Lors de cette étude, je vous propose de voir comment gérer les flux d'entrée-sortie, la gestion des fichiers, des répertoires, des entrée-sortie standard, éventuellement du flux réseau, etc. Pour commencer, je vous propose de voir comment récupérer les arguments proposés en ligne de commande.

Récupérer les arguments en ligne de commande

Lorsque vous êtes en mode console, vous pouvez lancer vos programmes en tapant le nom de votre exécutable à la suite de l'invite du système, avec le préfixe « ./ ». Si vous procédez de cette façon, vous pouvez rajouter des arguments supplémentaires afin de prendre en compte des options qui rendent ainsi votre exécutable paramétrable.

L'exécutable suivi des arguments dans la ligne de commande est considérée comme une chaîne de caractères par le système d'exploitation, chaque élément de cette chaîne étant séparé des autres par au moins un caractère espace.

*Dans le code, pour récupérer les arguments de la ligne de commande, il existe une fonction spécialisée qui s'appelle tout simplement **args()** intégrée dans le module **env**. Voici un exemple complet pour calculer les factoriels des valeurs numériques proposées en ligne de commande.*

Récupération des arguments en ligne de commande

```
use std::env;

fn factoriel(n: u64) -> u64 {
  match n {
    0 | 1 => 1,
    _ => n * factoriel(n-1)
  }
}

fn main() {
  let arguments : Vec<String> = env::args().collect();
  match arguments.len() {
    1 => println!("Vous devez proposez vos arguments à la suite de la commande"),
    _ =>  for argument in &arguments[1..] {
      match argument.parse::<u64>() {
        Ok(x) =>  println!("{}! = {}", x, factoriel(x)),
        Err(_) => println!("{:?} argument non valide", argument)
      }
    }
  }
}
```

Résultat

```
manu@N550JV:~/CloudStation/ProjetRust/test-rust/target/debug$ ./test-rust 3 bon 5 7 11
3! = 6
"bon" argument non valide
5! = 120
7! = 5040
11! = 39916800
```

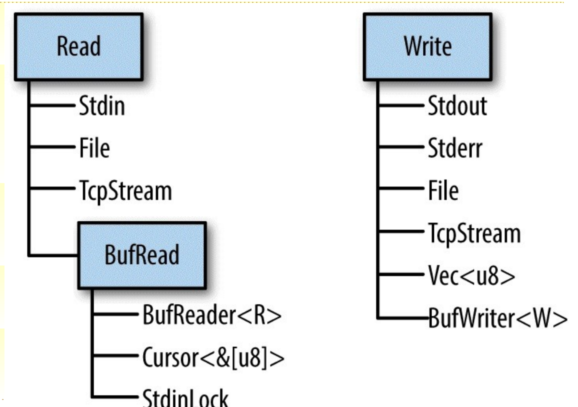
Architecture des entrées-sorties

Pour les opérations de lecture (**entrées**) et d'écritures (**sorties**) vers des fichiers, la librairie **Rust** standard propose les trois traits **Read**, **BufRead** et **Write**, accompagnés des différents types qui implémentent ces traits.

- *Les valeurs qui implémentent le trait **Read** disposent des méthodes pour les entrées de données au niveau octet. Ce sont des **lecteurs**.*
- *Les valeurs qui implémentent le trait **BufRead** sont des **lecteurs** avec **tampons mémoires**. Ils disposent de toutes les méthodes du trait **Read** et de quelques méthodes pour lire des lignes de texte et autres paquets de données.*
- *Les valeurs qui implémentent le trait **Write** supportent l'écriture au niveau octet et en tant que texte UTF-8. Ce sont des **scripteurs** (writers).*

Au cours de cette étude, nous allons découvrir comment utiliser les traits et les méthodes concernées, ainsi que les types qui les implémentent.

Nous verrons également comment interagir avec des fichiers, avec un terminal et avec le réseau.



Lecteurs et scripteurs

Un **lecteur** est une entité auprès de laquelle le programme peut lire des octets. En voici quelques exemples ci-dessous avec différents types d'infrastructure.

