C++ now

2025

Zngur

Simplified Rust/C++ Integration

David Sankel

Zngur

Simplified Rust/C++ Integration

David Sankel | Principal Scientist
Software Technology Lab

Adobe

Image by Shruti Singh for Adobe Creative Residency

Adobe has (and will have) open positions!

Use the QR code to learn more and sign up for our newsletter!

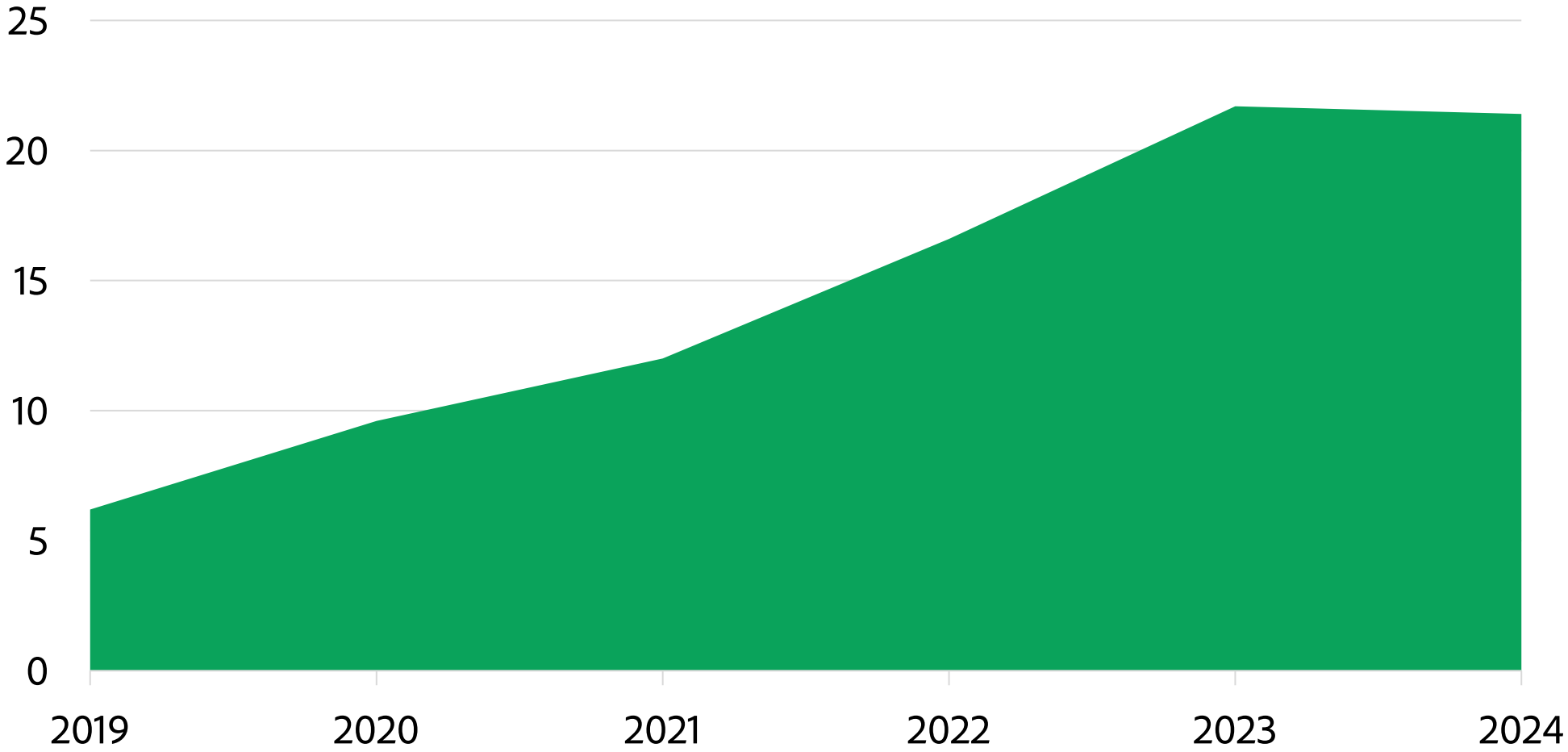
Why is Rust/C++ interop interesting?

On balance, Rust is a better tech for most applications

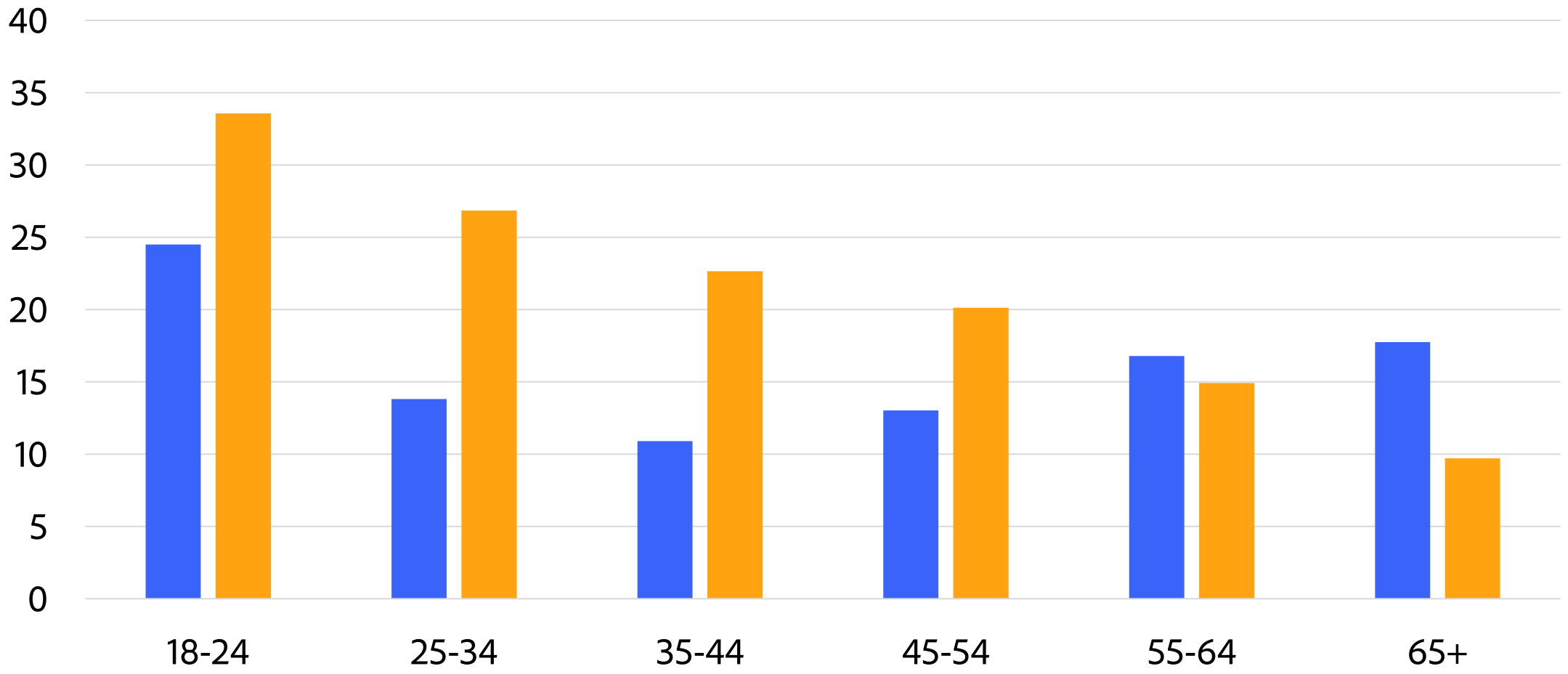
- Powerful language mechanisms
  - enums (language variants)
  - pattern matching
  - traits (~checked concepts)
- Excellent procedural-macro driven libraries
- Engineering features (documentation, side-by-side tests, etc.)
- Excellent tooling
- Ergonomics

Trends in programming languages are clear

Percentage of C++ developers also using Rust



Age vs. Percentage of Developers Wanting to use C++ vs. Rust



Evolving expectations for software safety

- Ensuring customer security against malicious exploitation is paramount.
- Adoption of memory-safe languages has the potential to reduce vulnerabilities by ~70% for new code
- Memory safety legislation is on the horizon

C++ and memory safety

- Great improvements coming…
  - Hardened standard library
  - Profiles will do something, maybe, someday to help mitigate risk
- However…
  - “Safe C++” is not moving forward
  - Safety theater abounds
  - There’s little hope

Adobe

Great! Go use Rust and leave us alone.

But all that C++ code out there!

- There's a lot
  - Photoshop alone has >30 million lines of C++
  - Brian Cantwell Smith estimated 100 billion lines of C++ in 1997
  - Sage McEnry estimated 2.8 trillion lines of all code in 2020
- That's more than the number of stars in the Milky Way

We're using that code, and we're not going to rewrite most of it.

Rust/C++ interop

What do we want in Rust/C++ interop?

- Principled design
- Good ergonomics on both sides
- Automate as much as we can
- Complete
- Practical

Adobe

Limitations of current solutions

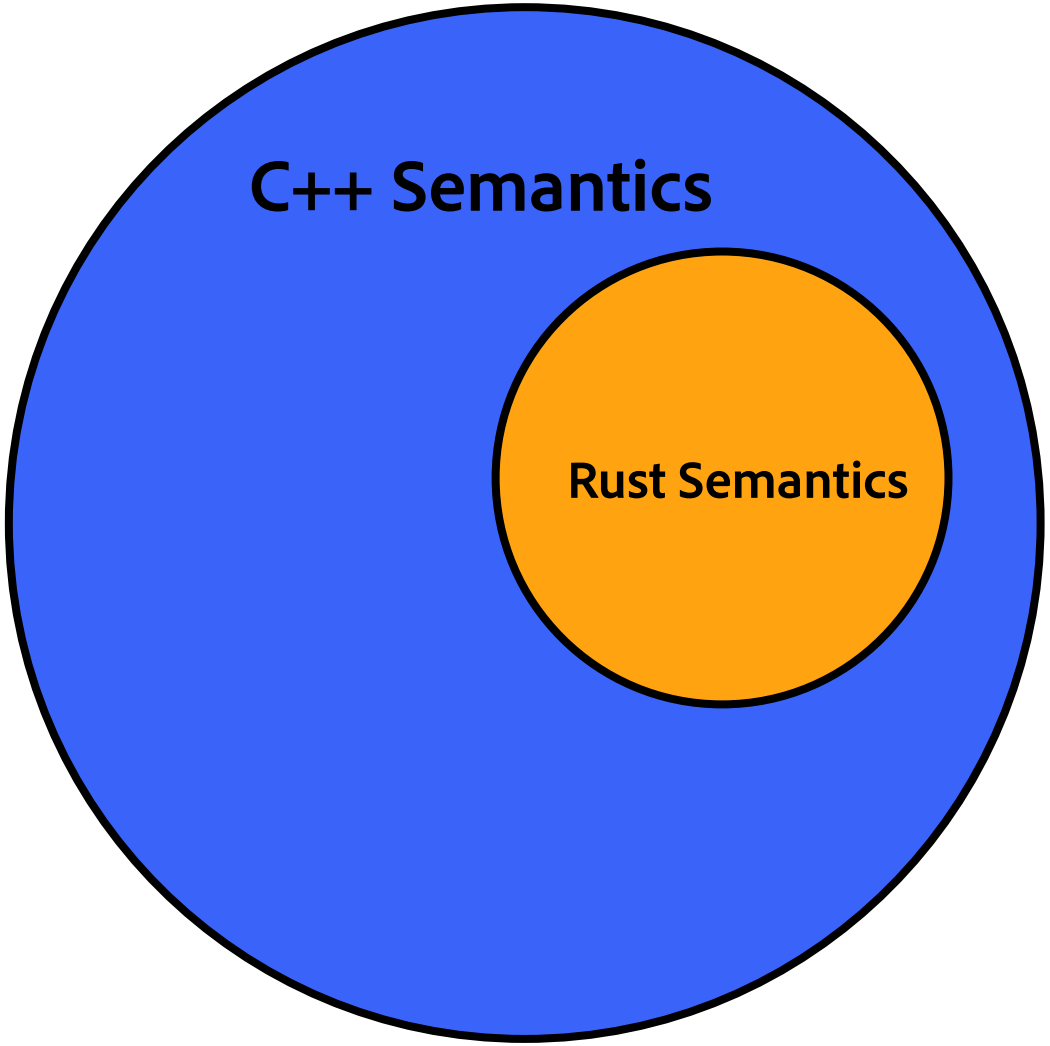
- capigen—write C APIs in Rust and generate headers
  - Requires C++ scaffolding to make decent C++ interfaces
  - Lots of complicated, unsafe code on the Rust side
  - Dealing with callbacks, allocators, etc. is painful
- cxx—write bridge code in Rust and make smart C++/Rust translations
  - "intentionally restrictive and opinionated"
  - Lacks core functionality, like function pointers
- autocxx—automate most bridging code
  - Performance (UniquePtr everywhere) and ergonomics (Pin everywhere) issues

