



Умные указатели

ЛЕКЦИЯ №8

Core Guideline

R.20: Use `unique_ptr` or `shared_ptr` to represent ownership

<https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>

Область видимости

```
1. auto x = 42;  
2. auto foo() {  
3.     return x;  
4. }  
5. int main() {  
6.     auto value = foo();  
7.     return 1;  
8. }
```

Область видимости

```
1. auto x = 42;
2. auto foo( ) {
3.     auto x = 10;
4.     return x;
5. }
6. int main( ) {
7.     auto value = foo( );
8.     return 1;
9. }
```

Область видимости

```
1. auto x = 42;  
2. auto foo() {  
3.     auto x = 10;  
4.     return x;  
5. }  
6. int main() {  
7.     int x = 37;  
8.     auto value = foo();  
9.     return 1;  
10. }
```

Область видимости

```
1. auto x = 42;

2. auto foo() {
3.     return x;
4. }

5. template<class T> auto bar(T y) {
6.     return x + y;
7. }

8. int main() {
9.     auto value = foo();
10.    auto value2 = bar(10);
11.    return 1;
12.}
```

Время жизни

1. статическое (static) – глобальные переменные и static
2. потоковое – thread_local
3. автоматическое – scope (на стэке)
4. динамическое – new / delete, malloc / free

Сырые указатели

```
int main() {  
    int a;  
    a = 10; // Не знаем адрес, да и ладно  
    int * ptr = &a;  
    *ptr += 5; // Зачем-то знаем адрес, но не используем  
    int * ptr2 = &a;  
    10. *(ptr+42) += 5; // Знаем адрес, но используем как-то неправильно  
    int& ref = a;  
    ref += 3; // Не знаем адрес, но ссылаемся  
    return 1;  
}
```


Сырые указатели

```
int main() {  
    {  
        int * ptr = new int{42};  
    } // выход за scope - утечка  
  
    {  
        int value = 0;  
        int * ptr = new int{50};  
        // опять потеряли адрес - утечка  
        ptr = &value;  
    }  
    {  
        int * ptr = new int{79};  
        // Утечка, если функция бросит исключение  
        someFunctionHere();  
        delete ptr;  
    }  
}
```

Сырые указатели

1. нет контроля создания / удаления
2. может указывать в неизвестность, nullptr
3. может указывать в известность, но чужу

RAII

Resource Acquisition Is Initialization

Получение ресурса есть инициализация (RAII) — программная идиома объектно-ориентированного программирования, смысл которой заключается в том, что с помощью тех или иных программных механизмов получение некоторого ресурса неразрывно совмещается с инициализацией, а освобождение — с уничтожением объекта.

Типичным (хотя и не единственным) способом реализации является организация получения доступа к ресурсу в **конструкторе**, а освобождения — в **деструкторе** соответствующего класса.

Поскольку деструктор автоматической переменной вызывается при выходе её из области видимости, то ресурс гарантированно освобождается при уничтожении переменной. Это справедливо и в ситуациях, в которых возникают **исключения**.

Не сильно умный указатель `custom_unique.cpp`

```
template<class T>
struct smart_ptr {
    smart_ptr() : m_ptr{new T{}} {
    }
    ~smart_ptr() {
        delete m_ptr;
    }
    T* get() {
        return m_ptr;
    }
private:
    T* m_ptr;
};
```

Чуть умнее указатель custom_unique_2.cpp

```
auto main() -> int {
    smart_ptr<SomeClass> ptr1;

    std::cout << "start" << std::endl;
    {
        smart_ptr<SomeClass> ptr2{new SomeClass()};
        ptr1 = ptr2; //дублируем
        ptr2 = smart_ptr<SomeClass>{new SomeClass()}; // затираем
    }
    std::cout << "end" << std::endl;
    return 0;
}
```

std::unique_ptr #include<memory.h>

UNIQUE_PTR.CPP

```
std::unique_ptr<int> ptr{new int{10}};  
    assert(ptr);  
    assert(*ptr == 10);  
    assert(*ptr.get() == 10);  
    std::cout << "sizeof(ptr) = " << sizeof(ptr) << std::endl;
```