- *Un fichier ouvert avec **std::fs::File::open(nomFichier)**.*
- *Un flux **std::net::TcpStream** pour recevoir des données sur le réseau.*

- *Le flux d'entrée standard du processus en cours : **std::io::stdin()**.*
- *Un lecteur mémoire **std::io::Cursor<&[u8]>** qui permet de lire un tableau d'octets stocké en mémoire.*

Un **scripteur** est une entité permettant au programme d'envoyer des données vers l'extérieur. Là aussi, je vous propose un certain nombre d'exemples avec des contextes différents.

- *Un fichier ouvert avec **std::fs::**File::**create**(nomFichier).*
- *Un flux réseau **std::net::TcpStream**.*
- *La sortie standard du processus en cours : **std::io::stdout()** et la sortie d'erreur **std::io::stderr()** du terminal.*
- *L'entité **std::io::Cursor<&mut [u8]>** qui permet d'envoyer une tranche d'octets mutables en tant que fichier en sortie.*
- *L'entité **Vec<u8>** qui est un scripteur dont la méthode **write()** permet d'ajouter des données dans le vecteur.*

*Grâce aux traits standard de lecture et d'écriture **std::io::Read** et **std::io::Write**, il est souvent possible d'écrire du code générique capable de traiter des canaux d'entrée et/ou de sortie assez divers. Voici par exemple une fonction qui permet de copier tous les octets depuis un lecteur vers un scripteur.*

Fonction de copie d'octets

```
use std::io::{self, Read, Write, ErrorKind};
const BUFFER : usize = 8 * 1024;

fn copie<R: ?Sized, W: ?Sized> (lecture: &mut R, ecriture: &mut W) -> io::Result<u64>
  where R: Read, W: Write {
  let mut buf = [0; BUFFER];
  let mut ecrits = 0;
  loop {
    let lire = match lecture.read(&mut buf) {
      Ok(0) => return Ok(ecrits),
      Ok(lire) => lire,
      Err(ref e) if e.kind() == ErrorKind::Interrupted => continue,
      Err(e) => return Err(e)
    };
    ecriture.write_all(&buf[..lire])?;
    ecrits += lire as u64;
  }
}
```

*Ce code source est en fait l'implémentation de la fonction **std::io::copy()** de la librairie standard. Du fait qu'elle est générique, elle permet de copier des données depuis un fichier **File** vers un flux **TcpStream** depuis l'entrée standard **Stdin** vers un vecteur en mémoire **Vec<u8>**, cela parmi d'autres possibilités.*

*Les quatre traits **Read**, **BufRead**, **Write** et **Seek** servent tellement souvent qu'un module « **prelude** » a été créé pour définir seulement ces traits spécifiques :*

```
use std::io::prelude ::* ;
```

*Nous pouvons aussi prendre l'habitude dès à présent d'importer aussi le module **std::io** :*

```
use std::io::{self, Read, Write, ErrorKind} ;
```

*Le mot-clé **self** ci-dessus stipule que **io** est un alias du module **std::io**. Cela permet d'écrire ensuite **std::io::Result** et **std::io::Error** de façon abrégée, en **io::Result** et **io::Error**.*

Lecteur (readers)

Le trait **std::io::Read** dispose bien sûr de plusieurs méthodes de lecture de données. Toutes attendent le lecteur en entrée par référence mutable :

- ***reader.read(&mut tampon)** : lit plusieurs octets depuis la source et les stocke dans le tampon. Le type du tampon doit être **&mut [u8]**. Le nombre d'octets lu correspond à **tampon.len()**.*

  *La méthode renvoie une valeur de type **io::Result<u64>**, qui est un alias de **Result<u64, io::Error>**. En cas de réussite, la valeur indique le nombre d'octets lus, qui est nécessairement inférieur ou égal à **tampon.len()**, même si d'autres données vont venir, mais c'est au bon vouloir de la source. Une valeur **Ok(0)** signifie qu'il n'y a plus rien à lire.*

  *En cas d'erreur, la méthode renvoie **Err(err)**, la valeur **err** étant du type **io::Error**, et cette valeur peut être affichée pour la rendre lisible. La gestion des erreurs par un programme profitera de la méthode **kind()** qui renvoie un code d'erreur du type **io::ErrorKind**. Cette énumération réunit des champs portant des noms tel que **PermissionDenied** et **ConnectionRest**, la plupart correspondant à des erreurs qu'il ne faut pas ignorer.*

