Building Bridges

C++ Interop., Foreign Function Interfaces & ABI

Qt Quality Assurance

Gareth Williamson

Software Engineer

Qt Quality Assurance

Motivation

Why build a multi-language system?

Motivation

Why build a multi-language system?

Use the right tool for the job

Motivation

Why build a multi-language system?

Use the right tool for the job

Performance Critical Code

Expressive API

Motivation

Why build a multi-language system?

Use the right tool for the job

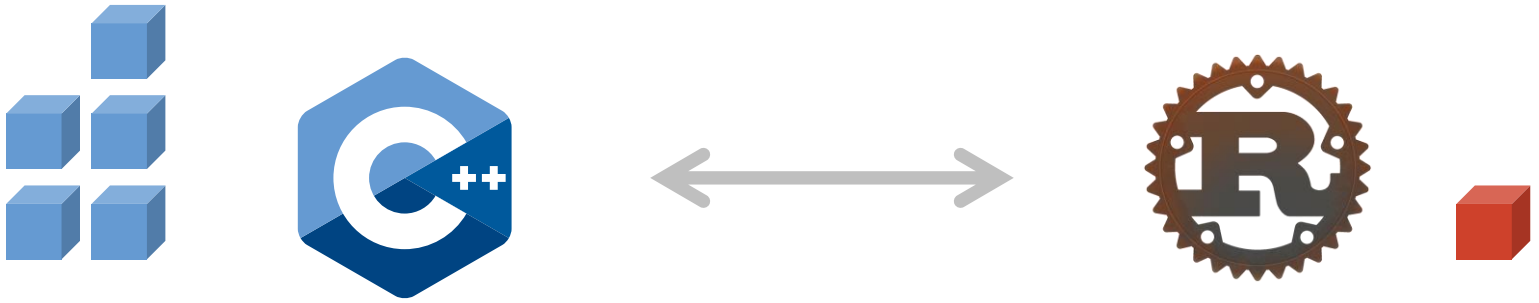
Re-use existing code

Motivation

Why build a multi-language system?

Use the right tool for the job

Re-use existing code



Existing Codebase

New Library

Motivation

How can we build a multi-language system?

Motivation

How can we build a multi-language system?

Inter-process

Motivation

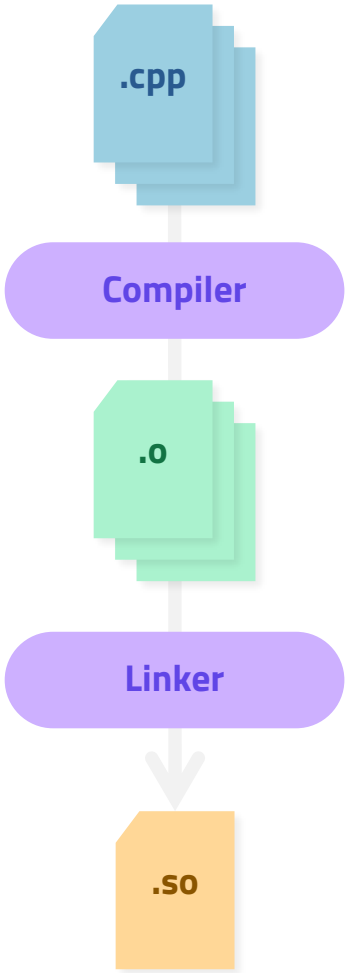
How can we build a multi-language system?

Inter-process

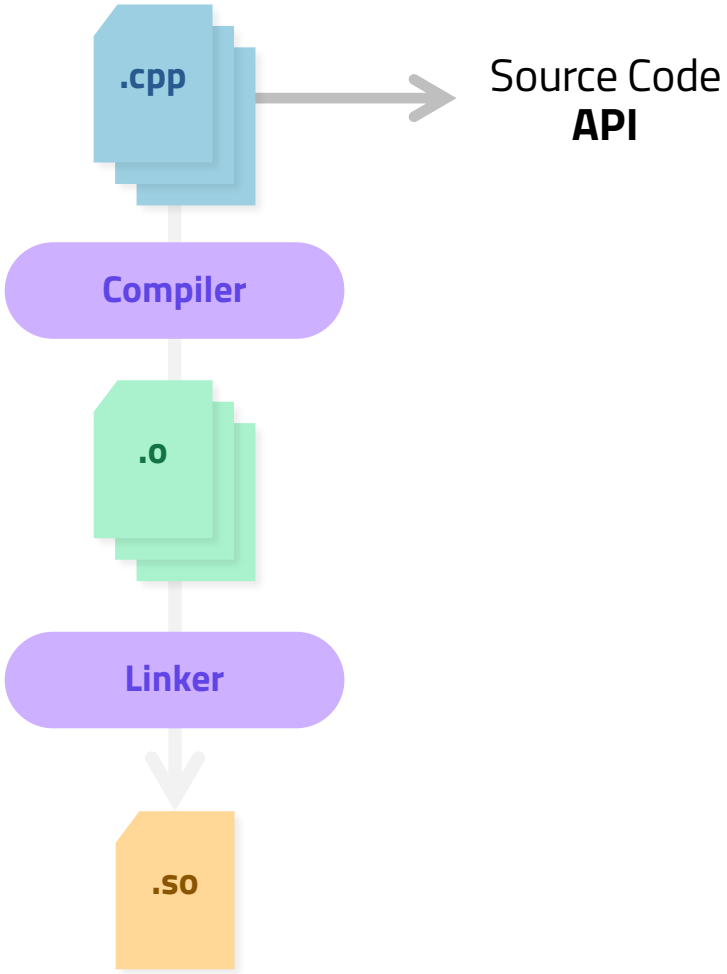
Link Object Code

Application Binary Interface

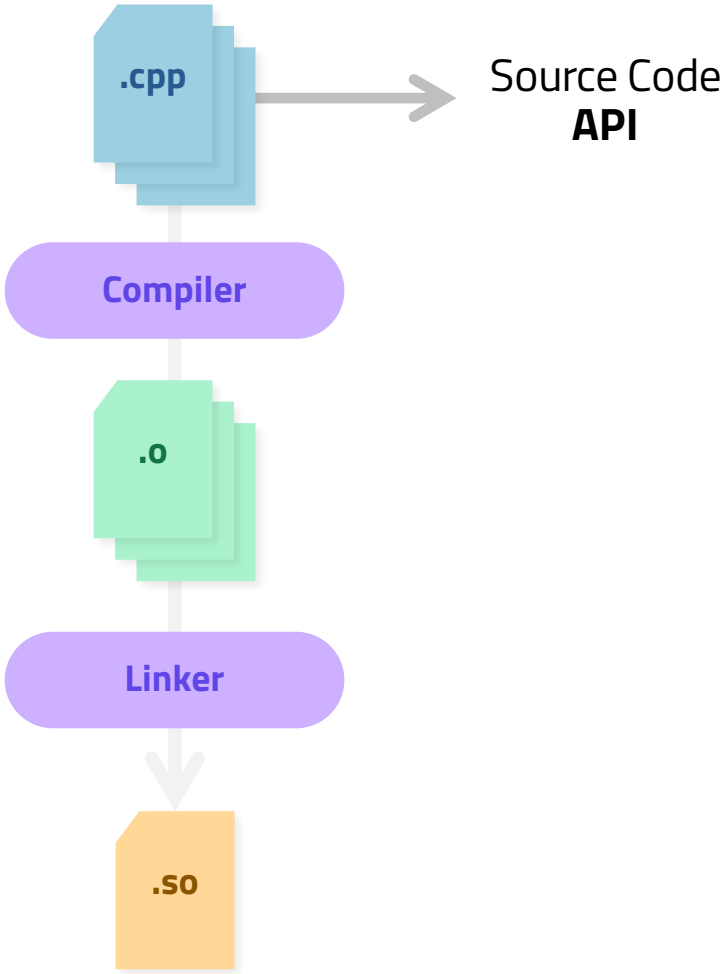
ABI



ABI

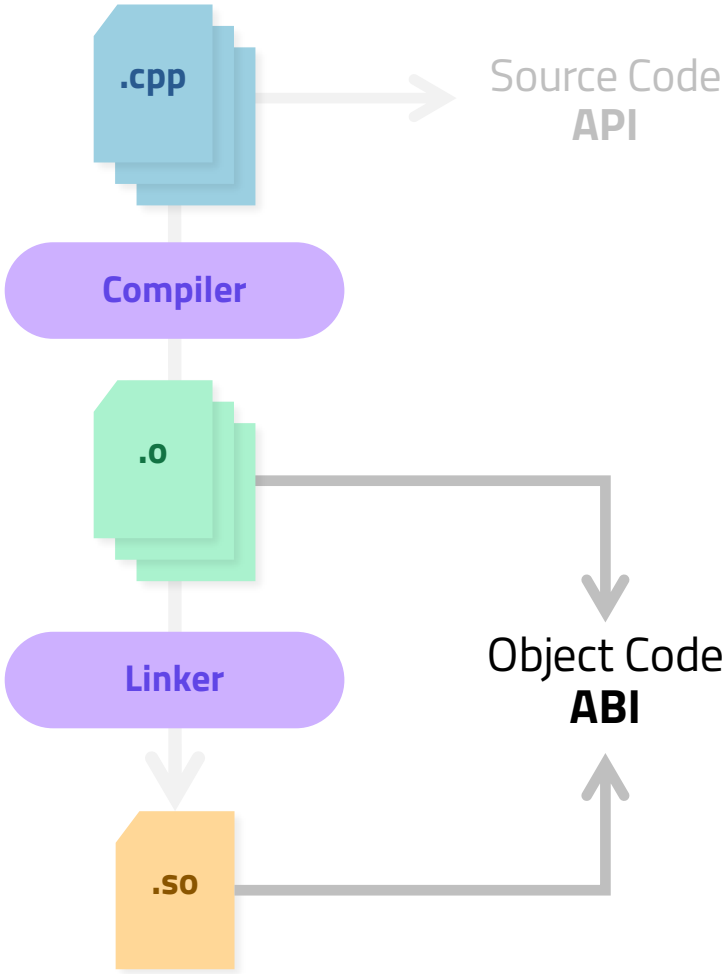


ABI



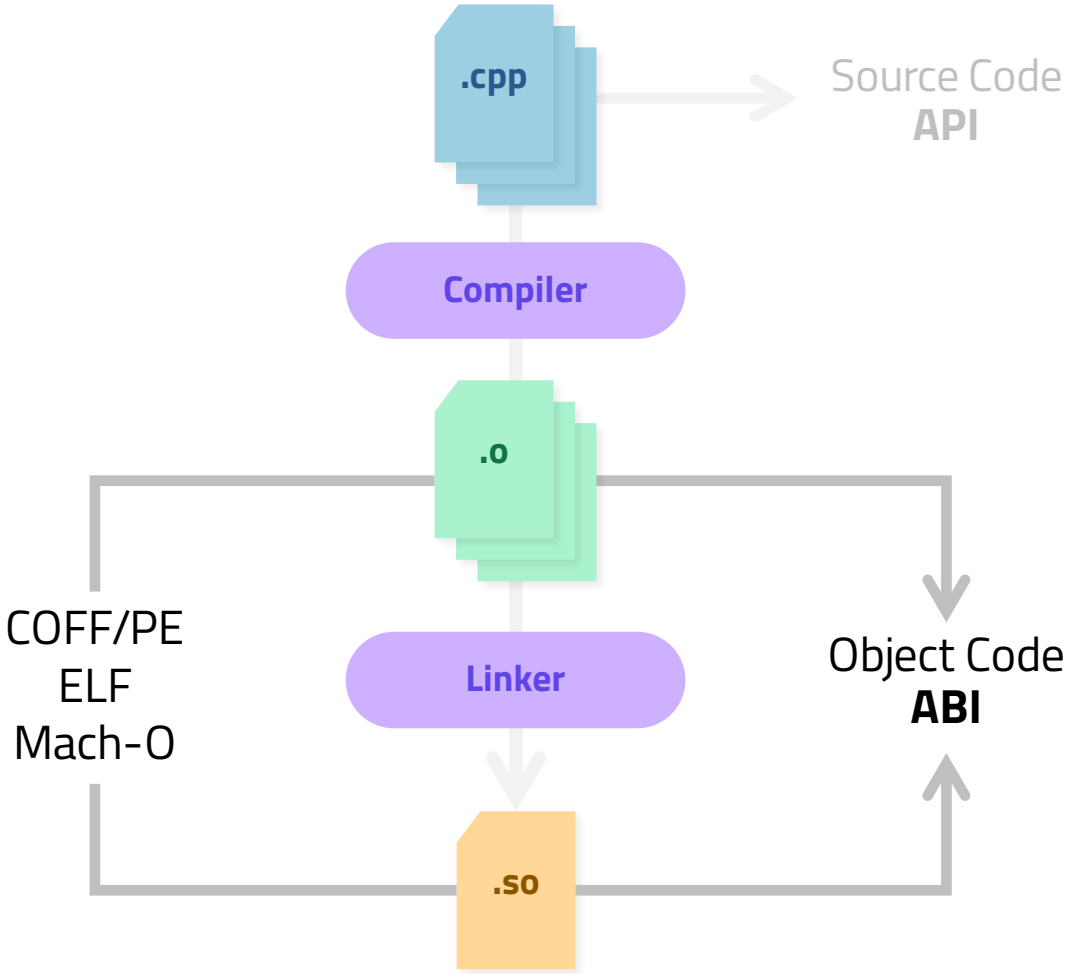
API is the **source code** level interface

ABI



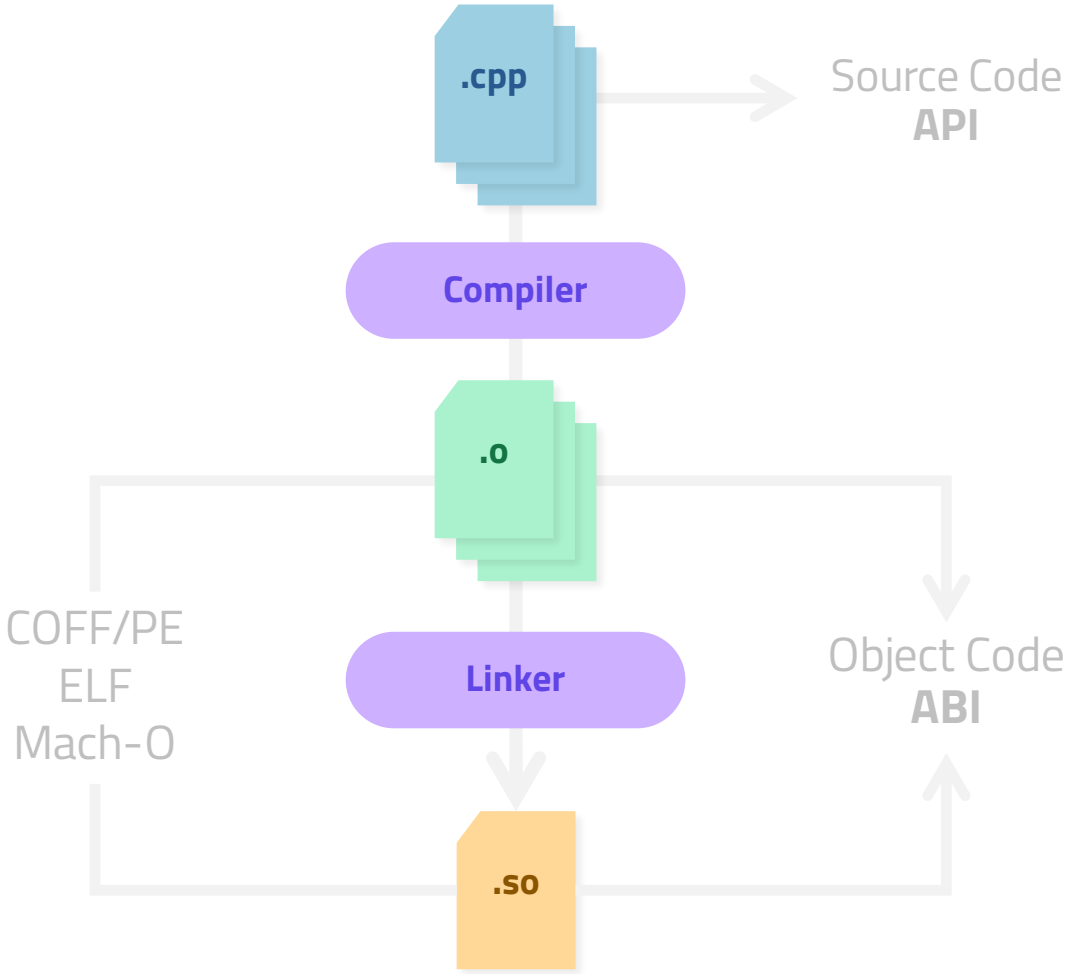
API is the **source code** level interface

