Nous avons souvent utilisé les types texte principaux de **Rust** que sont **String**, **str** et **char** tout au long de nos différentes expérimentations. Cette étude va nous permettre de découvrir plus en détail comment traiter les textes dans leurs ensembles. Voici les principaux sujets que nous allons traiter :

- *Nous commencerons par quelques rappels au sujet du standard **Unicode**, ce qui nous aidera à comprendre les choix en vigueur dans la librairie standard.*
- *Nous décrirons ensuite le type **char** qui correspond à un point de code **Unicode**.*
- *Nous verrons ensuite les deux types **String** et **str** qui correspondent aux séquences de caractères **Unicode** possédées et empruntées. Ces deux types disposent d'une vaste famille de méthodes pour créer, chercher, modifier et balayer les contenus.*
- *Nous verrons ensuite des utilitaires pour contrôler le format des chaînes **Rust**, et notamment les deux macros **println!()** et **format!()**.*

Quelques rappels sur Unicode

Le standard **Unicode** est tellement riche qu'il mérite des livres entiers, mais cette étude ce concentre sur **Rust**. Les types caractères et chaîne de **Rust** ont été conçus en fonction du standard **Unicode**. Voici quelques rappels sur **Unicode** qui permettent de mieux comprendre **Rust**.

*Les valeurs **Unicode** sont les mêmes que les valeurs **ASCII**. Dans les deux codages, le symbole **« * »** correspond au code décimal **42**. **Unicode** utilise les mêmes valeurs de **0** à **0xFF** que le jeu de caractères **ISO/IEC 8859-1**, qui est le jeu **ASCII étendu** utilisé dans les langues d'Europe de l'Ouest.*

*Dans **Unicode**, cela correspond aux blocs de code **Latin-1 Basic** et **Supplement**, et nous utiliserons plutôt le terme **Latin-1** en remplacement de **ISO/IEC 8859-1**.*

*Puisque **Unicode** est un sur-ensemble de **Latin-1**, nous n'avons pas besoin de table spéciale pour convertir entre **Latin-1** et **Unicode.** La conversion inverse est facile aussi, lorsque les points de code restent dans les limites de la plage **Latin-1**.*

Fonctions de transformations

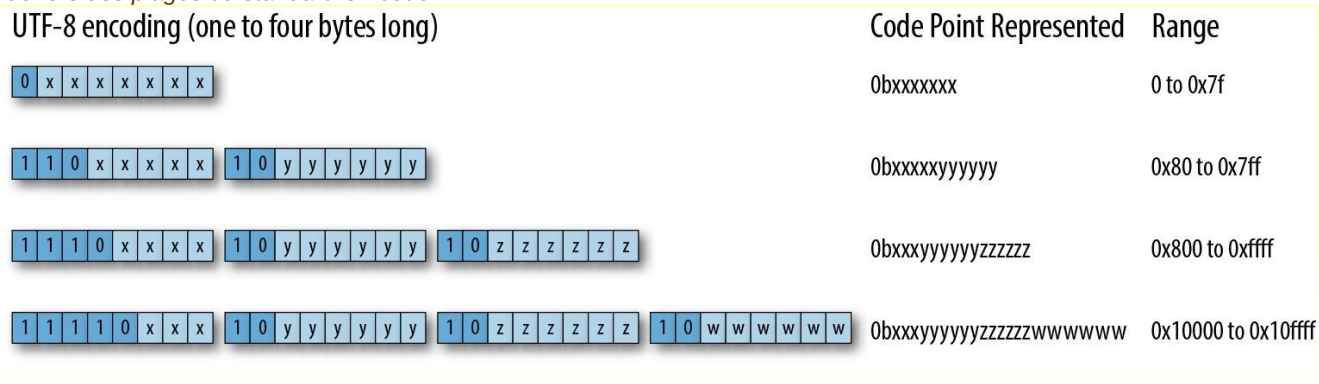
```
fn latin1_to_char(latin1: u8) -> char {
  latin1 as char
}

fn char_to_latin1(c: char) -> Option<u8> {
  if c as u32 <= 0xFF { Some(c as u8) }
  else { None }
}
```

UTF-8

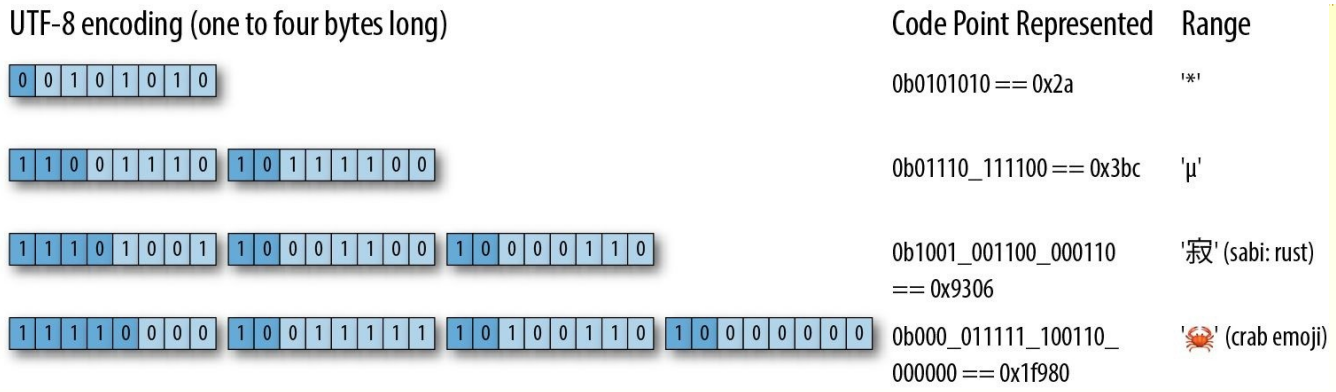
Les deux types **Rust** nommés **String** et **str** contiennent du texte codé selon la norme **UTF-8**. Chaque caractère est représenté par une séquence constituée de un à quatre octets. Le concept de séquence **UTF-8** bien formé comporte des restrictions.

*Tout d'abord, pour chaque point de code, seul l'encodage le plus court est considéré comme bien formé. Vous ne pouvez pas utiliser le codage sur quatre octets pour un point code qui tient dans trois seulement. Cela garantit qu'il n'existe qu'un seul codage **UTF-8** pour chaque point de code.*

*La seconde restriction est que les codes **UTF-8** corrects ne doivent pas coder la plage entre **0xD800** jusqu'à **0xDFFF** ni au-delà de **0x10FFFF**, car ce sont soit des zones réservées pour des non-caractères, soit ce sont des valeurs totalement en dehors des plages du standard Unicode.*



*Dans la page qui suit, nous proposons quelques exemples en suivant les critères proposés ci-dessus. Remarquez le dernier codage qui correspond à **l'emoji du crabe**. Son premier octet ne contient que des bits à zéro, ce qui n'apporte rien au point du code, mais il requiert néanmoins quatre octets : les encodages **UTF-8** sur trois octets ne permettent de coder que des valeurs sur **16 bits**, et notre valeur **0x1F980** fait **17 bits** de long.*



*Nos figures permettent de constater quelques propriétés très intéressantes de l'encodage **UTF-8**.*

- *Du fait que **UTF-8** encode les points de code entre **0** et **0x7F** exactement comme les octets de **0** à **0x7F** du codage **ASCII**, une série d'octets contenant du texte **ASCII anglais** est un codage **UTF-8** valide. Inversement, lorsqu'une chaîne **UTF-8** ne contient que des caractères du jeu **ASCII anglais**, le codage **UTF-8** est considéré comme de l'**ASCII** valide. Cette équivalence ne vaut plus pour la seconde moitié du jeu **ASCII Latin-1** : par exemple, en **Latin-1**, le **'é'** est codé par la valeur **0xE9**, alors que **UTF-8** considère cette valeur comme le premier octet d'un codage sur trois octets.*
- *Il suffit de tester les bits de poids fort de chaque octet pour savoir immédiatement s'il s'agit du début de l'encodage d'un caractère **UTF-8** ou d'un octet du corps de son encodage.*
- *Le premier octet d'un caractère permet de savoir la longueur de l'encodage, en analysant les bits de poids fort.*
- *Aucun encodage n'étant plus long que quatre octets, **UTF-8** ne demande jamais d'effectuer des traitements par boucle infinie, ce qui est fort pratique lorsque vous devez traiter des données dont vous n'êtes pas assuré de la source.*
- *Dans une séquence **UTF-8** bien formée, vous pouvez toujours déterminer où commence et où termine chaque caractère, même si vous commencez au milieu d'un flux d'octets. Le premier octet et les octets suivants en **UTF-8** sont toujours différents ; vous ne risquez pas de commencer la lecture au milieu d'un caractère.*

*Puisque le premier octet permet de connaître la longueur totale, aucun codage ne peut devenir le préfixe d'un autre. Cela offre d'intéressantes perspectives. Si vous cherchez par exemple dans une chaîne **UTF-8** un caractère délimiteur **ASCII**, il suffit de chercher l'octet de ce délimiteur.*

*Cet octet ne peut pas apparaître dans le corps d'un encodage sur plusieurs octets, puisque dans ces octets, il n'est pas du tout nécessaire de maintenir la structure **UTF-8** du premier. De même, un algorithme qui doit chercher une chaîne constituée d'un seul octet dans une autre n'a pas à être retouché pour traiter des chaînes **UTF-8**, et il n'est pas nécessaire d'examiner chaque octet du texte source.*

Bien sûr, un encodage de longueur variable est plus complexe qu'un encodage de longueur fixe, mais ces choix permettent à UTF-8 d'être plus facile à exploiter que ce que vous pourriez craindre. En effet, la librairie standard gère tous les détails à votre place.

