

# Variables et types primitifs (et cast)

En rust les variables sont définie avec `let` et sont immuable par défaut

```
// Cette variable est immuable
let x = 42;

// Cette variable est muable (peut être modifiée)
let mut y = 42;

// On peut préciser un type avec :
// Préciser les différents types n'est pas obligatoire, je le fais ici uniquement pour
l'explication
// Rust a la capacité dans beaucoup de cas de deviner de quel type la variable doit être
let z: i32 = 42;

// Une variable qui commence par _ est un message au compilateur qu'il ne doit pas se soucier
qu'elle n'est jamais utilisée
let _a = 42;

// On peut bien évidemment faire des opération mathématiques dans nos variables
let b = x + y;
```

Maintenant intéressons nous un peu aux types de base. Pour ce qui est des nombres :

Longueur	Signé	Non signé	Flottant (à virgule)
8-bit	<code>i8</code>	<code>u8</code>	
16-bit	<code>i16</code>	<code>u16</code>	
32-bit	<code>i32</code>	<code>u32</code>	<code>f32</code>
64-bit	<code>i64</code>	<code>u64</code>	<code>f64</code>
128-bit	<code>i128</code>	<code>u128</code>	
arch	<code>isize</code>	<code>usize</code>	

Pour ce qui est des chaînes de caractères, le type primitif qui correspond est le `str` (a ne pas confondre avec `String` qui n'est pas un type primitif):

```
// str est toujours sous la forme &str car c'est une référence
let x: &str = "hello ";
let y: &str = "world";

// &str est statique, on ne peut donc pas concaténer x et y
let z: &str = x + y;
```

Ensuite on peut créer un tuple

```
// mut pour pouvoir le modifier après
let mut tuple: (i32, &str, f32) = (42, "answer to life", 42.0);

// On peut ensuite faire référence aux éléments du tuple avec .0 .1 .2 etc
println!("{}", tuple.1, tuple.0);

// On peut bien évidemment modifier notre tuple aussi
tuple.0 = 27;
println!("{}", tuple.1, tuple.0);
```

Maintenant on va voir un autre type de groupement d'élément : l'array. A ne pas confondre avec le [Vecteur](#) qui n'est pas un type primitif.

```
// mut pour pouvoir le modifier après
// Les arrays ont eu aussi une taille fixe, mais contrairement aux tuples, ils ne peuvent
contenir qu'un seul type
// Ici la taille est directement indiquée dans le type par le "; 3" qui signifie que l'array
ne peut avoir que 3 éléments
let mut array: [i32; 3] = [42, 2048, 196883];

// On peut accéder aux éléments de l'array avec [0] [1] [2], etc
println!("{}", array[0], array[1], array[2]);

// Et voici comment modifier un array
array[2] = 24;
println!("{}", array[0], array[1], array[2]);
```

Bien sur on a le type booléen

```
// Valeurs possibles: true, false
let mon_booléen: bool = true;
```

```
println!("{}", mon_booleen);
```

Et enfin comme dernier type on va lister `char` pour contenir un caractère

```
let premier_a: char = 'A';

// Pour le deuxieme on va encoder la valeur UTF 8 de 'A' en hexadécimal
let deuxiem_a: char = '\u{0041}';

// On peut maintenant afficher les deux
println!("{}", et {} sont les même caractères.", premier_a, deuxiem_a);
```

Il reste encore deux types à traiter, les `slice` et les `reference` mais on va en parler dans un autre chapitre.

## Casting

On peut transformer un type primitif en un autre type primitif en utilisant le mot clé `as`, par exemple :

```
// On converti un f32 en i32 ici mais ça aurait pu être autre chose
let float: f32 = 67.957;
let int: i32 = float as i32;
println!("this is a float: {int}");
```

Pour les `&str` et `String` on peut utiliser `.parse()` :

```
let string_number = "67.957";
let float: f32 = string_number.parse().expect("The input is incorrect");
println!("this is a float: {float}");
```

C'est normal si vous ne comprenez pas encore tout de ce code car on a pas encore vu le `.expect()` (voir sur le chapitre de `Result` et `Option`) ni le `String`.

## En savoir plus

- [docs.rs - Primitive Types](#)

