

# Кроссязыковые библиотеки

# Written once, used everywhere

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

**ASM  $\rightarrow$  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?*

# Вызов функции на ASM

```
_start:                                sum:
    push    $40                          ...
    push    $2                            pop     %rdi
    call    sum                          pop     %rsi
                                           add     %rsi, %rdi
    pop     %rdi                          push    %rdi
                                           ...
                                           ret
```

*Можем ли использовать стек?*

# 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  $\rightarrow$  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"
);
```

# 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
```

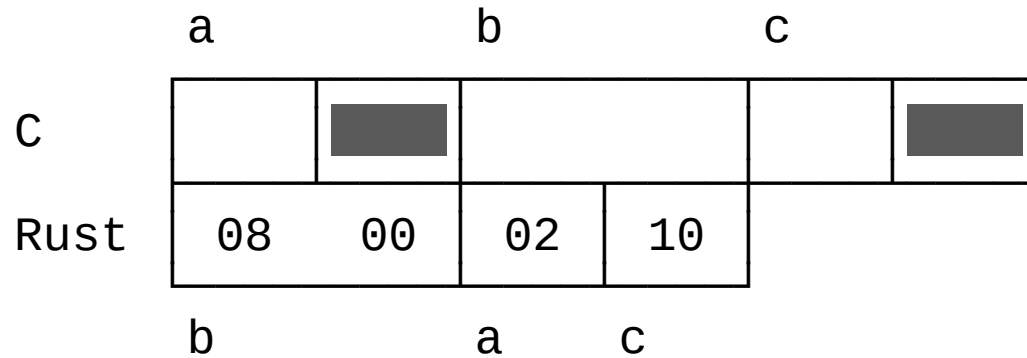
```
$ cat lib.ll
```

```
...
```

```
%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
- Раскрытие 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
```

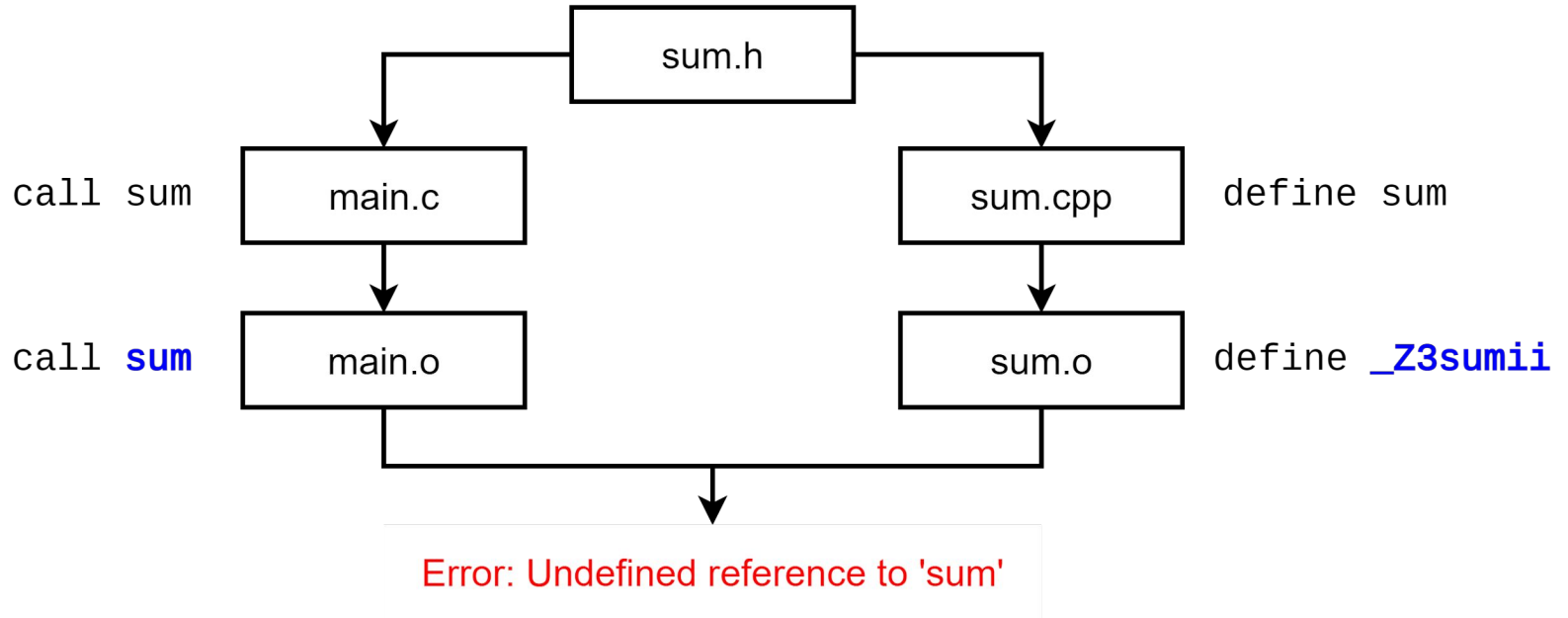
```
$ gcc -c -o obj/sum.o src/sum.c
```

```
$ 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)'
```

