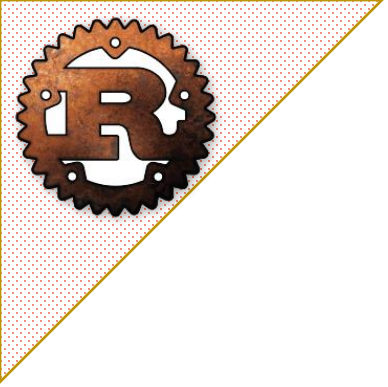
Rust programming

Course – 11

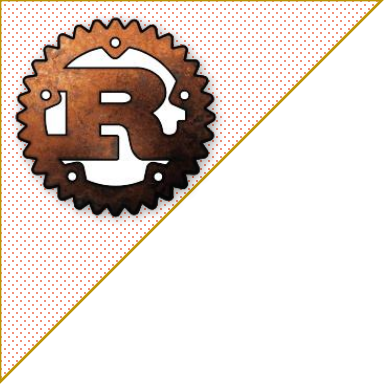
Gavrilut Dragos

Agenda for today

1. Macros
2. Procedural macros
3. Procedural derive macros
4. Function-like procedural macros
5. File operations
6. File System operations
7. Execution Environment
8. FFI



Macros



Macros

Macros are a kind of metaprogramming designed to extend the capabilities of a program in regard to problems like the following ones:

- **DRY** (**D**on't **R**epeat **Y**ourself) → for cases where similar code must be repeated in your program
- **DSL** (**D**omain **S**pecific **L**anguages) → for cases where you need a special syntax within your own code
- **Variadic parameters** or **repetitions** → keep in mind that Rust does not have variadic parameters (like with the case of **…** in C/C++). Instead, it uses macros to achieve similar results.

Rust macros don't work like C/C++ ones (by performing some sort of replacement in the precompile phase). They work directly with the compiled AST and their output is also integrated in the **AST** (***A***bstract ***S***yntax ***T***ree). As such, their parameters have a type (a token type as it is represented by Rust AST).

Macros

A macro in Rust is defined in the following way:

```
macro_rules! name {
    rule1 ;
    rule2 ;
    rule3 ;
    ....
    rulen ;
}
```

where $rule_i$ is defined as ($pattern_matcher) => {$expansion};
where a ***pattern_matcher*** is a sequence of tokens (as they are defined in the Rust AST) including some regular expression like operators such as *, + or ? , and "i" in $rule_i$ is in [1..n]
The process that matches some parameters and converts them into rust code is also called **transcribing**.

This is very similar to how the match construct works (with at least one rule needed to be present).

To use a macro just use the name define in the ***macro_rules*** definition followed by the ! character: **name**!(*parameters*), where ***parameters*** is a sequence of tokens as they are obtained by Rust compiler.

Macros

Let's start with a very simple example where a macro translates to a number (similar to how a simple macro works in C/C++).

Rust

```
macro_rules! MAX_VALUE {
    () => { 5 };
}
fn main() {
    let x = MAX_VALUE!();
    let y = MAX_VALUE![];
    let z = MAX_VALUE!{};
    println!("{x},{y},{z}");
}
```

Output

```
5,5,5
```

C/C++

```
#define MAX_VALUE 5

void main() {
    auto x = MAX_VALUE;
    auto y = MAX_VALUE;
    auto z = MAX_VALUE;
    printf("%d,%d,%d",x,y,z);
}
```

OBS: *Notice that you can use [] or () or {} with a macro in Rust, as specified in the macro rules.*

Macros

Not using *[]* *or* *()* *or* *{}* with a macro in Rust will trigger a compiler error. Similar, you have to add the ! sign after the macro in order for the syntax to be correct.

Rust

```
macro_rules! MAX_VALUE {
    () => { 5 };
}
fn main() {
    let x = MAX_VALUE!;
    let y = MAX_VALUE();
    let z = MAX_VALUE!();
    println!("{x},{y},{z}");
}
```

Error

```
error: expected one of `(`, `[`, or `{`, found `;`
 --> src\main.rs:5:23
  |
5 |     let x = MAX_VALUE!;
  |                       ^ expected one of `(`, `[`, or `{`
```

Macros

Rust however adds more context to a macro (in the sense that any variable that is being used must exists in the macro context). Let's analyze the following scenario:

Rust

```
macro_rules! PRINT_INDEX {
    () => { println!("{}",i); };
}
fn main() {
    for i in 0..5 {
        PRINT_INDEX!();
    }
}
```

C/C++

```
#define PRINT_INDEX printf("%d\n",i);

void main() {
    for (auto i = 0;i<5; i++) {
        PRINT_INDEX;
    }
}
```

Error

```
error[E0425]: cannot find value `i` in this scope
 --> src\main.rs:2:27
  |
2 |     () => { println!("{}",i); };
  |                           ^ not found in this scope
...
6 |         PRINT_INDEX!();
  |         -------------- in this macro invocation
```

Output

```
0
1
2
3
4
```

Macros

Rust however adds more context to a macro (in the sense that any variable that is being used must exists in the macro context). Let's analyze the following scenario:

Rust

```
macro_rules! PRINT_INDEX {
    () => { println!("{}",i); };
}
fn main() {
    for i in 0..5 {
        PRINT_INDEX!();
    }
}
```

C/C++

```
#define PRINT_INDEX printf("%d\n",i);

void main() {
    for (auto i = 0;i<5; i++) {
        PRINT_INDEX;
    }
}
```

Err

```
error[E0425]: cannot find value `i` in this scope
 --> src\main.rs:2:27
  |
2 |     () => { println!("{}",i); };
  |                           ^ not found in this scope
...
6 |         PRINT_INDEX!();
  |         ---------------
```

```
0
1
2
3
4
```

This behavior is related to how each language processes a macro:

- **C/C++** replaces the macro before AST is being created (and as such if there is a variable "I" that exists within the execution context → this will suffice)
- **Rust** compiles the macro into its own AST and then ads it to the program AST

The main advantage of this technique is that it limits the usage of a macro to a specific context, and as such less unwanted errors due to misuse of a macro. This is referred by Rust as **macro hygiene**.

Macros

Let's see an even more complex example where we use macro parameters. In Rust, each parameter is defined using the $ (*dollar sign*) followed by a name, and a type (with the mention that the type refers to how that variable is referred to by the AST.

Format: $name:type where ***type*** can be one of the following:

Type	Observations
expr	An expression in the AST
ident	An identifier (including Rust specific keywords)
block	A block of either statements and/or expression surrounded by {..} (braces)
item	A function, struct , module, …
literal	A constant value (e.g. 100, "some string", true, 'a')
tt	A single token tree
ty	A type
stmt	A statement

Type	Observations
pat	A pattern
pat_params	A pattern (backwards compatibility with 2018 edition)
lifetime	A lifetime
path	A path to a module (e.g. mod1::mod2::…)
vis	A visibility keyword (e.g. pub)
meta	A meta item #[…]

Macros

Each one of these constructs ($name:type) can be followed by:

Type	Can be followed by
expr	=> , ;
ident	No restrictions
block	No restrictions
item	No restrictions
literal	No restrictions
tt	No restrictions
ty	=> , = | ; : > >> [{ as where
stmt	=> , ;
pat	=> , = if in
path	=> , = | ; : > >> [{ as where
meta	No restrictions

Macros

Let see how a function like macro that adds two number looks like:

Rust

```
macro_rules! sum {
    ($p1: expr , $p2: expr) => {
        $p1 + $p2
    };
}
fn main() {
    let x = sum!(1 , 2);
    println!("{}",x);
}
```

Output
3

C/C++

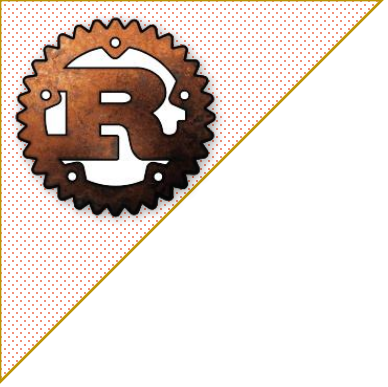
```
#define SUM(x,y) x+y

void main() {
    auto x = SUM(1,2);
    printf("%d\n",x);
}
```

Output
3

So … our macro sum has two parameters:

- $p1 → of type expr (meaning it can be any type of expression)
- $p2 → of type expr

But … why do we actually need to explain each parameter (metavariable) type ?



Macros

Let’s consider the following macro:

Rust

```
macro_rules! multiply {
    ($p1: expr , $p2: expr) => {
        $p1 * $p2
    };
}
fn main() {
    let x = multiply!(10 , 20);
    let y = multiply!(1.2 , 1.5);
    let z = multiply!(1+2 , 3+1);
    println!("{x},{y},{z}");
}
```

Output

```
200,1.7999999999999998,12
```

C/C++

```
#define MULTIPLY(x,y) x*y

void main() {
    auto x = MULTIPLY(10,20);
    auto y = MULTIPLY(1.2,1.5);
    auto z = MULTIPLY(1+2,3+1);
    printf("%d,%d,%d",x,y,z);
}
```

Output

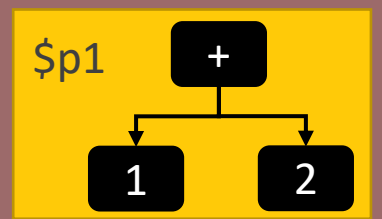
```
200,1.7999999999999998,8
```

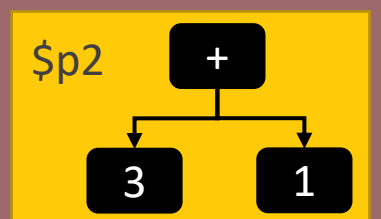
Why do we have this difference ?

Macros

Let’s consider the following macro:

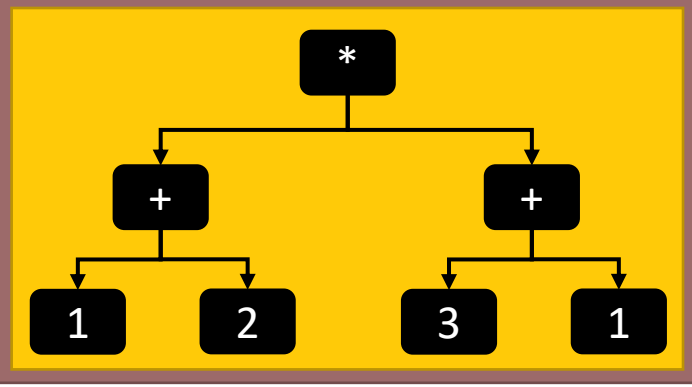
Rust sees the $p1 and $p2 as expressions (meaning they are already parsed and have a tree-like format):






```
let x = multiply!(10 , 20);
```

The output of the macro consists in modifying the AST directly → as such, the macro will translate into something the looks like this:



C/C++

```
#define MULTIPLY(x,y) x*y

void main() {
    auto x = MULTIPLY(10,20);
    auto y = MULTIPLY(1.2,1.5);
    auto z = MULTIPLY(1+2,3+1);
    printf("%d,%d,%d",x,y,z);
}
```

Output

```
200,1.7999999999999998,8
```

Macros

Let’s consider the following macro:

Rust

```
macro_rules! multiply {
    ($p1: expr , $p2: expr) => {
        $p1 * $p2
    };
}
fn main() {
    let x = multiply!(10 , 20);
    let y = multiply!(1.2 , 1.5);
    let z = multiply!(1+2 , 3+1);
    println!("{x},{y},{z}");
}
```

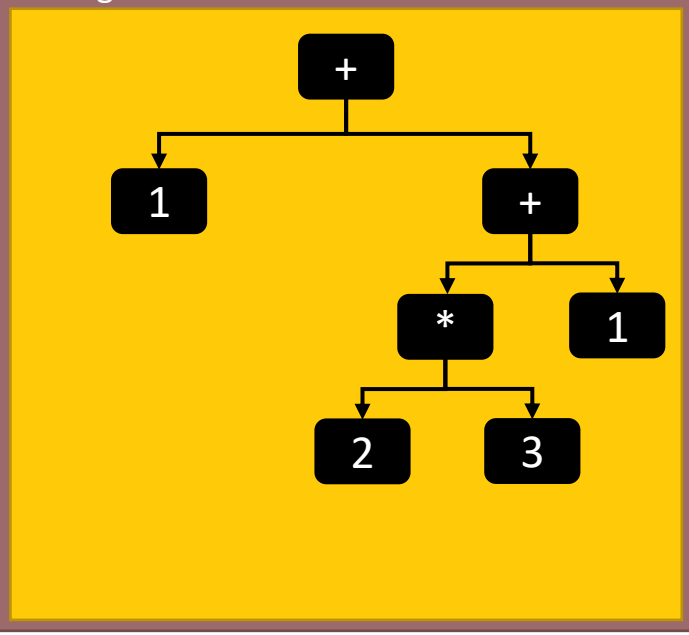
Output

```
200,1.7999999999999998,12
```

C++ just makes a simple replacement (meaning that the statement: `auto z = MULTIPLY(1+2,3+1);` will translate into `auto z = 1+2*3+1;`

```
#define MULTIPLY(x,y) x*y
```

The AST will be built after this phase, and will look like the following:



Macros

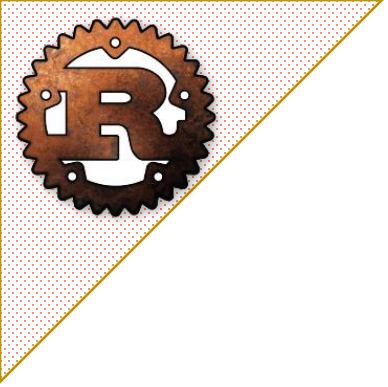
Let’s see a slightly more complex example:

Rust

```rust
trait PrintMe { fn print_me(&self); }

macro_rules! implement_print_me {
    ($t: ty) => {
        impl PrintMe for $t {
            fn print_me(&self) {
                println!("{:?}",self);
            }
        }
    };
}
implement_print_me!(i32);
implement_print_me!(bool);

fn main() {
    5.print_me();
    true.print_me();
}
```

Output

```
5
true
```



Macros

Let’s see a slightly more complex example:

Rust

```
macro_rules! eval {
    ($b: block) => {
        let x = $b;
        println!("{x}");
    };
}
fn main() {
    eval!({1+3+4});
    eval!({
        let mut sum = 0;
        for i in 0..10 {
            sum+=i;
        }
        sum
    });
}
```

Output
8
45

Notice that we call eval! macro with the following format:
eval! (**{…}**)
This is required as the parameter b is of type ***block*** !

Macros

Let’s see another example, but this time we will use multiple rules to create a simple calculator that gets a command followed by a two values and then applies that command over the two numbers/expressions.

Rust

```rust
macro_rules! eval {
    (add $p1: expr, $p2: expr) => { println!("{}",$p1+$p2); };
    (mul $p1: expr, $p2: expr) => { println!("{}",$p1*$p2); };
    (sub $p1: expr, $p2: expr) => { println!("{}",$p1-$p2); };
    (div $p1: expr, $p2: expr) => { println!("{}",$p1/$p2); };
}
fn main() {
    eval!(add 2,3);
    eval!(sub 10,20);
    eval!(mul 5,4);
    eval!(div 100,3);
}
```

Output

```
5
-10
20
33
```

Macros

But → what if we want to have a variable number of expression (instead of two) for the previous example: e.g. to be able to write something like: `eval!(add 1,2,3);`

To create a loop in a Rust macro use the following format:

```
$ ( … ) separator rep_operator
```

Where:

- (…) is a list of tokens, parameters that need to be repeated
- separator is an optional parameter that describes a separator between (…)
- rep_operator is a parameter that explains how repetition is being performed:
  - * → 0 or multiple repetitions
  - + → 1 or multiple repetitions
  - ? → at most one repetition

Macros

Let’s see an example that uses repetition:

Rust

```
macro_rules! eval {
    (add $p1: expr, $($p2: expr),+) => {
        let mut result = $p1;
        $(
            result += $p2;
        )+
        println!("{}", result);
    };
}
fn main() {
    eval!(add 1,2,3);
    eval!(add 1,2,3,4,5,6,7,8,8,10);
    eval!(add 1,2);
}
```

Output
6
54
3

Let’s analyze this macro :

Macros

Let’s see an example that uses repetition:

Rust

```
macro_rules! eval {
    (add $p1: expr, $($p2: expr),+) => {
        let mut result = $p1;
        $(
            result
        )+
        println!("
    };
}
fn main() {
    eval!(add 1,2
    eval!(add 1,2
    eval!(add 1,2
}
```

The pattern that we are searching for is the following:

1. Text “add”
2. An expression ($p1)
3. A repeating sequence $(…),+ that has to be repeated at least one time (notice the + sign at the end). Between repetition, a comma must be present ,
    1. The repeating sequence is form out of another expression ($p2)

Output

```
6
54
3
```

Macros

Let’s see an example that uses repetition:

Rust

```
macro_rules! eval {
    (add $p1: expr, $($p2: expr),+) => {
        let mut result = $p1;
        $(
            result += $p2;
        )+
        println!("{}", result);
    };
}
fn main() {
    eval!(add 1,2,3);
    eval!(add 1,2,3,4,5,6,7,8,8,10);
    eval!(add 1,2);
}
```

The code part of this macro is form out of 3 components:

1. First the creation of the mutable variable result that has the value of $p1
2. Second a loop that corresponds to the loop from the parameter list that adds each value of $p2 from the loop to the result variable created on step 1
3. A println!(…) that prints the result.