Sens d'écriture des textes

Un certain nombre de langues comme le latin, le cyrillique ou le thaï s'écrivent bien de gauche à droite, mais d'autres comme l'hébreu ou l'arabe s'écrivent de droite à gauche. **Unicode** stocke les caractères dans l'ordre de l'écriture physique. Ainsi, une chaîne en hébreu contient comme premier caractère celui qui se trouve en fin de chaîne.

*Prenez bien note du fait que certaines des méthodes de la librairie standard portent des noms comportant la mention **left** ou **right**, mais cela signifie le début et la fin du texte, non la position physique. Nous donnerons plus de précisions lors de la description de ces fonctions particulières.*

Caractère (char)

Le type **char** de **Rust** est une valeur sur **32 bits** correspondant à un point de code Unicode. La valeur ne peut être que l'une de celles dans la plage **0** à **0xD7FF** ou la plage **0xE000** à **0x10FFFF** ; toutes les méthodes de création et de traitement de ces valeurs char le garantissent.

*Le type **char** implémente les traits **Copy** et **Clone**, ainsi que tous les traits utiles aux comparaisons, aux calculs de hachage et à l'application d'un format.*

*Le type **char** dispose d'une série de méthodes pour classer les caractères en catégories. Les noms de ces catégories sont inspirés de ceux utilisés dans **Unicode** :*

Méthode	Description	Exemples
ch.is_numeric()	Caractère numérique. Regroupe les catégories générales Unicode « Nombre ; chiffre » et « Nombre ; lettre », mais pas « Nombre ; autres ».	'4'.is_numeric()
ch.is-alphabetic()	Caractère alphabétique. Correspond à la propriété dérivée "alphabétique" de Unicode.	'q'.is_alphabetic().
ch.is_alphanumeric()	Regroupe les numériques et les alphabétiques.	'4'.is_alphanumeric(). 'q'.is_alphanumeric().

ch.is_whitespace()	Caractère espace, propriété Unicode nommée « WSpace=Y »	' '.is_whitespace(). '\n'.is_whitespace().
ch.is_control()	Caractère de contrôle. Catégorie Unicode « Autre, contrôle »	'\n'.is_control().

Gestion des chiffres pour chaque caractère (char)

Voici les méthodes permettant de manipuler les chiffres :

- ***ch.to_digit(base)*** *: permet de vérifier si* ***ch*** *est un chiffre dans la base mentionnée. Si c'est le cas, renvoie* ***Some(num)*** *pour indiquer la valeur en type* ***u32****. Renvoie* ***None*** *sinon. Ne détecte que les chiffres* ***ASCII****, pas la vaste classe de caractères correspondant à* ***char::is_numeric****. Le paramètre de la base numérique est autorisé entre* ***2*** *et* ***36****. pour les bases supérieures à la base décimal* ***10****, les lettres* ***ASCII*** *sont acceptées en majuscules et en minuscules, et équivalent à des valeurs entre* ***10*** *et* ***35****.*
- *La fonction libre* ***std::char::from_digit(num, base)*** *sert à convertir une valeur de chiffre* ***u32*** *vers le type* ***char*** *si possible. Si la valeur peut être représentée comme un seul chiffre dans la base demandée,* ***from_digit()*** *renvoie* ***Some(ch)****, avec le chiffre dans* ***ch****. Si la base est supérieure à* ***10****,* ***ch*** *peut être une minuscule. Si ce n'est pas un chiffre, renvoie* ***None****.*

  Cette fonction est donc l'inverse de ***to_digit()****. Si* ***std::char::from_digit(num, base)*** *renvoie* ***Some(ch)****, alors* ***ch.to_digit(base)****. Si* ***ch*** *est un chiffre* ***ASCII*** *ou une lettre minuscule, la conversion reste vrai.*
- ***ch.is_digit(base)*** *renvoie* ***true*** *si* ***ch*** *est un chiffre* ***ASCII*** *dans la base demandée. C'est l'équivalent de* ***ch.to_digit(base) != None****.*

Traitement de caractère isolé

```
fn main() {
  let phrase = "Dans cette phrase nous avons 6 mots.";
  let mut alphabetique = 0;
  let mut espace = 0;
  let mut chiffre = 0;
  for caractere in phrase.chars() {
    if caractere.is_alphabetic() { alphabetique+=1; }
    else if caractere.is_whitespace() { espace+=1; }
    else if caractere.is_numeric() {
      if let Some(x) = caractere.to_digit(10) { chiffre = x; }
    }
    else { println!("Caractère particulier ({})", caractere) }
  }
  println!("Alphabétiques={}, Espaces={}, Chiffre={}", alphabetique, espace, chiffre);
  if let Some(octal) = char::from_digit(chiffre, 8) {
    println!("Chiffre={}, Octal={}", chiffre, octal);
  }
}
```

Résultat

```
Caractère particulier (.)
Alphabétiques=28, Espaces=6, Chiffre=6
Chiffre=6, Octal=6
```

Conversion Maj/Min des caractères

Plusieurs méthodes permettent de modifier la casse des lettres (lettres majuscules ou lettres minuscules dites aussi « bas de casse »).

- ***ch.is_lowercase()*** *et* ***ch.is_uppercase()*** *: permettent de savoir si* ***ch*** *est un caractère alphabétique en majuscule ou en bas de casse. La discrimination obéit aux règles définies par* ***Unicode*** *et couvre donc également les alphabets non latins comme le grec ou le cyrillique, tout en conservant les résultats attendus en* ***ASCII*** *pur.*
- ***ch.to_lowercase()*** *et* ***ch.to_uppercase()*** *: renvoient un itérateur qui produit l'équivalent en minuscules ou en majuscules de* ***ch****, selon les normes de conversion* ***Unicode****. Toutes ces méthodes renvoient un itérateur et non directement un caractère parce que la conversion de casse en* ***Unicode*** *ne peut pas toujours résulter en un seul caractère à partir d'un caractère d'entrée. Pour plus de confort, ces itérateurs implémentent le trait* ***std::fmt::Display****, ce qui nous permet de les transmettre directement à une macro* ***println!()*** *ou* ***write!()****.*

Chemin d'accès absolu

```
fn main() {
  let bonjour = "Bonjour";
  for lettre in bonjour.chars() {
    print!("{}", lettre.to_uppercase());
  }
}
```

Résultat

```
BONJOUR
```

Conversion de et vers les entiers

L'opérateur **as** de **Rust** sait convertir une valeur char vers n'importe quel type entier, en éliminant en silence les bits de poids fort. Cet opérateur **as** peut convertir une valeur **u8** vers **char**, et **char** implémente de son côté **From<u8>**. En revanche, les types entiers de plus grande taille peuvent correspondre à des points de code invalides. Il faut donc utiliser **std::char::from_u32()** dans ce cas et qui renvoie un **Option<char>**.

Code ASCII

```
fn main() {
  println!("{}", 'A' as u8);
  println!("'{}'", char::from(65));
}
```

Résultat

```
65
'A'
```

Chaînes String et str

Nous savons maintenant que les deux types **String** et **str** sont certains de ne contenir que des valeurs **UTF-8** bien formées. C'est la librairie standard qui garantit cela en limitant ce que vous pouvez faire avec des valeurs **String** et **str** et quelles opérations vous pouvez leur appliquer.

*Une valeur bien formée le reste quels que soient vos traitements. Toutes les méthodes protègent cette vérité : aucune opération sûre ne peut rendre une séquence **UTF-8** invalide. Cela simplifie grandement le code de traitement de texte.*

***Rust** distribue les méthodes concernant les textes entre les deux types **str** et **String**, en fonction des besoins de traitement ; suivant qu'un tampon variable est nécessaire, ou que nous puissions traiter le texte en place. En effet, **String** se déréférence automatiquement en **&str** et chacune des méthodes définies pour **str** devient directement accessible aux valeurs de type **String**.*

Notez que ces méthodes utilisent des valeurs d'octets pour les décalages dans les indices et mesurent les longueurs en octets et non en nombre de caractères. L'indexation en unités de caractères n'est pas aussi pratique qu'il y paraît. Si vous proposez un décalage en octets et que le résultat vous fait arriver au milieu des octets d'un caractère UTF-8, la méthode va paniquer. Vous ne pouvez ainsi jamais rendre invalide une séquence UTF-8.

