Cette étude aborde les expressions **Rust**, qui constituent les blocs de construction du corps des fonctions **Rust**, avec la mise en œuvre des alternatives, des itératives, des sélections multiples, etc. par un ensemble d'instructions ou de blocs d'instructions qui compose une application classique.

Alternatives

Dans la plupart des langages, les instructions **if** et **else** permettent d'implémenter des alternatives. À la suite de **if** nous devons avoir une condition d'évaluation qui doit toujours être une expression de type **bool**. Rappelons que **Rust** ne convertit jamais d'office les valeurs numériques ni les pointeurs vers des valeurs logiques booléennes.

*À la différence du **C++**, vous n'êtes pas forcé d'ajouter des parenthèses autour des conditions. D'ailleurs, le compilateur **Rust** affiche un avertissement s'il y a des parenthèses inutiles. En revanche, les accolades sont obligatoires.*

*Vous pouvez imbriquer plusieurs alternatives. En revanche, les alternatives **else if** et le **else** final sont facultatives. Lorsqu'une expression **if** ne possède pas de branche **else**, elle se comporte comme si elle avait un bloc **else** vide.*

Alternatives

```
use std::io::stdin;

fn main() {
    println!("Quel est votre note ?");
    let mut saisie = String::new();
    stdin().read_line(&mut saisie).expect("saisie incorrecte");
    let note : f32 = saisie.trim().parse().expect("Il faut une valeur numérique");
    if note > 15. {
        println!("C'est très bien !");
    }
   else if note > 10.0 {
       println!("C'est bien !");
   }
    else {
        println!("Attention ! vous n'avez pas la moyenne.");
    }
}
```

Résultat

```
Quel est votre note ?
15.5
C'est très bien !
```

Langage d'expressions, déclarations

Une expression est évaluée pour produire une valeur alors qu'une instruction classique ne le fait pas spécialement. **Rust** est un langage d'expressions. En langage **C++** par exemple, **if** et **switch** sont des instructions qui ne produisent pas de valeur et ne peuvent donc pas être insérées au milieu d'une expression. Dans **Rust**, **if** et **match** peuvent produire une valeur.

*Dans le même registre un bloc **Rust**, entouré d'accolades, est également considéré comme une expression, sachant que c'est la dernière instruction du bloc qui peut retourner une valeur. Dans ce cas là, la dernière instruction ne possède pas de ponctuation point-virgule.*

*Un bloc contient donc des expressions, des signes de point virgules, mais également des déclarations, localisées au bloc, les plus fréquentes étant des variables locales avec l'opérateur **let**. La possibilité pour un bloc de pouvoir contenir des déclarations tout en produisant une valeur à la fin est très pratique, ce qui nous permet d'avoir un code plus lisible, plus logique et souvent plus concis, qui donne aussi la possibilité d'avoir des variables locales avec une durée de vie très courte.*

*La déclaration **let** permet éventuellement de déclarer le nom d'une variable sans lui donner de valeur initiale, celle-ci pouvant être attribuée plus tard. Ce report est parfois utile, lorsque la variable doit être initialisée au milieu d'un bloc de contrôle.*

Expressions sous forme de blocs et d'alternatives

```
use std::io::stdin;

fn main() {
    let note : f32 = {
        println!("Quel est votre note ?");
        let mut saisie = String::new();
        stdin().read_line(&mut saisie).expect("saisie incorrecte");
        saisie.trim().parse().expect("Il faut une valeur numérique")
    };
    println!("{}",
        if note > 15.       { "C'est très bien !" }
        else if note > 10.0 { "C'est bien !" }
        else { "Attention ! vous n'avez pas la moyenne" }
    );
}
```

Résultat

```
Quel est votre note ?
```

15.5
C'est très bien !

En reprenant le projet précédent, nous retrouvons exactement le même résultat, avec un code structuré différemment. Nous montrons ici que le programme comporte deux parties principales, la saisie de la note et ensuite l'affichage circonstancié.