Macros

Let’s see an example that uses repetition:

Rust

```
macro_rules! eval {
    (add $p1: expr, $($p2: expr),+) => {
        let mut result = $p1;
        $(
            result += $p2;
        )+
        result
    };
    (add $p1: expr) => {
        $p1
    };
}
fn main() {
    eval!(add 1,2,3);
    eval!(add 1,2,3,4,5,6,7,8,8,10);
    eval!(add 1,2);
}
```

To obtain the translation, just click on the eval! from Visual Studio Code, open Command Pallete (Ctrl+Shift+P) and search for “rust-analyzer: Expand macro recursively”

Rust (translated)

```
let mut result = 1;
result += 2;
result += 3;
result += 4;
result += 5;
result += 6;
result += 7;
result += 8;
result += 8;
result += 10;
{
    $crate::io::_print($crate::fmt::Arguments::new_v1(
        &[],
        &[$crate::fmt::ArgumentV1::new(
            &(result),
            $crate::fmt::Display::fmt,
        )],
    ));
};
```

$crate is a special macro that allows expantion in the following way:

- If the call to a function is within the same crate where the function was defined, it expands to nothing
- Otherwise, it expands to the name of the crate.

Macros

A macro can call another macro (including the same macro thus creating a recursive call). There are some limitations on how many such calls the compiler will do, but they can be overwritten by the use of `#![recursion_limit = "256"]` (with different values).

Rust

```rust
macro_rules! count {
    () => {0usize};
    ($first:tt) => {1usize};
    ($first:tt,$($tail:tt),*)=> {
        1usize + count!($($tail),*)
    }
}
fn main() {
    let x = count!(1,2,3,4,5);
    println!("{x}");
}
```

Output

```
5
```

Macros

Rust also has several system macros that can be used in different scenarios:

Macro	Usage
print, println, write, writeln	String formatting
stringify	Converts argument(s) to a string
concat	Concatenates multiple strings/literals into a static string
line, column, file	Information about current file, current line and current column
compile_error	Triggers a compiler error
include, include_str, include_bytes	Allows loading an external file directly into the code as an expression, as an UTF-8 string or as a sequence of bytes
env	A macro to access environment variables
unimplemented	A macro to trigger a panic if unimplemented code is reached
assert, assert_eq, assert_ne, debug_assert, debug_assert_eq,…	Used in tests function to evaluate an expression. These macros will panic if the evaluation is false. The difference between the debug_xxx version and the rest of the macros is that debug_xxx are removed from release build.

Macros

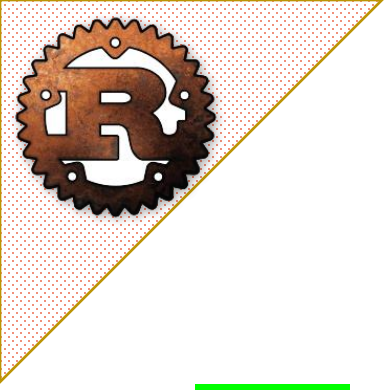
- stringify! Has singular tokens as parameters → meaning it can be used with everything (including keywords, items, etc). It also adds an extra space after a token (except for comma). **OBS**: *Its output might change in future versions*
- concat! has literals as parameters (so it can be used with numbers, strings, etc)

Rust

```rust
fn main() {
    let x = stringify!(1,2,3,4,5);
    let y = stringify!(1+2+3+4+5);
    let z = concat!("abc", 10, true, 1.54);
    let t = stringify!(if a>b then a=a-b else a=a+b);
    println!("{x}");
    println!("{y}");
    println!("{z}");
    println!("{t}");
}
```

Output

```
1, 2, 3, 4, 5
1 + 2 + 3 + 4 + 5
abc10true1.54
if a > b then a = a - b else a = a + b
```



Macros

line()! , column()! and file()! return literals. As such, they can be used with concat()! to get a log-like string:

Rust

```rust
fn main() {
    let x = concat!("We are at line ",line!()," in file ", file!());
    println!("{x}");
}
```

Output

```
We are at line 2 in file src\main.rs
```

Similarly, env(<system_variable>)! can be used to read the value of a system variable:

Rust

```rust
fn main() {
    let x = env!("PATH");
    println!("{x}");
}
```

Output

```
C:\Program Files\.....
```

Macros

compile_error()! can be used to trigger an error if some parameters are incorrect (in our case we can not sum up 0 or 1 values).

Rust

```
macro_rules! sum {
    () => { compile_error!("Expecting at least two values to sum up"); };
    ($one:expr) => { compile_error!("Expecting at least two values to sum up, only one was provided"); };
    ($first:expr, $($the_rest:expr),+) => {
        {
            let mut result = $first;
            $(
                result+=$the_rest;
            )+
            result
        }
    };
}
fn main() {
    let x = sum!(1,2,3,4,5);
    println!("{x}");
}
```

Output

```
10
```

Macros

compile_error()! can be used to trigger an error if some parameters are incorrect (in our case we can not sum up 0 or 1 values).

Rust

```
macro_rules! sum {
    () => { compile_error!("Expecting at least two values to sum up"); };
    ($one:expr) => { compile_error!("Expecting at least two values to sum up, only one was provided"); };
    ($first:expr, $($the_rest:expr),+) => {
        {
            let mut result = $first;
            $(
                result += $the_rest;
            )+
            result
        }
    };
}
fn main() {
    let x = sum!(1);
    println!("{x}");
}
```

Error

```
error: Expecting at least two values to sum up, only one was provided
  --> src\main.rs:3:22
   |
3  |     ($one:expr) => { compile_error!("Expecting at least two values to sum up, only one was provided"); };
   |                      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
...
15 |     let x = sum!(1);
   |             ------- in this macro invocation
   |
```

Macros

Let’s consider that we have a C++ file (called temp.cpp) that contains the following code and it is located in the src file where the main.rs is located as well.

C++

```
#define MULTIPLY(x,y) x*y
void main() {
    auto x = MULTIPLY(1,2);
    auto y = MULTIPLY(1.2,1.5);
    printf("%d,%d",x,y);
}
```

Then , the execution of the following code in Rust will produce the following output:

Rust

```
fn main() {
    let x = include_str!("temp.cpp");
    println!("{x}");
}
```

Output

```
#define MULTIPLY(x,y) x*y
void main() {
   auto x = MULTIPLY(1,2);
   auto y = MULTIPLY(1.2,1.5);
   printf("%d,%d",x,y);
}
```

Procedural macros

Procedural macros

There are cases where a simple macro will not suffice , as it implies a more complex logic on the transformation that needs to be made before compiling the code (e.g. for example converting a code written in a different language in a Rust language).

The general approach here is to use procedural macros.

A procedural macro can be used in the following way

```
#[name]
```

or

```
#[name(arguments)]
```

Where:

- `name` ➔ the name of the procedural macro
- *`arguments`* ➔ arguments that are being passed to procedural macro (optional)

Procedural macros

To create a procedural macro, the following steps must be performed:

1. Create a separate library where the procedural macro will be defined.
2. In that library define an export with the name of the procedural macro
3. Modify cargo.toml to explain that the new library is to be used by the compiler as a procedural macro
4. Compile the new library and linked it to a new project (where you want to use it) via dependencies from cargo.toml (you can either uploaded to ***crates.io*** , or use it directly from the ***local folder*** or a ***git repo***).
5. Rust compiler will load the library and whenever the procedural macro is being called it will execute your exported function with a list of tokens from the code (not an AST – but sufficient to build one) that needs to be translated into Rust code.

Procedural macros

Let’s discuss an actual example. Rust does not have the concept of a bit flag enum (an **enum** where each variant is a flag: 1,2,4,8,16,32,64, …).

The reason for this is that bitflags can be combined and since an enum must have a value from the specified list , this can be tricky to evaluate.

For example – lets consider the following snippet:

Rust

```rust
enum Test {
    V1 = 1,
    V2 = 2,
    V3 = 4
}
fn main() {
    let t = Test::V1 | Test::V2;
}
```

Error

```
error[E0369]: no implementation for `Test | Test`
   --> src\main.rs:90:22
    |
90  |     let t = Test::V1 | Test::V2;
    |             -------- ^ -------- Test
    |             |
    |             Test
    |
note: an implementation of `BitOr<_>` might be missing for `Test`
```

The code will not work as BitOr trait is not implemented !

Procedural macros

But this is not enough → to have a proper bitflag **enum** the following operations should exist:

Requirement	Example/Observations
BitOr	Test::V1 | Test::V2 | Test::V3
BitAnd	Test::V1 & Test::V2
BitOrAssign	flags |= Test::V1;
BitAndAssign	flags &= Test::V1;
PartialEq	The possibility to compare two values and see if they are equal or not
Display	If we want to print an enum value
Empty variant	The case where no bits are set up (the memory value is 0)
Representation	A bit-flag enum is ultimately an unsigned value of type u8,u16,u32,u64 or u128.
Methods	Various method (e.g. contains, clear, set, ….)

Procedural macros

Obviously, all of the precedent requirements can be written manually. But this implies a lot of work for a very simple enum.

In particular, if we are to compare this with other languages (like C/C++ where this kind of feature is already there), having to write the same code over and over for each bit flag enum can be frustrating.

So … let’s see how we can build a procedural macro that can do this for us (automatically).

Procedural macros

Step 1: Create a separate library where the procedural macro will be defined.

- let's call the new library EnumBitFlags
- To create it run the following command : cargo new EnumBitFlags --lib

The procedural macro will also be called ***EnumBitFlags***, and ultimately, we will need to be able to write something like this:

Rust

```rust
#[EnumBitFlags]
enum Test {
    V1 = 1,
    V2 = 2,
    V3 = 4
}
```

Procedural macros

Step 2: Modify lib.rs to export this function

Rust (lib.rs)

```rust
use proc_macro::*;

extern crate proc_macro;

#[allow(non_snake_case)]
#[proc_macro_attribute]
pub fn EnumBitFlags(args: TokenStream, input: TokenStream) -> TokenStream {
    …
}
```

Notice the `#[proc_macro_attribute]` attribute and the pub visibility specifier (this tells Rust that this function should be called whenever something like #[EnumBitFlags] is found in another Rust program.

Procedural macros

Step 2: Modify lib.rs to export this function

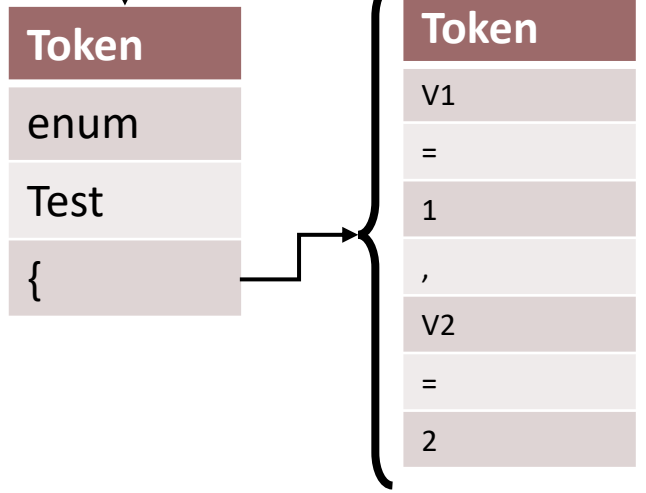
```rust
pub fn EnumBitFlags(args: TokenStream, input: TokenStream) -> TokenStream {…}
```

Assuming we have the following code:

Rust

```rust
#[EnumBitFlags(bits=16,debug=true)]
enum Test {
    V1 = 1,
    V2 = 2
}
```

This part will be sent as the **input** parameter for the ***EnumBitFlags*** method. In reality this is seen more like a tree (AST). In this case it will receive the following



Procedural macros

Step 2: Modify lib.rs to export this function

```
pub fn EnumBitFlags(args: TokenStream, input: TokenStream) -> TokenStream {…}
```

Assuming we have the following code:

Rust

```
#[EnumBitFlags(bits=16,debug=true)]
enum Test {
    V1 = 1,
    V2 = 2
}
```

This part will be sent as the **args** parameter for the ***EnumBitFlags*** method. This is useful if you want to control the way the code is being generated.

Token
Bits
=
16
,
debug
=
true

Procedural macros

Step 2: Modify lib.rs to export this function

```rust
pub fn EnumBitFlags(args: TokenStream, input: TokenStream) -> TokenStream {…}
```

This part represents the output stream (the new code that will be inserted instead of the old one).

Assuming we have the following code:

Rust

```rust
#[EnumBitFlags(bits=16,debug=true)]
enum Test {
    V1 = 1,
    V2 = 2
}
```

For complex code changes, usually the new code is being built in a **String** variable and then converted into a TokenStream using `TokenStream::from_str` method

Procedural macros

Step 2: Modify lib.rs to export this function

Let's see a very simple example:

Rust (lib.rs)

```rust
use proc_macro::*;
extern crate proc_macro;

#[allow(non_snake_case)]
#[proc_macro_attribute]
pub fn MyProcMacro(args: TokenStream, input: TokenStream) -> TokenStream {
    let s = "fn sum(x:i32,y:i32)->i32 { x+y }";
    return TokenStream::from_str(s).expect("Failed to parse parse Rust code");
}
```

In this code we replace the existing code with a function that sums up two integer variables.

OBS: *If an error occurs during processing the token stream from **input** or **args**, a panic! must be thrown. This will be intercepted by the compiler and used to show the error (e.g. very useful in IDEs like Visual Studio Code)*

Procedural macros

Step 2: Modify lib.rs to export this function

Let's see a very simple example:

Rust (lib.rs)

```rust
use proc_macro::*;
extern crate proc_macro;

#[allow(non_snake_case)]
#[proc_macro_attribute]
pub fn MyProcMacro(args: TokenStream, input: TokenStream) -> TokenStream {
    let s = "fn sum(x:i32,y:i32)->i32 { x+y }";
    return TokenStream::from_str(s).expect("Failed to parse parse Rust code");
}
```

While a stream of tokens can be process programmatically, it implies a lot of code to be written. A common practice is to use two crates (syn and quote) that simplify AST processing and code generation

In ms up two integer va

OBS: *If an error occurs during processing the token stream from **input** or **args**, you should panic!. This will be intercepted by the compiler and used to should the error.*

Procedural macros

Step 3: Modify cargo.toml to explain that the new library should be used as a procedural macro.

Cargo.toml

```
[package]
name = "EnumBitFlags"
version = "1.0.7"
edition = "2021"
authors = ["..."]
description = "EnumBitFlags is an implementation of flags support for enums"
license = "MIT"
keywords = […]
categoies = ["development-tools::procedural-macro-helpers"]
repository = "https://github.com/gdt050579/EnumBitFlags/"
readme = "README.md"

[lib]
proc-macro = true
```

Make sure that you specify the name of the library !

Add a new section [lib] and specify that the key "proc-macro" has the value "**true**"

Procedural macros

Step 4: Publish the new library (or store it locally or into a git repository).