*Le type **String** est une couche autour de **Vec<u8>** qui garantit que le contenu du vecteur reste du code **UTF-8** bien formé. **Rust** ne va jamais modifier le type **String** pour introduire une représentation plus complexe. Vous pouvez donc espérer que **String** offre le même niveau de performance que **Vec**. Dans toutes les explications qui suivent, nous utilisons les variables suivantes avec les types indiqués :*

Variable	Type correspondant
string	(Chaîne) de type String.
slice	(Tranche) &str ou un type qui peut être déréférencé vers ce type, comme String ou Rc<String>.
ch	char.
n	usize, une longueur
I, j	usize, un décalage en octets
Plage	Une plage de décalage d'octets usize, soit bornée comme i..j, soit semi-bornée i.. , ..j ou ..
motif	Un type motif : char, String, &str, &[char] ou FnMut(char) → bool

Création de valeurs chaînes

V oici les méthodes permettant de créer une valeur de type **String** :

- ***String::new()** : renvoie une nouvelle chaîne vide, sans tampon associé dans le tas. Il sera implanté ultérieurement dès que nécessaire.*
- ***String::with_capacity(n)** : renvoie une chaîne vide avec un tampon pré-réservé pour accepter au moins **n** octets. Ce constructeur permet de dimensionner correctement le tampon dès le départ si vous connaissez sa taille future. La chaîne peut néanmoins plus de données si le tampon risque de déborder. Comme pour les vecteurs, vous disposez pour les chaînes des méthodes **capacity()**, **reserve()** et **shrink_to_fit()**, mais la réservation initiale est en général suffisante.*
- ***slice.to_string()** : crée une nouvelle chaîne à partir du contenu de la tranche **String**. Nous avons souvent utilisé les expressions suivantes **« texte litteral ».to_string()** pour créer une chaîne à partir d'un littéral.*
- ***iter.collect()** : construit une chaîne en concaténant les éléments produits par l'itérateur qui peuvent être du type char, **&str** ou **String**. Avec la méthode **collect()**, vous profitez de l'implémentation dont dispose **String** du trait **std::iter::FromIterator**.*
- *Sachez que le type **&str** ne peut pas implémenter le trait **Clone**, car celui-ci requiert **clone()** sur un type **&T** pour renvoyer une valeur **T** alors que **str** n'est pas à taille fixe. En revanche, **&str** implémente le trait **ToOwned** qui permet à l'implémentation de définir l'équivalent possédé. Ainsi, **slice.to_owned()** renvoie une copie de la tranche slice sous la forme d'une nouvelle chaîne.*

Création de chaînes

```
fn main() {
  let mut vide = String::new();
  let mut vide_aussi = String::with_capacity(20);
  let un_texte = "Un texte".to_string();
  let espaces = "Bien ve nue";
  let bienvenue : String = espaces.chars().filter(|c| !c.is_whitespace()).collect();
  let copie = espaces.to_owned();
  print!("'{}' ", vide);
  print!("'{}' ", vide_aussi);
  print!("'{}' ", un_texte);
  print!("'{}' ", bienvenue);
  print!("'{}' ", copie);
}
```

Résultat

```
'' '' 'Un texte' 'Bienvenue' 'Bien ve nue'
```

Inspection d'une chaîne

V oici les méthodes permettant d'obtenir quelques informations au sujet d'une tranche de chaîne :

- ***slice.len()*** *: renvoie la longueur de la tranche en octet.*
- ***slice.is_empty()*** *: renvoie* ***true*** *si* ***slice.len() == 0****.*
- ***slice[page]*** *: renvoie une tranche qui emprunte la partie demandée de* ***slice****. La plage peut être semi-ouverte ou ouverte.*
- *Vous ne pouvez pas prélever une tranche d'un seul caractère dans une chaîne, comme dans* ***slice[i].*** *vous devez vous servir d'un itérateur de* ***char*** *sur la tranche en lui demandant de récupérer un caractère au format* ***UTF-8****. En réalité, vous aurez rarement besoin de faire cela.* ***Rust*** *offre des moyens bien plus efficaces pour balayer des tranches, ce que nous verrons dans la section du même chapitre concernant l'itération.*
- ***slice.split_at(i)*** *: renvoie un tuple constitué de deux tranches partagées qui sont empruntées dans* ***slice*** *: la portion du début jusqu'à l'octet* ***i*** *puis l'autre portion. Cela équivaut à renvoyer* ***slice[..i]*** *et* ***slice[i..]****.*
- ***slice.is_char_boundary(i)*** *: renvoie* ***true*** *si l'octet en position* ***i*** *fait partie des valeurs comprises comme délimiteurs entre caractères et peut donc servir de point de décalage dans une tranche* ***slice****.*

Vous pouvez bien sûr comparer des tranches pour voir si elles sont égales, triées ou hachées. Dans le cas d'une comparaison triée, la chaîne est vue comme une séquence de point Unicode et la comparaison est faite dans l'ordre lexicographique.

Création de chaînes

```
fn main() {
  let poste = "ministériel";
  print!("{:?} ", &poste[..4]);
  print!("{:?} ", &poste[5..]);
  print!("{:?} ", &poste[2..4]);
  print!("{:?} ", &poste[..]);
  print!("({} octets) ", &poste[..].len());
  print!("{:?} ", &poste[6..8]); // 2 octets pour 'é'
  print!("({} octets) ", &poste[6..8].len());
  print!("{:?} ", &poste[6..].chars().next().unwrap()); // retrouver le caractère 'é'
  let (debut, fin) = poste.split_at(5);
  print!("({:?} - {:?})", debut, fin);
}
```

Résultat

```
"mini" "tériel" "ni" "ministériel" (12 octets) "é" (2 octets) 'é' ("minis" - "tériel")
```

Ajout et insertion de texte

V oici les méthodes permettant d'ajouter du texte dans une chaîne **String** :

- ***string.push(ch)*** *: ajoute le caractère* ***ch*** *à la fin de* ***string****.*
- ***string.push_str(slice)*** *: ajoute tout le contenu de la tranche* ***slice****.*
- ***string.extend(iter)*** *: ajoute les éléments produits par l'itérateur* ***iter*** *à la chaîne. Les valeurs peuvent être de type* ***char****,* ***str*** *ou* ***string****. Vous disposez d'une implémentation pour* ***String*** *de* ***std::iter::Extend****.*
- ***string.insert(i, ch)*** *: insère le caractère* ***ch*** *à l'octet* ***i*** *dans* ***string****. Provoque le décalage des caractères suivants. Construire une chaîne de cette façon peut nécessiter un délai égal au carré de la longueur de la chaîne.*
- ***string.insert_str(i, slice)*** *: fait la même chose pour une tranche avec le même inconvénient de lenteur.*

Insertion de parties de chaînes

```
fn main() {
  let mut texte = String::from("Un");
  texte.push('e');
```

```
  texte.push_str(" petite ");
  texte.extend("con tri bu tion".split_whitespace());
  texte.insert_str(10, " ou une grande");
  print!("{:?}", &texte);
}
```

Résultat

```
"Une petite ou une grande contribution"
```

*Le type **String** implémente **std::fmt::Write**, ce qui permet d'utiliser les macros **write!()** et **writeln!()** pour ajouter du texte formaté à une chaîne **String**.*

*Les deux macros **write!()** et **writln!()** sont conçues d'abord pour écrire vers des flux en sortie et renvoient donc une valeur **Result**, et **Rust** peut se plaindre si vous n'en tenez pas compte. Dans l'exemple ci-dessous, nous utilisons la méthode **unwrap()**, sachant qu'écrire en direction d'une chaîne en mémoire est normalement infaillible.*

Création d'une chaîne formatée en mémoire

```
use std::fmt::Write;

fn main() {
  let mut lettre = String::new();
  writeln!(lettre, "Un texte {} formaté", "entièrement").unwrap();
  println!("{}", lettre);
}
```

Résultat

```
Un texte entièrement formaté
```

*Le type **String** implémente également **Add<&str>** et **AddAssign<&str>** ce qui nous permet de réaliser une concaténation de plusieurs chaînes en utilisant des opérateurs adaptés à ce genre de situation.*

*Lorsqu'il est appliqué à des chaînes, l'opérateur de concaténation « **+** » utilise son opérande gauche par valeur, ce qui permet de réutiliser cet emplacement pour y stocker le résultat. Autrement dit, dès que le tampon de ce membre gauche est assez vaste pour contenir le résultat, il n'y a pas de perte de temps à réserver un nouvel espace mémoire. De ce fait, l'opérande gauche ne peut pas être de type **&str**.*

Cette contrainte enlève tout intérêt à la construction d'une grande chaîne en poussant les nouveaux morceaux par le début. Les performances sont en effet mauvaises, puisqu'il faut sans cesse décaler le texte existant vers l'arrière du tampon.

En revanche, créer une grande chaîne en ajoutant les sous-chaînes les unes à la suite des autres est efficace. Le type String fonctionne comme un vecteur et à chaque repositionnement mémoire, la taille est doublée. Cela maintient les opérations de recopie proportionnelle à la taille.

*Mais dès que possible, il est préférable d'utiliser **String::with_capacity()** pour créer au départ une chaîne ayant la taille qui sera nécessaire une fois les sous-chaînes ajoutées, car cela réduit le nombre d'appels au mécanisme d'allocation mémoire dans le tas.*

Concaténation de chaînes

```
fn main() {
  let gauche = "complices".to_string();
  let mut droite = "le crime".to_string();
  println!("{}", gauche + " dans " + &droite);
  droite += " ne paie pas";
  println!("{}", &droite);
}
```

Résultat

```
complices dans le crime
le crime ne paie pas
```

Suppression de texte

Le type **String** offre plusieurs méthodes pour enlever du contenu. Notez que ce n'a pas d'effet sur la longueur théorique de la chaîne ; pour libérer un peu de mémoire, il faut utiliser la méthode **shrink_to_fit()**.

