Les énumérations existent en **C++**. Elles vous permettent de définir des types pour des valeurs correspondant à une série de constantes nommées. Vous définissez par exemple un type nommé **Couleur** et possédant les valeurs **Rouge, Orange, Jaune**, etc. **Rust** accepte ce genre d'énumération, mais il permet d'aller beaucoup plus loin. Une énumération **Rust** peut contenir des données de types complètements différents.

Vous utiliserez les énumérations dès qu'une valeur peut correspondre à une chose ou à une autre, qu'elle puisse prendre une valeur parmi un choix disponible quelque soit ce choix. Le prix à payer pour cette souplesse d'écriture est qu'il faut accepter que l'accès aux données soit sûr, en utilisant les motifs de sélection.

*Les motifs **Rust** peuvent être vus comme un système d'expressions régulières pour les données. Un motif permet de tester si une valeur a ou pas la forme désirée. Un motif permet d'extraire des champs depuis une **structure** ou un **tuple** des variables locales, en un seul geste. Comme pour les expressions régulières, les motifs sont compacts, permettant généralement de tout exprimer en une seule ligne.*

*Ensuite, nous examinerons une énumération particulièrement utile qui s'appelle **Option** et qui permet de décrire des situations où la valeur peut être soit quelque chose, soit rien. Ensuite, nous regarderons comment le filtrage par motif avec l'expression **match** peut faciliter l'exécution de codes différents pour chaque valeur d'une énumération.*

*Enfin, nous analyserons pourquoi la construction **if let** est un autre outil commode et concis à votre disposition pour traiter les énumérations dans votre code.*

Définir une énumération

Pour notre première énumération, nous avons besoin de travailler avec des adresses IP. Pour le moment, il existe deux normes principales : **IPv4** ou **IPv6**. Ce seront les seules possibilités d'adresse IP que notre programme va rencontrer : nous pouvons énumérer toutes les variantes possibles, d'où vient le nom de l'énumération.

*N'importe quelle adresse **IP** peut être soit une adresse **IPv4**, soit une **IPv6**, mais pas les deux en même temps. Cette propriété des adresses **IP** est appropriée à la notion d'énumérations, car les valeurs de l'énumération ne peuvent être qu'une de ses variantes. Les adresses **IPv4** et **IPv6** sont toujours fondamentalement des adresses **IP**, donc elles doivent être traitées comme étant du **même type** lorsque le code traite des situations qui s'appliquent à n'importe quelle sorte d'adresse.*

Énumération d'adresse IP

```
enum TypeAdresse {
    IPv4,
    IPv6
}

fn main() {
    let _quatre = TypeAdresse::IPv4;
    let _six = Typedresse::IPv6;

    router(TypeAdresse::IPv4);
    router(TypeAdresse::IPv6);
}

fn router(_adresse: TypeAdresse) { }
```

*Nous remarquons que les deux variantes de l'énumération sont dans un espace de nom qui se situe avant leur nom, et nous utilisons un double deux-points « **::** » pour les séparer tous les deux. C'est utile car maintenant les deux valeurs **Typedresse::IPv4** et **TypeAdresse::IPv6** sont du même type : **TypeAdresse**. Ensuite, nous avons défini une fonction qui accepte n'importe quelle **TypeAdresse**.*

Mélanger énumération et structure

L'utilisation des énumérations possède beaucoup plus d'avantages. En étudiant un peu plus notre type d'adresse IP, nous constatons que pour le moment, nous ne pouvons pas stocker la donnée de l'adresse IP ; nous savons seulement de quelle sorte elle est. Avec ce que nous connaissons sur les structures, nous pouvons résoudre ce problème comme suit :

Énumération + Structure

```
fn main() {
    enum TypeAdresse {
        IPv4,
        IPv6
    }

    struct AdresseIP {
        sorte: TypeAdresse,
        adresse: String,
    }

    let local = AdresseIP {
        sorte: TypeAdresse::IPv4,
        adresse: String::from("127.0.0.1"),
    };

    let rebouclage = AdresseIP {
```

```
        sorte: TypeAdresse::IPv6,
        adresse: String::from("::1"),
    };
}
```

*Nous définissons une structure **AdresseIP** qui possède deux champs : un champ **sorte** qui est du type **TypeAdresse** (l'énumération que nous avons définie précédemment) et un champ **adresse** qui est du type **String**. Nous avons deux instances de cette structure. La première, **local**, a la valeur **TypeAdresse::IPv4** pour son champ **sorte**, associé à la donnée d'adresse qui est **127.0.0.1**. La seconde instance, **rebouclage**, a comme valeur de champ **sorte** l'autre variante **TypeAdresse ::IPv6**, et à l'adresse **::1** qui lui est associée. Nous avons utilisé une **structure** pour relier ensemble la sorte et l'adresse, donc maintenant la variante est liée à la valeur.*

Énumérations plus complexes

Nous pouvons appliquer le même principe de manière plus concise en utilisant uniquement une énumération, plutôt que d'utiliser une énumération dans une structure, en insérant directement la donnée dans chaque variante de l'énumération. Cette nouvelle définition de l'énumération **AdresseIP** indique que chacune des variantes **IPv4** et **IPv6** auront des valeurs associées de type **String** :

*Avec cette pratique qui fait la grande richesse de **Rust**, nous relions les données de chaque variante directement à **l'énumération**, donc il n'est pas nécessaire d'avoir une **structure** en plus.*

