Кроссязыковые библиотеки

Written once, used everywhere

- sqlite
- curl
- libgit
- pcre
- llvm-c

ASM −> ASM

Вызов функции на ASM

```
sum:
        add     %rdi, %rsi  # %rsi += rdi
        mov     %rsi, %rax
        ret

_start:
        mov     $40, %rdi
        mov     $2, %rsi
        call    sum

        mov     %rax, %rdi
```

Вызов функции на ASM

```
sum:
        add     %r8, %r9  # %r9 += %r8
        mov     %r9, %rax
        ret

_start:
        mov     $40, %r8
        mov     $2, %r9
        call    sum

        mov     %rax, %rdi
```

Можем ли использовать другие регистры?

https://godbolt.org/z/Kob75sh9x

Вызов функции на ASM

```
_start:
        push    $40
        push    $2
        call    sum

        pop     %rdi
```

```
sum:
        ...
        pop     %rdi
        pop     %rsi
        add     %rsi, %rdi
        push    %rdi
        ...
        ret
```

Можем ли использовать стек?

https://godbolt.org/z/Gv4EGaf5n

Предварительный вывод

- Мы можем использовать любой способ передачи аргументов в функцию, если этот способ согласован.
- В большинстве случаев нам не хватает контекста, чтобы изобретать новый способ для каждой функции.

ASM –> C

Напишем функцию на C

```
void repeat(char c, int count) {
  for (int i = 0; i < count; ++i) {
    printf("%c", c);
  }
  printf("\n");
}
```

AMD64 ABI: Register Usage

…

`%rax` [...] 1st return register

`%rsi` used to pass 2nd argument to functions

`%rdi` used to pass 1st argument to functions

…

...и вызовем ее из asm-вставки

```
void repeat(char c, int count);

// repeat(')', 33);
asm(
  "mov $')', %rdi\n"
  "mov $33, %rsi\n"
  "call repeat\n"
);
```

...и перепутаем аргументы

```
void repeat(char c, int count);
```

```
asm(
  "mov $33, %rdi\n"
  "mov $')', %rsi\n"
  "call repeat\n"
);
```

…и перепутаем аргументы

```
void repeat(char c, int count);

// repeat((char)33, (int)')');
// repeat(     '!',       41);
asm(
  "mov $33, %rdi\n"
  "mov $')', %rsi\n"
  "call repeat\n"
);
```


https://godbolt.org/z/Mv4vEe5z4

Наблюдения

1. Компилятор придерживается определенных соглашений вызова функций — calling conventions.

C –> Pascal

Pascal

```
unit math;
interface
function add(a, b: longInt) : longInt;

implementation
function add(a, b: longInt) : longInt;
begin
    add := a + b;
end;
end.
```

Функция math.add в объектном файле

```
$ fpc -a math.pas
$ objdump -t math.o
…
MATH_$$_ADD$LONGINT$LONGINT$$LONGINT
```

main.c

```
int32_t MATH_$$_ADD$LONGINT$LONGINT$$LONGINT(
    int32_t, int32_t);

...
const int32_t x =
MATH_$$_ADD$LONGINT$LONGINT$$LONGINT(a, b);
```

strlen

```
// Pascal
function mylen(const x: string): longInt;
alias: ‘mylen’;
begin
    mylen := length(x);
end;

// C, len == ?
const char* str = "hello";
int32_t len = mylen(str);
```

strlen

```
// Pascal
function mylen(const x: string): longInt;
alias: ‘mylen’;
begin
    mylen := length(x);
end;

// C, len == 104
const char* str = "hello";
int32_t len = mylen(str);
```

Наблюдения

1. Компилятор придерживается определенных соглашений вызова функций — calling conventions.
2. Компилятор определяет способ искажения имен функций (mangling).