ABI



API is the **source code** level interface

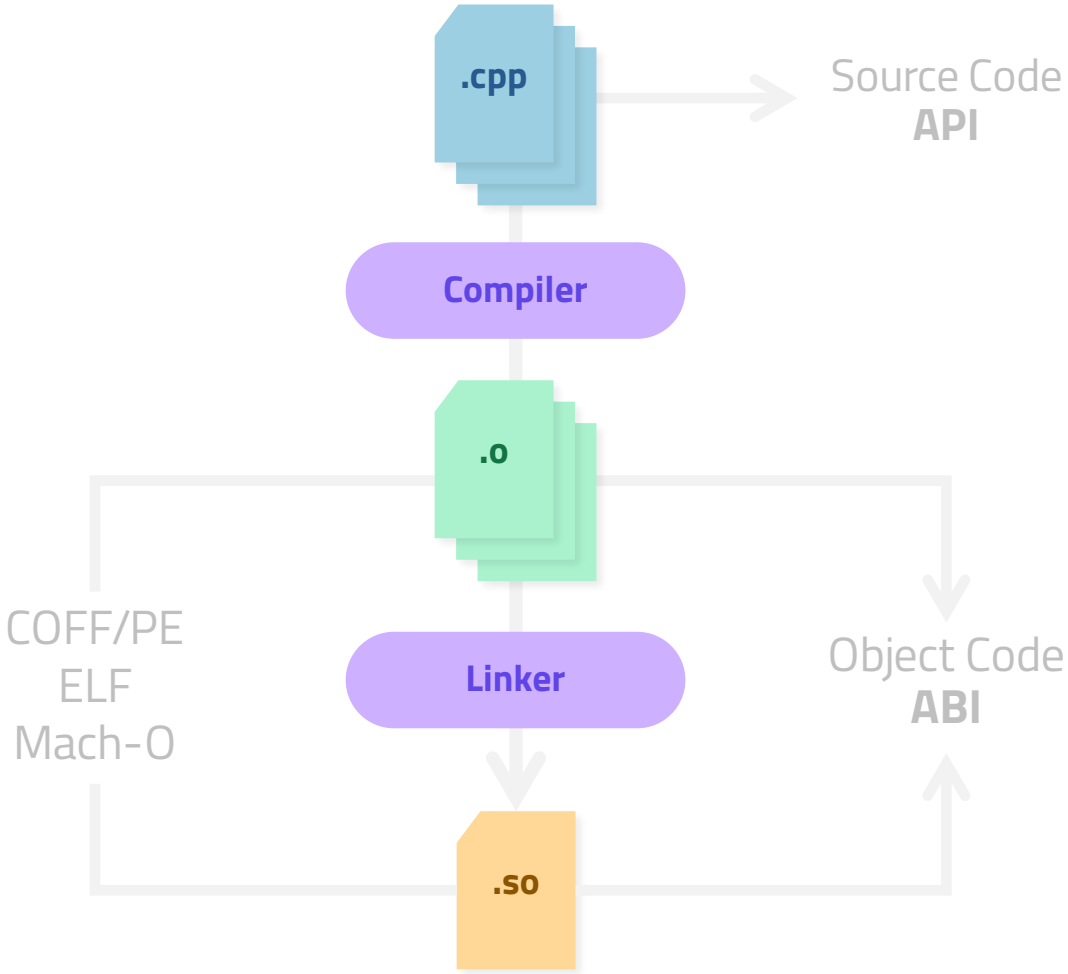
ABI



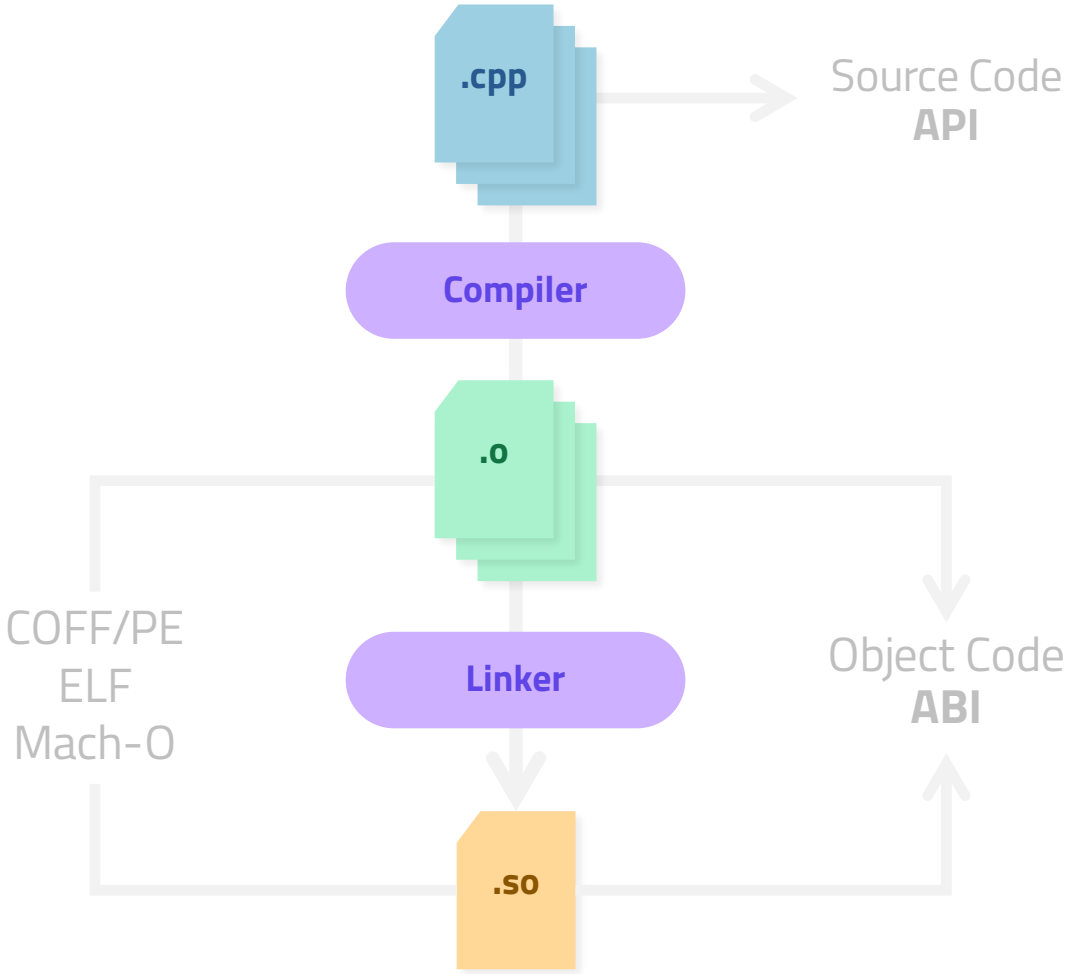
API is the **source code** level interface

ABI is the **object code** level interface

ABI

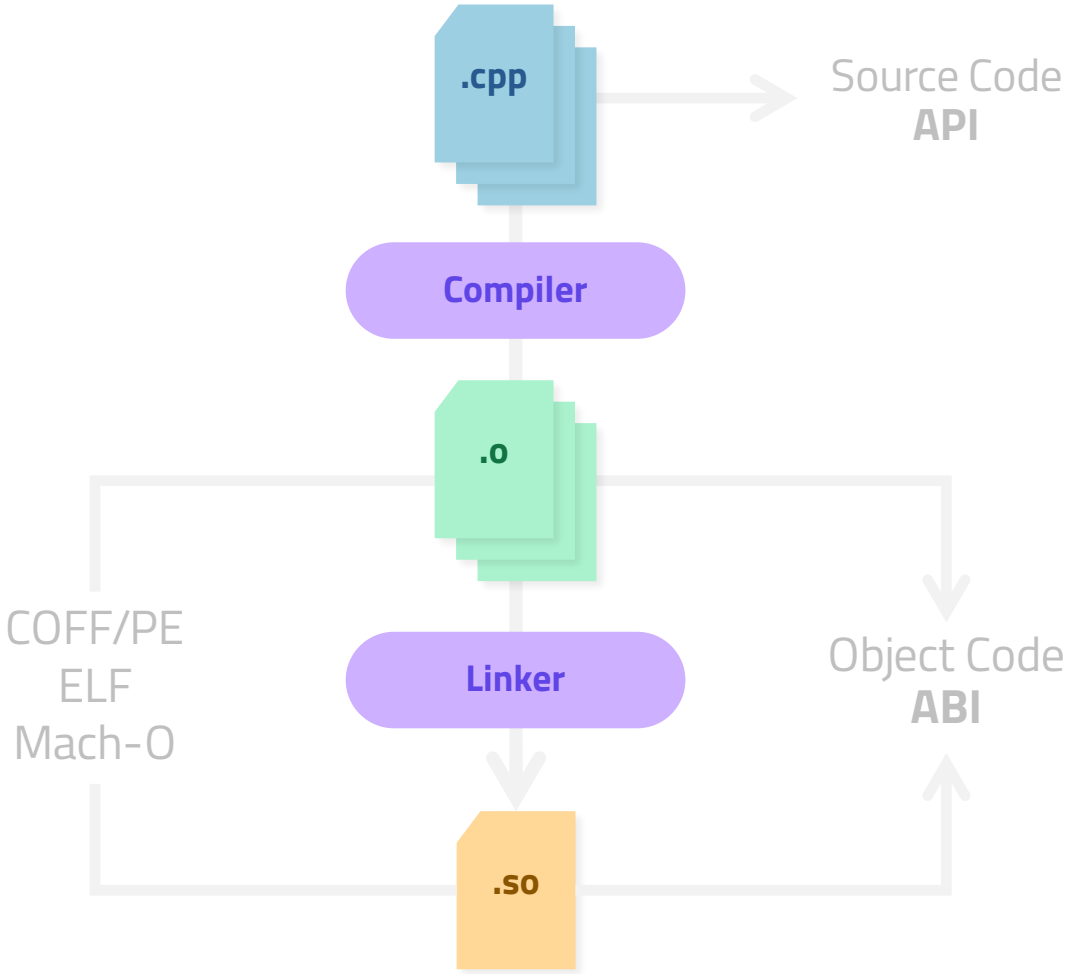


ABI



What's inside object files?

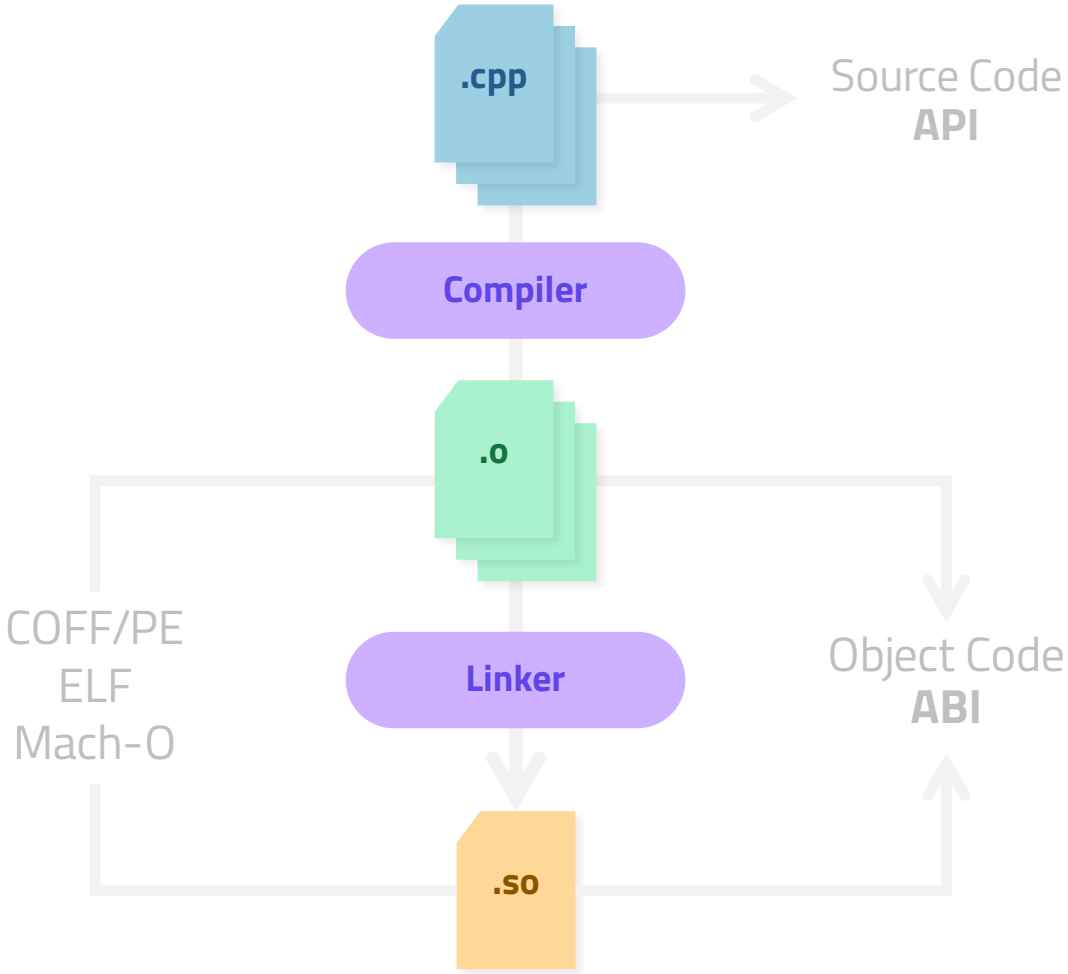
ABI



What's inside object files?

Machine Code

ABI

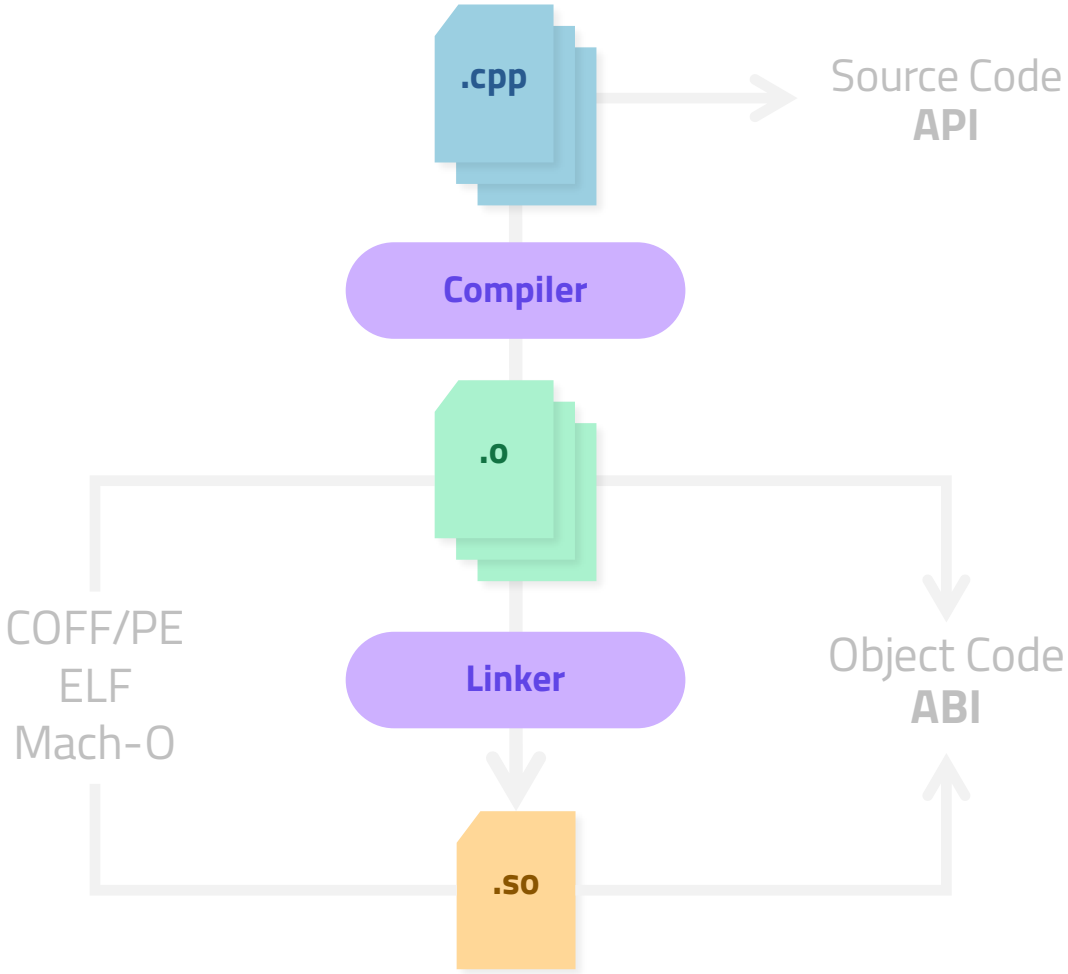


What's inside object files?

Machine Code

Static Data

ABI



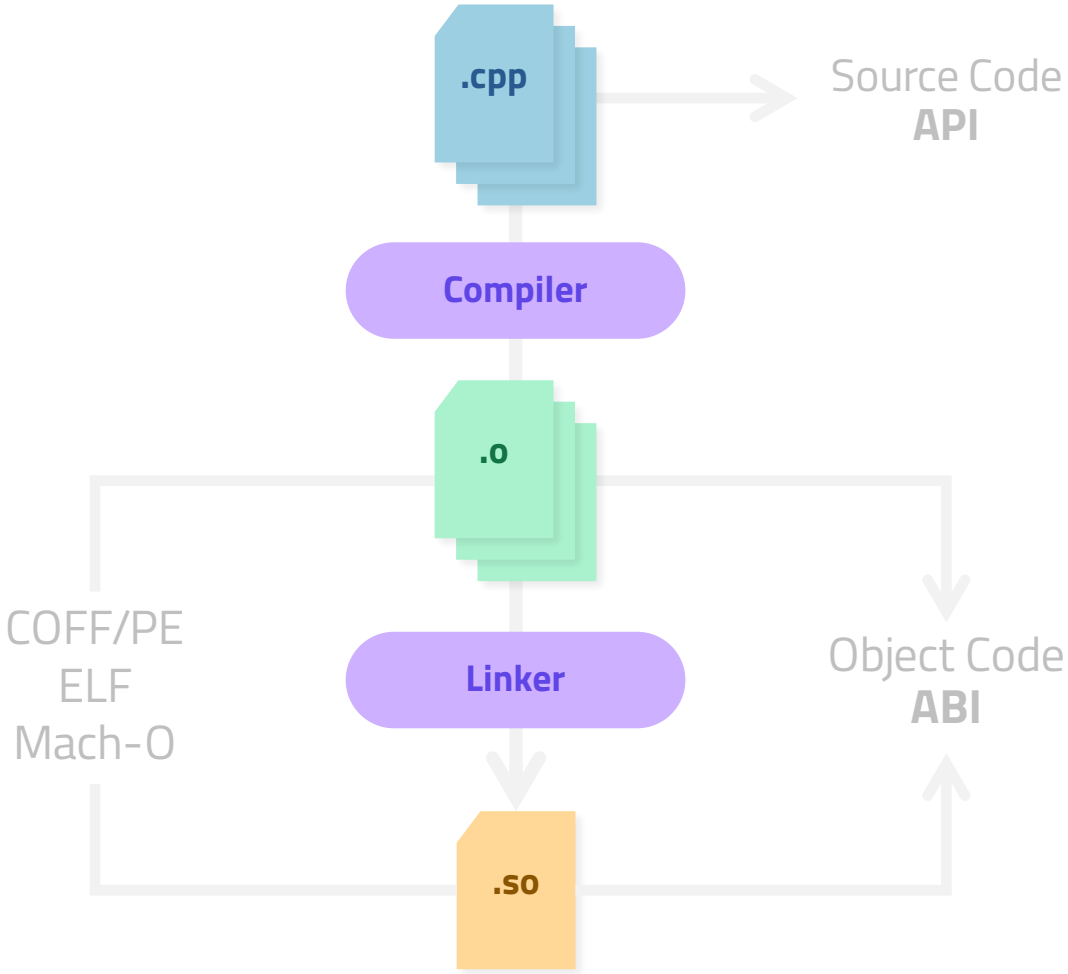
What's inside object files?

Machine Code

Static Data

Symbol Table

ABI



What's inside object files?