Énumération plus complexe

```
fn main() {
    enum AdresseIP {
        IPv4(String),
        IPv6(String)
    }

    let local = AdresseIP::IPv4(String::from("127.0.0.1"));
    let rebouclage = AdresseIP::IPv6(String::from("::1"));
}
```

Énumération avec des types différents

Il existe un autre avantage à utiliser une **énumération** plutôt qu'une **structure** : chaque variante peut stocker des types différents, et aussi avoir une quantité différente de données associées. Les adresses IP version quatre vont toujours avoir quatre composantes numériques qui auront une valeur entre **0** et **255**. Si nous voulions stocker les adresses **IPv4** avec quatre valeurs de type **u8** mais continuer à stocker les adresses **IPv6** dans une chaîne de type **String**, nous ne pourrions pas le faire avec une **structure**, alors que les énumérations permettent de le faire facilement.

Énumération avec des types différents

```
fn main() {
    enum AdresseIP {
        IPv4(u8, u8, u8, u8),
        IPv6(String)
    }

    let local = AdresseIP::IPv4(127, 0, 0, 1);
    let rebouclage = AdresseIP::IPv6(String::from("::1"));
}
```

*Dans le même ordre d'idée, je vous propose de fabriquer une énumération **Message** dont chaque variante stocke des valeurs de différents types et en différentes quantités. Cette énumération possède quatre variantes avec des types différents.*

- ***Quitter** ne possède aucune donnée associée.*
- ***Deplacer** intègre une structure anonyme..*
- ***Ecrire** intègre une chaîne de caractères.*
- ***ChangerCouleur** intègre trois valeurs de type **i32**.*

Énumération avec des variantes différentes

```
enum Message {
    Quitter,
    Deplacer { x: i32, y: i32 },
    Ecrire(String),
    ChangerCouleur(i32, i32, i32)
}
```

*Définir une énumération avec des variantes comme ci-dessus ressemble à la définition de différentes sortes de **structures**, sauf que **l'énumération** n'utilise pas le mot-clé **struct** et que toutes les variantes sont regroupées ensemble sous le type **Message**.*

Énumération avec des méthodes

Tout comme nous pouvons définir des méthodes sur les structures en utilisant **impl**, nous pouvons aussi définir des méthodes sur des énumérations. Voici la méthode ***appeler()*** définie sur notre énumération **Message** :

Méthode sur l'énumération

```
#[derive(Copy, Clone, Debug, PartialEq)]
enum Message {
    Quitter,
    Deplacer { x: i32, y: i32 },
    Ecrire(String),
    ChangerCouleur(i32, i32, i32)
}

impl Message {
    fn appeler(&self) {
        // Corps de la méthode
    }
}

fn main() {
    let m = Message::Ecrire(String::from("Bienvenue!"));
    m.appeler();
}
```

*Le corps de la méthode utilise **self** pour obtenir la valeur sur laquelle nous avons utilisé la méthode. Dans cet exemple, nous créons une variable **m** qui possède la valeur **Message::Ecrire(String::from("Bienvenue !"))**, et c'est cela que **self** retrouve comme valeur dans le corps de la méthode **appeler()** lorsque nous lançons **m.appeler()**.*

Nous retrouvons ici aussi, comme avec les structures, les traits par défaut qui permettent les affichages, les copies, les clonages, etc. grâce à l'attribut #[derive].

Énumération et structure de données complexes

Une énumération permet de constituer rapidement une structure de données arborescente. Prenons l'exemple d'un projet **Rust** qui doit manipuler des données **JSON**. Un document **JSON** est implanté en mémoire sous forme d'une valeur de type **Rust** suivant :

Énumération relative au format JSON

```
enum Json {
    Null,
    Boolean(bool),
    Number(f64),
    String(String),
    Array(Vec<Json>),
    Object(Box<HashMap<String, Json>>)
}
```

*Ce code **Rust** est suffisamment lisible pour ne réclamer aucune description en prose française. Le standard **JSON** prédéfinit les types de données qui peuvent apparaître dans un document **JSON** : nul, valeur booléenne, valeur numérique, chaîne, tableau de valeurs JSON et objet avec clé chaîne et valeur JSON. Notre énumération ne fait que stipuler ces différents types.*

Ce que nous pouvons remarquer est la facilité avec laquelle nous avons défini la structure de données au travers de cette énumération. En C++, il faudrait d'abord écrire une classe comme celle présentée ci-dessous :

Classe C++ équivalente pour exploiter pleinement le format JSON

```
class JSON {
    enum Tag {
        Null, Boolean, Number, String, Array, Object
    };
    union Data {
        bool boolean;
        double number;
        shared_ptr<string> str;
        shared_ptr<vector<JSON>> array;
        shared_ptr<unordered_map<string, JSON>> object;
        Data() {}
        ~Data() {}
        ...
    };
    Tag tag;
    Data data;
public:
    bool is_null() const { return tag == Null; }
    bool is_boolean() const { return tag == Boolean; }
    bool get_boolean() const {
        assert(is_boolean());
        return data.boolean;
    }
    void set_boolean(bool value) {
        this->~JSON(); // clean up string/array/object valuetag = Boolean;
```

```
        data.boolean = value;
    }
...
};
```

*Nous sommes déjà à trente lignes de code **C++**, et ce n'est pas terminé ! Il faut y ajouter des **constructeurs**, un **destructeur** et un **opérateur d'affectation**. Une autre approche serait de créer une hiérarchie de classes ; une classe de base **JSON** avec des sous classes **JSONBoolean**, **JSONString**, etc. Une fois la rédaction achevée, notre librairie **Json** en **C++** aura défini plus d'une douzaine de méthodes.*

L'énumération Option et ses avantages par rapport à la valeur null

Dans la section précédente, nous avons découvert comment l'énumération **AdresseIP** nous a permis d'utiliser le système de types de **Rust** pour enregistrer dans nos programmes encore plus d'informations qu'une simple donnée. Cette section étudie le cas de **Option**, qui est une autre énumération prédéfinie dans la bibliothèque standard.

