СЕМИНАР III

ОШИБКИ С ПРОШЛОГО СЕМИНАРА

- алгоритмы с квадратичным временем
- popZeros — нужно делать inplace
- int вместо size_t
- 0, NULL, nullptr
- адресная арифметика за пределами массива
- ladderCombinations - энергичные вычисления вместо ленивых

strcmp

strcmp

strcmp

strcmp

`findNearestSame`

- строка может быть оооочень большой; искать сперва в одну сторону, потом в другую - что может пойти не так?
- указатель `s-1` невалидный, тонкие грабли для оптимизирующего компилятора

countNumbers

countNumbers

- `int counters[10] = {};` - что здесь принципиально не так?

countNumbers

- `int counters[10] = {};` - что здесь принципиально не так?
- `bool counters[10]`

countNumbers

- `int counters[10] = {};` - что здесь принципиально не так?
- `bool counters[10]`
- `sum(counters, 10)`

countNumbers

- `int counters[10] = {};` - что здесь принципиально не так?
- `bool counters[10]`
- `sum(counters, 10)`
- можно ли не бежать по всему массиву?

popZeros

popZeros

- сложное логическое выражение `a[i] != nullptr && a[i] == 0` - можно ли уменьшить копи-пасту?

popZeros

- сложное логическое выражение `a[i] != nullptr && a[i] == 0` - можно ли уменьшить копи-пасту?
- `bool isZero(const_int_pointer elem)`

ВЫРОЖДЕННЫЕ СЛУЧАИ

Что здесь не так?

```
size_t first_zero_item = s - 1;
while (first_zero_item >= 0 && isZero(a[i])) { ..... }
```

УЛЬТРА-ОПТИМИЗАЦИЯ

```
for (auto ptr = a; ptr < a + (s - 1); ++ptr) { ..... }

// пусть s = 0
auto s1 = s - 1;    // s1 = 0xFFFF....FFFF
for (auto ptr = a; ptr < a + s1; ++ptr) { ..... }
for (auto ptr = a; true; ++ptr) { ..... }
```

0, NULL, nullptr

- NULL - сишный дефайн, означает `(void*)0`, `0`, `__null` в зависимости от Си/C++ и компилятора
- `auto x = NULL` - это int или long
- `auto y = NULL + 1`, компилятор выдаст варнинг (в лучшем случае)
- nullptr - имеет особый тип nullptr_t, приводимый только к указателям!

ladderCombinations

```
uint64_t ladderCombinations(uint8_t steps) {
    static uint64_t res[21] = {};
    res[0] = 1;
    res[1] = 1;
    for (int i = 2; i <= steps; ++i) {
        res[i] = res[i-1] + res[i-2];
    }
    return res[steps];
}
```

Как сделать ленивой?

ladderCombinations

```
uint64_t ladderCombinations(uint8_t steps) {
    static uint64_t res[21] = {};
    res[0] = 1;
    res[1] = 1;
    for (int i = 2; i <= steps; ++i) {
        res[i] = res[i-1] + res[i-2];
    }
    return res[steps];
}
```

Как сделать ленивой?

ladderCombinations

```
uint64_t ladderCombinations(uint8_t steps) {
    static uint64_t res[21] = {};
    res[0] = 1;
    res[1] = 1;
    for (int i = 2; i <= steps; ++i) {
        res[i] = res[i-1] + res[i-2];
    }
    return res[steps];
}
```

Как сделать ленивой?

ИСПРАВЛЕННАЯ РЕАЛИЗАЦИЯ

```
uint64_t ladderCombinations(uint8_t steps) {
    static uint64_t res[21] = {1, 1};
    static uint8_t evaluated = 1;

    if (evaluated < steps) {
        for (uint8_t i = evaluated+1; i <= steps; ++i) {
            res[i] = res[i-1] + res[i-2];
        }
        evaluated = steps;
    }
    return res[steps];
}
```

НЕМНОГО ПРО EXTRACTEXPONENT

Если делать по феншую, следует учесть, что

- float может быть не 4 байтным (видимо, стоит сообщать об ошибке в виде асерта?)
- на системе может быть различный endianness
- требование к выравниванию типа, используемого при `reinterpret_cast` может отличаться от выравнивания `float`

СЕГОДНЯ НА ЗАНЯТИИ

- ссылки
- аргументы функции
- динамическая память
- структуры
- inline trick

ССЫЛКИ

```
int i = 3;
int &r = i;
```

ССЫЛКИ

```
int i = 3;
int &r = i;
```

- `sizeof(r)`?