  *Un genre d'erreur doit recevoir une gestion particulière : il s'agit de **io::ErrorKind::Interrupted** qui correspond au code erreur **UNIX EINTR**, et qui signale que la lecture a été interrompue par un signal système. Normalement, il faut essayer à nouveau la lecture, à moins que le programme ait été conçu pour gérer ce genre de signal.*

  *La méthode **read()** travaille à bas niveau, puisqu'elle peut être troublée par le système d'exploitation. Vous avez ainsi une grande liberté si vous décidez d'implémenter ce trait **read** pour un nouveau type de donnée source. Mais si vous vous en servez seulement pour lire vite fait, bien fait des données, elle est fastidieuse.*

  *C'est pourquoi **Rust** propose quelques méthodes plus confortables à utiliser, qui se basent toutes sur l'implémentation par défaut de **read()**. Toutes se chargent de gérer **ErrorKind::Interrupted** à votre place.*

- ***reader.read_to_end(&mut byte_vec)*** *: lit ce qui reste à lire depuis le lecteur en l'ajoutant au vecteur* ***byte_vec*** *qui est de type* ***Vec<u8>****. Renvoie le nombre d'octets lus en tant que* ***io::Result<usize>****.*

  La méthode n'est à priori pas limitée en termes de volume de données à lire et à stocker dans ***byte_vec****. Ne l'utilisez pas avec une source inconnue. Vous pouvez toutefois ajouter une limite au moyen de la méthode* ***take()*** *présentée un peu plus bas.*
- ***reader.read_to_string(&mut chaine)*** *: ressemble à la précédente en ajoutant des données à la chaîne de type* ***String****. Si le flux ne contient pas de données* ***UTF-8*** *correctes, la méthode renvoie une erreur* ***ErrorKind::InvalidData****.*
- ***reader.read_exact(&mut tampon)*** *: lit exactement la quantité de données permettant de remplir le tampon spécifié, de type* ***&[u8]****. Si le lecteur ne trouve pas assez de données pour le remplir, avant d'avoir atteint la quantité* ***tampon.len()****, il renvoie une erreur* ***ErrorKind::UnexpectedEof****.*

Nous venons de voir les principales méthodes du trait ***Read****. Viennent s'y ajouter quatre méthodes adaptateurs qui utilisent le lecteur* ***reader*** *par valeur, ce qui le transforme en un itérateur ou un autre lecteur.*

- ***reader.bytes()*** *: renvoie un itérateur sur les octets du flux d'entrée. Le type de retour est* ***io::Result<u8>****, ce qui permet d'appliquer un test d'erreur pour chaque octet. Cette méthode appelle* ***reader.read()*** *pour chaque octet, ce qui est très peu efficace si le lecteur n'utilise pas un tampon.*
- ***reader.chars()*** *: ressemble à la précédente mais en progressant parmi les caractères, et en considérant l'entrée comme* ***UTF-8****. Si ce n'est pas de l'****UTF-8*** *valide, la méthode renvoie une erreur* ***InvalidData****.*
- ***reader.chain(reader2)*** *: renvoie un nouveau lecteur produisant toutes les données d'entrée provenant du lecteur* ***reader****, puis toutes celles de l'autre lecteur* ***reader2****.*
- ***reader.take(n)*** *: renvoie un nouveau lecteur qui prend ses données de la même source que* ***reader****, mais en s'arrêtant au bout de* ***n*** *octets.*

Vous ne trouverez pas de méthode pour fermer un lecteur, car aussi bien les lecteurs que les scripteurs implémentent normalement Drop, ce qui permet de les refermer automatiquement.

