ЖИЗНЬ ОБЪЕКТА

Жизнь, смерть и тление объектов в C++

К. Владимиров, Syntacore, 2022
mail-to: konstantin.vladimirov@gmail.com

- Физическая организация кода
- Область видимости и время жизни
- Перегрузка и преобразования
- Пространства имён и поиск имён

Объявления и определения

- Что написано ниже в каждом случае?

```
int x;
struct S;
extern int foo();
extern S *ps;
int bar { return foo(); }
struct T { int x; };
extern int y = 0;
```

Объявления и определения

- Что написано ниже в каждом случае?

```
int x; // объявление и определение переменной
struct S; // объявление типа
extern void foo() {} // объявление и определение функции
extern S *ps; // объявление переменной
int bar(); // объявление функции
struct T { int x; }; // объявление и определение типа
extern int y = 0; // объявление и определение переменной
```

Единица трансляции

- Файл main.c

```
int printf(const char*, ...);
int fact(int x);

int main() {
  int x = fact(5);
  printf("%d\n", x);
}
```

- Файл fact.c

```
int fact(int x) {
  int res = 1;
  while (x-- > 1) {
    res *= x;
  }
  return res;
}
```

- Раздельная трансляция позволяет в том числе переиспользовать fact.c и распространять его в пре-транслированном виде (как библиотеку).

Проблема расхождения

- Файл main.c

```
int printf(const char*, ...);
int fact(double x);

int main() {
  int x = fact(5);
  printf("%d\n", x);
}
```

- Файл fact.c

```
int fact(int x) {
  int res = 1;
  while (x-- > 1) {
    res *= x;
  }
  return res;
}
```

- Защитой является вынесение объявлений функций в заголовочные файлы.

Внешнее связывание

- Файл main.c

```
#include <stdio.h>
#include "fact.h"

int main() {
  int x = fact(5);
  printf("%d\n", x);
}
```

- Файл fact.c

```
static int fact(int x) {
  int res = 1;
  while (x-- > 1) {
    res *= x;
  }
  return res;
}
```

- Внутри единицы трансляции мы можем делать вещи невидимыми извне её то есть задавать класс связывания: extern или static.

Классы связывания

- Какие классы связывания у сущностей ниже?

```
extern int x;
int y;
static int z;
extern int foo();
int bar();
static int buz();
struct S { static int x; };
```

Классы связывания

- Какие классы связывания у сущностей ниже?

```
extern int x; // внешнее
int y; // внешнее
static int z; // внутреннее
extern int foo(); // внешнее
int bar(); // внешнее
static int buz(); // внутреннее
struct S { static int x; }; // внешнее (!)
```

Проблемы связывания

- Файл user.c

```
extern int g;

int foo();

void inc() {
  g += foo();
}
```

- Файл first.c

```
int g = 5;

int foo() {
  return 42;
}
```

- Файл second.c

```
int g = 14;

int foo() {
  return 45;
}
```

- Если определение встречается более одного раза, то неизбежны проблемы.
- К счастью у нас есть ODR.

One Definition Rule

- No translation unit shall contain more than one definition of any variable, function, class type, enumeration type, template [...].
- Every program shall contain exactly one definition of every non-inline function or variable that is odr-used in that program outside of a discarded statement.

```
// header.h

#pragma once

int x; // потенциальное нарушение ODR

int foo(int n) { return n; } // потенциальное нарушение ODR

struct S { int x; }; // всё хорошо
```

Дополнительное условие на типы

- Типы с одним именем должны полексемно совпадать.

```
// header.h
struct S {
  int x;
#if defined(MYDEF)
  int y;
#endif
};
```

- Такой вариант очень чреват недиагностируемыми проблемами.

```
// ---- src1.cc ----
#define MYDEF
#include "header.h"

int foo(S *s) {
  return s->y;
}

// ---- src2.cc ----
#undef MYDEF
#include "header.h"

int main() {
  S s = {1}; foo(&s);
}
```

Инлайн

- Every program shall contain exactly one definition of every non-inline function or variable that is odr-used in that program outside of a discarded statement.
- Это исключение порождает большую разницу.

```
// header.h

#pragma once

inline int foo(int n) { return n; } // ok, исключение из ODR

static int foo(int n) { return n; } // ok, multiple defs
```

- Ключевое слово static означает новое определение на каждый TU.

