Une des grands avancées en programmation est de permettre l'utilisation du même bloc de code source pour traiter des valeurs de types différents, même des types qui n'existent pas encore au moment de la compilation de ce code. En voici deux exemples types :

- ***Vec<T> est générique :*** *vous pouvez créer un vecteur de n'importe quel type, y compris un type défini dans le programme utilisant* ***Vec*** *d'une façon que les créateurs du code n'avaient même pas imaginé.*
- *De nombreuses entités disposent de méthodes d'écriture « **.write()** », notamment* ***File*** *et* ***TcpStream****. Dans votre code, vous pouvez récupérer un scripteur (****writer****) par référence, et lui envoyer des données. Votre code n'a pas besoin de se soucier du type du scripteur. Si quelqu'un ajoute un nouveau type plus tard, votre code acceptera cette extension du mécanisme d'écriture.*

Bien sûr cette possibilité n'est pas une innovation de ***Rust****. Elle correspond au* ***polymorphisme*** *et avait constitué une grande nouveauté en technologie informatique dans les années 70.* ***Rust*** *supporte le* ***polymorphisme*** *au moyen de deux caractéristiques apparentés : les* ***traits*** *et les* ***génériques****.*

Les types de données génériques

Nous pouvons utiliser la généricité pour créer des définitions pour des éléments comme les signatures de fonctions ou de structures, que nous pouvons ensuite utiliser sur de nombreux types de données concrets. Commençons par regarder comment définir des fonctions, des structures, des énumérations, et des méthodes en utilisant la généricité.

L'exemple typique pour bien comprendre la généricité est le fait de pouvoir construire une fonction qui sert de modèle à d'autres fonctions qui comporte le même code interne pour résoudre l'algorithme souhaité. Dans l'exemple qui suit, nous définissons deux fonctions dont l'objectif est de trouver la valeur la plus grande dans un tableau fourni en argument, d'une part avec des valeur entière, d'autre part, avec des caractères.

Fonctions similaires

```
fn le_plus_grand_i32(liste: &[i32]) -> &i32 {
    let mut le_plus_grand = &liste[0];
    for element in &liste[1..] {
        if element > le_plus_grand {
            le_plus_grand = element;
        }
    }
    le_plus_grand
}

fn le_plus_grand_caractere(liste: &[char]) -> &char {
    let mut le_plus_grand = &liste[0];
    for element in &liste[1..] {
        if element > le_plus_grand {
            le_plus_grand = element;
        }
    }
    le_plus_grand
}

fn main() {
    let resultat = le_plus_grand_i32(&[34, 50, 25, 100, 65]);
    println!("Le plus grand nombre est {}", resultat);

    let resultat = le_plus_grand_caractere(&['y', 'm', 'a', 'q']);
    println!("Le plus grand caractère est {}", resultat);
}
```

Résultat

```
Le plus grand nombre est 100
Le plus grand caractère est y
```

Généricité dans la définition d'une fonction

Lorsque nous définissons une fonction en utilisant la généricité, nous utilisons des génériques dans la signature de la fonction où nous précisons habituellement les types de données des paramètres et la valeur de retour. Faire ainsi rend notre code plus flexible et apporte plus de fonctionnalités au code appelant notre fonction, tout en évitant la duplication de code.

Les corps des fonctions précédentes sont identiques puisque l'algorithme est le même, seule la signature change. Essayons d'éviter cette duplication en utilisant cette fois-ci un paramètre de type générique dans une seule et unique fonction.

Pour paramétrer les types dans la nouvelle fonction que nous allons définir, nous avons besoin de donner un nom au type de paramètre, comme nous l'avons fait pour les valeurs des paramètres des fonctions.

Vous pouvez utiliser tout ce que vous souhaitez pour nommer le type de paramètres. Mais ici nous utilisons ***T*** *par convention car les noms de paramètres en* ***Rust*** *sont généralement assez courts. Ici la version courte de "type" c'est* ***T*** *et c'est le choix par défaut de nombreux développeurs* ***Rust****.*

Lorsque nous utilisons un paramètre dans le corps de la fonction, nous devons déclarer le nom du paramètre dans la signature afin que le compilateur puisse savoir à quoi ce réfère ce nom. De la même manière, lorsque nous utilisons un nom

*de type de paramètre dans la signature d'une fonction, nous devons déclarer le nom du type de paramètre avant de pouvoir l'utiliser. Pour déclarer la fonction générique **le_plus_grand()**, il faut placer la déclaration du nom du type entre des chevrons **<>**, le tout entre le nom de la fonction et la liste des paramètres, comme ceci :*

Signature d'une fonction générique

```
fn le_plus_grand<T> (liste: &[T]) -> &T {
```

*Cette définition se lit comme ceci : la fonction **le_plus_grand()** est générique en fonction du type **T**. Cette fonction possède un paramètre qui s'appelle **liste**, qui est un **slice** de valeurs de type **T**. Cette fonction **le_plus_grand()** retourne une référence vers la valeur du même type **T**.*

*L'exemple ci-dessous nous montre la définition de la fonction **le_plus_grand()** avec le type de données générique présent dans sa signature. La fonction principale nous montre aussi que nous pouvons appeler la fonction avec un **slice** soit de valeurs **i32**, soit de valeurs **char**. **L'inférence** de type est mise en action comme le reste du code source dans **Rust**.*

Fonction générique le_plus_grand()

```
fn le_plus_grand<T>(liste: &[T]) -> &T {
    let mut le_plus_grand = &liste[0];
    for element in &liste[1..] {
        if element > le_plus_grand {
            le_plus_grand = element;
        }
    }
    le_plus_grand
}

fn main() {
    let resultat = le_plus_grand(&[34, 50, 25, 100, 65]);
    println!("Le plus grand nombre est {}", resultat);

    let resultat = le_plus_grand(&['y', 'm', 'a', 'q']);
    println!("Le plus grand caractère est {}", resultat);
}
```

Résultat

```
error[E0369]: binary operation `>` cannot be applied to type `&T`
 --> src/main.rs:4:20
  |
4 |         if element > le_plus_grand {
  |            ------- ^ ------------- &T
  |            |
  |            &T
  |
help: consider restricting type parameter `T`
  |
1 | fn le_plus_grand<T: std::cmp::PartialOrd>(liste: &[T]) -> &T {
  |                   ^^^^^^^^^^^^^^^^^^^^^^
```

*Nous obtenons une erreur à la compilation. Ceci se produit parce que le système n'est pas sûr que le type que nous proposerons pour résoudre notre fonction soit capable de déterminer si la valeur correspondante est plus grande qu'une autre. Par exemple, pour la structure **Eleve** que nous avons créé précédemment ne comprend pas du tout cette notion là.*

*La note évoque **std::cmp::PartialOrd,** qui est un **trait** que nous aborderons ultérieurement. Pour le moment, cette erreur nous informe que le corps de **le_plus_grand()** ne fonctionnera pas pour tous les types possibles que **T** peut représenter.*

*Comme nous voulons comparer les valeurs de type **T** dans le corps, nous pouvons utiliser uniquement des types dont les valeurs peuvent être triées dans l'ordre. Pour effectuer des comparaisons, la librairie standard propose le **trait PartialOrd** que vous pouvez implémenter sur n'importe quel type. Il existe aussi le trait **Ord** qui est plus complet que **PartialOrd**.*

*Dans l'exemple suivant, je propose donc la mention **<T : Ord>** qui signifie que la fonction générique **le_plus_grand()** est utilisable avec un paramètre d'entrée de n'importe quel type **T** à partir du moment où il implémente le trait **Ord**, c'est-à-dire un type **ordonné**. Le compilateur génère automatiquement du code exécutable différent selon le type **T** réellement utilisé.*

Fonction générique le_plus_grand()

```
fn le_plus_grand<T: Ord>(liste: &[T]) -> &T {
    let mut le_plus_grand = &liste[0];
    for element in &liste[1..] {
        if element > le_plus_grand {
            le_plus_grand = element;
        }
    }
    le_plus_grand
}

fn main() {
    let resultat = le_plus_grand(&[34, 50, 25, 100, 65]);
```

```
    println!("Le plus grand nombre est {}", resultat);

    let resultat = le_plus_grand(&['y', 'm', 'a', 'q']);
    println!("Le plus grand caractère est {}", resultat);
}
```

Résultat

```
Le plus grand nombre est 100
Le plus grand caractère est y
```

*Dans l'exemple précédent, **Rust** choisi automatiquement le type de paramètre lors de l'appel de la fonction, ce qui pour l'utilisateur de la fonction est très confortable. Il est toutefois possible de confirmer le type de paramètre, comme ci-dessous :*

Fonction générique le_plus_grand()

```
...
fn main() {
    let resultat = le_plus_grand::<i32>(&[34, 50, 25, 100, 65]);
    println!("Le plus grand nombre est {}", resultat);

    let resultat = le_plus_grand::<char>(&['y', 'm', 'a', 'q']);
    println!("Le plus grand caractère est {}", resultat);
}
```

*En pratique, c'est rarement utile parce que **Rust** déduit normalement le paramètre de type en analysant les arguments. Si par contre la fonction générique ne possède pas de paramètres permettant cette déduction, il vous reste à l'appeler de façon explicite comme dans l'exemple suivant :*

Appel d'une fonction générique explicite

```
fn main() {
    let collection = (1..10).collect(); // Ne fonctionne pas, type inconnu
    let collection = (1..10).collect::<Vec<u32>>(); // ok
    println!("{:?}", collection);
}
```

Résultat

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

*Vous aurez parfois besoin qu'un paramètre de type propose plusieurs capacités. Si dans notre fonction **le_plus_grand()** nous désirons afficher à la fois la suite des valeurs ainsi que le résultat du traitement, nous devons rajouter les contraintes **Debug** et **Display** en sachant que les types que nous proposerons seront capable d'implémenter ces deux traits spécifiques.*

Plusieurs contraintes dans la généricité

```
use std::fmt::{Debug, Display};

fn le_plus_grand<T: Ord + Debug + Display>(liste: &[T]) -> &T {
    let mut le_plus_grand = &liste[0];
    for element in &liste[1..] {
        if element > le_plus_grand {
            le_plus_grand = element;
        }
    }
    println!("'{}' est le plus grand de la liste {:?}", le_plus_grand, liste);
    le_plus_grand
}

fn main() {
    let resultat = le_plus_grand(&[34, 50, 25, 100, 65]);
    println!("Le plus grand nombre est {}", resultat);

    let resultat = le_plus_grand(&['y', 'm', 'a', 'q']);
    println!("Le plus grand caractère est {}", resultat);
}
```

Résultat

```
'100' est le plus grand de la liste [34, 50, 25, 100, 65]
Le plus grand nombre est 100
'y' est le plus grand de la liste ['y', 'm', 'a', 'q']
Le plus grand caractère est y
```

*Une fonction générique peut disposer de plusieurs paramètres de types. Il est possible bien entendu que chacun des paramètres proposent plusieurs contraintes particulières. La lisibilité de ces contraintes peut alors être très perturbée. Heureusement, **Rust** autorise une syntaxe plus aérée basée sur le mot clé **where**.*

Plusieurs contraintes dans la généricité

```
use std::fmt::{Debug, Display};

fn le_plus_grand<T>(liste: &[T]) -> &T   where T: Ord+Debug+Display  {
    let mut le_plus_grand = &liste[0];
    for element in &liste[1..] {
        if element > le_plus_grand {
            le_plus_grand = element;
        }
    }
    println!("'{}' est le plus grand de la liste {:?}", le_plus_grand, liste);
    le_plus_grand
}

fn main() {
    let resultat = le_plus_grand(&[34, 50, 25, 100, 65]);
    println!("Le plus grand nombre est {}", resultat);

    let resultat = le_plus_grand(&['y', 'm', 'a', 'q']);
    println!("Le plus grand caractère est {}", resultat);
}
```

Résultat

```
'100' est le plus grand de la liste [34, 50, 25, 100, 65]
Le plus grand nombre est 100
'y' est le plus grand de la liste ['y', 'm', 'a', 'q']
Le plus grand caractère est y
```

Généricité dans la définition des structures

Nous pouvons aussi définir des structures en utilisant des paramètres de type générique dans un ou plusieurs champs en utilisant la syntaxe **<>.** L'exemple suivant nous montre comment définir une structure **Rectangle<T>** qui peut s'exprimer avec des dimensions entières ou réelles.

