ОНЛАЙН-ОБРАЗОВАНИЕ

OTUS

Разработчик C++

Базовый курс

Умные указатели

Сергей Кольцов
профессиональный программист

Что нас ждёт

- область видимости
- время жизни
- проблемы управления ресурсами
- глупый умный указатель
- unique_ptr
- shared_ptr
- weak_ptr
- enable_shared_from_this

Область видимости


```
auto x = 42;

auto foo() {
      return x;
}

int main() {
      auto value = foo();
}
```

Область видимости


```
auto x = 42;

auto foo() {
      auto x = 10;
      return x;
}

int main() {
      auto value = foo();
}
```

Область видимости


```
auto x = 42;

auto foo() {
      auto x = 10;
      return x;
}

int main() {
      int x = 37;
      auto value = foo();
}
```

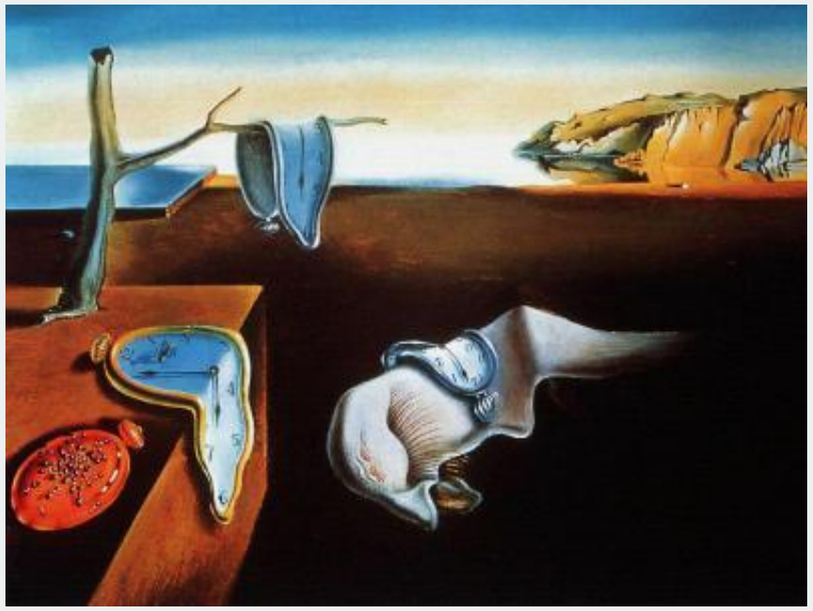
Область видимости

```
auto x = 42;
auto foo() {
      return x;
}
```


```
template<typename T>
auto bar(T y) {
      return x + y;
}

int main() {
      auto value = foo();
      auto value2 = bar(10);
}
```

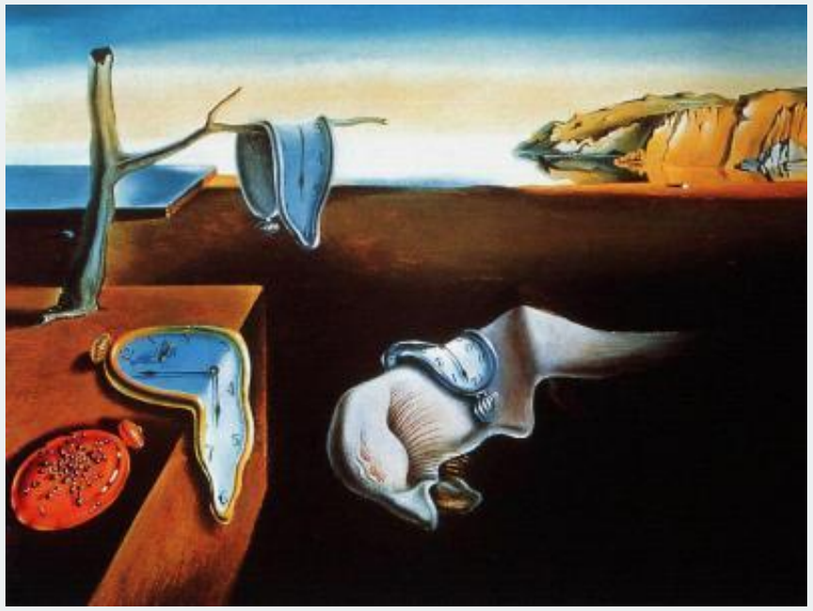
Время жизни



```
1. auto str_1 = std::string{};
2.
3. thread_local auto str_2 = std::string{};
4.
5. std::string foo(std::string str_3) {
6.    static std::string str_4{};
7.    return str_4;
8. }
9.
10. int main() {
11.   auto str_5 = "Local string";
12.   foo(std::string("argument"));
13.   auto str_6 = new std::string("new string");
14.   // ... long code here
15.   delete str_6;
16. }
```