À titre d'exemple, je vous propose de réaliser un programme qui permet d'afficher à l'écran le contenu d'un fichier texte.

bienvenue.txt

```
Bonjour à tous!
Salut les copains.
```

Consultation d'un fichier texte

```rust
use std::fs::read_to_string;

fn main() {
  match read_to_string("bienvenue.txt") {
    Ok(texte) => println!("{}", texte),
    Err(_) => println!("Ce fichier n'existe pas")
  }
}
```

Résultat

```
Bonjour à tous!
Salut les copains.
```

Consultation d'un fichier texte

```rust
use std::fs::read_to_string;

fn main() {
  match read_to_string("Bienvenue.txt") {
    Ok(texte) => println!("{}", texte),
    Err(_) => println!("Ce fichier n'existe pas")
  }
}
```

Résultat

```
Ce fichier n'existe pas
```

Lecteurs tamponnés

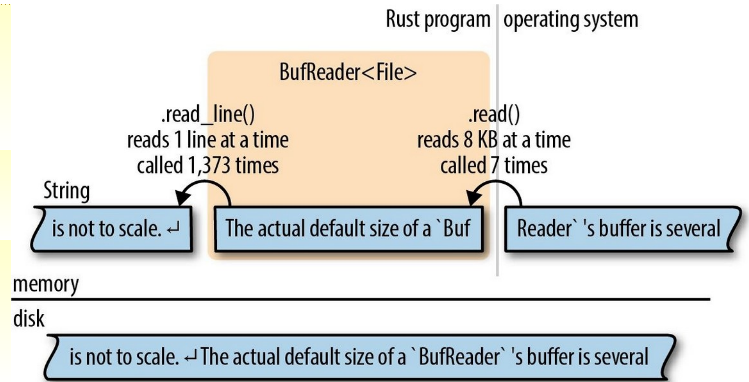
Un lecteur autant qu'un scripteur peut être tamponné afin d'être plus efficace, c'est-à-dire doté d'une zone mémoire dédiée au stockage temporaire des données en entrée ou en sortie. Ce tampon réduit le nombre d'appels système à réaliser, comme le montre la figure dans la page suivante.

L'application lit les données depuis le tampon ***BufReader****, en appelant la méthode* ***read_line()****. De l'autre côté, le tampon est automatiquement alimenté par le système d'exploitation par grands blocs où il intervient beaucoup plus rarement. L'illustration n'est pas à l'échelle. La taille réelle par défaut du tampon* ***BufReader*** *est de plusieurs kilooctets. Un seul appel à* ***read()*** *au système peut alimenter des centaines d'appels à* ***read_line()****. Les appels système sont généralement très lent. Un lecteur tamponné implémente en fait deux traits, le trait* ***Read*** *et le trait* ***BufRead*** *qui vient ajouter les quatre méthodes suivantes :*

- ***reader.read_line(&mut ligne)*** *: lit une ligne de texte puis l'ajoute à ligne qui est de type* ***String****. Le caractère* ***'\n'*** *est ajouté dans ligne. Si l'entrée est au format Windows, le couple de caractères* ***« \r\n »****, est inclus dans ligne.*

  La valeur renvoyée de type ***io::Result<usize>*** *indique le nombre d'octets lus, y compris la fin de ligne éventuelle.*

- ***reader.lines()*** *: renvoie un itérateur sur la ligne d'entrée. Le type est* ***io::Result<String>****, mais les caractères de saut de ligne ne sont pas ajoutés dans la chaîne. De même dans le format Windows, les deux caractères* ***« \r\n »*** *ne sont pas ajoutés.*



  C'est une méthode que vous utiliserez en général pour vos entrées de texte.

- ***reader.read_until(stop_byte, &mut byte_vec)*** *et* ***reader.split(stop_byte)*** *: sont similaires à* ***read_line()*** *et* ***lines()****, mais fonctionnent au niveau des octets. Elles produisent des valeurs* ***Vec<u8>*** *et non des* ***String****. C'est vous qui décidez du caractère délimiteur* ***stop_byte****.*

Le trait ***BufRead*** *propose également les méthodes de bas niveau* ***fill_buf()*** *et* ***consume(n)*** *qui permettent d'accéder directement au tampon interne du lecteur. Pour en savoir plus, vous consulterez la documentation en ligne.*

Saisie clavier

```
use std::io::stdin;

fn main() {
  let mut clavier= String::new();
  println!("Saisissez votre texte :");
  stdin().read_line(&mut clavier).unwrap();
  println!("Voici votre texte : {:?}", &clavier);
  println!("Voici votre texte sans saut de ligne : {:?}", clavier.trim());
}
```

