Extending the Gaudi Software Framework outside of C++

M. Clemencic *on behalf of the LHCb Collaboration*

CHEP 2024, Kraków, PL

CERN - LHCb

Introduction

- C++ is still a very relevant language, but
  - too many footguns, more or less hidden
  - memory safety as an afterthought
    (out of bounds access, invalid references, etc.)
  - maintenance is very costly (compared to other languages)
- Then what else?
  - successor languages: Carbon, Cpp2/cppfront
  - alternatives: Rust, Swift, Go, ...
- No need to rewrite all our code base
  - Google Security showed the benefits of integrating safe code in existing projects
- Can we do it in the Gaudi Software Framework? At what cost?

Table of contents

Rust? Why Rust?

Rust? Why Rust?

I had to choose a language for this exercise

- Rust has been quoted many times as an *alternative* to C++
- Heavy weights are betting on Rust (Google, Microsoft)
- The only alternative to C in Linux kernel
- I was familiar with Rust and the options to bind with C++
- No garbage collector

Note: Carbon and Cpp2/cppfront are interesting, but not ready for production.

Introduction to Rust/C++ interoperability

Interoperability between programming languages

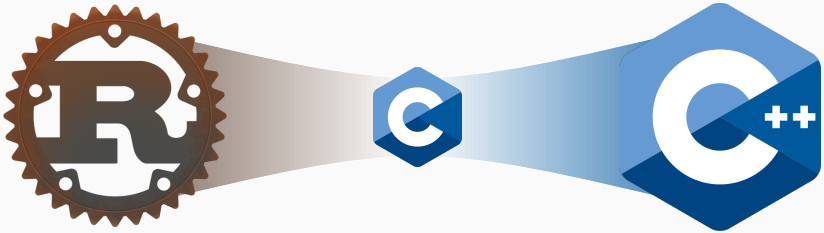
- Direct interoperability between languages is rare
  - C++ and Cpp2, ObjectiveC and Swift, languages based on JVM, ...

Interoperability between programming languages

- Direct interoperability between languages is rare
  - C++ and Cpp2, ObjectiveC and Swift, languages based on JVM, ...
- In all other cases, we use the C calling convention as a bridge

Interoperability between programming languages

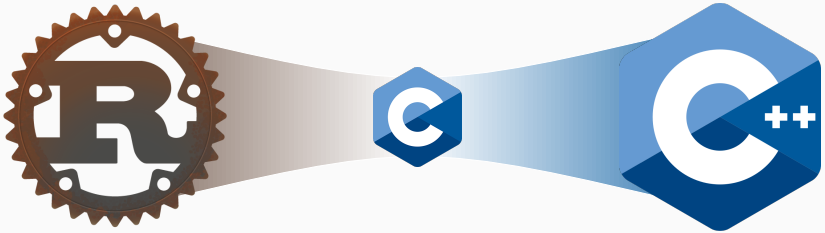
- Direct interoperability between languages is rare
  - C++ and Cpp2, ObjectiveC and Swift, languages based on JVM, ...
- In all other cases, we use the C calling convention as a bridge
  - often referred to as *Hourglass API/Interfaces/Pattern*



Interoperability between programming languages

- Direct interoperability between languages is rare
  - C++ and Cpp2, ObjectiveC and Swift, languages based on JVM, ...
- In all other cases, we use the C calling convention as a bridge
  - often referred to as *Hourglass API/Interfaces/Pattern*

Note:
Foreign Function Interface (FFI) is the set of utilities a language provides to interact with C APIs.