*Le type **Option** est utilisé dans de nombreux cas de figure car il décrit un scénario très courant où une valeur peut être soit une valeur particulière, soit rien du tout. Exprimer ce concept avec le système de types implique que le compilateur puisse vérifier si nous avons géré tous les cas que nous pourrions rencontrer ; cette fonctionnalité permet d'éviter des bogues qui sont très courants dans d'autres langages de programmation.*

*La conception d'un langage de programmation est souvent pensée en fonction des fonctionnalités qui sont inclus, mais les fonctionnalités qui sont refusées sont elles aussi importantes. **Rust** n'a pas de fonctionnalité **null** que possèdent beaucoup de langages. **Null** est une valeur qui signifie qu'il n'y a pas de valeur à cet endroit. Avec les autres langages qui utilisent **null**, les variables peuvent toujours être dans deux états : soit **null** soit **non null**.*

*Cependant, le concept que **null** essaye d'exprimer reste utile : une valeur nulle est une valeur qui est actuellement invalide ou absente pour une raison ou une autre.*

*Le problème ne vient pas vraiment du concept, mais de son implémentation. C'est pourquoi **Rust** ne possède pas de valeurs nulles, mais il propose une **énumération** qui décrit le concept d'une valeur qui peut être soit présente, soit absente. Cette énumération est **Option<T>**, et elle est définie dans la bibliothèque standard comme ci-dessous :*

Énumération Option

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

*L'énumération **Option<T>** est tellement utile qu'elle est intégrée dans l'étape préliminaire ; vous n'avez pas besoin de l'importer explicitement dans la portée. De plus, voici ses variantes : vous pouvez utiliser directement **Some** (quelque chose) et **None** (rien) sans les préfixer par « **Option::** ». L'énumération **Option<T>** reste une énumération normale, et **Some(T)** ainsi que **None** sont toujours des variantes de type **Option<T>**.*

La syntaxe <T> est une fonctionnalité de Rust que nous n'avons pas encore abordée. Il s'agit d'un paramètre de type générique, et nous verrons la généricité plus en détail dans une autre étude. Pour le moment, dites-vous que ce <T> signifie que la variante Some de l'énumération Option peut stocker un élément de donnée de n'importe quel type.

Utilisation de l'énumération Option

```
fn main() {
    let nombre = Some(5);
    let chaine = Some("une chaîne");

    let sans_nombre: Option<i32> = None;
}
```

*Si nous utilisons **None** plutôt que **Some**, nous devons spécifier à **Rust** quel type de **Option<T>** nous avons choisi, car le compilateur ne peut pas déduire le type que cette variante **Some** va stocker en considérant uniquement une valeur **None**.*

*Lorsque nous avons une valeur dans **Some**, nous savons que la valeur est présente et que la valeur est stockée dans le **Some**. Lorsque nous avons une valeur **None**, en quelque sorte, cela veut dire la même chose que **null** : nous n'avons pas de valeur valide. Donc pourquoi obtenir **Option<T>** est meilleur que d'avoir **null** ?*

*En bref, comme **Option<T>** et **T** (où **T** représente n'importe quel type) sont de types différents, le compilateur ne nous autorise pas à utiliser une valeur **Option<T>** comme si cela était bien une valeur valide. Par exemple, le code suivant ne se compile pas car il essaye d'additionner un **i8** et une **Option<i8>** :*

Utilisation de l'énumération Option

```
fn main() {
    let x: i8 = 5;
    let y: Option<i8> = Some(5);

    let somme = x + y;
}
```

Résultat

```
error[E0277]: cannot add `Option<i8>` to `i8`
```

```
 --> src/main.rs:5:17
  |
5 |     let somme = x + y;
  |                 ^ no implementation for `i8 + Option<i8>`
  |
  = help: the trait `Add<Option<i8>>` is not implemented for `i8`
```

*Ce message d'erreur signifie que **Rust** ne comprend pas comment additionner un **i8** et une **Option<i8>**, car ils sont de types différents. Quand nous avons une valeur d'un type comme **i8** avec **Rust**, le compilateur va s'assurer que nous avons toujours une valeur valide.*

*Nous pouvons continuer en toute confiance sans avoir à vérifier que cette valeur n'est pas nulle avant de l'utiliser. Ce n'est que lorsque nous avons une **Option<i8>** (ou tout autre type de valeur avec lequel nous travaillons) que nous devons nous inquiéter de ne pas avoir de valeur, et le compilateur va s'assurer que nous gérons ce cas avant d'utiliser la valeur.*

*Autrement dit, vous devez convertir une **Option<T>** en **T** pour pouvoir faire avec elle des opérations du type **T**. Généralement, cela permet de résoudre l'un des problèmes les plus courants avec **null** : supposer qu'une valeur n'est pas nulle alors qu'en réalité, elle l'est.*

Ne pas avoir à s'inquiéter que des valeurs nulles puissent être mal gérées vous aide à être plus confiant en votre code. Pour avoir une valeur qui peut potentiellement être nulle, vous devez l'indiquer explicitement en déclarant que le type de cette valeur est Option<T>.