Zngur

- /zængɑr/
- Created by Hamidreza Kalbasi
- https://github.com/HKalbasi/zngur

"[T]ries to expose arbitrary Rust types, methods and functions, while preserving its semantics and ergonomics as much as possible. Using Zngur, you can use arbitrary Rust crates in your C++ code as easily as using it in normal Rust code, and you can write idiomatic Rusty APIs for your C++ library inside C++."

Zngur's driving principle: Rust is a subset of C++ Semantics



Adobe

Example | Result errors

Rust

```
enum Result<T, E> {
  Ok(T),
  Err(E),
}
```

≈

C++

```
template<typename T>
struct Ok{ T t; };

template<typename E>
struct Err{ E e; };

template<typename T, typename E>
using Result = std::variant<Ok<T>, Err<E>>;
```

Example | exceptions

Rust

No equivalent

C++

```
throw std::runtime_error("Some error");
```

Example | memcopy relocation

Rust

```
let mut x = SomeType::new();
mem::replace(
    &mut x,
    SomeType::new());
```

C++

```
// C++23: SomeType must be “trivially copyable”
// C++26: SomeType must be “trivially relocatable”

auto x = SomeType();
alignas(SomeType) char buffer[sizeof(SomeType)];
SomeType* x_new = new (buffer) SomeType();
std::memcpy(&x, x_new, sizeof(SomeType));
```

Example | memcopy relocation

Rust

```
let mut x = SomeType::new();
mem::replace(
    &mut x,
    SomeType::new());
```

C++

```
// C++23: SomeType must be “trivially copyable”
// C++26: SomeType must be “trivially relocatable”

auto x = SomeType();
alignas(SomeType) char buffer[sizeof(SomeType)];
SomeType* x_new = new (buffer) SomeType();
std::memcpy(&x, x_new, sizeof(SomeType));
```

Example | memcopy relocation

Rust

```
let mut x = SomeType::new();
mem::replace(
    &mut x,
    SomeType::new());
```

C++

```
// C++23: SomeType must be “trivially copyable”
// C++26: SomeType must be “trivially relocatable”

auto x = SomeType();
alignas(SomeType) char buffer[sizeof(SomeType)];
SomeType* x_new = new (buffer) SomeType();
std::memcpy(&x, x_new, sizeof(SomeType));
```

Example | memcopy relocation

Rust

```
let mut x = SomeType::new();
mem::replace(
    &mut x,
    SomeType::new());
```

C++

```
// C++23: SomeType must be “trivially copyable”
// C++26: SomeType must be “trivially relocatable”

auto x = SomeType();
alignas(SomeType) char buffer[sizeof(SomeType)];
SomeType* x_new = new (buffer) SomeType();
std::memcpy(&x, x_new, sizeof(SomeType));
```

Example | memcopy relocation

Rust

```
let mut x = SomeType::new();
mem::replace(
    &mut x,
    SomeType::new());
```

C++

```
// C++23: SomeType must be “trivially copyable”
// C++26: SomeType must be “trivially relocatable”

auto x = SomeType();
alignas(SomeType) char buffer[sizeof(SomeType)];
SomeType* x_new = new (buffer) SomeType();
std::memcpy(&x, x_new, sizeof(SomeType));
```

Adobe

Example | General moves

C++

```
class SomeType {
  public:
    SomeType(SomeType&&);
    SomeType& operator=(SomeType&&);
  private:
    std::array<int, 64> buffer;
    int * selection; // points to buffer element
};
```

Rust

No equivalent

Example | General moves

C++

```
class SomeType {
  public:
    SomeType(SomeType&&);
    SomeType& operator=(SomeType&&);
  private:
    std::array<int, 64> buffer;
    int * selection; // points to buffer element
};
```

Rust

No equivalent

Example | General moves

C++

```
class SomeType {
  public:
    SomeType(SomeType&&);
    SomeType& operator=(SomeType&&);
  private:
    std::array<int, 64> buffer;
    int * selection; // points to buffer element
};
```

Rust

No equivalent

Moves in Rust

- Moves in Rust, unlike C++, are always memcpy
- Rust uses moves in assignment, passing arguments by value, and returning values from functions
  - Note that actual moves may be optimized out

Important consequence: C++ objects cannot be put on a Rust stack!

Example | law of exclusivity

C++

```
void f(int& x, int& y);

int main() {
  int i = 3;
  f(i, i);
}
```

Rust

```
fn f(x: &mut i32, y: &mut i32) { /*... */ }

fn main() {
    let mut a = 3;
    f(&mut a, &mut a); // error: law of exclusivity
}
```

Example | law of exclusivity

C++

```
void f(int& x, int& y);

int main() {
  int i = 3;
  f(i, i);
}
```

Rust

```
fn f(x: &mut i32, y: &mut i32) { /*... */ }

fn main() {
    let mut a = 3;
    f(&mut a, &mut a); // error: law of exclusivity
}
```

Example | law of exclusivity

C++

```
void f(int& x, int& y);

int main() {
  int i = 3;
  f(i, i);
}
```

Rust

```
fn f(x: &mut i32, y: &mut i32) { /*... */ }

fn main() {
    let mut a = 3;
    f(&mut a, &mut a); // error: law of exclusivity
}
```

Example | law of exclusivity

C++

```
void f(int& x, int& y);

int main() {
  int i = 3;
  f(i, i);
}
```

Rust

```
fn f(x: &mut i32, y: &mut i32) { /*... */ }

fn main() {
    let mut a = 3;
    f(&mut a, &mut a); // error: law of exclusivity
}
```

Example | law of exclusivity

C++

```
void f(int& x, int& y);

int main() {
  int i = 3;
  f(i, i);
}
```

Rust

```
fn f(x: &mut i32, y: &mut i32) { /*... */ }

fn main() {
    let mut a = 3;
    f(&mut a, &mut a); // error: law of exclusivity
}
```

Adobe

Example | law of exclusivity

C++

```
void f(int& x, int& y);

int main() {
  int i = 3;
  f(i, i);
}
```

Rust

```
fn f(x: &mut i32, y: &mut i32) { /*... */ }

fn main() {
    let mut a = 3;
    f(&mut a, &mut a); // error: law of exclusivity
}
```

Example | law of exclusivity

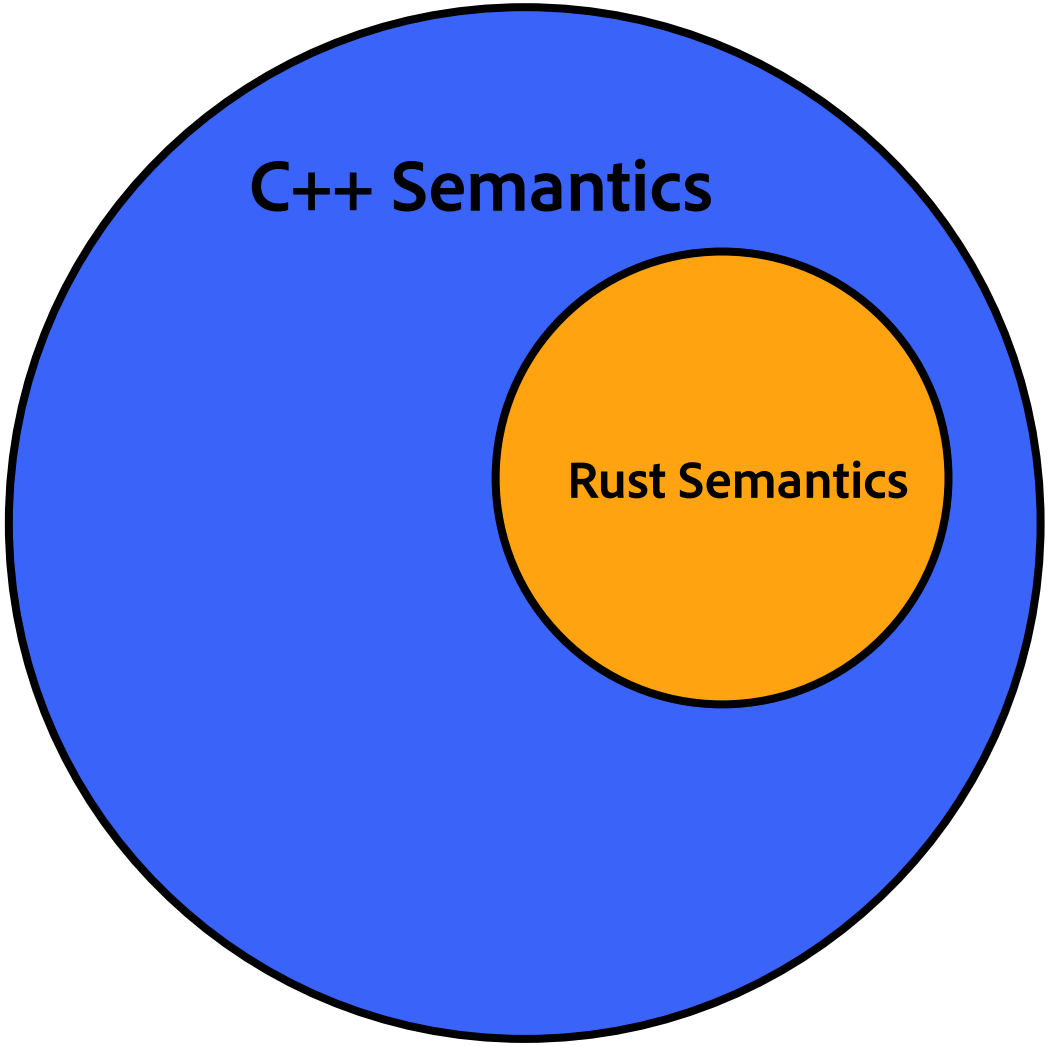
C++

```
void f(int& x, int& y);

int main() {
  int i = 3;
  f(i, i);
}
```