- Физическая организация кода
  - Область видимости и время жизни
- Перегрузка и преобразования
- Пространства имён и поиск имён

Область видимости

- У любого имени есть область видимости (scope): совокупность всех **мест** в программе, откуда к нему можно обратиться.

```
int a = 2;

void foo() {
  int b = a + 3; // ok, we are in scope of a

  if (b > 5) {
    int c = (a + b) / 2; // ok we are in scope of a and b
  }

  b += c; // compilation fail
}
```

Время жизни

- У любой переменной есть время жизни (lifetime): совокупность всех **моментов времени** в программе, когда её состояние **валидно**.
- Первый такой момент случается после окончания инициализации.

```
int main() {
  int a = a; // a declared, but lifetime of "a" not started
```

- Это довольно редкий пример, когда мы пытаемся использовать нечто до его рождения.
- Куда более часто мы будем пытаться использовать нечто после его смерти.

Провисшие указатели

- Указатель, ссылающийся на переменную с истекшим временем жизни называется провисшим (dangling)

```
int a = 2;

void foo() {
  int b = a + 3; int *pc;

  if (b > 5) {
    int c = (a + b) / 2; pc = &c;
  } // c scope end; c lifetime end; pc dangles

  b += *pc; // this is parrot no more
} // b scope end; b lifetime end;
```

Провисшие ссылки

- Сделать висячую ссылку чуть сложнее, чем указатель, но можно
- Классика: ссылка внутрь удалённой памяти

```
int *p = new int[5];

int &x = p[3];

delete [] p; // x dangles
```

- Сама по себе провисшая ссылка ничего не значит. Проблемы будут только если по ней куда-то обратятся

```
x += 1; // it ceased to be
```

Провисшие ссылки

- Сделать висячую ссылку чуть сложнее, чем указатель, но можно
- Ещё классика: вернуть ссылку на временное значение

```
int& foo() {
  int x = 42;
  return x;
}

int x = foo(); // it expired and gone
```

- Компиляторы довольно плохи в диагностике провисших ссылок и указателей

Продление жизни

- Константные (и только они) lvalue ссылки продлевают жизнь временных объектов

```
const int &lx = 0;
int x = lx; // ok

int foo();

const int &ly = 42 + foo();
int y = ly; // ok
```

- Но не стоит соблазняться. Ссылка связывается со значением, а не со ссылкой, так что константная ссылка тоже может провиснуть при возврате из функции

Жизнь временных объектов

- Временный объект живёт до конца полного выражения

```
struct S {
  int x;
  const int &y;
};

S x{1, 2}; // ok, lifetime extended

S *p = new S{1, 2}; // this is a late parrot
```

- На первой строчке у нас не временный, а постоянный объект
- На второй будет висячая ссылка потому что временный объект продлявший жизнь константе закончился в конце выражения

Иногда временный объект не создаётся

- Неконстантные левые ссылки не создают временных объектов и просто отказываются связываться с литералами

```
int foo(int &x);

foo(1); // ошибка компиляции
```

- И даже проще

```
int &x = 1; // ошибка компиляции
```

- И это одна из лучших новостей в этой части лекции
- Попробуйте догадаться отчего так сделано

Decaying

```
int foo(const int& t) {
  return t;
}
```

- Ссылка на объект в выражениях ведёт себя как сам объект
- Мы это где-то встречали

Decaying

- Массив деградирует (decays) к указателю на свой первый элемент, когда он использован как rvalue

```
void foo(int *);

int arr[5];
int *t = arr + 3; // ok
foo(arr); // ok
arr = t; // fail
```

- Все ли помнят чем отличается lvalue от rvalue?

Lvalue & rvalue

- В языке C концепция lvalue означала "left-hand-side value"

```
y = x;
```

- Здесь y это lvalue, x это rvalue
- В языке C можно отделить синтаксически: вызов функции, имя массива, выражение сложения – всё это никогда не lvalue и технически не может встретиться в присваивании слева
- Так ли это в C++?

Lvalue & rvalue

- В языке C концепция lvalue означала "left-hand-side value"

```
y = x;
```

- Здесь y это lvalue, x это rvalue
- В языке C можно отделить синтаксически: вызов функции, имя массива, выражение сложения – всё это никогда не lvalue и технически не может встретиться в присваивании слева
- Увы, C++ усложняет вещи

```
int& foo();
foo() = x; // ok
```

Lvalue & rvalue

- В языке C++ lvalue это скорее "location value" – в смысле что-то у чего есть положение (location) в памяти
- В языке C++11 также есть более точный термин glvalue объединяющий положения с временными положениями, мы поговорим о нём на лекции по rvalue ссылкам
- Ссылки рассматриваемые здесь это lvalue ссылки
- Технически может существовать lvalue ссылка на массив. Это происходит именно потому, что, хотя массив и не может быть слева в присваивании, но он всегда lvalue в C++ потому что у него всегда есть локация (сам массив это локация по определению)

- Физическая организация кода
- Область видимости и время жизни
  - Перегрузка и преобразования
- Пространства имён и поиск имён

Одна забавная странность в языке C

- Функция `strstr(haystack, needle)` ищет подстроку needle в строке haystack
- Она определена мягко скажем странно

```
char *strstr(const char* str, const char* substr);
```

- Почему аргументы const?