declare int sum(int a, int b)





# Воспроизведение

```
$ gcc -c main.c && g++ -c sum.cpp
```

```
$ objdump -t main.o | grep sum
```

```
... sum
```

```
$ objdump -t sum.o | grep sum
```

```
... _Z3sumi
```

```
$ gcc main.o sum.o
```

```
main.c:(.text+0x17): undefined reference to `sum'
```

# 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)
```

# Применение манглирования

Методы классов:

- `Point Point::move(Vec2d) -> _ZN5Point4moveE5Vec2d`

Перегрузка:

- `int sum(int, int) -> _Z3sumii`
- `double sum(double, double) -> _Z3sumdd`

Пространства имен, шаблоны, перегрузка операторов...

# Манглирование

Это полезный механизм, который мешает нашей задаче.

Нам нужен способ отключить его.

extern "C" — language linkage

# Синтаксис Linkage specification

```
extern "C" int sum(int a, int b);
```

```
// или
```

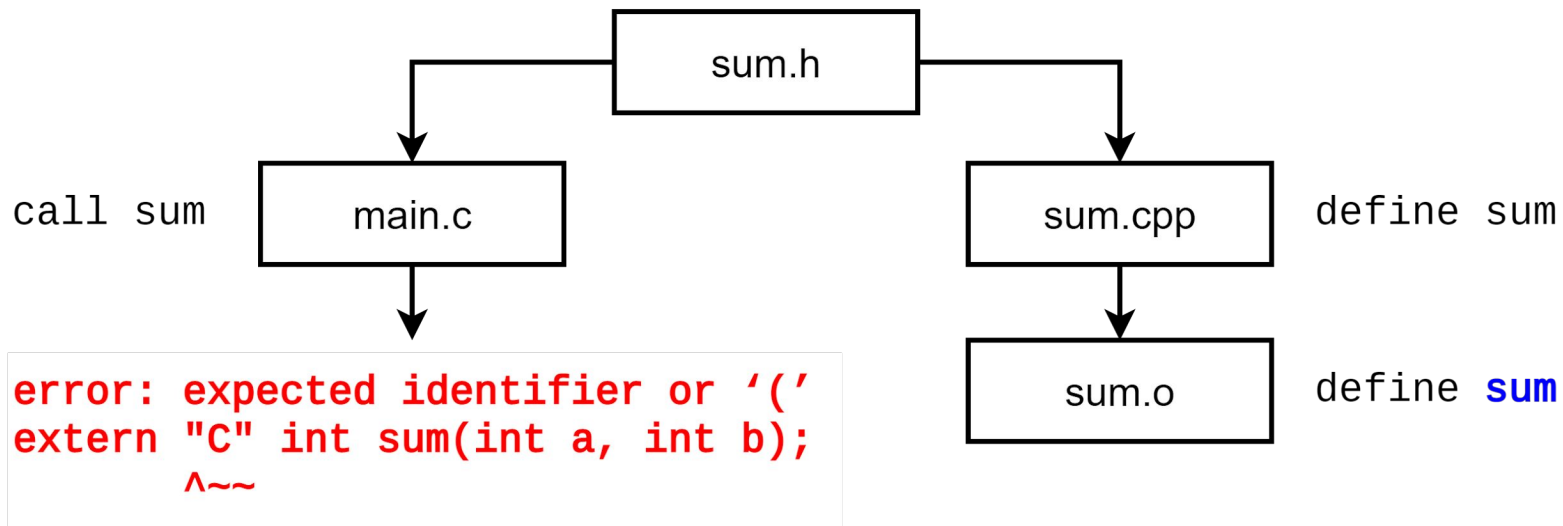
```
extern "C" {
```

```
    int sum(int a, int b);
```

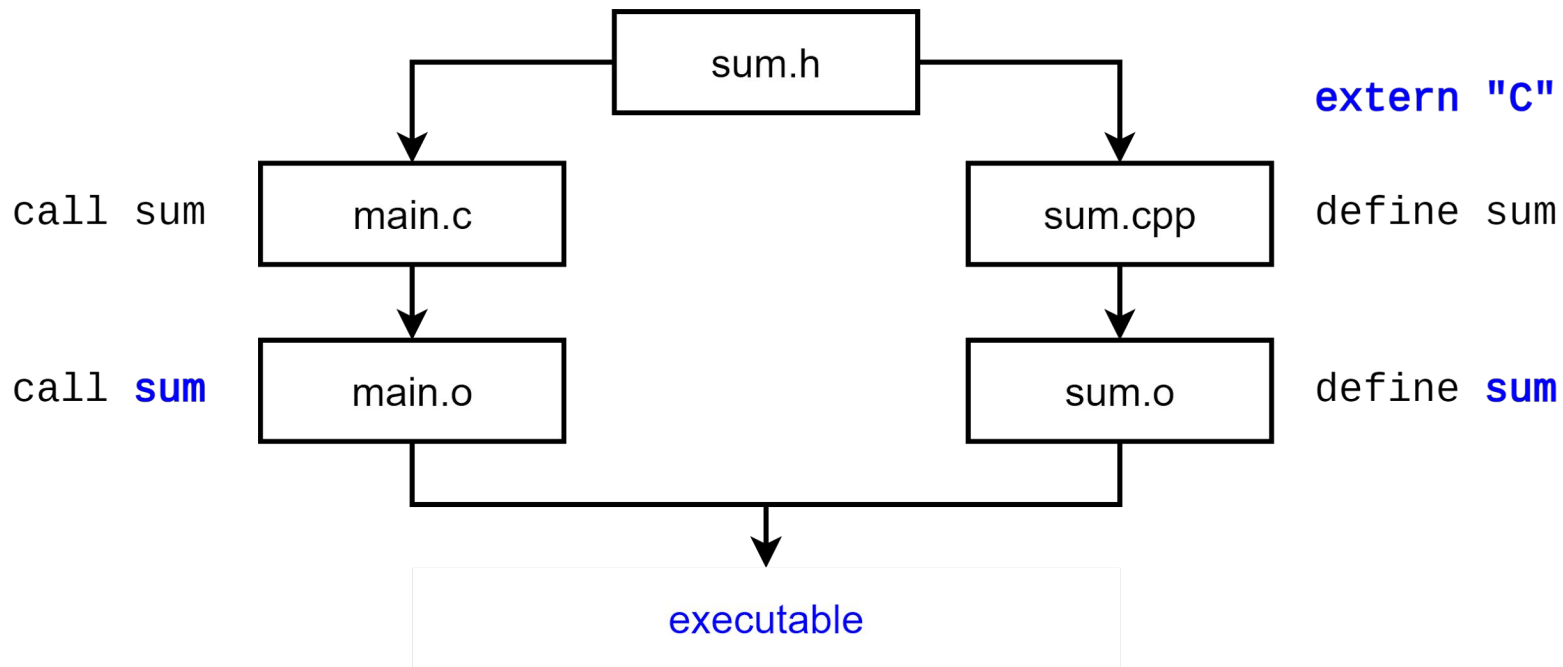
```
    // Еще объявления
```