Rust

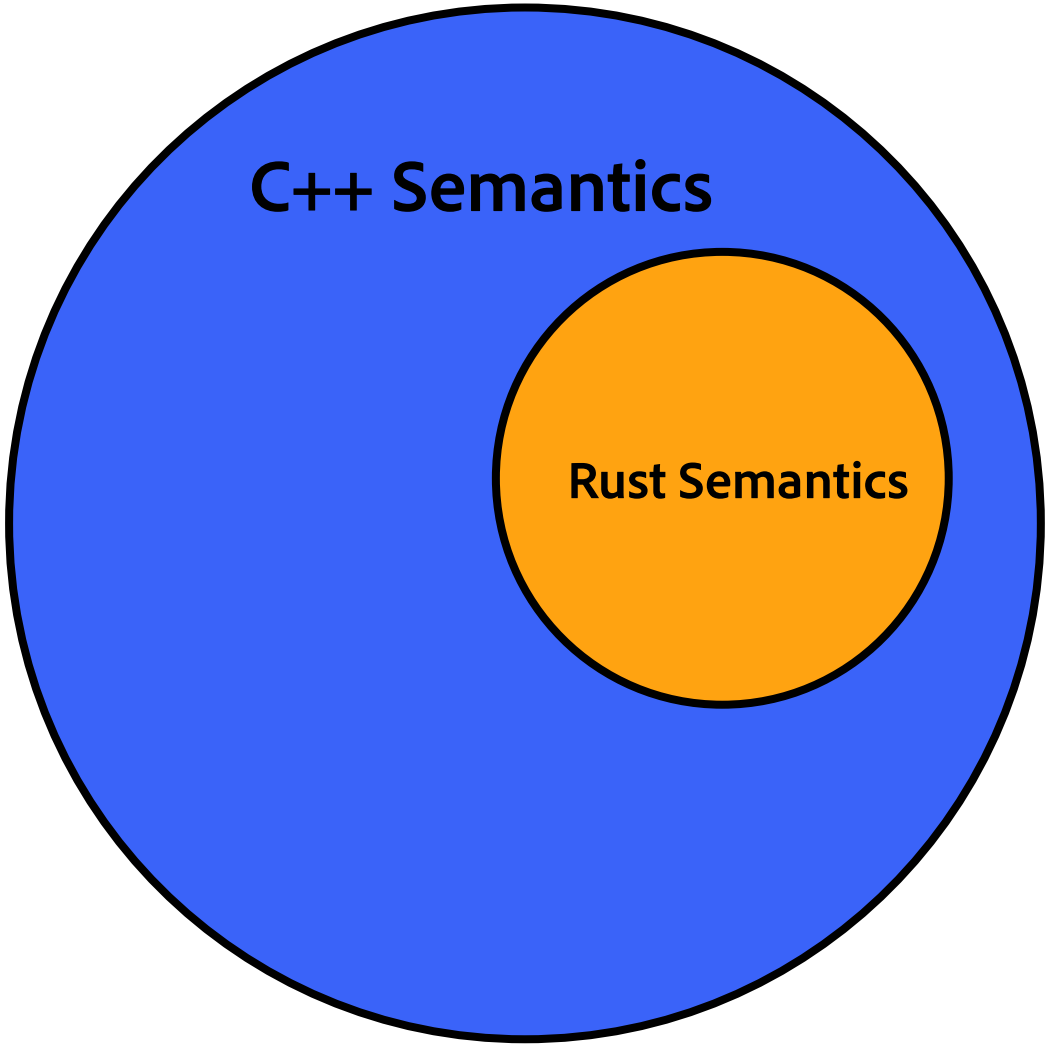
```
fn f(x: &mut i32, y: &mut i32) { /*... */ }

fn main() {
    let mut a = 3;
    f(&mut a, &mut a); // error: law of exclusivity
}
```

Zngur's driving principle: Rust is a subset of C++ Semantics



Adobe

Zngur's driving principle: Rust is a subset of C++ Semantics



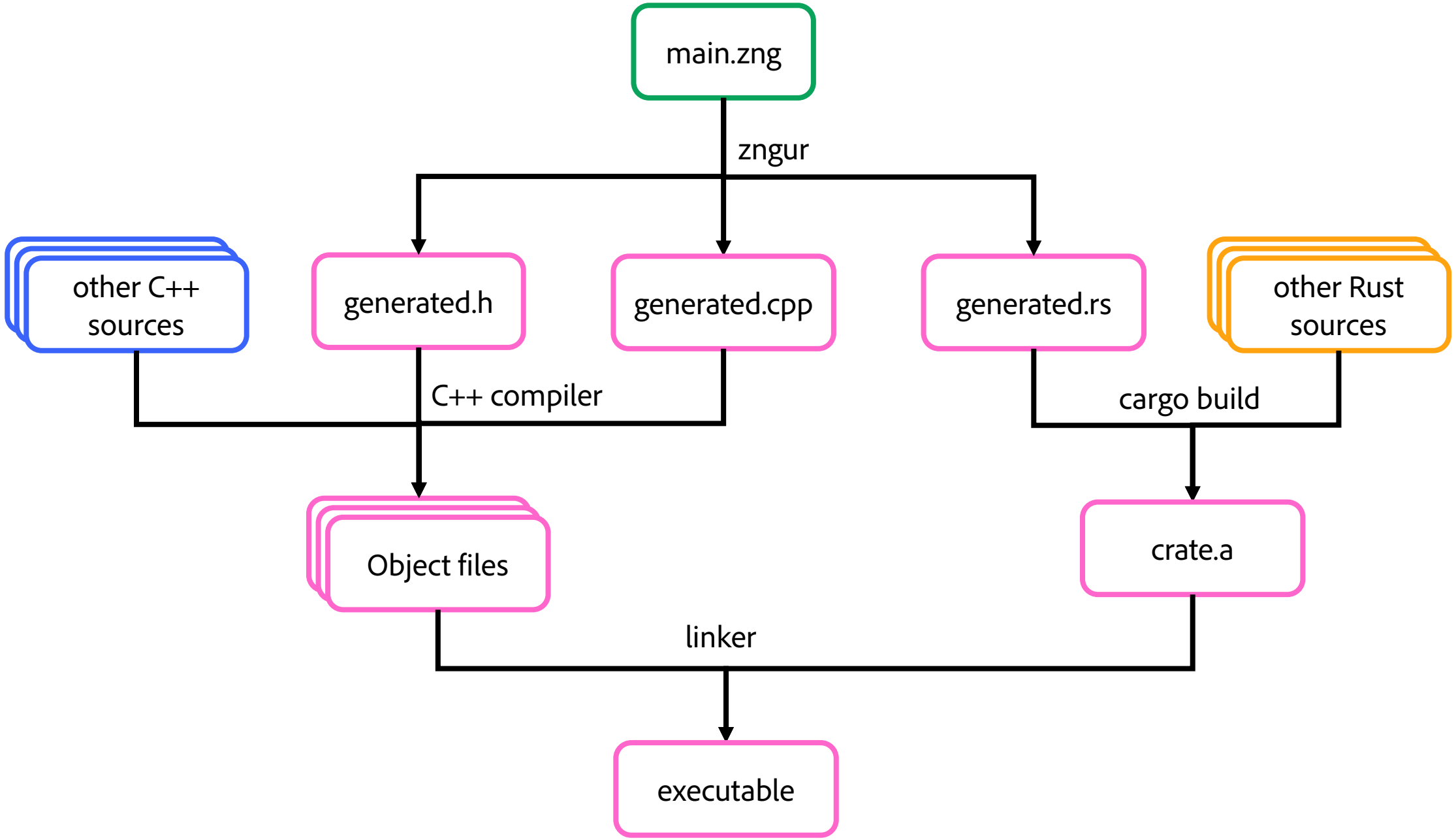
Observations

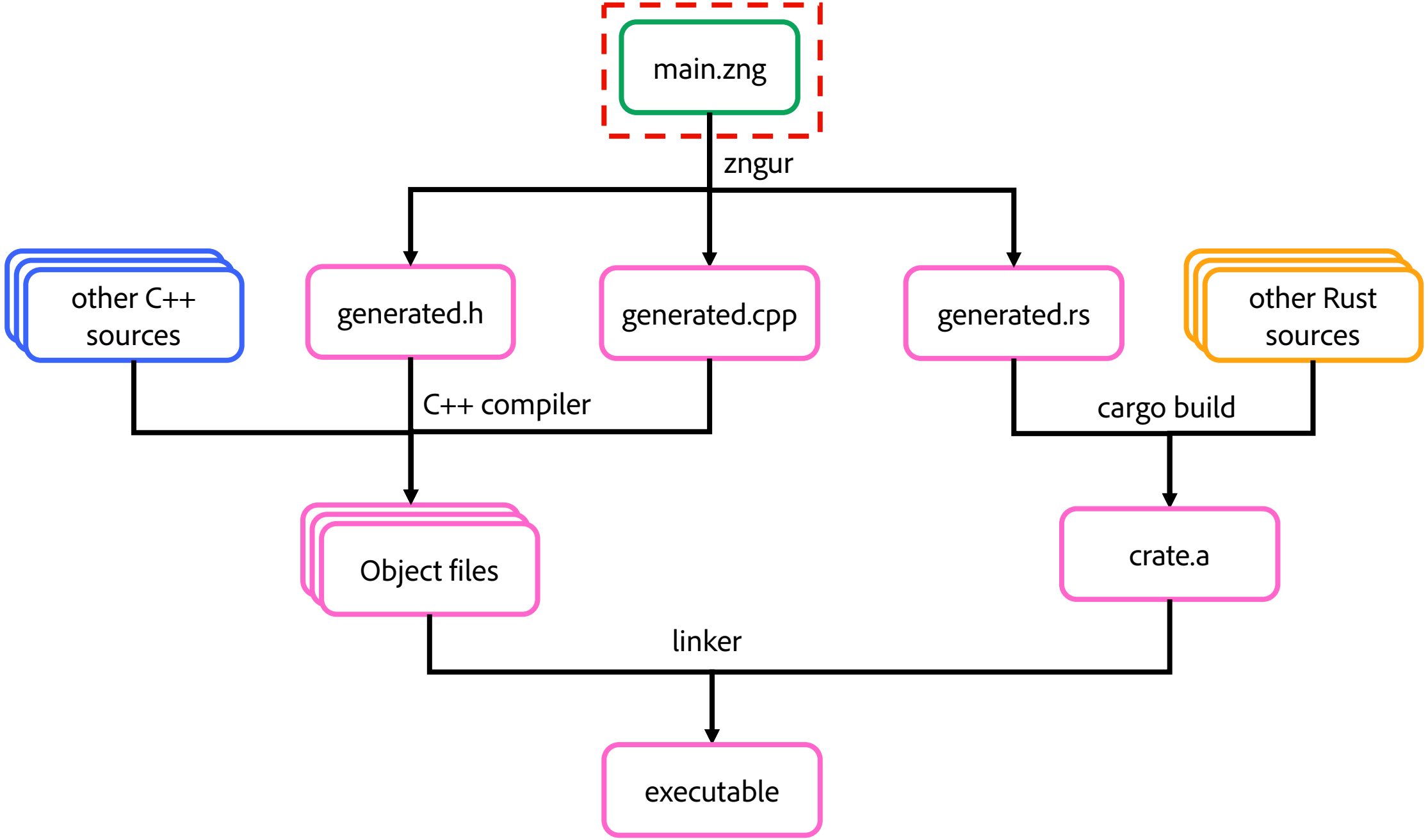
- Rust is less expressive than C++
- Rust's complexity lies in how its semantic subset is drawn