*Le fait qu'un bloc soit une expression en **Rust** et beaucoup plus riche que le langage **C++** qui considère qu'un bloc est juste un ensemble d'instructions. Ici, par exemple, nous montrons que la variable **saisie** n'a aucune importance en tant que tel, elle est seulement utile pour générer la variable **note**.*

*Les instructions **if** et **else** sont aussi des expressions. Du coup, nous pouvons les intégrer directement au sein d'une instruction comme par exemple à l'intérieur de la macro **println!()**.*

Séquences multiples - match

Rust possède un opérateur de contrôle très puissant appelé **match** qui vous permet de comparer une valeur avec une série de motifs et d'exécuter du code en fonction du motif qui correspond. Les motifs peuvent être constitués de valeurs littérales, de noms de variables, de jokers, parmi tant d'autres. Ce qui fait la puissance de **match** est l'expressivité des motifs et le fait que le compilateur vérifie que tous les cas possibles sont bien gérés.

*L'expression **match** ressemble à l'instruction **switch** du langage **C++**, mais elle est beaucoup plus polyvalente. L'idée est la même, c'est de choisir une expression parmi un ensemble de propositions. Le caractère de soulignement « _ »correspond à l'opérateur **default** du langage **C++**.*

Sélection multiple avec l'expression match

```
use std::io::stdin;

fn main() {
    let moment = {
        println!("Quel moment de la journée préférez-vous ? (matin, après-midi, soir, nuit) :");
        let mut saisie = String::new();
        stdin().read_line(&mut saisie).expect("Saisie incorrecte");
        &saisie.trim().to_lowercase()[..]
    };
    println!("{}",
        match moment {
            "matin"       => "Bonne matinée !",
            "après-midi" => "Bonne après-midi !",
            "soir"        => "Bonne soirée !",
            "nuit"        => "Bonne nuit !",
            _             => "Bonne journée !"
        }
    );
}
```

*La grande souplesse de **match** découle du vaste nombre de motifs supportés dans la partie gauche de l'opérateur **=>** de chaque branche. Dans cet exemple, chaque motif se résume à une constante littérale chaîne (ce qui est totalement impossible en **C++**).*

Ce n'est qu'un avant goût de ce qu'il est possible de faire avec des motifs. Un motif peut englober toute une plage de valeurs, déballer le contenu de tuples ou encore comparer par rapport à un champ de structure. Il permet de trouver les cibles des références, emprunter une partie de valeurs, etc.

*Les motifs **Rust** constituent une sorte de mini-langage que nous exploiterons pleinement lorsque nous aurons étudié les énumérations et les structures. Voici le format générique d'une expression **match** :*

Format d'une expression match

```
    match moment {
        motif    => expression,
        motif    => expression,
        ...
        _        => expression

    }
```

*Notez que la virgule à la fin d'une branche peut être omise si l'expression correspond à un bloc délimité. **Rust** compare la valeur de chaque motif, en commençant par le premier. Dès qu'un motif correspond, son expression est évaluée, ce qui termine l'expression **match** ; les autres motifs ne sont pas testés.*
*Autrement dit, au moins un motif doit correspondre et **Rust** interdit les expressions **match** qui ne gèrent pas toutes les valeurs possibles. Le choix par défaut « _ » devient vite indispensable :*

Expressions sous forme de blocs et d'alternatives

```
use std::io::stdin;

fn main() {
    let moment = {
        println!("Quel moment de la journée préférez-vous ? (matin, après-midi, soir, nuit) :");
```

```
    let mut saisie = String::new();
    stdin().read_line(&mut saisie).expect("Saisie incorrecte");
    &saisie.trim().to_lowercase()[..]
  };
  println!("{}",
      match moment {
        "matin"      => "Bonne matinée !",
        "après-midi" => "Bonne après-midi !",
        "soir"       => "Bonne soirée !",
        "nuit"       => "Bonne nuit !",
//          _          => "Bonne journée !"
      }
  );
}
```

Résultat

```
11 |         match moment {
   |               ^^^^^^^ pattern `&_` not covered
```

Utiliser if dans une instruction let

Comme **if** est une expression, nous pouvons l'utiliser à la suite d'une instruction **let**, comme cela vous est montré dans le programme ci-dessous :

let if

```
fn main() {
  let nombre = -5.3;
  let positif = if nombre>=0. { nombre } else { -nombre };

  println!("Nombre={}, Positif={}", nombre, positif);
}
```