- ***string.clear()** : vide la chaîne.*
- ***string.truncate(n)** : supprime tous les caractères à partir de **n**. N'a aucun effet si la chaîne est déjà plus courte.*
- ***string.pop()** : enlève le dernier caractère de la chaîne en le renvoyant en tant que valeur **Option<char>**.*
- ***string.remove(i)** : enlève le caractère à la position **i** et le renvoie en ramenant tous les caractères suivants d'une position. La durée de l'opération est proportionnelle au nombre de caractères à ramener.*
- ***string.drain(plage)** : renvoie un itérateur sur la plage d'indices fournie et supprime les caractères une fois l'itérateur consommé. Les caractères qui suivent la plage sont ramenés vers le début. Si le but est de supprimer la plage de contenu, il suffit d'abandonner immédiatement l'itérateur sans chercher à récupérer les éléments qu'il génère.*

Suppression de parties de chaînes

```
fn main() {
  let mut texte = "chocolats".to_string();
  texte.pop();
  println!("{}", &texte);
  texte.truncate(5);
  println!("{}", &texte);
  texte+="lat";
  texte.drain(3..6);
  println!("{}", &texte);
}
```

Résultat

```
chocolat
choco
choat
```

Syntaxes particulières pour les recherches et les itérations

La famille de fonctions standard **Rust** pour chercher du texte et itérer dans du texte obéit à quelques conventions de nommage afin d'être plus facile à mémoriser :

- *La plupart des opérations travaillent du début à la fin de la chaîne ; les opérations dont le nom commence par la lettre « **r** » travaillent dans l'autre sens. C'est ainsi que **rsplit()** est la version de la fin vers le début de **split()**.*
- *Les itérateurs dont le nom se termine par un « **n** » sont limités à un certain nombre de tours.*
- *Les itérateurs dont le nom se termine par **_indices()** produisent en plus des valeurs de l'itérateur un décalage en octets dans la tranche correspondante à la position.*

Toutes les opérations ne sont pas disponibles. La variante limitée par n n'est par exemple pas utile pour de nombreuses opérations, puisqu'il est facile d'arrêter d'itérer dès que nécessaire.

Motifs de recherche de texte

Les fonctions standard de recherches, de sélection, de raccourcissement ou de découpage acceptent plusieurs types de valeurs servant de motifs. Voici les quatre genres de motifs reconnus par la librairie standard.

- *Un motif de type **char** fait chercher ce caractère.*
- *Un motif de type **String** ou **&str** ou **&&str** permet de trouver une sous-chaîne.*
- *Un motif sous forme de clôture **FnMut(char) → bool** permet de trouver un caractère pour lequel la clôture renvoie **true**.*
- *Un motif **&[char]** (pas **&str**, une vrai tranche de valeurs **char**) trouve n'importe quel caractère faisant partie de la liste. Si vous fournissez la liste sous forme d'un tableau littéral, vous devrez peut-être utiliser une expression as pour obtenir le type attendu.*

Recherche dans une chaîne

```
fn main() {
  let texte = "Un beau jour, ou peut-être une nuit";
  if let Some(virgule) = texte.find(',') {
    println!("Position de ',' : {}", virgule);
  }
  if let Some(jour) = texte.find("jour") {
    println!("Position de 'jour' : {}", jour);
  }
  if let Some(espace) = texte.find(char::is_whitespace) {
    println!("Position de ' ' : {}", espace);
  }
  let original = "\n  Bonjour";
  let debut = original.trim_start_matches(&[' ', '\n'] as &[char]);
  let fin = debut.trim_end_matches(&['o', 'j', 'u', 'r'][..]); // as &[char] <=> [..]
  println!("{} > {}", debut, fin);
  let enlever_espaces = original.trim_start_matches(|c: char| c=='\n' || c.is_whitespace());
  print!("{}", enlever_espaces);
}
```

Résultat

```
Position de ',' : 12
Position de 'jour' : 8
Position de ' ' : 2
Bonjour > Bon
Bonjour
```

Recherches et remplacements

R ust propose une poignée de méthodes pour rechercher un motif dans une tranche et le remplacer par un autre texte.

- ***slice.contains(motif)*** *: renvoie* ***true*** *si la tranche contient le motif.*
- ***slice.starts_with(motif)*** *et* ***slice.ends_with(motif)*** *: renvoient* ***true*** *si le début ou la fin du texte de la tranche correspond au motif.*
- ***slice.find(motif)*** *et* ***slice.rfind(motif)*** *: renvoient* ***Some(i)*** *si la tranche contient le motif, i indiquant l'offset en octets de la position du motif. La méthode find() trouve la première occurrence et l'autre la dernière.*
- ***slice.replace(motif, remplacement)*** *: renvoie une nouvelle chaîne après remplacement de toutes les occurrences du motif par remplacement.*
- ***slice.replacen(motif, remplacement)*** *: fait comme la précédente en n'effectuant que* ***n*** *remplacements.*

Recherche dans une chaîne

```
fn main() {
  println!("Numérique : {}", "2021".starts_with(char::is_numeric));
  let mut lavons = "Nous savons user du savon.";
  if let Some(savon) = lavons.find("savon") {
    println!("Position de 'savon' : {}", savon);
  }
  if let Some(savon) = lavons.rfind("savon") {
    println!("Position de 'savon' : {}", savon);
  }
  println!("Position de 'un savon' : {:?}", lavons.find("un savons"));
  if let Some(majuscule) = lavons.rfind(char::is_uppercase) {
    println!("Position de la majuscule : {}", majuscule);
  }
  println!("{}", lavons.replace("user", "utiliser"));
  println!("{}", "Con fiden tiel".replace(|ch: char| !ch.is_alphabetic(), ""));
}
```

Résultat

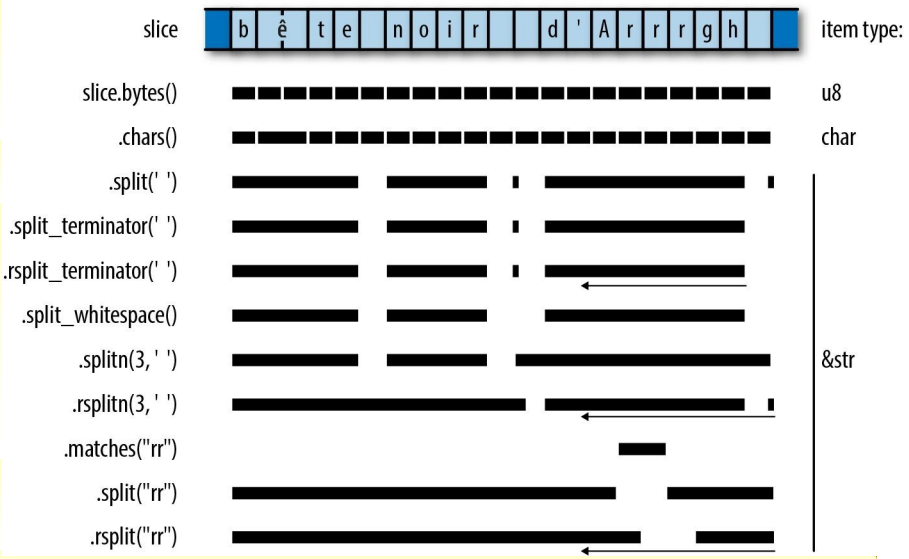
```
Numérique : true
Position de 'savon' : 5
Position de 'savon' : 20
Position de 'un savon' : None
Position de la majuscule : 0
Nous savons utiliser du savon.
Confidentiel
```

Itération ou balayage de texte

Les deux familles **split()** et **match** sont complémentaires l'une de l'autre : les découpages **split()** correspondent aux plages entre sélections **match**. Pour certains motifs, le fait de commencer par la fin de la chaîne a un effet sur les valeurs produites.

*Voyez par exemple les motifs basés sur « **rr** » dans la figure. Le problème ne se pose pas lorsque l'itérateur produit le même jeu d'éléments dans les deux sens, c'est-à-dire que lorsque l'ordre seulement change.*

Dans ce cas, l'itérateur est de type ***DoubleEndedIterator*** *et vous pouvez appliquer sa méthode* ***rev()*** *pour progresser dans les deux sens et extraire des éléments d'un côté ou de l'autre.*



- ***slice.chars()*** *: renvoie un itérateur sur les caractères de la tranche.*
- ***slice.char_indices()*** *: renvoie un itérateur sur les caractères de la tranche avec leur décalage en octets. Vous aurez remarqué que ce n'est pas identique à* ***chars().enumerate()****, car nous fournissons le décalage en octets de chaque caractère dans la tranche au lieu de simplement numéroter les caractères.*
- ***slice.bytes()*** *: renvoie un itérateur sur les octets individuels de la tranche en montrant l'encodage* ***UTF-8****.*
- ***slice.lines()*** *: renvoie un itérateur sur les lignes de la tranche slice. Chaque ligne se termine par « **\n** » ou « **\r\n** ». chacun des éléments produits est une chaîne* ***&str*** *provenant de la tranche et les caractères de fin de ligne ne sont pas inclus dans la réponse.*
- ***slice.split(motif)*** *: renvoie un itérateur sur les portions de la tranche qui sont séparées par des occurrences du motif. Lorsque deux motifs se touchent, la chaîne résultante est vide, ainsi qu'au début et à la fin de la tranche.*
- ***slice.rsplit(motif)*** *: est la même que la précédente sauf qu'elle commence à partir de la fin de la chaîne.*