In particular for our example, EnumBitFlags was published to crates.io:

- Git repo: https://github.com/gdt050579/EnumBitFlags/
- Crates: https://crates.io/crates/EnumBitFlags

Procedural macros

To use proc-macro, add the following in your dependencies section from cargo.toml:

Cargo.toml

```
[dependencies]
EnumBitFlags = "1.0.9"
```

Then, you can write the following in your Rust programs:

Rust

```
use EnumBitFlags::EnumBitFlags;

#[EnumBitFlags]
enum Test {
    V1 = 1,
    V2 = 2,
    V3 = 4
}
```

Procedural macros

Upon compilation, the following code will be converted into:

Rust

```rust
#[EnumBitFlags(bits=16]
pub enum Test_16bit {
    V1 = 1,
    V2 = 2,
    V3 = 4,
    V4 = 0x8000
}
```

Rust (lib.rs)

```rust
#[derive(Copy, Clone, Debug)]
pub struct Test_16Bits { value: u16 }
impl Test_16Bits {
    pub const V1: Test_16Bits = Test_16Bits { value: 0x1u16 };
    pub const V2: Test_16Bits = Test_16Bits { value: 0x2u16 };
    pub const V3: Test_16Bits = Test_16Bits { value: 0x4u16 };
    pub const V4: Test_16Bits = Test_16Bits { value: 0x8000u16 };
    pub const None: Test_16Bits = Test_16Bits { value: 0 };

    pub fn contains(&self, obj: Test_16Bits) -> bool {…}
    pub fn contains_one(&self, obj: Test_16Bits) -> bool {…}
    pub fn is_empty(&self) -> bool {…}
    pub fn clear(&mut self) {…}
    pub fn remove(&mut self, obj: Test_16Bits) {…}
    pub fn set(&mut self, obj: Test_16Bits) {…}
    pub fn get_value(&self) -> u16 {…}
}
impl std::ops::BitOr for Test_16Bits {…}
impl std::ops::BitOrAssign for Test_16Bits {…}
impl std::ops::BitAnd for Test_16Bits {…}
impl std::ops::BitAndAssign for Test_16Bits {…}
impl std::cmp::PartialEq for Test_16Bits {…}
impl std::default::Default for Test_16Bits {…}
impl std::fmt::Display for Test_16Bits {…}
```

Procedural macros

Once this is done, you can write a code like this:

Rust

```rust
#[EnumBitFlags(bits : 8)]
enum Test {
    V1 = 1,
    V2 = 2,
    V3 = 128,
}

#[test]
fn test_bit_or_assign()
{
    let mut t = Test::V1;
    t |= Test::V1;
    t |= Test::V2;
    assert!(t.contains(Test::V1));
    assert!(t.contains(Test::V2));
    assert!(t.contains(Test::V3)==false);
    t = Test::None;
    assert!(t.is_empty()==true);
}
```

Other more complex ProcMacros

- AppCUI: https://github.com/gdt050579/AppCUI-rs

Derive procedural macros

Derive procedural macros

Derive procedural macros are very similar to procedural macros, except that they provide additional input for the `#[derive(…)]` attribute.

Just like any procedural macro, you need to create a separate library and mark it as a procedural macro from ***cargo.toml*** file.

As a general rule, a derive procedural macro will implement a trait and its additional method for a structure or enum. It is a common practice that larger application create a library that exposes a trait and another library that allows deriving out of that trait via `#[derive(…)]` attribute.

Derive procedural macros

The difference (from how a standard proc-macro looks) lies in the how the entry point function looks like (notice that there is no arguments in this case → as it is not required by the way #[derive(name)] works:

Rust (lib.rs)

```rust
use proc_macro::*;
extern crate proc_macro;

#[proc_macro_derive(<DeriveName>)]
pub fn name_of_derive_function(input: TokenStream) -> TokenStream {
    …
}
```

The usage will be something like this:

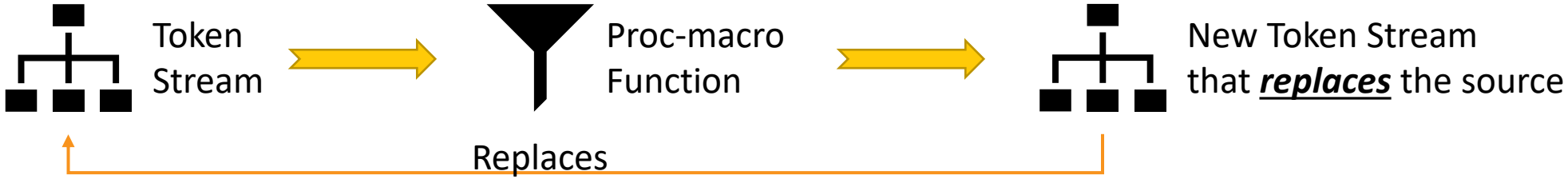
Rust

```rust
#[derive(<DeriveName>)]
struct MyData {
    ...
}
```

Derive procedural macros

Another difference between a proc-macro and a derive proc-macro is how the output is being used.

A) In case of proc-macro



B) In case of derive proc-macro

Derive procedural macros

Another observation is that a derive procedural macros don't have any way to customize them (keep in mind that deriving is done via #[derive(*<name>*)] with not specific customizations.

However, there might be cases where such a customization is required. For such cases, Rust has a helper attributes system, with the following format:

```
#[proc_macro_derive(<DeriveName>,attributes(Attr1,Attr2,…))]
pub fn name_of_derive_function(input: TokenStream) -> TokenStream { … }
```

To use this helper attributes: #[$Attr_i$] or #[$Attr_i$=value]

These attributes (if used) will be present in the input parameter from the macro derive function and can be used to change how the deriving will be implemented.

Derive procedural macros

Let’s see an example:

A) At the library (crate) source

Rust (lib.rs)

```rust
#[proc_macro_derive(Convertor,attributes(ConvertToValue))]
pub fn convertor_function(input: TokenStream) -> TokenStream { … }
```

B) At another rust source that uses this custom derive

Rust

```rust
#[derive(Convertor)]
#[ConvertToValue = "Some information related to convertion"]
struct MyData {
    ...
}
```

Function-like procedural macros

Function-like procedural macros

Function-like procedural macros are procedural macros, that can be called like a normal macro. However, they work just like a procedural macro → they receive a stream of tokens and must return an output (a new code that will be inserted in the program AST).

Just like any procedural macro, you need to create a separate library and mark it as a procedural macro from ***cargo.toml*** file.

You will also need to mark the function that will be called with a specific attribute:

```
#[proc_macro]
```

Function-like procedural macros

Let’s see an example:

A) The library should have something like this:

Rust (lib.rs)

```rust
use proc_macro::*;
extern crate proc_macro;

#[proc_macro]
pub fn my_macro(input: TokenStream) -> TokenStream {
    …
}
```

B) The application that uses this will use it like this:

Rust

```rust
use <library>::my_macro;

my_macro!(<tokens>);
```

Function-like procedural macros

This type of macros are very suited for different type of optimizations and automatizations:

- Computing best hashes
- Generating code from another language (maybe something that is received as a string parameter)
- Generating tables with data (e.g. an automaton)
- Generating code based on the data from a file or multiple files
- Etc.

File operations

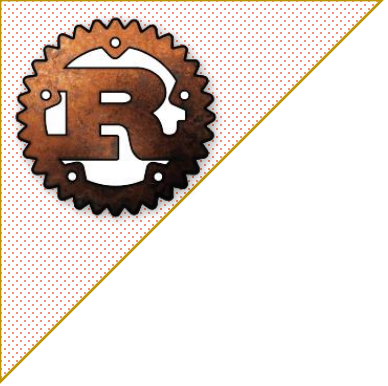
File operations

Rust supports file operations via a specific type called File (located in std::fs::File).

A file object can be created in the following way:

Method	Usage
`fn create (path: &Path) -> Result<File, Error>`	Creates a new file. If the file already exists, the file will be truncated.
`fn open (path: &Path) -> Result<File, Error>`	Opens a file that already exists. If the file is missing or in case of other system errors, it returns Error.
`fn options() -> OpenOptions`	Returns an OpenOption object that can be used to decide how a file should be opened.

Notice that the Error result is part of the namespace (std::io::Error).



File operations

Let’s see a simple example:

Rust

```rust
fn main() -> Result<(), std::io::Error> {
    let mut f = File::create("a.txt")?;
    f.write("Hello world".as_bytes())?;
    println!("File a.txt has been created !");
    Ok(())
}
```

Output

```
File a.txt has been created !
```

Let’s analyze this code:

File operations

Let’s see a simple example:

Rust

```
fn main() -> Result<(), std::io::Error> {
    let mut f = File::create("a.txt")?;
    f.write("Hello world".as_bytes())?;
    println!("File a.txt has been created !");
    Ok(())
}
```

Output

```
File a.txt has been created !
```

The return value from function main is a Result, where:

- **Ok** is represented by void ‘()’ type
- **Err** is represented by std::io::Error type

Let’s analyze this code:

File operations

Let’s see a simple example:

Rust

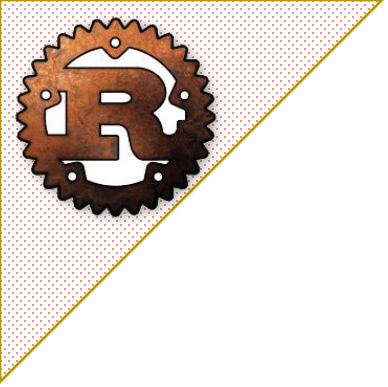
```rust
fn main() -> Result<(), std::io::Error> {
    let mut f = File::create("a.txt")?;
    f.write("Hello world".as_bytes())?;
    println!("File a.txt has been created !");
    Ok(())
}
```

Output

```
File a.txt has been created !
```

This translates in the following way. If File::create(…) returns an error, return the same error and terminate function main. If File::create(…) manages to open a file, move the file object into variable “f”.

Let’s analyze this code:



File operations

Let’s see a simple example:

Rust

```rust
fn main() -> Result<(), std::io::Error> {
    let mut f = File::create("a.txt")?;
    f.write("Hello world".as_bytes())?;
    println!("File a.txt has been created !");
    Ok(())
}
```

Output

```
File a.txt has been created !
```

If we reach this point, we manage to create a file and write a text into that file, so we can return Ok(…) from the main.

Let’s analyze this code:

File operations

If you don’t want to used the “?” operator, the previous code can be written in a different way:

Rust

```rust
use std::{fs::File, io::Write};

fn main() {
    if let Ok(mut f) = File::create("a.txt") {
        if let Ok(_) = f.write("Hello world".as_bytes()) {
            println!("File a.txt has been created !");
        }
    }
}
```

Output

```
File a.txt has been created !
```

File operations

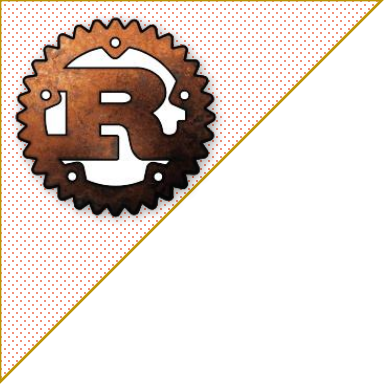
Or … in a more classical way, by checking the results each time:

Rust

```rust
use std::{fs::File, io::Write};

fn main() {
    let f_res = File::create("a.txt");
    if f_res.is_err() {
        return;
    }
    let mut f = f_res.unwrap();
    let f_write_result = f.write("Hello world".as_bytes());
    if f_write_result.is_err() {
        return;
    }
    println!("File a.txt has been created !");
}
```

Output

```
File a.txt has been created !
```



File operations

Notice that there is no .close() method in type File. This is because it is not needed. When “f” scope is over, any handle to the file object will be closed.

If you need to control when file is being close, you can open/create it into its own scope. When its scope has ended, any handle that a File might have will be closed.

Rust

```
fn main() -> Result<(),std::io::Error> {
    println!("Prepare to create a file...");
    {
        let mut f = File::create("a.txt")?;
        f.write("Hello world".as_bytes())?;
    }
    println!("File a.txt has been created !");
    Ok(())
}
```

The scope of variable “f”.

File operations

Alternatively, you can force dropping the variable (via function **drop**). This function moves the object (takes the ownership of the memory the object has). Under the hood, if the Drop trait is implemented, it acts as a destructor and it is called. For an object of type File, the immediate result is that its handle is closed.

Rust

```rust
use std::fs::File;
use std::io::Write;

fn main() -> Result<(), std::io::Error> {
    println!("Prepare to create a file...");
    let mut f = File::create("a.txt")?;
    f.write("Hello world".as_bytes())?;
    drop(f);
    println!("File a.txt has been created !");
    Ok(())
}
```

Output

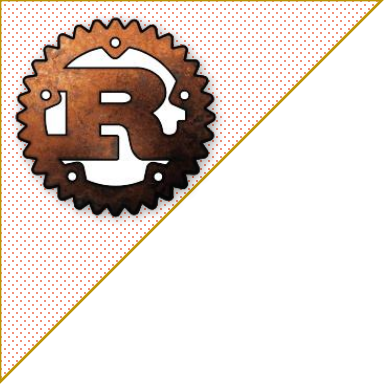
```
Prepare to create a file...
File a.txt has been created !
```

At this point “a.txt” is being closed.

File operations

But what if we want to open a file in a different way (for example, create it if it does not exist, but if it does, just open it and move the file position at the end of the file so that we can add data at its end). For this we have the .options() method that returns an **OpenOptions** type with the following methods:

Method (OpenOptios)	Usage
`fn read(&mut self, value: bool) -> &mut Self`	Specifies that ***Read*** mode should be used when opening a file.
`fn write(&mut self, value: bool) -> &mut Self`	Specifies that ***Write*** mode should be used when opening a file.
`fn append(&mut self, value: bool) -> &mut Self`	Specifies that the file pointer will be position at the end of the file.
`fn truncate(&mut self, value: bool) -> &mut Self`	Specifies that the file (if exists) should be truncated.
`fn create(&mut self, value: bool) -> &mut Self`	Specifies that the file should be created if it does not exist.
`fn open (&mut self, path: &Path) -> Result<File,Error>`	Opens a file using the specific open file options specified.



File operations

Let’s see how options() can be used:

Rust

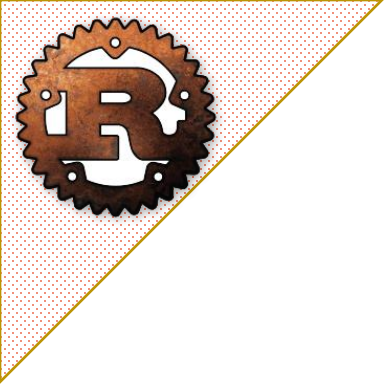
```rust
fn main() -> Result<(), std::io::Error> {
    let mut f = File::options().read(true)
                               .write(true)
                               .append(true)
                               .create(true)
                               .open("a.txt")?;
    f.write("123\n".as_bytes())?;
    println!("File written !");
    Ok(())
}
```

Output

```
File written !
```

After 3 consecutive executions, assuming that initially “a.txt” was not present on the storage device, a new file “a.txt” should appear with the following values:

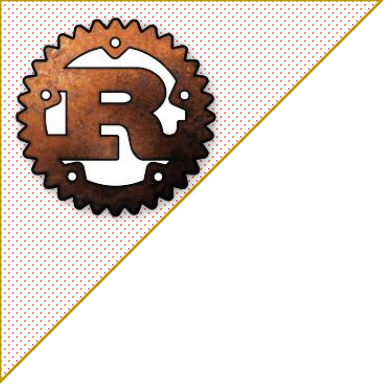
```
123
123
123
```



File operations

Once a file has been opened / created, there are a set of method that can be used:

Method	Usage
`fn read(&mut self, buf: &mut [u8]) -> Result<usize, Error>`	Reads the content of the file into the buffer. Returns the number of bytes read, or an error.
`fn read_to_end(&mut self, buf: &mut Vec<u8>) -> Result<usize, Error>`	Reads the content of the file (from the current position to the file ends) into a vector.
`fn read_to_string(&mut self, buf: &mut String) -> Result<usize, Error>`	Reads the content of a string into a string. The content must be a valid UTF-8 text.
`fn read_exact(&mut self, &mut [u8]) -> Result<(), Error>`	Reads the exact size of the buffer or returns an Err otherwise.
`fn write(&mut self, buf: &[u8]) -> Result<usize, Error>`	Writes the content of buf into the file. The result, if Ok represents the number of bytes written.
`fn write_all(&mut self, buf: &[u8]) -> Result<(), Error>`	Writes the entire content of buf into the file. If it can not write the content of buf into a file, it returns an error.
`fn flush(&mut self) -> Result<(), Error>`	Flushes the content of the file to disk.
`fn seek(&mut self, pos: SeekFrom) -> Result<usize, Error>`	Changes the position of the current file pointer.
`fn set_len(&self, size: u64) -> Result<(), Error>`	Sets the length of the file. Truncates the file if needed or adds 0 until the new size.