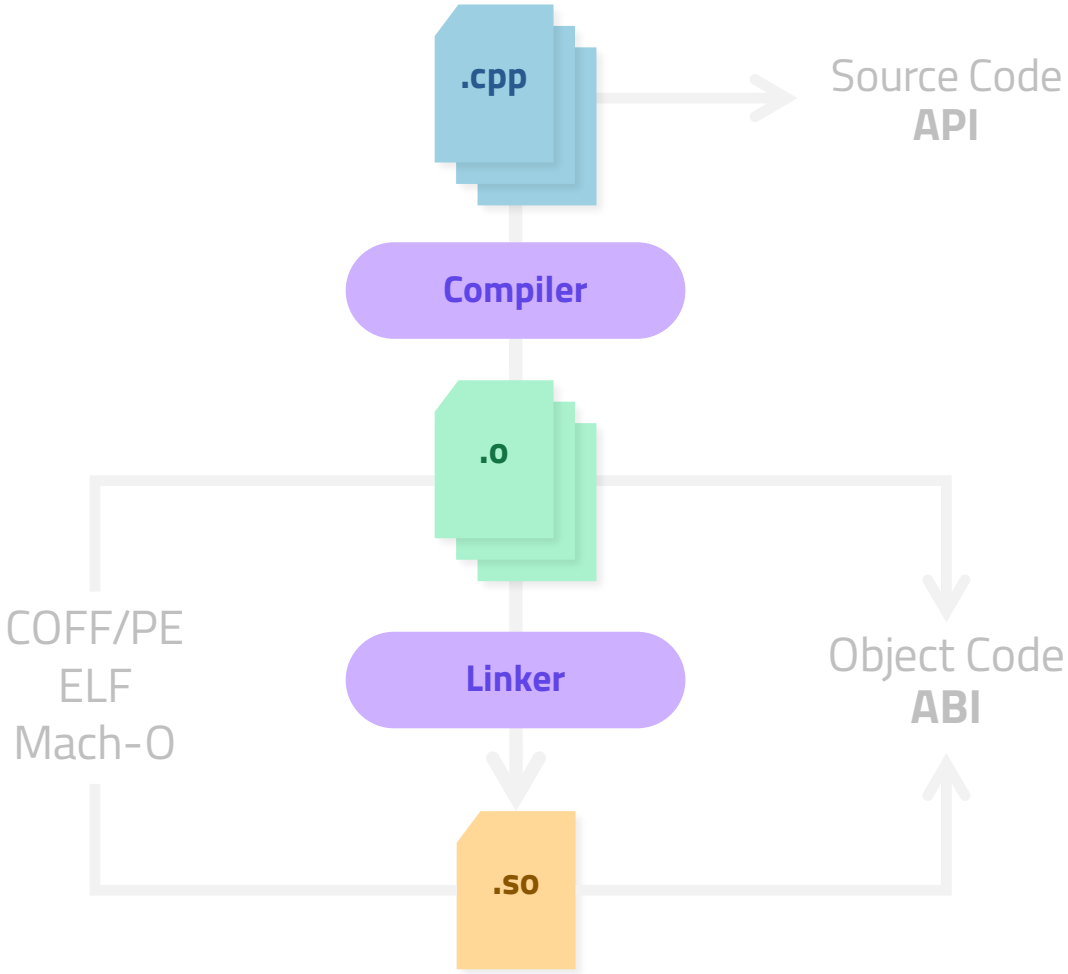
Machine Code

Static Data

Symbol Table

```
linux $ nm --extern-only lib.so | c++filt
```

ABI



What's inside object files?

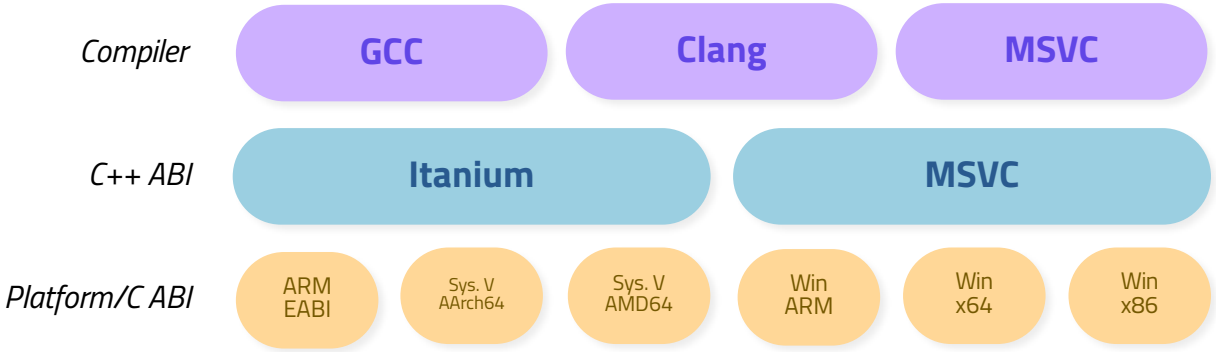
Machine Code

Static Data

Symbol Table

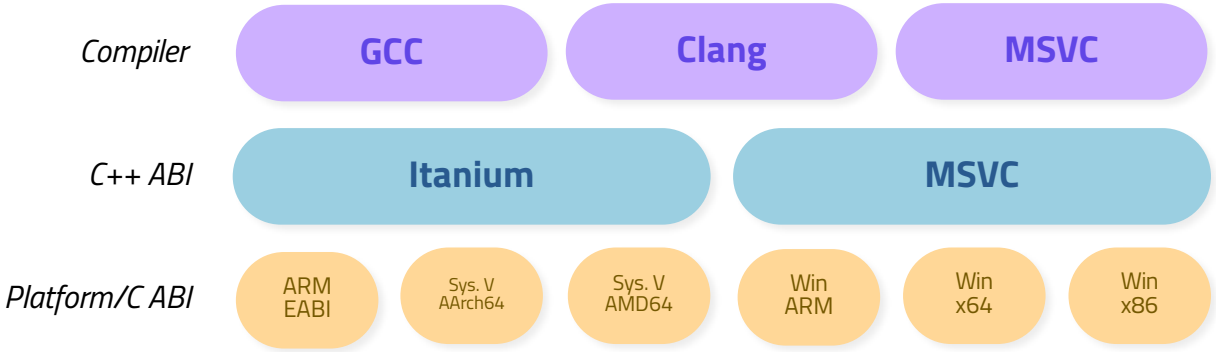
```
  linux $ nm --extern-only lib.so | c++filt

windows $ dumpbin /exports lib.dll
```

ABI

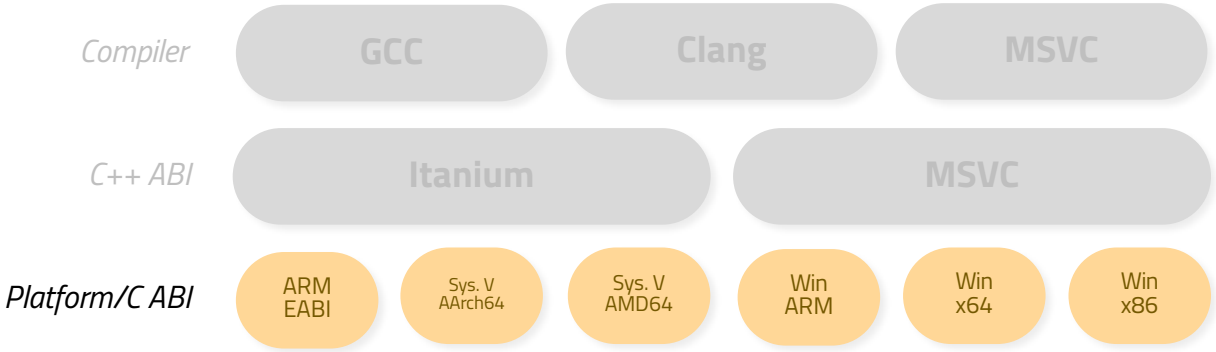


ABI

The Layers of ABI



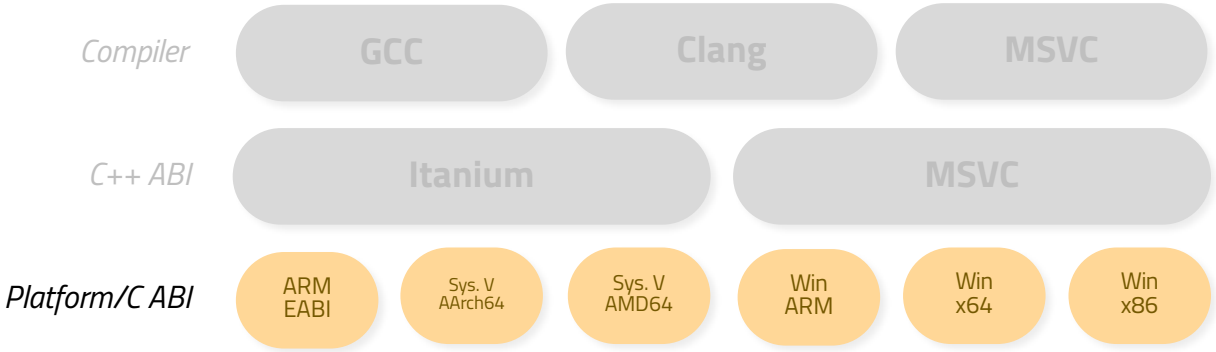
ABI



The Layers of ABI

Processor Architecture

ABI

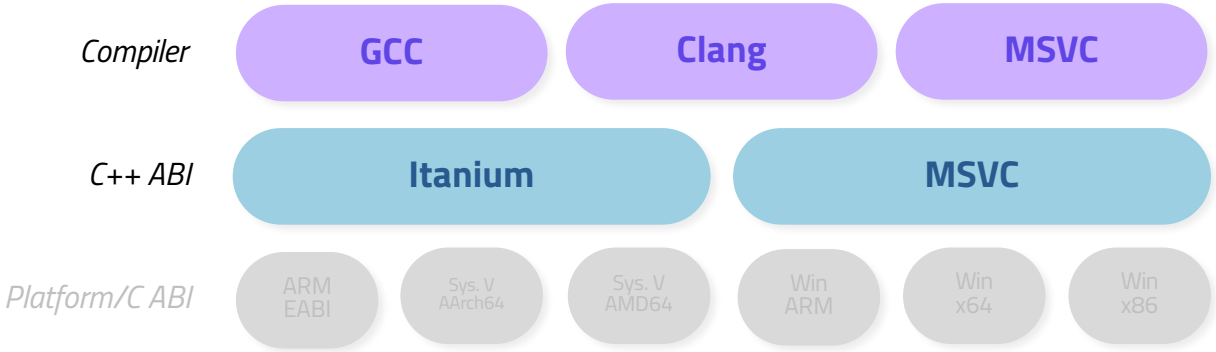


The Layers of ABI

Processor Architecture

Operating System

ABI



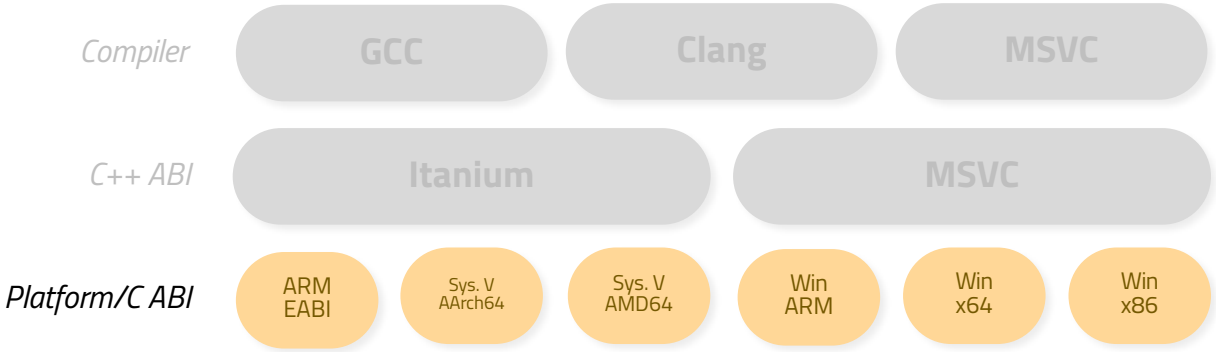
The Layers of ABI

Processor Architecture

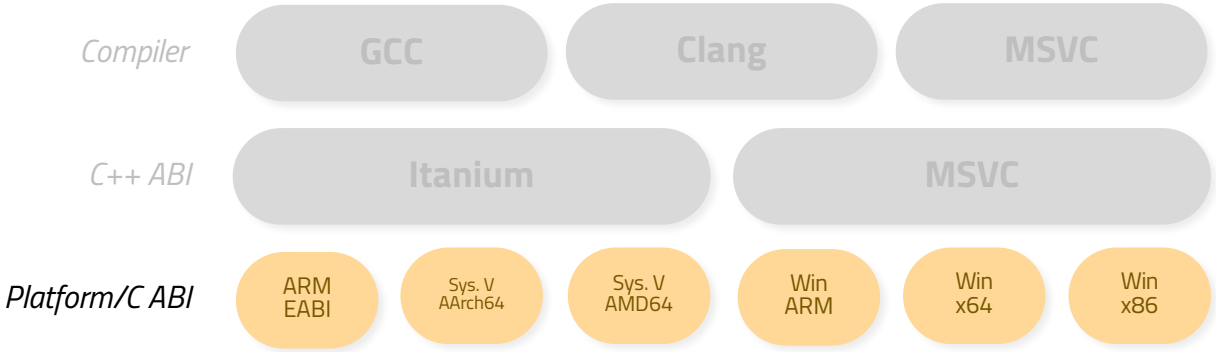
Operating System

Compiler

Platform/C ABI

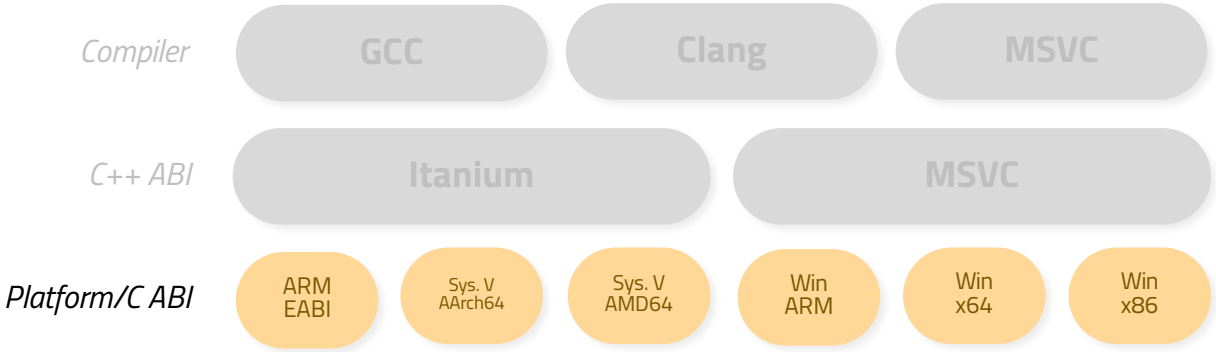


Platform/C ABI



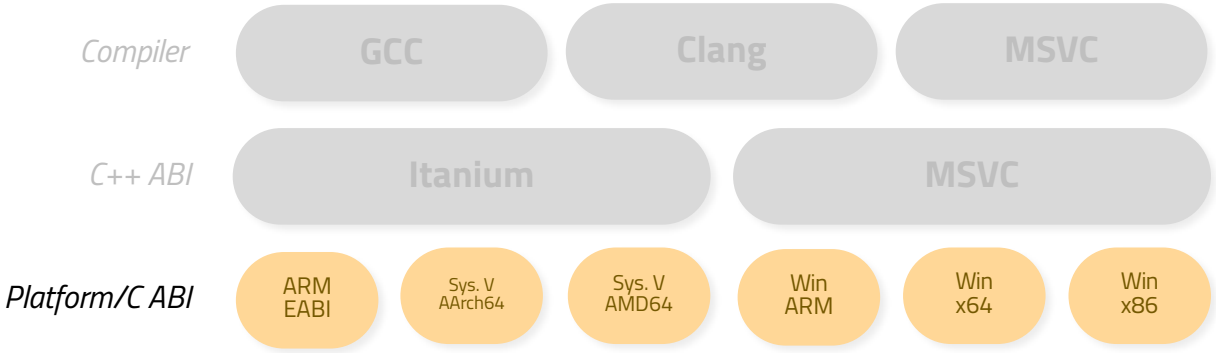
Platform/C ABI is the
lowest common denominator

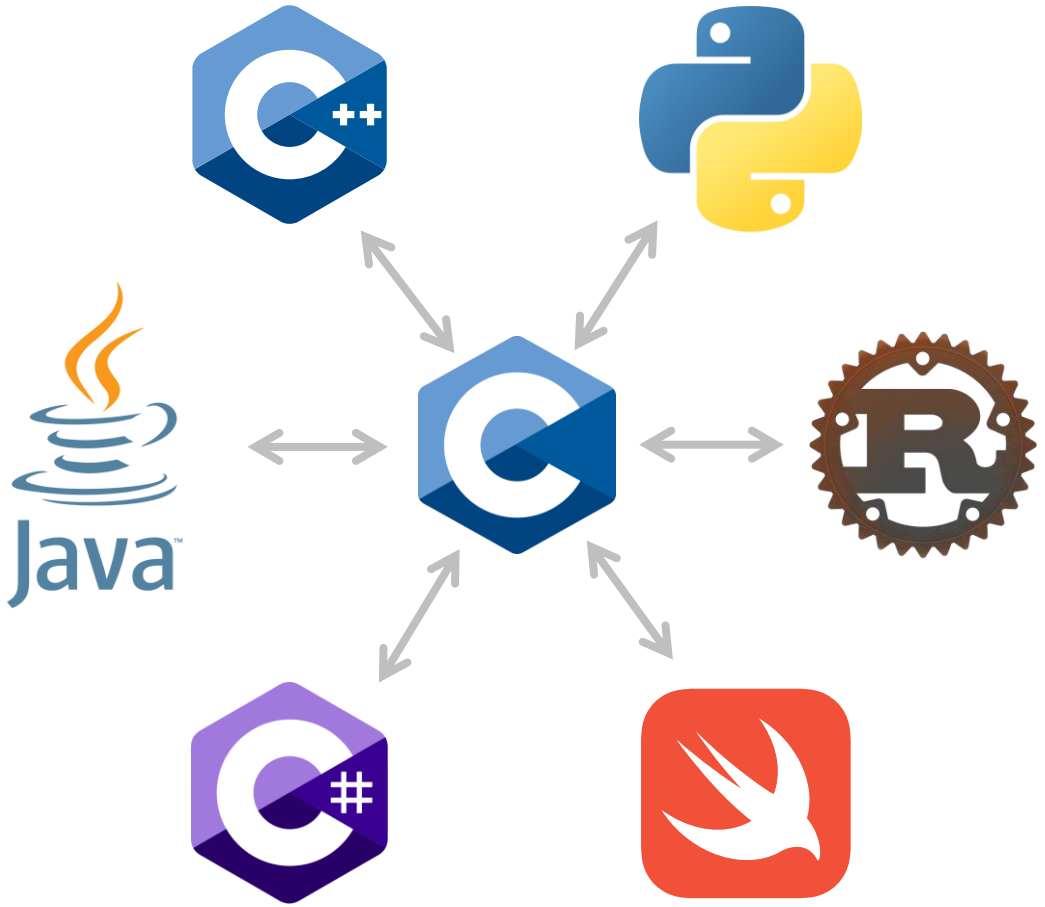
Platform/C ABI



Platform/C ABI is the
lowest common denominator

Platform/C ABI





Python/C

Python/C

```
01
02 import ctypes
03
04 libc = ctypes.cdll.LoadLibrary("libc.so.6")
05
06 libc.printf.restype = ctypes.c_int
07 libc.printf.argtypes = [
08     ctypes.c_char_p,
09     ctypes.c_char_p,
10 ]
11
12 libc.printf(b"Hello, %S\n", "World!")
13
14
```

Python ctypes

Python/C

```
01
02 import ctypes
03
04 libc = ctypes.cdll.LoadLibrary("libc.so.6")
05
06 libc.printf.restype = ctypes.c_int
07 libc.printf.argtypes = [
08     ctypes.c_char_p,
09     ctypes.c_char_p,
10 ]
11
12 libc.printf(b"Hello, %S\n", "World!")
13
14
```