Structure générique

```
#[derive(Debug)]
struct Rectangle<T> {
    largeur: T,
    hauteur: T
}

fn main() {
    let entiers = Rectangle { largeur: 10, hauteur: 5 };
    let reels = Rectangle { largeur: 1.5, hauteur: 4. };
    println!("{:?}", entiers);
    println!("{:?}", reels);
}
```

Résultat

```
Rectangle { largeur: 10, hauteur: 5 }
Rectangle { largeur: 1.5, hauteur: 4.0 }
```

La syntaxe pour l'utilisation des génériques dans les définitions de structures est similaire à celle utilisée dans la définition des fonctions. D'abord, nous déclarons le nom du type de paramètre entre des chevrons juste après le nom de la structure. Ensuite, nous pouvons utiliser le type générique dans la définition de la structure où nous utilisions précédemment des types de données précis.

*Notez que comme nous n'avons utilisé qu'**un seul** type générique pour définir **Rectangle<T>**, cette définition dit que la structure **Rectangle<T>** est générique en fonction du type **T**, et les champs **largeur** et **hauteur** sont tous les deux du même type, quel que soit ce type. Si nous créons une instance de **Rectangle<T>** qui possède des valeurs de différents types, comme dans l'exemple qui suit, notre code ne pourra pas se compiler.*

Structure générique

```
struct Rectangle<T> {
    largeur: T,
    hauteur: T
}

fn main() {
    let ne_fonctionne_pas = Rectangle { largeur: 5, hauteur: 4. };
}
```

Résultat

```
error[E0308]: mismatched types
```

```
   --> src/main.rs:34:62
    |
34  | let ne_fonctionne_pas = Rectangle { largeur: 5, hauteur: 4. };
    |                                                          ^^ expected integer, found floating-point number
```

*Dans cet exemple, lorsque nous assignons l'entier **5** à **largeur**, nous laissons entendre au compilateur que le type générique **T** sera un entier pour cette instance de **Rectangle<T>**. Ensuite, lorsque nous assignons **4.** à **hauteur**, que nous avons défini comme ayant le même type que **largeur**, nous obtenons une erreur pour mauvais type comme ci-dessus :*

*Pour définir une structure **Rectangle** où **largeur** et **hauteur** sont tous les deux génériques mais peuvent avoir des types différents, en utilisant les paramètres multiples de types génériques. Dans l'exemple qui suit, nous pouvons changer la définition de **Rectangle** pour être générique en fonction des types **T** et **U** où **largeur** est de type **T** et **hauteur** est de type **U**.*

Structure générique

```rust
#[derive(Debug)]
struct Rectangle<T, U> {
    largeur: T,
    hauteur: U
}

fn main() {
    let fonctionne = Rectangle { largeur: 5, hauteur: 4. };
    println!("{:?}", fonctionne);
}
```

Résultat

```
Rectangle { largeur: 5, hauteur: 4.0 }
```

Vous pouvez utiliser autant de paramètres de type génériques que vous souhaitez dans la déclaration de la définition, mais en utiliser plus de quelques-uns rend votre code difficile à lire. Lorsque vous avez besoin de nombreux types génériques, cela peut être un signe que votre code a besoin d'être remanié dans des éléments plus petits.

Généricité dans la définition des énumérations

Comme nous l'avons fait avec les structures, nous pouvons définir des énumérations qui utilisent des types de données génériques dans leurs variantes. Commençons par regarder à nouveau l'énumération **Option<T>** que fournit la bibliothèque standard, et que nous avons utilisé lors de l'étude sur les énumérations

Énumération générique

```rust
enum Option<T> {
    Some(T),
    None,
}
```

*Cette définition doit avoir maintenant plus de sens pour vous. Comme vous pouvez le constater, **Option<T>** est une énumération qui est générique en fonction du type **T** et possède deux variantes : **Some**, qui contient une valeur de type **T**, et une variante **None** qui ne contient aucune valeur.*

*En utilisant l'énumération **Option<T>**, nous pouvons exprimer le concept abstrait d'avoir une valeur optionnelle, et comme **Option<T>** est générique, nous pouvons utiliser cette abstraction peu importe le type de la valeur optionnelle.*

*Les énumérations peuvent aussi utiliser plusieurs types génériques. La définition de l'énumération **Result** que nous avons utilisé lors de l'étude précédente en est un exemple :*

Énumération générique

```rust
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

*L'énumération **Result** est générique en fonction de deux types, **T** et **E**, et possède deux variantes : **Ok**, qui contient une valeur de type **T**, et **Err**, qui contient une valeur de type **E**. Cette définition rend possible l'utilisation de l'énumération **Result** n'importe où lorsque nous avons besoin d'une opération qui peut réussir (et retourner une valeur du type **T**) ou échouer (et retourner alors une erreur du type **E**).*

Généricité dans la définition des méthodes

Nous pouvons implémenter des méthodes sur des structures et des énumérations et aussi utiliser des types génériques dans leurs définitions. Je vous propose d'implémenter des méthodes sur notre structure générique **Rectangle**, pour connaître la surface et le périmètre de ce rectangle.

*Nous devons déclarer **T** juste après le mot-clé **impl** afin de pouvoir l'utiliser pour préciser que nous implémentons des méthodes sur le type **Rectangle<T>**. En déclarant **T** comme un type générique après **impl**, **Rust** peut comprendre que le type entre les chevrons dans **Rectangle** est un type générique plutôt qu'un type concret.*

*Nous pouvons également implémenter des méthodes en choisissant uniquement des instances de **Rectangle<u32>** pour réaliser des traitements spécifiques en rapport à ce type plutôt que sur des instances de n'importe quel type **Rectangle<T>**. Dans ce cas là, nous n'avons plus besoin de déclarer un type après le mot-clé **impl**.*

*Dans l'exemple qui suit, je choisi d'avoir **largeur** et **hauteur** avec le même type, soit des entiers, soit des réels. Effectivement, lorsque vous devez faire des calculs sur ces types primitifs, **Rust** ne permet pas d'avoir des types différents.*

Implémentation de méthodes associées à la structure générique Rectangle

```
#[derive(Debug)]
struct Rectangle<T> {
   largeur: T,
   hauteur: T
}

use core::ops::Mul;
impl<T> Rectangle<T> where T: Mul<Output=T> + Copy
{
   fn carre(cote: T) -> Rectangle<T> {
      Rectangle {largeur: cote, hauteur: cote}
   }
   fn surface(&self) -> T {
      self.largeur*self.hauteur
   }
}

impl Rectangle<u32>
{
   fn perimetre(&self) -> u32 {  (self.largeur + self.hauteur)*2  }
}

impl Rectangle<f64>
{
   fn perimetre(&self) -> f64 {   (self.largeur + self.hauteur)*2.0   }
}

fn main() {
   let entiers = Rectangle { largeur: 10, hauteur: 5 };
   let reels = Rectangle { largeur: 1.5, hauteur: 4. };
   let carre = Rectangle::carre(5);
   println!("rectangle={:?}, périmètre={}, surface={}", entiers, entiers.perimetre(), entiers.surface());
   println!("rectangle={:?}, périmètre={}, surface={}", reels, reels.perimetre(), reels.surface());
   println!("rectangle={:?}, périmètre={}, surface={}", carre, carre.perimetre(), carre.surface());
}
```

Résultat

```
rectangle=Rectangle { largeur: 10, hauteur: 5 }, périmètre=30, surface=50
rectangle=Rectangle { largeur: 1.5, hauteur: 4.0 }, périmètre=11, surface=6
rectangle=Rectangle { largeur: 5, hauteur: 5 }, périmètre=20, surface=25
```

*En consultant le code précédent, nous remarquons que pour implémenter la méthode **carre()**, nous avons besoin que le type **T** prenne en compte le trait **Copy** afin que nous soyons capable de copier le **côté** du carré pour les attributs **largeur** et **hauteur**.*

*De même, pour la méthode **surface()**, le type **T** doit prendre en compte le trait **Mul** afin de spécifier que le type est capable de faire des multiplications. Dans le même registre, il existe aussi les traits : **Add** pour les additions, **Div** pour les divisions, **Sub** pour les soustractions, etc.*

*Lorsque vous utilisez des constantes littérales spécifiques tel que la valeur entière **2** ou la valeur réelle **2.0**, nous devons passer par des implémentations de méthode spécifiques puisque le langage **Rust** ne permet pas de réaliser des calculs avec des types différents. C'est pour cela que nous proposons les implémentations **Rectangle<u32>** et **Rectangle<f64>**.*

Utilisation des traits

Comme nous venons de le découvrir dans les chapitres précédents, un **trait** possède une capacité d'action qu'un type peut ou ne pas posséder. En général, un **trait** représente une capacité, quelque chose qu'un type est capable de faire.

*Un **trait** décrit une fonctionnalité que possède un type particulier et qu'il peut partager avec d'autres types. Nous pouvons utiliser les **traits** pour définir un comportement partagé de manière abstraite. Nous pouvons lier ces **traits** à un générique pour exprimer le fait qu'il puisse être de n'importe quel type à condition qu'il ait un comportement donné.*

- *Une valeur qui implémente **std::io::Write** est capable d'écrire des octets vers une sortie.*
- *Une valeur qui implémente **std::iter::Iterator** est capable de produire une séquence de valeurs.*
- *Une valeur qui implémente **std::clone::Clone** est capable de se dupliquer en mémoire.*
- *Une valeur qui implémente **std::fmt::Debug** peut être affichée avec **println !()** avec le spécificateur de format **{:?}**.*

Tous ces traits font partie de la librairie de Rust, et un grand nombre de types standard les connaissent.

- *__std::fs::File__ implémente le trait __Write__ pour écrire des octets vers un fichier local. __std::net::TcpStream__ écrit vers une connexion réseau. Enfin, __Vec<u8>__ implémente aussi __Write__ et chaque appel à __.write()__ pour un vecteur d'octets ajoute des données à la fin du vecteur.*
- *__Range<i32>__ (le type de la plage __0,,10__) implémente le trait __Iterator__, de même que certains itérateurs qui sont associés avec les tranches, les __maps__, etc.*
- *La plupart des types standards implémentent le trait __Clone__, à l'exception en particulier de __TcpStream__ parce qu'il ne se limite pas à des données en mémoire.*
- *Enfin, la plupart des types standard supportent le trait __Debug__.*

Définition d'un trait

La définition d'un **trait** est très simple, puisqu'il suffit de lui donner un nom et de dresser la liste des signatures de type pour les **méthodes** du **trait**. Un **trait** décrit une fonctionnalité que propose un type particulier et qu'il peut partager avec d'autres types. Nous pouvons utiliser les traits pour définir un comportement partagé de manière abstraite. Nous pouvons lier ces traits à un générique pour exprimer le fait qu'il puisse être de n'importe quel type à condition qu'il ait un comportement donné.

Le comportement d'un type s'exprime via les méthodes que nous pouvons appeler sur ce type. Différents types peuvent partager le même comportement si nous pouvons appeler les mêmes méthodes sur tous ces types. Définir un __trait__ est une manière de grouper ensemble les signatures des __méthodes__ pour définir un comportement nécessaire pour accomplir un objectif. À titre d'exemple, je vous propose de fabriquer le __trait Description__ qui permettra à l'usage de nommer un type particulier et de décrire sa structure.

Trait Description