Résultat

```
Nombre=-5.3, Positif=5.3
```

*Souvenez-vous que les blocs de code s'exécutent jusqu'à la dernière expression qu'ils contiennent, et que les nombres tout seuls sont aussi des expressions. Dans notre cas, la valeur de toute l'expression **if** dépend de quel bloc de code elle va exécuter. Cela veut dire que chaque valeur qui peut être le résultat de chaque branche du **if** doivent impérativement être du même type. Si les types ne sont pas identiques nous obtenons systématiquement une erreur.*

Les répétitions avec les boucles

Nous avons souvent besoin d'exécuter un bloc de code plusieurs fois. Dans ce but, **Rust** propose plusieurs types de boucles. Une boucle parcourt le code à l'intérieur du corps de la boucle jusqu'à la fin et recommence immédiatement du début. **Rust** propose trois expressions pour les boucles de répétition. Ces trois types de boucle sont associés au mots clés suivants dont l'ossature vous est montrée ci-dessous :

let if

```
while condition {
  bloc
}
loop {
  bloc
}
for motif in collection {
  bloc
}
```

*Les boucles sont bien des expressions mais elles ne produisent pas de valeurs utiles en **Rust**. Leur valeur se résume à un tuple vide : **()**. Avec la boucle **loop**, il est toutefois possible de retourner une valeur au moyen de l'expression **break valeur_de_retour ;***
*La boucle **while** fonctionne exactement comme le C++, à la seule exception que la condition est systématiquement de type **bool**.*
*La boucle **loop** est particulière. Elle permet d'écrire des boucles infinies puisqu'elle exécute le bloc de façon répétée sans s'arrêter. Heureusement, **Rust** fournit un moyen de sortir malgré tout de cette boucle, avec **break**, **continue** ou **return**, comme en **C++**, ou lorsque nous lançons une exception (panique de processus).*
*Enfin, la boucle **for** ressemble à la syntaxe du langage **Python**. Cette boucle commence par évaluer l'expression collection puis évalue le bloc une fois pour chaque élément de la collection. La boucle **for** est très bien adaptée pour parcourir une collection en récupérant chaque valeur trouvée. De nombreux types de collections sont reconnus, tableaux, vecteurs, chaînes de caractères, **map**, etc.*

__Rust__ propose une syntaxe très intuitives (contrairement à __Python__) avec la boucle __for__ pour parcourir une plage de valeurs grâce à l'opérateur « __..__ ». Cette plage est une structure constituée de deux champs nommés __start__ et __end__. Ainsi, __0..20__ est équivalent à __Range__ {__start__:0, __end__:20}. Vous pouvez utiliser une plage dans une boucle __for__ parce que le type __Range__ est itérable. Toutes les collections standards sont itérables, ainsi que les tableaux et les tranches.

Des boucles partout

```
fn main() {
    let mut nombre=1;

    print!("While : ");
    while nombre<=10 {
        print!("{} ", nombre*nombre);
        nombre+=1;
    }

    print!("\nFor 1..11 : ");
    for nombre in 1..11 {
        print!("{} ", nombre*nombre);
    }

    print!("\nFor 1..=10 : ");
    for nombre in 1..=10 {
        print!("{} ", nombre*nombre);
    }

    print!("\nWhile : ");
    let mut vecteur = vec![];
    nombre=0;
    while nombre<50 {
        nombre+=1;
        if nombre%3==0 { continue; }
        if nombre*nombre > 300 { break; }
        vecteur.push(nombre*nombre);
        print!("{} ", nombre*nombre);
    }

    print!("\nVecteur {:?} : ", vecteur);
    for nombre in &vecteur {
        print!("{} ", nombre);
    }

    print!("\nVecteur {:?} : ", &vecteur[..5]);
    for nombre in &vecteur[..5] {
        print!("{} ", nombre);
    }

    print!("\nLoop : ");
    nombre = 0;
    let suite = loop {
        nombre+=1;
        print!("{} ", nombre*nombre);
        if nombre==10 { break (nombre+1)*(nombre+1); }
    };

    print!("{}", suite);
    let message = "Bonjour";
    print!("\nMessage => {} : ", &message);
    for caractere in message.chars() {
        print!("{} ", caractere);
    }

    print!("\nMessage => {} : ", &message[3..]);
    for caractere in message[3..].chars() {
        print!("{} ", caractere);
    }
}
```

