MODULE 106
ENVIRONNEMENT

PLAN

- Platforme et chaîne d'outil
- Système de build --cargo
- Organisation du code
- Tests
- Documentation

PLATFORME ET CHAÎNE D'OUTIL

PLATEFORMES 1/2

Les plateformes cibles pour Rust sont identifiées par un *target triple* indiquant l'architecture, le système, etc.

- `x86_64-pc-windows-msvc`: 64-bit MSVC (Windows 7+)
- `x86_64-unknown-linux-gnu`: 64-bit Linux (2.6.18+)

PLATEFORMES 2/2

List complète des plateformes avec différents degrés de support :

- `std` : la librairie standard est disponible
- `rustc` un compilateur natif pour la plateforme
- `cargo` un système de build natif pour le plateforme

La cross-compilation est toujours possible dans les autres cas.

CHAÎNE D'OUTIL

Une chaîne d'outil Rust contient:

- `rustc`: le compilateur
- `rust-std`: la librairie standard
- `cargo`: gestionnaire de paquet et système de build
- `rust-docs` documentations associées
- ... d'autres composants incluant `rustfmt`, `clippy`, etc.

Une chaîne d'outil peut cibler plusieurs plateformes.

CHAÎNE D'OUTIL -- INSTALLATION

`rustup`:

- gestion des chaînes d'outils
- gestion des plateformes cible
- gestion des composants additionnels

Installer `rustup` depuis https://rustup.rs/.

CHAÎNE D'OUTIL -- INSTALLATION

Ajouter une nouvelle chaîne d'outil:

```
rustup toolchain install stable-x86_64-pc-windows-msvc
```

Ajouter ue cible supplémentaire:

```
rustup target add arm-linux-androideabi
rustup target add --toolchain stable-x86_64-pc-windows-msvc\
    arm-linux-androideabi
```

SYSTÈME DE BUILD -- CARGO

SYSTÈME DE BUILD

Compiler avec `rustc`.

Exemple d'un fichier `main.rs`:

```rust
fn main() {
    println!("Hello, world!");
}
```

```
# compile and execute
rustc main.rs
./main
```

EXERCICE

EXERCICE 1 - COMPILER AVEC RUSTC

SYSTÈME DE BUILD

Créer un projet avec `cargo` (build system et gestionnaire de paquet).

```
# create a new project
cargo new cargo_hello
# build
cd cargo_hello
cargo build
# execute
./target/debug/cargo_hello
# or
cargo run
```

EXERCICE

EXERCICE 2 - COMPILER AVEC CARGO

STRUCTURE PROJET CARGO 1/2

- `Cargo.toml`: Manifeste et configuration du projet, incluant les métadonnées et les dépendances
- `Cargo.lock`: Géré automatiquement par `cargo`, configuration final des dépendances
- `src`: Dossier pour les sources
  - `main.rs`, Point d'entrée pour un exécutable
  - `lib.rs`, Point d'entrée pour une librairie

STRUCTURE PROJET CARGO 1/2

- `tests` Dossier pour les tests d'intégration
- `examples` Pour les librairies, examples d'utilisation

USAGE DE CARGO 1/2

```
# compile in release mode
cargo build --release
# run the program
cargo run
# check that the package compiles
# but do not produce any binary output
cargo check
# execute unit tests
cargo test
```

USAGE DE CARGO 2/2

```
# generate the documentation
cargo doc
# delete all build artifacts
cargo clean
```

EXERCICE

EXERCICE 3 - COMMANDES CARGO

MANIFESTE CARGO

Exemple de manifeste `Cargo.toml`:

```toml
[package]
# package name
name = "cargo_hello"
# package version
version = "0.1.0"
authors = ["Laurent Wouters <lwouters@cenotelie.fr>"]
# language's edition
edition = "2021"
```

Cargo.toml reference

SPÉCIFICATION DE DÉPENDANCES 1/2

```
1 [dependencies]
2 foo = "1.2.3"
3 # equivalent to:
4 foo = { version = "1.2.3" }
```

- La version de ``foo'' utilisée est au moins 1.2.3.
- Les paquets *doivent* suivre le semantic versioning
- Référence sur la gestion des versions

SPÉCIFICATION DE DÉPENDANCES 2/2

- Cargo est responsable de la construction du graphe de dépendances.
- Il sélectionne les versions les plus hautes disponibles à ce moment là et correspondant aux contraintes données.
- Sauvegardé dans `Cargo.lock`.
- Pour mettre à jour ce graphe:

```
# re-compute the dependency graph
cargo update
```

GESTION DES PAQUETS 1/4

- Par défaut les paquets sont résolus contre crates.io.
- Les paquets publiés sont en fait les sources Rust et les ressources nécessaires à leur compilation.