Python ctypes

Load Shared Libraries

Python/C

```
01
02 import ctypes
03
04 libc = ctypes.cdll.LoadLibrary("libc.so.6")
05
06 libc.printf.restype = ctypes.c_int
07 libc.printf.argtypes = [
08     ctypes.c_char_p,
09     ctypes.c_char_p,
10 ]
11
12 libc.printf(b"Hello, %S\n", "World!")
13
14
```

Python ctypes

Basic C Types

Python/C

```
01
02 import ctypes
03
04 libc = ctypes.cdll.LoadLibrary("libc.so.6")
05
06 libc.printf.restype = ctypes.c_int
07 libc.printf.argtypes = [
08     ctypes.c_char_p,
09     ctypes.c_char_p,
10 ]
11
12 libc.printf(b"Hello, %S\n", "World!")
13
14
```

Python ctypes

C-like Function Signatures & Calls

Python/C

```
01
02
03 import ctypes
04
05 class Point(ctypes.Structure):
06     _fields_ = [
07         ("x", c_int),
08         ("y", c_int),
09     ]
10
11 point = Point(10, 20)
12
13
14
```

Python ctypes

Python/C

```
01
02
03 import ctypes
04
05 class Point(ctypes.Structure):
06     _fields_ = [
07         ("x", c_int),
08         ("y", c_int),
09     ]
10
11 point = Point(10, 20)
12
13
14
```

Python ctypes

C-like Structure Types

Python/C

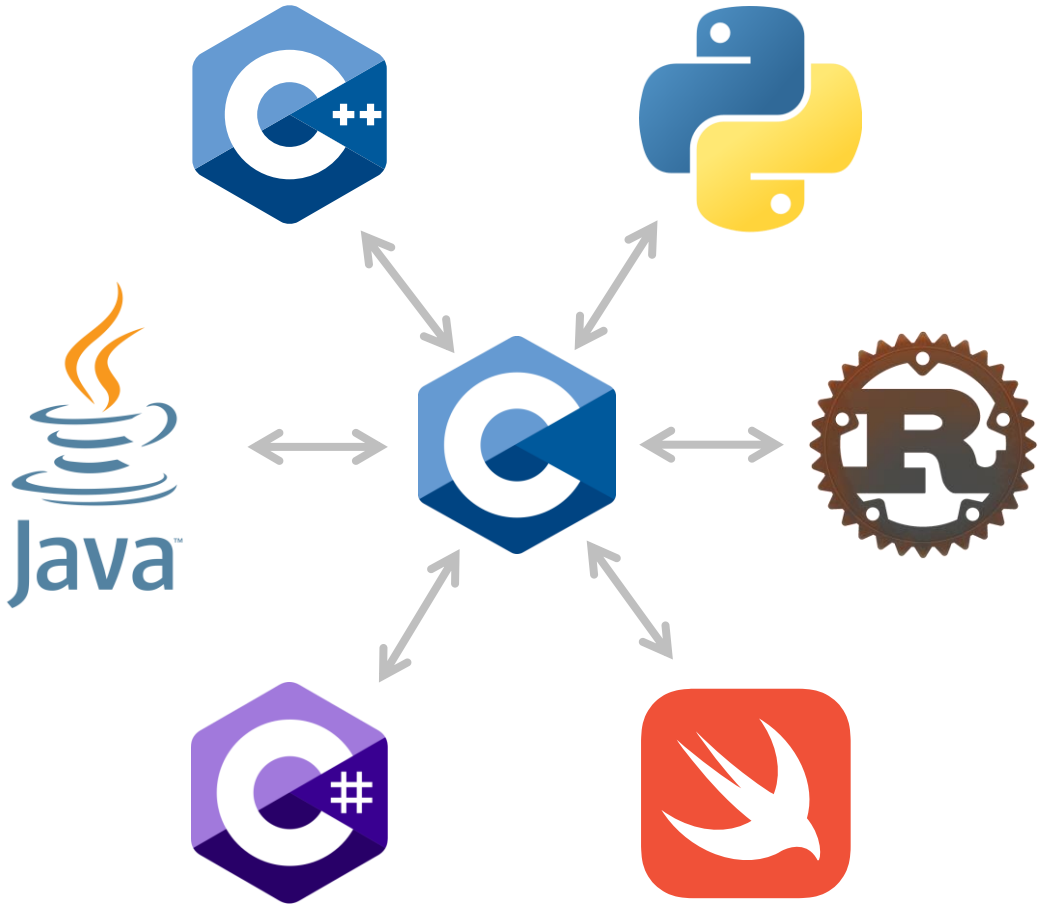
```
01
02
03 import ctypes
04
05 class Point(ctypes.Structure):
06     _fields_ = [
07         ("x", c_int),
08         ("y", c_int),
09     ]
10
11 point = Point(10, 20)
12
13
14
```

Python ctypes

Most languages have similar APIs

Platform/C ABI

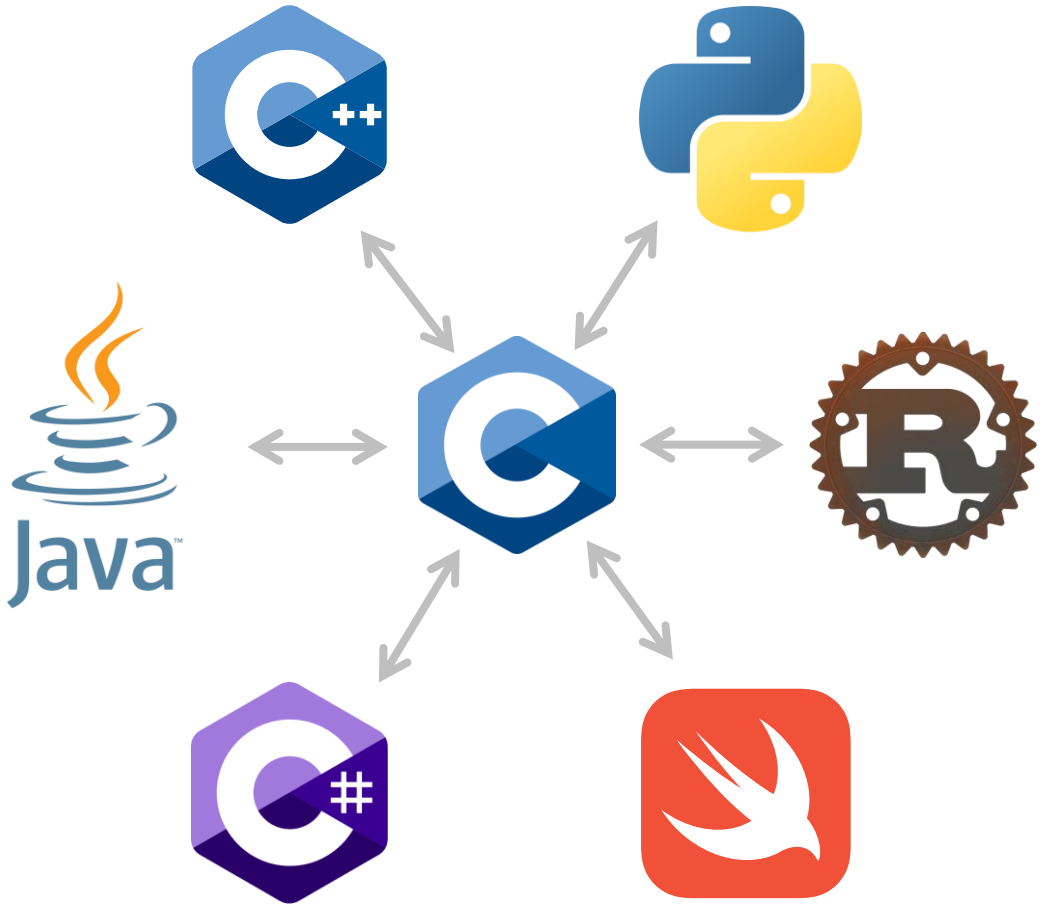
Platform/C ABI



Platform/C ABI

Platform/C ABI

A good foundation for cross-lang. interop.

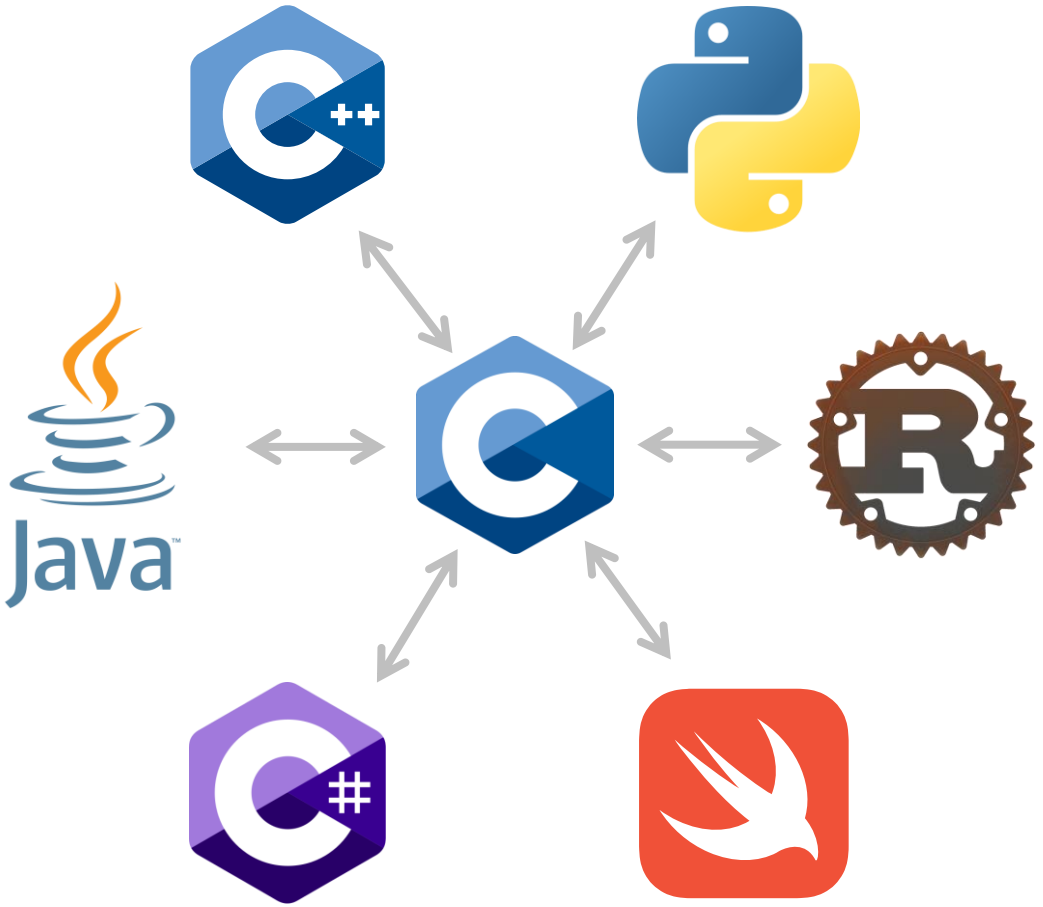


Platform/C ABI

Platform/C ABI

A good foundation for cross-lang. interop.