```
}
```

declare **extern "C"** int sum(int a, int b)



declare (**extern "C"**)? int sum(int a, int b)



# 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
- ...

# Промежуточный итог

- Пока мы обходились аннотированием существующего кода.
- Часто этого недостаточно:
  - Пространства имен.
  - Методы классов.
  - Перегрузка (не будем рассматривать).

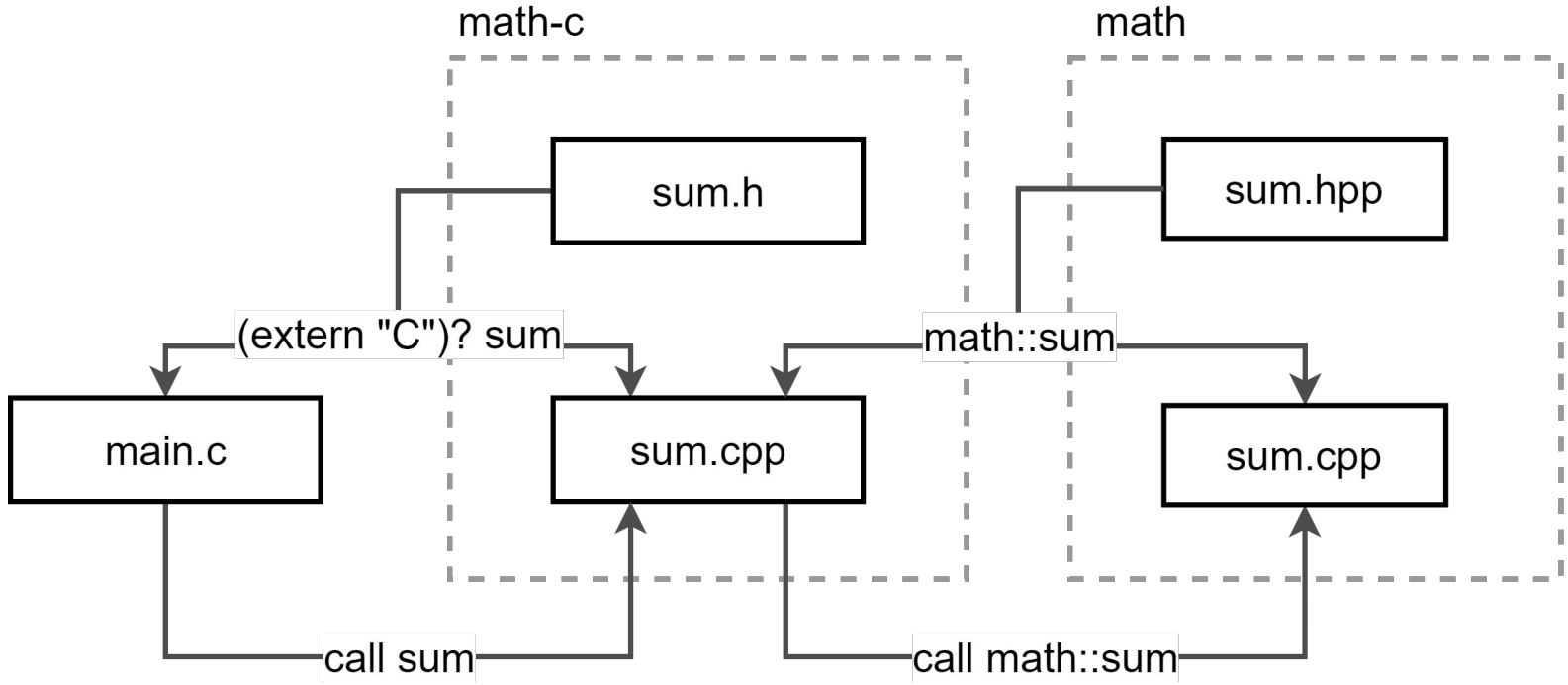
# Пространства имен

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

int sum(int a, int b);

#ifdef __cplusplus
} // namespace math
} // extern "C"
#endif
```

*Это скомпилируется,  
но кажется не очень удобным*



# main.c

```
#include <math-c/sum.h>
```

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

```
int main() {  
    int x = sum(40, 2);  
    printf("x = %d\n", x);  
}
```

# math-c/sum.h

```
#pragma once
```

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

```
int sum(int a, int b);
```

```
#ifdef __cplusplus  
}  
#endif
```



## math-c/sum.cpp

```
#include <math-c/sum.h>
```

```
#include <math/sum.hpp>
```

```
int sum(int a, int b) {  
    return math::sum(a, b);  
}
```

## math/sum.hpp, math/sum.cpp

```
#pragma once
```

```
namespace math {
```

```
int sum(int a, int b);
```

```
}
```

```
#include <math/sum.hpp>
```

```
namespace math {
```

```
int sum(int a, int b) {  
    return a + b;
```

```
}
```

```
}
```

## В следующей серии

- Как передавать типы через границу языков
- Как вызывать виртуальные и не виртуальные методы
- Как вызывать сишные функции из python/go/java/whatever

Продолжение следует

PS

# Raw FFI

## Воспользуемся libffi напрямую

```
// LIBM_SO == "libm.so.6"  
void* handle = dlopen(LIBM_SO, RTLD_LAZY);  
void* pow_ptr = dlsym(handle, "pow");
```

## 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*



# 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())
```

# 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](#)

Q&A