MODULE 108

PROGRAMMATION RUST AVANCÉE

PLAN

- Macros
- Rust unsafe
- Utiliser C/C++ en Rust
- Utiliser Rust en C/C++

MACROS

INTRODUCTION

Exemples of macros

```rust
#[test] // 👈 here
fn test_function() {
    assert!(true); // 👈 here
    println!("Hello!"); // 👈 here
}

#[derive(Debug)]  // 👈 here
struct MyData {}
```

TYPES DE MACRO

- Macros déclaratives
- Macros procédurales

La différence est dans le mode de définition.

MACROS EN RUST

- Outil de méta-programmation
- Manipulation des tokens, pas du texte

MACROS EN RUST - TOKENS

```rust
#[derive(Debug)]
struct MyData {}
```

Tokens:

```
[
    StructKeyword,
    Identifier(Value),
    BlockOpening,
    BlockEnding
]
```

MACROS DÉCLARATIVES

MACROS DÉCLARATIVES - INTRO

- le plus simple,
- Pattern matching

Exemple avec `vec!`:

```
let v: Vec<u32> = vec![1, 2, 3];
```

MACROS DÉCLARATIVES - ANATOMIE 1/5

```
#[macro_export]
macro_rules! vec {
    [ $( $x:expr ),* ] => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

MACROS DÉCLARATIVES - ANATOMIE 2/5

```
#[macro_export] // 👉 export from this lib
// 👇 declare macro with pattern matching rules
macro_rules! vec {
//           ^^^ macro name
    [ $( $x:expr ),* ] => { };
}
```

MACROS DÉCLARATIVES - ANATOMIE 3/5

```
#[macro_export]
macro_rules! vec {
    [ $( $x:expr ),* ] => {
//  ^^^^^^^^^^^^^^^^^^ pattern of tokens to match
//          to be replaced by 👇
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

MACROS DÉCLARATIVES - ANATOMIE 4/5

```
    [ $( $x:expr ),* ] => {
//  ^                 ^ begins and ends with [ ]
    [ $( $x:expr ),* ] => {
//    ^^~~~~~~~~~^^^ repeats the pattern 0-n times
//                  use , to separate
    [ $( $x:expr ),* ] => {
//       ^^^^^^^ captures an expression, named $x
```

MACROS DÉCLARATIVES - ANATOMIE 5/5

```
#[macro_export]
macro_rules! vec {
    [ $( $x:expr ),* ] => {
        // 👇 template of code to be interpolated
        {
            let mut temp_vec = Vec::new();
            $( // start repetition
                temp_vec.push($x);
                //           ^^ use captured $x
            )* // end repetition
            temp_vec
        }
    };
}
```

EXERCICE

EXERCICE 1 - MACROS DÉCLARATIVES

MACROS PROCÉDURALES

MACROS PROCÉDURALES - INTRO

- Fonction Rust
  - exécutée par le compilateur
  - prend en entré un stream de tokens
  - sort un autre stream
  - code arbitraire executé à la compilation

```
pub fn my_macro(input: TokenStream) -> TokenStream {
    /* rewrite stuff */
}
```

MACROS PROCÉDURALES - USAGES

Dérivation automatique de traits:

```
#[derive(MyTrait)]
struct MyStruct {}
```

Attributs, sur des fonctions, types, etc:

```
#[test]
fn my_little_test() {}
```

Comme des fonctions:

```
format!("{}", 42)
```

MACROS PROCÉDURALES - DÉFINITION 1/2

Définies dans une lib. spécifiquement marquée dans `Cargo.toml`

```
[lib]
proc-macro = true
```

MACROS PROCÉDURALES - DÉFINITION 2/2

Un peu d'aide

- proc_macro, définie `TokenStream`,
- proc-macro2, wrappers et autres facilitateurs,
- quote, pour du templating de code
- syn, pour parser un stream comme un AST

EXERCICE

EXERCICE 2 - MACROS PROCÉDURALES

RUST UNSAFE

RUST UNSAFE - INTRODUCTION

- Documentation spécifique: Rust-nomicon
- La majorité du code peut (devrait) être safe
- impose au programmeur d'assurer lui-même les garanties que Rust fournie normalement dans la partie safe

RUST UNSAFE - POSSIBILITÉS

- Pointeurs nus (`*const T`, `*mut T`),
- Utiliser des fonctions `unsafe`,
- Implémenter des traits `unsafe`,
- Muter des variables `static`.

RUST UNSAFE - USAGE 1/3

sur un bloc:

```rust
// 👇 function itself is safe
fn index(idx: usize, arr: &[u8]) -> Option<u8> {
    if idx < arr.len() {
        unsafe { // 👈 unsafe block
            Some(*arr.get_unchecked(idx))
        }
    } else {
        None
    }
}
```

RUST UNSAFE - USAGE 2/3

sur une fonction:

```
   unsafe fn do_dangerous_things() {/* -- snip -- */}
// ^^^^^^ whole function is unsafe
```

RUST UNSAFE - USAGE 3/3

sur un trait

```
unsafe impl Sync for MyBox {}
```

RUST UNSAFE - INTERDICTION

Une librairie peut interdire l'utilisation de code `unsafe`:

```
#![forbid(unsafe_code)]
```

UTILISER C/C++ DEPUIS RUST

BINDGEN

- documentation de bindgen
- génère des bindings rust depuis des en-tête .h

build.rs

Rust permet l'écriture d'un "script" Rust de build du projet qui s'exécute avant le build normal:

```
myproject/
+-> Cargo.toml
+-> build.rs
+-> src/
    +-> ...
```

Dans `Cargo.toml`:

```
[package]
build = "build.rs"
```

APPROCHE GÉNÉRALE

- Créer un paquet Rust servant de wrapper pour la librairie C/C++,
- Générer automatiquement les bindings au moment du build,
- Exposer une API "Rust-ish".

EXERCICE

EXERCICE 3 - UTILISER C/C++ DEPUIS RUST

UTILISER RUST DEPUIS C/C++

PRÉPARER LES STRUCTURES RUST

```rust
// 👇 Use a C-compatible representation
#[repr(C)]
pub struct MyStruct { }
```

PRÉPARER LES FONCTIONS RUST

```rust
#[no_mangle] // 👉 Export the function with this name exactly
// 👇 Export under a different name
#[export_name = "my_function"]
pub fn my_function() {}
```

EXERCICE

EXERCICE 4 - UTILISER UNE LIBRAIRIE RUST DANS UN PROJET C