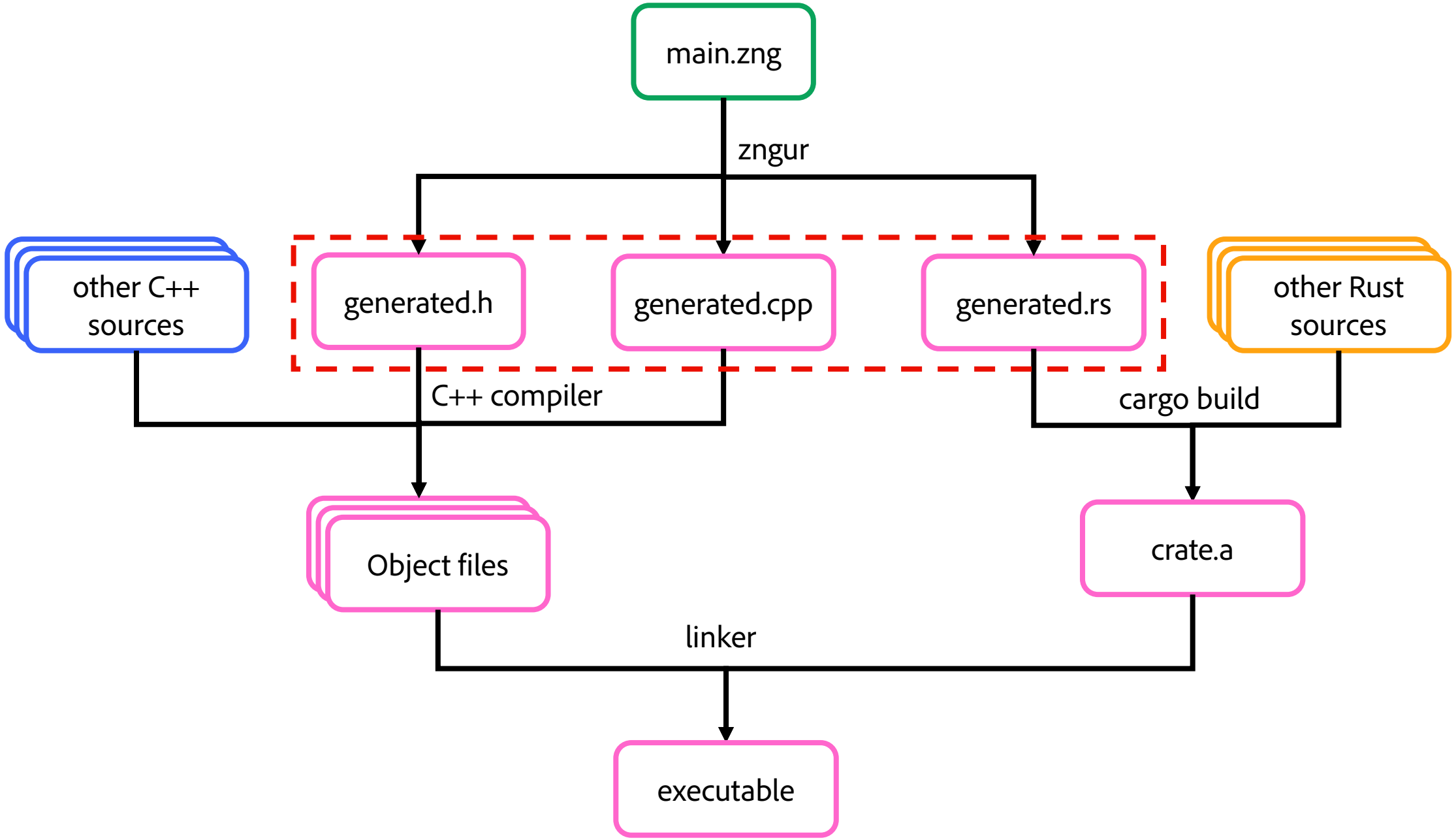
Implications

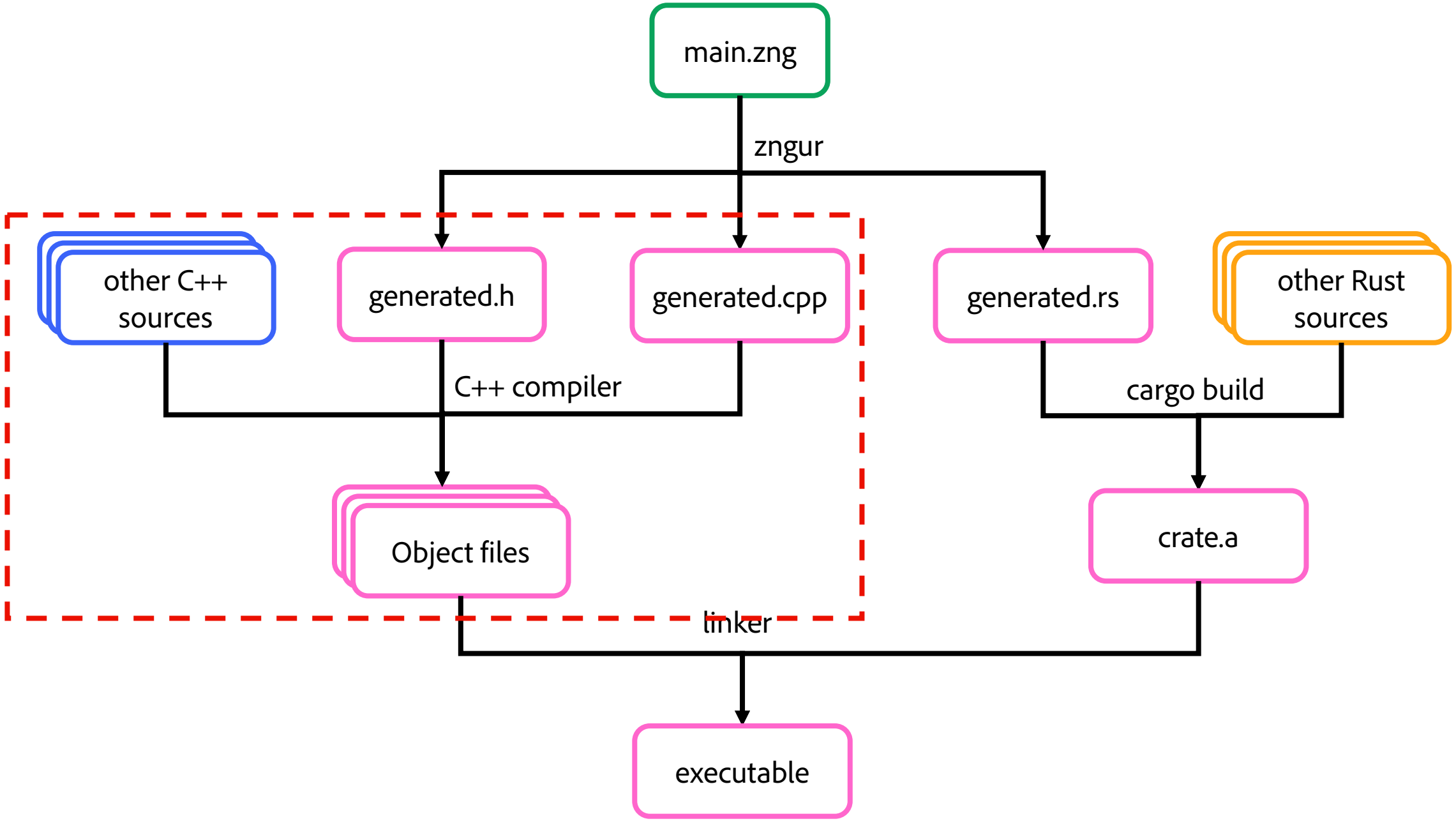
- Using Rust code from C++ should be easy
- Using C++ from Rust will be more challenging

