Runtime-библиотека

Лекция №6

Теория языков программирования

Примеры к лекции

- pstigercpp: sourcecraft.dev/sshambir-public/pstigercpp
- pythonishc (2019): github.com/sergey-shambir/pythonishc

Вопросы этой лекции

Допустим, мы пишем компилятор для AMD64 и Linux

1. Как реализовать функции ввод-вывода?
2. Как реализовать строковый тип?
3. Как управлять памятью?

Вопросы этой лекции

Допустим, мы пишем компилятор для AMD64 и Linux

1. Как реализовать функции ввод-вывода?
2. Как реализовать строковый тип?
3. Как управлять памятью?

Варианты решения:

1. Использовать runtime-библиотеку языка C (libc)
2. Написать свой runtime

Хватит ли нам libc?

Ввод-вывод средствами C

1. **`printf`**`(format, …)` может почти всё

Ввод-вывод средствами C

1. **`printf`**`(format, …)` может почти всё
2. **`scanf`**`(format, …)` может почти всё

Ввод-вывод средствами C

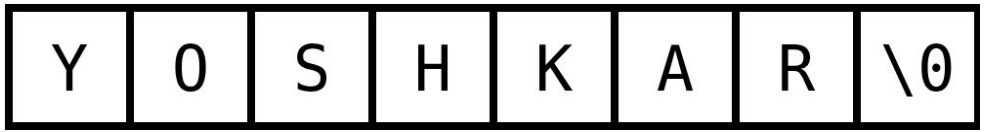
1. **`printf`**`(format, …)` может почти всё
2. **`scanf`**`(format, …)` может почти всё

у всего остального есть нюансы

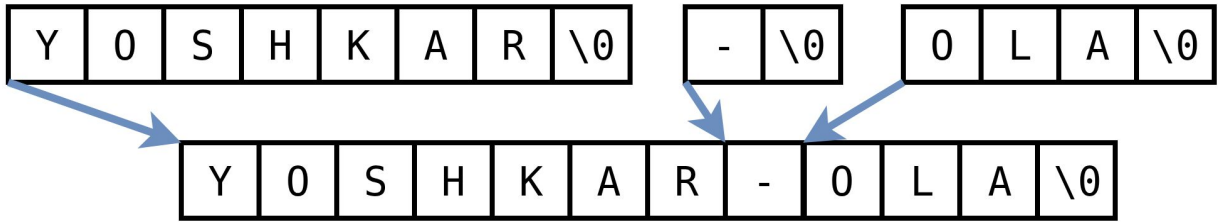
- `fflush(file)` — принимает FILE*
  - `stdout`, `stderr` — глобальные переменные
- `getchar()` — возвращает int
  - EOF — признак конца файла, *обычно* это `-1`
- `puts()` — печатает перенос строки
  - переносы строк бывают разными: CR, LF, CRLF

Работа со строками средствами C

Строка в стиле C — массив байт, завершённый `'\0'`



Конкатенация строк в стиле C




```c
int main() {
    const char* a = "Yoshkar";
    const char* b = "-";
    const char* c = "Ola";
    // ???
    puts(result);
}
```

Конкатенация строк в стиле C

Нам потребуются:

```
size_t strlen(const char* str);
void*  malloc(size_t size);
void   free  (void* ptr);
void*  memcpy(
  void* restrict dest,
  const void* restrict src,
  size_t count
);
```

Конкатенация строк в стиле C

```
// a = "Yoshkar", b = "-", c = "Ola"
size_t lenA = strlen(a);
size_t lenB = strlen(b);
size_t lenC = strlen(c);

char* data = malloc(lenA + lenB + lenC + 1);
memcpy(data, a, lenA * sizeof(char));
memcpy(data + sizeof(char) * lenA, b, lenB * sizeof(char));
memcpy(data + sizeof(char) * (lenA+lenB), c, lenC * sizeof(char));
data[lenA + lenB + lenC] = 0;

puts(data);
free(data);
```

Как управлять памятью?