Rust/C++ interoperability

- A few options are available
  - bindgen
    - “automatically generates Rust FFI bindings to C and C++ libraries”
  - cbindgen
    - “creates C/C++11 headers for Rust libraries which expose a public C API”
  - cxx
    - “provides a safe mechanism for calling C++ code from Rust and Rust code from C++”
  - manually written bindings
    - see the FFI chapter in The Rustonomicon

Rust/C++ interoperability

- A few options are available
  - bindgen
    - “automatically generates Rust FFI bindings to C and C++ libraries”
  - cbindgen
    - “creates C/C++11 headers for Rust libraries which expose a public C API”
  - cxx
    - “provides a safe mechanism for calling C++ code from Rust and Rust code from C++”
  - manually written bindings
    - see the FFI chapter in The Rustonomicon
- For this exercise I used `cxx` and a bit of handwritten code

Building Rust code in a C++ project

- Rust has a standard build and package manager: Cargo
  - requires full control of the build
  - but it can produce dynamic or static libraries
- Corrosion "is a tool for integrating Rust into an existing CMake project"
  - great help to wrap the execution of Cargo from CMake
  - partial, but good enough, support for `cxx` bindings

Calling a Rust function from C++

The simplest use case

- You want to use a functionality available as a Rust crate
  - assuming C bindings are not available
- We need to add C bindings for the functions we need

Some simple Rust example: count events

```rust
struct JobStats { events_count: AtomicUsize }

impl JobStats {
    pub fn new() -> Self {
        Self { events_count: AtomicUsize::new(0) }
    }
    pub fn add_event(&self) {
        self.events_count.fetch_add(1, Relaxed);
    }
    pub fn events_count(&self) -> usize {
        self.events_count.load(Relaxed)
    }
}
```

`struct` to hold data

factory fuction

increment counter

get current count

Defining C++ bindings

```
#[cxx::bridge]
mod ffi {
    extern "Rust" {
        type JobStats;
        fn add_event(&self);
        fn events_count(&self) -> usize;

    }
}
```

Functions taking `&self` are methods.
“Static” methods cannot be declared...

Defining C++ bindings

```
#[cxx::bridge]
mod ffi {
    extern "Rust" {
        type JobStats;
        fn add_event(&self);
        fn events_count(&self) -> usize;
        fn new_job_stats() -> Box<JobStats>;
    }
}
fn new_job_stats() -> Box<JobStats> {
    Box::new(JobStats::new())
}
```

Functions taking `&self` are methods.
"Static" methods cannot be declared... so we need to add a factory function.

Gaudi Algorithm

```cpp
class RustyAlg : public Gaudi::Algorithm {
public:
  using Algorithm::Algorithm;
  StatusCode execute( EventContext const& ) const override {
    m_stats->add_event();
    return StatusCode::SUCCESS;
  }
  StatusCode finalize() override {
    info() << "total event count -> " << m_stats->events_count() << endmsg;
    return Algorithm::finalize();
  }
private:
  rust::Box<JobStats> m_stats = new_job_stats();
};
```

Gaudi Algorithm

```
class RustyAlg : public Gaudi::Algorithm {
public:
  using Algorithm::Algorithm;
  StatusCode execute( EventContext const& ) const override {
    m_stats->add_event();
    return StatusCode::SUCCESS;
  }
  StatusCode finalize() override {
    info() << "total event count -> " << m_stats->events_count() << endmsg;
    return Algorithm::finalize();
  }
private:
  rust::Box<JobStats> m_stats = new_job_stats();
};
```

Rust functions

Implementing a Gaudi Service in Rust

A more complex use case

- A Gaudi Service is used by other services and algorithms
  - integrated with Gaudi component model (plugin, properties, ...)
  - transparently switch from C++ to Rust implementation

A more complex use case

- A Gaudi Service is used by other services and algorithms
  - integrated with Gaudi component model (plugin, properties, ...)
  - transparently switch from C++ to Rust implementation
- Requires
  - to register a factory function to Gaudi Plugin Service
  - a C++ facade to channel interface calls to Rust functions

An example of an abstract interface

Just a minimal read-only key-value store

```
struct IKeyValueStore : virtual IInterface {
    // ...
    virtual std::optional<std::string>
        get( std::string_view key ) const = 0;
};
```

Simple enough, but `cxx` does not support `string_view` or `optional`.