```
trait Description {
   fn decrire(&self) -> String;
   fn identite(&self) -> String;
}
```

Nous déclarons un __trait__ en utilisant le mot-clé __trait__ et ensuite le nom du __trait__, qui est __Description__ dans notre cas. Entre les accolades, nous déclarons ensuite les signatures des méthodes qui soulignent le comportement des types qui implémentent ce __trait__, ici __decrire()__ et __identite()__.

À la fin de la signature des méthodes, au lieu de renseigner une implémentation entre des accolades, nous utilisons un point-virgule. Il s'agit juste d'une déclaration. Chaque type qui implémente ce __trait__ doit renseigner son propre comportement dans le corps de la méthode. Le compilateur assure que tous les types qui veulent utiliser le __trait Description__ posséderont bien les méthodes __decrire()__ et __indentite()__ définis avec ces signatures précises. Ces types doivent alors respecter le « __contrat__ » défini dans la définition du __trait__.

Implémentation d'un trait

Maintenant que nous avons défini le comportement souhaité du trait Description, nous pouvons maintenant l'implémenter sur tous les types qui désirent avoir une description. Le gros avantage de cette technique et ce qui est remarquable, c'est que ce trait peut être utilisé quelque soit le type, et éventuellement sur des types qui n'ont absolument rien en commun comme vous pouvez le constater ci-dessous.

L'implémentation d'un __trait__ sur un type est similaire à l'implémentation d'une méthode classique. La différence est que nous ajoutons le nom du __trait__ que nous voulons implémenter après le __impl__, et que nous utilisons ensuite le mot-clé __for__ ainsi que le nom du type sur lequel nous souhaitons implémenter le __trait__.

A l'intérieur du bloc __impl__, nous ajoutons les signatures des méthodes présentes dans la définition du __trait__. Au lieu d'ajouter un point-virgule après chaque signature, nous plaçons les accolades et nous remplissons le corps de la méthode avec le comportement spécifique que nous souhaitons résoudre pour le type en cours. Après avoir implémenté le __trait__, nous pouvons appeler les méthodes de l'instance comme s'il s'agissait de méthodes classiques.

Implémentation du trait Description

```
trait Description {
   fn decrire(&self) -> String;
   fn identite(&self) -> String;
}

enum Forme {
   Cercle { rayon: f64 },
   Carre { cote: f64 },
   Rectangle { largeur: f64, hauteur: f64 }
}

use Forme::{Cercle, Carre, Rectangle};
use core::f64::consts::PI;

impl Forme {
   fn perimetre(&self) -> f64 {
      match self {
         Cercle {rayon}                  => 2. * PI * rayon,
         Carre {cote}                    => 4. * cote,
         Rectangle {largeur, hauteur}    => 2. * (largeur+hauteur)
```

```
        }
    }
    fn surface(&self) -> f64 {
        match self {
            Cercle {rayon}                  => PI * rayon * rayon,
            Carre {cote}                    => cote * cote,
            Rectangle {largeur, hauteur}    => largeur*hauteur
        }
    }
}

impl Description for Forme {
    fn decrire(&self) -> String {
        match self {
            Cercle{rayon}                   => format!("Cercle (rayon={})", rayon),
            Carre {cote}                    => format!("Carré (côté={})", cote),
            Rectangle {largeur, hauteur}    => format!("Rectangle (largeur={}, hauteur={})", largeur, hauteur)
        }
    }
    fn identite(&self) -> String {
        match self {
            Cercle{..}      => "Cercle".to_string(),
            Carre {..}      => "Carré".to_string(),
            Rectangle {..}  => "Rectangle".to_string()
        }
    }
}

struct Eleve {
    nom: String,
    prenom: String,
    notes: Vec<f32>
}

impl Description for Eleve {
    fn decrire(&self) -> String {
        format!("{}, notes {:?}", self.identite(), self.notes)
    }
    fn identite(&self) -> String {
        format!("{} {}", self.prenom, self.nom)
    }
}

fn main() {
    let cercle = Cercle {rayon: 5.};
    let carre = Carre {cote: 2.5};
    let rectangle = Rectangle {largeur: 2.5, hauteur:1.5};
    let elle = Eleve {
        nom: "BORÉALE".to_string(),
        prenom: "Aurore".to_string(),
        notes: [8.5, 12.5, 15., 18.5].to_vec()
    };
    println!("{}, perimètre={:.3}, surface={:.3}", cercle.decrire(), cercle.perimetre(), cercle.surface());
    println!("{}, perimètre={}, surface={}", carre.decrire(), carre.perimetre(), carre.surface());
    println!("{}, perimètre={}, surface={}", rectangle.decrire(), rectangle.perimetre(), rectangle.surface());
    println!("Identité : {}\n{}", elle.identite(), elle.decrire());
}
```

Résultat

```
Cercle (rayon=5), perimètre=31.416, surface=78.540
Carré (côté=2.5), perimètre=10, surface=6.25
Rectangle (largeur=2.5, hauteur=1.5), perimètre=8, surface=4
Identité : Aurore BORÉALE
Aurore BORÉALE, notes [8.5, 12.5, 15.0, 18.5]
```

> *Il existe une limitation que nous devons souligner avec l'implémentation des **traits**. L'implémentation d'un **trait** sur un type ne peut se faire qu'à la condition que le **trait** ou le type soit défini localement dans notre **caisse**.*

Méthodes par défaut

Il est parfois utile d'avoir un comportement par défaut pour toutes ou une partie des méthodes d'un trait plutôt que de demander l'implémentation de toutes les méthodes sur chacun des types qui implémente le trait. Ainsi, lorsque nous implémentons le trait sur un type particulier, nous pouvons garder ou réécrire le comportement par défaut de chaque méthode.

> *Pour définir vos méthodes par défaut, vous devez l'implémenter directement dans la déclaration du **trait** en utilisant la syntaxe des accolades en lieu et place du point-virgule.*

Vous spécifier alors dans le bloc de la méthode l'algorithme ou l'information que vous souhaitez soumettre. Dans l'exemple ci-dessous, nous n'avons plus besoin de proposer l'identité des formes qui n'a pas beaucoup de sens dans ce contexte là. Nous pouvons utiliser, si le besoin s'en fait sentir, la méthode par défaut que nous venons de définir.

Implémentation du trait Description avec une méthode par défaut

```
trait Description {
    fn decrire(&self) -> String;
    fn identite(&self) -> String {
        String::from("(En savoir plus ...)")
    }
}
...

use Forme::{Cercle, Carre, Rectangle};
use core::f64::consts::PI;

impl Description for Forme {
    fn decrire(&self) -> String {
        match self {
            Cercle{rayon}                   => format!("Cercle (rayon={})", rayon),
            Carre {cote}                    => format!("Carré (côté={})", cote),
            Rectangle {largeur, hauteur}    => format!("Rectangle (largeur={}, hauteur={})", largeur, hauteur)
        }
    }
}
...

impl Description for Eleve {
    fn decrire(&self) -> String {
        format!("{}, notes {:?}", self.identite(), self.notes)
    }
    fn identite(&self) -> String {
        format!("{} {}", self.prenom, self.nom)
    }
}

fn main() {
    let cercle = Cercle {rayon: 5.};
    let carre = Carre {cote: 2.5};
    let rectangle = Rectangle {largeur: 2.5, hauteur:1.5};
    let elle = Eleve {
        nom: "BORÉALE".to_string(),
        prenom: "Aurore".to_string(),
        notes: [8.5, 12.5, 15., 18.5].to_vec()
    };
    println!("{}, perimètre={:.3}, surface={:.3}", cercle.decrire(), cercle.perimetre(), cercle.surface());
    println!("{}, perimètre={}, surface={}", carre.decrire(), carre.perimetre(), carre.surface());
    println!("{}, perimètre={}, surface={}", rectangle.decrire(), rectangle.perimetre(), rectangle.surface());
    println!("Identité : {}\n{}", elle.identite(), elle.decrire());
}
```

Résultat

```
Cercle (rayon=5), perimètre=31.416, surface=78.540
Carré (côté=2.5), perimètre=10, surface=6.25
Rectangle (largeur=2.5, hauteur=1.5), perimètre=8, surface=4
Identité : Aurore BORÉALE
Aurore BORÉALE, notes [8.5, 12.5, 15.0, 18.5]
```

*Le fait de proposer des méthodes par défaut permet d'avoir moins de contrainte pour certains types. Du coup, le contrat à respecter devient plus allégé. Ceci dit, la plupart du temps, lorsque nous proposons un **trait**, c'est pour faire en sorte de résoudre les fonctionnalités prévues au départ pour chacun des types associés.*

Des traits en paramètres de fonctions

Maintenant que vous savez comment définir et implémenter les **traits**, nous pouvons regarder comment utiliser les **traits** pour définir des fonctions qui acceptent plusieurs types différents (avec toutefois le fait que chaque type implémente bien le **trait** spécifié en paramètre de la fonction).

Trait en paramètre de fonction