Ensuite, quand vous utiliserez cette valeur, il vous faudra gérer explicitement le cas où cette valeur est nulle. Si vous utilisez une valeur qui n'est pas une Option<T>, alors vous pouvez considérer que cette valeur ne sera jamais nulle sans prendre de risques. Il s'agit d'un choix de conception délibéré de Rust pour limiter l'omniprésence de null et augmenter la sécurité du code en Rust.

La structure de contrôle match associé aux motifs d'accès (pattern)

Je rappelle que **Rust** possède un opérateur de contrôle très puissant **match** qui nous permet de comparer une valeur avec une série de motifs et d'exécuter du code en fonction du motif correspondant. Les motifs peuvent être constitués de valeurs littérales, de noms de variables, de jokers, parmi tant d'autres.

*Considérez l'expression **match** comme une machine à trier les pièces de monnaie : les pièces descendent le long d'une piste avec des trous de tailles différentes, et chaque pièce tombe dans le premier trou à sa taille qu'elle rencontre. De manière similaire, les valeurs parcourent tous les motifs dans un **match**, et au premier motif auquel la valeur "correspond", la valeur va descendre dans le bloc de code correspondant afin d'être utilisée pendant son exécution.*

*Je vous propose un motif d'accès au travers d'un exemple. Dans certains projets, il est nécessaire d'afficher la date et l'heure de façon précise, à la milliseconde près, mais plus souvent encore, une valeur approximative suffit, d'autant qu'elle est plus ergonomique. Par exemple, nous pourrions ainsi vouloir afficher « **voici six mois** », où alors « **dans quinze jours** » ou tout simplement « **maintenant** ». Une énumération peut nous offrir ce confort de conception :*

Énumération temporelles

```
#[derive(Copy, Clone, Debug, PartialEq)]
enum UniteTemps {
    Secondes, Minutes, Heures, Jours, Mois, Annees
}

#[derive(Copy, Clone, Debug, PartialEq)]
enum TempsArrondi {
    DansLePasse(UniteTemps, u32),
    Maintenant,
    DansLeFutur(UniteTemps, u32)
}

fn main() {
    let voiciQuatreVingtSeptAns = TempsArrondi::DansLePasse(UniteTemps::Annees, 4*20+7);
    let dansTroisHeures = TempsArrondi::DansLeFutur(UniteTemps::Heures, 3);
}
```

*Deux des trois variantes de l'énumération **TempsArrondi** (**DansLePasse** et **DansLeFutur**) attendent des paramètres, et ce sont des variantes de **tuple**.*

*Supposons que nous voulions afficher la valeur d'un élément **TempsArrondi**. Il faut alors permettre l'accès aux éléments **UniteTemps** et **u32** de la valeur pour un affichage circonstancié, mais **Rust** vous interdit d'y accéder directement en écrivant par exemple **tempsArrondi.0** ou **tempsArrondi.1**., car la valeur pourrait tout aussi bien être à ce moment précis **TempsArrondi::Maintenant** qui lui ne possède pas de paramètre. Alors comment accéder aux données ?*

Méthodes associées aux méthodes temporelles avec la structure de contrôle match

```
#[derive(Copy, Clone, Debug, PartialEq)]
enum UniteTemps {
    Secondes, Minutes, Heures, Jours, Mois, Annees
}

impl UniteTemps {
    fn toStr(self) -> &'static str {
        match self {
            UniteTemps::Secondes => "secondes",
            UniteTemps::Minutes => "minutes",
```

```
            UniteTemps::Heures => "heures",
            UniteTemps::Jours => "jours",
            UniteTemps::Mois => "mois",
            UniteTemps::Annees => "années"
        }
    }
}

#[derive(Copy, Clone, Debug, PartialEq)]
enum TempsArrondi {
    DansLePasse(UniteTemps, u32),
    Maintenant,
    DansLeFutur(UniteTemps, u32)
}

impl TempsArrondi {
    fn affiche(self) -> String {
        match self {
            TempsArrondi::DansLePasse(unite, valeur) => format!("Il y a {} {}", valeur, unite.toStr()),
            TempsArrondi::Maintenant => format!("Juste maintenant"),
            TempsArrondi::DansLeFutur(unite, valeur) => format!("Dans {} {}", valeur, unite.toStr())
        }
    }
}

fn main() {
    let voiciQuatreVingtSeptAns = TempsArrondi::DansLePasse(UniteTemps::Annees, 4*20+7);
    let dansTroisHeures = TempsArrondi::DansLeFutur(UniteTemps::Heures, 3);
    let toutDeSuite = TempsArrondi::Maintenant;

    println!("({}) ({}) ({})", voiciQuatreVingtSeptAns.affiche(), dansTroisHeures.affiche(), toutDeSuite.affiche());
}
```