GESTION DES PAQUETS 2/4

Il est possible de faire référence à un autre projet dans le système de fichier :

```
1 [dependencies]
2 foo = { path = "<path to foo, relative to Cargo.toml>" }
```

... ou alors vers un repository git:

```
1 [dependencies]
2 foo = {
3     git = "https://github.com/bar/foo",
4     branch = "master"
5 }
```

GESTION DES PAQUETS 3/4

Pour remplacer la version résolue (ex. pour tester un changer dans une dépendance):

```
# usual specification
[dependencies]
foo = "1.2.3"

# override
[patch.crates-io]
foo = {
    git = "https://github.com/bar/foo",
    branch = "master"
}
```

GESTION DES PAQUETS 4/4

Fonctionne aussi pour les dépendances transitives:

```
[dependencies]
bar = "4.5.6"

# bar depends on foo, override here
[patch.crates-io]
foo = {
    git = "https://github.com/bar/foo",
    branch = "master"
}
```

ORGANISATION DU CODE

MODULES - INTRO

Les modules (namespaces) peuvent être utilisés pour organiser le code et gérer l'exposition de symboles. Les modules sont définis, soit:

- inline avec `mod {}`,
- ou avec le système de fichier

MODULES - INLINE IN CODE

Exemple de définition de modules dans un fichier:

```
mod a {
    mod b {}
    mod c {}
}
```

La hiérarchie des modules définis est alors:

```
crate
+--> a
     +--> b
     +--> c
```

La racine du paquet est toujours le mot réservé `crate`.

MODULES - SYSTÈME DE FICHIER

Même hiérarchie avec le système de fichier:

```
src/
+--> main.rs
+--> a/
     +--> mod.rs # content for module a
     +--> b.rs
     +--> c.rs
```

MODULES - SYSTÈME DE FICHIER

En plus:

```
// in main.rs:
mod a; // declare module a

// ----------------
// in a/mod.rs
mod b; // declare modules b, and c
mod c;
```

CHEMIN D'ACCÈS - ABSOLUTE / RELATIVE

```
// absolute paths
   crate::x::y::z;
// ^^^^^ always the package (crate)'s root

// relative paths
   x::y;
```

CHEMIN D'ACCÈS - EXEMPLES

```
mod a {
    fn f() {}
    mod b { fn g() {} }
    fn x() {
        b::g();
    }
}
fn main() {
    crate::a::f();
}
```

CHEMIN D'ACCÈS - SUPER

```
mod a {
    fn f() {}
    mod b {
        fn g() {
            super::f();
//          ^^^^^ refers to the *parent* package
            super::super::h();
//          ^^^^^^^^^^^^ we can chain those
        }
    }
}
fn h() { }
```

ACCESSIBILITÉ - MODIFIERS

- Accessibilité relative au module de définition
- Les symboles sont privés par défaut
- `pub` pour exposer publiquement un symbole
- `pub(crate)` pour exposer publiquement dans le paquet courant, mais pas ses dépendants

ACCESSIBILITÉ - EXEMPLES - MODULES

```
mod a {
    pub fn f() {
        b::g(); // ok
    }
    mod b {
        pub fn g() {}
    }
}
fn main() {
    a::f(); // ok
    a::b::g(); // error
}
```

ACCESSIBILITÉ - EXEMPLES - TYPES

```
mod a {
  pub struct X {
    id: i32, // private
    pub name: String // public
  }
  impl X {
    pub fn new() -> X {X { x: 0, name: String::from("x") }}
  }
}
fn main() {
    let x = a::X::new(); // ok, new is public
    println!("{}", &x.name); // ok, name is public
    println!("{}", x.id); // error, id is private
}
```

IMPORT - USE

```rust
mod a {
    pub mod b {
        pub fn f() {}
    }
}
fn x() {
    // import scoped to the function
    use a::b::f; // 👈 import here
    f();
}
fn y() {
    use a::b; // 👈 import here
    b::f();
}
```

IMPORT - IDIOMES

```
// for functions, import the module and use the function
use modulex;
modulex::f();

// for types / constants, directly import
use modulex::StructX;
let x = StructX::new();
```

IMPORT - RENOMMAGE À L'IMPORT

```
use modulex::MyType as MyTypeX;
use moduley::MyType as MyTypeY;
let x = MyTypeX::new();
let y = MyTypeY::new();
```

IMPORT - RÉ-EXPORT DE SYMBOLES

```
mod a {
    pub mod b {
        pub fn f() {}
    }
}
mod c {
    pub use crate::a::b::f as abf;
//  ^^^^^^^ re-export with another name
}
fn main() {
    c::abf();
}
```

IMPORT - RÉ-EXPORT DE SYMBOLES

Le ré-export de symboles est le mécanisme utilisé pour importer automatiquement les symboles d'une dépendance.

