Référence rapide Rust

Propriété, types, traits, correspondance de motifs essentiels

Bases

Hello World

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

Commandes Cargo

```
cargo new my_project    # create new project
cargo build             # compile (debug)
cargo build --release   # compile (optimized)
cargo run               # build and run
cargo test              # run tests
```

Structure du projet

`Cargo.toml`	Manifeste du projet (dépendances, métadonnées)
`src/main.rs`	Point d'entrée du crate binaire
`src/lib.rs`	Racine du crate bibliothèque
`tests/`	Répertoire des tests d'intégration

Variables et mutabilité

Liaison et mutabilité

```
let x = 5;              // immutable by default
let mut y = 10;         // mutable
y += 1;
const MAX: u32 = 100;   // compile-time constant
```

Masquage (shadowing)

```
let x = 5;
let x = x + 1;          // shadows previous x
let x = "now a string"; // can change type
```

Types scalaires

`i8..i128, isize`	Entiers signés
`u8..u128, usize`	Entiers non signés
`f32, f64`	Virgule flottante (f64 par défaut)
`bool`	**`true`** / **`false`**
`char`	Valeur scalaire Unicode (4 octets)

Types composés

```
let tup: (i32, f64, char) = (42, 6.4, 'z');
let (a, b, c) = tup;        // destructure
let arr: [i32; 3] = [1, 2, 3];
let first = arr[0];
```

Fonctions

Définition

```
fn add(a: i32, b: i32) -> i32 {
    a + b   // no semicolon = return expression
}
```

Fermetures (closures)

```
let double = |x: i32| x * 2;
let sum: i32 = vec![1, 2, 3]
    .iter()
    .map(|x| x * 2)
    .sum();
```

Pointeurs de fonction et traits

`fn(T) -> U`	Type pointeur de fonction
`Fn(T) -> U`	Fermeture qui emprunte
`FnMut(T) -> U`	Fermeture qui emprunte mutuellement
`FnOnce(T) -> U`	Fermeture qui prend la propriété

Flux de contrôle

If / Else

```
let status = if score >= 90 { "A" }
             else if score >= 80 { "B" }
             else { "C" };  // if is an expression
```

Boucles

```
loop { break; }                 // infinite
while condition { }             // while
for item in &vec { }            // iterator
for i in 0..10 { }              // range
for (i, v) in vec.iter().enumerate() { }
```

Étiquettes de boucle

```
'outer: for i in 0..5 {
    for j in 0..5 {
        if i + j > 6 { break 'outer; }
    }
}
```

Propriété et emprunt

Règles de propriété

1. Chaque valeur a exactement un propriétaire.
2. Quand le propriétaire sort de la portée, la valeur est libérée.
3. Les valeurs peuvent être déplacées ou clonées.

Déplacement et clonage

```
let s1 = String::from("hello");
let s2 = s1;          // s1 is moved, no longer valid
let s3 = s2.clone();  // deep copy, both valid
```

Emprunt

```
fn len(s: &String) -> usize { s.len() }  // shared ref
fn push(s: &mut String) { s.push('!'); }  // mutable ref
// Rule: many &T OR one &mut T, never both
```

Durées de vie

```
fn longest<'a>(a: &'a str, b: &'a str) -> &'a str {
    if a.len() > b.len() { a } else { b }
}
```

Structs et enums

Struct

```
struct User {
    name: String,
    age: u32,
    active: bool,
}
let u = User { name: String::from("Alice"), age: 30, active: true };
```

Bloc impl

```
impl User {
    fn new(name: &str, age: u32) -> Self {
        Self { name: name.to_string(), age, active: true }
    }
    fn greeting(&self) -> String {
        format!("Hi, {}", self.name)
    }
}
```

Enums

```
enum Shape {
    Circle(f64),
    Rect { w: f64, h: f64 },
    Point,
}
let s = Shape::Circle(5.0);
```

Correspondance de motifs

Expression match

```
match shape {
    Shape::Circle(r) => std::f64::consts::PI * r * r,
    Shape::Rect { w, h } => w * h,
    Shape::Point => 0.0,
}
```

If Let et While Let

```
if let Some(val) = optional {
    println!("{val}");
}
while let Some(top) = stack.pop() {
    println!("{top}");
}
```

Syntaxe de motifs

`_`	Joker, correspond à tout
`x @ 1..=5`	Lier l'intervalle correspondant à **`x`**
`(a, b, ..)`	Déstructurer le tuple, ignorer le reste
`Some(x) if x > 0`	Garde de correspondance
`Foo { x, .. }`	Struct, ignorer les autres champs