Résultat

```
Saisissez votre texte :
Bonjour à tout le monde!
Voici votre texte : "Bonjour à tout le monde!\n"
Voici votre texte sans saut de ligne : "Bonjour à tout le monde!"
```

Lecture de plusieurs lignes

La fonction suivante reprend le fonctionnement de l'outil Unix nommé « **grep** » qui permet de chercher parmi de nombreuses lignes de texte. En général, l'outil est alimenté par la sortie d'une autre commande sous forme de chaîne :

Saisie clavier

```
use std::io;
use std::io::prelude::*;

fn grep(cible: &str) {
  let clavier = io::stdin();
  for ligne in clavier.lock().lines() {
    if let Ok(saisie) = ligne {
      if saisie.contains(cible) { break; }
      println!("Ligne saisie : {:?}", saisie);
    }
  }
}

fn main() {
  println!("Saisissez vos différentes lignes,");
  println!("Tapez 'fin' pour quitter la saisie.");
  grep("fin");
}
```

Résultat

```
Saisissez vos différentes lignes,
Tapez 'fin' pour quitter la saisie.
Bienvenue
Ligne saisie : "Bienvenue"
Bonjour
Ligne saisie : "Bonjour"
fin
```

*Puisque nous désirons appeler la méthode **lines()**, il faut que notre source de données implémente **BufRead**. Nous prenons ici **io::stdin()** pour récupérer les données qui nous sont transmises.*

*La librairie standard de Rust protège l'entrée **stdin** par un verrou **mutex**. Nous appelons donc **lock()** pour verrouiller **stdin**, afin que **l'exétron** actuel soit le seul à pouvoir s'en servir. En sortie de boucle, **StdinLock** est largué, ce qui libère le verrou **mutex**.*

Changeons notre fonction grep() de telle sorte qu'elle permette de rechercher un mot ou un texte dans un fichier texte plutôt que l'entrée standard.

bienvenue.txt

```
Bonjour à tous!
Salut les copains.
Bienvenue aux autres.
```

Saisie au clavier et lecture d'un fichier

```
use std::io::*;
use std::fs::*;

fn grep<R>(cible: &str, lecture: R) where R: BufRead {
  for ligne in lecture.lines() {
    if let Ok(saisie) = ligne {
      if saisie.contains(cible) {
        println!("Texte récupéré : {:?}", saisie);
        break;
      }
      println!("Texte ignoré : {:?}", saisie);
    }
  }
}

fn main() {
  println!("Saisissez vos différentes lignes,");
  println!("Utilisez le mot 'fin' pour quitter la saisie.");
  grep("fin", stdin().lock());
  if let Ok(fichier) =  File::open("bienvenue.txt") {
    let tampon = BufReader::new(fichier);
    grep("copain", tampon);
  }
}
```

Résultat

```
Saisissez vos différentes lignes,
Utilisez le mot 'fin' pour quitter la saisie.
Bienvenue à tout le monde
Texte ignoré : "Bienvenue à tout le monde"
C'est la fin de la saisie
Texte récupéré : "C'est la fin de la saisie"
Texte ignoré : "Bonjour à tous!"
Texte récupéré : "Salut les copains."
```

*Il faut savoir que **File** n'est pas automatiquement doté d'un tampon. Il implémente le trait **Read**, mais pas **BufRead**. Il est cependant facile d'obtenir un lecteur tamponné pour File ou pour tout autre lecteur qui n'en possède pas au départ.*

*C'est ce que nous faisons avec **BufRead::new(lecteur)**. Vous pouvez régler la taille du tampon en choisissant un autre appel **BufReader::with_capacity(taille, lecteur)**.*

Dans Rust, File et BufReader sont deux caractéristiques différentes de la librairie, ce qui permet de travailler parfois avec un fichier sans tampon, et parfois avec un tampon sans fichier, par exemple pour gérer des données arrivant depuis le réseau.

Collecte de plusieurs lignes

Plusieurs des méthodes de lecture, et notamment **lines()**, renvoient un itérateur qui produit des valeurs de type **Result**. Lorsqu'il s'agit de regrouper toutes les lignes d'un fichier dans un seul vecteur, vous pouvez faire face à des soucis pour gérer ces valeurs de type **Result**.