Перемещаем `unique_ptr`

`unique_copy.cpp`

Единственные доступные операторы перемещения и копирования:

```
unique_ptr& operator= (unique_ptr&& x) noexcept; // копируем из rvalue  
assign null pointer (2)    unique_ptr& operator= (nullptr_t) noexcept
```

Для явного перемещения содержимого можно использовать `std::move`

std::move

```
template< class T >  
typename std::remove_reference<T>::type&& move( T&& t );
```

Возвращает объект LValue с помощью шаблона структуры std::remove_reference, которая помогает получить тип без ссылок:

```
template< class T > struct remove_reference      {typedef T type;};  
template< class T > struct remove_reference<T&>  {typedef T type;};  
template< class T > struct remove_reference<T&&> {typedef T type;};
```


`std::unique_ptr` `#include<memory.h>`

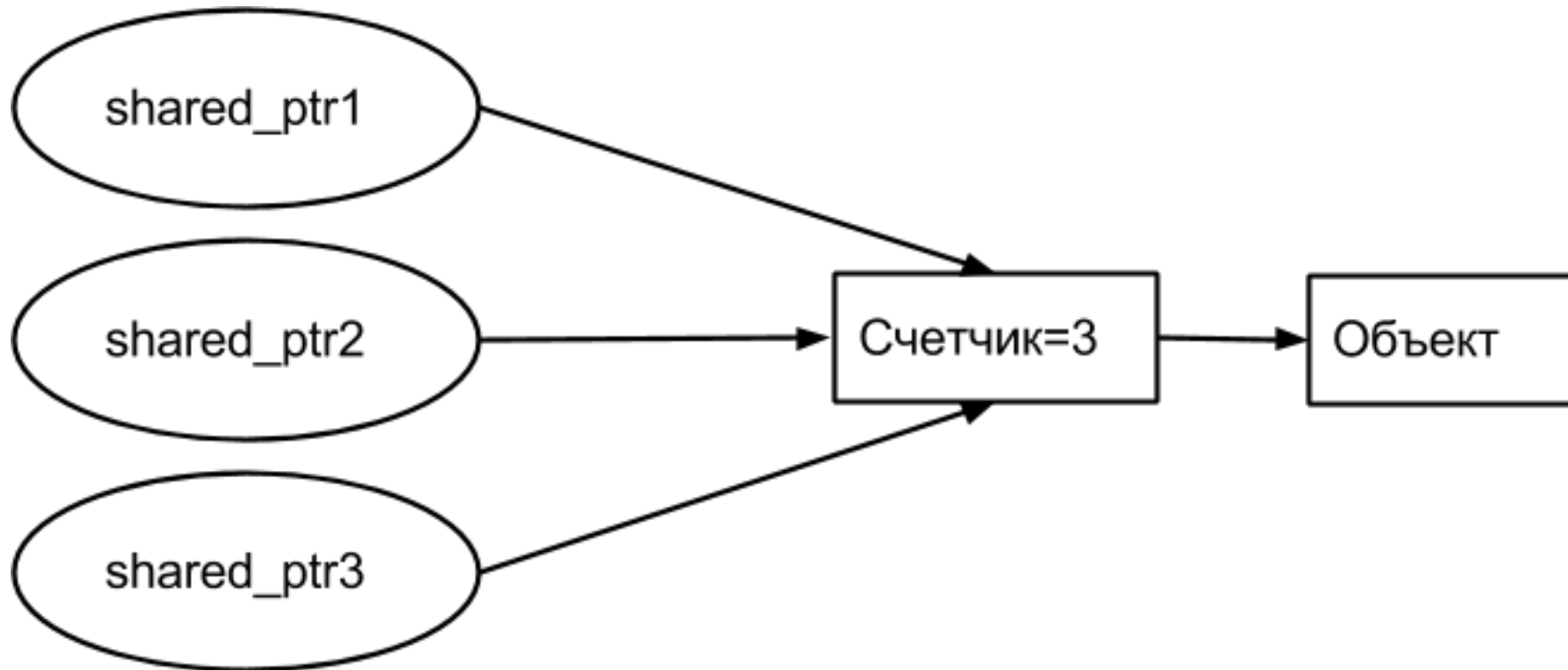
1. нераздельное владение объектом
2. нельзя копировать (только перемещение)
3. размер зависит от пользовательского deleter-a
4. без особой логики удаления издержки чаще отсутствуют
5. `std::make_unique` – только в качестве «синтаксического сахара»

