Умные указатели

ЛЕКЦИЯ №6

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**.

Пара слов о шаблонах (template)

подробнее на следующей лекции

Шабло́ны (англ. template) — средство языка C++, предназначенное для кодирования обобщённых алгоритмов, без привязки к некоторым параметрам (например, типам данных, размерам буферов, значениям по умолчанию).

Шаблоны позволяют создавать **параметризованные классы и функции**. Параметром может быть любой тип или значение одного из допустимых типов (целое число, enum, указатель на любой объект с глобально доступным именем, ссылка)

Использование шаблонов классов:

имя_шаблона<параметр_шаблона_1,параметр_шаблона2> имя_объекта

Шаблон std::shared_ptr<T>

`#include<memory>`

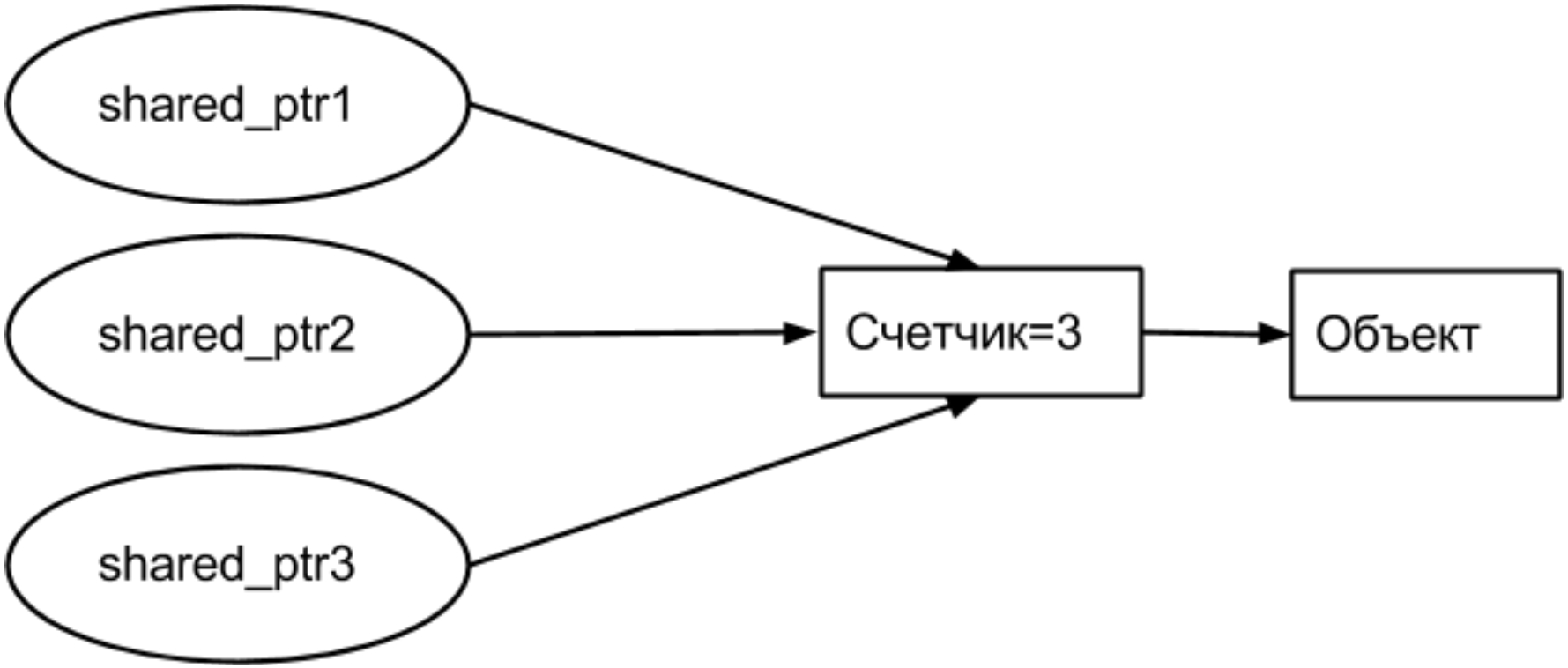
1. Предоставляет возможности по обеспечению автоматического удаления объекта, за счет подсчета ссылок указатели на объект;
2. Хранит ссылку на один объект;
3. При создании std::shared_ptr<T> счетчик ссылок на объект увеличивается;
4. При удалении std::shared_ptr<T> счетчик ссылок на объект уменьшается;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