C −> Rust

Функция Rust для вызова из C

```
pub struct IntsRust {
    a: i8, b: i16, c: i8,
}

#[no_mangle]
pub unsafe extern "C"
fn fill_rust(obj: *mut IntsRust) {
    (*obj).a = 2;
    (*obj).b = 8;
    (*obj).c = 16;
}
```

Клиент на C

```
struct Ints {
  int8_t a;
  int16_t b;
  int8_t c;
};
```

```
struct Ints ints_rust;
fill_rust(&ints_rust);
printf(
      "Rust layout:\n"
      "a = %d\n"
      "b = %d\n"
      "c = %d\n\n",
      ints_rust.a,
      ints_rust.b,
      ints_rust.c);
```

Вывод приложения

```
pub struct IntsRust { a: i8, b: i16, c: i8, }
```

Ожидаем значения 2, 8, 16.

Фактический вывод:

```
a = 8
b = 4098
c = 0
```

Объясните вывод

Вывод приложения

```
pub struct IntsRust { a: i8, b: i16, c: i8, }
```

Ожидаем значения 2, 8, 16 **(hex: 02, 08, 10)**.

Фактический вывод:

```
          hex
a = 8     08
b = 4098  1002
c = 0     0
```

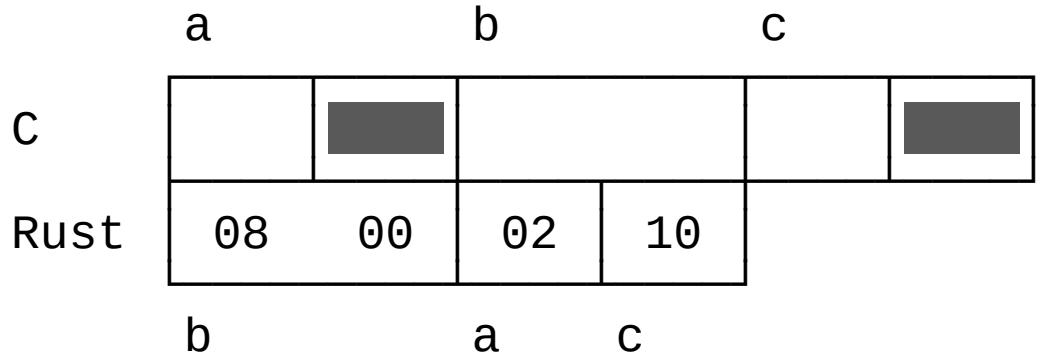
В hex гораздо нагляднее

Структура в LLVM IR

```
pub struct IntsRust { a: i8, b: i16, c: i8, }
```

```
$ rustc --crate-type=staticlib --emit=llvm-ir lib.rs
$ cat lib.ll
...
%IntsRust = type { i16, i8, i8 }
...
```

Struct Layout



C-layout in Rust

```
#[repr(C)]
pub struct IntsC {
    a: i8,
    b: i16,
    c: i8,
}
```

Наблюдения

1. Компилятор придерживается определенных соглашений вызова функций — calling conventions.
2. Компилятор определяет способ искажения имен функций (mangling).
3. Компилятор отчасти определяет представление объектов в памяти.

Все это только частности

Application Binary Interface (ABI)

ABI определяет:

- Использование регистров
- Способ передачи параметров в функции
- Раскрутку стека
- Многое другое

https://gitlab.com/x86-psABIs/x86-64-ABI

Argument Register Overview

Argument Type	Registers
Integer/Pointer Arguments 1-6	RDI, RSI, RDX, RCX, R8, R9
Floating Point Arguments 1-8	XMM0 - XMM7
Excess Arguments	Stack
Static chain pointer	R10

Взаимодействие через ASM

Взаимодействие через ~~ASM~~ C

Дальнейшие действия

- При вызове стороннего кода из C мы видели разброд и шатания
- При вызове кода C из других языков мы увидим стабильность

C++ –> C

Простой пример

```
// sum.h
#pragma once
int sum(int a, int b);

// sum.c
#include "sum.h"
int sum(int a, int b) {
  return a + b;
}
```

```
// main.cpp
#include "sum.h"
#include <iostream>

int main() {
  std::cout
     << sum(40, 2) << '\n';
}
```

Компилируем…

```
$ g++ -c -o obj/main.o src/main.cpp
$ gcc -c -o obj/sum.o src/sum.c
$ g++ -o bin/sum obj/main.o obj/sum.o
```

Компилируем...