File operations

In terms of Read/Write operations, there are two scenarios:

1. You either want to write an entire buffer/data to a file, or you want to read a content of a specific (fix-sized) buffer from a file. If this is the case, you should use the .read_exact(…) or .write_all(…) methods. This is usually one of the most common case, where you write/read binary data that reflects a certain structure into a file and writing/reading partial data is consider an error.

2. It is possible to read/write partial data. In this case, you should use .read(…) and .write(…) methods. This is a less common case, but it is useful in certain situations. For example, writing data to a log (even if the disk space does is not enough, it is still important to write as mush information as you can).

File operations

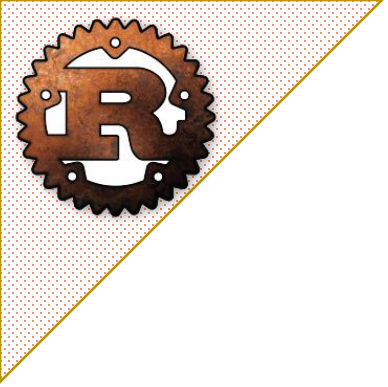
Let’s see an example:

Rust

```rust
fn main() -> Result<(), std::io::Error> {
    let mut f = File::open("a.txt")?;
    let mut content = Vec::<u8>::new();
    f.seek(SeekFrom::Start(0))?;
    f.read_to_end(&mut content)?;
    println!("Content size: {}", content.len());
    println!("Content: {:?}", content);
    Ok(())
}
```

Output (possible)

```
Content size: 8
Content: [49, 50, 51, 10, 49, 50, 51, 10]
```



File operations

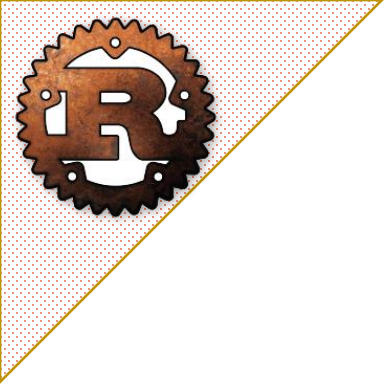
You might have notice that there is no equivalent for a ***tell/ftell*** methods. This is because it is not needed → we can use seek.

The **SeekFrom** enum is defined as follows:

Rust

```rust
pub enum SeekFrom {
    Start(u64),
    End(i64),
    Current(i64),
}
```

This means that we can use .seek(SeekFrom.Current(0)) and get the same result as a ftell/tell method. Similarly, if we want to obtained the size of a file, we can use something similar: .seek(SeekFrom.End(0))

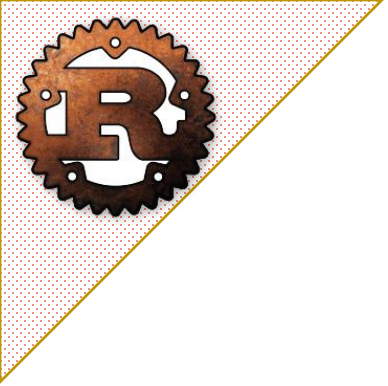


File operations

Every file has a set of meta data information associated with it (such as permission, times (created, modified, last accessed), attributes, etc).

Method (for File)	Usage
`fn metadata(&self) -> Result<Metadata, Error>`	Returns an instance of type Metadata.

Method (for Metadata)	Usage
`fn accessed (&self) -> Result<SystemTime, Error>`	Return the last access time for the current file.
`fn created (&self) -> Result<SystemTime, Error>`	Return the time when the file was created.
`fn modified (&self) -> Result<SystemTime, Error>`	Return the time when the file was last modified.
`fn is_dir(&self) -> bool`	True if current object is a directory, false otherwise
`fn is_file(&self) -> bool`	True if current object is a file, false otherwise
`fn is_symlink(&self) -> bool`	True if current object is a symlink, false otherwise
`fn len(&self) -> u64`	Returns the length of the file
`fn permissions(&self) -> Permissions`	Return the associated permissions with the open file.



File operations

Let’s see an example:

Rust

```rust
fn main() {
    let meta = File::open("a.txt").unwrap().metadata().unwrap();
    println!("Created: {:?}",meta.created().unwrap());
    println!("Modified: {:?}",meta.modified().unwrap());
    println!("Last access: {:?}",meta.accessed().unwrap());
    println!("Len: {}",meta.len());
    println!("IsFile: {}",meta.is_file());
    println!("Permission: {:?}",meta.permissions());
}
```

Output (possible)

```
Created: SystemTime { intervals: 133121473875242934 }
Modified: SystemTime { intervals: 133121474894848321 }
Last access: SystemTime { intervals: 133121589503343172 }
Len: 8
IsFile: true
Permission: Permissions(FilePermissions { attrs: 32 })
```

File operations

Every file operation, if it fails returns an Error object with details. In the next example, let's assume that file "blablabla.txt" does not exist. In this case, opening that file will return an error as follows:

Rust

```rust
fn main() {
    let f = File::open("blablabla.txt");
    if f.is_err() {
        println!("{:?}",f.err().unwrap());
    }
}
```

Output (possible)

```
Os
{
    code: 2,
    kind: NotFound,
    message: "The system cannot find the file specified."
}
```

File operations

For simplicity, the file system namespace (std::fs) has two methods that can read and write the entire content of a file.

Method (for std::fs)	Usage
`fn read (path: &Path) -> Result<Vec<u8>, Error>`	Reads the content of a file into a vector.
`fn read_to_string (path: &Path) -> Result<String, Error>`	Reads the content of a file into a string.
`fn write (path: &Path, contents: &[u8]) -> Result<(), Error>`	Writes the content of a buffer into a file.

… and an example:

Rust

```rust
fn main() {
    let content_binary = std::fs::read("a.txt").unwrap();
    let content_string = std::fs::read_to_string("a.txt").unwrap();
    println!("Bin = {:?}", content_binary);
    println!("Str = {:?}", content_string);
}
```

Output (possible)

```
Bin = [49, 50, 51, 10, 49, 50, 51, 10]
Str = "123\n123\n"
```

File operations

Keep in mind that File type does not have any caching mechanism. As such, multiple small operations (e.g. read file byte by byte) will be slow. However, Rust has two types: **BufReader** and **BufWriter** that can be used for caching.

Method (for BufReader/BufWritter)	Usage
`fn new(inner: R) -> BufReader<R>` `fn new(inner: W) -> BufWritter<W>`	Creates a new buffered reader/writer over the inner object (the inner object has to have the Read/Write trait). The default cache size is used (for most OS-es is 8k)
`fn with_capacity(capacity: usize, inner: R) -> BufReader<R>` `fn with_capacity(capacity: usize, inner: W) -> BufWriter<W>`	Creates a new buffered reader/writer, but if a specified cache size.

File operations

Once a BufReader/BufWriter object has been created, the following methods are available:

Method (BufReader)	Usage

`fn read(&mut self, buf: &mut [u8]) -> Result<usize, Error>`
`fn read_to_end(&mut self, buf: &mut Vec<u8>) -> Result<usize, Error>`
`fn read_to_string(&mut self, buf: &mut String) -> Result<usize, Error>`
`fn read_exact(&mut self, &mut [u8]) -> Result<(), Error>`
	Similar to what File type provides (the only difference is that the data is cached first).

`fn read_line(&mut self, line: &mut String) -> Result<usize>`	Reads a line from the file and puts it in the line variable.
`fn read_until(&mut self, delim: u8, buf: &mut Vec<u8>)-> Result<usize>`	Reads all bytes into a vector until "**delim**" u8 value is found.
`fn bytes(self) -> Bytes<Self>`	Returns an iterator over all bytes from the reader.
`fn lines(self) -> Lines<Self>`	Returns an iterator over all lines from the reader.

Method (BufWriter)	Usage
`fn write(&mut self, buf: &[u8]) -> Result<usize, Error>` `fn flush(&mut self) -> Result<(), Error>`	Similar to what File type provides.

File operations

Let’s see an example that reads a file line by line and prints all lines that have a specific text:

Rust

```rust
fn main() -> Result<(), std::io::Error> {
    let f = File::open("log.txt")?;
    let r = BufReader::new(f);
    for item in r.lines() {
        if let Ok(line) = item {
            if line.contains("[ERROR]") {
                println!("{line}");
            }
        }
    }
    Ok(())
}
```

Output (possible)

```
[ERROR] Unable to connect to 192.168.0.1
```

File operations

Keep in mind that .lines() iterator builds a new string for each line. If you want to reuse an existing one, use the .read_line() method. Keep in mind that if .read_line() method returns Ok(0) then the end of file (reader) has been reached.

Rust

```rust
fn main() -> Result<(), std::io::Error> {
    let f = File::open("log.txt")?;
    let mut r = BufReader::with_capacity(0x10000, f);
    let mut line = String::with_capacity(1024);
    while let Ok(bytes_read) = r.read_line(&mut line) {
        if bytes_read == 0 { break; /* EOF reached */ }
        if line.contains("[ERROR]") {
            println!("--> {line}");
        }
    }
    Ok(())
}
```

Output (possible)

→ [ERROR] Unable to connect to 192.168.0.1

File operations

Keep in mind that once a BufReader/BufWriter has been created, the ownership for the file object is transferred to this object and as such, the original File object can not be used anymore:

Rust

```rust
fn main() -> Result<(), std::io::Error> {
    let f = File::open("log.txt")?;
    let mut r = BufReader::with_capacity(0x10000, f);
    let mut text = String::new();
    f.read_to_string(&mut text);
    Ok(())
}
```

Error

```
error[E0382]: borrow of moved value: `f`
 --> src\main.rs:7:5
  |
4 |     let f = File::open("log.txt")?;
  |         - move occurs because `f` has type `File`, which does not
  |           implement the `Copy` trait
5 |     let mut r = BufReader::with_capacity(0x10000, f);
  |                                                  - value moved here
6 |     let mut text = String::new();
7 |     f.read_to_string(&mut text);
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^ value borrowed here after move
```

File operations

However, if for some reason, access to the original object is required, there are two method .get_ref() and .get_mut() that can be used with BufReader/BufWriter to get a reference to the original object:

Rust

```rust
fn main() -> Result<(), std::io::Error> {
    let f = File::open("log.txt")?;
    let r = BufReader::with_capacity(0x10000, f);
    let mut text = String::new();
    let mut f_ref = r.get_ref();
    f_ref.read_to_string(&mut text);
    println!("{}", text);
    Ok(())
}
```

Output (possible)

```
123
123
123
```

File operations

Rust implementation of file operations is based on 3 traits (Write, Read and Seek), defined as follows:

Rust (Write trait – from mod.rs)

```rust
pub trait Write {
    // need to be implemented
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    // with implementation
    fn write_vectored(&mut self, bufs: &[IoSlice<'_>]) -> Result<usize> { ... }
    fn write_all(&mut self, mut buf: &[u8]) -> Result<()> { ... }
    fn write_fmt(&mut self, fmt: fmt::Arguments<'_>) -> Result<()> { ... }
}
```

Rust (Seek trait – from mod.rs)

```rust
pub trait Seek {
    // need to be implemented
    fn seek(&mut self, pos: SeekFrom) -> Result<u64>;

    // with implementation
    fn rewind(&mut self) -> Result<()> { ... }
    fn stream_position(&mut self) -> Result<u64> { ... }
}
```

File operations

Rust implementation of file operations is based on 3 traits (Write, Read and Seek), defined as follows:

Rust (Write trait – from mod.rs)

```rust
pub trait Read {
    // need to be implemented
    fn read(&mut self, buf: &mut [u8]) -> Result<usize>;

    // with implementation
    fn read_vectored(&mut self, bufs: &mut [IoSliceMut<'_>]) -> Result<usize> { ... }
    fn read_to_end(&mut self, buf: &mut Vec<u8>) -> Result<usize> { ... }
    fn read_to_string(&mut self, buf: &mut String) -> Result<usize> { ... }
    fn read_exact(&mut self, buf: &mut [u8]) -> Result<()> { ... }

    fn bytes(self) -> Bytes<Self> where Self: Sized { ... }
    fn chain<R: Read>(self, next: R) -> Chain<Self, R> where Self: Sized { ... }
    fn take(self, limit: u64) -> Take<Self> where Self: Sized { ... }
}
```

File operations

The logic behind the implementation of these traits is that you can perform agnostic operation on every type that implements these traits, such as:

- File
- TcpStream
- Sink (just Write)
- Stdin (just Read) and Stdout/Stderr (just Write)
- ChildStdin (just Read), ChildStdout/Childstderr (just Write) → for stream related to child processes
- … and many more …

These traits can be used to perform generic operation (for example you can provide to a function a file or a TCP stream and it will behave in a similar manner as they both implement Read trait).

File system operations

File System operations

Rust library (std::fs) provides a series of methods that can be used for file system operations:

Method	Usage
`fn copy (from: &Path, to: &Path) -> Result<u64, Error>`	Copies the content of one file into another. If the destination file exists, it will be overwritten. The function returns the number of bytes copied.
`fn create_dir (path: &Path) -> Result<(), Error>` `fn create_dir_all (path: &Path) -> Result<(), Error>`	Creates a directory (simple or recursively) if a full path is provided.
`fn hard_link (from: &Path, to: &Path) -> Result<(), Error>` `fn soft_link (from: &Path, to: &Path) -> Result<(), Error>`	Creates a hard or a soft link.
`fn remove_dir (path: &Path) -> Result<(), Error>` `fn remove_dir_all (path: &Path) -> Result<(), Error>`	Removes a directory (empty or recursively).
`fn remove_file (path: &Path) -> Result<(), Error>`	Removes a file.
`fn rename (from: &Path, to: &Path) -> Result<(), Error>`	Renames a file/directory to a new name.
`fn read_dir (path: &Path) -> Result<ReadDir, Error>`	Returns an iterator that allows enumerating the content of a directory

File System operations

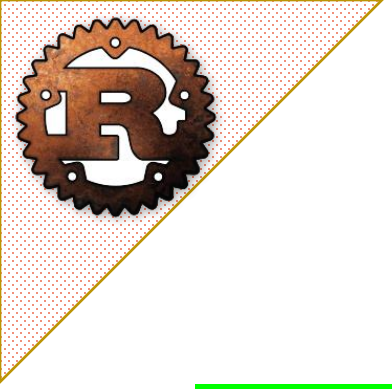
In the next example we will do the following:

- Create a new folder (called temp)
- Write a file (named a.txt) in folder temp
- Rename that file from a.txt to b.txt
- Delete folder temp and all of its content

Rust

```rust
fn main() -> Result<(), Error> {
    std::fs::create_dir("temp")?;
    std::fs::write("temp/a.txt", "Rust".as_bytes())?;
    std::fs::rename("temp/a.txt", "temp/b.txt")?;
    std::fs::remove_dir_all("temp")?;
    println!("All ok");
    Ok(())
}
```

Output (possible)
All ok



File System operations

std::fs::read_dir(…) method returns an iterator that can be used to enumerate the files and subdirectories from a given folder. The iterator yells a structure for type **DirEntry** for each file, with the following properties:

- file_name ➔ the name of the file (as represented in the OS you are running).
- path ➔ the path of the file (as represented in the OS you are running).
- metadata ➔ access to a entry metadata (times, type, permission, etc).

Rust

```rust
fn main() {
    if let Ok(dir_it) = std::fs::read_dir(".") {
        for dir_entry_result in dir_it {
            if let Ok(dir_entry) = dir_entry_result {
                println!("Name   : {:?}", dir_entry.file_name());
                println!("Full path: {:?}", dir_entry.path());
                println!("Folder  : {:?}", dir_entry.metadata().unwrap().is_dir());
                println!("-----------------------------------------------");
            }
        }
    }
}
```

Output (possible)

```
Name     : ".git"
Full path: ".\\.git"
Folder   : true
---------------------------
Name     : ".gitignore"
Full path: ".\\.gitignore"
Folder   : false
---------------------------
Name     : ".vscode"
Full path: ".\\.vscode"
Folder   : true
---------------------------
Name     : "a.txt"
Full path: ".\\a.txt"
Folder   : false
-----------------------------------------------
```

File System operations

Every operating system has a different way of storing strings.

Since the API from std::fs has to be generic, the methods that return a string (such as read_dir) return a string in the OS format. To simplify, these methods store a vector of bytes (**u8**) that represent how a specific string is store in the OS the code is running.