- Почему если оба аргумента const, результат non-const?

https://en.cppreference.com/w/c/string/byte/strstr

Одна забавная странность в языке C

- Функция `strstr(haystack, needle)` ищет подстроку needle в строке haystack
- Она определена мягко скажем странно

```
char *strstr(const char* str, const char* substr);
```

- Почему аргументы const?
  - Потому что иначе не будет работать передача const строк
- Почему если оба аргумента const, результат non-const?
  - Потому что иначе не будет работать возврат non-const строк
- При этом мы сознательно жертвуем возвратом const строк. Омерзительно.

https://en.cppreference.com/w/c/string/byte/strstr

Обсуждение

- Как решить эту проблему?

Обсуждение

- Как решить эту проблему?
- Пункт первый: разрешить в языке перегрузку функций
- Пункт второй: перегрузить функции

```
char const * strstr(char const * str, char const * target);

char * strstr(char * str, char const * target);
```

- Теперь константность первого аргумента правильно согласована с константностью результата
- Так и сделано в C++. Увы, этого нельзя сделать в C

https://en.cppreference.com/w/cpp/string/byte/strstr

Гарантии по именам

- Язык C предоставляет строгие гарантии по именам

```
double sqrt(double); // метка не будет зависеть от сигнатуры
```

- Язык C++ не даёт гарантий по именам

```
double sqrt(double); // метка может зависеть от сигнатуры
```

- Кроме случая extern "C"

```
extern "C" double sqrt(double); // то же что и в C
```

- Последний случай введён чтобы согласовать API
- Процесс искажения имён называется манглированием

mangle.c, mangle.cc

Обсуждение

- Догадайтесь можно ли делать вот так:

```
extern "C" template <typename T> void foo(T x); // 1
struct S { extern "C" void foo(); };            // 2
```

- Обоснуйте свои догадки

Обсуждение

- Догадайтесь можно ли делать вот так:

```
extern "C" template <typename T> void foo(T x); // 1
struct S { extern "C" void foo(); };            // 2
```

- Обоснуйте свои догадки
- Оба ответа: нельзя
- Оба механизма с первой лекции: (1) обобщение данных и (2) объединение данных с методами невозможны без манглирования имён
- Точно так же перегрузка функций невозможна без манглирования имён

Разрешение перегрузки

- Наличие перегрузки вносит некоторые сложности

```
float sqrtf(float x);  // 1
double sqrt(double x); // 2
sqrtf(42); // вызовет 1, неявно преобразует int → float
```

- В языке C нет перегрузки и нет проблем, программист всегда явно указывает какую функцию нужно вызвать
- В языке C есть строгие гарантии по именам. Нельзя сказать, что в C нет манглирования. Оно может и быть. Чего в C нет так это манглирования типом.

Разрешение перегрузки

- Наличие перегрузки вносит некоторые сложности

```
float sqrt(float x);   // 1
double sqrt(double x); // 2
sqrt(42); // неясно что вызвать, оба варианта подходят
```

- В языке C++ есть перегрузка и компилятор должен разрешить имя, то есть связать упомянутое в коде имя с обозначаемой им сущностью
- В коде выше как по вашему будет сделан вызов и почему?

Разрешение перегрузки

- Наличие перегрузки вносит некоторые сложности

```
float sqrt(float x);   // 1
double sqrt(double x); // 2
sqrt(42); // неясно что вызвать, оба варианта подходят
```

- В языке C++ есть перегрузка и компилятор должен разрешить имя, то есть связать упомянутое в коде имя с обозначаемой им сущностью
- В коде выше как по вашему будет сделан вызов и почему?
- Разумеется будет ошибка компиляции. Оба варианта одинаково хороши

Правила разрешения перегрузки

- Первое приближение (здесь много чего не хватает)

1. Точное совпадение (int → int, int → const int&, etc)
   Обратите внимание: nullptr точно совпадает с любым указателем.
2. Точное совпадение с шаблоном (int → T)
3. Стандартные преобразования (int → char, float → unsigned short, etc)
4. Переменное число аргументов
5. Неправильно связанные ссылки (literal → int&, etc)

- Мы вернёмся к перегрузке когда подробнее поговорим о шаблонах функций

Обсуждение: nullptr

- Теперь мы можем дополнительно аргументировать почему наш выбор nullptr.

```
int foo(int *);

// call for null pointer
foo(0);
```

- В чём хрупкость этой конструкции?
- Как nullptr это исправляет?