The C++ facade

```
class KeyValueStoreBridge:
    public extends<Service, IKeyValueStore> {
public:
  using extends::extends;
  // delegation ot IKeyValueStore methods
  std::optional<std::string>
  get( std::string_view key ) const override {
    return to_optional(
      m_impl->get( rust::Str( key.data(), key.size() ) ) );
  }
private:
  rust::Box<RustKvs> m_impl = new_rust_kvs( *this );
};
```

The C++ facade

```
class KeyValueStoreBridge:
    public extends<Service, IKeyValueStore> {
public:
  using extends::extends;
  // delegation ot IKeyValueStore methods
  std::optional<std::string>
  get( std::string_view key ) const override {
    return to_optional(
      m_impl->get( rust::Str( key.data(), key.size() ) ) );
  }
private:
  rust::Box<RustKvs> m_impl = new_rust_kvs( *this );
};
```

- Easy to map `string_view` → `&str`

The C++ facade

```
class KeyValueStoreBridge:
    public extends<Service, IKeyValueStore> {
public:
  using extends::extends;
  // delegation ot IKeyValueStore methods
  std::optional<std::string>
  get( std::string_view key ) const override {
    return to_optional(
      m_impl->get( rust::Str( key.data(), key.size() ) ) );
  }
private:
  rust::Box<RustKvs> m_impl = new_rust_kvs( *this );
};
```

- Easy to map `string_view` → `&str`
- For `optional` we need a helper (for the moment)

Service implementation

```rust
struct RustKvs {}

impl RustKvs {
    fn get_internal(&self, key: &str) -> Option<String> {
        Some(format!("{0}-{0}", key))
    }

    fn get(&self, key: &str) -> ffi::OptString {
        self.get_internal(key).into()
    }
}
```

The component factory: Rust

```
fn new_rust_kvs() -> Box<RustKvs> { Box::new(RustKvs{}) }
#[cxx::bridge]
mod ffi {
    extern "Rust" {
        type RustKvs;
        fn get(&self, key: &str) -> OptString;
        fn new_rust_kvs() -> Box<RustKvs>;
    }
}
```

The component factory: C++

```cpp
#include <generated_rust_header.h>

// ... KeyValueStoreBridge implementation ...

DECLARE_COMPONENT_WITH_ID(
    KeyValueStoreBridge, "RustKeyValueStore"
)
```

Gaudi Algorithms in Rust

A different kind of problem

- Algorithms require that framework abstractions are shared
  - finite state machine transitions
  - configuration from job options (component properties)
  - access to framework services (e.g. `MessageSvc`, event data)

A different kind of problem

- Algorithms require that framework abstractions are shared
  - finite state machine transitions
  - configuration from job options (component properties)
  - access to framework services (e.g. `MessageSvc`, event data)
- We need to provide
  - bindings to access the framework from Rust
  - wrappers to present Rust code as algorithms
  - bindings for event model classes (experiment specific)
- All must go through C APIs, so we cannot use generics or templates

How it can be solved

- Define a wrapper class
  - hold a pointer to the Rust Algorithm instance
  - forward C++ calls to Rust functions
  - provide helpers to call framework functions from Rust
- Special way to declare factory functions for the plugin service
- Helpers to map event model classes

- Rust facade to C++ wrapper
  - Rust counterparts of C++ helper functions
- Generic object to host callbacks for initialize, finalize, execute, etc.
- Factory functions for algorithms using the *builder pattern*

Algorithm wrapper: C++

```
class AlgWrapper : public Gaudi::Algorithm {
public:
  AlgWrapper( std::string const& name, ISvcLocator* svcLoc,
              WrappedAlg* dyn_alg_ptr );

  StatusCode initialize() override;
  StatusCode execute( EventContext const& ctx ) const override;
  StatusCode finalize() override;
  void addInputHandle( std::string const& typename, ... );
  // ... various helpers ...
private:
  WrappedAlg* m_dyn_alg_ptr;
  // etc.
};
```