- ***slice.split_terminator(motif)*** *et* ***slice.rsplit_terminator(motif)*** *: sont des variantes dans lesquelles le motif sert de marque de fin et non de séparateur. Lorsque le motif est trouvé à la fin de la tranche, ses itérateurs ne produisent pas une tranche vide correspondant à la fin de la tranche sans le motif, alors que c'est ce que font* ***split()*** *et* ***rsplit()****.*
- ***slice.splitn(n, motif)*** *et* ***slice.rsplitn(n, motif)*** *: ressemblent à* ***split()*** *et à* ***rsplit()*** *sauf qu'elles découpent la chaîne en maximum* ***n*** *tranches, en s'arrêtant à la première ou à la dernière occurrence* ***n-1*** *du motif.*
- ***slice.split_whitespace()*** *: renvoie un itérateur sur les parties de la tranche séparées par l'espace au sens large. Plusieurs espaces de suite sont considérées comme une seule et les espaces de fin de chaîne sont ignorées. Cette méthode utilise la même définition de l'espace que* ***char::is_whitespace****.*
- ***slice.matches(motif)*** *: renvoie un itérateur sur les correspondantes du motif dans la tranche. Sa collègue* ***slice.rmatches(motif)*** *travaille depuis la fin de la chaîne.*
- ***slice.match_indices(motif)*** *et* ***slice.rmatch_indices(motif)*** *: sont apparentées sauf que les éléments produits sont des paires* ***(décalage, match)****, le décalage correspondant à la valeur de l'octet de début du motif et* ***match*** *à la tranche correspondant au motif.*

Itération et balayage de texte

```
fn main() {
  println!("{:?}", "élan".char_indices().collect::<Vec<_>>());
  println!("{:?}", "élan".bytes().collect::<Vec<_>>());
  let lignes = "Un\nDeux\nTrois";
  for ligne in lignes.lines() {
    println!("{}", ligne);
  }
  let delimiteur = "Un:Deux:Trois";
  println!("{:?}", delimiteur.split(':').collect::<Vec<_>>());
  println!("{:?}", lignes.split_terminator('\n').collect::<Vec<_>>());
  let lignes = lignes.to_string() + " Beaucoup";
  println!("{:?}", lignes.split_whitespace().collect::<Vec<_>>());
}
```

Résultat

```
[(0, 'é'), (2, 'l'), (3, 'a'), (4, 'n')]
[195, 169, 108, 97, 110]
Un
Deux
Trois
["Un", "Deux", "Trois"]
["Un", "Deux", "Trois"]
["Un", "Deux", "Trois", "Beaucoup"]
```

Raccourcissements (trim)

Raccourcir une chaîne consiste à enlever du texte, en général, des espaces, au début ou à la fin. Cela permet notamment de nettoyer une chaîne reçue par saisie d'un utilisateur ou d'un fichier, la saisie comportant parfois des espaces pour augmenter la lisibilité.

- ***slice.trim()*** *: renvoie une sous-tranche après suppression des espaces au début et à la fin.*
- ***slice.trim_start()*** *: n'enlève que les espaces au début.*
- ***slice.trim_end()*** *: n'enlève que les espaces de la fin.*
- ***slice.trim_matches(motif)*** *: renvoie une sous-tranche après suppression de toutes les occurrences du motif au début et à la fin.*
- ***slice.trim_start_matches(motif)*** *: ne traite les occurrences du motif au début.*
- ***slice.trim_end_matches(motif)*** *: ne traite les occurrences du motif à la fin.*

Itération et balayage de texte

```
fn main() {
  let heure = "  08:30  ";
  let heure_sans_espaces = heure.trim();
  let heure_sans_zero = heure_sans_espaces.trim_start_matches('0');
  println!("{}", heure_sans_espaces);
  println!("{}", heure_sans_zero);
}
```

Résultat

```
08:30
8:30
```

Changement de casse dans les chaînes

Pour les conversions entre minuscules et majuscules, les deux méthodes **slice.to_uppercase()** et **slice.to_lowercase()** sont à l'œuvre. Notez que le résultat peut ne pas être la même longueur que slice comme décrit dans la section antérieure de ce chapitre qui abordait la casse des caractères.

Analyse d'autres types de chaînes

Rust propose quelques traits pour chercher des valeurs particulières dans des chaînes et en obtenir une représentation textuelle. Dès qu'un type implémente le trait **std::str::FromStr**, il vous procure un moyen standardisé de chercher une valeur dans une tranche de chaîne.

Trait FromStr

```
pub trait FromStr : Sized {
  type Err;
  fn from_str(s: &str) -> Result<Self, Self::Err>;
}
```

*Tous les types machine prédéfinis implémentent ce trait **FromStr**. Le type spécialisé **std::net::IpAddr** qui est une énumération pouvant accueillir une adresse **IPv4** ou **IPv6** implémente aussi ce trait **FromStr**.*

*Les contenus des chaînes disposent d'une méthode nommée **parse()** qui permet de transformer la tranche en un type choisi, à condition que ce type implémente **FromStr**. Dans certains cas, comme pour **Iterator::collect()**, vous devez spécifier le type désiré, ce qui fait que **parse()** n'offre pas une lisibilité meilleure que lorsque vous devez appeler directement **from_str()**.*

Chaînes associées à des valeurs numériques

```
use std::str::FromStr;
use std::net::IpAddr;

fn main() {
  if let Ok(nombre) = usize::from_str("368800") {
    println!("usize = {}", nombre);
  }
  if let Ok(nombre) = f64::from_str("128.5625") {
    println!("f64 = {}", nombre);
  }
  if let Ok(test) = bool::from_str("true") {
    println!("bool = {}", test);
  }
  println!("Erreur = {:?}", f64::from_str("Non flottant"));
  println!("(Non flottant) n'est pas un nombre' = {}", f64::from_str("Non flottant").is_err());
  println!("Valeur = {}", match u32::from_str("45") {
    Ok(nombre) => nombre,
    Err(_) => 0
  });
  let adresse = IpAddr::from_str("172.16.10.23");
  println!("{:?}", adresse);
}
```

Résultat

```
usize = 368800
f64 = 128.5625
bool = true
Erreur = Err(ParseFloatError { kind: Invalid })
(Non flottant) n'est pas un nombre' = true
Valeur = 45
Ok(172.16.10.23)
```

Conversion d'autres types vers les chaînes

Trois autres techniques permettent de convertir une valeur qui n'est pas du texte vers une chaîne :

- *Tous les types pour lesquels une valeur peut être lisible par un humain peut implémenter le trait **std::fmt::Display** avec lequel vous pouvez vous servir du formateur {} dans la macro **format!()**.*

  *Par exemple, tous les types numériques standard de **Rust** implémentent **Display**, tout comme les caractères, les chaînes et les tranches. Les types pointeur intelligent **Box<T>**, **Rc<T>** et **Arc<T>** implémentent aussi **Display** si le **T** l'implémente ; le format résultant est celui de leur cible. Les conteneurs comme **Vec** et **HashMap** n'implémentent pas **Display** car il n'y a aucune forme humainement lisible pour ces types.*

- *Dès qu'un type implémente **Display**, la librairie standard implémente pour lui le trait **std::str::ToString** qui n'offre qu'une seule méthode, **to_string()**, pratique dès que vous n'avez pas besoin de la souplesse de **format!()**.*

  *Le trait **ToString** est antérieur à **Display**, et est moins souple. Pour les types que vous définissez, préférez **Display**.*

- *Tous les types publics de la librairie standard implémentent le trait **std::fmt::Debug** qui reformate la valeur fournie pour la présenter en tant que chaîne utile au débogage. La façon habituelle d'utiliser **Debug** consiste à exploiter le formateur de la macro **format!()** qui s'écrit **{:?}**.*

  *Nous profitons ici d'une implémentation de **Debug** qui enveloppe le type **Vec<T>**, pour tout type **T** qui implémente aussi **Debug**. Tous les types de collection **Rust** en disposent. Pensez à implémenter **Debug** pour vos propres types. En général, demandez à **Rust** de dériver l'implémentation comme nous l'avons fait pour la structure **Complexe**.*

Ces deux traits de formatage Display et Debug font partie d'une famille complète en lien avec la macro format!(). Nous verrons tous les autres et montrerons comment les implémenter dans une section ultérieurement dans cette même étude consacrée au formatage.

Génération de chaînes à partir d'autres types

```
use std::net::IpAddr;

#[derive(Debug)]
struct Complexe {reel: f64, imaginaire: f64}

fn main() {
  let message = format!("{} = {}", "Valeur", 15);
  println!("{}", message);
  let calcul = format!("{:.3}, {:.3}", 0.5, f64::sqrt(3.0)/2.0);
  println!("{}", calcul);
  println!("{}", format!("{}", IpAddr::from([172, 16, 10, 21])));
  println!("{}", IpAddr::from([172, 16, 10, 21]).to_string());
  println!("{:?}", IpAddr::from([172, 16, 10, 21]));
  let i = Complexe{ reel: 0.0, imaginaire: 1.0};
  println!("{:?}", i);
}
```

Résultat

```
Valeur = 15
0.500, 0.866
172.16.10.21
172.16.10.21
172.16.10.21
Complexe { reel: 0.0, imaginaire: 1.0 }
```

Accès à du texte en tant que UTF-8

Deux méthodes permettent d'accéder aux octets qui représentent du texte, le choix entre les deux se faisant en fonction de votre besoin de prendre possession des octets ou juste de les emprunter.