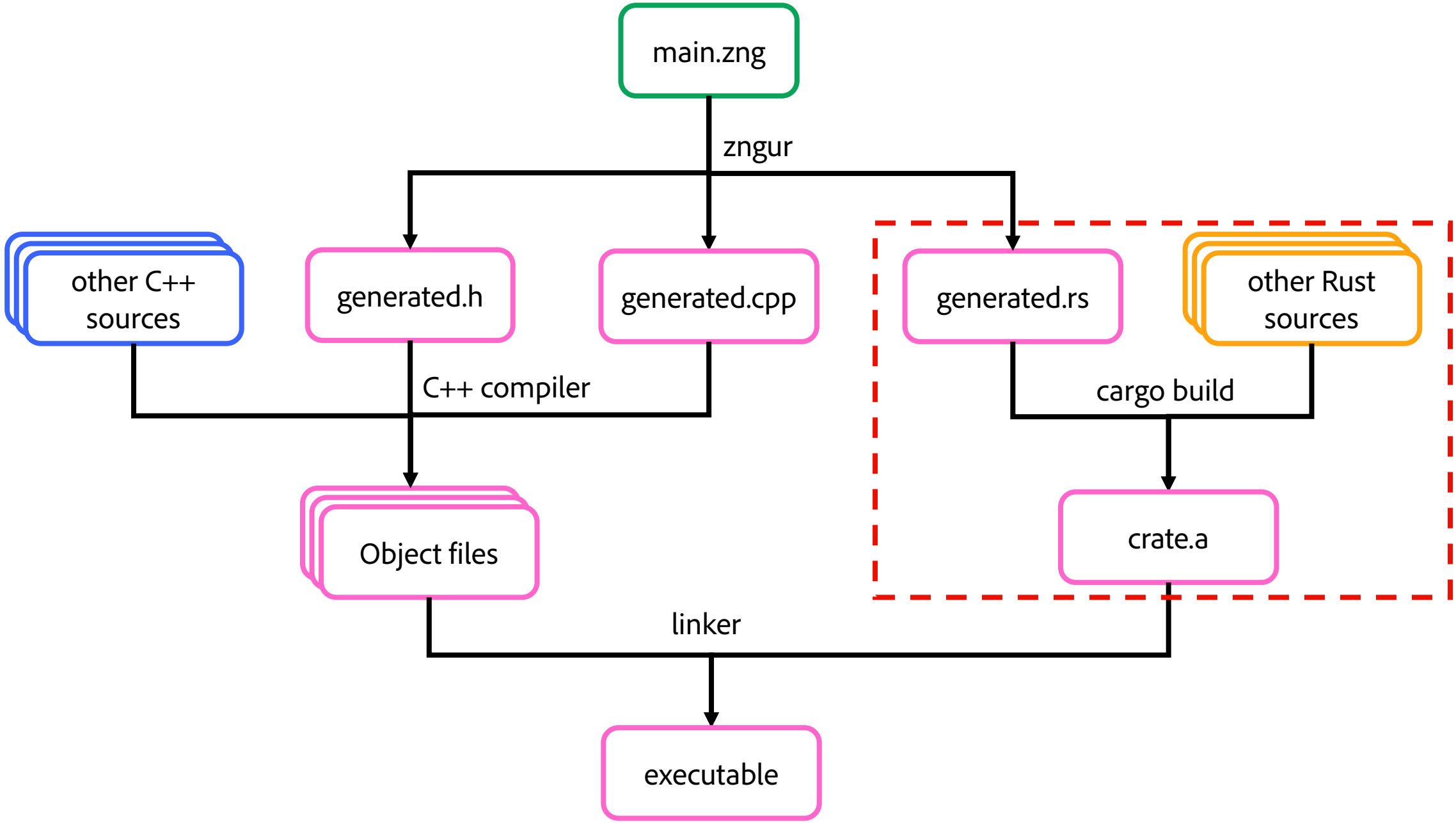
Zngur architecture

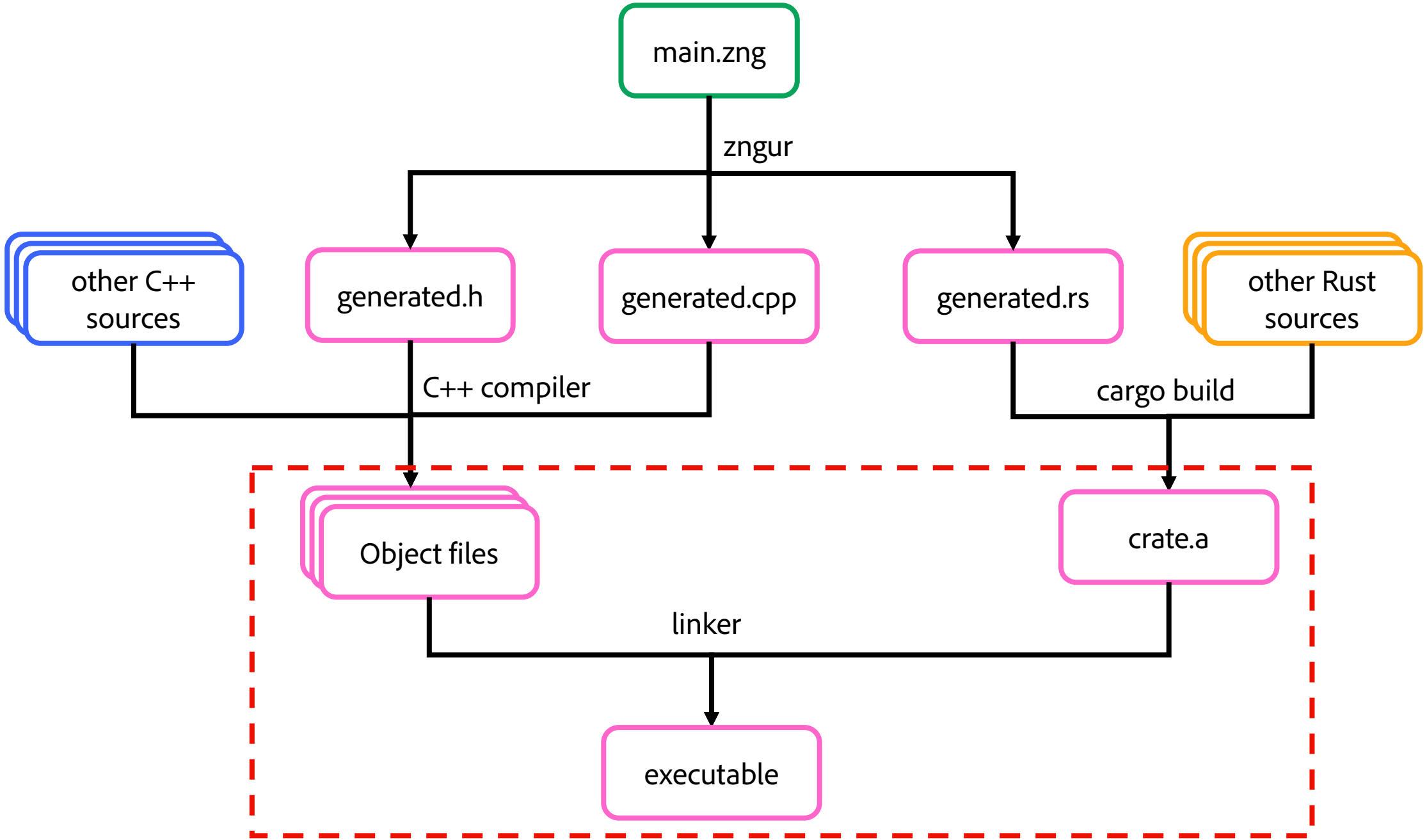

main.zng
zngur
other C++ sources
generated.h
generated.cpp
generated.rs
other Rust sources
C++ compiler
cargo build
Object files
crate.a
linker
executable


main.zng
zngur
other C++ sources
generated.h
generated.cpp
generated.rs
other Rust sources
C++ compiler
cargo build
Object files
crate.a
linker
executable


main.zng
zngur
other C++ sources
generated.h
generated.cpp
generated.rs
other Rust sources
C++ compiler
cargo build
Object files
crate.a
linker
executable


main.zng
zngur
other C++ sources
generated.h
generated.cpp
generated.rs
other Rust sources
C++ compiler
cargo build
Object files
crate.a
linker
executable


main.zng
zngur
other C++ sources
generated.h
generated.cpp
generated.rs
other Rust sources
C++ compiler
cargo build
Object files
crate.a
linker
executable


main.zng
zngur
other C++ sources
generated.h
generated.cpp
generated.rs
other Rust sources
C++ compiler
cargo build
Object files
crate.a
linker
executable

main.zng

main.zng

- Defines the interface boundary between C++ and Rust
- Self-contained
- Rust-like syntax
- Generatable

```
type str {
    wellknown_traits(?Sized);

    fn as_ptr(&self) -> *const u8;
    fn len(&self) -> usize;
    fn to_string(&self) -> ::std::string::String;
}

extern "C++" {
    impl crate::Inventory {
        fn new_empty(u32) -> crate::Inventory;
        fn add_banana(&mut self, u32);
        fn add_item(&mut self, crate::Item);
    }
}
```

Adobe

Rust → C++

Rust → C++

```
// lib.rs
#[derive(Clone)]
pub struct Person {
    name: String,
    age: u32,
}

impl Person {
    pub fn new(name: String, age: u32) -> Self {
        Self { name, age }
    }
    pub fn name(&self) -> &str {
        &self.name
    }
    pub fn age(&self) -> u32 {
        self.age
    }
}
```

Adobe

Rust → C++

```
// lib.rs
#[derive(Clone)]
pub struct Person {
    name: String,
    age: u32,
}

impl Person {
    pub fn new(name: String, age: u32) -> Self {
        Self { name, age }
    }
    pub fn name(&self) -> &str {
        &self.name
    }
    pub fn age(&self) -> u32 {
        self.age
    }
}
```

Adobe

Rust → C++

```
// lib.rs
#[derive(Clone)]
pub struct Person {
    name: String,
    age: u32,
}
```

```
impl Person {
    pub fn new(name: String, age: u32) -> Self {
        Self { name, age }
    }
    pub fn name(&self) -> &str {
        &self.name
    }
    pub fn age(&self) -> u32 {
        self.age
    }
}
```

Adobe

Rust → C++

```rust
// lib.rs
#[derive(Clone)]
pub struct Person {
    name: String,
    age: u32,
}

impl Person {
    pub fn new(name: String, age: u32) -> Self {
        Self { name, age }
    }
    pub fn name(&self) -> &str {
        &self.name
    }
    pub fn age(&self) -> u32 {
        self.age
    }
}
```

```
// main.zng
type crate::Person {
    #layout(size = 32, align = 8);

    fn new(::std::string::String, u32) -> crate::Person;
    fn name(&self) -> &str;
    fn age(&self) -> u32;

    // Clone methods
    fn clone(&self) -> crate::Person;
}
```

Adobe

Rust → C++ | initialization

```
// main.zng
type crate::Person {
   #layout(size = 32, align = 8);

   fn new(::std::string::String, u32) -> crate::Person;
   fn name(&self) -> &str;
   fn age(&self) -> u32;

   // Clone methods
   fn clone(&self) -> crate::Person;
}
```

```
// generated.h
namespace rust::crate {
class Person {
  public:
    //...
    // Creates an uninitialized `Person` object
    Person();
    //...
};
}
```

- Rust objects may be created uninitialized
- User code is responsible for initialization prior to use.
- Failure to initialize results in a segmentation fault (defined behavior).

Adobe

Rust → C++ | copy/clone

```
// main.zng
type crate::Person {
    #layout(size = 32, align = 8);

    fn new(::std::string::String, u32) -> crate::Person;
    fn name(&self) -> &str;
    fn age(&self) -> u32;

    // Clone methods
    fn clone(&self) -> crate::Person;
}
```

```
// generated.h
namespace rust::crate {
class Person {
  public:
    //...
    // Implicit copy operations disabled
    Person(const Person& other) = delete;
    Person& operator=(const Person& other) = delete;
    //...
    Person clone() const;
    //...
};
}
```

Copying objects (cloning in Rust) is explicit unless the type supports bitwise-copying. This carries over to the C++ interface.

Rust → C++ | methods

```
// main.zng
type crate::Person {
    #layout(size = 32, align = 8);

    fn new(::std::string::String, u32) -> crate::Person;
    fn name(&self) -> &str;
    fn age(&self) -> u32;

    // Clone methods
    fn clone(&self) -> crate::Person;
}
```

```
// generated.h
namespace rust::crate {
class Person {
  public:
    //...
    ::rust::Ref<::rust::Str> name() const;
    ::uint32_t age() const;
    // ...
};
}
```

Methods work as you might expect

Adobe

Rust references and C++ references

```
::rust::Ref<::rust::Str> name() const;
```

- Rust references are different from C++ references
  - Rust's mutable references (&mut) are exclusive. C++'s non-const references are not exclusive.
  - Rust's shared references (&) imply value stability. C++'s const references do not.
  - Rust references may be "fat" (e.g. encode additional information). C++'s references are always "thin".
- To account for differences, Zngur provides Ref and RefMut templates
  - Specializations are generated for every interface-layer type
  - Specializations include the pointer and (for dynamically-sized types) additional metadata
  - Specializations include appropriate member functions for the pointed-to objects

Rust references and C++ references

```
::rust::Ref<::rust::Str> name() const;
```

- Rust references are different from C++ references
  - Rust's mutable references (&mut) are exclusive. C++'s non-const references are not exclusive.
  - Rust's shared references (&) imply value stability. C++'s const references do not.
  - Rust references may be "fat" (e.g. encode additional information). C++'s references are always "thin".
- To account for differences, Zngur provides Ref and RefMut templates
  - Specializations are generated for every interface-layer type
  - Specializations include the pointer and (for dynamically-sized types) additional metadata
  - Specializations include appropriate member functions for the pointed-to objects

Rust references and C++ references

```
::rust::Ref<::rust::Str> name() const;
```

- Rust references are different from C++ references
  - Rust's mutable references (&mut) are exclusive. C++'s non-const references are not exclusive.
  - Rust's shared references (&) imply value stability. C++'s const references do not.
  - Rust references may be "fat" (e.g. encode additional information). C++'s references are always "thin".
- To account for differences, Zngur provides Ref and RefMut templates
  - Specializations are generated for every interface-layer type
  - Specializations include the pointer and (for dynamically-sized types) additional metadata
  - Specializations include appropriate member functions for the pointed-to objects

Rust references and C++ references

```
::rust::Ref<::rust::Str> name() const;
```

- Rust references are different from C++ references
  - Rust's mutable references (&mut) are exclusive. C++'s non-const references are not exclusive.
  - Rust's shared references (&) imply value stability. C++'s const references do not.
  - Rust references may be "fat" (e.g. encode additional information). C++'s references are always "thin".
- To account for differences, Zngur provides Ref and RefMut templates
  - Specializations are generated for every interface-layer type
  - Specializations include the pointer and (for dynamically-sized types) additional metadata
  - Specializations include appropriate member functions for the pointed-to objects

