Лекция 3

Livetime & scope

Хак с delete

`malloc, new[]` при выделении памяти по указателю хранят до указателя число - количество выделенных ячеек памяти. Она используется функциями `free, delete[]`. *А что будет, если мы на аллокатор вызовем неправильный деаллокатор? Это UB (например, вызов большего числа деструкторов, чем нужно).*

Это продемонстрировано в newdelete.cc.

> **Как перестраховать себя от путаницы?**
>
> Выделять память в конструкторах, а их чистку - в деструкторах. Или использовать умные указатели.

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; // ок, we are in scope of a
    if (b > 5) {
        int c = (a + ь) / 2; // ок не 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 (это UB). **Декларация (`int a`) заканчивается после первого инициализатора (`=`). Время жизни начинается после всех инициализаторов (после `;`).**
Куда более часто мы будем пытаться использовать нечто после его смерти.

Провисшие ссылки и указатели

Провисшие ссылки и указатели - это очень опасная ошибка.

Константные ссылки умеют продлевать жизнь временным объектам.

```
const int &lx = 0; // сформирует объект на стеке
int x = lx; // ok (no deadness)

int &ref = 10; // error, 10 has no mem loc, but lval ref has
const int &cref = 10; // ok, it increases livetime of tmp object (10)

// int foo();
const int &ly = 42 + foo();
int y = ly; // ok (no deadness)

// Продлевание жизни работает не всегда.
/* Не стоит соблазняться.
* Ссылка связывается со значением, а не со ссылкой,
* так что константная ссылка тоже может провиснуть при возврате из функции.
*/
// const int& goo();
const int &ly = 42 + goo(); // UB
```

Временные объекты живут до конца полного выражения (например, до `;`).

```
struct S {
    int x;
    const int &y;
};
S x{1, 2}; // ок, lifetime extended
S *p new S{1, 2}; // this is a late parrot // 2nd field id dead.

```

На 5-й строчке у нас не временный, а постоянный объект.
На 6-й будет висячая ссылка потому что временный объект, продлявший жизнь константе, закончился в конце выражения.

> **Замечание:**
>
> Не использовать в классах членах ссылок! Это потенциальные проблемы.

Decaying

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

```
void foo(int *);

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

Аналогично: `f(const T& arg)` - аргумент деградирует к объекту.

Манглирование

Гарантия по именам есть только в C. В C++ для возможности перегрузок, пространств имен и прочего используется **манглирование**.

> В манглировании не участвует возвращаемый тип.

Отключить интерфейс можно с помощью `extern "C"`

```
extern "C" int foo(int);
```

Правила разрешения перегрузок

1. Точное совпадение (+ `int --> const int&`, ...)?
2. Точное совпадение с шаблоном (int --> T) или случай SFINAE?
3. Совпадение при стандартных преобразованиях типов?
4. Совпадение с пользовательским преобразованием?
5. Переменное число аргументов?
6. Неправильно связанные ссылки (literal --> int&, ...)?

В цепочке может быть ровно одно пользовательское преобразование.

Подробнее: reference manual: implicit cast

Если есть ровно один кандидат под какой-то пункт (при этом предыдущие пункты уже пройдены по всем проверкам, борьба до него дошла), то он побеждает. Если два или более - ошибка компиляции. Иначе борьба продолжается для следующих пунктов.

В файле overload.cc конфликт будет возникать в том случае, если на перегрузку будут претендовать **одинаковые по правам** функции.

Перегрузка конструкторов

```
class A {
    int m_x{0};
public:
    A::A(int x = 0) : m_x(x) {/* ... */} // хороший тон, если одно поле (более локально и меньше строк кода)
}
```

namespace

Структуры и классы тоже определяют пространство имен.
Пространство имен есть **всегда** (глобальное пространство имен ::)

```
int x;

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

На макросы пространство имен не распространяются.
Заголовочники для поддержки пространства имен в стандартной библиотеке были переписаны.

```cpp
#include <cstdio> // std::atoi();
#include "stdio.h" // atoi();
```

В простарнство имен можно запихивать из другого:

```cpp
namespace X {
    int foo();
    using std::vector;
}
```

```cpp
#include <iostream>
namespace Y {
    int y = 1;
}
namespace X {
    using Y::y;
    int x = 0;
}
int main() {
    std::cout << X::x + X::y << std::endl;
}
```

Анонимные пространства имен

```cpp
namespace {
    int a;
    int foo()
}
...
```

преобразуется в:

```cpp
namespace pdvinpvom {
    ...
}
using pdvinpvom;
```

что эквивалентно использованию `static`

```cpp
static int a;
static int foo();
```

- Не засорять глобальное пространтсво имен;
- Никогда не писать using namespace в заголовочных файлах;
- Не использовать анонимные пространства имен в заголовочных файлах;

C Notes

Cdecl

Правило чтения:

Вправо --> влево --> вверх (из скобок)

```
int *x[20]; // массив указателей
int (*y)[20] // указатель на массив
int (&z)[20] = *y; // ссылка на массив
```

Можно шагать большими интервалами по памяти, если сделать массив из указателей на массивы произвольной фиксированной длины.

```
// ссылка на массив из 10-ти указателей на функцию, принимающую ссылку на указатель и возвращающую указатель на char.
char *(*(&c)[10])(int *&p);

// используем typedef / using
void (*bar(int x, void (*func)(int&))) (int&);

typedef void (*ptr_to_fref) (int&);
ptr_to_fref bar(int x, ptr_to_ref func);
using ptr_to_fref = void (*) (int&);
ptr_to_fref bar(int x, ptr_to_fref func);
```

> Лучше использовать `using`, т.к. он поддерживает шаблоны.