- ***slice.as_bytes()*** *: emprunte les octets de* ***slice*** *en tant que* ***&[u8]****. La référence étant non modifiables,* ***slice*** *peut supposer que les octets resteront en* ***UTF-8*** *bien formés.*
- ***string.into_bytes()*** *: prend possession de* ***string*** *pour renvoyer un* ***Vec<u8>*** *par valeur constitué des octets de la chaîne. C'est une conversion simplifiée : elle se contente de renvoyer le vecteur* ***Vec<u8>*** *que la chaîne utilisait comme tampon. Après l'opération, la chaîne n'existe plus, ce qui fait qu'il est inutile de vouloir continuer à maintenir les octets au format* ***UTF-8****. L'appelant est donc libre de modifier le contenu de* ***Vec<u8>*** *selon ses besoins.*

Tableau ou vecteur d'octets de chaînes

```
fn main() {
  let bienvenue = "Bienvenue".as_bytes();
  let bonjour = "Bonjour".to_string().into_bytes();
  println!("tableau = {:?}", bienvenue);
  println!("vecteur = {:?}", bonjour);
}
```

Résultat

```
tableau = [66, 105, 101, 110, 118, 101, 110, 117, 101]
vecteur = [66, 111, 110, 106, 111, 117, 114]
```

Production de texte à partir de données UTF-8

En partant d'un bloc d'octets qui contient des données **UTF-8** valables, plusieurs options sont disponibles pour obtenir des chaînes **String** ou des **tranches**. Vous choisirez en fonction de la façon dont vous voulez gérer les erreurs.

- ***str::from_utf8(byte_slice)*** *: reçoit une tranche d'octets* ***&[u8]*** *pour renvoyer un* ***Result*** *qui vaut soit* ***Ok(&str)*** *si le paramètre contient du code* ***UTF-8*** *correct, soit une erreur.*
- ***String::from_utf8(vec)*** *: tente de construire une chaîne en partant d'une valeur* ***Vec<u8>*** *reçue par valeur. Si le contenu était du* ***UTF-8****, la méthode renvoie* ***Ok(string)****, et la chaîne renvoyée possède* ***vec*** *pour s'en servir comme tampon. Aucune réservation dans le tas ni une copie du texte ne sont réalisées.*

  Si le contenu n'est pas du codage ***UTF-8****, la méthode renvoie* ***Err(e)****, la valeur e étant choisie dans* ***FromUtf8Error****. Vous pouvez récupérer le vecteur d'origine par un appel à* ***e.into_bytes()****, car il n'est pas perdu en cas d'erreur de conversion.*
- ***String::from_utf8_lossy(byte_slice)*** *: tente de construire une valeur* ***String*** *ou* ***&str*** *à partir de la tranche d'octets partagée* ***&[u8]****. La conversion réussit toujours, en substituant à tous les codes* ***UTF-8*** *mal formés un caractère de remplacement* ***Unicode****.*

  La valeur renvoyée est de type ***Cow<str>*** *et elle emprunte soit une* ***&str*** *directement depuis la tranche d'entrée* ***byte_slice*** *si le contenu était du* ***UTF-8****, soit une nouvelle chaîne* ***String*** *après remplacement des octets défectueux par*

ceux du remplacement. Autrement dit, si le paramètre d'entrée était bien formé, il n'y a pas de réservation dans le tas, ni de copie.

- *Si vous êtes certain que votre **Vec<u8>** contient de l'UTF-8 bien formé, vous pouvez tenter d'appeler la fonction non sûre **String::from_utf8_unchecked(vec)**. Elle se contente d'emballer l'élément **Vec<u8>** dans une chaîne **String** qu'elle renvoie sans vérifier le contenu. C'est à vous de vous assurer que vous n'avez pas inséré ainsi des codes **UTF-8** mal formés, et c'est pourquoi la fonction comporte la mention **unsafe**.*
- *Sa collègue **str::from_utf8_unchecked(slice)** traite un élément **&[u8]** pour le renvoyer sous forme d'une **&str** sans vérifier le contenu. Ici aussi, c'est à vous de vérifier que l'opération ne pollue pas vos données.*

Chaînes à partir de tableaux d'octets

```
use std::*;

fn texte_from_utf8(octets: Vec<u8>) {
  match String::from_utf8(octets) {
    Ok(chaine) => println!("Valide : {}", chaine),
    Err(erreur) => println!("Non valide : {:X?}", erreur.into_bytes())
  }
}

fn main() {
  let octets = [66, 111, 110, 106, 111, 117, 114];
  let vecteur = vec![66, 111, 110, 106, 111, 117, 114];

  if let Ok(bonjour) = str::from_utf8(&octets) {
    println!("{}", bonjour);
  }
  if let Ok(bonjour) = str::from_utf8(&vecteur) {
    println!("{}", bonjour);
  }
  texte_from_utf8(vec![0xF0, 0x9F, 0xA6, 0x80]); // emoji crabe
  texte_from_utf8(vec![0x9F, 0xF0, 0xA6, 0x80]); // n'importe quoi
  println!("{}", String::from_utf8_lossy(&[65, 111, 195, 187, 116]));
  println!("{:X?}", "Août".as_bytes());
  println!("{}", String::from_utf8_lossy(&[0x41, 0x6F, 0xC3, 0xBB, 0x74]));
}
```

Résultat

```
Bonjour
Bonjour
Valide :
Non valide : [9F, F0, A6, 80]
Août
[41, 6F, C3, BB, 74]
Août
```

Formatage de valeurs

Nous avons souvent utilisé des macros de formatage de texte, notamment **println!()**. Le littéral chaîne sert de modèle pour le résultat : chacun des blocs **{...}** dans ce modèle est remplacé par la valeur d'un des paramètres après formatage.

*La chaîne modèle doit donc être une constante pour que **Rust** puisse la confronter aux types des paramètres dès la compilation. Tous les paramètres doivent être utilisés, sous peine d'erreur de compilations. Plusieurs macros de la librairie standard s'appuient sur ce petit langage de formatage pour les chaînes :*

- *La macro **format!()** s'en sert pour construire des valeurs **String**.*
- *Les macros **println!()** et **print!()** affichent du texte formaté sur la sortie standard.*
- *Les macros **writeln!()** et **write!()** écrivent le texte formaté dans le flux de sortie indiqué.*
- *La macro **panic!()** s'en sert pour construire une expression informant sur l'abandon précipité d'exécution.*

*Le sous-système de formatage de **Rust** est ouvert, c'est-à-dire que vous pouvez enrichir les macros pour qu'elles traitent vos types. Il suffit d'implémenter les traits de formatage du module **std::fmt**. Vous disposez également de la macro **format_args!()** et du type **std::fmt::Arguments** pour que vos propres fonctions et macros exploitent ce langage de format.*

Les macros de formatage utilisent toujours des références partagées empruntées pour leurs paramètre et ne prennent jamais possession d'eux, ni ne les modifient.

*Dans le modèle, les blocs **{...}** sont des paramètres de format qui se basent sur la syntaxe **{lequel:format}**. Ces deux éléments sont facultatifs et la paire d'accolades vide **{}** est souvent utilisée.*

*L'élément **lequel** permet de désigner lequel des paramètres dans le modèle doit être inséré à cette place. Vous pouvez désigner le paramètre par un indice ou par son nom. Lorsque le rang n'est pas indiqué avec **lequel**, les paramètres sont pris dans l'ordre de gauche à droite.*

*La valeur **format** (comment) détermine le format à appliquer : remplissage, précision numérique, base numérique, etc. Si cette valeur est fournie, le signe deux-points doit la précéder.*

Modèles de chaîne	Liste de paramètres	Résultat
« nombre de {} : {}	« éléphants », 19	« nombre de éléphants : 19 »
« du {1} au {0} »	“berceau”, “tombeau”	« du berceau au tombeau »
« v = {:?} »	vec![0, 1, 2, 5, 12, 29]	« v = [0, 1, 2, 5, 12, 29] »
« nom = {:?} »	« Nemo »	« nom = ‘Nemo’ »
« {:8.2} km/s »	11.186	« 11.19 km/s »
«{:20} {:02x} {:02x} »	« adc #42 », 105, 42	« adc #42 69 2a »
« {1:02x} {2:02x} {0} »	« adc #42 », 105, 42	« 69 2a adc #42 »
« {lsb:02x} {msb:02x} {isns} »	Insn= « adc #42 », lsb=105, msb=42	« 69 2a adc #42 »

*Lorsque vous avez besoin de faire apparaître les deux accolades dans le résultat formaté, il faut les redoubler dans le modèle **(‘{‘ ou ‘}’)**.*

Formatage d’affichage

```
struct Complexe {
  reel: f64,
  imaginaire: f64
}

fn main() {
  let i = Complexe { reel: 0.0, imaginaire: 1.0 };
  println!("{{reel={reel:.2}, imaginaire={imaginaire:0.2}}}", reel=i.reel, imaginaire=i.imaginaire);
}
```

Résultat

```
{reel=0.00, imaginaire=1.00}
```

Formatage de valeurs textes

Pour les types texte comme **&str** ou **String** (le type **char** est compris comme une chaîne d’un seul caractère), la valeur **format** ou formateur est constituée de plusieurs parties, toutes facultatives.