Examples:

- Windows stores paths as WTF-16 (followed by a \0 char) or ASCIIZ
- MAC/OSC uses UTF-8 (also followed by a \0 char)
- Linux work well UTF-8 but in reality, Linux is encoding agnostic.

Rust does not have the \0 char as a concept (all strings are UTF-8 and have a size). As such it has to store the OS result in a buffer and converted in a string that has follows the internal requirements (UTF-8).

File System operations

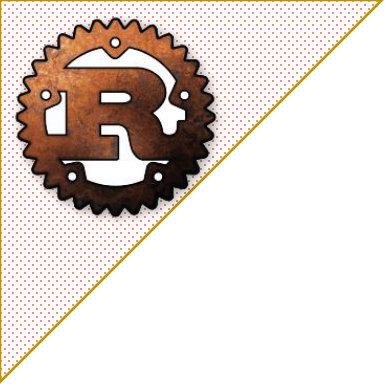
The following code iterates through all files and subdirectories from the current folder and converts their name into a Rust String object (via the .into_string() method).

Rust

```rust
use std::io::Error;

fn main() -> Result<(), Error> {
    let dir = std::fs::read_dir(".")?;
    for dir_entry_result in dir {
        let dir_entry = dir_entry_result?;
        if let Ok(fname) = dir_entry.file_name().into_string() {
            println!("name: {}", fname);
        }
    }
    Ok(())
}
```

Output (possible)

```
name: .git
name: .gitignore
name: .vscode
name: a.txt
name: Cargo.lock
name: Cargo.toml
name: src
name: target
```



File System operations

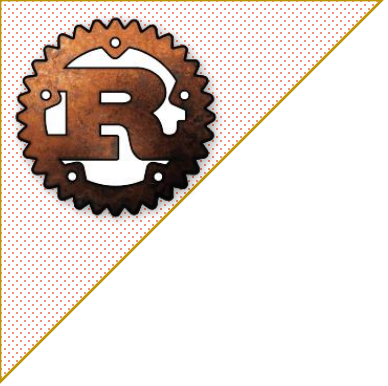
Let’s see how we can use read_dir(…) to recursively traverse a folder:

Rust

```rust
use std::{io::Error, path::Path};

fn enumerate_dir(path: &Path) -> Result<(), Error> {
    let dir = std::fs::read_dir(path)?;
    for dir_entry_result in dir {
        let file_path = dir_entry_result?.path();
        if file_path.is_dir() {
            enumerate_dir(&file_path)?;
        } else {
            println!("File: {:?}", file_path);
        }
    }
    Ok(())
}
fn main() {
    println!("Result = {:?}", enumerate_dir(Path::new(".")));
}
```

Output (possible)

```
File: ".\\.git\\config"
File: ".\\.git\\description"
File: ".\\.git\\HEAD"
File: ".\\.git\\hooks\\applypatch-msg.sample"
File: ".\\.git\\hooks\\commit-msg.sample"
File: ".\\.git\\hooks\\fsmonitor-watchman.sample"
...
File: ".\\.vscode\\launch.json"
File: ".\\a.txt"
File: ".\\Cargo.lock"
File: ".\\Cargo.toml"
....
File: ".\\target\\release\\first.exe"
File: ".\\target\\release\\first.pdb"
Result = Ok(())
```



File System operations

Let’s see how we can use read_dir(…) to recursively traverse a folder:

Rust

```
use std::{io::Error, path::Path};

fn enumerate_dir(path: &Path)->Result<(),Error> {
    let dir = std::fs::read_dir(path)?;
    for dir_entry_result in dir {
        let file_path = dir_entry_result?.path();
        if file_path.is_dir() {
            enumerate_dir(&file_path)?;
        }
    }
    Ok(())
}
fn main() {
    println!("Result = {:?}",enumerate_dir(Path::new(".")));
}
```

Output (possible)

```
File: ".\\.git\\config"
File: ".\\.git\\description"
File: ".\\.git\\HEAD"
File: ".\\.git\\hooks\\applypatch-msg.sample"
File: ".\\.git\\hooks\\commit-msg.sample"
File: ".\\.git\\hooks\\fsmonitor-watchman.sample"
...
File: ".\\.vscode\\launch.json"
File: ".\\a.txt"
e\\first.exe"
e\\first.pdb"
```

Notice the usage of “**?.**”; this translates in the following way:

- If the left-most expression results in an error, then return the error
- Else, grab the object returned by the left-most expression and call member operator (‘**.**’) on it
- In our case ➔ if dir_entry_result is not an error, then call method **.path()** from dir_entry and return the result.

File System operations

We can also use a callback so that we can externalize what we do with the file name:

Rust

```rust
fn enumerate_dir<CallBack>(path: &Path, cbk: &mut CallBack) -> Result<(), Error>
where
    CallBack: FnMut(&str),
{
    let dir = std::fs::read_dir(path)?;
    for dir_entry_result in dir {
        let file_path = dir_entry_result?.path();
        if file_path.is_dir() {
            enumerate_dir(&file_path, cbk)?;
        } else {
            if let Some(file_path_string) = file_path.to_str() {
                cbk(&file_path_string);
            }
        }
    }
    Ok(())
}
fn main() {
    enumerate_dir(Path::new("."), &mut |path| { println!("Path: {path}"); });
}
```

Output (possible)

```
Path: .\target\debug\first.d
Path: .\target\debug\first.exe
Path: .\target\debug\first.pdb
...
Path: .\target\release\.cargo-lock
...
Path: .\target\release\deps\first.d
Path: .\target\release\deps\first.exe
Path: .\target\release\deps\first.pdb
...
Path: .\target\release\first.d
Path: .\target\release\first.exe
Path: .\target\release\first.pdb
```

File System operations

We can also use a closure to store some data locally and then process them afterwards:

Rust

```rust
fn enumerate_dir<CallBack>(path: &Path, cbk: &mut CallBack) -> Result<(), Error>
where
    CallBack: FnMut(&str),
{
    ...
}
fn main() {
    let mut s = Vec::<String>::new();
    enumerate_dir(Path::new("."), &mut |path| {
        if path.contains(".git") {
            s.push(path.to_string());
        }
    });
    println!("Items in s: {}", s.len());
}
```

Output (possible)
Items in s: 18

Execution environment

Execution environment

Every program upon execution has an execution context / environment that consists of command line arguments, current directory, home directory, environment variables, etc. All of these can be obtained via std::env namespace.

Method (std::env)	Usage
`fn args() -> Args` `fn args_os() -> ArgsOs`	Returns an iterator over command line arguments. All arguments must be valid Unicode strings. Use ***args_os*** if youre program can receive non valid UTF-8 characters.
`fn current_dir() -> Result<PathBuf, Error>`	Returns the current directory.
`fn current_exe() -> Result<PathBuf, Error>`	Returns the full path of the current program.
`fn home_dir() -> Option<PathBuf>`	Returns the home directory.
`fn temp_dir() -> PathBuf`	Returns the path of the temporary folder.
`fn var (key: &OsStr) -> Result<String, VarError>` `fn var_os (key: &OsStr) -> Option<OsString>`	Return the value of a system variable. The name of the variable must be provided via an OsStr object.
`fn vars() -> Vars` `fn vars_os() -> VarsOs`	Returns an iterator that can be used to iterate over the existing OS variables.

Execution environment

Let’s see an example:

Rust

```rust
fn main() {
    println!("Current path: {:?}", std::env::current_dir());
    println!("Current exe : {:?}", std::env::current_exe());
    println!("Home folder : {:?}", std::env::home_dir());
    println!("Temp folder : {:?}", std::env::temp_dir());
}
```

Output (possible)

```
Current path: Ok("E:\\Lucru\\Rust\\first")
Current exe : Ok("E:\\Lucru\\Rust\\first\\target\\debug\\first.exe")
Home folder : Some("C:\\Users\\<USER>")
Temp folder : "C:\\Users\\<USER>\\AppData\\Local\\Temp\\"
```

OBS: *Keep in mind that the result of this snipped might vary depending on the OS. Furthermore, .home_dir() is considered deprecated.*

Execution environment

Let’s consider that we run our executable (first.exe) with 3 arguments (“Rust”, “1” and “true”). Then this script should output the following:

Rust

```rust
fn main() {
    for arg in std::env::args() {
        println!("arg: {}", arg);
    }
}
```

Output (possible)

```
arg: first.exe
arg: Rust
arg: 1
arg: True
```

OBS: *If arguments order is required, consider using enumerate (std::env::args::enumerate)*

Execution environment

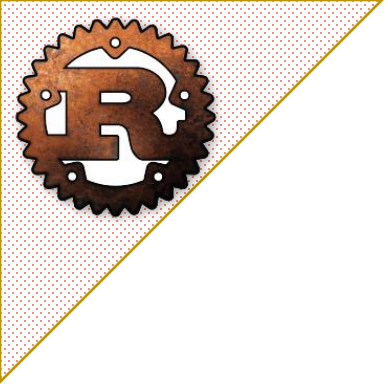
Similarly, the following code lists all system variables available upon execution of a Rust program (specific to Windows environment):

Rust

```rust
fn main() {
    for var in std::env::vars() {
        println!("var: {} -> {}", var.0, var.1);
    }
}
```

Output (possible)

```
var: ALLUSERSPROFILE -> C:\ProgramData
...
var: CARGO_HOME -> C:\Users\<User>\.cargo
...
var: COLORTERM -> truecolor
var: CommonProgramFiles -> C:\Program Files\Common Files
var: HOMEPATH -> \Users\<USER>
var: LANG -> en_US.UTF-8
var: OS -> Windows_NT
...
var: PROCESSOR_IDENTIFIER -> Intel64 Family 6 Model 140 Stepping 1, GenuineIntel
...
var: RUSTUP_TOOLCHAIN -> stable-x86_64-pc-windows-msvc
...
```



FFI

FFI

FFI (**F**oreign **F**unction **I**nterface) is used for interoperability between different programming languages and interfacing with system-level APIs. In particular for Rust, we will discuss C/C++ and Rust interoperability.

FFI can be divided into two major categories:

A) Access from Rust code written in a different language
- Functions
- Classes / structs

B) Access from another language code written in Rust
- Functions
- Classes / structs

FFI

1. Accessing a function from an external library

We also have two scenarios in this case:

1. We want to link the external library statically
2. We want to dynamically load an external library and then load the function

Scenario no. 2 is more suitable for building a plugin-based application where each plugin is a library that is loaded dynamically.

FFI

1. Accessing a function from an external library (static linkage)

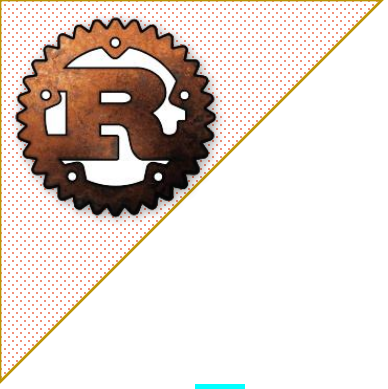
The general format in this case is as follows:

```
#[link(…)]
extern “ABI type”
{
        // other attributes
        // functions
}
```

Where:

- #[link(…)] attribute provides information on the library that needs to be linked
- “ABI type” provides information on the calling conversion

OBS: *It is important to notice that every call to a function imported in this way is* ***unsafe*** *as Rust can not guarantee the safeness of the imported code.*



FFI

1. Accessing a function from an external library (static linkage)

The #[link(…)] attribute is defined in the following way:

```
#[link(name="…", kind="…", modifiers="…", …)]
```

Where:

Parameter	Optional	Purpose
name	No	Name of the library to link
kind	Yes	Type of library: could be one of dylib , static , framework, raw-dylib. If this parameter is not provided, its value will be defaulted to **dylib**.
modifiers	Yes	How the linkage has to be performed: values like bundle, whole-archive, verbatim Some specifier can be used only with a certain type of linkage.

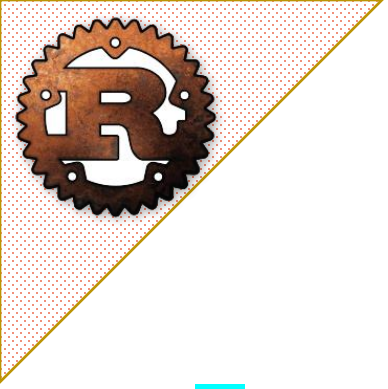
More details on: https://doc.rust-lang.org/reference/items/external-blocks.html

FFI

1. Accessing a function from an external library (static linkage)

The ABI type can be one of the following:

ABI Type	Purpose
Rust	The default calling conversion used by Rust
C	The default calling conversion used by C/C++ compiler
system	Default calling conversion used by the OS. Usually, the same as the one used by C/C++ compiler except for Win32 where is **stdcall**
cdecl	Default for C code (X86_32)
stdcall	Default for win32
fastcall	In MS C compiler (__fastcall), in GCC or C-Lang (__attribute__((fastcall))
vectorcall	In MS C compiler (__vectorcall), in GCC or C-Lang (__attribute__((vectorcall))
thiscall	In MS C compiler (__thiscall), in GCC or C-Lang (__attribute__((thiscall))
efiapi	Used for UEFI ABI
…	*There are other ABIs available (for various systems).*



FFI

1. Accessing a function from an external library (static linkage)

Let's assume the following simple function in C++ and see how the assembly looks like if we call it with different specifiers:

C/C++

```
int sum(int x, int y) { return x + y; }
```

ABI Type ➔	cdecl	stdcall	fastcall

Function Code	`push ebp`
	`mov ebp, esp`
	`mov eax, [ebp+8] // x`
	`add eax, [ebp+12] // y`
	`mov esp, ebp`
	`pop ebp`
	`ret`
		`push ebp`
		`mov ebp, esp`
		`mov eax, [ebp+8] // x`
		`add eax, [ebp+12] // y`
		`mov esp, ebp`
		`pop ebp`
		`ret 8`
			`mov eax, ecx // x`
			`add eax, edx // y`
			`ret`

Calling Code	`push y`
	`push x`
	`call sum`
	`add esp,8 // 8=2(params)*4`
		`push y`
		`push x`
		`call sum`
			`mov ecx, x`
			`mov edx, y`
			`call sum`

FFI

1. Accessing a function from an external library (static linkage)

There are also some scenarios where:

- We don't want to use the same name for the imported function (e.g. the imported function does not use snake case formatting – ***GetTickCount*** is the name of a Windows API function, but it is written in camel case → the similar name in Rust should have been ***get_tick_count***)
- When the name of the exported function is mangled.
- The function we want to import is exported via an ***ordinal*** (instead of a name)

In these cases, we can use other attributes directly when describing the imported function:

Rust

```rust
extern "..." {
    #[link_name = "actual_name"]
    fn rust_name(...);
}
```

Rust

```rust
extern "..." {
    #[link_ordinale(<value>)]
    fn rust_name(...);
}
```

FFI

1. Accessing a function from an external library (static linkage)

In this example we import ***GetTickCount*** function from Windows API (located in library **kernel32.dll**):

Rust

```rust
#[link(name = "kernel32")]
extern "system" {
    fn GetTickCount()->u32;
}
fn main() {
    let result = unsafe { GetTickCount() };
    println!("{}",result);
}
```

Output (possible)
70998046

FFI

1. Accessing a function from an external library (static linkage)

In this example we import ***GetTickCount*** function from Windows API (located in library **kernel32.dll**):

Rust

```
#[link(name = "kernel32")]
extern "system" {
    fn GetTickCount()->u32;
}
fn main() {
    let result = unsafe { GetTickCount() };
    println!("{}",result);
}
```

This method is defined in Windows SDK as follows:
DWORD GetTickCount();
DWORD mean a 32bit unsigned value so we can use u32 instead.

FFI

1. Accessing a function from an external library (static linkage)

In this example we import ***GetTickCount*** function from Windows API (located in library **kernel32.dll**):

Rust

```
#[link(name = "kernel32")]
extern "system" {
    fn GetTickCount()->u32;
}
fn main() {
    let result = unsafe {
    println!("{}",result);
}
```

Notice the usage of unsafe to call this method (this is required as Rust can not guarantee that the code being called is safe).