SharedPtr
Example38_SharedPtr

```
1.void Foo(std::shared_ptr<A> a) { //shared_ptr передаем по значению, сам объект не копируется
2.    std::shared_ptr<A> b(a); // Копируем ссылку
3.    a->DoSomething(); // С shared_ptr мы можем работать как с обычным указателем
4.    b->DoSomething();
5.}

6.int main(int argc, char** argv) {
7.    std::shared_ptr<A> a(new A()); //В передаем указатель на нужный объект
8.    Foo(a);
9.    return 0;
10.}
```

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



Более реалистичный пример
Example39_SharedPtr2

```
1.class A {
2.private:
3.    std::shared_ptr<A> next;
4.public:
5.    A() {
6.        std::cout << "I'm alive!" << std::endl;
7.    }
8.    A(A* next_ptr) : A(){
9.        next = std::shared_ptr<A>(next_ptr);
10.    }
11.   virtual ~A() {
12.        std::cout << "O no! I'am dead!" << std::endl;
13.    }
14.};
15.int main(int argc, char** argv) {
16.    std::shared_ptr<A> a(new A(new A(new A(new A())))); // Ни один объект не потеряется
17.    return 0;}
```

Наследование работает как обычно

Example45_SharedPtrInheretance

```
1.class A {
2.private:
3. const char* name;
4.public:
5. A(const char*value) : name(value) {};
6.};
7.
8.class B : public  A {
9.public:
10. B(const char*value) : A(value) {};
11.};

12.int main(int argc, char** argv) {
13.std::shared_ptr<A> b(new B("My name is B!"));
14.b->Print();}
```

Swap – обмениваемся указателями
Example40_SharedPtr3

```
1.int main(int argc, char** argv) {
2.    std::shared_ptr<A> a(new A("My name is A"));
3.    std::shared_ptr<A> b(new A("My name is B"));
4.    std::swap(a,b);

5.//template <class T>
6.//void swap (shared_ptr<T>& x, shared_ptr<T>& y) noexcept;

7.    a->WhoAmI();
8.    b->WhoAmI();
9.    return 0;
10.}
```

Внешний деструктор
Example41_SharedPtr4

```
1.void deleter(A *a){
2.    std::cout << "Nobody kills in my ship!" << std::endl;
3.}

4.int main(int argc, char** argv) {
5.    A a("My name is A");
6.    std::shared_ptr<A> a_ptr(&a,&deleter); // Вместо деструктора вызывается наша функция
7.    return 0;
8.}
```

shared_ptr в списке параметров функций
Example48_SharedPtr5

```
1.try{
2.      //* Так плохо!
3.      A *a=new A("A");
4.      foo(foofoofoo(),std::shared_ptr<A>(a));
5.      /*/
6.      std::shared_ptr<A> a(new A("B"));
7.      foo(foofoofoo(),a);
8.      //*/
9.      }catch(...){
10.     }
```

std::dynamic_pointer_cast<T> example42_dynamicpointercast

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

std::shared_ptr<C> c(new C());
std::shared_ptr<A> ptr = b;


if(std::shared_ptr<B> ptr_b = std::dynamic_pointer_cast<B>(ptr)){

   ptr_b->Do();

}


if(std::shared_ptr<C> ptr_c = std::dynamic_pointer_cast<C>(ptr)){

   ptr_c->Do();