Разделяемый указатель custom_shared.cpp

```
template<class T> struct smart_ptr {
    smart_ptr(T* ptr) : m_counter{new std::size_t{1}}, m_ptr{ptr} {
    }
    smart_ptr(const smart_ptr& other)
    : m_counter{ other.m_counter }, m_ptr{ other.m_ptr } {
    ++*m_counter;
    }
    ~smart_ptr() {
    if (--*m_counter == 0) {
        delete(m_ptr);
        delete(m_counter);
    }
    }
private:
    T* m_ptr;
    std::size_t* m_counter;
};
```



Простой подсчет ссылок на объекты (то есть, копий shared_ptr)



Шаблон `std::shared_ptr<T>`

`#include<memory>`

1. Предоставляет возможности по обеспечению автоматического удаления объекта, за счет подсчета ссылок указатели на объект;
2. Хранит ссылку на один объект;
3. При создании `std::shared_ptr<T>` счетчик ссылок на объект увеличивается;
4. При удалении `std::shared_ptr<T>` счетчик ссылок на объект уменьшается;
5. При достижении счетчиком значения 0 – объект автоматически удаляется;

std::shared_ptr shared_ptr.cpp

1. можно копировать с разделением владения
2. но дешевле перемещать
3. всегда внутри два указателя
4. std::make_shared – выделяет память сразу под объект и счетчик за один раз!
5. потокобезопасный (и хорошо, и плохо)
6. можно создать из unique_ptr

Двойное удаление

```
int * ptr = new int{42};  
  
{  
  
    std::shared_ptr<int> smartPtr1{ptr};  
    std::shared_ptr<int> smartPtr2{ptr};  
  
} // двойное удаление
```

shared_ptr в списке параметров функций

shared_exception.cpp

```
try{

    /*
    A *a=new A("A");
    foo(foofoofoo(),std::shared_ptr<A>(a));
    */
    //std::shared_ptr<A> a(new A("B"));
    foo(std::shared_ptr<A>(new A("B")),foofoofoo());

    /**/

}catch(...){
    std::cout<< "Catch" << std::endl;
}
```

std::dynamic_pointer_cast<T> dynamic_pointer_cast.cpp

```
std::shared_ptr<B> b(new B());

std::shared_ptr<A> ptr = b;

if(std::shared_ptr<B> ptr_b = std::dynamic_pointer_cast<B>(ptr)){
    ptr_b->Do();
}
```