Rust references and C++ references

```
::rust::Ref<::rust::Str> name() const;
```

- Rust references are different from C++ references
  - Rust's mutable references (&mut) are exclusive. C++'s non-const references are not exclusive.
  - Rust's shared references (&) imply value stability. C++'s const references do not.
  - Rust references may be "fat" (e.g. encode additional information). C++'s references are always "thin".
- To account for differences, Zngur provides Ref and RefMut templates
  - Specializations are generated for every interface-layer type
  - Specializations include the pointer and (for dynamically-sized types) additional metadata
  - Specializations include appropriate member functions for the pointed-to objects

Adobe

Working with Rust references in C++

```
using namespace rust;
using namespace rust::crate::Person;

int main() {
    Person p = Person::new_( /*...*/ );
    {
        Ref<Person> pRef(p);
        std::cout << pRef.age() << std::endl;
    }
    {
        RefMut<Person> pRef(p);
        std::cout << pRef.age() << std::endl;
    }
}
```

Adobe

Working with Rust references in C++

```
using namespace rust;
using namespace rust::crate::Person;

int main() {
    Person p = Person::new_( /*...*/ );
    {
        Ref<Person> pRef(p);
        std::cout << pRef.age() << std::endl;
    }
    {
        RefMut<Person> pRef(p);
        std::cout << pRef.age() << std::endl;
    }
}
```

Adobe

Working with Rust references in C++

```
using namespace rust;
using namespace rust::crate::Person;

int main() {
    Person p = Person::new_( /*...*/ );
    {
        Ref<Person> pRef(p);
        std::cout << pRef.age() << std::endl;
    }
    {
        RefMut<Person> pRef(p);
        std::cout << pRef.age() << std::endl;
    }
}
```

The C++ code must uphold that p is not modified while pRef exists

Adobe

Working with Rust references in C++

```
using namespace rust;
using namespace rust::crate::Person;

int main() {
    Person p = Person::new_( /*...*/ );
    {
        Ref<Person> pRef(p);
        std::cout << pRef.age() << std::endl;
    }
    {
        RefMut<Person> pRef(p);
        std::cout << pRef.age() << std::endl;
    }
}
```

Methods can be called on Ref objects directly.

Working with Rust references in C++

```
using namespace rust;
using namespace rust::crate::Person;

int main() {
    Person p = Person::new_( /*...*/ );
    {
        Ref<Person> pRef(p);
        std::cout << pRef.age() << std::endl;
    }
    {
        RefMut<Person> pRef(p);
        std::cout << pRef.age() << std::endl;
    }
}
```

The C++ code must uphold:

a) *There is at most one RefMut referencing p at any given time.*

b) *p may only be modified through pRef.*

Adobe

Rust → C++ | additional function syntax

```
// main.zng
type crate::Person {
    #layout(size = 32, align = 8);

    fn new(::std::string::String, u32) -> crate::Person;
    fn name(&self) -> &str;
    fn age(&self) -> u32;

    // Clone methods
    fn clone(&self) -> crate::Person;
}
```

```
// generated.h
namespace rust::crate {
class Person {
  public:
    //...
    static ::rust::Ref<::rust::Str> name(
        ::rust::Ref< ::rust::crate::Person >);

    static ::uint32_t age(
        ::rust::Ref< ::rust::crate::Person >);
    // ...
};
}
```

Methods are provided as static functions as well for Rust syntax consistency

Adobe

Rust → C++ | layout

```
// main.zng
type crate::Person {
    #layout(size = 32, align = 8);

    fn new(::std::string::String, u32) -> crate::Person;
    fn name(&self) -> &str;
    fn age(&self) -> u32;

    // Clone methods
    fn clone(&self) -> crate::Person;
}
```

- The layout directive declares the size and alignment of the Rust type
- Required for placing Rust objects on the C++ stack
- When incorrect, *generated.rs* will not compile.
- Hovering over a type will give you this information in a Rust IDE.

C++ → Rust

C++ → Rust is more complex

- Requires adapter code for most things
- C++'s object model generally isn't directly representable in Rust

Adobe

Expose a C++ function to Rust

```
// main.zng

type crate::Person {
    //...
}

extern "C++" {
    fn older(&crate::Person) -> crate::Person;
}
```

Adobe

Expose a C++ function to Rust

```
// main.zng

type crate::Person {
    //...
}

extern "C++" {
    fn older(&crate::Person) -> crate::Person;
}
```

Adobe

Expose a C++ function to Rust

```
// main.zng

type crate::Person {
    //...
}

extern "C++" {
    fn older(&crate::Person) -> crate::Person;
}
```

```
// generated.h

namespace rust::exported_functions {
    ::rust::crate::Person older(
        ::rust::Ref< ::rust::crate::Person >);
}
```

```
// impl.cpp
using namespace rust::crate;
using namespace rust;

Person rust::exported_functions::older(Ref<Person> p) {
  return Person::new_(p.name().to_string(),
                      p.age()+5);
}
```

Adobe

Expose a C++ function to Rust

```
// main.zng

type crate::Person {
    //...
}

extern "C++" {
    fn older(&crate::Person) -> crate::Person;
}
```

```
// generated.h

namespace rust::exported_functions {
    ::rust::crate::Person older(
        ::rust::Ref< ::rust::crate::Person >);
}
```

```
// impl.cpp
using namespace rust::crate;
using namespace rust;

Person rust::exported_functions::older(Ref<Person> p) {
  return Person::new_(p.name().to_string(),
                      p.age()+5);
}
```

Adobe

Expose a C++ object to Rust

C++

```
struct People {
  std::vector<rust::crate::Person> data;
};

// ...
 ::People people;
people.data.push_back( Person::new_(
    Str::from_char_star("Beyoncé").to_string(),
    43));
people.data.push_back( Person::new_(
    Str::from_char_star("Jay-Z").to_string(),
    55));
print_people(people);
```

Adobe

Expose a C++ object to Rust

C++

```
struct People {
  std::vector<rust::crate::Person> data;
};
```

```
// ...
 ::People people;
people.data.push_back( Person::new_(
    Str::from_char_star("Beyoncé").to_string(),
    43));
people.data.push_back( Person::new_(
    Str::from_char_star("Jay-Z").to_string(),
    55));
print_people(people);
```

Expose a C++ object to Rust

C++

```
struct People {
  std::vector<rust::crate::Person> data;
};

// ...
 ::People people;
people.data.push_back( Person::new_(
    Str::from_char_star("Beyoncé").to_string(),
    43));
people.data.push_back( Person::new_(
    Str::from_char_star("Jay-Z").to_string(),
    55));
print_people(people);
```

Adobe

Expose a C++ object to Rust

C++

```
struct People {
  std::vector<rust::crate::Person> data;
};

// ...
 ::People people;
people.data.push_back( Person::new_(
    Str::from_char_star("Beyoncé").to_string(),
    43));
people.data.push_back( Person::new_(
    Str::from_char_star("Jay-Z").to_string(),
    55));
print_people(people);
```

Rust

```
pub fn print_people(p: &People) {
    println!("{} people:", p.len());
    for i in 0..p.len() {
        println!("- {}", p.index(i).name())
    }
}
```

Zngur

```
mod crate {
    fn print_people(&People);
}
```

Adobe

Expose a C++ object to Rust

C++

```
struct People {
  std::vector<rust::crate::Person> data;
};

::size_t Impl<rust::crate::People>::len(
        Ref<rust::crate::People> p) {
  return p.cpp().data.size();
}

Ref<Person> Impl<rust::crate::People>::index(
       Ref<rust::crate::People> p, ::size_t index) {
  if (index >= p.cpp().data.size() ) {
    abort();
  }
  return p.cpp().data[index];
}
```

Rust

```
pub struct People(
    ZngurCppOpaqueBorrowedObject);
```

Zngur

```
type crate::People {
    #cpp_ref "People";
}

extern "C++" {
    impl crate::People {
        fn len(&self) -> usize;
        fn index(&self, usize) -> &crate::Person;
    }
}
```

Adobe

Expose a C++ object to Rust

C++

```
struct People {
  std::vector<rust::crate::Person> data;
};

::size_t Impl<rust::crate::People>::len(
        Ref<rust::crate::People> p) {
  return p.cpp().data.size();
}

Ref<Person> Impl<rust::crate::People>::index(
       Ref<rust::crate::People> p, ::size_t index) {
  if (index >= p.cpp().data.size() ) {
    abort();
  }
  return p.cpp().data[index];
}
```

Rust

```
pub struct People(
    ZngurCppOpaqueBorrowedObject);
```

Zngur

```
type crate::People {
    #cpp_ref "People";
}

extern "C++" {
    impl crate::People {
        fn len(&self) -> usize;
        fn index(&self, usize) -> &crate::Person;
    }
}
```

Adobe

Expose a C++ object to Rust

C++

```
struct People {
  std::vector<rust::crate::Person> data;
};

::size_t Impl<rust::crate::People>::len(
        Ref<rust::crate::People> p) {
  return p.cpp().data.size();
}

Ref<Person> Impl<rust::crate::People>::index(
        Ref<rust::crate::People> p, ::size_t index) {
  if (index >= p.cpp().data.size() ) {
    abort();
  }
  return p.cpp().data[index];
}
```

Rust

```
pub struct People(
    ZngurCppOpaqueBorrowedObject);
```

Zngur

```
type crate::People {
    #cpp_ref "People";
}

extern "C++" {
    impl crate::People {
        fn len(&self) -> usize;
        fn index(&self, usize) -> &crate::Person;
    }
}
```

Adobe

Expose a C++ object to Rust

C++

```
struct People {
  std::vector<rust::crate::Person> data;
};

::size_t Impl<rust::crate::People>::len(
        Ref<rust::crate::People> p) {
  return p.cpp().data.size();
}

Ref<Person> Impl<rust::crate::People>::index(
       Ref<rust::crate::People> p, ::size_t index) {
 if (index >= p.cpp().data.size() ) {
   abort();
 }
 return p.cpp().data[index];
}
```

Rust

```
pub struct People(
    ZngurCppOpaqueBorrowedObject);
```

Zngur

```
type crate::People {
    #cpp_ref "People";
}

extern "C++" {
    impl crate::People {
        fn len(&self) -> usize;
        fn index(&self, usize) -> &crate::Person;
    }
}
```

Adobe

#cpp_ref

- Exposes *references* to C++ objects in Rust
- Cannot be used to obtain a value
- Efficient

Rust

```
pub struct People(
    ZngurCppOpaqueBorrowedObject);
```

Zngur

```
type crate::People {
    #cpp_ref "People";
}

extern "C++" {
    impl crate::People {
        fn len(&self) -> usize;
        fn index(&self, usize) -> &crate::Person;
    }
}
```

Adobe

#cpp_value

C++

```
rust::crate::People rust::Impl<rust::crate::People>::new_() {
  return rust::crate::People(
    rust::ZngurCppOpaqueOwnedObject::build<::People>());
}
```

Rust

```
pub struct People(
    ZngurCppOpaqueOwnedObject);
```

Zngur

```
type crate::People {
    #cpp_ref "People";
    #layout(size = 16, align = 8);
    #cpp_value "0" "::People";
    constructor(ZngurCppOpaqueOwnedObject);
}
extern "C++" {
    impl crate::People {
        fn new() -> crate::People;
        //...
    }
```

Adobe

#cpp_value

C++

```
rust::crate::People rust::Impl<rust::crate::People>::new_() {
  return rust::crate::People(
    rust::ZngurCppOpaqueOwnedObject::build<::People>());
}
```

