auto shared = std::make_shared<int>(52);
std::weak_ptr<int> weak(shared);

auto weak_shared = weak.lock();
if (weak_shared)
{
        std::cout << *weak_shared;
}
```

Вопросы?

Ставим “+”,
если вопросы есть

Ставим “–”,
если вопросов нет

Заполните, пожалуйста, опрос о занятии по ссылке в чате

Следующий вебинар

20 сентября 2023

Идея аллокаторов

- Ссылка на вебинар будет в ЛК за 15 минут
- Материалы к занятию в ЛК — можно изучать
- Обязательный материал обозначен красной лентой