3 **Не всегда область видимости и время жизни совпадают.**

Время жизни



Storage duration:

- статическое (static) – глобальные переменные и static
- потоковое – thread_local
- автоматическое – scope
- динамическое – new / delete, malloc / free

Сырые (raw) указатели

```
1.int main() {
2.    int a;
3.
4.    a = 10; // Не знаем адрес, да и ладно
5.
6.    int * ptr = &a;
7.    *ptr += 5; // Зачем-то знаем адрес, но не используем
8.
9.    int * ptr2 = &a;
10.   *(ptr+42) += 5; // Знаем адрес, но используем как-то неправильно
11.
12.   int& ref = a;
13.   ref += 3; // Не знаем адрес, но ссылаемся
14.}
```

Указатели – изящный способ выстрелить себе в ногу.

Сырые (raw) указатели

```
1.int main() {
2.     {
3.             int * ptr = new int{42};
4.     } // выход за scope - утечка
5.     {
6.             int value = 0;
7.             int * ptr = new int{50};
8.             // опять потеряли адрес - утечка
9.             ptr = &value;
10.    }
11.    {
12.            int * ptr = new int{79};
13.            // Утечка, если функция бросит исключение
14.            someFunctionHere();
15.            delete ptr;
16.    }
17.}
```

Указатели – изящный способ выстрелить себе в ногу.

Сырые (raw) указатели

- нет контроля создания / удаления
- может указывать в неизвестность, nullptr
- может указывать в известность, но чужую

Кастом

```
template<typename T>
struct smart_ptr {
	smart_ptr()
	: m_ptr{new T{}} {
	}
	~smart_ptr() {
		delete m_ptr;
	}

	T* get() {
		return m_ptr;
	}
private:
	T* m_ptr;
};
```

Хочешь сделать что-то хорошо – сделай это сам.

unique_ptr

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

```cpp
auto ptr2 = std::make_unique<int>(42);
```

```cpp
int * init();
void deinit(int *);

std::unique_ptr<int, decltype(deinit)>
      ptr{init(), &deinit};
```


Свой удалитель

```
struct Deleter {
      void operator()(int * ptr) {
            deinit(ptr);
      }
};
std::unique_ptr<int, Deleter> ptr4{init()};
```

```
std::unique_ptr<int, std::function<void(int*)>>
      ptr5{init(), [](int *ptr) {deinit(ptr);}};


auto deleter = [](int * ptr) { deinit(ptr); };
std::unique_ptr<int, decltype(deleter)> ptr6{init(), deleter};
```


Особые объекты требуют особого уничтожения.

unique_ptr

- нераздельное владение объектом
- нельзя копировать (только перемещение)
- размер зависит от пользовательского deleter-а
- без особой логики удаления издержки чаще отсутствуют
- std::make_unique – только в качестве сахара