Варианты для нового языка:

1. Вручную — как в C
2. Автоматически
    - Сборка мусора — как в Go / .NET / Java / PHP / Python / …
    - Подсчёт ссылок при выполнении — как в Objective C / Python
    - Деструкторы и семантика перемещения — как в C++
    - Контроль ссылок компилятором — как в Swift / Rust

Вывод

Если строить свой язык поверх libc:

1. Легко реализовать ввод/вывод
2. Сложнее реализовать работу со строками
3. Очень сложно — управление памятью

Своя runtime-библиотека

Библиотека tigerruntime

Операции ввода-вывода — функции

```
printi(i: int)
flush()
exit(i: int)
```

Работа со строками — объект строки, подсчёт ссылок, функции

```
print(s: string)          getchar(): string
ord(s: string): int       chr(int code)
concat(s1: string, s2: string)
substring(s: string, f: int, n: int): string
size(s: string): int
```

Ввод-вывод — TigerIO.h

```
// выводит целое число в стандартный поток вывода
void tiger_printi(int32_t i);

// записывает данные в буфере стандартного потока вывода
void tiger_flush(void);

/// завершает программу с кодом выхода i
void tiger_exit(int32_t i);
```

Ввод-вывод — TigerIO.c

```
void tiger_printi(int32_t i) {
    printf("%d", i);
}

void tiger_flush(void) {
    fflush(stdout);
}

void tiger_exit(int32_t i) {
    exit(i);
}
```

Объявление типа строки — TigerStrings.h

```
/*
* Представляет строку:
*  - управление памятью реализовано через подсчёт ссылок (reference
counting)
*  - со строкой хранится её размер
*  - данные строки завершаются нулевым символом '\0' (C-string)
*/
typedef struct TigerString TigerString;
```

Определение типа строки — TigerStrings.c

```
typedef struct TigerString
{
   char* data;
   size_t size;
   size_t refCount;
} TigerString;
```

Создание и управление памятью — TigerStrings.h

```
// Создаёт строку из строки в стиле C,
//   предназначена для создания строк из литералов.
TigerString* tiger_create_string(const char* src);

// Увеличивает счётчик ссылок для строки.
void tiger_add_reference(TigerString* s);

// Уменьшает счётчик ссылок для строки.
void tiger_remove_reference(TigerString* s);
```

Создание строки из литерала — TigerStrings.c

```
TigerString* tiger_create_string(const char* src) {
    size_t size = strlen(src);
    TigerString* str = tiger_allocate_string(size);
    memcpy(str->data, src, sizeof(char) * size);

    return str;
}
```

Аллокация памяти для строки — TigerStrings.c

```c
// Создаёт строку заданного размера,
//  инициализированную нулевыми символами.
static TigerString* tiger_allocate_string(size_t size) {
    char* data = calloc(sizeof(char), size + 1);
    TigerString* str = calloc(sizeof(TigerString), 1);
    // ... а что если память не выделена?

    str->data = data;
    str->size = size;
    str->refCount = 1;

    return str;
}
```

Освобождение ссылки на строку — TigerStrings.c

```
void tiger_remove_reference(TigerString* s) {
    --s->refCount;
    if (s->refCount == 0) {
        free(s->data);
        free(s);
    }
}
```

Освобождение ссылки на строку — TigerStrings.c

```
void tiger_remove_reference(TigerString* s) {
    --s->refCount;
    if (s->refCount == 0) {
        free(s->data);
        free(s);
    }
}
```

1. При создании строки счётчик равен 1
2. Компилятор добавляет вызовы `tiger_remove_reference`
   - автоматически в конце области видимости
   - для всех строк, созданных области видимости
3. Перед возвратом строки добавляем +1 к счётчику

Сравнение строк

В языке Tiger сравнение строк лексикографическое:

1. a = b
2. a <> b
3. a < b
4. a > b
5. a <= b
6. a >= b

Сравнение строк

```
// Сравнивает две строки лексикографически и возвращает число:
//   отрицательное, если s1 меньше s2
//   0, если s1 равна s2
//   положительное, если s1 больше s2
int32_t tiger_compare_strings(TigerString* s1, TigerString* s2) {
   return strcmp(s1->data, s2->data);
}
```

Сравнение строк

```
// Сравнивает две строки лексикографически и возвращает:
//   -1, если s1 меньше s2
//   0, если s1 равна s2
//   1, если s1 больше s2
int32_t tiger_compare_strings(TigerString* s1, TigerString* s2) {
   return strcmp(s1->data, s2->data);
}
```

Проверка на равенство строк в компиляторе:

```
llvm::Constant* zero = llvm::ConstantInt::getSigned(_int32Type, 0);
llvm::Value* cmp = _builder.CreateCall(_compare, {left, right});
llvm::Value* equal = _builder.CreateICmpEQ(cmp, zero);
llvm::Value* result = _builder.CreateIntCast(result, int32Type, false);
```

Проверка runtime-библиотеки

Проверка runtime-библиотеки

Если runtime написан на C, ему нужны:

1. Санитайзеры (Sanitizers)
2. Модульные тесты runtime-библиотеки
3. Приёмочные тесты компилятора