}
```

std::make_shared<T> example42_makeshared

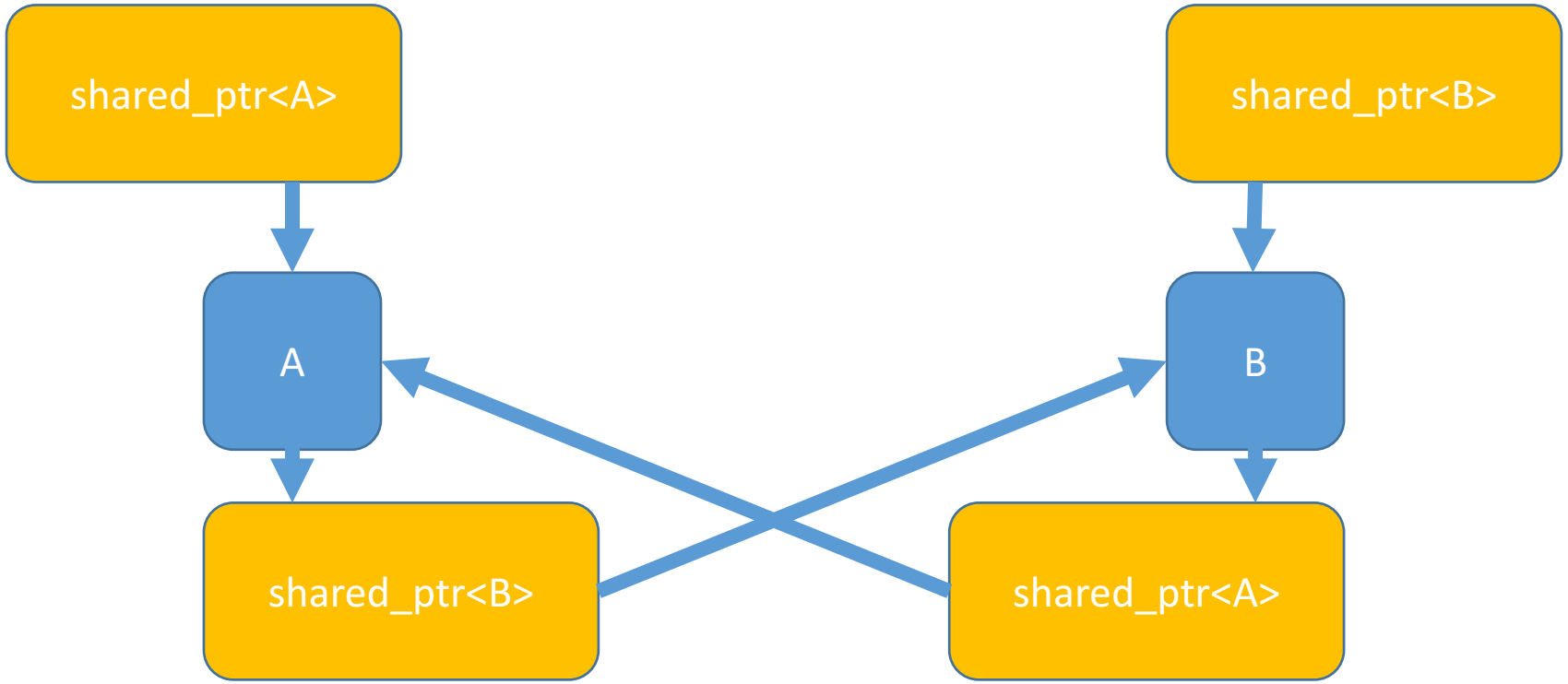
```
#include <iostream>
#include <memory>

int main () {

std::shared_ptr<int> foo = std::make_shared<int> (10);
// same as:
std::shared_ptr<int> foo2 (new int(10));
std::cout << "*foo: " << *foo << '\n';
std::cout << "*foo2: " << *foo2 << '\n';
return 0;
}
```

Перекрестные ссылки и std:shared_ptr
Example42_SharedPtrDeadlock

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.

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
Example43_Weak_Ptr

```
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.};
```

Объекты, не являющиеся владельцем Example44_Weak_Ptr2

```
1.class WeakPrint {
2.private:
3.    std::weak_ptr<A> array[5];
4.public:
5.    WeakPrint(std::shared_ptr<A> value[5]) {
6.        for (int i = 0; i < 5; i++)
7.            array[i] = value[i];
8.    }
9.    void Print() {
10.        for (int i = 0; i < 5; i++)
11.            if (std::shared_ptr<A> a = array[i].lock())
12.               a->Print();
13.    }
14.};
```

unique_ptr<T>

unique_ptr (определен в **<memory>**) и обеспечивает семантику строгого владения.

- Владеет объектом, на который хранит указатель.
- Не CopyConstructable и не CopyAssignable, однако MoveConstructible и MoveAssignable.
- При собственном удалении (например, при выходе из области видимости (6.7)) уничтожает объект (на который хранит указатель) с помощью заданного метода удаления (с помощью deleter-а).

Использование **unique_ptr** дает:

- безопасность исключений при работе с динамически выделенной памятью,
- передачу владения динамически выделенной памяти в функцию,
- возвращения динамически выделенной памяти из функции,
- хранение указателей в контейнерах

Исключительное право владения
Example46_UniquePtr

```
1.A* unsafe_function(){
2.     A* a = new A("I', only one!");
3.     return a;     }

4.std::unique_ptr<A> safe_function(){
5.     std::unique_ptr<A> a(new A("I', only one!"));
6.     return a;
7.}
```

Почему unique_ptr не копируется?
Example47_UniquePtr2

Единственные доступные операторы 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;};
```

Спасибо!

ВСЕ ИДЕМ НА ПЕРЕРЫВ