Comme dans beaucoup d'autres langages, les fonctions sont omniprésentes dans le code **Rust**. Nous avons largement utilisé l'une des fonctions les plus importantes du langage : la fonction ***main()***, qui est le point d'entrée de beaucoup de programmes. Nous avons aussi remarqué le mot-clé **fn**, qui nous permet de définir une nouvelle fonction.

*Il est à noter qu'à la différence du **C++**, nous n'avons pas deux phases possibles, la déclaration et le définition d'une fonction. **Rust** propose qu'une seule écriture où la déclaration de la fonction est aussi sa définition. Par ailleurs la notion de fonction « inline » n'existe pas, ni les paramètres par défaut et ni le nombre de paramètres variables. La structure d'une fonction en Rust est beaucoup plus simple et ne s'encombre pas d'artifices qui sont finalement assez peu utilisés.*

Définition et utilisation d'une fonction

La définition d'une fonction avec **Rust** commence toujours par le mot-clé **fn** qui sert à ouvrir la déclaration de cette fonction. Ce mot-clé est suivi du nom de la fonction et possède systématiquement d'une paire de parenthèses (ce qui permet de reconnaître une fonction) après le nom de cette fonction. Les accolades indiquent au compilateur où le corps de la fonction commence et où il se termine.

*Une fois que la fonction a été définie, nous pouvons l'appeler à tout moment en utilisant son nom, suivi de la paire de parenthèses. Dans l'exemple qui suit, comme **bonjour()** est définie dans le programme, elle peut être appelée à l'intérieur de la fonction **main()**. Remarquez que nous avons défini **bonjour()** après la fonction **main()** dans le code source ; nous aurions tout aussi bien pu la définir avant. **Rust** ne se soucie pas de l'endroit où vous définissez vos fonctions, du moment qu'elles sont bien définies quelque part.*

Définition et utilisation de la fonction bonjour

```
fn main() {
    println!("Bienvenue!");
    bonjour();
}

fn bonjour() {
    println!("Bonjour!");
}
```

Résultat

```
Bienvenue!
Bonjour!
```

Les paramètres d'une fonction

Les fonctions peuvent bien entendu être définies avec des paramètres (d'où l'intérêt des parenthèses), qui sont des variables spéciales qui font partie de la signature de la fonction. Au moment de l'appel de la fonction, vous devez alors lui fournir des valeurs concrètes (littéraux ou variables). Ces valeurs concrètes sont appelées les arguments.

*Dans la signature de la fonction, vous devez déclarer le type de chaque paramètre. C'est un choix délibéré de conception de **Rust** : exiger l'annotation de type dans la définition d'une fonction fait en sorte que le compilateur sache se qu'il doit faire. Si vous avez plusieurs paramètres, vous utilisez l'opérateur virgule « , » comme séparateur.*

Fonction factorielle

```
fn main() {
    let x=5;
    factoriel(x);
}

fn factoriel(n:u128) {
    match n {
        0...34 => {
            let mut resultat = 1;
            for x in 2..n+1 { resultat*=x; }
            println!("{}!={}", n, resultat);
        }
        _ => println!("Dépassement de capacité!")
    }
}
```

Résultat

```
5!=120
```

*Dans notre programme principal, nous faisons appel à la fonction **factoriel()** qui prend **x** en argument. La valeur de l'argument **x** est copiée dans le paramètre **n** de la fonction. Ce paramètre **n** sert ensuite au traitement de la fonction. Une fois que la fonction est terminée, cette variable **n** est automatiquement détruite au même titre que la variable **resultat** . Ce sont des **variables locales** à la fonction non visibles de l'extérieur (créations et suppressions dans la pile).*

Corps de fonction avec des instructions et des expressions

Les corps de fonctions sont constitués d'une série d'instructions qui se termine éventuellement par une expression. Jusqu'à présent, nous avons vu des fonctions sans expression à la fin, mais il est possible d'en rajouter.

*Dans l'étude précédente, nous avons vu que **Rust** est un langage basé sur des expressions, il est important de faire la distinction. D'autres langages ne font pas de telles distinctions, donc revoyons ce que sont les instructions et les expressions et comment leurs différences influent sur le corps des fonctions.*

Nous avons déjà utilisé des instructions et des expressions. Les instructions effectuent des actions et ne retournent aucune valeur. Les expressions sont évaluées pour retourner une valeur comme résultat.

*Créer une variable en lui assignant une valeur avec le mot-clé **let** constitue une instruction. La définition d'une fonction est aussi une instruction ; l'intégralité de l'exemple précédent est une instruction à elle toute seule.*
*Les expressions sont évaluées et seront ce que vous écrirez le plus en **Rust** (hormis les instructions bien sûr). Prenez une simple opération mathématique, comme **5 + 6**, qui est une expression qui s'évalue à la valeur **11**. L'appel de fonction est aussi une expression. L'appel de macro est une expression. Le bloc que nous utilisons pour créer une nouvelle portée, {}, est une expression, par exemple :*

Instruction et expression

```
fn main() {
    let x = 5;
    let y = {
        let x = 3;
        x-2
    };
    println!("x={}, y={}", x, y);
}
```

Résultat

```
x=5, y=1
```

*Nous observons un bloc qui, dans notre cas, s'évalue à **1**. Cette valeur est assignée à **y** dans le cadre de l'instruction **let**. Remarquez la ligne **x-2** qui ne se termine pas par un point-virgule, ce qui est différent de la plupart des lignes que vous avez vues jusque là.*
Les expressions n'ont pas de point-virgule de fin de ligne. Si vous ajoutez un point-virgule à la fin de l'expression, vous la transformez en instruction, qui ne retourne donc pas de valeur.