Résultat

```
(Il y a 87 années) (Dans 3 heures) (Juste maintenant)
```

*Un bloc **match** permet de faire la sélection par motif. Dans cet exemple, les motifs sont tout ce qui est avant le symbole « **=>** ». Ces motifs suivent la même syntaxe que pour créer une valeur de même type. Ce n'est pas un hasard puisqu'une **expression** produit une valeur alors qu'un **motif** consomme une valeur. C'est pourquoi la syntaxe est apparentée.*

***Rust** prend en compte les différentes branches des sélections par motif pour les énumérations, les structures et les tuples en travaillant de gauche à droite. Lorsqu'un cas ne convient pas, il passe au suivant.*

*Lorsque le **motif** contient des identifiants simples comme ici **unite** et **valeur**, ces derniers deviennent des **variables locales** dans le code qui suit le **motif**. Ce qui est trouvé dans la valeur est copiée ou transféré dans les nouvelles variables. C'est pourquoi par exemple **Rust** stocke **UniteTemps ::Heures** dans **unite**, et le nombre **3** dans **valeur** puis exécute la ligne associé au futur pour générer la chaîne « **Dans 3 heures** ».*

*Remarquez que la méthode **toStr()** de l'énumération **UniteTemps** renvoie une chaîne statique dont la durée de vie correspond à la durée de vie du programme dans son ensemble grâce au mot clé **static** (zone statique de la mémoire).*

Motifs de sélections

L'exemple précédent nous montre que la sélection par motif permet de bien travailler avec des énumérations en testant en profondeur le contenu, ce qui fait des blocs **match** un excellent remplacement aux instructions **switch** du **C++**. Les sélections par motif ne concernent pas que les valeurs d'énumération. Les motifs **Rust** constituent un véritable petit langage.

Type de motif	Exemple	Remarques
Littéral	**100** **« nom »**	Sélectionne la valeur exacte
Plage	**0 … 100** **'a' … 'z'**	Sélectionne n'importe quelle valeur dans la plage, borne de fin incluse
Métacaractère	_	Sélectionne n'importe quelle valeur puis l'ignore
Variable	**nom_var** **mut autre**	Comme _ mais transfère ou copie la valeur dans une nouvelle variable locale
Ref variable	**ref champ** **ref mut champ**	Emprunte une référence à la valeur sélectionnée au lieu de copier ou de transférer
Liaison avec sous- motif	**Val @ 0 ... 99** **ref cercle @ Forme::Cercle { .. }**	Sélectionne le motif à droite de @, avec le nom de variable à gauche
Motif **enum**	**Some(val)** **None** **TempsArrondi::Maintenant**	

Motif **tuple**	**(clé, valeur)** **(R, G, B)**	
Motif **struct**	**Couleur(r, g, b)** **Point {x, y}** **Carte { Couleur: Pique, valeur : As }** **Compte { nom, prenom, .. }**	
Référence	**&valeur** **&(k, v)**	Sélectionne des valeurs de **référence**
Motifs multiples	**'a' | 'A'**	Uniquement dans un bloc **match**
Expression multiple	**x if x*x <= r2**	Uniquement dans un bloc **match**

Littéraux, variables et génériques dans les motifs

Nous venons de voir comment écrire une expression **match** pour une **énumération**, mais bien sûr vous pouvez chercher dans d'autres types. Pour obtenir l'équivalent d'une instruction **switch** du **C++**, vous construisez un bloc **match** avec tout simplement une valeur entière. Les valeurs littérales **0** et **1** peuvent servir de motif.

Expression match avec des littéraux entiers

```
fn main() {
    let euro = 3;
    match euro {
        0 => {}
        1 => println!("J'ai 1 euro."),
        n => println!("J'ai {} euros.", n)
    }
}
```

Résultat

```
J'ai 3 euros.
```

*Dans cet exemple, le motif littéral **0** convient lorsque nous n'avons aucun euro, la valeur **1** lorsque nous avons juste un euro. Si nous avons au moins deux euros, nous arrivons au troisième motif, **n**, qui est le nom d'une variable très locale.*

*La variable évaluée dans l'expression **match** peut avoir n'importe quelle valeur, et celle-ci est transférée ou copiée dans une nouvelle variable locale. Ici, c'est donc la valeur de **euro** qui est copiée dans la variable locale **n** dont nous affichons la valeur. Dans cet exemple, nous pourrions nous passer de cette variable locale **n** puisque nous avons déjà la variable **euro**.*

Expression match avec des littéraux entiers

```
fn main() {
    let euro = 3;
    match euro {
        0 => {}
        1 => println!("J'ai 1 euro."),
        _ => println!("J'ai {} euros.", euro)
    }
}
```

Résultat

```
J'ai 3 euros.
```

*Le motif générique « _ » correspond à n'importe quelle valeur, mais ne stocke pas cette valeur, contrairement à ce que nous avions avec la variable locale **n**. Du fait que **Rust** oblige à prendre en charge toutes les valeurs possibles dans un **match**, le métacaractère générique est en général nécessaire pour clore ce bloc.*

Attention, même si vous êtes certain que les autres cas possibles ne peuvent pas se produire, il faut impérativement ajouter une branche générique qui puisse déclencher une panique.

Motifs tuple

Un **motif tuple** permet d'utiliser un **tuple** comme critère de sélection, et donc de comparer plusieurs paires de valeurs dans le même bloc **match**.