enable_shared_from_this.cpp

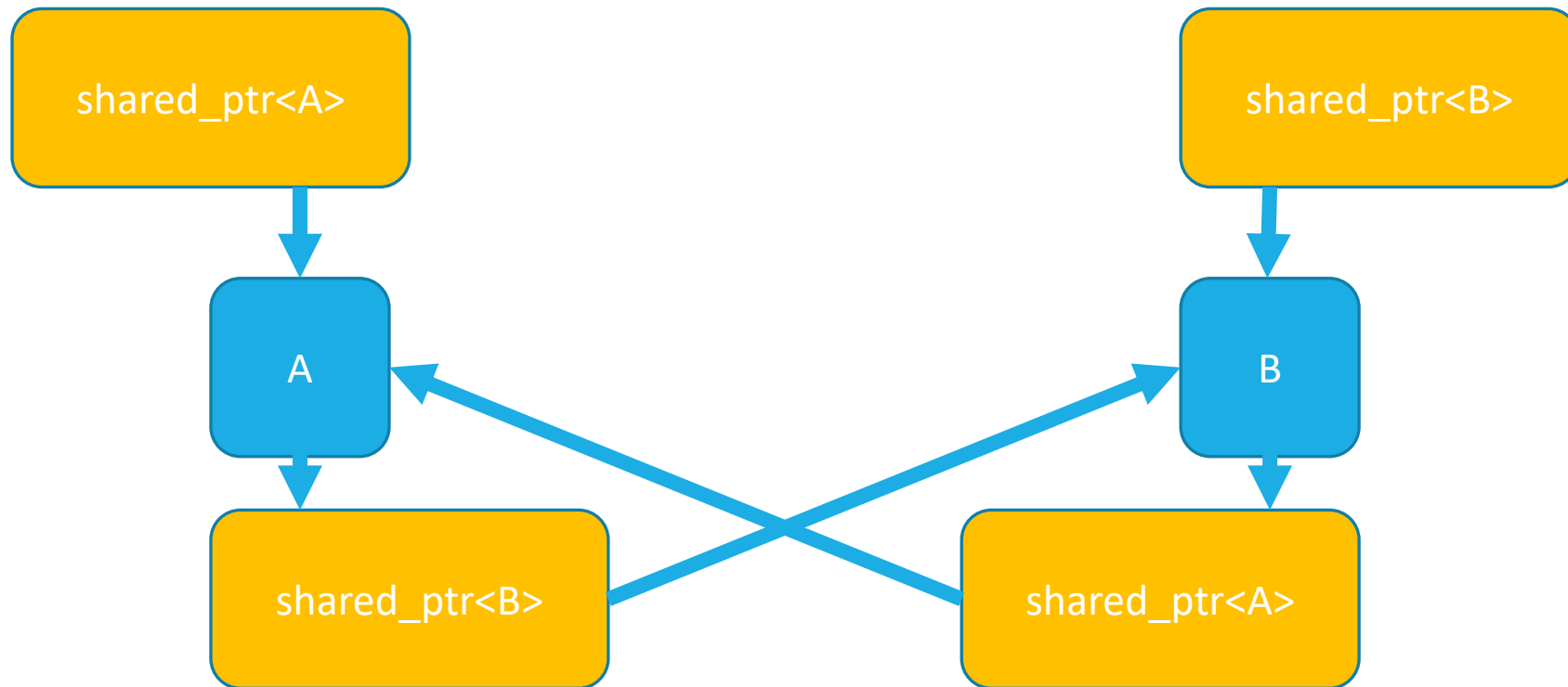
```
struct SomeStruct : std::enable_shared_from_this<SomeStruct> {
    SomeStruct() {
        std::cout << "ctor" << std::endl;
    }
    ~SomeStruct() {
        std::cout << "dtor" << std::endl;
    }

    std::shared_ptr<SomeStruct> getPtr() {
        return shared_from_this();
    }
};
```

Перекрестные ссылки и `std::shared_ptr`

`dead_lock.cpp`

Если зациклить объекты друг на друга, то появится «цикл» и объект ни когда не удалится! Т.к. деструктор не запустится!



Слабый указатель

`std::weak_ptr`

`shared_ptr` представляет *разделяемое владение*, но с моей точки зрения разделяемое владение не является идеальным вариантом: значительно лучше, когда у объекта есть конкретный владелец и его время жизни точно определено.

`std::weak_ptr`

1. Обеспечивает доступ к объекту, только когда он существует;
2. Может быть удален кем-то другим;
3. Содержит деструктор, вызываемый после его последнего использования (обычно для удаления анонимного участка памяти).

Теперь без dead lock

`weak_ptr_deadlock.cpp`

```
1.class A {  
2.private:  
3.    std::weak_ptr<B> b;  
4.public:  
5.    void LetsLock(std::shared_ptr<B> value) {  
6.        b = value;  
7.    }  
8.    ~A( ){  
9.        std::cout << "A killed!" << std::endl;  
10.    }  
11.};
```

weak_ptr.cpp

```
struct Observable {
    void registerObserver(const std::shared_ptr<Observer>& observer) {
        m_observers.emplace_back(observer);
    }

    void notify() {
        for (auto& obs : m_observers) {
            auto ptr = obs.lock();
            if (ptr)
                ptr->notify();
        }
    }
private:
    std::vector<std::weak_ptr<Observer>> m_observers;
};
```



Спасибо!

ВСЕ ИДЕМ НА ПЕРЕРЫВ