```
trait Description {
    fn decrire(&self) -> String;
    fn identite(&self) -> String {
        String::from("(En savoir plus ...)")
    }
}
```

```
...
fn afficher(element: &impl Description) {
    println!("{}", element.decrire());
}

fn main() {
    let cercle = Cercle {rayon: 5.};
    let carre = Carre {cote: 2.5};
    let rectangle = Rectangle {largeur: 2.5, hauteur:1.5};
    let elle = Eleve {
        nom: "BORÉALE".to_string(),
        prenom: "Aurore".to_string(),
        notes: [8.5, 12.5, 15., 18.5].to_vec()
    };
    afficher(&cercle);
    afficher(&carre);
    afficher(&rectangle);
    afficher(&elle);
}
```

Résultat

```
Cercle (rayon=5) P=31.416 S=78.540
Carré (côté=2.5) P=10 S=6.25
Rectangle (largeur=2.5, hauteur=1.5) P=8 S=4
Aurore BORÉALE, notes [8.5, 12.5, 15.0, 18.5]
```

*La syntaxe **&impl Description** en paramètre de fonction est tout-à-fait adapté pour des cas simples, mais c'est en réalité un raccourci syntaxique pour une forme plus longue, qui s'appelle le **trait lié** qui ressemble à ceci :*

Traits liés

```
fn afficher<T: Description>(element: &T) {
    println!("{}", element.decrire());
}
```

*Cette forme plus longue est équivalente à l'exemple précédent. Nous plaçons les **traits liés** dans la déclaration des paramètres de type générique après les double-points dans les chevrons. La syntaxe **&impl Description** est pratique pour rendre du code plus concis dans le cas où nous avons un seul paramètre à la fonction.*

*La syntaxe du **trait lié** exprime plus de complexité dans le cas où nous devons proposer plusieurs paramètres dans la fonction, comme dans l'exemple qui suit :*

Traits liés

```
...
fn tout_afficher<T: Description, U: Description>(elements: &[&T], element: &U) {
    for element in elements { println!("{}", element.decrire()); }
    element.decrire();
}

fn main() {
    let cercle = Cercle {rayon: 5.};
    let carre = Carre {cote: 2.5};
    let rectangle = Rectangle {largeur: 2.5, hauteur:1.5};
    let elle = Eleve {
        nom: "BORÉALE".to_string(),
        prenom: "Aurore".to_string(),
        notes: [8.5, 12.5, 15., 18.5].to_vec()
    };
    tout_afficher(&[&cercle, &carre, &rectangle], &elle);
}
```

Résultat

```
Cercle (rayon=5) P=31.416 S=78.540
Carré (côté=2.5) P=10 S=6.25
Rectangle (largeur=2.5, hauteur=1.5) P=8 S=4
Aurore BORÉALE, notes [8.5, 12.5, 15.0, 18.5]
```

Des traits liés plus clairs avec l'instruction where

L'utilisation de trop nombreux **traits liés** a aussi ses désavantages. Chaque générique a ses propres **traits liés**, donc les fonctions avec plusieurs paramètres de types génériques peuvent aussi avoir de nombreuses informations de traits liés entre le nom de la fonction et la liste de ses paramètres, ce qui rend la signature de la fonction difficile à lire.

*Pour cette raison, **Rust** propose une syntaxe alternative pour renseigner les traits liés, avec l'instruction **where** après la signature de la fonction que nous connaissons déjà et que nous avons utilisé dans les chapitres traitant des fonctions et des types génériques. Voici ce que cela donne avec l'exemple précédent :*

Traits avec la close where

```
fn tout_afficher<T, U>(elements: &[&T], element: &U)  where T: Description, U: Description {
   for element in elements { println!("{}", element.decrire()); }
   element.decrire();
}
```

Renseigner plusieurs traits liés avec la syntaxe +

Nous pouvons aussi préciser que nous attendons plus d'un trait lié dans les paramètres d'une fonction. La solution relativement simple est tout simplement d'utiliser la syntaxe **+** :

*Je profite de l'occasion pour changer les codes précédents en ayant une approche totalement différente de la notion de formes. Jusqu'à présent, il s'agissait d'une **énumération**. Dans l'exemple ci-dessous, **Forme** devient un **trait**, et chacune d'entre elles (les formes concrètes) propose une description séparée, ce qui peut paraître un inconvénient.*

*Avec le trait, nous sommes plus dans la démarche du **polymorphisme**. Effectivement, après coup, il vous est possible de créer de nouvelles formes, le tout étant de respecter les contraintes liées au **trait** et de définir les méthodes associées, par exemple, le calcul du périmètre et de la surface.*

Traits avec la close where et l'opérateur +

```
trait Forme {
   fn perimetre(&self) -> f64;
   fn surface(&self) -> f64;
}

trait Description {
   fn decrire(&self) -> String;
   fn identite(&self) -> String { String::from("(En savoir plus ...)") }
}

struct Rectangle {
   largeur: f64,
   hauteur: f64
}

struct Cercle { rayon: f64 }

impl Forme for Rectangle {
   fn perimetre(&self) -> f64 { 2. * (self.largeur + self.hauteur) }
   fn surface(&self) -> f64 { self.largeur * self.hauteur }
}

use std::f64::consts::PI;

impl Forme for Cercle {
   fn perimetre(&self) -> f64 { 2. * PI * self.rayon }
   fn surface(&self) -> f64 { PI * self.rayon.powf(2.) }
}

impl Description for Rectangle {
   fn decrire(&self) -> String {
         format!("Rectangle (largeur={}, hauteur={}) P={} S={}", self.largeur, self.hauteur, self.perimetre(),
self.surface())
   }
   fn identite(&self) -> String { format!("Rectangle (largeur={}, hauteur={})", self.largeur, self.hauteur) }
}

impl Description for Cercle {
   fn decrire(&self) -> String {
      format!("Cercle (rayon={}) P={:.3} S={:.3}", self.rayon, self.perimetre(), self.surface())
   }
   fn identite(&self) -> String { format!("Cercle (rayon={})", self.rayon) }
}

fn afficher<T>(element: &T) where T: Description + Forme {
   println!("{}", element.identite());
   println!("{}", element.decrire());
}


fn main() {
   let rectangle = Rectangle{largeur: 2.5, hauteur:1.5};
   let cercle = Cercle {rayon: 2.5};
   afficher(&rectangle);
   afficher(&cercle);
}
```

Résultat

```
Rectangle (largeur=2.5, hauteur=1.5)
Rectangle (largeur=2.5, hauteur=1.5) P=8 S=4
Cercle (rayon=2.5)
Cercle (rayon=2.5) P=15.708 S=19.635
```

Retourner des types qui implémentent des traits

Dans la souplesse du codage en **Rust**, nous pouvons générer des objets qui implémentent un ou plusieurs **traits**. Il est par contre nécessaire que le type de l'objet soit déjà connu avec l'implémentation des différents **traits** souhaités.

Fonction avec des traits en retour

```
trait Forme {
    fn perimetre(&self) -> f64;
    fn surface(&self) -> f64;
}

trait Description {
    fn decrire(&self) -> String;
    fn identite(&self) -> String { String::from("(En savoir plus ...)") }
}

...

fn carre(cote: f64) -> impl Description+Forme {
    Rectangle {
        largeur: cote,
        hauteur: cote
    }
}

fn main() {
    let rectangle = Rectangle{largeur: 2.5, hauteur:1.5};
    let cercle = Cercle {rayon: 2.5};
    afficher(&rectangle);
    afficher(&cercle);
    let une_forme = carre(3.5);
    println!("{}, Périmètre={}", une_forme.identite(), une_forme.perimetre());
}
```

Résultat

```
Rectangle (largeur=2.5, hauteur=1.5)
Rectangle (largeur=2.5, hauteur=1.5) P=8 S=3.75
Cercle (rayon=2.5)
Cercle (rayon=2.5) P=15.708 S=19.635
Rectangle (largeur=3.5, hauteur=3.5), Périmètre=14
```

Traits et types complémentaires

Vous pouvez en **Rust** implémenter n'importe quel **trait** sur n'importe quel type, à partir du moment où soit le **trait**, soit le type est déclaré dans la **caisse** courante. Autrement dit, dès que vous avec besoin d'ajouter une méthode à un type, vous pouvez vous servir d'un **trait** pour y arriver.

> *Dans l'exemple qui suit, le **trait** n'a qu'un usage temporaire : ajouter ponctuellement une méthode à un type existant, ici un **char**. Nous appelons cela un **trait d'extension**.*

Traits sur des types primitifs

```
trait Voyelle {
    fn voyelle_acsii(&self) -> String;
}

impl Voyelle for char {
    fn voyelle_acsii(&self) -> String {
        let voyelles = ['a', 'e', 'i', 'o', 'u', 'y'];
        if voyelles.contains(&self.to_ascii_lowercase()) { "Voyelle".to_string() }
        else { "Consonne".to_string()  }
    }
}

fn main() {
    for caractere in "Salut".chars() {
        print!("'{}' : {}   ", caractere, caractere.voyelle_acsii());
    }
}
```

Résultat

```
'S' : Consonne   'a' : Voyelle   'l' : Consonne   'u' : Voyelle   't' : Consonne
```

Sous-traits

Nous pouvons déclarer un trait qui devient une extension d'un autre **trait**, un **sous-trait**. Les **sous-traits** ressemblent à des sous-interfaces du Java. Ils permettent d'enrichir un trait existant, avec quelques méthodes supplémentaires. L'extension s'exprime avec la syntaxe « **:** ».

Sous-trait

```
trait Forme {
    fn perimetre(&self) -> f64;
    fn surface(&self) -> f64;
}

trait Description : Forme {
    fn decrire(&self) -> String;
    fn identite(&self) -> String { String::from("(En savoir plus ...)") }
}

struct Rectangle { largeur: f64,   hauteur: f64 }
struct Cercle { rayon: f64 }

impl Forme for Rectangle {
    fn perimetre(&self) -> f64 { 2. * (self.largeur + self.hauteur) }
    fn surface(&self) -> f64 { self.largeur * self.hauteur }
}

use std::f64::consts::PI;
impl Forme for Cercle {
    fn perimetre(&self) -> f64 { 2. * PI * self.rayon }
    fn surface(&self) -> f64 { PI * self.rayon.powf(2.) }
}

impl Description for Rectangle {
    fn decrire(&self) -> String {
            format!("Rectangle (largeur={}, hauteur={}) P={} S={}", self.largeur, self.hauteur, self.perimetre(),
self.surface())
    }
    fn identite(&self) -> String { format!("Rectangle (largeur={}, hauteur={})", self.largeur, self.hauteur) }
}

impl Description for Cercle {
    fn decrire(&self) -> String {
        format!("Cercle (rayon={}) P={:.3} S={:.3}", self.rayon, self.perimetre(), self.surface())
    }
    fn identite(&self) -> String { format!("Cercle (rayon={})", self.rayon) }
}

fn afficher<T: Description>(element: &T) {
    println!("{}", element.identite());
    println!("{}", element.decrire());
}


fn main() {
    let rectangle = Rectangle{largeur: 2.5, hauteur:1.5};
    let cercle = Cercle {rayon: 2.5};
    afficher(&rectangle);
    afficher(&cercle);
}
```

Résultat