Gestion des erreurs

Result et Option

```
enum Result<T, E> { Ok(T), Err(E) }
enum Option<T>    { Some(T), None }
```

L'opérateur ?

```
fn read_file(path: &str) -> Result<String, io::Error> {
    let mut s = String::new();
    File::open(path)?.read_to_string(&mut s)?;
    Ok(s)
}
```

Gérer les erreurs

```
match result {
    Ok(val) => println!("{val}"),
    Err(e) => eprintln!("Error: {e}"),
}
let val = result.unwrap_or(0);
let val = result.unwrap_or_else(|_| default());
```

Méthodes courantes

`.unwrap()`	Obtenir la valeur ou paniquer
`.expect(msg)`	Obtenir la valeur ou paniquer avec un message
`.unwrap_or(default)`	Obtenir la valeur ou utiliser la valeur par défaut
`.map(f)`	Transformer la valeur Ok/Some
`.and_then(f)`	Chaîner les opérations (flatmap)
`.is_ok() / .is_some()`	Vérification booléenne

Référence rapide Rust

Traits

Définir et implémenter

```
trait Summary {
    fn summarize(&self) -> String;
    fn preview(&self) -> String {   // default impl
        format!("{}...", &self.summarize()[..20])
    }
}
impl Summary for User {
    fn summarize(&self) -> String { self.name.clone() }
}
```

Limites de trait

```
fn notify(item: &impl Summary) { }
fn notify<T: Summary + Display>(item: &T) { }
fn notify(item: &(impl Summary + Display)) { }
```

Traits courants

`Display`	Formatage de chaîne orienté utilisateur
`Debug`	Formatage de débogage (**`{:?}`**)
`Clone, Copy`	Duplication (profonde / par bits)
`PartialEq, Eq`	Comparaison d'égalité
`PartialOrd, Ord`	Comparaison d'ordre
`Iterator`	**`next()`** pour l'itération
`From, Into`	Conversions de type
`Default`	Constructeur de valeur par défaut

Collections

Vec

```
let mut v: Vec<i32> = vec![1, 2, 3];
v.push(4);
v.pop();              // returns Option<i32>
let first = &v[0];   // panics if empty
let first = v.get(0); // returns Option<&i32>
```

HashMap

```
use std::collections::HashMap;
let mut m = HashMap::new();
m.insert("key", 42);
m.entry("key").or_insert(0);
if let Some(val) = m.get("key") { }
```

String

```
let s = String::from("hello");
let s = "hello".to_string();
let combined = format!("{} {}", s, "world");
for c in s.chars() { }   // iterate characters
```

Itérateurs

```
let sum: i32 = vec![1, 2, 3].iter().sum();
let doubled: Vec<_> = v.iter().map(|x| x * 2).collect();
let evens: Vec<_> = v.iter().filter(|x| *x % 2 == 0).collect();
```

Concurrence

Fils d'exécution

```
use std::thread;
let handle = thread::spawn(|| {
    println!("from spawned thread");
});
handle.join().unwrap();
```

Canaux

```
use std::sync::mpsc;
let (tx, rx) = mpsc::channel();
tx.send(42).unwrap();
let val = rx.recv().unwrap();
```

État partagé

`Arc<T>`	Comptage de références atomique (Rc thread-safe)
`Mutex<T>`	Exclusion mutuelle, verrou pour accéder à la valeur interne
`RwLock<T>`	Plusieurs lecteurs ou un seul écrivain
`Send`	Trait : sûr à transférer entre les fils
`Sync`	Trait : sûr à partager des références entre les fils

Macros et attributs

Macros courantes

`println!()`	Afficher avec saut de ligne
`format!()`	Retourner une String formatée
`vec![]`	Créer un Vec depuis des littéraux
`todo!()`	Espace réservé, panique à l'exécution
`assert!(expr)`	Paniquer si expr est faux
`assert_eq!(a, b)`	Paniquer si a != b

Attributs derive

```
#[derive(Debug, Clone, PartialEq)]
struct Point { x: f64, y: f64 }
// Auto-implements Debug, Clone, PartialEq
```

Attributs de test

```
#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn it_works() { assert_eq!(add(2, 2), 4); }
    #[test]
    #[should_panic]
    fn it_panics() { panic!("boom"); }
}
```