Ещё больше кастома

```
1. template<typename T>
2. struct smart_ptr {
3.     smart_ptr(T* ptr)
4.       : m_counter{new std::size_t{1}}, m_ptr{ptr} {
5.     }
6.     smart_ptr(const smart_ptr& other)
7.     : m_counter{ other.m_counter }, m_ptr{ other.m_ptr } {
8.           ++*m_counter;
9.     }
10.    ~smart_ptr() {
11.          if (--*m_counter == 0) {
12.                delete(m_ptr);
13.                delete(m_counter);
14.          }
15.    }
16.private:
17.    T* m_ptr;
18.    std::size_t* m_counter;
19.};
```

shared_ptr

- можно копировать с разделением владения
- но дешевле перемещать
- всегда внутри два указателя
- std::make_shared – не только ценный мех
- потокобезопасный (и хорошо, и плохо)
- можно создать из unique_ptr

Двойное удаление

```
1.int * ptr = new int{42};
2.
3.{
4.    std::shared_ptr<int> smartPtr1{ptr};
5.
6.    std::shared_ptr<int> smartPtr2{ptr};
7.
8.} // ooops. double delete here
```

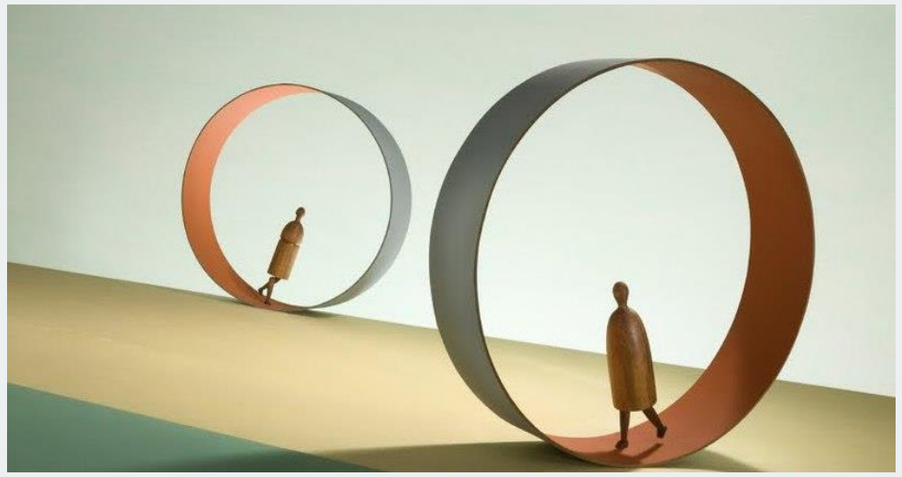
enable_shared_from_this

```
1. struct SomeStruct
2.   : std::enable_shared_from_this<SomeStruct> {
3.     SomeStruct() {
4.         std::cout << "ctor" << std::endl;
5.     }
6.     ~SomeStruct() {
7.         std::cout << "dtor" << std::endl;
8.     }
9.
10.    std::shared_ptr<SomeStruct> getPtr() {
11.        return shared_from_this();
12.    }
13. };
```

weak_ptr

```
1. int main() {
2.
3.     std::weak_ptr<int> weak;
4.     {
5.         auto shared = std::make_shared<int>(42);
6.
7.         weak = shared;
8.
9.         auto x = weak.lock();
10.        assert(x);
11.    }
12.    auto x = weak.lock();
13.    assert(!x);
14.
15.    return 0;
16.}
```

Циклические ссылки



```
1.struct SomeStruct {
2.     ~SomeStruct() {
3.           std::cout << "Desctructor!" << std::endl;
4.     }
5.
6.     std::shared_ptr<SomeStruct> partner;
7.};
8.
9.void test_func() {
10.    auto partnerA = std::make_shared<SomeStruct>();
11.    auto partnerB = std::make_shared<SomeStruct>();
12.    partnerA->partner = partnerB;
13.    partnerB->partner = partnerA;
14.
15.}
```

Спасибо
за внимание!

Заполните, пожалуйста
опрос о занятии.

Ответы на вопросы