Motif tuple

```
fn point(x: i32, y: i32) -> &'static str {
    use std::cmp::Ordering::*;
    match (x.cmp(&0), y.cmp(&0))  {
        (Equal, Equal)       => "Point origine",
        (_, Equal)           => "Point sur l'axe des x",
        (Equal, _)           => "Point sur l'axe des y",
        (Greater, Greater) => "Point dans le premier quadrant",
        (Less, Greater)      => "Point dans le deuxième quadrant",
        _                    => "Point sur le troisième ou le quatrième quadrant"
    }
```

```
    }
}

fn main() {
    println!("{}", point(3, 4));
    println!("{}", point(0, 4));
    println!("{}", point(3, 0));
    println!("{}", point(0, 0));
    println!("{}", point(-3, -4));
}
```

Résultat

```
Point dans le premier quadrant
Point sur l'axe des y
Point sur l'axe des x
Point origine
Point sur le troisième ou le quatrième quadrant
```

*Pour bien comprendre cette expression **match**, il est nécessaire de préciser que **Ordering** est une énumération prédéfinie qui propose trois variantes **Less**, **Equal** et **Greater**. Il faut savoir aussi que tous les types dits primitifs possèdent comme toutes les autres types des méthodes qui permettent de réaliser des traitements spécifiques au type en vigueur.*

*Ici, les variables de type **i32** possèdent une méthode **cmp()** qui permet de comparer leurs valeurs respectives avec une autre passée en argument et qui renvoie la variante de comparaison de type **Ordering**. Attention, l'argument doit être une **référence** sur un littéral ou une variable du **même type** que la variable qui propose la comparaison.*

Motifs struct

Un **motif structure** utilise des accolades pour chacune des branches, en tant qu'expression **struct**. Nous avons un sous-motif pour chacun des champs de la structure.

Motif struct

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

impl Point {
    fn description(self) -> &'static str {
        match self {
            Point {x: 0, y: 0} => "Point origine",
            Point {x: 0, y}    => "Point sur l'axe des x",
            Point {x, y:0}     => "Point sur l'axe des y",
            Point {x, y}       => "Point sur aucun des axes"
        }
    }
}

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!("{}", p0.description());
    println!("{}", px.description());
    println!("{}", py.description());
    println!("{}", p.description());
}
```

Résultat

```
Point origine
Point sur l'axe des x
Point sur l'axe des y
Point sur aucun des axes
```

Vous devez exprimer tous les champs de la structure avec le nom exact de chaque champ. Si vous devez sélectionner une valeur précise pour un champ (ou plusieurs) vous précisez alors la valeur particulière. Si par contre la valeur du champ n'a pas d'importance pour la sélection, vous donnez juste le nom du champ sans valeur initiale.

Si vous le souhaitez, et pour que cela soit peut-être plus clair, vous pouvez aussi utiliser le métacaractère générique « _ » comme valeur particulière, afin de bien exprimer le fait que toute valeur est autorisée.

Implémentation de la méthode description()

```
impl Point {
    fn description(self) -> &'static str {
```

```
        match self {
            Point {x: 0, y: 0}  => "Point origine",
            Point {x: 0, y: _}  => "Point sur l'axe des x",
            Point {x: _, y: 0}  => "Point sur l'axe des y",
            Point {x: _, y: _}  => "Point sur aucun des axes"
        }
    }
}
```

Le résultat du programme principal est totalement identique à l'implémentation précédente. Pour certains cette syntaxe paraît plus claire et ne demande pas d'écrire beaucoup de code supplémentaire. Nous pouvons aussi utiliser la syntaxe de mise à jour de structure qui peut alléger encore plus le code.

Implémentation de la méthode description()

```
impl Point {
    fn description(self) -> &'static str {
        match self {
            Point {x: 0, y: 0}  => "Point origine",
            Point {x: 0, .. }   => "Point sur l'axe des x",
            Point {y: 0, .. }   => "Point sur l'axe des y",
            Point { .. }        => "Point sur aucun des axes"
        }
    }
}
```

*Nous pouvons aussi récupérer les valeurs des champs de la structure en utilisant les raccourcis d'initialisation. Par contre, si vous devez formater une chaîne, la méthode description doit proposer le retour d'un **String** plutôt que le retour d'un **&str** afin de mettre en œuvre le mécanisme **d'emprunt**. Ici, nous prévoyons que la méthode **description()** affiche directement le résultat.*

Motif struct

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

impl Point {
    fn description(self) {
        match self {
            Point {x: 0, y: 0}  => println!("Point origine"),
            Point {x: 0, .. }   => println!("Point sur l'axe des x"),
            Point {y: 0, .. }   => println!("Point sur l'axe des y"),
            Point {x, y}        => println!("Point({}, {}) sur aucun des axes", x, y)
        }
    }
}

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};

    p0.description();
    px.description();
    py.description();
    p.description();
}
```

Résultat

```
Point origine
Point sur l'axe des x
Point sur l'axe des y
Point(3, 5) sur aucun des axes
```

Motifs références

Les motifs de sélection **Rust** reconnaissent deux techniques pour travailler avec des références. Un motif « **ref** » emprunte une partie de la valeur sélectionnée alors qu'un motif « **& »** sélectionne une référence. N'oubliez pas que le fait de sélectionner une valeur non copiable, comme **String** par exemple, transfère la possession de la valeur.