```
Rectangle (largeur=2.5, hauteur=1.5)
Rectangle (largeur=2.5, hauteur=1.5) P=8 S=3.75
Cercle (rayon=2.5)
Cercle (rayon=2.5) P=15.708 S=19.635
```

Appels de méthodes totalement qualifiés

Rappelons qu'une méthode est une fonction associée à un objet. Les quatre déclarations suivantes sont totalement équivalentes. Généralement, nous utilisons la première variante avec appel de méthode. Les autres sont toutes des **appels qualifiés** qui précisent le type ou le **trait associé** à la méthode. Le dernier appel correspond à un **appel de méthode totalement qualifié**, avec les chevrons. Pour les deux dernières lignes, vous devez **qualifier** la méthode statique **abs()** pour que cela fonctionne correctement, puisque un nombre entier peut être associés à plusieurs types (**i8, i16, i32, i64, u8, etc.**).

Méthodes totalement qualifiés

```
fn main() {
    let s1 = "salut".to_string();
    let s2 = str::to_string("salut");
    let s3 = ToString::to_string("salut");
    let s4 = <str as ToString>::to_string("salut");
    let nombre = -18;
    let absolu = i32::abs(nombre);
}
```

La surcharge des opérateurs - addition

En langage **Rust**, l'expression **a+b** est en fait une abréviation de **a.add(b)**, c'est-à-dire un appel à la méthode **add()** du **trait** de librairie standard **std::ops::Add**. Tous les types numériques standard de **Rust** implémente cette méthode. Pour que l'expression **a+b** fonctionne avec d'autres types, comme par exemple avec des valeurs du type **Complexe**, il faut implémenter le même **trait** pour ce type.

*D'autres **traits** servent aux autres opérations : **a*b** est l'abréviation de **a.mul(b)** qui est une méthode du trait **std::ops::Mul**, **std::ops::Neg** correspond à l'opérateur de négation **–** en préfixe, etc. Pour pouvoir appelez **complexe1.add(complexe2)**, il faut faire en sorte que le trait **Add** soit accessible dans la portée afin que les méthodes le soient aussi.*

Une fois cela fait, vous pouvez considérer toute votre arithmétique comme une série d'appel de méthodes.

Définition de std::ops::Add et std::ops::Mul et ...

```
trait Add<RHS=Self> {
   type Output;
   fn add(self, rhs: RHS) -> Self::Output;
}

trait Mul<RHS=Self> {
   type Output;
   fn mul(self, rhs: RHS) -> Self::Output;
}
...
```

*Ainsi, le trait **Add<T>** donne la capacité d'ajouter une valeur du type **T** à votre propre type. Si vous voulez par exemple pouvoir ajouter des valeurs des types **i32** et **u32** à votre type, celui-ci doit implémenter les deux traits **Add<i32>** et **Add<u32>**.*

*Le paramètre de type du trait qui s'écrit **RHS** correspond par défaut à **Self**. De ce fait, si vous implémentez l'addition entre deux valeurs du même type, vous pouvez le simplifier en écrivant seulement **Add**. Le type associé **Output** sert à décrire le résultat de l'addition.*

Alias de type : le mot clé type s'utilise un peu comme typedef en C++, pour déclarer un nouveau nom (un alias) pour un type existant :

type Table = HashMap<String, Vec<String>> ;

Dans cette déclaration, le nom Table devient un pseudo pour un type particulier de table de hachage HashMap. Cela permet d'avoir de bons raccourcis d'écritures notamment lors de la description de paramètres de fonctions un peu compliqué.

Dans le même registre, Self est un alias de type qui représente n'importe quel type lors de l'implémentation ou de la définitions des méthodes associées aux structures et énumérations.

*Ainsi, pour additionner des valeurs **Complexe<i32>**, il faut que **Complexe<i32>** implémente **Add<Complexe<i32>>**. Nous pouvons simplifier cette syntaxe en écrivant simplement **Add** puisque nous ajoutons un type à lui-même.*

Addition de deux Complexe<i32>

```
#[derive(Clone, Copy, Debug)]
struct Complexe<T> {
   reel : T,
   imaginaire: T
}

use std::ops::Add;
impl Add for Complexe<i32>{
   type Output = Complexe<i32>;
   fn add(self, rhs: Self) -> Self {
      Complexe {
         reel: self.reel + rhs.reel,
         imaginaire: self.imaginaire + rhs.imaginaire
      }
   }
}

fn main() {
   let a = Complexe { reel: 1, imaginaire: 2};
   let b = Complexe { reel: 2, imaginaire: 3};
   println!("{:?}", a+b);
}
```

Résultat

```
Complexe { reel: 3, imaginaire: 5 }
```

*Il serait intéressant de ne pas avoir à implémenter l'addition pour chacun des types nécessaires **Complexe<i32>**, **Complexe<f32>**, **Complexe<f64>**, etc. Les définitions seraient pratiquement des répétitions à l'exception du nom du type.*

Voyons comment écrire une implémentation générique qui conviendra à partir du moment où le type des composants du complexe supporte l'addition :

Addition de deux Complexe<T>

```
#[derive(Clone, Copy, Debug)]
struct Complexe<T> {
    reel : T,
    imaginaire: T
}

use std::ops::Add;

impl<T> Add for Complexe<T> where T: Add<Output=T> {
    type Output = Self;
    fn add(self, autre: Self) -> Self {
        Complexe {
            reel: self.reel + autre.reel,
            imaginaire: self.imaginaire + autre.imaginaire
        }
    }
}

fn main() {
    let a = Complexe { reel: 1, imaginaire: 2};
    let b = Complexe { reel: 2, imaginaire: 3};
    let c = Complexe { reel: 1., imaginaire: 2.5};
    let d = Complexe { reel: 2.2, imaginaire: 3.8};
    println!("{:?}", a+b);
    println!("{:?}", c+d);
}
```

Résultat

```
Complexe { reel: 3, imaginaire: 5 }
Complexe { reel: 3.2, imaginaire: 6.3 }
```

*La mention permet de déclarer que nous limitons **T** aux seuls types qui permettent une addition avec eux-mêmes, ce qui produit une autre valeur **T**.*

La surcharge des opérateurs – tests d'égalité

Rust propose le couple d'opérateur **==** et **!=** pour tester une égalité ou une inégalité. Ce sont des abréviations pour les appels aux méthodes **eq()** (equal) et **ne()** (not equal) du trait **std::cmp::PartialEq.** Voici la définition de **PartialEq** :

std::cmp::PartialEq

```
trait PartialEq<Rhs: ?Sized = Self> {
    fn eq(&self, other: &Rhs) -> bool;
    fn ne(&self, other: &Rhs) -> bool { !self.eq(other); }
}
```

*Vous constatez que la méthode **ne()** possède une définition par défaut, qui ne sera plus à implémenter puisque le comportement proposer est bien entendu ce que l'on attend, quelque soit le type que nous devons résoudre. Il nous suffit alors de définir uniquement **eq()** pour implémenter le trait **PartialEq**.*

Addition et comparaison de deux Complexe<T>

```
#[derive(Clone, Copy, Debug)]
struct Complexe<T> {
    reel : T,
    imaginaire: T
}
use std::ops::Add;
impl<T> Add for Complexe<T> where T: Add<Output=T> {
    ...
}

impl<T: PartialEq> PartialEq for Complexe<T> {
    fn eq(&self, autre: &Complexe<T>) -> bool {
        self.reel==autre.reel && self.imaginaire==autre.imaginaire
    }
}
```

```rust
fn main() {
   let a = Complexe { reel: 1., imaginaire: 2.5};
   let b = Complexe { reel: 2.2, imaginaire: 3.8};
   println!("{:?}", a+b);
   println!("{} <=> {}", a==b, a!=b);
}
```

Résultat

```
Complexe { reel: 3.2, imaginaire: 6.3 }
false <=> true
```

*Les implémentation de **PartialEq** se présentent très souvent dans ce format : elles comparent chaque champ de l'opérande gauche au champ correspondant de l'opérande droit. Cela peut rapidement devenir long à écrire.*

*Du fait que le test d'égalité est une opération basique, vous serez heureux d'apprendre que **Rust** peut générer une implémentation automatique pour **PartialEq**. Il suffit de citer **PartialEq** dans l'attribut **derive** de la définition du type :*

Addition et comparaison de deux Complexe<T>

```rust
#[derive(Clone, Copy, Debug, PartialEq)]
struct Complexe<T> {
   reel : T,
   imaginaire: T
}

use std::ops::Add;
impl<T> Add for Complexe<T> where T: Add<Output=T> {
    ...
}

fn main() {
   let a = Complexe { reel: 1., imaginaire: 2.5};
   let b = Complexe { reel: 2.2, imaginaire: 3.8};
   println!("{:?}", a+b);
   println!("{} <=> {}", a==b, a!=b);
}
```

Résultat

```
Complexe { reel: 3.2, imaginaire: 6.3 }
false <=> true
```

La surcharge des opérateurs – comparaisons ordonnées

Rust définit le comportement des opérateurs de comparaison ordonnée **<, >, ≤** et **≥** dans un seul trait **std::cmp::Ord.** Vous avez ci-dessous la définition de **PartialOrd**. Très logiquement **PartialOrd<Rhs>** est une extension de **PartialEq<Rhs>**, c'est-à-dire que vous ne pouvez faire une comparaison ordonnée que sur un type pour lequel vous pouvez faire une comparaison d'égalité.

std::cmp::PartialOrd

```rust
trait PartialOrd<Rhs=Self> : PartialEq<Rhs> where Rhs: ?Sized {
   fn partial_cmp(&self, autre: &Rhs) -> Option<Ordering>;

   fn lt(&self, autre: &Rhs) -> bool { self.partial_cmp(autre) == Some(Less) }          // x < y

   fn le(&self, autre: &Rhs) -> bool {                                                  // x <= y
      match self.partial_cmp(autre) {
         Some(Less) | Some(Equal) => true,
         _ => false
      }
   }
   fn gt(&self, autre: &Rhs) -> bool { self.partial_cmp(autre) == Some(Greater) }       // x > y

   fn ge(&self, autre: &Rhs) -> bool {                                                  // x >= y
      match self.partial_cmp(autre) {
         Some(Greater) | Some(Equal) => true,
         _ => false
      }
   }
}
```

*Nous n'avons plus qu'une seule méthode à implémenter dans **PartialOrd** : **partial_cmp()**. Lorsque cette fonction renvoie **Some(x)**, **x** correspond à la relation **self** avec **autre.** Si **partial_cmp()** renvoie **None**, cela signifie que **self** et **autre** ne sont pas ordonnés l'un par rapport à l'autre. Aucun des deux n'est plus grand que l'autre et ils ne sont pas égaux. Parmi tous les types primitifs de Rust, seules les valeurs flottantes peuvent renvoyer **None** (avec une valeur infinie représentée par **NaN**).*

énumération Ordering