Calling Convention & Type Layout



Platform/C ABI: Calling Convention

Calling Convention

Calling Convention

Caller & Callee Saved Registers

Platform/C ABI

Calling Convention

Caller & Callee Saved Registers

Passing Parameters & Return Values

Platform/C ABI

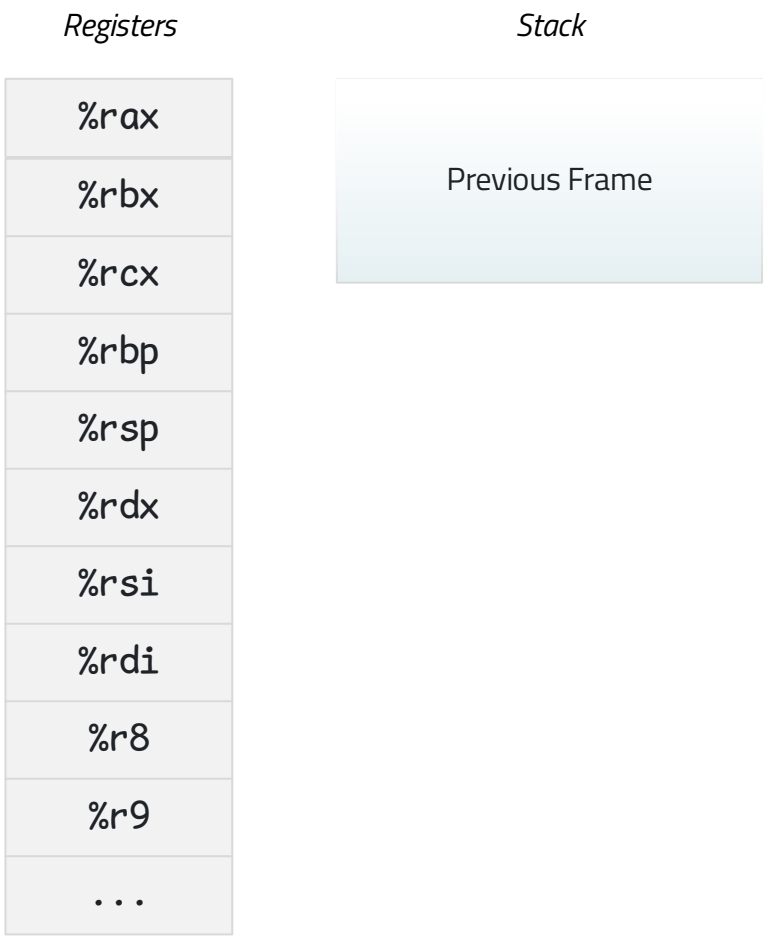
Calling Convention

Caller & Callee Saved Registers

Passing Parameters & Return Values

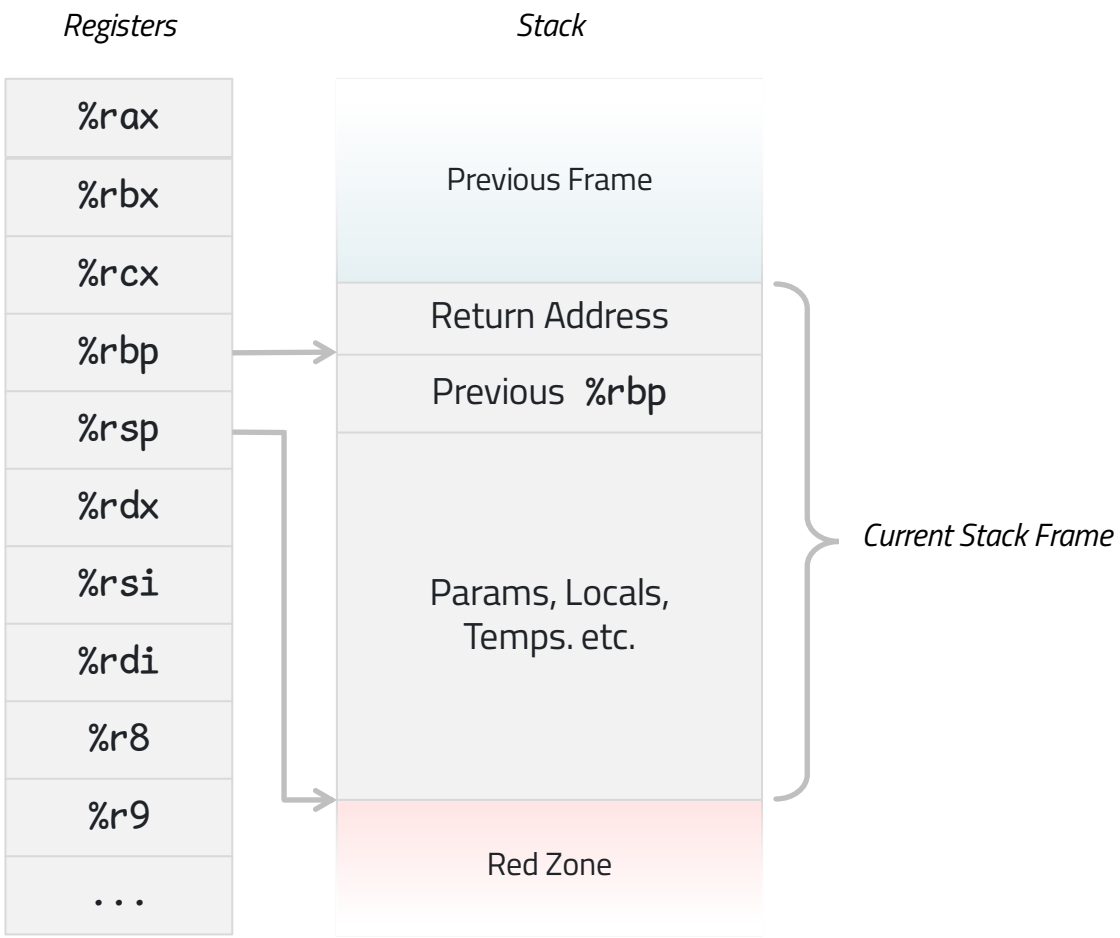
Stack Frame Layout

Platform/C ABI



System V AMD64 ABI

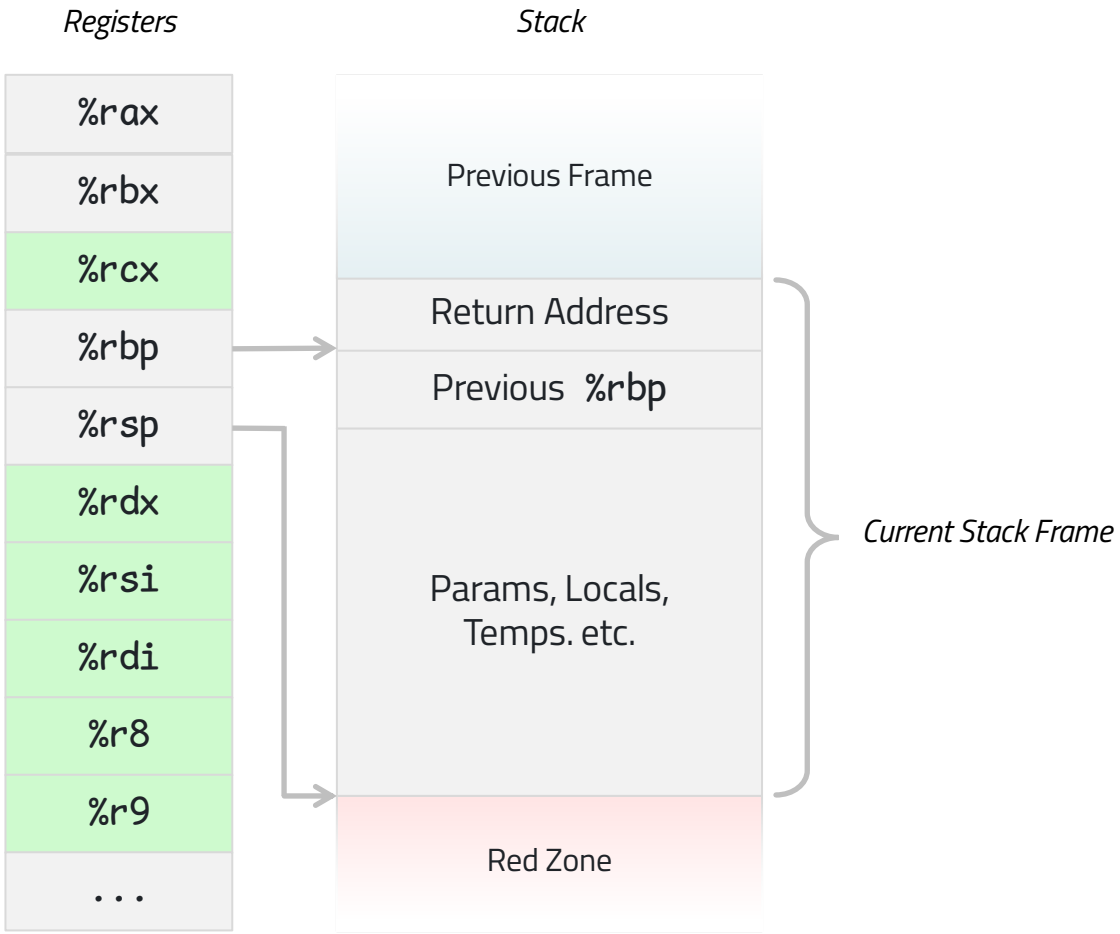
Platform/C ABI



System V AMD64 ABI

Push Return Address & Base-Pointer on Stack

Platform/C ABI

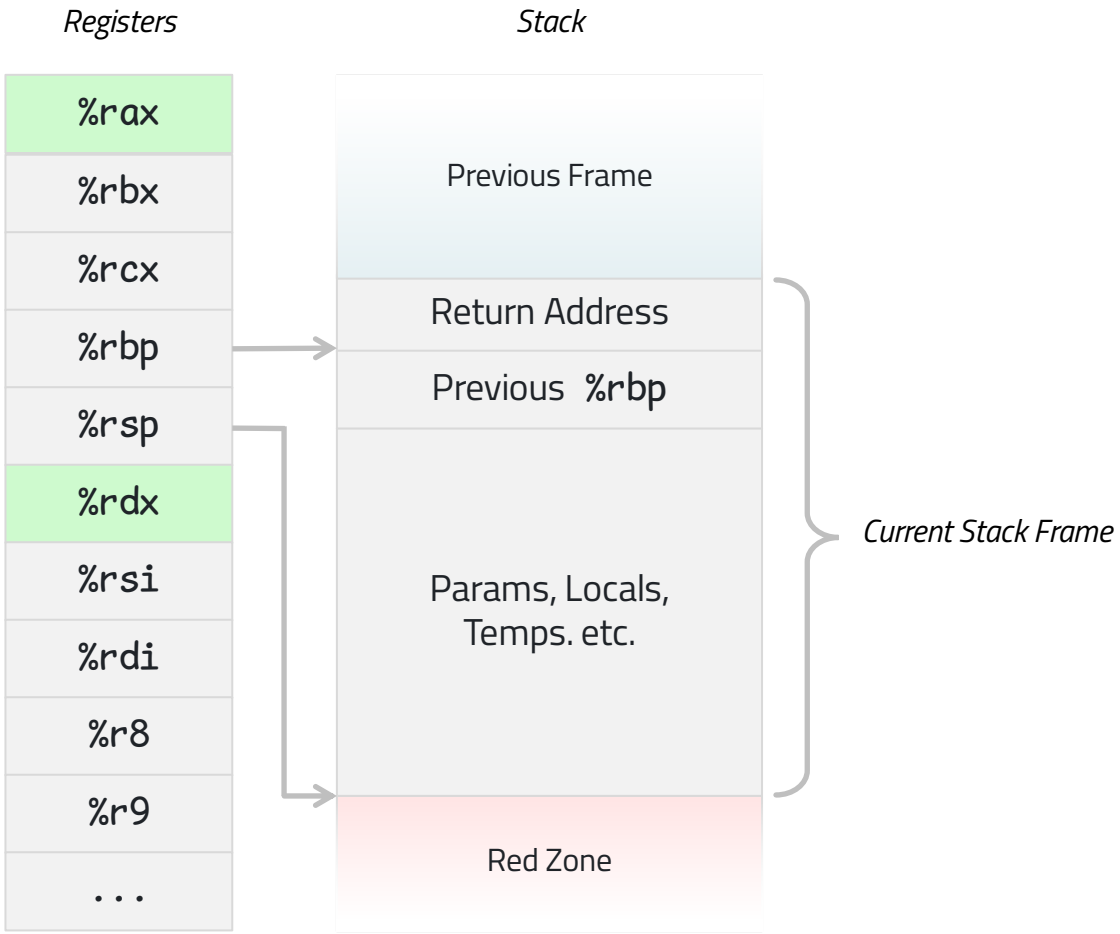


System V AMD64 ABI

Push Return Address & Base-Pointer on Stack

Parameters in `%rdi %rsi %rdx %rcx %r8 %r9*`

Platform/C ABI



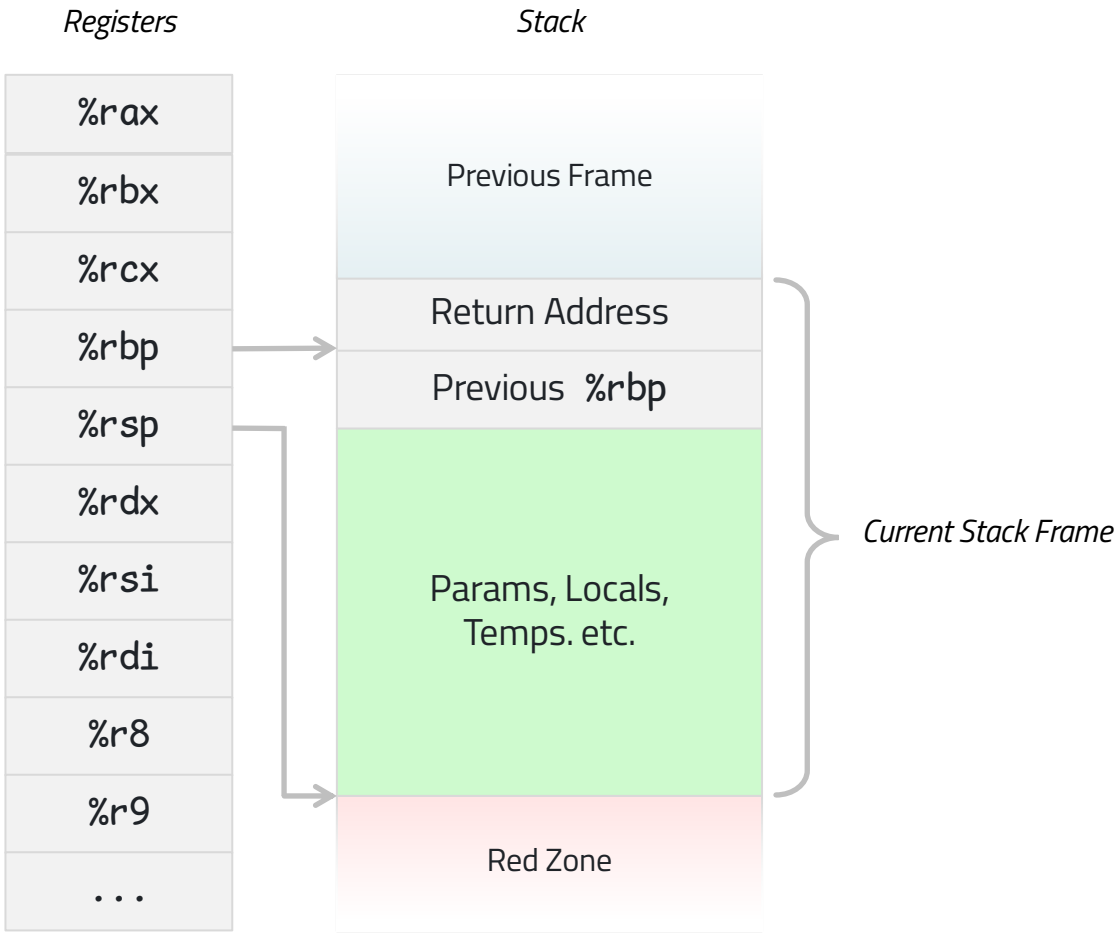
System V AMD64 ABI

Push Return Address & Base-Pointer on Stack

Parameters in `%rdi %rsi %rdx %rcx %r8 %r9`*

Return Values in `%rax %rdx`**

Platform/C ABI



System V AMD64 ABI

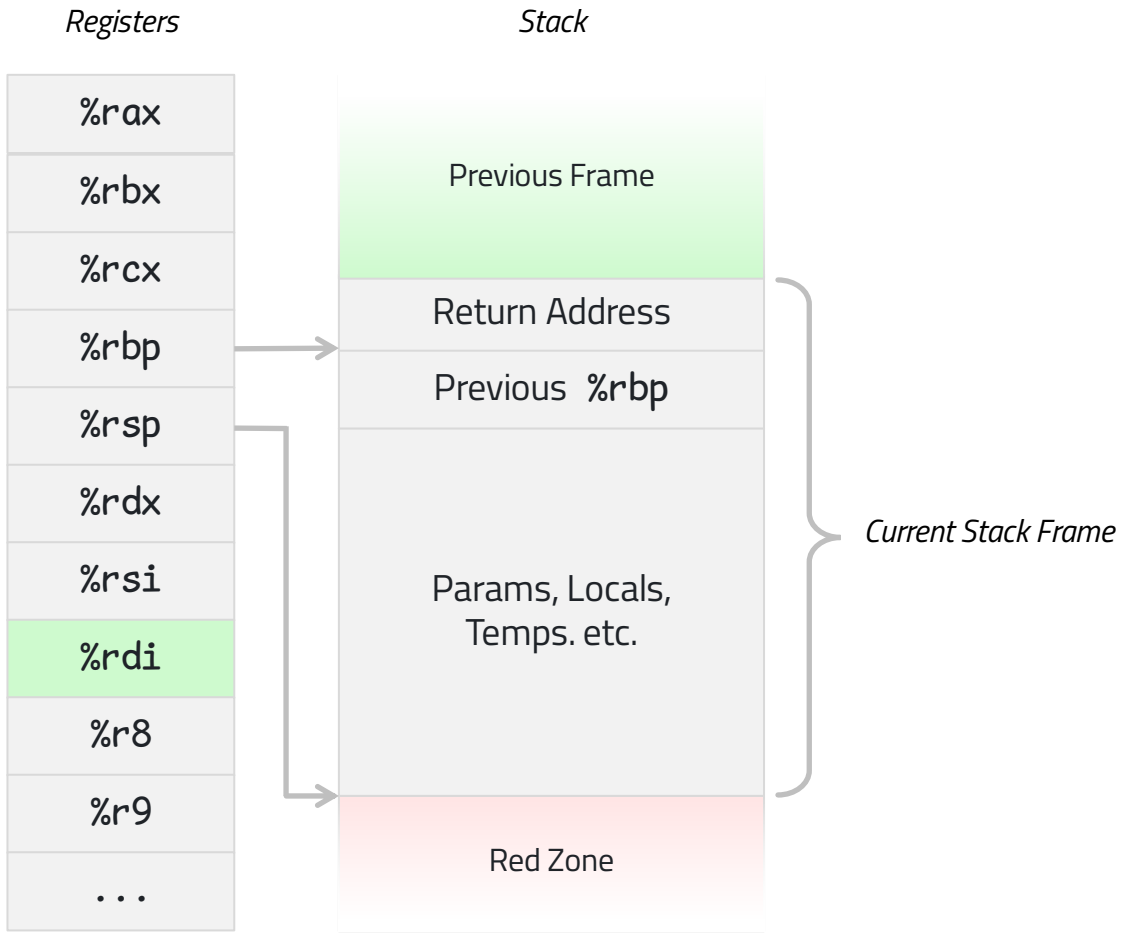
Push Return Address & Base-Pointer on Stack

Parameters in `%rdi %rsi %rdx %rcx %r8 %r9`*

Return Values in `%rax %rdx`**

*Large params spill onto stack

Platform/C ABI



System V AMD64 ABI

Push Return Address & Base-Pointer on Stack

Parameters in `%rdi %rsi %rdx %rcx %r8 %r9`*

Return Values in `%rax %rdx`**

*Large params spill onto stack

**Large return values & non-trivial types by ref.

Platform/C ABI

C++ ABI: Non-trivial

Platform/C ABI

C++ ABI: Non-trivial

Non-trivial *for the purposes of calls**

C++ ABI: Non-trivial

Non-trivial *for the purposes of calls**

**This is different from* `std::is_trivial`*,
which is deprecated in C++26 (P3247).*

Platform/C ABI

C++ ABI: Non-trivial

Non-trivial *for the purposes of calls**

Non-trivial copy-constructor, move-constructor, or destructor

**This is different from `std::is_trivial`, which is deprecated in C++26 (P3247).*

Platform/C ABI

C++ ABI: Non-trivial

Non-trivial *for the purposes of calls**

Non-trivial copy-constructor, move-constructor, or destructor

Or all copy and move constructors are deleted

**This is different from `std::is_trivial`, which is deprecated in C++26 (P3247).*

Platform/C ABI

```
01
02
03
04
05 #ifndef _MSC_VER
06 #  define __stdcall __attribute__((stdcall))
07 #endif
08
09 extern "C" void __stdcall baz();
10
11
12
13
14
```

C++ ABI: Calling Convention

Platform/C ABI

```
01
02
03
04
05 #ifndef _MSC_VER
06 #  define __stdcall __attribute__((stdcall))
07 #endif
08
09 extern "C" void __stdcall baz();
10
11
12
13
14
```

C++ ABI: Calling Convention

No Calling Convention in C++ Standard

Platform/C ABI

```
01
02
03
04
05 #ifndef _MSC_VER
06 #  define __stdcall __attribute__((stdcall))
07 #endif
08
09 extern "C" void __stdcall baz();
10
11
12
13
14
```

C++ ABI: Calling Convention

No Calling Convention in C++ Standard

Implementation defined keywords/attributes

Platform/C ABI

```
01
02
03
04
05 #ifndef _MSC_VER
06 #  define __stdcall __attribute__((stdcall))
07 #endif
08
09 extern "C" void __stdcall baz();
10
11
12
13
14
```

C++ ABI: Calling Convention

No Calling Convention in C++ Standard

Implementation defined keywords/attributes

`__stdcall`

Platform/C ABI

```
01
02
03
04
05 #ifndef _MSC_VER
06 #  define __stdcall __attribute__((stdcall))
07 #endif
08
09 extern "C" void __stdcall baz();
10
11
12
13
14
```

C++ ABI: Calling Convention

No Calling Convention in C++ Standard

Implementation defined keywords/attributes

`__stdcall`

`__attribute__((stdcall))`

Platform/C ABI

```
01
02
03 extern "C" void foo();
04 void foo();
05
06 namespace bar {
07     extern "C" void foo();
08     void foo();
09 }
10
11 extern "C" foo(std::string str);
12 extern "C" bar(int &num);
13
14
```

C++ ABI: Language Linkage

Platform/C ABI

```
01
02
03 extern "C" void foo();
04 void foo();
05
06 namespace bar {
07     extern "C" void foo();
08     void foo();
09 }
10
11 extern "C" foo(std::string str);
12 extern "C" bar(int &num);
13
14
```

C++ ABI: Language Linkage

`extern "C"` is a language linkage specification

Platform/C ABI

```
01
02
03 extern "C" void foo();
04 void foo();
05
06 namespace bar {
07     extern "C" void foo();
08     void foo();
09 }
10
11 extern "C" foo(std::string str);
12 extern "C" bar(int &num);
13
14
```

C++ ABI: Language Linkage

`extern "C"` is a language linkage specification

Names refer to the same entity

Platform/C ABI

```
01
02
03 extern "C" void foo();
04 void foo();
05
06 namespace bar {
07     extern "C" void foo();
08     void foo();
09 }
10
11 extern "C" foo(std::string str);
12 extern "C" bar(int &num);
13
14
```

C++ ABI: Language Linkage

`extern "C"` is a language linkage specification

Names refer to the same entity

No effect on calling convention

Platform/C ABI

```
01
02
03 extern "C" void foo();
04 void foo();
05
06 namespace bar {
07     extern "C" void foo();
08     void foo();
09 }
10
11 extern "C" foo(std::string str);
12 extern "C" bar(int &num);
13
14
```

C++ ABI: Language Linkage

`extern "C"` is a language linkage specification

Names refer to the same entity

No effect on calling convention

C++ language linkage is the default

Calling Convention

Platform/C ABI

Calling Convention

Platforms usually have a default

Platform/C ABI

Calling Convention

Platforms usually have a default

Default is usually denoted *C* or `cdecl`

Platform/C ABI

Calling Convention

Platforms usually have a default

Default is usually denoted *C* or `cdecl`

Windows x86 has `stdcall`, `fastcall`, `thiscall`

Platform/C ABI: Type Layout

Platform/C ABI

Type Layout

Platform/C ABI

Type Layout

Few guarantees in the C Standard

Platform/C ABI

Type Layout

Few guarantees in the C Standard

A `char` is 1 byte (`CHAR_BIT` bits) with weakest (usually 1) alignment

Platform/C ABI

Type Layout

Few guarantees in the C Standard

A `char` is 1 byte (`CHAR_BIT` bits) with weakest (usually 1) alignment

`struct` members are laid out in declaration order

Platform/C ABI

Type Layout

Few guarantees in the C Standard

A `char` is 1 byte (`CHAR_BIT` bits) with weakest (usually 1) alignment

`struct` members are laid out in declaration order

`structs` can have padding in the middle or at the end

Platform/C ABI

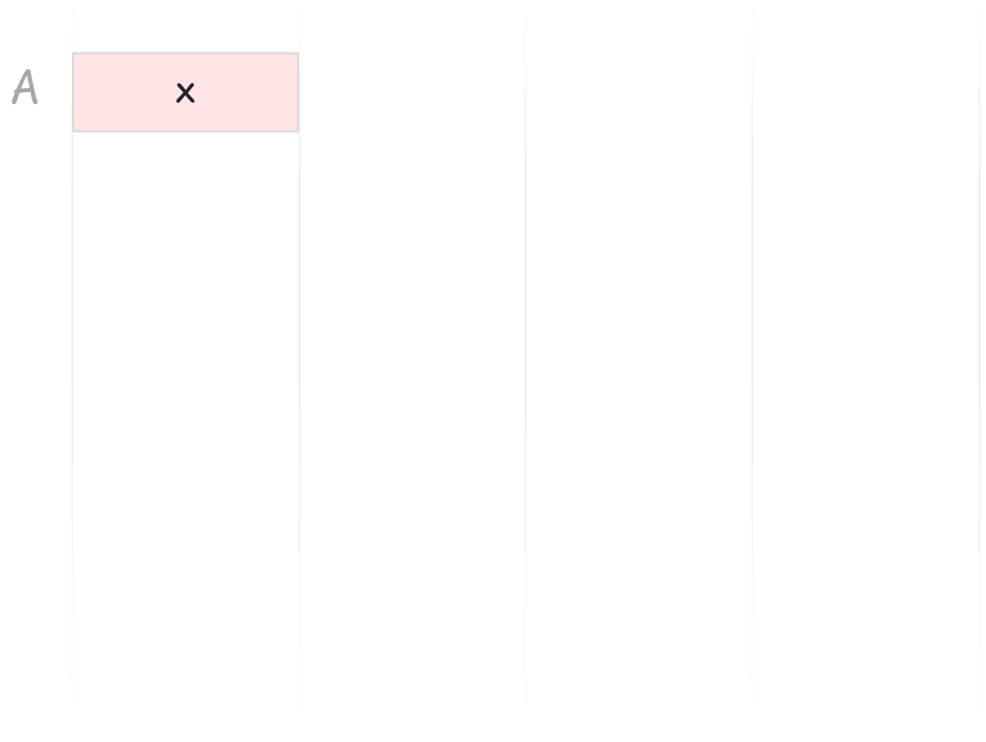
```
01
02
03
04 struct A { char x; };
05
06
07 struct B { char x; int y; };
08
09
10 struct C { char x; int y; char z; };
11
12
13
14
```