Algorithm wrapper: Rust

```rust
#[derive(Clone, Copy)]
pub struct Host<'a> {
    pub alg: &'a ffi::AlgWrapper, // reference to the wrapper instance
}
impl Host<'_> {
    pub fn add_input(&self, type_name: &str, name: &str, location: &str,
                     doc: Option<&str>) {
        let_cxx_string!(name = name);
        let_cxx_string!(type_name = type_name);
        let_cxx_string!(location = location);
        let_cxx_string!(doc = doc.unwrap_or(""));
        self.alg.addInputHandle(&type_name, &name, &location, &doc);
    }
    // ... more helpers ...
}
pub type WrappedAlg = Box<dyn AlgorithmTrait>; // Rust for `unique_ptr<Base>`
```

An Algorithm in Rust

```rust
#[derive(Default)] // struct to host Algorithm state and cache info
struct IntToFloatAlg {}

#[no_mangle] // declare the factory as a C function
extern "C" fn int_to_float_factory() -> *mut WrappedAlg {
    Box::into_raw(Box::new(Box::new(
        gaudi::AlgorithmBuilder::<IntToFloatAlg>::new()
            .add_input("int", "InputLocation", "\"/Event/Int\"", None)
            .add_output("float", "OutputLocation", "\"/Event/Float\"", None)
            .set_execute_action(|data, host, ctx| {
                let value: i32 = *host.get(ctx, "InputLocation");
                host.put(ctx, "OutputLocation", value as f32);
                Ok(())
            })
            .build()
    )))
}
```

Declare the Algorithm factory

```
#include <Gaudi/Rust/AlgWrapper.h>

DECLARE_RUST_ALG( int_to_float_factory, "RustIntToFloatAlg" )
```

It works!

With a few algorithms in C++ and Rust (and some printout):

```
...
ApplicationMgr       INFO Application Manager Initialized successfully
ApplicationMgr       INFO Application Manager Started successfully
IntDataProducer      INFO executing IntDataProducer, storing 7 into /Event/Int
R_IntDataProd        INFO executing R_IntDataProd, storing 10 into /Event/IntR (Rust)
IntToFloatAlg        INFO Converting: 7 from /Event/Int and storing it into /Event/Float
R_IntToFloatAlg      INFO Converting: 10 from /Event/IntR and storing it into /Event/Floa
FloatDataConsumer    INFO executing FloatDataConsumer: 7
R_FloatDataCons...   INFO executing R_FloatDataConsumer: 10 (Rust)
...
```

Conclusions and outlook

Conclusions

- Extending Gaudi with Rust (or another language) is possible
- The main limitation is the need to pass through C bindings
  - we rely a lot on inheritance and templates, which have to be wrapped
- Vocabulary types (`string`, `vector`, etc.) have to be converted
  - some overhead is unavoidable (it can be minimized with some work)
- An experiment event model requires bindings
  - existing C++ event models need work
  - and event model based on Apache Arrow works out of the box

Outlook

- The code showed here is in a private branch of Gaudi
  - planning to propose for integration as an example after some clean up
- There's room for improvement
  - make Service wrapping similar to that of Algorithm
  - reduce boilerplate (e.g. with Rust macros)
- Some areas can be further investigated
  - use `bindgen` for event model classes
  - produce plain Gaudi C bindings for other languages
- What to do next is to be discussed within the Gaudi community
  - keep Rust as an example for interested experiments?
  - interest in Rust as an *official* language in Gaudi?

Backup

Helper for optional: Rust

```rust
#[cxx::bridge]
mod ffi {
    struct OptString { value: String, is_set: bool }
}

impl From<Option<String>> for ffi::OptString {
    fn from(value: Option<String>) -> Self {
        let is_set = value.is_some();
        ffi::OptString {
            value: value.unwrap_or_default(), is_set,
        }
    }
}
```

Helper for optional: C++

```
std::optional<std::string> to_optional( const OptString& opt ) {
  if ( opt.is_set ) {
    return std::string( opt.value );
  } else {
    return std::nullopt;
  }
}
```