*Dans le cas de valeur non copiable, il est souvent utile d'emprunter une valeur plutôt que de la transférer définitivement, C'est justement la raison d'être du mot-clé **ref**. En reprenant l'exemple précédent, voilà comment emprunter des points :*

Motif struct

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

```
    y:i32
}

impl Point {
    fn description(&self) {
        match self {
            Point {x: 0, y: 0}  => println!("Point origine"),
            Point {x: 0, .. }   => println!("Point sur l'axe des x"),
            Point {y: 0, .. }   => println!("Point sur l'axe des y"),
            Point {x, y}        => println!("Point({}, {}) sur aucun des axes", x, y)
        }
    }
}

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};
    let ref a = p;
    let b = &p;

    p0.description();
    px.description();
    py.description();
    p.description();
    a.description();
    b.description();
}
```

Résultat

```
Point origine
Point sur l'axe des x
Point sur l'axe des y
Point(3, 5) sur aucun des axes
Point(3, 5) sur aucun des axes
Point(3, 5) sur aucun des axes
```

*Essayons d'appliquer le principe de valeurs **non copiables** comme des chaînes de caractère avec une expression **match** qui propose des sélections dont le choix propose une chaîne dynamique de type **String** en paramètre.*

Sélection avec une chaîne String en paramètre

```
fn main() {
    let lui = Some(String::from("Albert"));
    match lui {
        Some(n)     => println!("Bonjour {}", n),
        None        => println!("Bonjour personne")
    }
    let anonyme : Option<String> = None;
    match anonyme {
        Some(n)     => println!("Bonjour {}", n),
        None        => println!("Bonjour personne")
    }
    println!("Bonjour {}", lui.unwrap());
}
```

Résultat

```
error[E0382]: use of partially moved value: `lui`
  --> src/main.rs:12:28
   |
4  |         Some(n) => println!("Bonjour {}", n),
   |              - value partially moved here
...
12 |     println!("Bonjour {}", lui.unwrap());
   |                             ^^^ value used here after partial move
   |
   = note: partial move occurs because value has type `String`, which does not implement the `Copy` trait
help: borrow this field in the pattern to avoid moving `lui.0`
   |
4  |         Some(ref n) => println!("Bonjour {}", n),
   |              ^^^
```

*Lorsque nous exécutons ce programme, une erreur est automatiquement activée. Le problème c'est que la chaîne correspondant à **lui** à été **transféré** lors de la sélection **Some(n)**. Du coup, lorsque nous tentons de nouveau d'utiliser **lui**, la chaîne correspondante n'est plus accessible.*

*Par contre, dans l'affichage de l'erreur, on nous propose la solution, c'est-à-dire de réaliser un **emprunt** de la chaîne de caractères au lieu du **transfert** proposer par défaut en utilisant le mot réservé **ref**.*

Sélection avec une chaîne String en paramètre

```
fn main() {
    let lui = Some(String::from("Albert"));
    match lui {
        Some(ref n) => println!("Bonjour {}", n),
        None => println!("Bonjour personne")
    }
    let anonyme : Option<String> = None;
    match anonyme {
        Some(ref n) => println!("Bonjour {}", n),
        None => println!("Bonjour personne")
    }
    println!("Bonjour {}", lui.unwrap());
}
```

Résultat

```
Bonjour Albert
Bonjour personne
Bonjour Albert
```

Bien entendu, Si vous fabriquer une fonction qui réalise cette sélection, il est plus judicieux de proposer une référence sur l'énumération afin que l'emprunt soit effectué en amont.

Sélection avec une chaîne String en paramètre au travers d'une fonction

```
fn son_nom(nom: &Option<String>) {
    match nom {
        Some(n) => println!("Bonjour {}", n),
        None => println!("Bonjour personne")
    }
}

fn main() {
    let lui = Some(String::from("Albert"));
    let anonyme : Option<String> = None;
    son_nom(&lui);
    son_nom(&anonyme);
    println!("Bonjour {}", lui.unwrap());
}
```

Résultat

```
Bonjour Albert
Bonjour personne
Bonjour Albert
```

Sélections multiples et plages de valeurs

Le symbole barre verticale « **|** » permet de combiner plusieurs motifs de sélection dans la même branche **match**. Il correspond à l'opérateur « **ou binaire** », mais fonctionne plus comme une expression régulière. Pour désigner toute une plage de valeurs, vous utilisez l'annotation « **...** » pour cette plage. Un motif de plage englobe les bornes de début et de fin.

Sélection multiples et plages de valeurs

```
fn analyse_phrase_ASCII(texte: &str) {
    let mut chiffres = 0;
    let mut lettres = 0;
    let mut espaces = 0;
    let mut ponctuations = 0;
    for caractere in texte.chars() {
        match caractere {
            '0'...'9'             => chiffres+=1,
            'a'...'z' | 'A'...'Z' => lettres+=1,
            ' ' | '\t' | '\n'     => espaces+=1,
            _                     => ponctuations+=1
        }
    }
    print!("\"{}\" possède ", texte);
    println!("{} chiffres, {} lettres, {} espaces et {} ponctuations", chiffres, lettres, espaces, ponctuations);
}

fn main() {
    let message = "Bonjour!\t 3.14 est le nombre PI, du moins\nune valeur arrondie.";
    analyse_phrase_ASCII(message);
}
```

Résultat

```
"Bonjour!          3.14 est le nombre PI, du moins
une valeur arrondie." possède 3 chiffres, 44 lettres, 11 espaces et 4 ponctuations
```