- *Une **longueur maximale**. Le paramètre est tronqué s’il est plus long. Si rien n’est mentionné, tout le texte est utilisé.*
- *Une **largeur de champ minimale**. Après troncature éventuelle, si le paramètre est moins long que cette valeur, **Rust** ajoute à droite des espaces par défaut pour obtenir la bonne longueur. Rien n’est fait en l’absence d’une valeur.*
- *Un **alignement**. Si le paramètre doit subir un remplissage pour une longueur minimale, cette valeur d’alignement permet de choisir d’aligner à gauche, à droite ou de centrer, avec les trois symboles suivants « **< ^ >** ».*
- *Un **caractère de remplissage** à utiliser à la place de l’espace. Si vous spécifiez ce caractère, vous devez également préciser l’alignement.*

Fonction de formatage	Chaîne modèle	Résultat avec le texte « partages »
Par défaut	« {} »	« partages »
Largeur de champ minimale	“{:4}”	« partages »
	« {:12} »	« partages »
Limite de longueur de texte	« { :.4} »	« part »
	« { :.12} »	« partages »
Largeur de champ, Limite de longueur	« {:12.20} »	«partages »
	« {:4.20} »	« partages »
	« {:4.6} »	« parta »
	« {:6.4} »	« part »
Aligné à gauche, largeur	« {:<12} »	« partages »
Centré, largeur	« {:^12} »	« partages »
Aligné à droite, largeur	« {:>12} »	« partages »
Rempli avec ‘=’ centré, largeur	« {:=^12} »	« ==partages== »
Rempli avec ‘*’ aligné à droite, largeur, limite	« {:*>12.4} »	« **********part »

*Le mécanisme de formatage de **Rust** est très naïf au niveau des longueurs du contenu des chaînes. Il suppose qu'à chaque caractère correspond une seule position, et ne tient donc pas compte des caractères exotiques, des katakanas, des espaces à largeur nulle et autres réalités du vaste monde d'Unicode. Voici deux exemples :*

Formatage d'affichage

```
fn main() {
  println!("{:4}", "th\u{E9}");
  println!("{:4}", "the\u{301}");
}
```

Résultat

```
thé
thé
```

*Pour **Unicode**, ces deux chaînes sont équivalent et signifient le mot « **thé** ». Le formateur de **Rust** ne sait pas que le code **'\u{301}'**, **COMBINING ACUTE ACCENT**, est une lettre morte pour l'accent aigu. Il gère correctement la première des deux chaînes ci-dessus, mais pense que la seconde chaîne occupe quatre positions.*

*Les chemins d'accès et noms de fichier ne sont pas obligatoirement en code **UTF-8** correct. D'ailleurs, **std::path:Path** n'est pas un vrai type texte et vous ne pouvez pas le transmettre directement à une macro de format. Ceci dit, la méthode **display()** d'un tel chemin renvoie une valeur qui sera formatée correctement en fonction de la plate-forme.*

Contrôle des formats numériques

Lorsque le type d'un paramètre de format est numérique, comme **usize** ou **f64**, le membre **format** du paramètre peut contenir les éléments suivants, qui sont tous facultatifs :

- *Un **remplissage** et un **alignement :** fonctionnant comme pour les types texte.*
- *Un **caractère +** qui demande à toujours afficher le signe de la valeur, même si elle est positive.*
- *Un **caractère #** qui demande d'ajouter un préfixe de base numérique comme **0x** ou **0b**.*
- *Un **caractère 0** qui demande d'ajouter des zéros pour atteindre la largeur minimale demandée, au lieu d'utiliser le remplissage normal.*
- *Une **largeur minimale.** Si la valeur n'est pas suffisante après formatage, **Rust** ajoute des espaces à gauche par défaut.*
- *Une indication de **précision** pour les valeurs à virgule flottante, qui correspond au nombre de chiffres après la virgule (le point décimal). **Rust** arrondit ou ajoute des zéros pour atteindre le nombre demandé. S'il n'y a pas de précision, **Rust** utilise le moins de chiffres possible pour représenter la valeur complète. Ce paramètre est ignoré dans le cas des valeurs entières.*
- *Une **notation**. Pour les types entiers, le préfixe est un « **b** » pour le binaire, un « **o** » pour l'octal ou bien « **x** » ou un « **X** » pour l'hexadécimal. Si vous avez spécifié « **#** », la base numérique est affiché au format **Rust**, **0b**, **0o**, **0x** ou **0X**. Pour les types à virgule flottante, la base est « **e** » ou « **E** » pour la notation scientifique avec coefficient normalisé, la lettre correspondant à l'exposant. Si rien n'est mentionné pour la base, **Rust** utilise bien sûr la base décimale.*

Fonction de formatage	Chaîne modèle	Résultat avec l'entier 1234
Par défaut	« {} »	« 1234 »
Signe systématique	"{:+}"	« +1234 »
Largeur de champ minimale	"{:2}"	« 1234 »
	« {:12} »	« 1234 »
Signe, largeur	« { :+12} »	« +1234 »
Zéros de remplissage, largeur	« {:012} »	« 000001234 »
Signe, Zéros, largeur	« {:+012} »	« +00001234 »
Aligné à gauche, largeur	« {:<12} »	« 1234 »
Centré, largeur	« {:^12} »	« 1234 »
Aligné à droite, largeur	« {:>12} »	« 1234 »
Aligné à gauche, signe, largeur	"{:<+12}"	« +1234 »
Centré, signe, largeur	"{:^+12}"	« +1234 »
Aligné à droite, signe, largeur	« {:>+12} »	« +1234 »
Rempli par '=', centré, largeur	« {:=^12} »	« ===1234=== »
Binaire	« {:b} »	« 10011010010 »
Largeur octal	« {:12o} »	« 2322 »
Signe, largeur hexadécimal	« {:+12x} »	« +4d2 »

Signe, largeur, hexadécimal en capitales	« {:+12X} »	« +4D2 »
Signe, préfixe de base explicite, largeur, hexa	« {:+#12X} »	« +0x4d2 »
Signe, base, zéros, largeur, hexa	« {:+#012x} »	« +0000004d2 »
	« {:+#06x} »	« +0x4d2 »

Les deux derniers exemples montrent que la largeur minimum s'applique à la totalité de la chaîne, avec le signe, la base et les autres éléments.

*Le signe est toujours mentionné pour une valeur négative et les résultats correspondent à ce qui est indiqué pour « **signe systématique** ».*

Lorsque vous demandez des zéros en préfixe, l'alignement et le remplissage sont ignorés car les zéros sont ajouté pour remplir tout le champ.

Fonction de formatage	Chaîne modèle	Résultat avec le flottant « 1234.5678 »
Par défaut	« {} »	« 1234.5678 »
Largeur de champ minimum	"{:12}"	« 1234.5678 »
Précision	"{:.2}"	« 1234.57 »
	« {:.6} »	« 1234.567800 »
Minimum, précision	« {:12.2} »	« 1234.57 »
	« {:12.6} »	« 1234.567800 »
Zéros, minimum, précision	« {:012.6} »	« 01234.567800 »
Scientifique	« {:e} »	« 1.2345678e3 »
Scientifique, précision	« {:.3e} »	« 1.235e3 »
	« {:12.3E} »	« 1.234E3 »

Formatage pour le débogage

Pendant vos séances de mise au point et pour vos journalisations, vous tirerez avantage du paramètre **{:?}** qui sert à contrôler le format de tous les types publics de la librairie standard **Rust** en sorte d'être le plus lisible possible. Cela vous permet d'inspecter les valeurs des **vecteurs**, des **tranches**, des **tuples**, des **dictionnaires**, des **exétrons** et de centaines d'autres types.

Formatage pour le débogage

```
use std::collections::HashMap;

fn main() {
  let mut villes = HashMap::new();
  villes.insert("Paris", (48.856614, 2.3522219));
  villes.insert("Aurillac", (44.930953, 2.444997));
  println!("{:?}", villes);
}
```

Résultat

```
{"Paris": (48.856614, 2.3522219), "Aurillac": (44.930953, 2.444997)}
```

*Les deux types combinés, **HashMap** et **tuple** de **f64**, savent automatiquement comment se présenter, sans réclamer de formatage spécifique. Si vous ajouter le signe « # » au formateur, **Rust** répartit la valeur sur plusieurs lignes si nécessaire.*

Formatage pour le débogage

```
use std::collections::HashMap;

fn main() {
  let mut villes = HashMap::new();
  villes.insert("Paris", (48.856614, 2.3522219));
  villes.insert("Aurillac", (44.930953, 2.444997));
  println!("{:#?}", villes);
}
```

Résultat

```
{
  "Paris": (
    48.856614,
    2.3522219,
  ),
```

```
  "Aurillac": (
     44.930953,
     2.444997,
  ),
}
```

*L'ajout de la directive **#[derive(Debug)]** permet de présenter automatiquement vos propres types prédéfinis, avec le même formateur de **{:?}**.*

Formatage pour le débogage

```
#[derive(Debug)]
struct Complexe {
  reel: f64,
  imaginaire: f64
}

fn main() {
  let i = Complexe {reel: 0.0, imaginaire: 1.0 };
  println!("{:#?}", i);
}
```

Résultat

```
Complexe {
   reel: 0.0,
   imaginaire: 1.0,
}
```

*Le résultat convient bien pour le débogage, mais nous aimerions aussi pouvoir utiliser tout simplement le formateur par défaut **{}** pour obtenir un affichage classique des nombres complexes du style « **2 + 3i** ». Nous verrons comment faire dans la section ultérieure concernant le formatage de vos propres types.*

Référence aux paramètres par index ou par nom

Les formateurs contiennent naturellement une indication des paramètres qu'il doivent incorporer à la chaîne résultante. Vous pouvez ajouter vos paramètres de format après le signe deux-points.