ССЫЛКИ

```
int i = 3;
int &r = i;
```

- `sizeof(r)`?
- → возвратит размер типа за ссылкой == `sizeof(int)`

ССЫЛКИ

```
int i = 3;
int &r = i;
```

- `sizeof(r)`?
- → возвратит размер типа за ссылкой == `sizeof(int)`
- в то же время "передача по ссылке" — "легковесная"

В чем отличие ссылки от указателя?

В чем отличие ссылки от указателя?

- нет nullptr (и аналогов)

В чем отличие ссылки от указателя?

- нет nullptr (и аналогов)
- ссылку нельзя переназначить

```
int i = 3;
int &r = i;
int *p = &i;
```

```
int i = 3;
int &r = i;
int *p = &i;
```

- `i` и `r` обозначают один и тот же участок памяти

```
int i = 3;
int &r = i;
int *p = &i;
```

- `i` и `r` обозначают один и тот же участок памяти
- `p` — не факт, что в течение всего выполнения указывает на `i`

```
int i = 3;
int &r = i;
int *p = &i;
```

- `i` и `r` обозначают один и тот же участок памяти
- `p` — не факт, что в течение всего выполнения указывает на `i`
- ссылка *может быть* реализована через указатель (в конкретном случае)

DANGLING REFERENCE

```
int& foo() {
    int n = 10;
    return n;
}

int& i = foo();
```

- в чем проблема?
- как починить?

ПЕРЕДАЧА АРГУМЕНТОВ В ФУНКЦИЮ

- На лекции: по значению, по ссылке, по указателю
- Зачем передавать не по значению?

- изменить аргумент

```
void changeArg(int& a) { a += 24; };

int i = 42;

changeArg(i);
assert(i == 66);
```

- избежать копирования == передать меньше данных

```
// хранит длину + данные строки + ...
using string_t = std::string;

// передаем по адресу => 4/8 байт
void argPassedByRef(const string_t& longString);
```

ПРИМЕРЫ

Как передаются аргументы?

```
void foo(int);

int i = 42;

foo(i);        // (1)
foo(24);       // (2)
foo(i + 24);   // (3)
```

ПРИМЕРЫ

Как передаются аргументы?

```
void foo(int);

int i = 42;

foo(i);        // (1)
foo(24);       // (2)
foo(i + 24);   // (3)
```

(1): по значению

ПРИМЕРЫ

Как передаются аргументы?

```
void foo(int);

int i = 42;

foo(i);        // (1)
foo(24);       // (2)
foo(i + 24);   // (3)
```

(1): по значению

(2) и (3): по значению

```
void foo(int&);

int i = 42;

foo(i);        // (1)
foo(24);       // (2)
foo(i + 24);   // (3)
```

```cpp
void foo(int&);

int i = 42;

foo(i);        // (1)
foo(24);       // (2)
foo(i + 24);   // (3)
```

(1): по значению

```
void foo(int&);

int i = 42;

foo(i);        // (1)
foo(24);       // (2)
foo(i + 24);   // (3)
```

(1): по значению

(2) и (3): не скомпилируется — *почему?*

```
void foo(int*);

int i = 42;
int *p = &i;

foo(&i);         // (1)
foo(p);          // (2)
foo(new int{}); // (3)
foo(nullptr);    // (4)
```

```
void foo(int*);

int i = 42;
int *p = &i;

foo(&i);          // (1)
foo(p);           // (2)
foo(new int{}); // (3)
foo(nullptr);     // (4)
```

(1): копируется временное значение — адрес i

```
void foo(int*);

int i = 42;
int *p = &i;

foo(&i);          // (1)
foo(p);           // (2)
foo(new int{}); // (3)
foo(nullptr);     // (4)
```

(1): копируется временное значение — адрес i

(2): копируется значение — адрес *из* p

```
void foo(int*);

int i = 42;
int *p = &i;

foo(&i);          // (1)
foo(p);           // (2)
foo(new int{}); // (3)
foo(nullptr);     // (4)
```

(1): копируется временное значение — адрес i

(2): копируется значение — адрес *из* p

(3): копируется значение — адрес, указывающий на кучу

```
void foo(int*);

int i = 42;
int *p = &i;

foo(&i);          // (1)
foo(p);           // (2)
foo(new int{}); // (3)
foo(nullptr);     // (4)
```