Перегрузка конструкторов

- Методы класса, разумеется, тоже можно перегружать и наиболее полезно это для конструкторов

```
class line_t {
  float a_ = -1.0f, b_ = 1.0f, c_ = 0.0f;

public:
  // по умолчанию
  line_t() {}

  // из двух точек
  line_t(const point_t &p1, const point_t &p2);

  // явные параметры линии
  line_t(float a, float b, float c);
```

- Физическая организация кода
- Область видимости и время жизни
- Перегрузка и преобразования
- Пространства имён и поиск имён

Коротко о пространствах имён

- Любое имя принадлежит к какому-то пространству имён

```
// no namespace here

int x;

int foo() {
  return ::x;
}
```

- Здесь кажется, что x не принадлежит ни к какому пространству имён
- Но на самом деле x принадлежит к глобальному пространству имён

Пространство имён std

- Вся стандартная библиотека принадлежит к пространству имён std

`std::vector`, `std::string`, `std::sort`, ….

- Исключение это старые хедера наследованные от C, такие, как `<stdlib.h>`
- Чтобы завернуть `atoi` в `std`, сделаны новые хедера вида `<cstdlib>`
- Вы не имеете права добавлять в стандартное пространство имён свои имена
- Точно по той же причине по какой вы не можете начинать свои имена с подчёркивания и большой буквы

Ваши пространства имён

- Вы можете вводить свои пространства имён и неограниченно вкладывать их друг в друга
- При том структуры тоже вводят пространства имён

```
namespace Containers {
  struct List {
    struct Node {
      // .... whatever ....
    };
  };
}

Containers::List::Node n;
```

Переоткрытие пространств имён

- В отличии от структур, пространства имён могут быть переоткрыты

```
namespace X {
  int foo();
}

// теперь переоткроем и добавим туда bar
namespace X {
  int bar();
}
```

- Структура вводит **тип данных**. Тип не должен существовать если в программе не будет его объектов
- Для пространств имён куда удобнее (сюрприз) пространства имён

Директива using, второй смысл

- Мы можем вводить отдельные имена и даже целые пространства имён

```
namespace X {
  int foo();
}

using std::vector;

using namespace X;

vector<int> v; v.push_back(foo());
```

- Использовать эти механизмы следует осторожно так как пространства имён придуманы не просто так

Анонимные пространства имён

- Это распространённый механизм для замены статических функций

```
namespace {

int foo() {
  return 42;
}

}

int bar() { return foo(); } // ok!
```

- Означает сделать пространство имён со сложным уникальным именем и тут же сделать его using namespace

Анонимные пространства имён

- Это распространённый механизм для замены статических функций

```
namespace IdFgghbjhbklbkuU6 {

int foo() {
  return 42;
}

}

using namespace IdFgghbjhbklbkuU6;

int bar() { return foo(); } // ok!
```

- Поскольку имена из него не видны снаружи они как бы статические

Правила хорошего тона

- Не засорять глобальное пространство имён
- Никогда не писать using namespace в заголовочных файлах
- Использовать анонимные пространства имён вместо статических функций
- Не использовать анонимные пространства имён в заголовочных файлах

Правильный hello world

```
#include <iostream>

namespace {
  const char * const helloworld = "Hello, world";
}

int main() {
  std::cout << helloworld << std::endl;
}
```

- Обратите внимание: функция main обязана быть в глобальном пространстве имён

Литература

- [CC11] ISO/IEC 14882 – "Information technology – Programming languages – C++", 2011
- [BS] Bjarne Stroustrup – The C++ Programming Language (4th Edition) , 2013
- [GB] Grady Booch – Object-Oriented Analysis and Design with Applications, 2007
- [GCT] Eberly, Schneider – Geometric Tools for Computer Graphics, 2002
- [GS] Gilbert Strang – Introduction to Linear Algebra, Fifth Edition, 2016
- [BB] Ben Saks – Back to Basics: Pointers and Memory, CppCon, 2020

Обсуждение

- Есть ли реальная возможность скрыть класс, являющийся деталью реализации?

```
class Detail { /* .... */ };

class Entity {
  Detail d;

public:
  // ....
};
```