Formatage par indice

```
fn main() {
  println!("{1}, {0}, {2}", "primo", "deuxio", "tertio");
  println!("{2:#06x}, {1:b}, {0:=>10}", "primo", 10, 100);
}
```

Résultat

```
deuxio, primo, tertio
0x0064, 1010, =====primo
```

*Vous pouvez spécifier les paramètres à traiter par leur **nom** au lieu d'utiliser un **indice**, ce qui augmente la lisibilité des modèles de format complexe.*

Formatage par désignation des noms de paramètre

```
fn main() {
  println!("{description:.<25}{quantite:2} @ {prix:5.2}",
    prix=3.25,
    quantite=3,
    description="Café noisette"
  );
}
```

Résultat

```
Café noisette..... 3 @  3.25
```

*Les trois modes de désignation de paramètres sont possibles dans la même macro de formatage : par **indice**, par **nom** ou par **position** (c'est-à-dire sans rien spécifier). Rappelons que les paramètres positionnels sont appariés avec les éléments de format de la gauche vers la droite, comme si les paramètres par indice ou par nom n'existaient pas. Sachez enfin que les paramètres nommés doivent toujours être derniers dans la liste.*

Formatage par désignations multiples

```
fn main() {
  println!("{mode} {2} {} {}", "le", "vide", "dans", mode="saut");
}
```

Résultat

```
saut dans le vide
```

Largeur et précision dynamique

Certains éléments de formatage peuvent être déterminés pendant l'exécution : la largeur de champ, la longueur maximale de texte et la précision numérique.

Formatage variable

```
fn main() {
  for largeur in 7..10  {
    println!("{:>1$}", "Bonjour", largeur);
  }
}
```

Résultat

```
Bonjour
 Bonjour
  Bonjour
```

*La mention spéciale « **1$** » pour la largeur minimale du champ demande à la macro de se servir du dernier paramètre comme largeur, celui-ci devant être de type **usize** ou assimilé. Vous pouvez même désigner ce paramètre par son nom.*

Formatage variable

```
fn main() {
  for nombre in 7..10  {
    println!("{:>largeur$}", "Bonjour", largeur=nombre);
  }
}
```

Résultat

```
Bonjour
 Bonjour
  Bonjour
```

*Vous pouvez même indiquer le signe « ***** » à la place de la limite de longueur ou de la précision en flottant, ce qui signifie que le prochain paramètre doit spécifier la précision (avant la valeur à visualiser). Le paramètre de précision doit être du type **usize** ou assimilé.*

Cette technique n'est pas utilisable pour la largeur du champ.

Formatage variable

```
fn main() {
  for nombre in 0..5  {
    println!("{:.>12.*}", nombre, 12.568);
  }
}
```

Résultat

```
..........13
...........12.6
.........12.57
.......12.568
.....12.5680
```

Formatage variable

```
fn main() {
  for nombre in 1..5  {
    println!("{:.>position$.*}", nombre, 12.568, position=7+nombre);
  }
}
```

Résultat

```
....12.6
....12.57
....12.568
....12.5680
```

Formatage de vos types personnalisés

Les macros de format sont fondées sur plusieurs traits définis dans le module **std::fmt**, servant à convertir les valeurs vers du texte. Vous pouvez vous servir de ces macros pour vos types personnels en implémentant vous-même les traits nécessaires.

Notation	Exemple	Trait	Résultat
rien	{}	std::fmt::Display	Trait polyvalent : texte, nombre ou erreur

b	{bits:#b}	std::fmt::Binary	Nombre en binaire
o	{:#5o}	std::fmt::Octal	Nombre en octal
x	{:4x}	std::fmt::LowerHex	Nombre hexadécimal, minuscules
X	{:016X}	std::fmt::UpperHex	Nombre hexadécimal, majuscules
e	« {:.3e} »	std::fmt::LowerExp	Nombre flottant en notation scientifique
E	« {:.3E} »	std::fmt::UpperExp	Idem avec E majuscule
?	« {:# ?} »	std::fmt::Debug	Format de débogage pour programmeur
p	« {:.p} »	std::fmt::Pointer	Valeur adresse de pointeur, pour programmeur

*Lorsque vous ajoutez l'attribut **#[derive(Debug)]** à une définition de type afin de pouvoir utiliser le formateur **{:?}**, cela revient à demander à **Rust** d'implémenter à votre place le trait **std::fmt::Debug**.*

Tous les traits de formatage obéissent à la même structure, et seul de nom du Trait change. Voici par exemple le trait std::fmt::Display.

Trait Display

```
trait Display {
  fn fmt(&self, dest: &mut std::fmt::Formatter) -> std::fmt::Result;
}
```

*La méthode **fmt()** produit une représentation bien formé de **self** puis envoie les caractères vers **dest** qui ne contente pas d'incarner le flux de sortie. Le paramètre **dest** contient également les détails qui ont été trouvés pendant le formatage, et notamment l'alignement et la largeur de champ minimale.*

*Je vous propose de d'implémenter la structure **Complexe** pour le trait **Display** afin que nous puissions proposer le format classique « **a + bi** ».*

Formatage personnalisé

```
struct Complexe {
  reel: f64,
  imaginaire: f64
}

impl std::fmt::Display for Complexe {
  fn fmt(&self, dest: &mut std::fmt::Formatter) -> std::fmt::Result {
    let signe = if self.imaginaire < 0.0 { '-' } else { '+' };
    write!(dest, "{}{}{}i", self.reel, signe, f64::abs(self.imaginaire))
  }
}

fn main() {
  let c = Complexe { reel: 2.3, imaginaire: -5.6 };
  println!("c = {}", c);
}
```

Résultat

```
c = 2.3-5.6i
```

Vous pouvez avoir parfois besoin d'afficher les valeurs complexes en représentation polaire au lieu de la représentation algébrique. La longueur est alors tracée dans le plan complexe en partant de l'origine avec l'angle dans le sens horaire par rapport à l'axe des x positifs.

*L'ajout du signe **« # »** dans le formateur permet de choisir parmi plusieurs formats. Vous pouvez vous en servir dans l'implémentation de **Display** pour basculer ainsi en représentation polaire.*

Formatage personnalisé

```
struct Complexe {
  reel: f64,
  imaginaire: f64
}

impl std::fmt::Display for Complexe {
  fn fmt(&self, dest: &mut std::fmt::Formatter) -> std::fmt::Result {
    let (r, i) = (self.reel, self.imaginaire);
    if dest.alternate() {
      let module = f64::sqrt(r*r + i*i);
      let argument = f64::atan2(i, r) / std::f64::consts::PI * 180.0;
      write!(dest, "{:.1} \u{2220} {:.0}°", module, argument)
    }
    else {
      let signe = if self.imaginaire < 0.0 { '-' } else { '+' };
```

```
    write!(dest, "{:.3} {} {:.3}i", self.reel, signe, f64::abs(self.imaginaire))
   }
  }
}

fn main() {
  let c = Complexe { reel: 1./2., imaginaire: f64::sqrt(3.)/2. };
  println!("c = {}", c);
  println!("c = {:#}", c);
}
```

Résultat

```
c = 0.500 + 0.866i
c = 1.0 ∠ 60°
```

*La méthode **fmf()** des traits de formatage renvoie une valeur **fmt::Result** qui est un type **Result** spécifique aux modules, mais vous ne devez propager une erreur que depuis les opérations concernant **Formatter**, comme le fait l'implémentation **fmt::Display** lorsqu'elle appelle **write!()**.*

*Autrement dit, vos fonctions de formatage ne doivent jamais être elles-mêmes à l'origine d'une erreur. Cela permet ainsi aux macros comme **format!()** de renvoyer directement une **String** au lieu d'un **Result<String, ...>**, parce que l'ajout du texte formaté à une chaîne **String** n'échoue jamais.*

*De plus, cela vous garantit que les erreurs que vous recevez en provenance de **write!()** ou de **writln!()** correspondent à de vrais problèmes au niveau du flux d'entrée/sortie et non à un problème de formatage.*

***Formatter** dispose de toute une gamme de méthodes support. Cette structure représente les éléments de formatage que vous souhaitez soumettre a moment de l'affichage, largeur, précision, justification, etc. Dans l'exemple précédent, grâce à la méthode **alternate()**, nous avons pu proposer une alternative d'affichage au moyen du formateur « # ». Ci-dessous, je vous propose de rajouter la notion de précision avec la méthode **precision()**.*

Formatage personnalisé

```
struct Complexe {
  reel: f64,
  imaginaire: f64
}

impl std::fmt::Display for Complexe {
  fn fmt(&self, dest: &mut std::fmt::Formatter) -> std::fmt::Result {
    let (r, i) = (self.reel, self.imaginaire);
    let p = match dest.precision() {
      Some(x) => x,
      None => 0
    };
    if dest.alternate() {
      let module = f64::sqrt(r*r + i*i);
      let argument = f64::atan2(i, r) / std::f64::consts::PI * 180.0;
      match p {
        0 => write!(dest, "{:.1} \u{2220} {:.0}°", module, argument),
        p => write!(dest, "{:.precision$} \u{2220} {:.0}°", module, argument, precision=p)
      }
    }
    else {
      let signe = if self.imaginaire < 0.0 { '-' } else { '+' };
      match p {
        0 => write!(dest, "{} {} {}i", self.reel, signe, f64::abs(self.imaginaire)),
        p => write!(dest, "{:.pre$} {} {:.pre$}i", self.reel, signe, f64::abs(self.imaginaire), pre=p)
      }
    }
  }
}

fn main() {
  let c = Complexe { reel: 1./2., imaginaire: f64::sqrt(3.)/2. };
  println!("c = {}", c);
  println!("c = {:.2}", c);
  println!("c = {:#}", c);
  println!("c = {:#.2}", c);
}
```

Résultat

```
c = 0.5 + 0.8660254037844386i
c = 0.50 + 0.87i
c = 1.0 ∠ 60°
c = 1.00 ∠ 60°
```