Cargo importe par défaut les symboles définis ou ré-exportés par le module `prelude` des dépendances, y compris la librairie standard.

IMPORT - DÉPENDANCES EXTERNES

```
[dependencies]
mylib = "0.1.0"
```

```
use mylib::MyType;
```

IMPORT - FACILITÉS

```
// import multiple symbols:
use mylib::{X, Y};
// nested paths:
use mylib::{x::X, y::Y};
// import the module and some other symbols within it
use mylib::{self, X};
// import all symbols in a module
use mylib::*;
```

TESTS

FONCTION DE TEST

```
#[test]
fn my_test_function() {
    assert!(true);
}
```

Quelques macros utiles:

```
// all asserts raise a panic
// when the hypothesis is not verified
assert!(xxx);
assert_eq!(x, y);
assert_eq!(x, y, "{} is different from {}", x, y);
assert_ne!(x, y);
```

BUILD ET EXÉCUTION DES TESTS

- Les tests ne sont pas compilés lors de la compilation normale du projet avec `cargo build`.
- Les tests sont exécutés par la commande `cargo test`
  - Par défaut, en mode debug.

`Result` POUR LES TESTS

```rust
fn add_2(x: i32) -> i32 { x + 2 }

#[test]
// returning a result will fail the test
fn test_add_2_2_is_4() -> Result<(), String> {
//                          ^^^^^^^^^^^^^^^^^^
    match add_2(2) {
        4 => Ok(()),
        _ => Err(String::from("expected 4"))
    }
}
```

TESTER LA LEVÉE DE `panic!`

```rust
fn will_panic() {
    panic!("oops!");
}

#[test]
#[should_panic] // 👈 test fail if a panic is not raised
fn must_panic() {
    will_panic();
}
```

OÙ ÉCRIRE LES TESTS ? 1/2

Tests unitaires, dans le même module que les fonctions testées

```rust
fn add_2(x: i32) -> i32 { x + 2 }

#[test]
fn test_add_2_2_is_4() { assert_eq!(add_2(2), 4); }

// 👇 test module
#[cfg(test)]
mod my_tests {
    #[test]
    fn test_add_2_2_is_4() { assert_eq!(add_2(2), 4); }
}
```

OÙ ÉCRIRE LES TESTS ? 2/2

Tests d'intégration, dans le répertoire `tests` du projet Cargo.

```
project
+--> Cargo.toml
+--> src/
     +--> lib.rs
+--> tests/           # integration tests
     +--> my_tests.rs
```

EXERCICE

EXERCICE 4 - ECRITURE DE TESTS UNITAIRES

DOCUMENTATION

DOCUMENTATION - PRÉFIXES DE COMMENTAIRES

```rust
//! # Module XXX
//! This is the module-level documentation
//! This module defines the `add_2' function.

/// Adds two to the give number
///
/// The text here is in fact Markdown,
/// we can have
/// # Title
fn add_2(x: i32) -> i32 { x + 2 }
```

DOCUMENTATION - CONVENTIONS

Conventions courantes:

- `Examples`: pour les exemples d'utilisation
- `Panics`: pour les scénarios pour lesquels la fonction peut lever `panic`
- `Errors`: pour décrire les erreurs possibles, dans le cas où le type de retour est `Result`
- `Safety`: pour expliquer l'utilisation de code `unsafe` le cas échéant dans la fonction.

DOCUMENTATION - EXEMPLE DE CODE

```rust
/// Adds two to the give number
///
/// # Examples
/// ```
/// let arg = 3;
/// // note that we use the crate name here
/// let result = my_crate::add_2(arg);
/// assert_eq!(5, result);
/// ```
fn add_2(x: i32) -> i32 { x + 2 }
```

Ce code est compilé et exécuté lors des tests avec `cargo test`!

DOCUMENTATION - GÉNÉRATION

```
# generate the documentation
cargo doc

# the output goes to target/doc
```

EXERCICE

EXERCICE 5 - ECRITURE DE DOCUMENTATION