Type Layout

Platform/C ABI

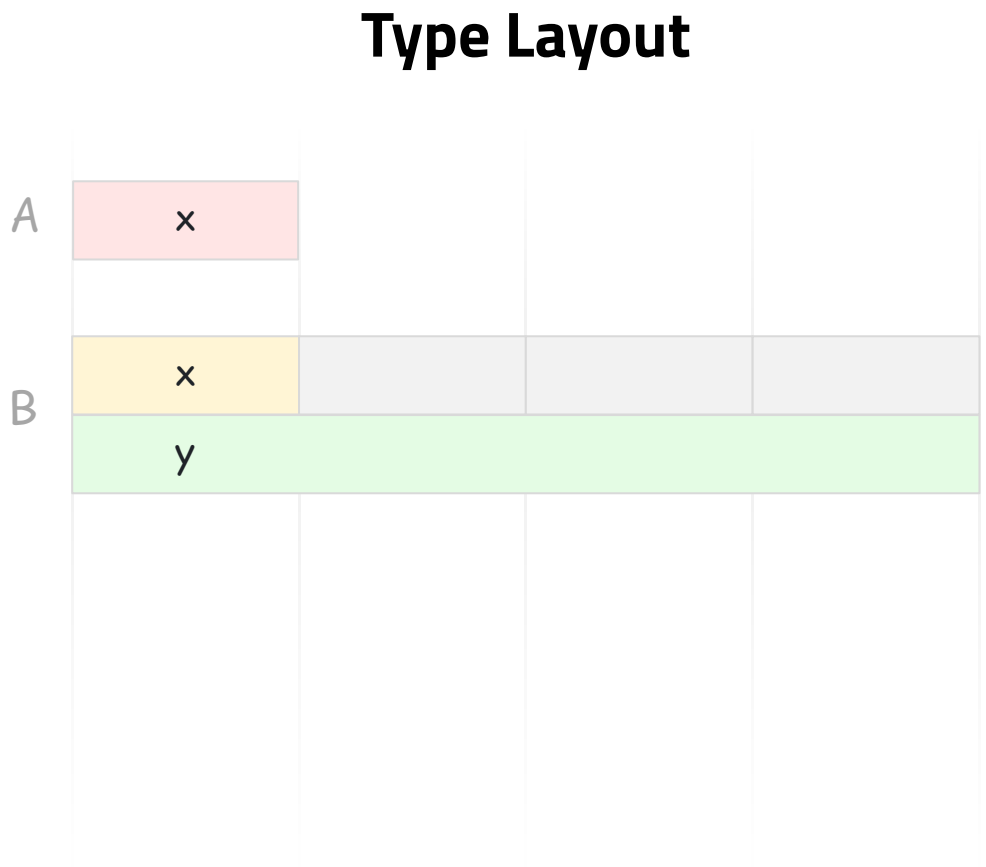
```
01
02
03
04 struct A { char x; };
05
06
07 struct B { char x; int y; };
08
09
10 struct C { char x; int y; char z; };
11
12
13
14
```

Type Layout



Platform/C ABI

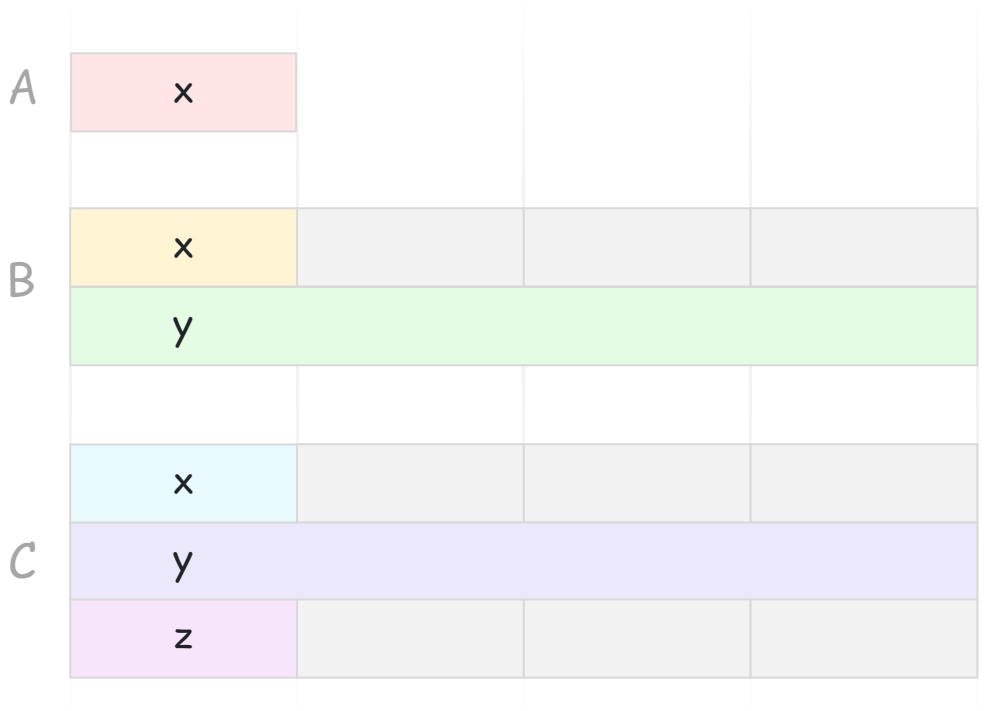
```
01
02
03
04 struct A { char x; };
05
06
07 struct B { char x; int y; };
08
09
10 struct C { char x; int y; char z; };
11
12
13
14
```



Platform/C ABI

```
01
02
03
04 struct A { char x; };
05
06
07 struct B { char x; int y; };
08
09
10 struct C { char x; int y; char z; };
11
12
13
14
```

Type Layout



Platform/C ABI

```
01
02
03
04 struct A { char x; };
05
06
07 struct B { char x; int y; };
08
09
10 struct C { char x; int y; char z; };
11
12
13
14
```

Type Layout

Platform/C ABI

```
01
02
03
04 struct A { char x; };
05 static_assert(sizeof(struct A) == 1);
06
07 struct B { char x; int y; };
08 static_assert(sizeof(struct B) == 8);
09
10 struct C { char x; int y; char z; };
11 static_assert(sizeof(struct C) == 12);
12
13
14
```

Type Layout

Use `static_assert` to guarantee type layout

Platform/C ABI

```
01
02
03
04
05 #include <stddef.h>
06
07 struct B { char x; int y; };
08 static_assert(offsetof(struct B, y) == 4);
09
10
11
12
13
14
```

Type Layout

Use `static_assert` to guarantee type layout

Use `offsetof` for member offsets

Platform/C ABI

```cpp
#include <type_traits>

struct A { int m; };
static_assert(std::is_standard_layout_v<A> == true);
```

C++ Standard-Layout Types

Platform/C ABI

```cpp
#include <type_traits>

struct A { int m; };
static_assert(std::is_standard_layout_v<A> == true);
```

C++ Standard-Layout Types

Standard-layout classes are useful for communication with code written in other programming languages. – *[class.prop]p6*

Platform/C ABI

```cpp
#include <type_traits>

struct A { int m; };
static_assert(std::is_standard_layout_v<A> == true);
```

C++ Standard-Layout Types

Standard-layout classes are useful for communication with code written in other programming languages. – *[class.prop]p6*

All NSDMs are also standard-layout

Platform/C ABI

```cpp
#include <type_traits>

struct A { int m; };
static_assert(std::is_standard_layout_v<A> == true);

class B: public A { int m; };
static_assert(std::is_standard_layout_v<B> == false);
```

C++ Standard-Layout Types

Standard-layout classes are useful for communication with code written in other programming languages. – *[class.prop]p6*

All NSDMs are also standard-layout

All NSDMs are in the same base class

Platform/C ABI

```cpp
#include <type_traits>

struct A { int m; };
static_assert(std::is_standard_layout_v<A> == true);

class B: public A { int m; };
static_assert(std::is_standard_layout_v<B> == false);

class C { int m; public: int n; };
static_assert(std::is_standard_layout_v<C> == false);
```

C++ Standard-Layout Types

Standard-layout classes are useful for communication with code written in other programming languages. – *[class.prop]p6*

All NSDMs are also standard-layout

All NSDMs are in the same base class

All NSDMs have the same access control*

Platform/C ABI

```cpp
#include <type_traits>

struct A { int m; };
static_assert(std::is_standard_layout_v<A> == true);

class B: public A { int m; };
static_assert(std::is_standard_layout_v<B> == false);

class C { int m; public: int n; };
static_assert(std::is_standard_layout_v<C> == false);
```

C++ Standard-Layout Types

Standard-layout classes are useful for communication with code written in other programming languages. – *[class.prop]p6*

All NSDMs are also standard-layout

All NSDMs are in the same base class

All NSDMs have the same access control*

**Re-ordering based on access control is removed in C++23 (P1847), but this requirement remains for standard-layout…*

Platform/C ABI

```cpp
#include <type_traits>

struct A { int m; };
static_assert(std::is_standard_layout_v<A> == true);

class B: public A { int m; };
static_assert(std::is_standard_layout_v<B> == false);

class C { int m; public: int n; };
static_assert(std::is_standard_layout_v<C> == false);

struct D { virtual void foo(); };
static_assert(std::is_standard_layout_v<D> == false);

```

C++ Standard-Layout Types

Standard-layout classes are useful for communication with code written in other programming languages. – [class.prop]p6

All NSDMs are also standard-layout

All NSDMs are in the same base class

All NSDMs have the same access control*

No virtual functions or virtual base classes

**Re-ordering based on access control is removed in C++23 (P1847), but this requirement remains for standard-layout...*

Building Bridges

Building Bridges

Can we do better than a C API?

Building Bridges

Can we do better than a C API?

Need to define APIs in terms of C

Building Bridges

Can we do better than a C API?

Need to define APIs in terms of C

Remove higher-level abstractions

Building Bridges

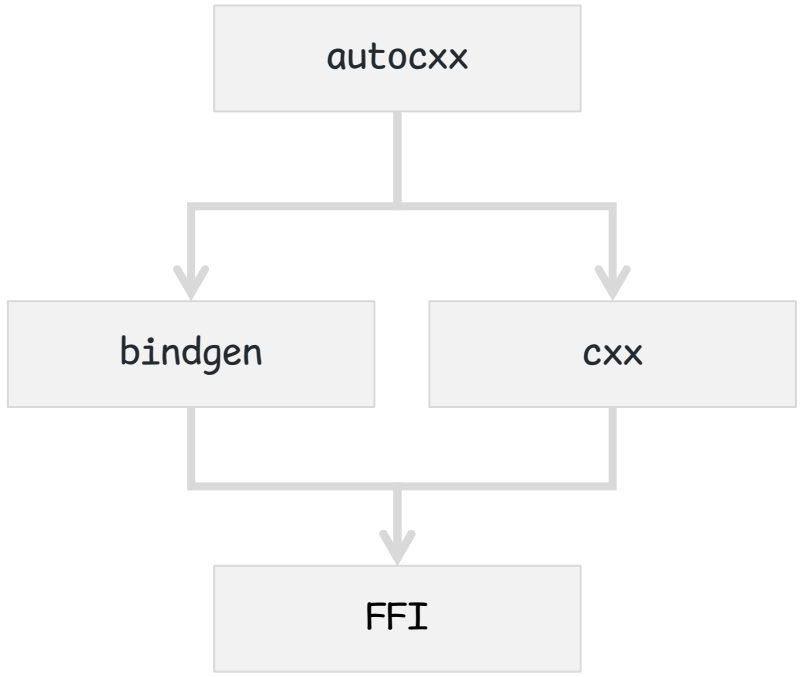
Can we do better than a C API?

Need to define APIs in terms of C

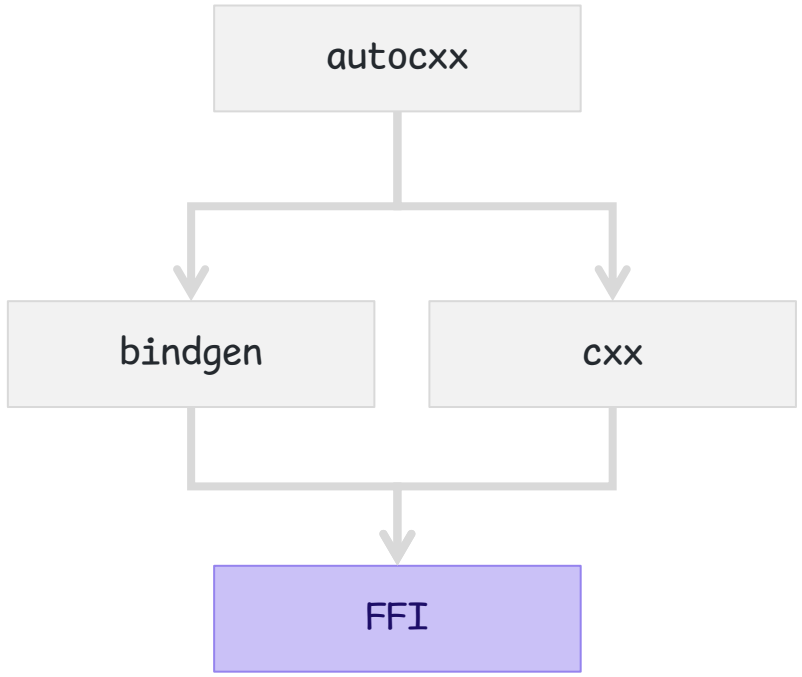
Remove higher-level abstractions

Case Study: Rust/C++

Rust/C++



Rust/C++ Ecosystem



Built-in C-like FFI

Rust/C++

```rust
use std::os::raw::*;

#[repr(C)]
struct Point {
    x: c_int,
    y: c_int,
}

unsafe extern "C" {
    fn dist(first: Point, second: Point) -> c_float;
}
```

Rust FFI

Rust/C++

```rust
use std::os::raw::*;

#[repr(C)]
struct Point {
    x: c_int,
    y: c_int,
}

unsafe extern "C" {
    fn dist(first: Point, second: Point) -> c_float;
}
```

Rust FFI

Basic C types

Rust/C++

```rust
use std::os::raw::*;

#[repr(C)]
struct Point {
    x: c_int,
    y: c_int,
}

unsafe extern "C" {
    fn dist(first: Point, second: Point) -> c_float;
}
```

Rust FFI

Basic C types

C-like structure types

Rust/C++

```
01 use std::os::raw::*;
02
03 #[repr(C)]
04 struct Point {
05     x: c_int,
06     y: c_int,
07 }
08
09 unsafe extern "C" {
10     fn dist(first: Point, second: Point) -> c_float;
11 }
12
13
14
```

Rust FFI

Basic C types

C-like structure types

C-like functions decls.

Rust/C++

```
01 int *foo() {
02     return nullptr;
03 }
```

```
04 unsafe extern "C" {
05     fn foo() -> Option<&i32>;
06 }
07
08 unsafe fn bar() {
09     if let Some(it) = foo() {
10         println!("{it}");
11     }
12 }
13
```

Rust FFI

Rust/C++

```
01 int *foo() {
02     return nullptr;
03 }
```

```
04 unsafe extern "C" {
05     fn foo() -> Option<&i32>;
06 }
07
08 unsafe fn bar() {
09     if let Some(it) = foo() {
10         println!("{it}");
11     }
12 }
13
```

Rust FFI

Nullable Pointer Optimization

Rust/C++

```cpp
int *foo() {
    return nullptr;
}
```

```rust
unsafe extern "C" {
    fn foo() -> Option<&i32>;
}

unsafe fn bar() {
    if let Some(it) = foo() {
        println!("{it}");
    }
}

```

Rust FFI

Nullable Pointer Optimization

`Option::None` has same repr. as `nullptr`

Rust/C++

```cpp
int *foo() {
    return nullptr;
}
```

```rust
unsafe extern "C" {
    fn foo() -> Option<&i32>;
}

unsafe fn bar() {
    if let Some(it) = foo() {
        println!("{it}");
    }
}

```

Rust FFI

Nullable Pointer Optimization

`Option::None` has same repr. as `nullptr`

Idiomatic handling in Rust

C++ ABI

```cpp
void foo() try {
    bar();
} catch (const std:: runtime_error& e) {
    std::println("{}", e.what());
}
```

```rust
#[no_mangle]
pub extern "C-unwind" fn bar() {
    let n = Box::new(42);
    unsafe { baz(); }
}
```

```cpp
void baz() {
    throw std::runtime_error("Hello from Rust!");
}
```

Rust FFI: Unwind ABIs

C++ ABI

```
01 void foo() try {
02     bar();
03 } catch (const std:: runtime_error& e) {
04     std::println("{}", e.what());
05 }
```

```
06 #[no_mangle]
07 pub extern "C-unwind" fn bar() {
08     let n = Box::new(42);
09     unsafe { baz(); }
10 }
```

```
11 void baz() {
12     throw std::runtime_error("Hello from Rust!");
13 }
```

Rust FFI

Rust has `"-unwind"` ABIs

C++ ABI

```cpp
void foo() try {
    bar();
} catch (const std:: runtime_error& e) {
    std::println("{}", e.what());
}
```

```rust
#[no_mangle]
pub extern "C-unwind" fn bar() {
    let n = Box::new(42);
    unsafe { baz(); }
}
```

```cpp
void baz() {
    throw std::runtime_error("Hello from Rust!");
}
```

Rust FFI

Rust has "-unwind" ABIs

C++ exceptions can passthrough Rust

C++ ABI

```
01 void foo() try {
02     bar();
03 } catch (const std:: runtime_error& e) {
04     std::println("{}", e.what());
05 }
```

```
06 #[no_mangle]
07 pub extern "C-unwind" fn bar() {
08     let n = Box::new(42);
09     unsafe { baz(); }
10 }
```

```
11 void baz() {
12     throw std::runtime_error("Hello from Rust!");
13 }
```

Rust FFI

Rust has `"-unwind"` ABIs

C++ exceptions can passthrough Rust

Correct cleanup in Rust

C++ ABI

```cpp
01 void foo() try {
02     bar();
03 } catch (const std:: runtime_error& e) {
04     std::println("{}", e.what());
05 }
```

```rust
06 #[no_mangle]
07 pub extern "C-unwind" fn bar() {
08     let n = Box::new(42);
09     unsafe { baz(); }
10 }
```

```cpp
11 void baz() {
12     throw std::runtime_error("Hello from Rust!");
13 }
```