Sanitizers в C++

В современных компиляторах C++ есть Sanitizers

Они могут найти:

1. утечки памяти
2. выход за границы контейнера / стека потока
3. обращение к освобождённой памяти
4. чтение неинициализированной памяти
5. undefined behavior

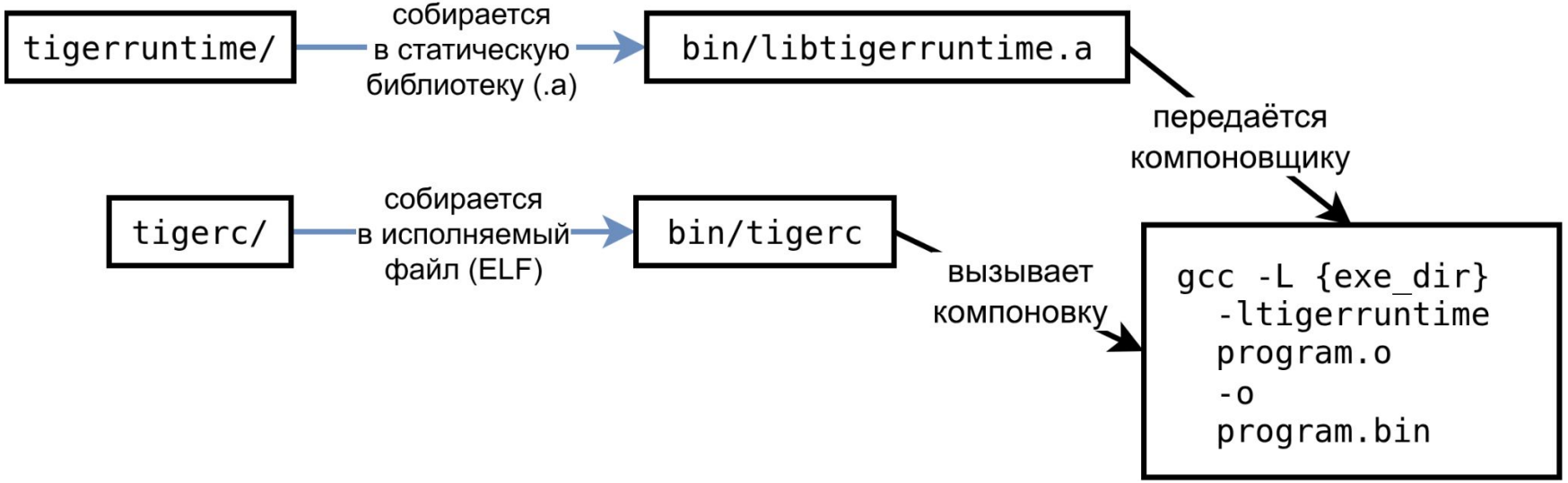
Sanitizers в C++

Базовый набор sanitizers

- GCC / Clang:
  - `-fsanitize=address,undefined`
  - `-fno-omit-frame-pointer`
- MSVC:
  - `/fsanitize=address`

Компоновка с runtime-библиотекой

Компоновка с runtime-библиотекой



Подытожим

Сравнение вариантов runtime для языка Tiger

	libc	**tigerruntime поверх libc**
Ввод-вывод	просто	просто
Операции со строками	средне (C-style string)	просто (свой тип + opaque pointer)
Управление памятью	сложно (ручное либо семантика владения)	просто (подсчёт ссылок + opaque pointer)
Компоновка программы	просто	средне (копируем libtigerruntime.a)

Примеры к лекции

- pstigercpp: sourcecraft.dev/sshambir-public/pstigercpp
  - ветка `03_int_expressions` — использует libc
    - нет операций со строками

Примеры к лекции

- pstigercpp: sourcecraft.dev/sshambir-public/pstigercpp
  - ветка `03_int_expressions` — использует libc
    - нет операций со строками
  - ветка `04_string_expressions` — использует tigerruntime поверх libc
    - управление памятью через автоматический подсчёт ссылок

Примеры к лекции

- pstigercpp: sourcecraft.dev/sshambir-public/pstigercpp
  - ветка `03_int_expressions` — использует libc
    - нет операций со строками
  - ветка `04_string_expressions` — использует tigerruntime поверх libc
    - управление памятью через автоматический подсчёт ссылок
- pythonishc (2019): github.com/sergey-shambir/pythonishc
  - использует libc
  - есть операции со строками
  - управление памятью через копирование + удаление в конце области видимости

Конец!