FFI

1. Accessing a function from an external library (static linkage)

In this example we import ***GetTickCount*** function from Windows API (located in library **kernel32.dll**):

Rust

```
#[link(name = "kernel32", kind="dylib")]
extern "system" {
    fn GetTickCount()->u32;
}
fn main() {
    let result = unsafe { GetTickCount() };
    println!("{}",result);
}
```

When using **dylib**, an import table is being created and the called is performed via that table.

```
mov   rax, qword ptr [__imp_GetTickCount]
call  rax
mov   dword ptr [result],eax
```

FFI

1. Accessing a function from an external library (static linkage)

In this example we import ***GetTickCount*** function from Windows API (located in library **kernel32.dll**):

Rust

```
#[link(name = "kernel32", kind="static")]
extern "system" {
    fn GetTickCount()->u32;
}
fn main() {
    let result = unsafe { GetTickCount() };
    println!("{}",result);
}
```

When using **static**, a direct call to the API is being used.

```
call   GetTickCount
mov    dword ptr [result],eax
```

FFI

1. Accessing a function from an external library (static linkage)

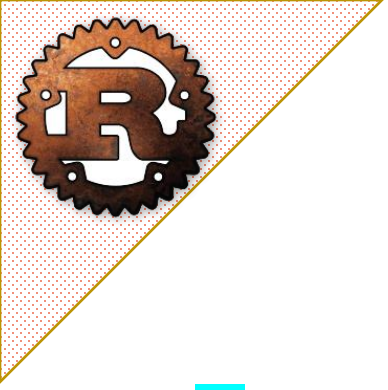
In this example we import ***GetTickCount*** function from Windows API (located in library **kernel32.dll**):

Rust

```rust
extern "system" {
    fn GetTickCount()->u32;
}
fn main() {
    let result = unsafe { GetTickCount() };
    println!("{}",result);
}
```

Output (possible)
70998046

OBS: Notice that we have removed the **#[link(...)]** attribute. In this case, Rust will search for the imported function in the libraries normally loaded by the current application (and as kernel32.dll is loaded in every user mode program in Windows the code will work as expected).



FFI

1. Accessing a function from an external library (static linkage)

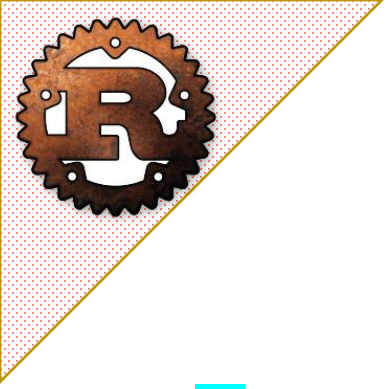
It is important to highlight that the more complex an exported function is, the more complicated the code that calls that function will be. Let's analyze how the code the calls the following function defined in Win32 API (***GetSystemTime***) should look like:

Definition:

```
void GetSystemTime( [out] LPSYSTEMTIME lpSystemTime);
```

with:

```
typedef struct _SYSTEMTIME {
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME, *LPSYSTEMTIME;
```



FFI

1. Accessing a function from an external library (static linkage)

Rust (SYSTEMTIME library)

```rust
#[repr(C)]
#[derive(Default)]
#[allow(non_snake_case)]
struct SYSTEMTIME {
    wYear: u16,
    wMonth: u16,
    wDayOfWeek: u16,
    wDay: u16,
    wHour: u16,
    wMinute: u16,
    wSecond: u16,
    wMilliseconds: u16
}
```

Where:

- #[repr(C)] → we don't want Rust to reorder data members order (we want to keep them exactly how C language will align them in memory)
- #[derive(Default)] → Rust will not allow us to create an uninitialized data object. As such we need a default way to initialize all fields.
- #[allow(non_snake_case)] → By default Win32 APIs are camel case. If we don't want Rust to yield a warning in this case, we have to tell him that this is in fact how we want the structure to be named.

FFI

1. Accessing a function from an external library (static linkage)

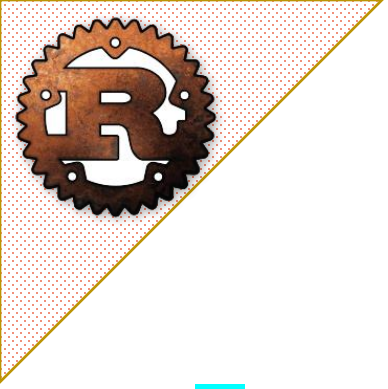
Rust (SYSTEMTIME library)

```rust
#[repr(C)]
#[derive(Default)]
#[allow(non_snake_case)]
struct SYSTEMTIME {
    wYear: u16,
    wMonth: u16,
    wDayOfWeek: u16,
    wDay: u16,
    wHour: u16,
    wMinute: u16,
    wSecond: u16,
    wMilliseconds: u16
}
```

Rust (linking the function)

```rust
#[link(name="kernel32")]
extern "system" {
    #[link_name = "GetSystemTime"]
    fn get_date_time(lpSystemTime: *mut SYSTEMTIME);
}
```

We will link this function with a rust name (get_date_time). Notice that the parameter is of type *mut SYSTEMTIME (as a correspondence to the **LPSYSTEMTIME** from C/C++)



FFI

1. Accessing a function from an external library (static linkage)

Rust (SYSTEMTIME library)

```rust
#[repr(C)]
#[derive(Default)]
#[allow(non_snake_case)]
struct SYSTEMTIME {
    wYear: u16,
    wMonth: u16,
    wDayOfWeek: u16,
    wDay: u16,
    wHour: u16,
    wMinute: u16,
    wSecond: u16,
    wMilliseconds: u16
}
```

Rust (linking the function)

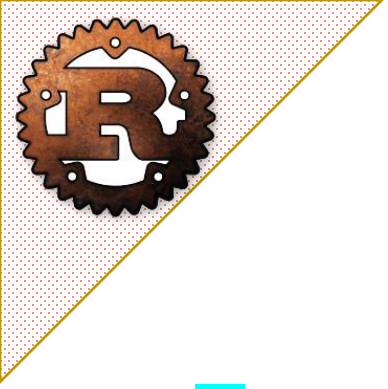
```rust
#[link(name="kernel32")]
extern "system" {
    #[link_name = "GetSystemTime"]
    fn get_date_time(lpSystemTime: *mut SYSTEMTIME);
}
```

Rust (main)

```rust
fn main() {
    let mut dt = SYSTEMTIME::default();
    unsafe { get_date_time(&mut dt) };
    println!(
        "Current time: {}:{}:{}.{}",
        dt.wHour, dt.wMinute, dt.wSecond, dt.wMilliseconds
    );
}
```

Output (possible)

```
20:27:15.134
```



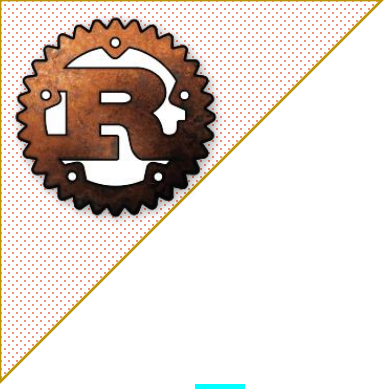
FFI

1. Accessing a function from an external library (static linkage)

Notice that the more complex a function is (in terms of parameters) the more code and structures we need to create to accommodate.

Up to this point, all of the examples used data types that were compatible with the ones from Rust (such as WORD that is an u16 in Rust, or DWORD that is a u32 in Rust). But what if there are some data types that Rust does not support ? (for example, an ASCIIZ string ?)

For this case, Rust has a special module (std::ffi) that helps make the transion more easily by providing different C/C++ data types and methods.



FFI

1. Accessing a function from an external library (static linkage)

The following types are available via std::ffi:

C/C++ type	Rust type (aprox)	Rust vi std::ffi
char	i8	c_char
unsigned char	u8	c_uchar
short	i16	c_short
unsigned short	u16	c_ushort
int	i32	c_int
unsigned int	u32	c_uint
long	i32 or i64	c_long
unsigned long	u32 or u64	c_ulong
long long	i64	c_longlong
unsigned long long	u64	c_ulonglong
float	f32	c_float
double	f64	c_double

C/C++ type	Rust type (aprox)	Rust vi std::ffi
void	?	c_void ()
void*	?	*mut c_void
const void*	?	*const c_void
char* (ASCIIZ)	?	CString
const char*	&[u8]+\0 char	CStr *const c_char

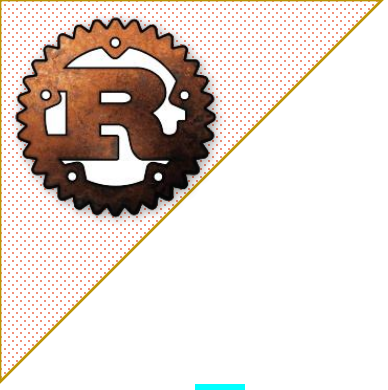
FFI

1. Accessing a function from an external library (static linkage)

Let’s try to create a file and write a text into it, by using Windows API. The C/C++ code will look like the following:

C++

```
#include "Windows.h"
#include <string.h>
void main() {
    HANDLE h = CreateFileA("test.txt", GENERIC_WRITE, FILE_SHARE_WRITE, NULL,
                                       CREATE_ALWAYS, 0, NULL);
    if (h != INVALID_HANDLE_VALUE) {
        const char* content = "some text";
        DWORD bytesWritten;
        WriteFile(h, content, strlen(content), &bytesWritten, NULL);
        CloseHandle(h);
    }
}
```



FFI

1. Accessing a function from an external library (static linkage)

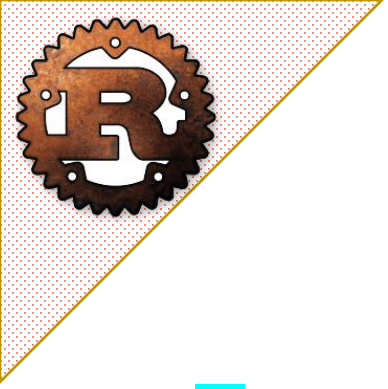
To translate the previous code into Rust, we first need to look on how the APIs used in that code look like:

API	Definition	Documentation

CreateFile	`HANDLE CreateFileA(`
	`  [in] LPCSTR lpFileName,`
	`  [in] DWORD dwDesiredAccess,`
	`  [in] DWORD dwShareMode,`
	`  [in, optional] LPSECURITY_ATTRIBUTES lpSecurityAttributes,`
	`  [in] DWORD dwCreationDisposition,`
	`  [in] DWORD dwFlagsAndAttributes,`
	`  [in, optional] HANDLE hTemplateFile`
	`);`
		https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilea

WriteFile	`BOOL WriteFile(`
	`  [in] HANDLE hFile,`
	`  [in] LPCVOID lpBuffer,`
	`  [in] DWORD nNumberOfBytesToWrite,`
	`  [out, optional] LPDWORD lpNumberOfBytesWritten,`
	`  [in, out, optional] LPOVERLAPPED lpOverlapped`
	`);`
		https://learn.microsoft.com/en-us/windows/win32/api/fileapi/nf-fileapi-writefile

CloseHandle	`BOOL CloseHandle( [in] HANDLE hObject );`	https://learn.microsoft.com/en-us/windows/win32/api/handleapi/nf-handleapi-closehandle



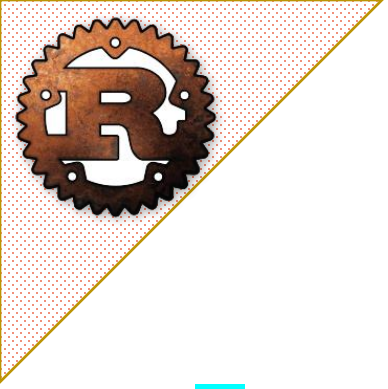
FFI

1. Accessing a function from an external library (static linkage)

First, let's define some constants and a type for HANDLE. We will also need std::ffi to convert a string to ASCIIZ and std::ptr to get the NULL value.

Rust (constants and modules)

```
use std::ffi::*;
use std::ptr;

type HANDLE = isize;
const GENERIC_WRITE: u32 = 0x40000000;
const FILE_SHARE_WRITE: u32 = 0x2;
const CREATE_ALWAYS: u32 = 2;
const INVALID_HANDLE_VALUE: HANDLE = -1;
```



FFI

1. Accessing a function from an external library (static linkage)

Secondly, let's import the 3 APIs (since all of them are located in kernel32.dll) we don't have to explicitly link it via #[link(name="kernel32")]

Rust (API inports)

```
extern "system" {
    fn CreateFileA(
        lpFileName: *const c_char,
        dwDesiredAccess: u32,
        dwShareMode: u32,
        lpSecurityAttributes: *mut c_void,
        dwCreationDisposition: u32,
        dwFlagsAndAttributes: u32,
        hTemplateFile: *mut c_void,
    ) -> HANDLE;

    fn WriteFile(
        hFile: HANDLE,
        lpBuffer: *const c_void,
        nNumberOfBytesToWrite: u32,
        lpNumberOfBytesWritten: *mut u32,
        lpOverlapped: *mut c_void,
    ) -> i32;

    fn CloseHandle(hObject: HANDLE) -> i32;
}
```

FFI

1. Accessing a function from an external library (static linkage)

Finaly, let's write the main function. Notice that we can use file_name.as_ptr() to get an ASCIIZ pointer as required by ***CreateFileA*** ABI.

Rust (main function)

```rust
fn main() {
    let file_name = CString::new("test.txt").unwrap();
    let handle = unsafe {
        CreateFileA(
            file_name.as_ptr(),
            GENERIC_WRITE,
            FILE_SHARE_WRITE,
            ptr::null_mut(),
            CREATE_ALWAYS,
            0,
            ptr::null_mut(),
        )
    };
    if handle != INVALID_HANDLE_VALUE {
        let content = "some text";
        let mut bytes_written: u32 = 0;
        unsafe {
            WriteFile(
                handle,
                content.as_ptr() as *const c_void,
                content.len() as u32,
                &mut bytes_written,
                ptr::null_mut(),
            );
            CloseHandle(handle);
        }
    }
}
```

FFI

1. Accessing a class/object from an external library (static linkage)

Let’s analyze a more complex scenario. Let’s assume that we want to access an object or class via FFI (for example, can we use a C++ class in Rust) ?

There are also two solutions here:

1. Build a C-language wrapper and access the elements from that class via that wrapper
2. Call the methods (ctor, dtor, other methods) directly.

FFI

1. Accessing a class/object from an external library (static linkage)

Let's assume the following C++ class:

C++

```
#include <string.h>

class Point {
    char* name;
public:
    int x, y;
    Point(const char* n) {
        x = y = 0;
        name = new char[strlen(n) + 1];
        memcpy(name, n, strlen(n) + 1);
    }
    ~Point() { delete []name; }
    bool is_origin() { return (x == 0) && (y == 0); }
    const char* get_name() { return name; }
};
```

FFI

1. Accessing a class/object from an external library (static linkage)

To export the class via a C wrapper we also need to add the following code. Notice that we need to create a method for ctor, dtor, and some methods to access public data members x and y.

C++

```
extern "C"
{
    __declspec(dllexport) void* new_Point(const char *name) { return new Point(name); }
    __declspec(dllexport) void delete_Point(void *instance) { delete ((Point *)instance); }
    __declspec(dllexport) bool point_is_origin(void *instance)
    {
        return ((Point *)instance)->is_origin();
    }
    __declspec(dllexport) int point_get_x(void *instance) { return ((Point *)instance)->x; }
    __declspec(dllexport) int point_get_y(void *instance) { return ((Point *)instance)->y; }
    __declspec(dllexport) const char *point_get_name(void *instance)
    {
        return ((Point *)instance)->get_name();
    }
}
```

FFI

1. Accessing a class/object from an external library (static linkage)

To export the class via a C wrapper we also need to add the following code. Notice that we need to create a method for ctor, dtor, and some methods to access public data members x and y.

C++

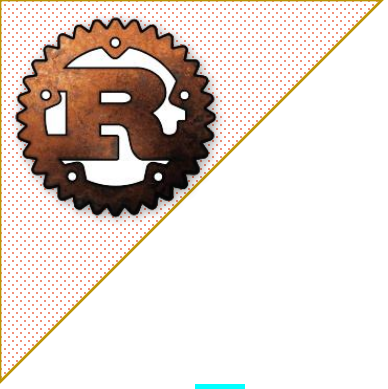
```
extern "C"
{
    __declspec(dllexport) void* new_Point(const char *name) { return new Point(
    __declspec(dllexport) void delete_Point(void *instance) { delete ((Point *)
    __declspec(dllexport) bool point_is_origin(void *instance)
    {
        return ((Point *)instance)->is_origin();
    }
    __declspec(dllexport) int
    __declspec(dllexport)
    __declspec(dllexport) cons
    {
        return ((Point *)instance)->get_name();
    }
}
```

This tells the compiler that the export should be in a “C” format (more exactly just the name of the function).