Rust FFI

Rust has `"-unwind"` ABIs

C++ exceptions can passthrough Rust

Correct cleanup in Rust

Cannot catch foreign exceptions

C++ ABI

```cpp
void foo() try {
    bar();
} catch (const std:: runtime_error& e) {
    std::println("{}", e.what());
}
```

```rust
#[no_mangle]
pub extern "C-unwind" fn bar() {
    let n = Box::new(42);
    unsafe { baz(); }
}
```

```cpp
void baz() {
    throw std::runtime_error("Hello from Rust!");
}
```

Exception ABI

C++ ABI

```cpp
void foo() try {
    bar();
} catch (const std:: runtime_error& e) {
    std::println("{}", e.what());
}
```

```rust
#[no_mangle]
pub extern "C-unwind" fn bar() {
    let n = Box::new(42);
    unsafe { baz(); }
}
```

```cpp
void baz() {
    throw std::runtime_error("Hello from Rust!");
}
```

Exception ABI

Unwind Tables

C++ ABI

```cpp
void foo() try {
    bar();
} catch (const std:: runtime_error& e) {
    std::println("{}", e.what());
}
```

```rust
#[no_mangle]
pub extern "C-unwind" fn bar() {
    let n = Box::new(42);
    unsafe { baz(); }
}
```

```cpp
void baz() {
    throw std::runtime_error("Hello from Rust!");
}
```

Exception ABI

Unwind Tables

C++ ABI

```cpp
void foo() try {
    bar();
} catch (const std:: runtime_error& e) {
    std::println("{}", e.what());
}
```

```rust
#[no_mangle]
pub extern "C-unwind" fn bar() {
    let n = Box::new(42);
    unsafe { baz(); }
}
```

```cpp
void baz() {
    throw std::runtime_error("Hello from Rust!");
}
```

Exception ABI

Unwind Tables

Base Exception ABI

C++ ABI

```cpp
void foo() try {
    bar();
} catch (const std:: runtime_error& e) {
    std::println("{}", e.what());
}
```

```rust
#[no_mangle]
pub extern "C-unwind" fn bar() {
    let n = Box::new(42);
    unsafe { baz(); }
}
```

```cpp
void baz() {
    throw std::runtime_error("Hello from Rust!");
}
```

Exception ABI

Unwind Tables

Base Exception ABI

Unwinds the stack using unwind tables: libunwind

C++ ABI

```cpp
void foo() try {
    bar();
} catch (const std:: runtime_error& e) {
    std::println("{}", e.what());
}
```

```rust
#[no_mangle]
pub extern "C-unwind" fn bar() {
    let n = Box::new(42);
    unsafe { baz(); }
}
```

```cpp
void baz() {
    throw std::runtime_error("Hello from Rust!");
}
```

Exception ABI

Unwind Tables

Base Exception ABI

Unwinds the stack using unwind tables: libunwind

C++ ABI

```cpp
void foo() try {
    bar();
} catch (const std:: runtime_error& e) {
    std::println("{}", e.what());
}
```

```rust
#[no_mangle]
pub extern "C-unwind" fn bar() {
    let n = Box::new(42);
    unsafe { baz(); }
}
```

```cpp
void baz() {
    throw std::runtime_error("Hello from Rust!");
}
```


Exception ABI

Unwind Tables

Base Exception ABI

Unwinds the stack using unwind tables: libunwind

C++ ABI

```cpp
void foo() try {
    bar();
} catch (const std:: runtime_error& e) {
    std::println("{}", e.what());
}
```

```rust
#[no_mangle]
pub extern "C-unwind" fn bar() {
    let n = Box::new(42);
    unsafe { baz(); }
}
```

```cpp
void baz() {
    throw std::runtime_error("Hello from Rust!");
}
```


Exception ABI

Unwind Tables

Base Exception ABI

Unwinds the stack using unwind tables: libunwind

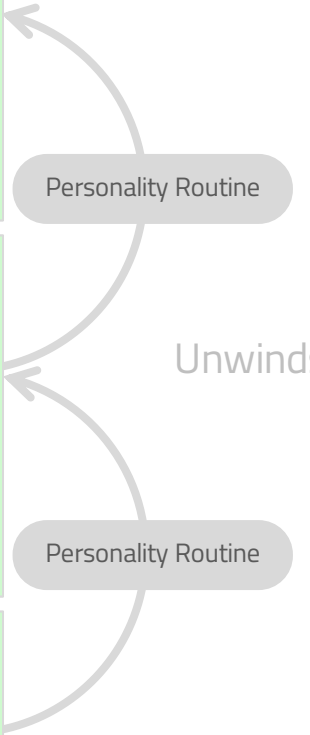
Language Exception ABI

C++ ABI

```
01 void foo() try {
02     bar();
03 } catch (const std:: runtime_error& e) {
04     std::println("{}", e.what());
05 }
```

```
06 #[no_mangle]
07 pub extern "C-unwind" fn bar() {
08     let n = Box::new(42);
09     unsafe { baz(); }
10 }
```

```
11 void baz() {
12     throw std::runtime_error("Hello from Rust!");
13 }
```



Exception ABI

Unwind Tables

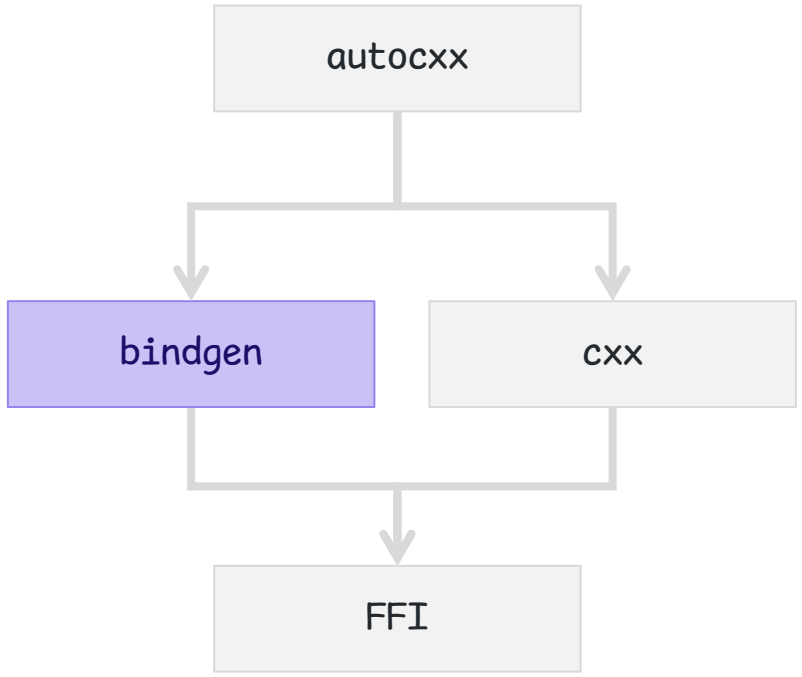
Base Exception ABI

Unwinds the stack using unwind tables: libunwind

Language Exception ABI

Personality routine

Rust/C++: bindgen



Automated binding generator

Rust/C++

```cpp
struct Point {
    int x;
    int y;
};

```

```rust

#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Point {
    pub x: c_int,
    pub y: c_int,
}

```

bindgen

Rust/C++

```
01 struct Point {
02     int x;
03     int y;
04 };
05
```

```
06
07 #[repr(C)]
08 #[derive(Debug, Copy, Clone)]
09 pub struct Point {
10     pub x: c_int,
11     pub y: c_int,
12 }
13
```

bindgen

Built with libclang

Rust/C++

```
01 struct Point {
02     int x;
03     int y;
04 };
05
```

```
06
07 #[repr(C)]
08 #[derive(Debug, Copy, Clone)]
09 pub struct Point {
10     pub x: c_int,
11     pub y: c_int,
12 }
13
```

bindgen

Built with libclang

Auto generates C/C++-Rust bindings

Rust/C++

```
01 struct Point {
02     int x;
03     int y;
04 };
05
```

```
06
07 #[repr(C)]
08 #[derive(Debug, Copy, Clone)]
09 pub struct Point {
10     pub x: c_int,
11     pub y: c_int,
12 }
13
```

bindgen

Built with libclang

Auto generates C/C++-Rust bindings

Command line tool or library for build integration

Rust/C++

```
01 struct Point {
02     int x;
03     int y;
04 };
05
```

```
06
07 #[repr(C)]
08 #[derive(Debug, Copy, Clone)]
09 pub struct Point {
10     pub x: c_int,
11     pub y: c_int,
12 }
13
```

bindgen

Built with libclang

Auto generates C/C++-Rust bindings

Command line tool or library for build integration

```
$ bindgen foo.hpp > out.rs
```

Rust/C++

```
01
02
03 const _: () = {
04     ["Size of Point"]
--         [::std::mem::size_of::<Point>() - 8usize];
05     ["Alignment of Point"]
--         [::std::mem::align_of::<Point>() - 4usize];
06     ["Offset of field: Point::x"]
--         [::std::mem::offset_of!(Point, x) - 0usize];
07     ["Offset of field: Point::y"]
--          [::std::mem::offset_of!(Point, y) - 4usize];
08 };
09
10
```

bindgen

Rust/C++

```
01
02
03 const _: () = {
04     ["Size of Point"]
--         [::std::mem::size_of::<Point>() - 8usize];
05     ["Alignment of Point"]
--         [::std::mem::align_of::<Point>() - 4usize];
06     ["Offset of field: Point::x"]
--         [::std::mem::offset_of!(Point, x) - 0usize];
07     ["Offset of field: Point::y"]
--          [::std::mem::offset_of!(Point, y) - 4usize];
08 };
09
10
```

bindgen

Static assertions for size and offset

Rust/C++

```c
int foo(char x);
int foo(char x, int y);
```

```rust


unsafe extern "C" {
    #[link_name = "_Z3fooc"]
    pub fn foo(x: c_char) -> c_int;

    #[link_name = "_Z3fooci"]
    pub fn foo1(x: c_char, y: c_int) -> c_int;
}


```

bindgen: Function Overloading

Rust/C++

```
01 int foo(char x);
02 int foo(char x, int y);
```

```
03
04
05 unsafe extern "C" {
06     #[link_name = "_Z3fooc"]
07     pub fn foo(x: c_char) -> c_int;
08
09     #[link_name = "_Z3fooci"]
10     pub fn foo1(x: c_char, y: c_int) -> c_int;
11 }
12
13
```

bindgen: Function Overloading

Auto-generates mangled names

Rust/C++

```
01 int foo(char x);
02 int foo(char x, int y);
```

```
03
04
05 unsafe extern "C" {
06     #[link_name = "_Z3fooc"]
07     pub fn foo(x: c_char) -> c_int;
08
09     #[link_name = "_Z3fooci"]
10     pub fn foo1(x: c_char, y: c_int) -> c_int;
11 }
12
13
```

bindgen: Function Overloading

Note `foo1`: No overloading in Rust

Rust/C++

```cpp
template<typename T>
struct S {
    T value;
};
void foo(S<int> s);
```

```rust
#[repr(C)]
pub struct S<T> {
    pub value: T,
}
unsafe extern "C" {
   #[link_name = "_Z3foo1SIiE"]
    pub fn foo(s: S<c_int>);
}
```

bindgen: Templates

Rust/C++

```
01 template<typename T>
02 struct S {
03     T value;
04 };
05 void foo(S<int> s);
```

```
06 #[repr(C)]
07 pub struct S<T> {
08     pub value: T,
09 }
10 unsafe extern "C" {
11    #[link_name = "_Z3foo1SIiE"]
12     pub fn foo(s: S<c_int>);
13 }
```

bindgen: Templates

Simple Templates use Rust Generics

Rust/C++

```
01 template<typename T>
02 struct S {
03     T value;
04 };
05 void foo(S<int> s);
```

```
06 #[repr(C)]
07 pub struct S<T> {
08     pub value: T,
09 }
10 unsafe extern "C" {
11     #[link_name = "_Z3foo1SIiE"]
12     pub fn foo(s: S<c_int>);
13 }
```

bindgen: Templates

Simple Templates use Rust Generics

Correct Usage as Parameter

Rust/C++

```
01 template<typename T>
02 struct S {
03     T value;
04 };
05 void foo(S<int> s);
```

```
06 #[repr(C)]
07 pub struct S<T> {
08     pub value: T,
09 }
10 unsafe extern "C" {
11    #[link_name = "_Z3foo1SIiE"]
12     pub fn foo(s: S<c_int>);
13 }
```

bindgen: Templates

Simple Templates use Rust Generics

Rust/C++

```
01 template<typename T>
02 struct S {
03     T value;
04 };
05 void foo(S<int> s);
```

```
06 #[repr(C)]
07 pub struct S<T> {
08     pub value: T,
09 }
10 unsafe extern "C" {
11    #[link_name = "_Z3foo1SIiE"]
12     pub fn foo(s: S<c_int>);
13 }
```

bindgen: Templates

Simple Templates use Rust Generics

No method generation

Rust/C++

```
01 template<typename T>
02 struct S {
03     T value;
04 };
05 void foo(S<int> s);
```

```
06 #[repr(C)]
07 pub struct S<T> {
08     pub value: T,
09 }
10 unsafe extern "C" {
11    #[link_name = "_Z3foo1SIiE"]
12     pub fn foo(s: S<c_int>);
13 }
```

bindgen: Templates

Simple Templates use Rust Generics

No method generation

No static assertions generated

C++ ABI

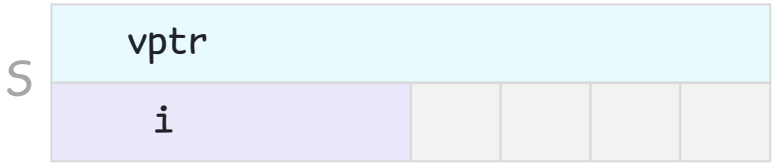
Polymorphic Types

```
01 struct S {
02     int i;
03     virtual void foo();
04 };
05
06 static_assert(sizeof(S) == 16);
```

C++ ABI

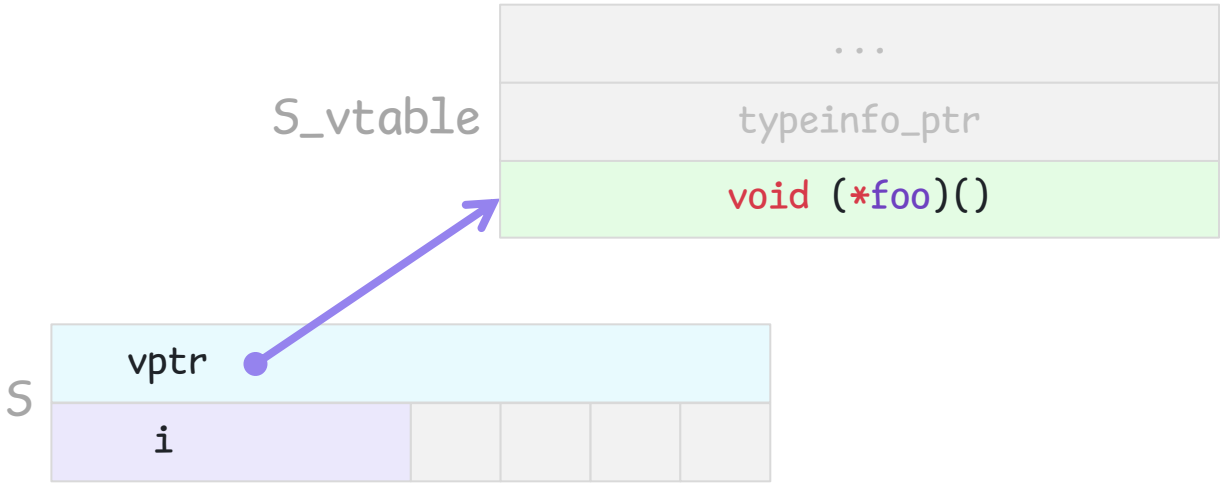
Polymorphic Types

Vptr inserted into layout




```
01 struct S {
02     int i;
03     virtual void foo();
04 };
05
06 static_assert(sizeof(S) == 16);
```

C++ ABI



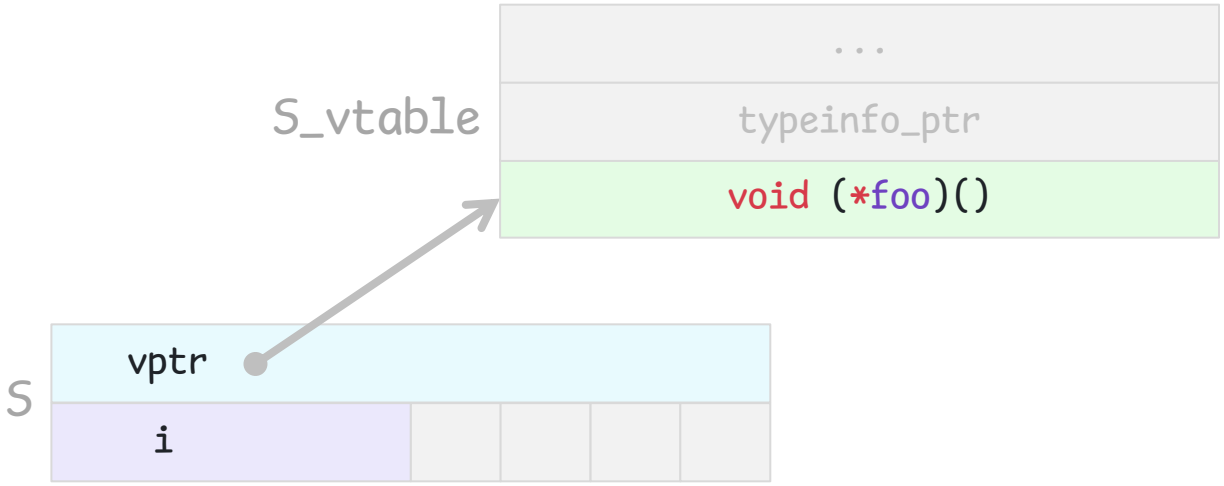
Polymorphic Types

vptr inserted into layout

vtable contains method overrides, RTTI, & more

```
01 struct S {
02     int i;
03     virtual void foo();
04 };
05
06 static_assert(sizeof(S) == 16);
```

C++ ABI




```
01 struct S {
02     int i;
03     virtual void foo();
04 };
05
06 static_assert(sizeof(S) == 16);
```