(1): копируется временное значение — адрес i

(2): копируется значение — адрес *из* p

(3): копируется значение — адрес, указывающий на кучу

(4): копируется значение — нулевой указатель

Как поменять указатель внутри функции?

```
void foo(???) { /* меняет аргумент */ };

int *p; // какой-то адрес
int *old_p = p;

foo(p);

assert(p != old_p);
```

Как поменять указатель внутри функции?

```
void foo(???) { /* меняет аргумент */ };

int *p; // какой-то адрес
int *old_p = p;

foo(p);

assert(p != old_p);
```

- `void foo(int** pptr) { *pptr = …; }`

Как поменять указатель внутри функции?

```
void foo(???) { /* меняет аргумент */ };

int *p; // какой-то адрес
int *old_p = p;

foo(p);

assert(p != old_p);
```

- `void foo(int** pptr) { *pptr = …; }`
- `void foo(int*& ptr) { ptr = …; }`

Как поменять указатель внутри функции?

```
void foo(???) { /* меняет аргумент */ };

int *p; // какой-то адрес
int *old_p = p;

foo(p);

assert(p != old_p);
```

- `void foo(int** pptr) { *pptr = …; }`
- `void foo(int*& ptr) { ptr = …; }`
- `void foo(int const* & ptr)` ???

Как поменять указатель внутри функции?

```
void foo(???) { /* меняет аргумент */ };

int *p; // какой-то адрес
int *old_p = p;

foo(p);

assert(p != old_p);
```

- `void foo(int** pptr) { *pptr = …; }`
- `void foo(int*& ptr) { ptr = …; }`
- `void foo(int const* & ptr)` ???
- `void foo(int* const& ptr)` ???

ДИНАМИЧЕСКАЯ ПАМЯТЬ

- new/new[] и парный delete/delete[]
- избегайте ошибок :)
- `new int[0] != nullptr`

НАХОЖДЕНИЕ ПРОБЛЕМ

- санитайзер — крутой инструмент
- в CI: сборка с `-fsanitize=address`
- примеры классов ошибок: wiki
- доклад, о внутренностях: AddressSanitizer anatomy

СТРУКТУРЫ

Минимальный вариант — это композиция типов

```
struct Simple { int b; };

struct Foo {
    int a;
    char *c;
    Simple s;
};

Foo f;

f.a = 42;
f.c = new char[3]{};
f.s.b = 24;
```

НАСЛЕДИЕ СИ

Если встретите, не удивляйтесь

```
typedef struct TagXYZ {
    int x, y, z;
} XYZ, *PtrXYZ;

// эквивалентно
struct TagXYZ {.....};
typedef struct TagXYZ XYZ;
typedef struct TagXYZ* PtrXYZ;
```

УКАЗАТЕЛИ НА СТРУКТУРЫ

```
struct Simple {
    int b;
};

Simple *s;

(*s).b = 42;
s->b = 42;
```

ЗАДАЧИ

#1 CONCAT

Напишите функцию `concat`, которая принимает два участка памяти и аллоцирует новый, содержащий в себе данные обоих участков

```
char const* concat(
    char const a[],
    size_t a_size,
    char const b[],
    size_t b_size,
    size_t& concat_size
);
```

Смотрите тесты для примеров поведения

#2 INT_VECTOR

Реализуйте простой вектор интов (на структурах) и вспомогательные методы

```
struct IntVector {
    int     *data = nullptr;
    size_t size = 0;
    size_t capacity_ = 0;
};

void pushBack(IntVector& v, int value);
void popBack(IntVector& v);
void deallocate(IntVector& v);
```

- `data_` — динамический массив из `capacity_` элементов
- Есть доступ к элементам `v.data[i]` при `i < v.size`

- pushBack — добавляет элемент в конец, выделяет новую память (*2, начинает с 1) при `size == capacity_`
- popBack — удаляет последний элемент из массива
- deallocate — очищает всю память data_, обнуляет размеры

#3 SINGLETON

- `singleton.hpp`
  - Определение функции `int inc()`, инкрементирующей счетчик и возвращающий его значение (начиная с нуля)

- `first.cpp`
  - Определение функции `int inc_first()`, вызывающей `inc`
- `second.cpp`
  - Определение функции `int inc_second()`, вызывающей `inc`

- Ожидание: `inc_first(), inc_second(), inc_first()` — вернут 0, 1, 2, ...

#4 STACK & HEAP GROWTH [*]

Покажите, что адреса стека убывают, а адреса кучи растут

Выведите в консоль соответствующие пары адресов (через `\n`) в функциях printStackGrowth и printHeapGrowth