Résultat

```
While : 1 4 9 16 25 36 49 64 81 100
For 1..11 : 1 4 9 16 25 36 49 64 81 100
For 1..=10 : 1 4 9 16 25 36 49 64 81 100
While : 1 4 16 25 49 64 100 121 169 196 256 289
Vecteur [1, 4, 16, 25, 49, 64, 100, 121, 169, 196, 256, 289] : 1 4 16 25 49 64 100 121 169 196 256 289
Vecteur [1, 4, 16, 25, 49] : 1 4 16 25 49
Loop : 1 4 9 16 25 36 49 64 81 100 121
```

```
Message => Bonjour : B o n j o u r
Message => jour : j o u r
```

Calcul du factoriel d'un nombre

```
use std::io::stdin;
fn main() {
    let nombre : u128 = {
        println!("Factoriel : n! n ?");
        let mut saisie = String::new();
        stdin().read_line(&mut saisie).expect("saisie incorrecte");
        saisie.trim().parse().expect("Il faut une valeur numérique")
    };

    let mut resultat:u128 = 1;
    print!("{}! = ", nombre);
    for n in 1..nombre {
        resultat*=n;
        print!("{}x", n);
    }
    println!("{} = {}", nombre, resultat*nombre);
}
```

Résultat

```
Factoriel : n! n ?
5
5! = 1x2x3x4x5 = 120
```

*Je vous propose une deuxième solution non optimisée qui permet de combiner la boucle **for** avec une sélection multiple en utilisant l'opérateur **match**. Remarquez au passage l'utilisation de l'opérateur « | » dans mon motif :*

Calcul du factoriel d'un nombre

```
use std::io::stdin;
fn main() {
    let nombre : u128 = {
        println!("Factoriel : n! n ?");
        let mut saisie = String::new();
        stdin().read_line(&mut saisie).expect("saisie incorrecte");
        saisie.trim().parse().expect("Il faut une valeur numérique")
    };

    let resultat: u128 = match nombre {
        0 | 1 => 1,
        _ => {
            let mut result :u128 = 1;
            for n in 2..nombre+1 { result*=n; }
            result
        }
    };
    println!("{}! = {}", nombre, resultat);
}
```

Résultat

```
Factoriel : n! n ?
5
5! = 120
```

Autre alternative pour le traitement du factoriel

```
let mut resultat: u128=1;
match nombre {
    0 | 1 => resultat=1,
    _ => for n in 2..nombre+1 { resultat*=n; }
};
println!("{}! = {}", nombre, resultat);
```

ou tout simplement

```
let mut resultat: u128=1;
for n in 2..nombre+1 { resultat*=n; }
println!("{}! = {}", nombre, resultat);
```

Résultat

```
Factoriel : n! n ?
0
0! = 1
```

```
Factoriel : n! n ?
1
1! = 1

Factoriel : n! n ?
5
5! = 120
```

Calcul du factoriel en tenant compte des limites du calcul

```
use std::io::stdin;
fn main() {
    let nombre : u128 = {
        println!("Factoriel : n! n ?");
        let mut saisie = String::new();
        stdin().read_line(&mut saisie).expect("saisie incorrecte");
        saisie.trim().parse().expect("Il faut une valeur numérique")
    };
    match nombre {
        0...34 => {
            let mut resultat: u128 = 1;
            for n in 2..nombre + 1 { resultat *= n; }
            println!("{}! = {}", nombre, resultat);
        }
        _ => println!("Votre nombre est trop grand")
    }
}
```

Résultat

```
Factoriel : n! n ?
15
15! = 1307674368000

Factoriel : n! n ?
36
Votre nombre est trop grand
```

*Avec une expression **match**, nous pouvons proposer un motif de sélection sur une plage de valeurs, ce qui est extrêmement séduisant et confortable à utiliser. Remarquez au passage que cette plage de valeurs est exprimée par « **...** » qui est équivalent à « **..=** ». Attention, la plage de valeur avec l'expression « **..** » n'est pas tolérée puisque nous avons une exclusion sur le dernier élément (c'est un non sens).*