```
enum Ordering {
   Less,        // self < autre
   Equal,       // self == autre
   Greater      // self > autre
}
```

*Les expressions telles que **x<y** ou **x>=y** sont en fait de raccourcis pour les appels de méthodes de **PartialOrd**. Si vous êtes certain que les valeurs de deux types sont toujours ordonnées l'une par rapport à l'autre, vous pouvez alors implémenter le trait le plus strict nommé **std::cmp ::Ord**.*

std::cmp::Ord

```
trait Ord: Eq + PartialOrd<Self> {
   fn cmp(&self, autre: &Self) -> Ordering;
}
```

*Cette méthode **cmp()** renvoie un **Ordering**, et non un **Option<Ordering>** comme le fait **partial_cmp()**. La méthode **cmp()** déclare toujours que ses paramètre sont égaux ou bien indique leur ordre relatif. Quasiment tous les qui implémentent **PartialOrd** doivent aussi implémenter **Ord** ; dans la librairie standard, les seuls exceptions sont les flottants **f32** et **f64**.*

*Puisque le concept d'ordre naturel n'a de sens pour des nombres complexes, nous n'allons pas utiliser le type **Complexe** pour illustrer une implémentation de **PartialOrd**. En guise d'exemple, choisissons un type qui représente l'ensemble des nombres qui appartiennent à une plage de valeurs.*

Comparaison de deux intervalles

```
#[derive(Debug, PartialEq)]
struct Intervalle<T> {
   bas: T,
   haut: T
}

use std::cmp::{Ordering, PartialOrd};

impl<T: PartialOrd> PartialOrd<Intervalle<T>> for Intervalle<T> {
   fn partial_cmp(&self, autre: &Intervalle<T>) -> Option<Ordering> {
      if self==autre              { Some(Ordering::Equal) }
      else if self.bas >= autre.haut  { Some(Ordering::Greater) }
      else if self.haut <= autre.bas  { Some(Ordering::Less) }
      else                        { None }
   }
}

fn main() {
   let gauche = Intervalle {bas: 10, haut: 30};
   let droite = Intervalle {bas: 20, haut: 40};
   print!("{}  ", gauche < droite);
   print!("{}  ", gauche >= droite);
   print!("{}  ", Intervalle {bas: 10, haut: 20} < Intervalle {bas: 20, haut: 40});
   print!("{}  ", Intervalle {bas: 7, haut: 8} >= Intervalle {bas: 0, haut: 1});
   print!("{}  ", Intervalle {bas: 7, haut: 8} <= Intervalle {bas: 7, haut: 8});
   print!("{}  ", Intervalle {bas: 7, haut: 8} > Intervalle {bas: 7, haut: 8});
   print!("{}  ", Intervalle {bas: 7, haut: 8} == Intervalle {bas: 7, haut: 8});
   print!("{}  ", Intervalle {bas: 7, haut: 8} != Intervalle {bas: 7, haut: 8});
}
```

Résultat

```
false  false  true  true  true  false  true  false
```

Notre but est de pouvoir comparer deux plages dans cet ordre partiel : une plage est considérée comme inférieure à une autre si elle est totalement située avant celle-ci, sans chevauchement. Dès qu'il y a chevauchement sans égalité (partiel), nous considérons que les plages ne sont pas ordonnées : seuls certains éléments d'une plage sont inférieurs à certains de l'autre plage. Enfin, deux plages en égalité sont vraiment égales.

Les traits dérivables

Dans de nombreux endroits, nous avons utilisé l'attribut **derive**, que vous pouvez appliquer lors d'une définition de structure ou d'énumération. L'attribut **derive** génère automatiquement du code qui implémente un **trait** avec son propre traitement par défaut. Cette démarche nous donne un raccourci d'écriture et nous évite de réécrire des algorithmes déjà prédéfinis.

*Les traits prédéfinis de la bibliothèque standard concerne les traits associés : à la comparaison (**Eq, PartialEq, Ord, PartialOrd**), au clonage (**Clone**), à la copie (**Copy**), au formatage de l'affichage (**Debug**), aux valeurs par défaut (**Default**).*

*Le reste des traits définis dans la bibliothèque standard ne peuvent pas être implémentés sur des types en utilisant **derive**. Ces traits n'ont pas de comportement logique par défaut, et c'est à vous de les implémenter de la façon la plus appropriée pour ce que vous souhaitez accomplir.*

*Un exemple de **trait** qui ne peut pas être dérivé est **Display**, qui permet de formater la donnée pour les utilisateurs finaux. Vous devez toujours réfléchir au formatage du type le plus approprié pour un utilisateur final. Quelles parties d'un type un utilisateur final devrait pouvoir voir ? Sous quelle forme les données devraient être les plus intéressantes pour eux ? Le compilateur de **Rust** n'a pas cette intuition, donc il ne peut pas fournir un comportement par défaut à votre place.*

Debug pour l'affichage au développeur

*Le trait **Debug** permet le formatage de débogage pour mettre en forme en tant que chaînes de caractères, que vous pouvez utiliser en ajoutant **:?** dans un espace réservé **{}**.*

*Le trait **Debug** vous permet d'afficher des instances d'un type pour des besoins de débogage, afin que vous et les autres développeurs qui utilisent votre type puissent inspecter une de ses instances à un endroit précis de l'exécution du programme.*

PartialEq et Eq pour comparer l'égalité

*Le trait **PartialEq** vous permet de comparer des instances d'un type pour vérifier leur égalité et permet l'utilisation des opérateurs **==** et **!=**.*

*L'application de **derive** avec **PartialEq** implémente la méthode **eq()**. Lorsque **PartialEq** est dérivé sur une structure, deux instances ne peuvent être égales seulement si tous leurs champs sont égaux, et les instances ne sont pas égales si un des champs n'est pas égal. Lorsque ce trait est dérivé sur une énumération, chaque variante est égale à elle-même et n'est pas égale aux autres variantes.*

*Le trait **Eq** n'a pas de méthode. Son rôle est de signaler que pour chaque valeur du type annoté, la valeur est égale à elle-même. Le trait **Eq** peut seulement être appliqué sur des types qui implémentent **PartialEq**, bien que tous les types qui implémentent **PartialEq** ne puissent pas implémenter **Eq**.*

*Un exemple de ceci sont les types de nombres à virgule flottante : l'implémentation des nombres à virgule flottante stipule que deux instances ayant la valeur **"not-a-number"** (**NaN**, c'est-à-dire "ceci n'est pas un nombre") ne sont pas égales entre elles.*

PartialOrd et Ord pour comparer les ordres de grandeur

*Le trait **PartialOrd** vous permet de comparer des instances d'un type pour pouvoir les trier. Un type qui implémente **PartialOrd** peut être utilisé avec les opérateurs **<, >, <=** et **>=**. Vous pouvez appliquer uniquement le trait **PartialOrd** aux types qui implémentent aussi **PartialEq**.*

*L'application de **derive** avec **PartialOrd** implémente la méthode **partial_cmp()**, qui retourne un **Option<Ordering>** qui vaudra **None** lorsque les valeurs fournies ne fournissent pas un ordre.*

*Un exemple de valeur qui ne produit pas d'ordre, même si la plupart des valeurs de ce type peuvent être comparées, est la valeur **"not-a-number"** (**NaN**) des virgules flottantes. L'appel à **partial_cmp()** entre n'importe quel nombre à virgule flottante et la valeur **NaN** de virgule flottante va retourner **None**.*

*Lorsqu'il est dérivé sur une structure, **PartialOrd** compare deux instances en comparant les valeurs de chaque champ dans l'ordre dans lequel les champs apparaissent dans la définition de la structure. Lorsqu'il est dérivé sur des énumérations, les variantes de l'énumération déclarées plus tôt dans la définition de l'énumération sont considérées inférieures aux variantes déclarées ensuite.*

*Le trait **Ord** vous permet de savoir si un ordre valide existe toujours entre deux valeurs du type annoté. Le trait **Ord** implémente la méthode **cmp()**, qui retourne un **Ordering** plutôt qu'une **Option<Ordering>** car un ordre valide sera toujours possible.*

*Vous pouvez appliquer le trait **Ord** uniquement sur les types qui implémentent aussi **PartialOrd** et **Eq** (et **Eq** nécessite **PartialEq**). Lorsqu'il est dérivé sur des structures et des énumérations, **cmp()** se comporte de la même manière que l'implémentation de **partial_cmp()** dérivée de **PartialOrd**.*

Clone et Copy pour dupliquer des valeurs

*Le trait **Clone** vous permet de créer explicitement une copie profonde d'une valeur, et le processus de duplication peut impliquer l'exécution d'un code arbitraire pour copier les données stockées dans le tas.*

*Utiliser **derive** avec **Clone** implémente la méthode **clone()**, qui, lorsqu'elle est implémentée sur tout le type, fait appel à **clone()** sur chaque constituant du type. Cela signifie que tous les champs ou les valeurs dans le type doivent aussi implémenter **Clone**.*

*Clone est par exemple nécessaire lorsque nous appelons la méthode **to_vec()** sur une **slice**. La **slice** ne prend pas possession des instances du type qu'il contient, mais le vecteur retourné par **to_vec()** va avoir besoin de prendre possession de ses instances, donc **to_vec()** fait appel à **clone()** sur chaque élément. C'est pourquoi le type stocké dans la slice doit implémenter **Clone**.*

*Le trait **Copy** vous permet de dupliquer une valeur en copiant uniquement les éléments stockés sur la pile ; il n'est pas nécessaire d'avoir de code arbitraire.*

*Le trait **Copy** ne définit pas de méthode, volontairement pour empêcher les développeurs de surcharger ces méthodes et ainsi violer l'affirmation qu'aucun code arbitraire est exécuté à la copie. Ainsi, tous les développeurs peuvent compter sur le fait qu'une copie de valeur est très rapide.*

*Vous pouvez utiliser **derive** avec **Copy** sur n'importe quel type constitué d'éléments qui implémentent aussi **Copy**. Vous ne pouvez appliquer le trait **Copy** que sur des types qui implémentent aussi **Clone**, car un type qui implémente **Copy** a aussi une implémentation triviale de **Clone** qui procède aux mêmes actions que **Copy**.*

*Le trait **Copy** est rarement nécessaire ; les types qui implémentent **Copy** peuvent être optimisés, ce qui veut dire que vous n'avez pas à appeler **clone()**, ce qui rend le code plus concis. Tout ce que vous pouvez accomplir avec **Copy**, vous pouvez le faire avec **Clone**, mais le code risque d'être plus lent ou doit parfois utiliser **clone()**.*

Default pour des valeurs par défaut

*Le trait **Default** vous permet de créer une valeur par défaut pour un type. Implémenter **Default** avec **derive** ajoute la fonction **default()**. Cette fonction **default()** fait elle-même appel à la fonction **default()** sur chaque élément du type, ce qui signifie que tous les champs ou les valeurs dans le type doit aussi implémenter **Default**.*

*La fonction **Default::default()** est couramment utilisé en association avec la syntaxe de modification de structures que nous avons vu dans la section associée.Vous pouvez personnaliser quelques champs d'une structure et ensuite définir et utiliser une valeur par défaut pour le reste des champs en utilisant **..Default::default()**.*

Implémentation du trait Display

Je vous propose de réaliser un affichage circonstancié sur la structure **Point** que nous avons déjà abordé lors de l'étude sur les énumérations et les motifs de sélection. L'objectif est de vérifier si le point se situe sur un des axes. Vous devez implémenter la méthode **fmt()** du trait **Display** pour cela.

Implémentation de fmt() de std::fmt::Display