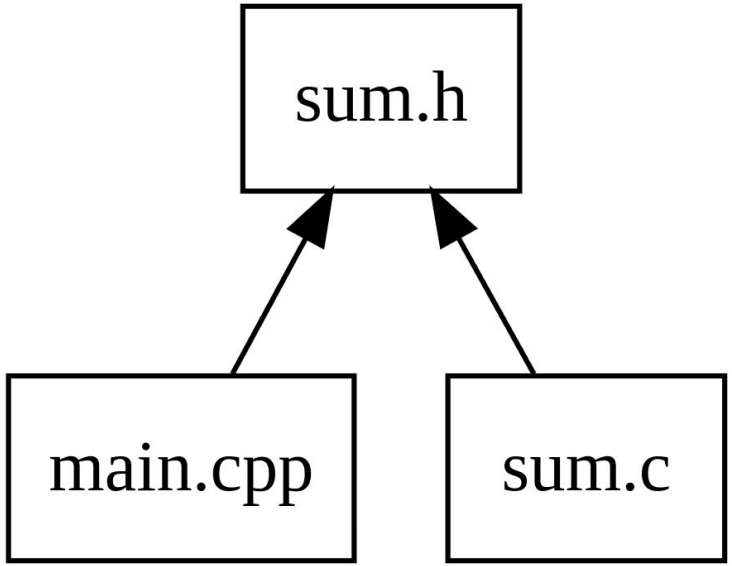
```
$ g++ -c -o obj/main.o src/main.cpp
$ gcc -c -o obj/sum.o src/sum.c
$ g++ -o bin/sum obj/main.o obj/sum.o

/usr/bin/ld: obj/main.o: in function `main':
main.cpp: undefined reference to `sum(int, int)'
```

Различия в манглировании

```
$ objdump -t obj/sum.o
sum

$ objdump -t obj/main.o
_Z3sumii
```



Манглирование

- В C++ разрешено манглировать имена типами
- Благодаря этому существует большинство возможностей языка
  - Методы, в т.ч. конструкторы и деструкторы
  - Перегрузка функций
  - Пространства имен
  - Шаблоны

```
$ c++filt _ZN8geometry5Point4moveEdd

geometry::Point::move(double, double)
```

Общий заголовочный файл для C и C++

```
#pragma once

#ifdef __cplusplus
extern "C" {
#endif

int sum(int a, int b);

#ifdef __cplusplus
}
#endif
```

Промежуточные выводы

- Мы можем вызывать C из C++
- Забегая вперед: мы можем вызывать C из любого языка
- Но мы хотели бы писать библиотеки на более удобном C++

C –> C++

C++ ABI

На C++ ABI нет стандарта

Что может влиять на ABI:

- Наличие virtual-методов
- Ссылки
- Исключения
- …

Задача: сделать тип доступным из C

```
namespace geometry {
class Circle : public Shape {
 public:
  Circle(Point center, double radius);

  double area() const override;
  double radius() const;
  ...
};
}
```

libgeometry-c/Point.h

```
// Определяем Point за пределами namespace geometry

#pragma once

typedef struct {
  double x_;
  double y_;
} Point;
```

libgeometry-c/Shape.h

```
#ifdef __cplusplus
extern "C" {
#endif

typedef void Shape;

...

#ifdef __cplusplus
}
#endif
```

libgeometry-c/Circle.h

```
// Оборачиваем конструктор Circle
#ifdef __cplusplus
extern "C" {
#endif

Shape* geometry_circle_new(
       Point center, double radius);

#ifdef __cplusplus
}
#endif
```