Polymorphic Types

vptr inserted into layout

vtable contains method overrides, RTTI, & more

vtable layout & behavior is complicated

Rust/C++

```
01
02
03 struct Base {
04     int i;
05     virtual void foo();
06 };
07
08 struct Derived : Base {
09     int j;
10     void foo() override;
11 };
12
13
14
```

bindgen: Polymorphic Types

Rust/C++

```
01 #[repr(C)]
02 pub struct Base__bindgen_vtable(c_void);
03
04 #[repr(C)]
05 pub struct Base {
06     pub vtable_: *const Base__bindgen_vtable,
07     pub i: c_int,
08 }
09
10 #[repr(C)]
11 pub struct Derived {
12     pub _base: Base,
13     pub j: c_int,
14 }
```

bindgen: Polymorphic Types

Rust/C++

```rust
#[repr(C)]
pub struct Base__bindgen_vtable(c_void);

#[repr(C)]
pub struct Base {
    pub vtable_: *const Base__bindgen_vtable,
    pub i: c_int,
}

#[repr(C)]
pub struct Derived {
    pub _base: Base,
    pub j: c_int,
}
```

bindgen: Polymorphic Types

Base has a vptr

Rust/C++

```
01
02
03
04 unsafe extern "C" {
05     #[link_name = "_ZN4Base3fooEv"]
06     pub fn Base_foo(this: *mut c_void);
07
08     #[link_name = "_ZN7Derived3fooEv"]
09     pub fn Derived_foo(this: *mut c_void);
10 }
11
12
13
14
```

bindgen: Polymorphic Types

Rust/C++

```
01
02
03
04 unsafe extern "C" {
05     #[link_name = "_ZN4Base3fooEv"]
06     pub fn Base_foo(this: *mut c_void);
07
08     #[link_name = "_ZN7Derived3fooEv"]
09     pub fn Derived_foo(this: *mut c_void);
10 }
11
12
13
14
```

bindgen: Polymorphic Types

Pure FFI functions; no methods

```
01
02
03
04 unsafe extern "C" {
05     #[link_name = "_ZN4Base3fooEv"]
06     pub fn Base_foo(this: *mut c_void);
07
08     #[link_name = "_ZN7Derived3fooEv"]
09     pub fn Derived_foo(this: *mut c_void);
10 }
11
12
13
14
```

bindgen: Polymorphic Types

Raw void pointers

Rust/C++

```
01
02
03
04
05 fn main() {
06     let mut d = get_derived();
07     unsafe {
08         Derived_foo(&mut d as *mut _ as *mut c_void);
09     }
10 }
11
12
13
14
```

bindgen: Polymorphic Types

Rust/C++

```
01
02
03
04
05 fn main() {
06     let mut d = get_derived();
07     unsafe {
08         Derived_foo(&mut d as *mut _ as *mut c_void);
09     }
10 }
11
12
13
14
```

bindgen: Polymorphic Types

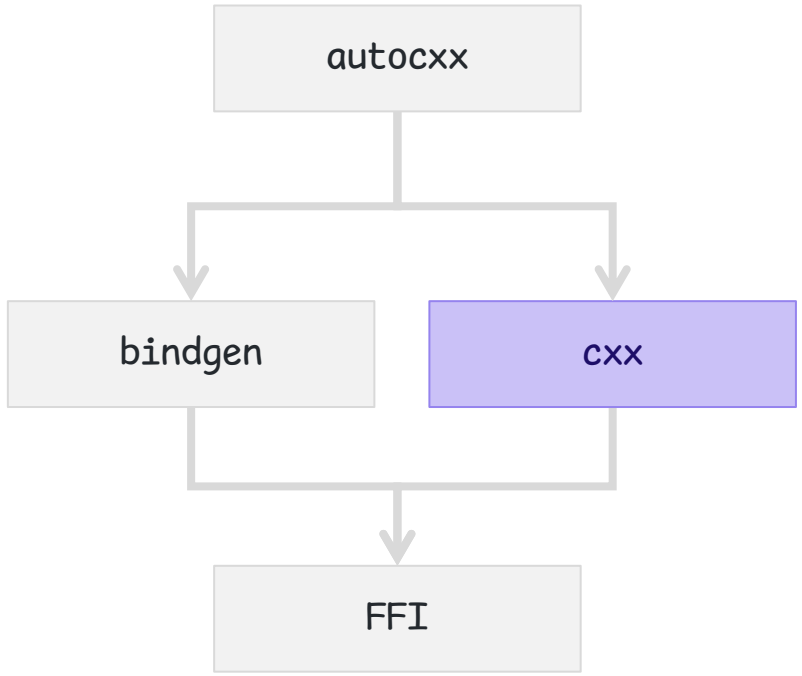
Unsafe… and a bit messy 🙈

```
01
02
03
04
05 fn main() {
06     let mut d = get_derived();
07     unsafe {
08         Derived_foo(&mut d as *mut _ as *mut c_void);
09     }
10 }
11
12
13
14
```

bindgen: Polymorphic Types

bindgen only tries to get the layout correct

Rust/C++: cxx



Automated bindings + bridged types

Rust/C++

cxx

cxx

Bindings for types like `std::string` and `std::vector`

cxx

Bindings for types like `std::string` and `std::vector`

Special consideration given to ownership

Rust/C++

cxx

Bindings for types like `std::string` and `std::vector`

Special consideration given to ownership

Patterns for error handling

Rust/C++

```
01
02
03
04 #[cxx::bridge]
05 mod ffi {
06     unsafe extern "C++" {
07         include!("foo.hpp");
08         type Foo;
09         fn new_foo() -> UniquePtr<Foo>;
10     }
11 }
12
13
14
```

cxx

Rust/C++

```
01
02
03
04 #[cxx::bridge]
05 mod ffi {
06     unsafe extern "C++" {
07         include!("foo.hpp");
08         type Foo;
09         fn new_foo() -> UniquePtr<Foo>;
10     }
11 }
12
13
14
```

cxx

Include C/C++ files

Rust/C++

```
01
02
03
04 #[cxx::bridge]
05 mod ffi {
06     unsafe extern "C++" {
07         include!("foo.hpp");
08         type Foo;
09         fn new_foo() -> UniquePtr<Foo>;
10     }
11 }
12
13
14
```

cxx

Declare types & functions

Rust/C++

```rust

fn foo(s: String) {}

fn main() {
    let s = "Hello, Meeting C++".to_string();
    foo(s); // Move
    foo(s); // Error: use of moved value: `s`
}


```

Move Semantics

Rust/C++

```
01
02
03
04 fn foo(s: String) {}
05
06 fn main() {
07     let s = "Hello, Meeting C++".to_string();
08     foo(s); // Move
09     foo(s); // Error: use of moved value: `s`
10 }
11
12
13
14
```

Move Semantics

Rust has destructive moves

Rust/C++

```rust

fn foo(s: String) {}

fn main() {
    let s = "Hello, Meeting C++".to_string();
    foo(s); // Move
    foo(s); // Error: use of moved value: `s`
}


```

Move Semantics

No destructor call for **s**

Rust/C++

```
01
02
03
04 void foo(std::string s) {}
05
06 int main() {
07     auto s = std::string("Hello, Meeting C++");
08     foo(std::move(s)); // Move
09     foo(s);            // Ok
10 } // `s.~S()`
11
12
13
14
```

Move Semantics

Rust/C++

```
01
02
03
04 void foo(std::string s) {}
05
06 int main() {
07     auto s = std::string("Hello, Meeting C++");
08     foo(std::move(s)); // Move
09     foo(s);            // Ok
10 } // `s.~S()`
11
12
13
14
```

Move Semantics

C++ leaves s in a valid state

Rust/C++

```
01
02
03
04 void foo(std::string s) {}
05
06 int main() {
07     auto s = std::string("Hello, Meeting C++");
08     foo(std::move(s)); // Move
09     foo(s);            // Ok
10 } // `s.~S()`
11
12
13
14
```

Move Semantics

Destructor is still run

cxx: Types

cxx: Types

Shared

cxx: Types

Shared **Opaque**

cxx: Types

Shared **Opaque** **Extern**

cxx: Types

Shared	Opaque	Extern
Visible to both languages		
Passed by value		
Trivial types		

Rust/C++

cxx: Types

Shared	Opaque	Extern
Visible to both languages	Visible to one language	
Passed by value	Handled behind indirection	
Trivial types	Non-trivial types	

cxx: Types

Shared	Opaque	Extern
Visible to both languages	Visible to one language	Defined outside of cxx
Passed by value	Handled behind indirection	Opaque or Trivial
Trivial types	Non-trivial types	

cxx: Types

Shared	Opaque	Extern
Visible to both languages	Visible to one language	Defined outside of cxx
Passed by value	**Handled behind indirection**	Opaque or Trivial
Trivial types	Non-trivial types	

Rust/C++

```
01 struct S {
02     ~S();
03 };
04 std::unique_ptr<S> make();
```

```
05 #[cxx::bridge]
06 mod ffi {
07     unsafe extern "C++" {
08         include!("foo.hpp");
09         type S;
10         fn make() -> UniquePtr<S>;
11     }
12 }
13
```

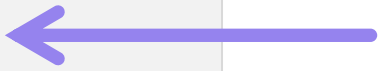
cxx: Opaque C++ Types

Rust/C++

```
01 struct S {
02     ~S();
03 };
04 std::unique_ptr<S> make();
```

```
05 #[cxx::bridge]
06 mod ffi {
07     unsafe extern "C++" {
08         include!("foo.hpp");
09         type S;
10         fn make() -> UniquePtr<S>;
11     }
12 }
13
```

cxx: Opaque C++ Types



Handled behind an indirection

Rust/C++

```
01 struct S {
02     ~S();
03     int foo() const;
04 };
```

```
05 #[cxx::bridge]
06 mod ffi {
07     unsafe extern "C++" {
08         include!("foo.hpp");
09         type S;
10         fn foo(self: &S);
11     }
12 }
13
```

cxx: Opaque C++ Types

Declare methods as expected, but…

Rust/C++

```
01 struct S {
02     ~S();
03     void bar();
04 };
```

```
05 #[cxx::bridge]
06 mod ffi {
07     unsafe extern "C++" {
08         include!("foo.hpp");
09         type S;
10         fn bar(self: Pin<&mut S>);
11     }
12 }
13
```

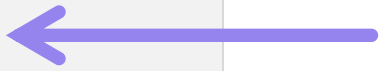
cxx: Opaque C++ Types

Rust/C++

```
01 struct S {
02     ~S();
03     void bar();
04 };
```

```
05 #[cxx::bridge]
06 mod ffi {
07     unsafe extern "C++" {
08         include!("foo.hpp");
09         type S;
10         fn bar(self: Pin<&mut S>);
11     }
12 }
13
```

cxx: Opaque C++ Types



Non-const methods require `Pin`

Rust/C++

```
01 struct S {
02     ~S();
03     void bar();
04 };
```

```
05 #[cxx::bridge]
06 mod ffi {
07     unsafe extern "C++" {
08         include!("foo.hpp");
09         type S;
10         fn bar(self: Pin<&mut S>);
11     }
12 }
13
```

cxx: Opaque C++ Types

`&mut T` can be used to swap memory

Rust/C++

```
01 struct S {
02     ~S();
03     void bar();
04 };
```

```
05 #[cxx::bridge]
06 mod ffi {
07     unsafe extern "C++" {
08         include!("foo.hpp");
09         type S;
10         fn bar(self: Pin<&mut S>);
11     }
12 }
13
```

cxx: Opaque C++ Types

`&mut T` can be used to swap memory

```
01 use std::mem::{swap, take, replace};
```

```
01 pub const fn swap<T>(x: &mut T, y: &mut T);
```

cxx: Types

Shared	Opaque	Extern
Visible to both languages	Visible to one language	Defined outside of cxx
Passed by value	Handled behind indirection	**Opaque or Trivial**
Trivial types	Non-trivial types	

Rust/C++

```
struct S {
    ~S();
};
```

```
struct S(i32);
unsafe impl ExternType for S {
    type Id = cxx::type_id!("S");
    type Kind = cxx::kind::Trivial;
}
#[cxx::bridge]
mod ffi {
    unsafe extern "C++" {
        include!("foo.hpp");
        type S = super::S;
    }
}
```

cxx: Trivial Extern C++ Types

Rust/C++

```
01 struct S {
02     ~S();
03 };
```

```
04 struct S(i32);
05 unsafe impl ExternType for S {
06     type Id = cxx::type_id!("S");
07     type Kind = cxx::kind::Trivial;
08 }
09 #[cxx::bridge]
10 mod ffi {
11     unsafe extern "C++" {
12         include!("foo.hpp");
13         type S = super::S;
14     }
15 }
```

cxx: Trivial Extern C++ Types

Existing (bindgen or hand-written) bindings

Rust/C++

```
01 struct S {
02     ~S();
03 };
```

```
04 struct S(i32);
05 unsafe impl ExternType for S {
06     type Id = cxx::type_id!("S");
07     type Kind = cxx::kind::Trivial;
08 }
09 #[cxx::bridge]
10 mod ffi {
11     unsafe extern "C++" {
12         include!("foo.hpp");
13         type S = super::S;
14     }
15 }
```

cxx: Trivial Extern C++ Types

Existing (bindgen or hand-written) bindings

`Trivial` = subject to Rust move semantics*

Rust/C++

```
01 struct S {
02     ~S();
03 };
```

```
04 struct S(i32);
05 unsafe impl ExternType for S {
06     type Id = cxx::type_id!("S");
07     type Kind = cxx::kind::Trivial;
08 }
09 #[cxx::bridge]
10 mod ffi {
11     unsafe extern "C++" {
12         include!("foo.hpp");
13         type S = super::S;
14     }
15 }
```

cxx: Trivial Extern C++ Types

Existing (bindgen or hand-written) bindings

`Trivial` = subject to Rust move semantics*

*C++26 introduces `std::is_trivially_relocatable` (P2786)

Rust/C++

```rust
fn foo() -> Result<(), String> {
    Err("Rust Error".to_string())
}

#[cxx::bridge]
mod ffi {
    extern "Rust" {
        fn foo() -> Result<()>;
    }
    unsafe extern "C++" {
        fn bar() -> Result<()>;
    }
}

```

cxx: Error Handling

Rust/C++

```rust
fn foo() -> Result<(), String> {
    Err("Rust Error".to_string())
}

#[cxx::bridge]
mod ffi {
    extern "Rust" {
        fn foo() -> Result<()>;
    }
    unsafe extern "C++" {
        fn bar() -> Result<()>;
    }
}

```

cxx: Error Handling

C++ exceptions to Rust `Result<T, E>`

Rust/C++

```rust
fn foo() -> Result<(), String> {
    Err("Rust Error".to_string())
}

#[cxx::bridge]
mod ffi {
    extern "Rust" {
        fn foo() -> Result<()>;
    }
    unsafe extern "C++" {
        fn bar() -> Result<()>;
    }
}

```

cxx: Error Handling

C++ exceptions to Rust `Result<T, E>`

Converted at the boundary

Rust/C++

```
01 namespace rust {
02 namespace behavior {
03
04 template <typename Try, typename Fail>
05 static void trycatch(Try &&func, Fail &&fail) noexcept
06 try {
07     func();
08 } catch (const std::exception &e) {
09     fail(e.what());
10 }
11
12 } // namespace behavior
13 } // namespace rust
14
```

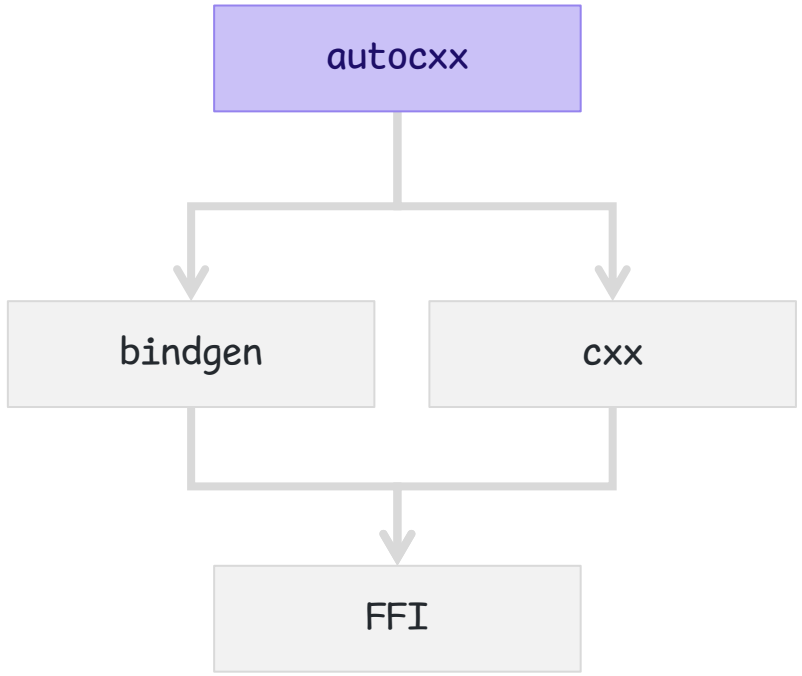
cxx: Error Handling

C++ exceptions to Rust `Result<T, E>`

Converted at the boundary

Custom conversion function

Rust/C++: autocxx



Glues bindgen and cxx together

Rust/C++

```rust
include_cpp! {
    #include "foo.hpp"
    safety!(unsafe)
    generate!("Foo")
    generate!("Bar")
}
fn main() {
    let foo = ffi::Foo::new(10.into());
    println!("{}", foo.val);

    let mut bar = ffi::Bar::new(20.into())
        .within_unique_ptr();
    println!("{}", bar.pin_mut().val());
}
```

autocxx

Rust/C++

```
01 include_cpp! {
02     #include "foo.hpp"
03     safety!(unsafe)
04     generate!("Foo")
05     generate!("Bar")
06 }
07 fn main() {
08     let foo = ffi::Foo::new(10.into());
09     println!("{}", foo.val);
10
11     let mut bar = ffi::Bar::new(20.into())
12         .within_unique_ptr();
13     println!("{}", bar.pin_mut().val());
14 }
```

autocxx

Glues bindgen and cxx together

Rust/C++

```
01 include_cpp! {
02     #include "foo.hpp"
03     safety!(unsafe)
04     generate!("Foo")
05     generate!("Bar")
06 }
07 fn main() {
08     let foo = ffi::Foo::new(10.into());
09     println!("{}", foo.val);
10
11     let mut bar = ffi::Bar::new(20.into())
12         .within_unique_ptr();
13     println!("{}", bar.pin_mut().val());
14 }
```

autocxx

Glues bindgen and cxx together

Provides a fluent bridge

Rust/C++

```
01 include_cpp! {
02     #include "foo.hpp"
03     safety!(unsafe)
04     generate!("Foo")
05     generate!("Bar")
06 }
07 fn main() {
08     let foo = ffi::Foo::new(10.into());
09     println!("{}", foo.val);
10
11     let mut bar = ffi::Bar::new(20.into())
12         .within_unique_ptr();
13     println!("{}", bar.pin_mut().val());
14 }
```

autocxx

Glues bindgen and cxx together

Provides a fluent bridge

Trivial types are simple values

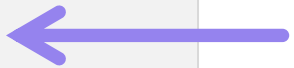
Rust/C++

```
01 include_cpp! {
02     #include "foo.hpp"
03     safety!(unsafe)
04     generate!("Foo")
05     generate!("Bar")
06 }
07 fn main() {
08     let foo = ffi::Foo::new(10.into());
09     println!("{}", foo.val);
10
11     let mut bar = ffi::Bar::new(20.into())
12         .within_unique_ptr();
13     println!("{}", bar.pin_mut().val());
14 }
```

autocxx

Glues bindgen and cxx together

Provides a fluent bridge



Non-trivial types behind an indirection + `Pin`

Building Bridges

Building Bridges

Summary

Use C as a foundation

Building Bridges

Summary

Use C as a foundation

Static assert type layout

Building Bridges

Summary

Use C as a foundation

Static assert type layout

Use code generation to overcome differences

Building Bridges

Summary

Use C as a foundation

Static assert type layout

Use code generation to overcome differences

Handle exceptions at the boundary

Qt Quality Assurance

Danke schön!
Fragen?