```
struct Point {
   x:i32,
   y:i32
}

use std::fmt::{Display, Formatter, Result};

impl Display for Point {
   fn fmt(&self, f: &mut Formatter<'_>) -> Result {
      let commentaire = match self {
         Point {x: 0, y: 0}  => "Origine",
         Point {x: 0, y: _}  => "Axe des X",
         Point {y: 0, x: _}  => "Axe des Y",
         Point { .. }        => "Aucun des axes"
      };
      write!(f, "[Point({}, {}) {}]", self.x, self.y, commentaire)
   }
}

fn main() {
   let p0 = Point{x:0, y:0};
   let px = Point{x:0, y:5};
   let py = Point{x:3, y:0};
   let p = Point{x:3, y:5};

   println!("po={}, px={}, py={}, p={}", p0, px, py, p);
}
```

Résultat

```
po=[Point(0, 0) Origine], px=[Point(0, 5) Axe des X],
py=[Point(3, 0) Axe des Y], p=[Point(3, 5) Aucun des axes]
```

Créer des synonymes de noms avec les alias de type

Rust fournit la possibilité de déclarer un alias de type pour donner un autre nom à un **type** déjà existant. Pour faire cela, nous utilisons le mot-clé **type**. Par exemple, nous pouvons créer l'alias **Kilometres** pour un **i32**.

*Après une telle déclaration, l'alias **Kilometres** est un synonyme de **i32**. Les valeurs qui possède le type **Kilometre** seront traités comme si elles étaient du type **i32** :*

Alias de type

```
fn main() {
   type Kilometres = i32;

   let x: i32 = 5;
   let y: Kilometres = 5;
   println!("x + y = {}", x + y);
}
```

Résultat

```
x + y = 10
```

*Comme **Kilometres** et **i32** sont du même type, nous pouvons additionner les valeurs des deux types et nous pouvons envoyer des valeurs **Kilometres** aux fonctions qui prennent des paramètres **i32**.*

*Un alias de type simplifie ce code en réduisant la répétition. Dans l'exemple ci-dessous, nous ajoutons un alias **Thunk** pour ce type verbeux et qui peut être utilisé n'importe où dans le code et qui sera alors beaucoup plus facile à lire.*

Avant

```
fn main() {
    let f: Box<dyn Fn() + Send + 'static> = Box::new(|| println!("salut"));

    fn prend_un_long_type(f: Box<dyn Fn() + Send + 'static>) {
        // -- partie masquée ici --
    }
    fn retourne_un_long_type() -> Box<dyn Fn() + Send + 'static> {
        // -- partie masquée ici --
        Box::new(|| ())
    }
}
```

Après

```
fn main() {
    type Thunk = Box<dyn Fn() + Send + 'static>;

    let f: Thunk = Box::new(|| println!("salut"));

    fn prend_un_long_type(f: Thunk) {
        // -- partie masquée ici --
    }
    fn retourne_un_long_type() -> Thunk {
        // -- partie masquée ici --
        Box::new(|| ())
    }
}
```

*Ce code est plus facile à lire et écrire ! Choisir un nom plus explicite pour un alias peut aussi vous aider à communiquer ce que vous voulez faire (**thunk** est un terme désignant du code qui peut être évalué plus tard, donc c'est un nom approprié pour une fermeture qui est stockée).*

*Les alias de type sont couramment utilisés avec le type **Result<T, E>** pour réduire la répétition. Regardez le module **std::io** de la bibliothèque standard. Les opérations d'entrée/sortie retournent parfois un **Result<T, E>** pour gérer les situations lorsque les opérations échouent.*

*Cette bibliothèque possède une structure **std::io::Error** qui représente toutes les erreurs possibles d'entrée/sortie. De nombreuses fonctions dans **std::io** vont retourner un **Result<T, E>** avec **E** qui est **std::io::Error,** ces fonctions sont dans le trait **Write** :*

Avant

```
fn main() {
    use std::fmt;
    use std::io::Error;

    pub trait Write {
        fn write(&mut self, buf: &[u8]) -> Result<usize, Error>;
        fn flush(&mut self) -> Result<(), Error>;

        fn write_all(&mut self, buf: &[u8]) -> Result<(), Error>;
        fn write_fmt(&mut self, fmt: fmt::Arguments) -> Result<(), Error>;
    }
}
```

*Le **Result<..., Error>** est répété plein de fois. Ainsi, **std::io** possède ce type de déclaration d'alias :*

type Result<T> = std::result::Result<T, std::io::Error> ;

*Comme cette déclaration est dans le module **std::io**, nous pouvons utiliser l'alias **std::io::Result<T>** — qui est un **Result<T, E>** avec le **E** qui est déjà renseigné comme étant un **std::io::Error**. Les fonctions du trait **Write** ressemblent finalement à ceci :*

Après

```
use std::fmt;

type Result<T> = std::result::Result<T, std::io::Error>;

pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    fn write_all(&mut self, buf: &[u8]) -> Result<()>;
    fn write_fmt(&mut self, fmt: fmt::Arguments) -> Result<()>;
}
```

*L'alias de type nous aide sur deux domaines : il permet de faciliter l'écriture du code et il nous donne une interface uniforme pour tout **std::io**. Comme c'est un alias, c'est simplement un autre **Result<T, E>**, ce qui signifie que nous pouvons utiliser n'importe quelle méthode qui fonctionne avec **Result<T, E>**, ainsi que les syntaxes spéciales comme l'opérateur **?**.*

Les types à taille dynamique et le trait Sized

Vu qu'il est nécessaire pour **Rust** de connaître certains détails, comme la quantité d'espace à allouer à une valeur d'un type donné, il existe un aspect de ce système de type qui peut être déroutant : le concept des types à taille dynamique. Parfois appelés **DST** (**D**ynamically **S**ized **T**ypes) ou types sans taille, ces types nous permettent d'écrire du code qui utilisent des valeurs qui ne peuvent être connues uniquement qu'à l'exécution.

*Voyons les détails d'un type à taille dynamique qui s'appelle **str**, que nous avons utilisé dans nos études. Plus précisément **&str**, car **str** en lui-même est un **DST**. Nous ne connaître la longueur de la chaîne de caractère qu'à l'exécution, ce qui signifie que nous ne pouvons pas ni créer une variable de type **str**, ni prendre prendre en argument un type **str**. Imaginons le code suivant, qui ne devrait pas fonctionner :*

Ne fonctionne pas

```
fn main() {
    let s1: str = "Salut tout le monde !";
    let s2: str = "Comment ça va ?";
}
```

__Rust__ a besoin de savoir combien de mémoire allouer pour chaque valeur d'un type donné, et toutes les valeurs de ce type doivent utiliser la même quantité de mémoire. Si __Rust__ nous avait autorisé à écrire ce code, ces deux valeurs __str__ devraient occuper la même quantité de mémoire. Mais elles possèdent deux longueurs différentes : __s1__ prend __21__ octets en mémoire alors que __s2__ en a besoin de __15__. C'est pourquoi il est impossible de créer une variable qui stocke un type à taille dynamique.

Donc que pouvons nous faire ? Dans ce cas, vous connaissez déjà la réponse : nous faisons en sorte que le type de __s1__ et __s2__ soit __&str__ plutôt que __str__. Souvenez-vous que nous avions dit que la structure de données __slice__ stockait l'emplacement de départ et la longueur de la slice.

Aussi, bien qu'un __&T__ soit une seule valeur qui stocke l'adresse mémoire d'où se trouve le __T__, un __&str__ représente deux valeurs : l'adresse du __str__ et sa longueur. Ainsi, nous pouvons connaître la taille d'une valeur __&str__ à la compilation : elle vaut deux fois la taille d'un __usize__.

Ce faisant, nous connaissons toujours la taille d'un __&str__, peu importe la longueur de la chaîne de caractères sur laquelle cela pointe. Généralement, c'est comme cela que les types à taille dynamique sont utilisés en __Rust__ : ils possèdent des métadonnées supplémentaires qui stockent la taille des informations dynamiques. La règle d'or des types à taille dynamique est que nous devons toujours placer les valeurs à types à taille dynamique dans une sorte de pointeur.

Pour pouvoir travailler avec les __DST__, __Rust__ possède un trait particulier __Sized__ pour déterminer si oui ou non la taille d'un type est connue à la compilation. Ce trait est automatiquement implémenté sur tout ce qui possède une taille connue à la compilation. De plus, __Rust__ ajoute implicitement le trait lié __Sized__ sur chaque fonction générique. Ainsi, la définition d'une fonction générique comme celle ci-dessous est équivalente à celle d'après :

Générique à taille fixée

```
fn generique<T>(t: T) {
    // -- partie masquée ici --
}
```

Générique équivalent

```
fn generique<T: Sized>(t: T) {
    // -- partie masquée ici --
}
```

Par défaut, les fonctions génériques fonctionne uniquement sur les types qui possède une taille connue à la compilation. Cependant, vous pouvez utiliser la syntaxe spéciale suivante pour éviter cette restriction :

Générique à taille variable

```
fn generique<T: ?Sized>(t: &T) {
    // -- partie masquée ici --
}
```

*Le trait lié **?Sized** est l'opposé du trait lié **Sized** : nous pourrions lire ceci comme étant **"T peut être ou non un Sized"**. Cette syntaxe est disponible uniquement pour **Sized**, et non pas pour les autres traits.*

*Remarquez aussi que nous avons changé le type du paramètre **t** de **T** en **&T**. Comme ce type pourrait ne pas être un **Sized**, nous devons l'utiliser avec quelque chose qui sert de pointeur. Dans ce cas, nous avons choisi une référence.*

Placer des types à remplacer dans les définitions des traits grâce aux types associés

Les types associés connectent un type à remplacer avec un trait afin que la définition des méthodes puisse utiliser ces types à remplacer dans leur signature. Celui qui implémente un trait doit renseigner un type concret pour être utilisé à la place du type à remplacer pour cette implémentation précise. Ainsi, nous pouvons définir un trait qui utilise certains types sans avoir besoin de savoir exactement quels sont ces types jusqu'à ce que ce trait soit implémenté.

*Un exemple de trait avec un type associé est le trait **Iterator** que fournit la bibliothèque standard. Le type associé **Item** permet de renseigner le type de chacune des valeurs.*

Le trait Iterator