libgeometry-c/Circle.h

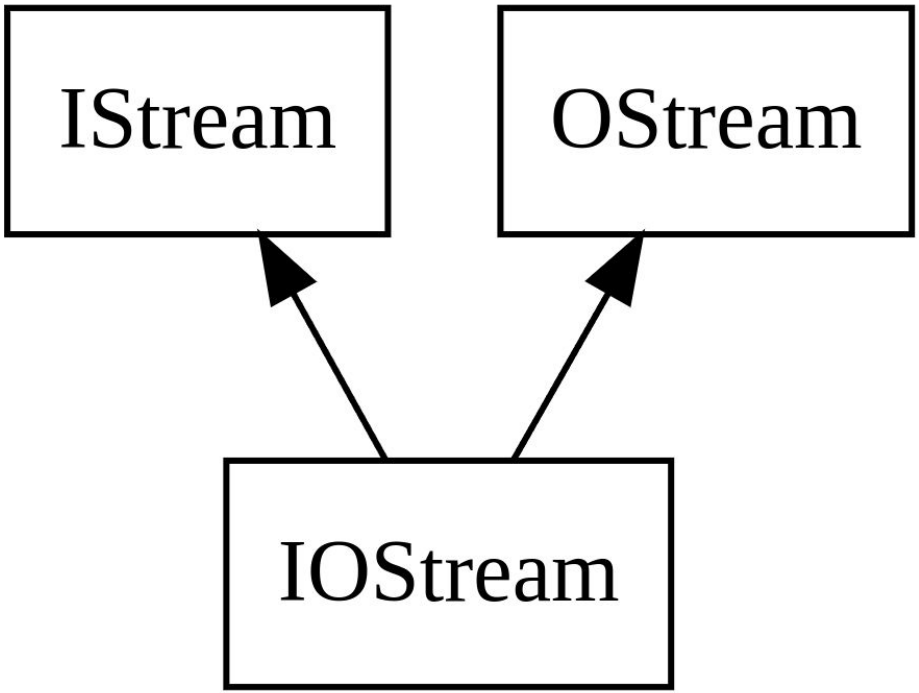
```
Shape* geometry_circle_new(Point center, double r) {
  const geometry::Point pt{center.x_, center.y_};

  return static_cast<geometry::Shape*>(
    new geometry::Circle(pt, r));
}
```

Реализация — код на C++

От языка C здесь только линковка

Зачем нужен каст к базовому классу



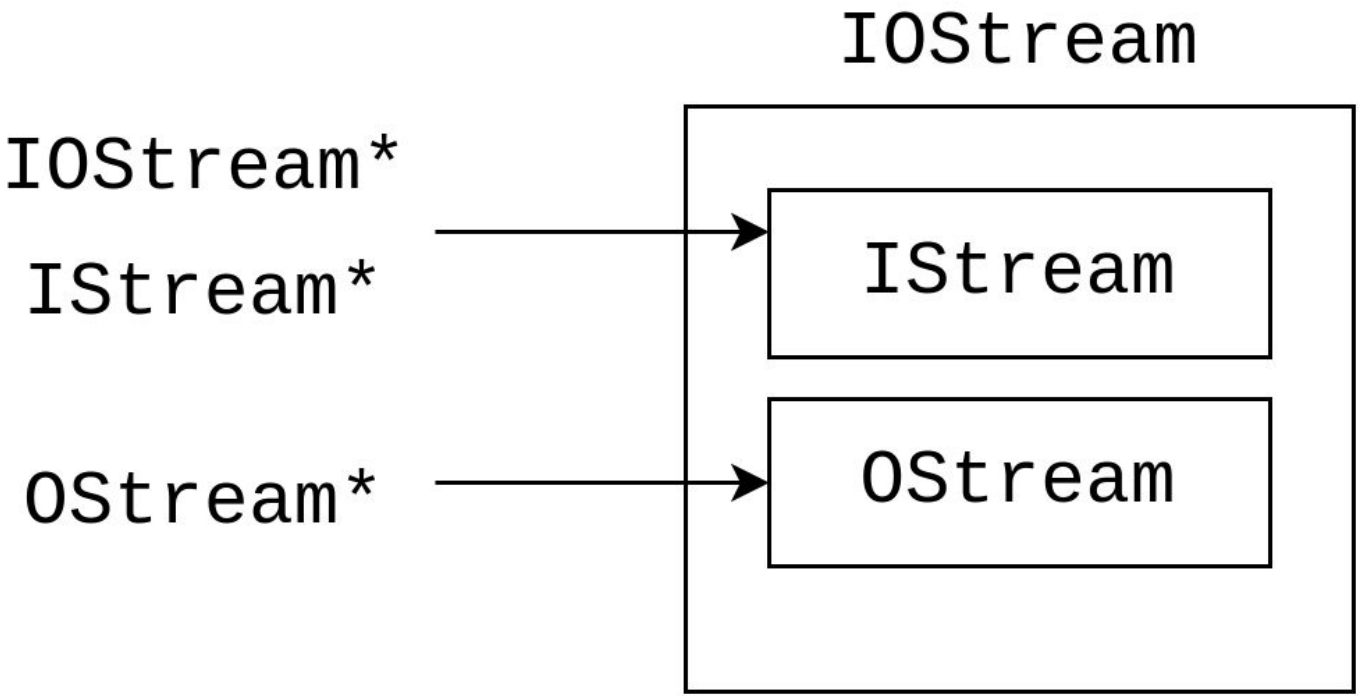
Зачем нужен каст к базовому классу

```
IOStream* ios = new IOStream();
IStream* is = ios;
OStream* os = ios;

fmt::print(
    "ios: {}\n"   // ios: 0x1935eb0
    "is:  {}\n"   // is:  0x1935eb0
    "os:  {}\n",  // os:  0x1935eb4
    static_cast<void*>(ios),
    static_cast<void*>(is),
    static_cast<void*>(os));
```