Rust

```
pub struct People(
    ZngurCppOpaqueOwnedObject);
```

Zngur

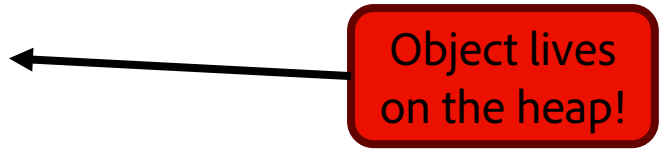
```
type crate::People {
    #cpp_ref "People";
    #layout(size = 16, align = 8);
    #cpp_value "0" "::People";
    constructor(ZngurCppOpaqueOwnedObject);
}
extern "C++" {
    impl crate::People {
        fn new() -> crate::People;
        //...
    }
```

Adobe

#cpp_value | use from Rust

Rust

```
fn test() {
    let p = People::new();
    println!("{}", p.len());
}
```

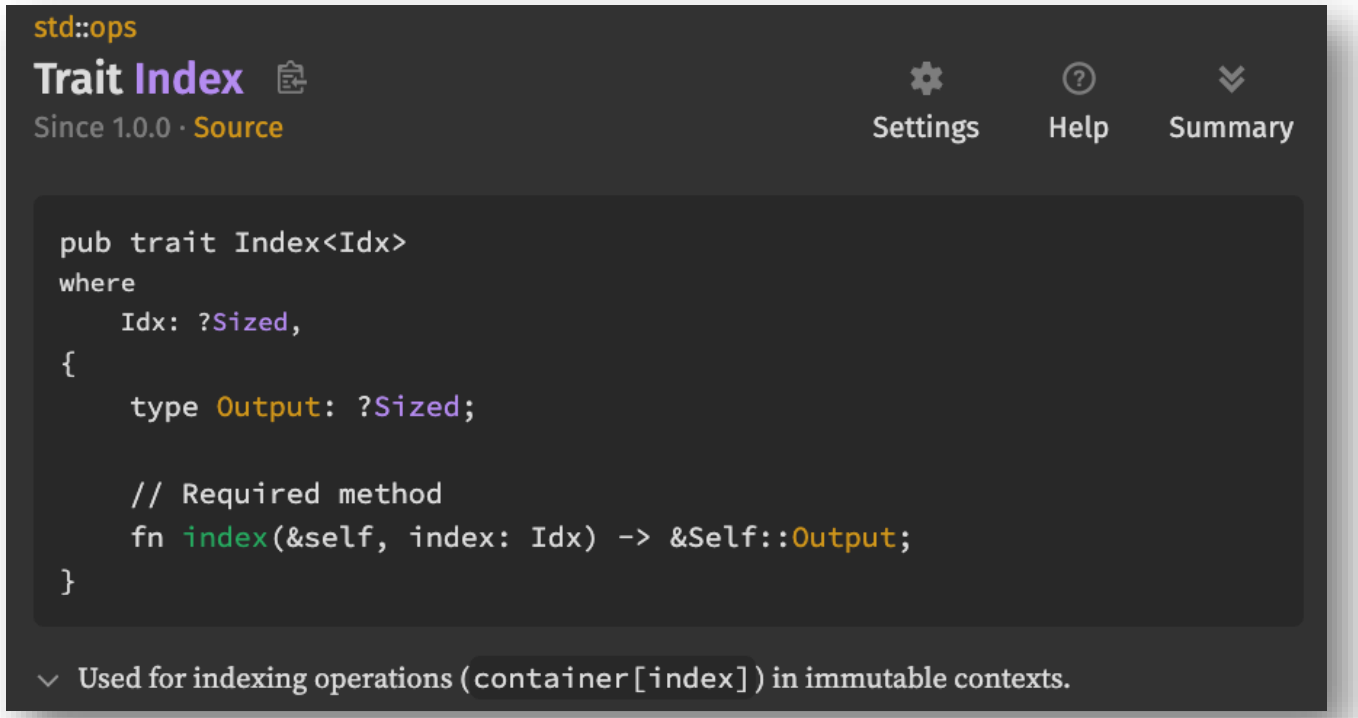


Traits

Zngur

```
extern "C++" {
    impl crate::People {
        //...
        fn index(&self, usize) -> &crate::Person;
        //....
    }
}
```

Traits



Zngur

```
extern "C++" {
   impl crate::People {
      //...
      fn index(&self, usize) -> &crate::Person;
      //....
   }
    impl ::std::ops::Index<
            usize,
            Output=crate::Person>
            for crate::People {
      fn index(&self, usize) -> &crate::Person;
   }
}
```

Traits

~~Ref<Person> Impl<rust::crate::People>::index(~~
~~Ref<rust::crate::People> p, ::size_t index) {~~

```
Ref<Person> Impl<rust::crate::People,
      rust::std::ops::Index<::size_t,
                            rust::crate::Person>>
   ::index(Ref<rust::crate::People> p,
           ::size_t index) {
  if (index >= p.cpp().data.size() ) {
    abort();
  }
  return p.cpp().data[index];
}
```

Zngur

```
extern "C++" {
   impl ::std::ops::Index<
            usize,
            Output=crate::Person>
            for crate::People {
      fn index(&self, usize) -> &crate::Person;
   }
}
```

Traits

```
pub fn print_people(p: &People) {
    println!("{} people:", p.len());
    for i in 0..p.len() {
        println!("- {}", p.index(i).name())
        println!("- {}", p[i].name())
    }
}
```

Zngur

```
extern "C++" {
    impl ::std::ops::Index<
            usize,
            Output=crate::Person>
            for crate::People {
        fn index(&self, usize) -> &crate::Person;
    }
}
```

Adobe

Dynamic traits

- Rust provides a mechanism that allows for dynamic dispatch of trait methods at runtime - dyn.
- Zngur allows C++ types to implement dynamic traits using abstract base classes (e.g. dyn)

```cpp
template <typename T>
class VectorIterator : public rust::std::iter::Iterator<T> {
  std::vector<T> vec;
  size_t pos;

public:
  VectorIterator(std::vector<T> &&v) : vec(v), pos(0) {}
  ~VectorIterator() {
    std::cout << "vector iterator has been destructed" << std::endl;
  }

  Option<T> next() override {
    if (pos >= vec.size()) {
      return Option<T>::None();
    }
    T value = vec[pos++];
    // You can construct Rust enum with fields in C++
    return Option<T>::Some(value);
  }
};
```

PNG parsing from C++

CVE List ▾ CNAs ▾ WGs ▾ Board ▾
About ▾ News ▾

Search CVE List | Downloads | Data Feeds | Update a CVE Record | Request CVE IDs

TOTAL CVE Records: 276577

NOTICE: Transition to the all-new CVE website at WWW.CVE.ORG and CVE Record Format JSON are underway.

NOTICE: Support for the legacy CVE download formats ended on June 30, 2024.
New CVE List download format is available now on CVE.ORG.

HOME > CVE > SEARCH RESULTS

Search Results

There are **75** CVE Records that match your search.

Name	Description
CVE-2021-4214	A heap overflow flaw was found in libpngs' pngimage.c program. This flaw allows an attacker with local network access to pass a specially crafted PNG file to the pngimage utility, causing an application to crash, leading to a denial of service.
CVE-2019-9423	In opencv calls that use libpng, there is a possible out of bounds write due to a missing bounds check. This could lead to local escalation of privilege with no additional execution privileges required. User interaction is not required for exploitation. Product: AndroidVersions: Android-10Android ID: A-110986616
CVE-2019-7317	png_image_free in png.c in libpng 1.6.x before 1.6.37 has a use-after-free because png_image_free_function is called under png_safe_execute.
CVE-2019-6129	** DISPUTED ** png_create_info_struct in png.c in libpng 1.6.36 has a memory leak, as demonstrated by pngcp. NOTE: a third party has stated "I don't think it is libpng's job to free this buffer."
CVE-2019-3572	An issue was discovered in libming 0.4.8. There is a heap-based buffer over-read in the function writePNG in the file util/dbl2png.c of the dbl2png command-line program. Because this is associated with an erroneous call to png_write_row in libpng, an out-of-bounds write might occur for some memory layouts.
CVE-2019-14373	An issue was discovered in image_save_png in image/image-png.cpp in Free Lossless Image Format (FLIF) 0.3. Attackers can trigger a heap-based buffer over-read in libpng via a crafted flif file.
CVE-2018-14876	An issue was discovered in image_save_png in image/image-png.cpp in Free Lossless Image Format (FLIF) 0.3. Attackers can trigger a longjmp that leads to an uninitialized stack frame after a libpng error concerning the IHDR image width.
CVE-2018-14550	An issue has been found in third-party PNM decoding associated with libpng 1.6.35. It is a stack-based buffer overflow in the function get_token in pnm2png.c in pnm2png.

PNG Decoder/Encoder

Rust CI passing | docs passing | crates.io v0.18.0-rc | license MIT OR Apache-2.0

Robust and performant PNG decoder/encoder in pure Rust. Also supports APNG.

No `unsafe` code, battle-tested, and fuzzed on OSS-fuzz.

Performance

Performance is typically on par with or better than libpng.

Includes a fast encoding mode powered by fdeflate that is dramatically faster than the fastest mode of libpng while *simultaneously* providing better compression ratio.

On nightly Rust compiler you can slightly speed up decoding of some images by enabling the `unstable` feature of this crate.

Adobe

Safe PNG parsing from C++

```
#include <iostream>
#include <generated.h>

int main() {
  auto path = rust::Str::from_char_star("test.png");

  auto decoder = rust::png::Decoder<rust::std::fs::File>
        ::new_(rust::std::fs::File::open(path).unwrap());

  auto header_info = decoder.read_header_info()
                                    .unwrap();
  std::cout << "Width=" << header_info.width() << '\n';
            "\nHeight=" << header_info.height() << '\n';
}
```

Safe PNG parsing from C++

```
#include <iostream>
#include <generated.h>

int main() {
 auto path = rust::Str::from_char_star("test.png");

  auto decoder = rust::png::Decoder<rust::std::fs::File>
        ::new_(rust::std::fs::File::open(path).unwrap());

  auto header_info = decoder.read_header_info()
                                    .unwrap();
  std::cout << "Width=" << header_info.width() << '\n';
             "\nHeight=" << header_info.height() << '\n';
}
```

Adobe

Safe PNG parsing from C++

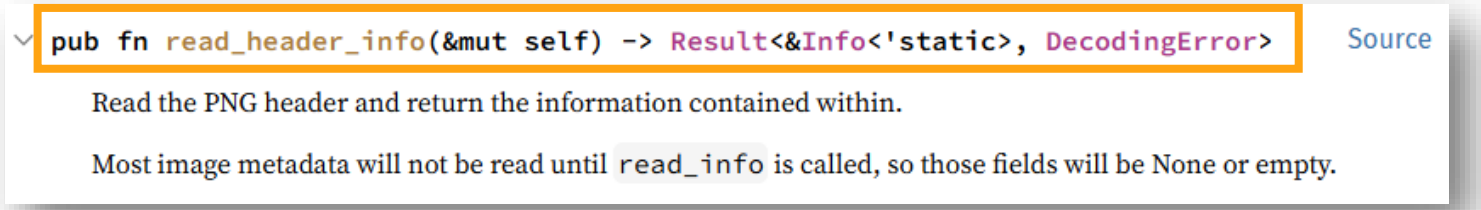
```
#include <iostream>
#include <generated.h>

int main() {
  auto path = rust::Str::from_char_star("test.png");

  auto decoder = rust::png::Decoder<rust::std::fs::File>
        ::new_(rust::std::fs::File::open(path).unwrap());

  auto header_info = decoder.read_header_info()
                                    .unwrap();
  std::cout << "Width=" << header_info.width() << '\n';
               "\nHeight=" << header_info.height() << '\n';
}
```