```
pub trait Iterator {
  type Item;

  fn next(&mut self) -> Option<Self::Item>;
}
```

*Le type **Item** est un type à remplacer, et la définition de la méthode **next()** informe qu'elle va retourner des valeurs du type **Option<Self::Item>**. Ceux qui implémenterons le trait **Iterator** devront renseigner un type concret pour **Item**, et la méthode **next()** retournera une **Option** qui contiendra une valeur de ce type concret.*

Implémentation du trait Iterator

```
struct Compteur {
  compteur: u32,
}

impl Compteur {
  fn new() -> Compteur {
    Compteur { compteur: 0 }
  }
}

impl Iterator for Compteur {
  type Item = u32;

  fn next(&mut self) -> Option<Self::Item> {
    // -- partie masquée ici --
    if self.compteur < 5 {
      self.compteur += 1;
      Some(self.compteur)
    } else {
      None
    }
  }
}
```

Cette syntaxe ressemble aux génériques. Donc pourquoi ne pas définir le trait Iterator à l'aide des génériques, comme dans l'exemple suivant ?

trait Iterator générique

```
pub trait Iterator<T> {
  fn next(&mut self) -> Option<T>;
}
```

*La différence est que lorsque vous utilisez les génériques, nous devons annoter les types dans chaque implémentation ; et comme nous pouvons aussi implémenter **Iterator<String> for Compteur** sur d'autres types, nous pourrions alors avoir plusieurs implémentations de **Iterator** pour **Compteur**.*

*Autrement dit, lorsqu'un trait possède un paramètre générique, il peut être implémenté sur un type plusieurs fois, en changeant à chaque fois le type concret du paramètre de type générique. Lorsque nous utilisons la méthode **next()** sur **Compteur**, nous devons appliquer une annotation de type pour indiquer quelle implémentation de **Iterator** nous souhaitons utiliser.*

*Avec les types associés, nous n'avons pas besoin d'annoter les types car nous n'implémentons pas un trait plusieurs fois sur un même type. Dans l'exemple ci-dessus qui contient la définition qui utilise les types associés, nous pouvons uniquement choisir une seule fois quel sera le type de **Item**, car il ne peut y avoir qu'un seul **impl Iterator for Compteur**. Nous n'avons pas eu besoin de préciser que nous souhaitions avoir un itérateur de valeurs **u32** à chaque fois que nous faisons appel à **next()** sur **Compteur**.*

Les paramètres de types génériques par défaut et la surcharge d'opérateur

Lorsque nous utilisons les paramètres de types génériques, nous pouvons renseigner un type concret par défaut pour le type générique. Cela évite de contraindre ceux qui implémentent ce trait d'avoir à renseigner un type concret si celui par défaut fonctionne bien.

*La syntaxe pour renseigner un type par défaut pour un type générique est **<TypeARemplacer=TypeConcret>** lorsque nous déclarons le type générique.*

*Un bon exemple d'une situation pour laquelle cette technique est utile est avec la surcharge d'opérateurs. La surcharge d'opérateur permet de personnaliser le comportement d'un opérateur (comme **+**) dans des cas particuliers.*

***Rust** ne vous permet pas de créer vos propres opérateurs ou de surcharger des opérateurs. Mais vous pouvez surcharger les opérations et les traits listés dans **std::ops** en implémentant les traits associés à l'opérateur. Dans l'exemple suivant*

*nous surchargeons l'opérateur **+** pour additionner ensemble deux instances de **Point**. Nous pouvons faire cela en implémentant le trait **Add** sur une structure **Point** :*

Le trait Add

```
use std::ops::Add;

#[derive(Debug, PartialEq)]
struct Point {
  x: i32,
  y: i32,
}

impl Add for Point {
  type Output = Point;

  fn add(self, other: Point) -> Point {
    Point {
      x: self.x + other.x,
      y: self.y + other.y,
    }
  }
}

fn main() {
  println!("{:?}", Point { x: 1, y: 0 } + Point { x: 2, y: 3 } );
}
```

Résultat

```
Point { x: 3, y: 3 }
```

*La méthode **add()** ajoute les valeurs **x** de deux instances de **Point** ainsi que les valeurs **y** de deux instances de **Point** pour créer un nouveau **Point**. Le trait **Add** possède un type associé **Output** qui détermine le type retourné pour la méthode **add()**. Vous devez impérativement spécifier le type de **Output**, puisque la définition du trait **Add** le réclame. Voici sa définition :*

Le trait Add

```
trait Add<Rhs=Self> {
  type Output;

  fn add(self, rhs: Rhs) -> Self::Output;
}
```

*Le trait **Add** possède une méthode avec un type associé. La nouvelle partie concerne **Rhs=Self :** cette syntaxe s'appelle les paramètres de types par défaut. Le paramètre de type générique **Rhs** (c'est le raccourci de **"Right Hand Side"**) qui définit le type du paramètre **rhs** dans la méthode **add()**. Si nous ne renseignons pas de type concret pour **Rhs** lorsque nous implémentons le trait **Add**, le type de **Rhs** sera par défaut **Self**, qui sera le type sur lequel nous implémentons **Add**.*

*Lorsque nous implémentons **Add** sur **Point**, nous utilisons la valeur par défaut de **Rhs** car nous voulons additionner deux instances de **Point**. Voyons un exemple d'implémentation du trait **Add** dans lequel nous souhaitons personnaliser le type **Rhs** plutôt que d'utiliser celui par défaut.*

*Nous possédons deux structures, **Millimetres** et **Metres**, qui stockent des valeurs dans différentes unités. Nous voulons pouvoir additionner les valeurs en millimètres avec les valeurs en mètres et appliquer l'implémentation de **Add** pour pouvoir faire la conversion correctement. Nous pouvons implémenter **Add** sur **Millimetres** avec **Metres** comme étant le **Rhs**, comme dans l'exemple ci-dessous.*

Le trait Add

```
fn main() {
  use std::ops::Add;

  struct Milimetres(u32);
  struct Metres(u32);

  impl Add<Metres> for Milimetres {
    type Output = Milimetres;

    fn add(self, other: Metres) -> Milimetres {
      Milimetres(self.0 + (other.0 * 1000))
    }
  }
}
```

*Pour additionner **Milimetres** et **Metres**, nous renseignons **impl Add<Metres>** pour régler la valeur du paramètre de type **Rhs** au lieu d'utiliser la valeur par défaut **Self**. Vous utiliserez les paramètres de types par défaut dans deux principaux cas :*

- *Pour étendre un type sans casser le code existant*
- *Pour permettre la personnalisation dans des cas spécifiques que la plupart des utilisateurs n'auront pas.*

*Le trait **Add** de la bibliothèque standard est un exemple du second cas : généralement, vous additionnez deux types similaires, mais le trait **Add** offre la possibilité de personnaliser cela. L'utilisation d'un paramètre de type par défaut dans la définition du trait **Add** signifie que vous n'aurez pas à renseigner de paramètre en plus la plupart du temps. Autrement dit, il n'est pas nécessaire d'avoir recours à des assemblages de code, ce qui facilite l'utilisation du trait.*

Le premier cas est similaire au second mais dans le cas inverse : si vous souhaitez ajouter un paramètre de type à un trait existant, vous pouvez lui en donner un par défaut pour permettre l'ajout des fonctionnalités du trait sans casser l'implémentation actuelle du code.

Utiliser les supertraits pour utiliser la fonctionnalité d'un trait dans un autre trait

Quelques fois, vous pourriez avoir besoin d'un **trait** pour utiliser une autre fonctionnalité d'un **trait**. Dans ce cas, vous devez pouvoir compter sur le fait que le **trait** dépendant soit bien implémenté. Le **trait** sur lequel vous comptez est alors un **supertrait** du **trait** que vous implémentez.

*Par exemple, imaginons que nous souhaitons créer un trait **OutlinePrint** qui offre une méthode **outline_print()** affiche une valeur entourée d'astérisques. Pour une structure **Point** qui implémente **Display** pour afficher **(x, y)**, lorsque nous faisons appel à **outline_print()** sur une instance de **Point** qui a **1** pour valeur de **x** et **3** pour **y**, nous devrions afficher ceci :*

Résultat

```
**********
*        *
* (1, 3) *
*        *
**********
```

*Dans l'implémentation de **outline_print()**, nous souhaitons utiliser la fonctionnalité du trait **Display**. Toutefois, nous devons renseigner que le trait **OutlinePrint** fonctionnera uniquement pour les types qui auront aussi implémenté **Display** et qui fourniront la fonctionnalité dont a besoin **OutlinePrint**.*

*Nous pouvons faire ceci dans la définition du trait en renseignant **OutlinePrint: Display**. Cette technique ressemble à l'ajout d'un **trait** lié au **trait**. L'exemple suivant montre une implémentation du trait **OutlinePrint**.*

Le trait OutlinePrint

```
use std::fmt;

trait OutlinePrint: fmt::Display {
  fn outline_print(&self) {
    let valeur = self.to_string();
    let largeur = valeur.len();
    println!("{}", "*".repeat(largeur + 4));
    println!("*{}*", " ".repeat(largeur + 2));
    println!("* {} *", valeur);
    println!("*{}*", " ".repeat(largeur + 2));
    println!("{}", "*".repeat(largeur + 4));
  }
}
```

*Comme nous précisons que **OutlinePrint** nécessite le trait **Display**, nous pouvons utiliser la fonction **to_string()** qui est automatiquement implémentée pour n'importe quel type qui implémente **Display**. Si nous avions essayé d'utiliser **to_string()** sans ajouter un double-point et en renseignant le trait **Display** après le nom du trait, nous obtiendrons alors une erreur qui nous informerait qu'il n'y a pas de méthode **to_string()** pour le type **&Self** dans la portée courante.*

*D'après cette définition, n'oublions pas que nous devons implémenter **Display** sur **Point** afin de répondre aux besoins de **OutlinePrint**, comme ceci :*

Le trait Display

```
impl fmt::Display for Point {
  fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
    write!(f, "({}, {})", self.x, self.y)
  }
}
```

*Suite à cela, l'implémentation du trait **OutlinePrint** sur **Point** se compile avec succès, et nous pourrons appeler **outline_print()** sur une instance de **Point** pour l'afficher dans le cadre constitué d'astérisques.*

Utilisation du trait OutlinePrint

```
fn main() {
  let p = Point { x: 1, y: 3 };
  p.outline_print();
}
```