*Si nous consultons les messages d'alerte, il semble que l'annotation « **...** » est désuète et qu'il est préférable de prendre plutôt la syntaxe que nous connaissons déjà pour exprimer les plages de valeurs en intégrant la borne de fin « **..=** » :*

Fonction de la sélection multiples et plages de valeurs

```
fn analyse_phrase_ASCII(texte: &str) {
    let mut chiffres = 0;
    let mut lettres = 0;
    let mut espaces = 0;
    let mut ponctuations = 0;
    for caractere in texte.chars() {
        match caractere {
            '0'..='9'              => chiffres+=1,
            'a'..='z' | 'A'..='Z'  => lettres+=1,
            ' ' | '\t' | '\n'      => espaces+=1,
            _                      => ponctuations+=1
        }
    }
    print!("\"{}\" possède ", texte);
    println!("{} chiffres, {} lettres, {} espaces et {} ponctuations", chiffres, lettres, espaces, ponctuations);
}
```

Gardien de motif

Un **gardien de motif** est une condition **if** supplémentaire renseignée après le **motif** d'une branche d'un **match** qui doit elle aussi correspondre, de même que le filtrage par motif, pour que cette branche soit choisie. Les contrôles de correspondance sont utiles pour exprimer des idées plus complexes que celles permises par les motifs tout seuls.

*Attention, vous ne pouvez pas mettre en place de gardien si le motif effectue un transfert de valeur. Si le gardien n'est pas valide, **Rust** passe à la branche suivante ou au motif suivant. Mais c'est impossible si vous avez entre-temps dépossédé la valeur à comparer.*

Gardien de motif

```
fn evaluer(x:i32, y:i32) {
    match x {
        4 | 5 | 6 | 7 if y==17 => print!("Oui "),
        _ => print!("Non "),
    }
}

fn main() {
    evaluer(4, 17);
    evaluer(9, 17);
    evaluer(5, 15);
    evaluer(7, 17);
}
```

Résultat

```
Oui Non Non Oui
```

Motif @

L'opérateur @ nous permet de créer une variable qui stocke une valeur en même temps que nous testons cette valeur pour vérifier si elle correspond à un motif. La notation « **x @ motif** » procède à la même sélection qu'avec un motif simple, sauf qu'en cas de réussite, elle ne crée pas de variables pour les différentes parties de la valeur trouvée. Elle crée une seule variable nouvelle puis transfère ou copie toute la valeur de celle-ci.

Motif @

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

impl Point {
    fn description(&self) {
        match self {
            Point {x: 0, y: 0}                  => println!("Point origine"),
            Point {x: 0, .. }                   => println!("Point sur l'axe des x"),
            Point {y: 0, .. }                   => println!("Point sur l'axe des y"),
            Point {x : a @ 1...4, y: b @ 1...4 } => println!("Point proche de l'origine (x={}, y={})", a, b),
            Point {x, y}    => println!("Point éloigné de l'origine et sur aucun des axes (x={}, y={})", x, y)
        }
    }
}
```

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

    p0.description();
    px.description();
    py.description();
    loin.description();
    proche.description();
}
```

Résultat

```
Point origine
Point sur l'axe des x
Point sur l'axe des y
Point éloigné de l'origine et sur aucun des axes (x=3, y=5)
Point proche de l'origine (x=3, y=3)
```

*Ici nous utilisons le motif @ pour examiner des plages de valeurs pour les coordonnées **x** et **y** afin de déterminer si ces coordonnées sont proches de l'origine. Nous devons associer des variables temporaires **a** et **b** afin de pouvoir connaître les valeurs soumises et les afficher. Les champs **x** et **y** de la structure ne sont pas accessibles à ce moment là. Le motif @ devient donc indispensable pour ce genre d'analyse.*

La structure de contrôle concise : if let

La syntaxe **if let n**ous permet de combiner **if** et **let** afin de gérer les valeurs qui correspondent à un motif donné, tout en ignorant les autres. Imaginons le programme suivant dont l'expression est un **match** sur une valeur de type **Option<u8>** mais n'a besoin d'exécuter le code que si la valeur est **3**.

if let

```
fn main() {
    let valeur = Some(0u8);
    match valeur {
        Some(3) => println!("trois"),
        _ => (),
    }
}
```

*Nous désirons faire quelque chose avec la valeur **Some(3)** en ignorant les autres valeurs de type **Some<u8>** ou la valeur **None**. Pour satisfaire l'expression **match**, nous devons ajouter **_ => ()** après avoir géré une seule variante, ce qui fait beaucoup de code inutile. À la place, nous pourrions écrire le même programme de manière plus concise en utilisant **if let**.*

if let

```
fn main() {
    let valeur = Some(0u8);
    if let Some(3) = valeur {
        println!("trois");
    }
}
```

*La syntaxe **if let** prend un motif et une expression séparés par un signe égal. Elle fonctionne de la même manière qu'un **match** où l'expression est donnée au **match** et où le **motif** est sa première branche. Utiliser **if let** permet d'écrire moins de code, et de moins l'indenter. Cependant, vous perdez la vérification de l'exhaustivité qu'assure le **match**. Choisir entre **match** et **if let** dépend de la situation : à vous de choisir s'il vaut mieux être **concis** ou appliquer une **vérification exhaustive**.*

*Autrement dit, vous pouvez considérer le **if let** comme du sucre syntaxique pour un **match** qui exécute du code uniquement quand la valeur correspond à un motif donné et ignore toutes les autres valeurs.*