Using the decoder

```rust
use std::fs::File;
// The decoder is a build for reader and can be used to set various decoding options
// via `Transformations`. The default output transformation is `Transformations::IDENTITY`.
let decoder = png::Decoder::new(File::open("tests/pngsuite/basi0g01.png").unwrap());
let mut reader = decoder.read_info().unwrap();
// Allocate the output buffer.
let mut buf = vec![0; reader.output_buffer_size()];
// Read the next frame. An APNG might contain multiple frames.
let info = reader.next_frame(&mut buf).unwrap();
// Grab the bytes of the image.
let bytes = &buf[..info.buffer_size()];
// Inspect more details of the last read frame.
let in_animation = reader.info().frame_control.is_some();
```

Safe PNG parsing from C++

```
#include <iostream>
#include <generated.h>

int main() {
  auto path = rust::Str::from_char_star("test.png");

  auto decoder = rust::png::Decoder<rust::std::fs::File>
        ::new_(rust::std::fs::File::open(path).unwrap());

  auto header_info = decoder.read_header_info()
                            .unwrap();
  std::cout << "Width=" << header_info.width() << '\n';
            "\nHeight=" << header_info.height() << '\n';
}
```



Safe PNG parsing from C++

```
#include <iostream>
#include <generated.h>

int main() {
  auto path = rust::Str::from_char_star("test.png");

  auto decoder = rust::png::Decoder<rust::std::fs::File>
        ::new_(rust::std::fs::File::open(path).unwrap());

  auto header_info = decoder.read_header_info()
                                    .unwrap();
  std::cout << "Width=" << header_info.width() << '\n';
               "\nHeight=" << header_info.height() << '\n';
}
```

Safe PNG parsing from C++

```
#include <iostream>
#include <generated.h>

int main() {
  auto path = rust::Str::from_char_star("test.png");

  auto decoder = rust::png::Decoder<rust::std::fs::File>
        ::new_(rust::std::fs::File::open(path).unwrap());

  auto header_info = decoder.read_header_info()
                                  .unwrap();
  std::cout << "Width=" << header_info.width() << '\n';
               "\nHeight=" << header_info.height() << '\n';
}
```

```
include!("generated.rs");

trait PngInfoExt {
    fn width(&self) -> u32;
    fn height(&self) -> u32;
}

impl<'a> PngInfoExt for png::Info<'a> {
    fn width(&self) -> u32 {
        self.width
    }
    fn height(&self) -> u32 {
        self.height
    }
}
```

Safe PNG parsing from C++

```
#include <iostream>
#include <generated.h>

int main() {
  auto path = rust::Str::from_char_star("test.png");

  auto decoder = rust::png::Decoder<rust::std::fs::File>
        ::new_(rust::std::fs::File::open(path).unwrap());

  auto header_info = decoder.read_header_info()
                                    .unwrap();
  std::cout << "Width=" << header_info.width() << '\n';
               "\nHeight=" << header_info.height() << '\n';
}
```

```
include!("generated.rs");

trait PngInfoExt {
    fn width(&self) -> u32;
    fn height(&self) -> u32;
}

impl<'a> PngInfoExt for png::Info<'a> {
    fn width(&self) -> u32 {
        self.width
    }
    fn height(&self) -> u32 {
        self.height
    }
}
```

Adobe

Safe PNG parsing from C++

```
#include <iostream>
#include <generated.h>

int main() {
  auto path = rust::Str::from_char_star("test.png");

  auto decoder = rust::png::Decoder<rust::std::fs::File>
        ::new_(rust::std::fs::File::open(path).unwrap());

  auto header_info = decoder.read_header_info()
                                    .unwrap();
  std::cout << "Width=" << header_info.width() << '\n';
             "\nHeight=" << header_info.height() << '\n';
}
```

```
// main.zng
type str {
    wellknown_traits(?Sized);
}

type std::fs::File {
    #layout(size = 8, align = 8);
    fn open(&str) -> std::io::Result<std::fs::File>;
}

type std::io::Result<std::fs::File> {
    #layout(size = 16, align = 8);
    fn unwrap(self) -> std::fs::File;
}

type png::Decoder<std::fs::File> {
```

Adobe

Safe PNG parsing from C++

```
#include <iostream>
#include <generated.h>

int main() {
  auto path = rust::Str::from_char_star("test.png");

  auto decoder = rust::png::Decoder<rust::std::fs::File>
        ::new_(rust::std::fs::File::open(path).unwrap());

  auto header_info = decoder.read_header_info()
                                    .unwrap();
  std::cout << "Width=" << header_info.width() << '\n';
               "\nHeight=" << header_info.height() << '\n';
}
```

```
// main.zng
type str {
    wellknown_traits(?Sized);
}

type std::fs::File {
    #layout(size = 8, align = 8);
    fn open(&str) -> std::io::Result<std::fs::File>;
}

type std::io::Result<std::fs::File> {
    #layout(size = 16, align = 8);
    fn unwrap(self) -> std::fs::File;
}

type png::Decoder<std::fs::File> {
```

Safe PNG parsing from C++

```
#include <iostream>
#include <generated.h>

int main() {
  auto path = rust::Str::from_char_star("test.png");

  auto decoder = rust::png::Decoder<rust::std::fs::File>
        ::new_(rust::std::fs::File::open(path).unwrap());

  auto header_info = decoder.read_header_info()
                                    .unwrap();
  std::cout << "Width=" << header_info.width() << '\n';
               "\nHeight=" << header_info.height() << '\n';
}
```

```
}

type std::fs::File {
    #layout(size = 8, align = 8);
    fn open(&str) -> std::io::Result<std::fs::File>;
}

type std::io::Result<std::fs::File> {
    #layout(size = 16, align = 8);
    fn unwrap(self) -> std::fs::File;
}

type png::Decoder<std::fs::File> {
    #layout(size = 656, align = 8);
    fn new(std::fs::File) -> png::Decoder<std::fs::File>;
    fn read_header_info(&mut self) ->
        core::result::Result<&png::Info, png::DecodingError>;
}

type core::result::Result<&png::Info,
png::DecodingError> {
```

Adobe

Safe PNG parsing from C++

```
#include <iostream>
#include <generated.h>

int main() {
  auto path = rust::Str::from_char_star("test.png");

  auto decoder = rust::png::Decoder<rust::std::fs::File>
        ::new_(rust::std::fs::File::open(path).unwrap());

  auto header_info = decoder.read_header_info()
                                    .unwrap();
  std::cout << "Width=" << header_info.width() << '\n';
               "\nHeight=" << header_info.height() << '\n';
}
```

```
type std::io::Result<std::fs::File> {
    #layout(size = 16, align = 8);
    fn unwrap(self) -> std::fs::File;
}

type png::Decoder<std::fs::File> {
    #layout(size = 656, align = 8);
    fn new(std::fs::File) -> png::Decoder<std::fs::File>;
    fn read_header_info(&mut self) ->
        core::result::Result<&png::Info, png::DecodingError>;
}

type core::result::Result<&png::Info,
png::DecodingError> {
    #layout(size = 32, align = 8);
    fn unwrap(self) -> &png::Info;
}

type png::Info {
    #layout(size = 432, align = 8);
```

Adobe

Safe PNG parsing from C++

```
#include <iostream>
#include <generated.h>

int main() {
  auto path = rust::Str::from_char_star("test.png");

  auto decoder = rust::png::Decoder<rust::std::fs::File>
        ::new_(rust::std::fs::File::open(path).unwrap());

  auto header_info = decoder.read_header_info()
                                    .unwrap();
  std::cout << "Width=" << header_info.width() << '\n';
             "\nHeight=" << header_info.height() << '\n';
}
```

```
type std::io::Result<std::fs::File> {
    #layout(size = 16, align = 8);
    fn unwrap(self) -> std::fs::File;
}

type png::Decoder<std::fs::File> {
    #layout(size = 656, align = 8);
    fn new(std::fs::File) -> png::Decoder<std::fs::File>;
    fn read_header_info(&mut self) ->
        core::result::Result<&png::Info, png::DecodingError>;
}

type core::result::Result<&png::Info,
png::DecodingError> {
    #layout(size = 32, align = 8);
    fn unwrap(self) -> &png::Info;
}

type png::Info {
    #layout(size = 432, align = 8);
```

Adobe

Safe PNG parsing from C++

```
#include <iostream>
#include <generated.h>

int main() {
  auto path = rust::Str::from_char_star("test.png");

  auto decoder = rust::png::Decoder<rust::std::fs::File>
        ::new_(rust::std::fs::File::open(path).unwrap());

  auto header_info = decoder.read_header_info()
                                  .unwrap();
  std::cout << "Width=" << header_info.width() << '\n';
             "\nHeight=" << header_info.height() << '\n';
}
```

```
type std::io::Result<std::fs::File> {
    #layout(size = 16, align = 8);
    fn unwrap(self) -> std::fs::File;
}

type png::Decoder<std::fs::File> {
    #layout(size = 656, align = 8);
    fn new(std::fs::File) -> png::Decoder<std::fs::File>;
    fn read_header_info(&mut self) ->
        core::result::Result<&png::Info, png::DecodingError>;
}

type core::result::Result<&png::Info,
png::DecodingError> {
    #layout(size = 32, align = 8);
    fn unwrap(self) -> &png::Info;
}

type png::Info {
    #layout(size = 432, align = 8);
```

Adobe

Safe PNG parsing from C++

```
#include <iostream>
#include <generated.h>

int main() {
  auto path = rust::Str::from_char_star("test.png");

  auto decoder = rust::png::Decoder<rust::std::fs::File>
        ::new_(rust::std::fs::File::open(path).unwrap());

  auto header_info = decoder.read_header_info()
                                .unwrap();
  std::cout << "Width=" << header_info.width() << '\n';
               "\nHeight=" << header_info.height() << '\n';
}
```

```
type png::Decoder<std::fs::File> {
    #layout(size = 656, align = 8);
    fn new(std::fs::File) -> png::Decoder<std::fs::File>;
    fn read_header_info(&mut self) ->
        core::result::Result<&png::Info, png::DecodingError>;
}

type core::result::Result<&png::Info,
png::DecodingError> {
    #layout(size = 32, align = 8);
    fn unwrap(self) -> &png::Info;
}

type png::Info {
    #layout(size = 432, align = 8);
    fn width(&self) -> u32;
    fn height(&self) -> u32;
}
```

build.bat

```
zngur generate main.zng
cargo build
clang++ -I. main.cpp target/debug/png_example.lib -lws2_32 -luserenv -lntdll
```

PNG example retrospective

- Safe PNG parsing in C++ took about an hour
- Adding a Rust dependency *was easier than adding a C++ dependency*
- Happy with how the C++ code looks
- Integration into a build system (e.g. CMake) would take more time, but no major obstacles anticipated

Summing it up…

- Many good reasons to adopt Rust
  - Features
  - Demographic shift
  - Safety improvements
- Zngur makes for a great C++/Rust interop experience
  - Strong semantics subset model
  - Ergonomic on both sides
- Door is now open for using Rust libraries in C++

C++ @ Adobe!


Adobe