Les fonctions qui retournent des valeurs

Les fonctions peuvent retourner des valeurs au code qui les appelle. Nous ne nommons pas les valeurs de retour, mais nous devons déclarer leur type après une flèche « **->** ». En **Rust**, la valeur de retour de la fonction est la même que la valeur de l'expression finale dans le corps de la fonction.

*Vous pouvez sortir prématurément d'une fonction en utilisant le mot-clé **return** et en précisant la valeur de retour, mais la plupart des fonctions vont retourner implicitement la dernière expression.*

La fonction factorielle

```
fn main() {
    let x=5;
    println!("{}!={}", x, factoriel(x));
}

fn factoriel(n:u128) -> u128 {
    let mut resultat = 1;
    for x in 2..n+1 { resultat*=x; }
    resultat
}
```

Résultat

```
5!=120
```

*La valeur dans la variable **resultat** est la valeur de retour de la fonction, ce qui explique le type de retour de **u128**. Regardons cela plus en détail. Le paramètre **n** est de type **u128**, **resultat** par l'intermédiaire de **x** est également considéré comme un type **u128**. Enfin, l'argument **x**, dans la fonction principale, est lui-même considéré comme un **u128**, puisqu'il est rattaché au paramètre **n**. **Rust** influence ses types par induction.*

Attention, si nous ajoutons un point-virgule à la fin de la dernière ligne de la fonction évoquant le retour du résultat, ce qui était une expression devient une instruction, nous obtenons alors une erreur.

La fonction factorielle

```
fn main() {
    let x=5;
    println!("{}!={}", x, factoriel(x));
}

fn factoriel(n:u128) -> u128 {
    let mut resultat= 1;
    for x in 2..n+1 { resultat*=x; }
```

```
    resultat;
}
```

Résultat

```
6 | fn factoriel(n:u128) -> u128 {
  |    ---------                ^^^^ expected `u128`, found `()`
  |    |
  |    implicitly returns `()` as its body has no tail or `return` expression
...
9 |    resultat;
  |             - help: consider removing this semicolon
```

*La définition de la fonction **factoriel()** dit qu'elle va retourner un u128, mais les instructions ne retournent pas de valeur, ceci est donc représenté par **()**, un tuple vide. Par conséquent, rien n'est retourné, ce qui contredit la définition de la fonction et provoque une erreur. **Rust** affiche un message qui peut aider à corriger ce problème : il suggère d'enlever le point-virgule, ce qui va résoudre notre problème.*

*Une fonction qui ne retourne pas de valeur peut proposer la flèche suivie du tuple vide **→()**, ce qui est équivalent à ne rien mettre. Revoyons notre toute première fonction avec cette nouvelle écriture :*

Définition et utilisation de la fonction bonjour

```
fn main() {
    println!("Bienvenue!");
    bonjour();
}

fn bonjour() -> () {
    println!("Bonjour!");
}
```

Résultat

```
Bienvenue!
Bonjour!
```

La récursivité

Comme beaucoup d'autres langages, nous pouvons définir des méthodes avec une écriture récursive, c'est-à-dire une fonction qui s'appelle elle-même. Attention, dans ce genre d'écriture, vous devez impérativement avoir un critère d'arrêt, sinon, les appels récursifs s'effectuent indéfiniment.

*Grâce à la notion **d'expression** plus l'utilisation de motifs particuliers associés à l'expression **match**, au fait également qu'une fonction **Rust** est très habile avec les **expressions**, nous obtenons une écriture vraiment très séduisante et intuitive :*

La fonction factorielle en écriture récursive

```
fn main() {
    let x=5;
    println!("{}!={}", x, factoriel(x));
}

fn factoriel(n:u128) -> u128 {
    match n {
        0 | 1 => 1,
        _     => n*factoriel(n-1)
    }
}
```

Résultat

```
5!=120
```

Retourner plusieurs valeurs

Grâce à l'utilisation de **tuples**, nous pouvons retourner plusieurs valeurs pour une même fonction. Dans la syntaxe de retour lors de la définition, vous devez alors préciser les différents types de retour avec une expression de type **tuple**.