https://godbolt.org/z/hYM9sKs9o

Схематичное строение объекта



libgeometry-c/Shape.h

```
#ifdef __cplusplus
extern "C" {
#endif

typedef void Shape;

double geometry_shape_area(const Shape* self);
void geometry_shape_delete(const Shape* self);

#ifdef __cplusplus
}
#endif
```

libgeometry-c/Shape.cpp

```
double geometry_shape_area(const Shape* self) {
  const auto* shape
      = static_cast<const geometry::Shape*>(self);
  return shape->area();
}

void geometry_shape_delete(const Shape* self) {
  const auto* shape
      = static_cast<const geometry::Shape*>(self);
  delete shape;
}
```

testapp/main.c

```
const Point pt = {0, 0};
const Shape* circle = geometry_circle_new(pt, 2);
const double area = geometry_shape_area(circle);
printf("circle area = %.6lf\n", area);
geometry_shape_delete(circle);
```

Задача: сделать тип доступным из C

```
namespace geometry {
class Circle : public Shape {
 public:
  Circle(Point center, double radius);

  double area() const override; // DONE
  double radius() const;        // ???
  ...
};
}
```

libgeometry-c/Circle.h

```
// extern "C"


Shape* geometry_circle_new(
       Point center, double radius);

double geometry_circle_radius(const Shape* self);
```

libgeometry-c/Circle.cpp

```
double geometry_circle_radius(const Shape* self) {
  const auto* shape
    = static_cast<const geometry::Shape*>(self);

  const auto* circle
    = dynamic_cast<const geometry::Circle*>(shape);

  return circle->radius();
}
```

Down Cast

```
Shape* s = new Circle();

auto* c = static_cast<Circle*>(s);   // OK

auto* t = static_cast<Triangle*>(s); // No Error, UB
```

Down Cast

```
Shape* s = new Circle();

auto* c = dynamic_cast<Circle*>(s);   // OK

auto* t = dynamic_cast<Triangle*>(s); // nullptr
```

dynamic_cast выполняет приведение с учетом динамического типа объекта

Резюме

В C-Api могут быть только:

- Standard-layout типы
- Функции с C-linkage

Не может быть:

- Пространств имен
- Исключений
- Всего остального, что мы так любим

Opaque Ptrs

Нежелательный void*

```
#ifdef __cplusplus
extern "C" {
#endif

typedef void Shape;

double geometry_shape_area(const Shape* self);

#ifdef __cplusplus
}
#endif
```

Opaque Ptr

```
#ifdef __cplusplus
extern "C" {
#endif

typedef struct Shape Shape;

double geometry_shape_area(const Shape* self);

#ifdef __cplusplus
}
#endif
```

static_cast больше не работает

```
double geometry_shape_area(const Shape* self) {
  const auto* shape
    = reinterpret_cast<const geometry::Shape*>(self);
  return shape->area();
}
```

Python −> C++

Вспомним, что у нас есть

- Библиотека geometry на C++
- Библиотека geometry-c на C++ и интерфейсом с C-linkage

Хотим использовать типы из geometry в Python.