This tells the compiler that the function should be exported

The resulted DLL will export:

- **new_Point**
- **delete_Point**
- **point_is_origin**
- **point_get_x**
- **point_get_y**
- **point_get_name**



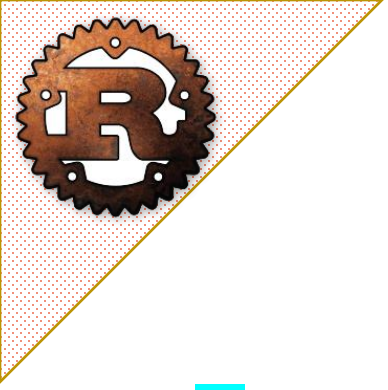
FFI

1. Accessing a class/object from an external library (static linkage)

Now for the Rust part → first we need to import the exported method. We will assume that the C++ code results in a DLL file called mydllfile.dll

Rust

```
use std::ffi::*;
use std::ptr;

#[link(name="mydllfile")]
extern "C" {
    fn new_Point(name: *const c_char)->*mut c_void;
    fn delete_Point(instance: *mut c_void);
    fn point_is_origin(instance: *mut c_void)->bool;
    fn point_get_x(instance: *mut c_void)->i32;
    fn point_get_y(instance: *mut c_void)->i32;
    fn point_get_name(instance: *mut c_void)->&str;
}
```



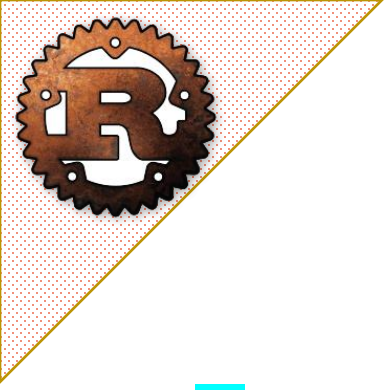
FFI

1. Accessing a class/object from an external library (static linkage)

Secondly, we need to write a Rust class that will resemble the C++ one.

Rust

```rust
pub struct Point {
    instance: *mut c_void
}

impl Point {
    fn new(name: &str)->Self {
        let c_name = CString::new(name).unwrap();
        Self { instance : unsafe { new_Point(c_name.as_ptr())} }
    }
    fn get_x(&self)->i32 { unsafe { point_get_x(self.instance) } }
    fn get_y(&self)->i32 { unsafe { point_get_y(self.instance) } }
    fn is_origin(&self)->bool { unsafe { point_is_origin(self.instance) } }
    fn get_name(&self)->&str {...}
}

impl Drop for Point {
    fn drop(&mut self) { unsafe { delete_Point(self.instance); } }
}
```



FFI

1. Accessing a class/object from an external library (static linkage)

Secondly, we need to write a Rust class that will resemble the C++ one.

Rust

```
pub struct Point {
    instance: *mut c_void
}

impl Point {
    fn new(name: &str)->Self {
        let c_name = CString::new(name).unwrap();
        Self { instance : unsafe { new_Point(c_name.as_ptr())} }
    }
    fn get_name(&self)->&str {
        unsafe { let c_buf = get_name(self.instance); ce) } }
        unsafe { let c_buf = get_name(self.instance); ce) } }
        unsafe { let c_buf = get_name(self.instance); lf.instance) } }
    }
}

impl Drop for Point {
    fn drop(&mut self) { unsafe { delete_Point(self.instance); } }
}
```

Notice the implementation of Drop trait. We need this to be able to call the C++ destructor for class Point (that will deallocate the memory allocated for the name field).

FFI

1. Accessing a class/object from an external library (static linkage)

Secondly, we need to write a Rust class that will resemble the C++ one.

Rust

```
pub struct Point {
    instance: *mut c_void
}

impl Point {
    fn new(name: &str)->Self {
        let c_name = CString::new(name).unwrap();
        Self { instance : unsafe { new_Po
    }
    fn get_x(&self)->i32 { unsafe { point
    fn get_y(&self)->i32 { unsafe { point
    fn is_origin(&self)->bool { unsafe
    fn get_name(&self)->&str {...}
}

impl Drop for Point {
    fn drop(&mut self) { unsafe { delete_
}
```

```
fn get_name(&self)->&str {
    unsafe {
        let c_str = point_get_name(self.instance);
        if c_str.is_null() { panic!("... some error ..."); }
        if let Ok(name) = CStr::from_ptr(c_str).to_str() {
            return name;
        }
        panic!("Conversion from const char* to &str failed !");
    }
}
```

FFI

1. Accessing a class/object from an external library (static linkage)

This method comes with pros and cons:

PROS:

- **Every type** of object can be exported in this way
- **Virtual methods can be called** as the call is being performed in the C wrapper

CONS:

- You need to **write a lot of code** and wrappers to make this work
- **Slower** (you can not call the method from a class directly; you need to go through a wrapper)
- You **can not access public data members**; instead, getters and setters need to be created.
- You **can not create such an object directly on the stack**. The wrapper forces you to create every object on the heap. For smaller object this could be a disadvantage.

FFI

1. Accessing a class/object from an external library (static linkage)

The second option is to try to create a class with the same memory layout as the on from C/C++ and when a method is being called, call the exported method from C++ code. Since Rust optimizes the calls that perform one single call within their code, this will achieve the same level of performance as with C/C++.

On the other hand, it is more complicated to link.

FFI

1. Accessing a class/object from an external library (static linkage)

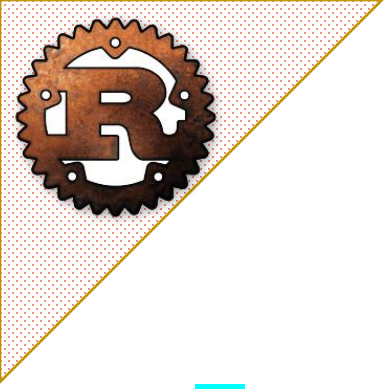
First, lets export the Point class (notice that we no longer use the extern “C”) format;
The __declspec(dllexport) are still required for MS compiler to export this class.

C++

```
class __declspec(dllexport) Point { … };
```

The resulted DLL will have the symbols mangled as follows:

Symbol (method)	Exported name (mangled) for MS cl.exe compiler
Point::Point(char const*)	??0Point@@QEAA@PEBD@Z
Point::~Point()	??1Point@@QEAA@XZ
Point& Point::operator=(Point const&)	??4Point@@QEAAAEAV0@AEBV0@@Z
void Point::__autoclassinit2(unsigned long long)	?__autoclassinit2@Point@@QEAAX_K@Z
char const* Point::get_name()	?get_name@Point@@QEAAPEBDXZ
bool Point::is_origin()	?is_origin@Point@@QEAA_NXZ



FFI

1. Accessing a class/object from an external library (static linkage)

First, lets export the Point class (notice that we no longer use the extern “C”) format; The __declspec(dllexport) are still required for MS compiler to export this class.

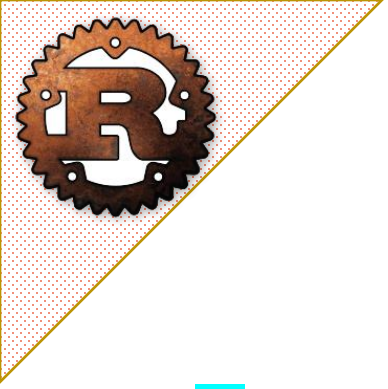
C++

```
class __declspec(dllexport) Point { … };
```

The resulted DLL will have the symbols mangled as follows:

Symbol (method)	Exported name (mangled) for MS cl.exe compiler
Point::Point(char const*)	??0Point@@QEAA@PEBD@Z
Point::~Point()	
Point& Point::operator=(Point const&)	
void Point::__autoclassinit2(unsigned long l	
char const* Point::get_name()	?get_name@Point@@QEAAPEBDXZ
bool Point::is_origin()	?is_origin@Point@@QEAA_NXZ

This is a **copy constructor** that was automatically generated by the compiler. Since it was automatically generated, it performs bitwise copy over the content of the object.



FFI

1. Accessing a class/object from an external library (static linkage)

First, lets export the Point class (notice that we no longer use the extern “C”) format; The __declspec(dllexport) are still required for MS compiler to export this class.

C++

```
class __declspec(dllexport) Point { … };
```

The resulted DLL will have the symbols mangled as follows:

Symbol (method)	Exported name (mangled) for MS cl.exe compiler
Point::Point(char const*)	??0Point@@QEAA@PEBD@Z
Point::~Point()	??1Point@@QEAA@XZ
Point& Point::operator=(Point const&)	
void Point::__autoclassinit2(unsigned long long)	
char const* Point::get_name()	
bool Point::is_origin()	?is_origin@Point@@QEAA_NXZ

This is a method used to initialized the pointer in the class Point (for the name) to a NULL pointer (often used to create static variables).

FFI

1. Accessing a class/object from an external library (static linkage)

Just like in the previous case, we need to import the exported files from mydllfile.dll. We can skip some of them (for example if we don't want to allow a Pointer object to be copied (so only MOVE semantics), we don't really need the copy constructor).

Rust

```rust
use std::ffi::*;
use std::ptr;
#[link(name = "mydllfile")]
extern "C" {
    #[link_name = "??0Point@@QEAA@PEBD@Z"]
    fn point_ctor(instance: *mut Point, name: *const c_char);
    #[link_name = "??1Point@@QEAA@XZ"]
    fn point_dtor(instance: *mut Point);
    #[link_name = "?is_origin@Point@@QEAA_NXZ"]
    fn point_is_origin(instance: *const Point) -> bool;
    #[link_name = "?get_name@Point@@QEAAPEBDXZ"]
    fn point_get_name(instance: *const Point) -> *const c_char;
}
```

FFI

1. Accessing a class/object from an external library (static linkage)

The Rust class is similar (as form) with the one from C++:

Rust

```rust
#[repr(C)]
pub struct Point {
    name: *mut c_char,
    pub x: i32,
    pub y: i32,
}
impl Point {
    fn new(name: &str) -> Self {
        unsafe {
            let c_name = CString::new(name).unwrap();
            let mut memory: MaybeUninit<Point> = std::mem::MaybeUninit::uninit();
            point_ctor(memory.as_mut_ptr(), c_name.as_ptr());
            return memory.assume_init();
        }
    }
    fn is_origin(&self) -> bool { unsafe { point_is_origin(self) } }
    fn get_name(&self) -> &str {…}
}
```

```rust
impl Drop for Point {
    fn drop(&mut self) {
        unsafe {
            point_dtor(self);
        }
    }
}
```

FFI

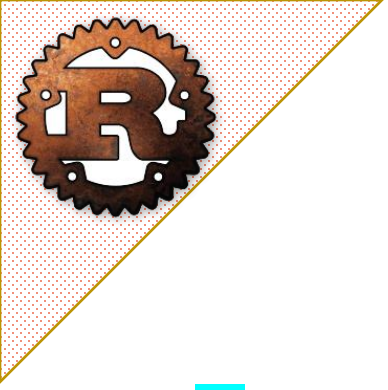
1. Accessing a class/object from an external library (static linkage)

The Rust class is similar (as form) with the one from C++:

Rust

This is required so that we are certain that the fields are allocated in the same order in memory in Rust and C++ for structure Point.

```
#[repr(C)]
pub struct Poi
    name: *mut c_char,
    pub x: i32,
    pub y: i32,
}
impl Point {
    fn new(name: &str) -> Self {
        unsafe {
            let c_name = CString::new(name).unwrap();
            let mut memory: MaybeUninit<Point> = std::mem::MaybeUninit::uninit();
            point_ctor(memory.as_mut_ptr(), c_name.as_ptr());
            return memory.assume_init();
        }
    }
    fn is_origin(&self) -> bool { unsafe { point_is_origin(self) } }
    fn get_name(&self) -> &str {…}
}
```



FFI

1. Accessing a class/object from an external library (static linkage)

The Rust class is similar (as form) with the one from C++:

Rust

```
#[repr(C)]
pub struct Point {
    name: *mut c_char,
    pub x: i32,
    pub y: i32,
}
impl Point {
    fn new(name: &str) -> Sel
        unsafe {
            let c_name = CString::new(name).unwrap();
            let mut memory: MaybeUninit<Point> = std::mem::MaybeUninit::uninit();
            point_ctor(memory.as_mut_ptr(), c_name.as_ptr());
            return memory.assume_init();
        }
    }
    fn is_origin(&self) -> bool { unsafe { point_is_origin(self) } }
    fn get_name(&self) -> &str {…}
}
```

Allocates an **uninialized** memory zone of the same size of structure Point.

FFI

1. Accessing a class/object from an external library (static linkage)

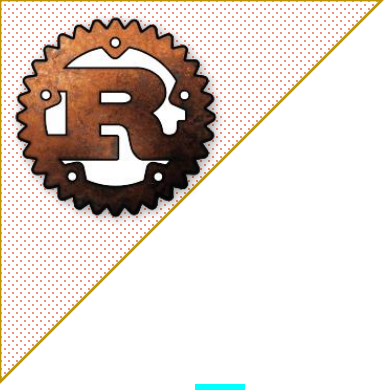
The Rust class is similar (as form) with the one from C++:

Rust

```
#[repr(C)]
pub struct Point {
    name: *mut c_char,
    pub x: i32,
    pub y: i
}
impl Point
    fn new(
        unsafe {
            let c_name = CString::
            let mut memory: MaybeUninit<Point> = std::mem::MaybeUninit::uninit();
            point_ctor(memory.as_mut_ptr(), c_name.as_ptr());
            return memory.assume_init();
        }
    }
    fn is_origin(&self) -> bool { unsafe { point_is_origin(self) } }
    fn get_name(&self) -> &str {…}
}
```

This calls the Point constructor from C++ (and provides the memory where the data should be stored and the name of the point).

This call will use RCX/ECX to pass the memory from memory.as_mut_ptr() to Point ctor. For x86 use “thiscall” when linking the ctor with Rust >=1.73



FFI

1. Accessing a class/object from an external library (static linkage)

The Rust class is similar (as form) with the one from C++:

Rust

```
#[repr(C)]
pub struct Point {
    name: *mut c_char,
    pub x: i32,
    pub y: i32,
}
imp
            let c_name = CS    ng::new(name).unwrap();
            let mut memory     beUninit<Point> = std::mem::MaybeUninit::uninit();
            point_ctor(memor    .as_mut_ptr(), c_name.as_ptr());
            return memory.assume_init();
        }
    }
    fn is_origin(&self) -> bool { unsafe { point_is_origin(self) } }
    fn get_name(&self) -> &str {…}
}
```

This tells the Rust compiler that the memory zone was initialed and as such it can return an object of type Point safely.

FFI

1. Accessing a class/object from an external library (static linkage)

The Rust class is similar (as form) with the one from C++:

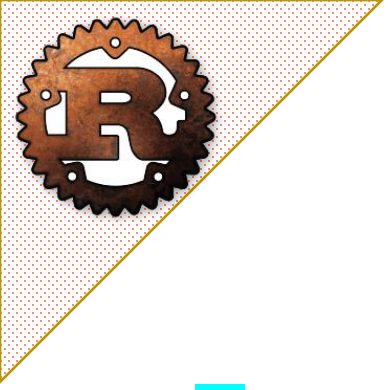
Rust

```
#[repr(C)]
pub struct Point {
    name: *mut c_char,
    pub x: i32,
    pub y: i32,
}
impl Point {
    fn new(
        unsa
                point_ctor(memory.as_mut_ptr(), c_na      _ptr());
            return memory.assume_init();
        }
    }
    fn is_origin(&self) -> bool { unsafe { point_is_origin(self) } }
    fn get_name(&self) -> &str {…}
}
```

Keep in mind that we have imported point_is_origin as follows:

```
fn point_is_origin(instance: *const Point) -> bool;
```

This translates that we need to provide a pointer to an object of type Point. This allows us to easily covert &self to a const pointer.