*Grâce à la notion **d'expression** plus l'utilisation de motifs particuliers associés à l'expression **match**, au fait également qu'une fonction **Rust** est très habile avec les **expressions**, nous obtenons une écriture vraiment très séduisante et intuitive :*

La fonction factorielle en écriture récursive

```
fn main() {
    let x=5.8;
    let (entiere, decimale) = reel(x);
    println!("{} = {} + {:.1}", x, entiere, decimale);
}
```

```
fn reel(nombre:f64) -> (i64, f64)
{
   let entier = nombre as i64;
   (entier, nombre-entier as f64)
}
```

Résultat

```
5.8 = 5 + 0.8
```

Passage par valeurs ou par références

Jusqu'à présent, nous avons utilisé des variables de type primitif. Que se passe-t-il lorsque nous utilisons des variables de type composé, comme les tableaux, les vecteurs, les chaînes de caractères, etc.

*Le point important est de bien maîtriser la notion de **possession** et **d'emprunt** qui vont influencer largement le choix du type de paramètres. Dans le cas des types primitifs, l'argument est copié dans le paramètre. Pour les variables qui sont déclarées dans le tas, soit il s'agit d'un changement de **possession**, soit il s'agit d'un **emprunt**.*

Gestion de notes

```
fn main() {
   let notes = vec![12.5, 8.5, 15.5, 17.5];
   println!("Notes : {:?}", &notes);
   println!("Moyenne = {}", moyenne(&notes));
   println!("Note maxi = {}", maxi(&notes));
   println!("Note mini = {}", mini(&notes));
   println!("[5.0, 10., 15.0], moyenne = {}", moyenne(&[5., 10., 15.].to_vec()));
   println!("{:?} => {:?}", &notes, gestion_notes(&notes));
}

fn moyenne(notes: &Vec<f64>) -> f64 {
   match notes.len() {
      0 => 0.,
      _ => {
         let mut somme= notes[0];
         for note in &notes[1..] { somme += *note; }
         somme/notes.len() as f64
      }
   }
}

fn maxi(notes: &Vec<f64>) -> f64 {
   match notes.len() {
      0 => 0.,
      _ => {
         let mut maxi= notes[0];
         for note in &notes[1..] {
            if *note>maxi { maxi = *note; }
         }
         maxi
      }
   }
}

fn mini(notes: &Vec<f64>) -> f64 {
   match notes.len() {
      0 => 0.,
      _ => {
         let mut mini= notes[0];
         for note in &notes[1..] {
            if *note<mini { mini = *note; }
         }
         mini
      }
   }
}

fn gestion_notes(notes: &Vec<f64>) -> (f64, f64, f64) {
   match notes.len() {
      0 => (0., 0., 0.),
      _ => {
         let mut somme= notes[0];
         let mut maxi= notes[0];
         let mut mini= notes[0];
         for note in &notes[1..] {
            somme += *note;
            if *note>maxi { maxi = *note; }
```

```
            if *note<mini { mini = *note; }
        }
        (somme/notes.len() as f64, maxi, mini)
    }
  }
}
```

Résultat

```
Notes : [12.5, 8.5, 15.5, 17.5]
Moyenne = 13.5
Note maxi = 17.5
Note mini = 8.5
[5.0, 10., 15.0], moyenne = 10
[12.5, 8.5, 15.5, 17.5] => (13.5, 17.5, 8.5)
```

*Nous pouvons aussi utiliser l'instruction **return** plutôt que de prendre une expression **match**. Voici par exemple la modification que nous pouvons apporter sur la dernière fonction :*

Gestion de notes avec return

```
fn gestion_notes(notes: &Vec<f64>) -> (f64, f64, f64) {
    if notes.len()==0 { return (0., 0., 0.) }
    let mut somme= notes[0];
    let mut maxi= notes[0];
    let mut mini= notes[0];
    for note in &notes[1..] {
        somme += *note;
        if *note>maxi { maxi = *note; }
        if *note<mini { mini = *note; }
    }
    (somme/notes.len() as f64, maxi, mini)
}
```

Cette dernière fonction nous permet de faire une analyse d'une chaîne de caractères passé en argument, en discriminant le nombre de lettres, le nombre de mots, le nombre de voyelles, de consonnes et de caractères de ponctuation :

Nombre de lettres et de mots

```
fn main() {
   let message = "Bonjour à tout le monde!";
   let (nombre, mots, voyelles, consonnes, ponctuations) = phrase(&message);
   println!("'{}' comporte {} lettres, {} mots, {} voyelles, {} consonnes et {} ponctuations",
         message, nombre, mots, voyelles, consonnes, ponctuations);
}

fn phrase(texte:&str) -> (usize, usize, usize, usize, usize) {
   let nombre= texte.chars().count();
   let mut ponctuations = 0;
   let mut mots = 0;
   let mut voyelles = 0;
   for caractere in texte.chars() {
      if caractere == ' ' { mots += 1; }
      if "aeiouyAEIOUYàâéèêëüûïî".contains(caractere) { voyelles += 1; }
      if caractere.is_ascii_punctuation() { ponctuations +=1; }
   }
   let consonnes = nombre-mots-voyelles-ponctuations;
   (nombre, mots+1, voyelles, consonnes, ponctuations)
}
```

Résultat

```
'Bonjour à tout le monde!' comporte 24 lettres, 5 mots, 9 voyelles, 10 consonnes et 1 ponctuations
```