Foreign Function Interface (FFI)

Механизм вызова функций из других языков.

В Python представлен модулем ctypes.

client/geometry.py: определение структуры

```
import ctypes

geometry_lib = \
    ctypes.cdll.LoadLibrary('libgeometry-c.so')

class Point(ctypes.Structure):
    _fields_ = [
        ('x', ctypes.c_double),
        ('y', ctypes.c_double)
    ]
```

client/geometry.py: определение функций

```
...
circle_new = geometry_lib.geometry_circle_new
circle_new.argtypes = [Point, ctypes.c_double]
circle_new.restype = ctypes.c_void_p


shape_area = geometry_lib.geometry_shape_area
shape_area.argtypes = [ctypes.c_void_p]
shape_area.restype = ctypes.c_double
```

client/main.py

```
import geometry


circle = geometry.circle_new(geometry.Point(0, 0), 2)
print(geometry.shape_area(circle))
geometry.shape_delete(circle)
```

Нам хочется более нативного интерфейса

client/geometry.py

```
class Shape:
    def __init__(self, obj):
        self._obj = obj

    def __del__(self):
        shape_delete(self._obj)

    def area(self):
        return shape_area(self._obj)
```

client/geometry.py

```
class Circle(Shape):
    def __init__(self, center: Point, radius: float):
        obj = circle_new(center, radius)
        super(Circle, self).__init__(obj)

    @property
    def radius(self):
        return circle_radius(self._obj)
```

client/main.py

```
c = geometry.Circle(geometry.Point(0, 0), 2)

print(c.area())

print(c.radius)
```

Резюме

- FFI — универсальный механизм.
  - Удобство C++ внутри библиотеки
  - Доступность для всех языков снаружи
- Требует много ручной работы:
  - Обертка над C++-кодом
  - Обертка над C-API
  - Нативная обертка для целевого языка

Raw FFI

Воспользуемся libffi напрямую

```
// LIBM_SO == "libm.so.6"

void* handle = dlopen(LIBM_SO, RTLD_LAZY);

void* pow_ptr = dlsym(handle, "pow");
```


https://linux.die.net/man/3/dlopen

Вызов функции по указателю

```
using PowFn = double (*)(double, double);

auto pow_fn = reinterpret_cast<PowFn>(pow_ptr);

auto result = pow_fn(2, 4);
```

Такое приведение типа

возможно только на этапе компиляции

https://linux.die.net/man/3/dlopen

Вызов через FFI

- FFI Позволяет вызвать функцию с неизвестной на этапе компиляции сигнатурой
- Для этого сначала нужно создать список типов параметров

```
std::vector<ffi_type*> arg_types {&ffi_type_double,
                                  &ffi_type_double};
```

Вызов через FFI

- Затем подготовить CIF (Call InterFace)

```
ffi_cif cif;

ffi_prep_cif(&cif, FFI_DEFAULT_ABI,
             arg_types.size(),
             &ffi_type_double,  // rtype
             arg_types.data()
```


https://linux.die.net/man/3/ffi_prep_cif

Вызов через FFI

- Создать массив из void* для передачи аргументов
- Вызвать функцию

```
double result = 0, base = 2, power = 4;
std::vector<void*> arg_ptrs = {&base, &power};

using VoidFn = void (*)();
ffi_call(&cif, reinterpret_cast<VoidFn>(pow_ptr),
         &result, arg_ptrs.data());
```

Вызов через FFI

`ffi_call` выберет способ вызова функции в соответствии с переданным ABI.

Например: ffi_call_unix64