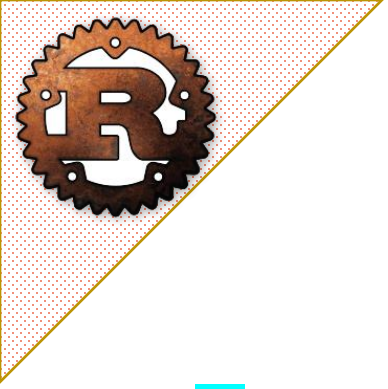
FFI

1. Accessing a class/object from an external library (static linkage)

The Rust class is similar (as form) with the one from C++:

Rust

```
#[repr(C)]
pub struct Point {
    name: *mut c_char,
    pub x: i32,
    pub y: i32,
}

fn get_name(&self) -> &str {
    unsafe {
        let c_str = point_get_name(self);
        if c_str.is_null() {
            panic!("... some error ...");
        }
        return CStr::from_ptr(c_str).to_str().expect("Fail to convert");
    }
}

impl Point {
    pub fn new() -> Point {
        unsafe {
            let mut memory = uninit::uninit();
            point_init(memory.as_mut_ptr());
            return memory.assume_init();
        }
    }
    fn is_origin(&self) -> bool { unsafe { point_is_origin(self) } }
    fn get_name(&self) -> &str {…}
}
```



FFI

1. Accessing a class/object from an external library (static linkage)

This second method also comes with some PROS and CONST

PROS:

- **Faster**; it works just like C++ code in terms of calling (no wrappers).
- You **can access public data members** just like in C++ case
- **Can be created on the stack**

CONS:

- **Mangling is different between various C++ compilers**. As such, you need specific code for cases where OS interoperability is required.
- **Calling virtual functions is tricky** as you would have to handle calling them directly instead of the C++ wrapper.
- **Adding a copy-ctor that has a different behavior makes things more complicated** as you will have to mimic that behavior in an environment where COPY semantics means bitwise copy.

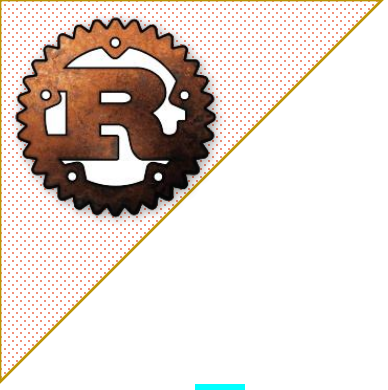
FFI

1. Accessing a class/object from an external library (static linkage)

What about exceptions ?

Or various un-secure behavior that can be encountered in C++ language (for example throwing an exception while in constructor body).

As a general note, no exception should be sent to Rust, so we will have to identify them at the C++ level and treat them accordingly.



FFI

1. Accessing a class/object from an external library (static linkage)

Let's analyze the original C++ class Point:

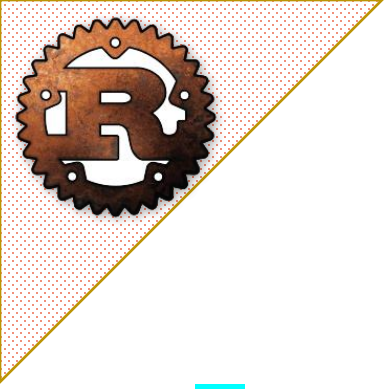
C++

```
#include <string.h>
class Point {
    Point(const char* n) {
        x = y = 0;
        name = new char[strlen(n) + 1];
        memcpy(name, n, strlen(n) + 1);
    }
};
```

This allocation might throw an error if the required memory can not be allocated.

In this case we have 2 options:

1. Use noexcept to stop the program at this point. The disadvantage is that we can not control the flow in this case
2. Use std::nothrow and change the logic in the Rust code.



FFI

1. Accessing a class/object from an external library (static linkage)

The second solution implies the following changes:

C++

```
#include <string.h>
#include <new>

class Point {
…
    Point(const char* n) {
        x = y = 0;
        name = new (std::nothrow) char[strlen(n) + 1];
        if (name != nullptr) {
            memcpy(name, n, strlen(n) + 1);
        }
    }
…
};
```

This implies that if no memory can be allocated for data member **name**, then the data member name will be **NULL**

FFI

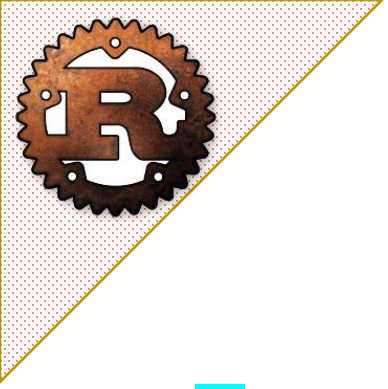
1. Accessing a class/object from an external library (static linkage)

On the Rust side, the Point::new(…) method has to be changed as well to return an Option<Point>. This way, if an allocation error is triggered on the C++ side, we can handle it on the Rust side by not returning an object.

Rust

```rust
#[repr(C)]
pub struct Point {…}
impl Point {
    fn new(name: &str) -> Option<Self> {
        unsafe {
            let c_name = CString::new(name).unwrap();
            let mut memory: MaybeUninit<Point> = std::mem::MaybeUninit::uninit();
            point_ctor(memory.as_mut_ptr(), c_name.as_ptr());
            let point = memory.assume_init();
            if point.name.is_null() {
                return None;
            }
            return Some(point);
        }
    }
…
}
```

If the name pointer is NULL, we discard the object and return None. If not, we return Some(Point) and the user can access it.



FFI

1. Accessing a class/object from an external library (static linkage)

As a general observation, importing an object from an external library is a tedious job and implies a lot of knowledge on how compilers work internally.

To make things easier, Rust also provides an internal tool called bindgen that can be used to automatically generate the code needed to link an external library to a Rust program.

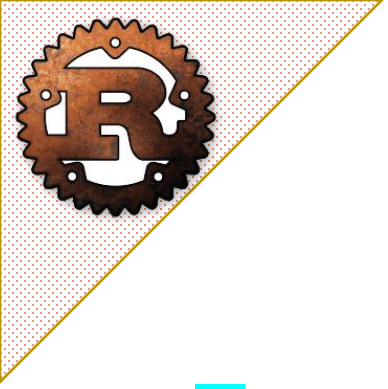
More details on: https://rust-lang.github.io/rust-bindgen/

FFI

1. Accessing a class/object from an external library (static linkage)

It’s also important to notice that **polymorphism** is quite complicated to achieve using object generated from another library. The main reason is that the concept of virtual functions and tables are treated differently in Rust than in C++ (for example).

It is not without a solution, but the solution implies to actually make the calls yourself and understand how the virtual table looks like for different C++ compilers and versions.



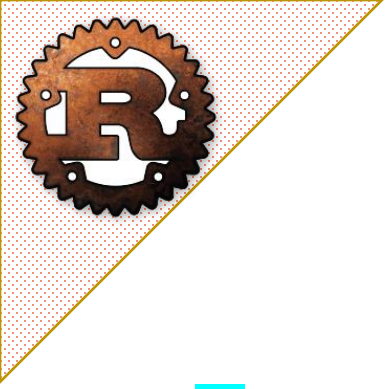
FFI

1. Accessing a function from an external library (dynamic linkage)

The other scenario is when we try to dynamically load a library during runtime (this is useful for cases where your program supports plugins).

Let’s consider the following example:

- We want to create a Rust program that receives integer values two values and the name of a plugin that can perform a mathematical operation using those values
- Rust will try to load that plugin and assume that a specific method (let’s call it perform_op is exported, and if so, calls that method with those two values received and prints the result).
- The interesting thing is that the plugin does not have to be written in Rust (but can be written in other languages as well – such as C++).



FFI

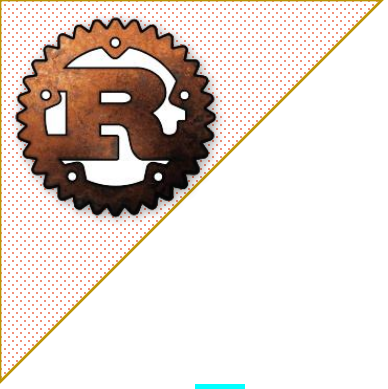
1. Accessing a function from an external library (dynamic linkage)

To do this we need to functionalities:

- A way to load a library given its name / path on the disk
- A way to identify the location in memory of a function exported by the loaded library

These functionalities are different from one OS to another as presented in the next table.

	Windows (API)	Linux (API)	MAC(OSX)
Loading a library given a name/path	LoadLibraryA or LoadLibraryW	dlopen	dlopen
Locating an exported function	GetProcAddress	dlsym	dlsym



FFI

1. Accessing a function from an external library (dynamic linkage)

Let's see how LoadLibraryA and GetProcAddress are defined in Windows API:

1. LoadLibraryA

- Definition: `HMODULE LoadLibraryA([in] LPCSTR lpLibFileName);`
- Documentation: https://learn.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-loadlibrarya

2. GetProcAddress

- Definition:
```
FARPROC GetProcAddress([in] HMODULE hModule,
                       [in] LPCSTR  lpProcName);
```
- Documentation: https://learn.microsoft.com/en-us/windows/win32/api/libloaderapi/nf-libloaderapi-getprocaddress

FFI

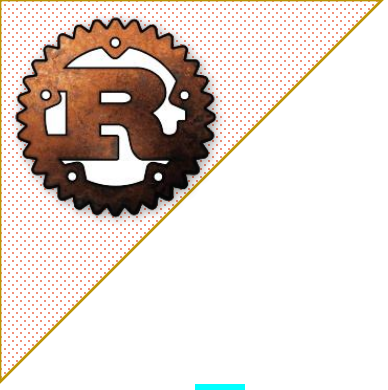
1. Accessing a function from an external library (dynamic linkage)

Let’s start by creating the import code (for Windows):

Rust

```rust
use std::ffi::*;

type HMODULE = *const c_void;
type FARPROC = *const c_void;

#[link(name = "kernel32")]
extern "system" {
    fn LoadLibraryA(lpFileName: *const c_char) -> HMODULE;
    fn GetProcAddress(hModule: HMODULE, lpProcName: *const c_char) -> FARPROC;
}
```

Notice that the result of ***GetProcAddress*** is a raw pointer (we will need to convert it to a pointer to a function to use it in Rust).



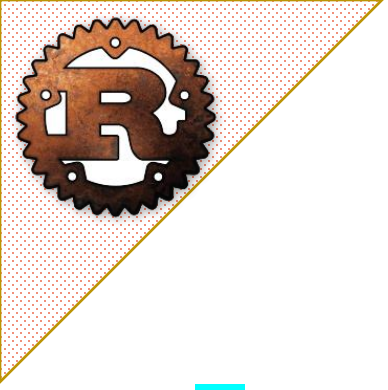
FFI

1. Accessing a function from an external library (dynamic linkage)

Secondly, we need a Rust function that can load a plugin and returns a pointer to a function (provided that function is exported from the loaded plugin). Our function will have the following signature: `fn(i32,i32)->i32`

Rust

```rust
type PLUGIN_FUNCTION = extern "C" fn(i32,i32)->i32;
fn load_plugin(library: &str, function: &str) -> Option<PLUGIN_FUNCTION>
{
    unsafe {
        let library_cstr = std::ffi::CString::new(library).unwrap();
        let library_handle = LoadLibraryA(library_cstr.as_ptr());
        if library_handle.is_null() { return None; }

        let function_cstr = std::ffi::CString::new(function).unwrap();
        let function_ptr = GetProcAddress(library_handle, function_cstr.as_ptr());
        if function_ptr.is_null() { return None; }

        Some(std::mem::transmute_copy(&function_ptr))
    }
}
```



FFI

1. Accessing a function from an external library (dynamic linkage)

Secondly, we need a Rust function that can load a plugin and returns a pointer to a function (provided that function is exported from the loaded plugin). Our function will have the following signature: `fn(i32,i32)->i32`

Rust

```
type PLUGIN_FUNCTION = extern "C" fn(i32,i32)->i32;
fn load_plugin(library: &str, function: &str) -> Option<PLUGIN_FUNCTION>
{
    unsafe {
        let library_cstr = std::ffi::CString::new(library).unwrap();
        let function_cstr = std::ffi::CString::new(function).unwrap();
        let function_ptr = GetProcAddress(library_handle, function_cstr.as_ptr());
        if function_ptr.is_null() { return None; }

        Some(std::mem::transmute_copy(&function_ptr))
    }
}
```

We need this to convert a raw pointer to a function pointer that we can use in a normal Rust Code.

FFI

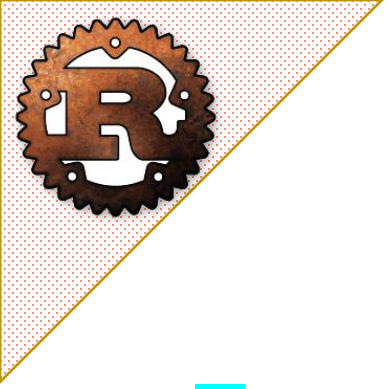
1. Accessing a function from an external library (dynamic linkage)

Now, let's write a plugin (in C++). We will create something that computes the sum of two numbers:

C/C++

```
extern "C" {
    int __declspec(dllexport) operation(int x, int y) {
        return x + y;
    }
}
```

The plugin exports one function (operation). Let's also assume that the plugin is called my_plugin.dll



FFI

1. Accessing a function from an external library (dynamic linkage)

And the code from Rust that puts all of this together:

Rust

```
fn main() {
    if let Some(operation) = load_plugin("<path to..>\\my_plugin.dll", "operation") {
        let result = operation(10,20);
        println!("result = {}",result)
    } else {
        println!("Fail to load plugin or missing 'operation' export !");
    }
}
```

Output

```
result = 30
```

Keep in mind that this code is designed for Windows. For Linux/MacOSX you will need to modify the list of imports (use **dlopen** instead of **LoadLibraryA**) and recompile the C++ plugin with gcc or clang.

FFI

1. Accessing a class/object from an external library (dynamic linkage)

To access an object/class via dynamic linkage, the steps are similar to the ones used to access a function.

- You will need to use GetProcAddress / LoadLibrary as well
- However, when using GetProcAddress, you will load a mangles export/symbol instead of a name for a function
- You will also need to store that raw pointer to be able to access it later.

The rest of the steps are similar (creating a wrapper class in Rust over the C++ object) and calling the methods obtained in the previous step.

FFI

2. Exporting a function to be used by an external library

To export a library from Rust, there are a couple of things that need to be performed on cargo.toml (to explain the Rust compiler that we need to create a library and not an executable).

cargo.toml

```
[package]
name = "my_math_lib"
version = "0.1.0"
edition = "2021"

[dependencies]
…

[lib]
crate-type = ["cdylib"]
```

The first step is to add a [lib] section and specify that the crate type (cdylib [**C** **dy**namic **Lib**rary])

Crate-type supports other types as well:

- cdylib
- dylib
- rlib
- staticlib
- proc-macro
- …

FFI

2. Exporting a function to be used by an external library

We will also need to change the way we write our exported functions:

Rust

```rust
#[no_mangle]
pub extern "C" fn add(x: i32, y: i32) -> i32 {
    x + y
}
```

1. The “#[no_mangle]” attribute is required as Rust mangles symbols and as such other applications can’t use them.
2. The “extern "C" “ specifier is required so that Rust exports that function in a way a program like C/C++ can understand.

FFI

2. Exporting a function to be used by an external library

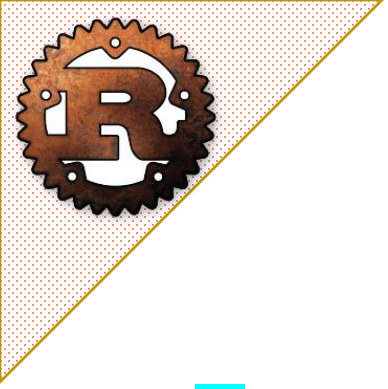
Upon compiling, a .dll or a .so or a .dylib will be created that export the function add.

That library can be used in a C/C++ program in the following way:

C/C++ program (for Windows)

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

typedef int32_t (* FNADD)(int32_t,int32_t);
void main() {
    auto handle = LoadLibraryA("my_math_lib.dll");
    auto add = (FNADD)GetProcAddress(handle,"add");
    printf("%d",add(1,2));
}
```

OBS: *We consider my_math_lib.dll the binary result obtain when running cargo build.*



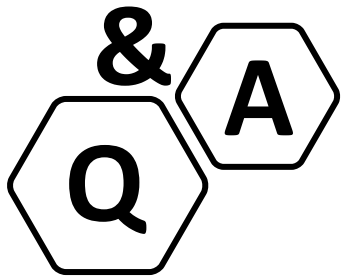
FFI

2. Exporting a function to be used by an external library

The same logic for exporting object can be applied here as well, but we will need an equivalent code on the C++ part that loads and manages objects writtin in Rust.

And , just like in the case of C++ to Rust, there is a helper library called **cbindgen** that can help converting Rust structures to C++:

Link: https://github.com/mozilla/cbindgen


&
Q
A