*La solution, comme pour toutes générations de collections est de faire appel à la méthode **collect()**, le tout étant de savoir comment définir le type que nous désirons exactement comme cela vous est montré ci-dessous.*

Création d'un vecteur de chaînes à partir d'un fichier

```
use std::io::*;
use std::fs::*;

fn main() {
  if let Ok(fichier) =  File::open("bienvenue.txt") {
    let tampon = BufReader::new(fichier);
    if let Ok(lignes) = tampon.lines().collect::<Result<Vec<String>>>() {
```

```
      for ligne in lignes {
        println!("{}", ligne);
      }
    }
  }
}
```

Résultat

```
Bonjour à tous!
Salut les copains.
Bienvenue aux autres.
```

Scripteur (writers)

Nous venons de voir que les entrées se basaient surtout sur des méthodes adaptées. Il en sera de même pour les sorties. Nous connaissons déjà un moyen d'afficher sur la sortie standard, nous nous sommes très souvent servi de la macro **println !()**. Nous allons découvrir comment exploiter toutes les autres sorties que Rust est capable de gérer.

*Rappelons au passage qu'il existe aussi la macro **print!()** qui est une variation sans ajout du saut de ligne final. Les options de formatage de ces deux macros sont les mêmes que celles de la macro **format!()** que nous connaissons déjà.*

*D'une façon plus générale, pour alimenter n'importe quel scripteur en données de sortie, nous pouvons utiliser les deux macros **write!()** et **writeln!()**. Elles se distinguent de **print!()** et de **println!()** que sur deux points.*

*La première des différences est que les macros d'écriture réclament toutes un premier paramètre qui identifie le scripteur. L'autre différence est qu'elle renvoient un e valeur de type **Result**, ce qui oblige à gérer les erreurs.*

Création d'un vecteur de chaînes à partir d'un fichier

```
use std::io::*;
use std::fs::*;

fn main() {
  if let Ok(fichier) =  File::open("bienvenue.txt") {
    let tampon = BufReader::new(fichier);
    let mut texte = Vec::new();
    if let Ok(lignes) = tampon.lines().collect::<Result<Vec<String>>>() {
      for ligne in lignes {
        writeln!(stdout(), "{}", ligne);
        writeln!(&mut texte, "{}", ligne);
      }
      if let Ok(chaines) = String::from_utf8(texte) {
        println!("{}", chaines);
      }
    }
  }
}
```

Résultat

```
Bonjour à tous!
Salut les copains.
Bienvenue aux autres.
Bonjour à tous!
Salut les copains.
Bienvenue aux autres.
```

- ***writer.write(&tampon)** : écrit tout ou une partie des octets de la tranche **tampon** dans le flux concerné et renvoie un **io::Result<usize>**. En cas de réussite, cela correspond au nombre d'octets écrit qui peut être inférieur à **tampon.len()**, en fonction des possibilités de flux. Comme sa collègue **Reader::read()**, vous ne devez pas utiliser directement cette méthode de bas niveau.*
- ***writer.write_all(&tampon)** : écrit tous les octets de la tranche **tampon**. Renvoie **Result<()>**.*
- ***writer.flush()** : purge toutes les données du tampon vers le flux et renvoie **Result<()>**.*

*Tout comme les lecteurs, les scripteurs sont automatiquement refermés lorsqu'ils sont largués. De même que **BufReader::new(lecteur)** ajoute un tampon à un lecteur, **BufWriter::new(scripteur)** ajoute un tampon à un scripteur. Vous pouvez également définir la taille du tampon avec **BufWriter::with_capacity(taille, scripteur)**.*

*Lorsqu'un **BufWriter** est largué, toutes les données restant dans le tampon sont envoyées pour écriture au scripteur. Si une erreur surgit pendant l'opération, cette erreur est ignorée. Il n'y a pas d'endroit convenable pour rapporter l'erreur puisque cela se produit dans la méthode **drop()**.*

*Pour garantir que votre application est informée de toutes les erreurs de sortie, procédez à une purge de tampon manuelle avec **flush()** avant de libérer ou larguer le scripteur.*

bienvenue.txt

```
Bonjour à tous!
Salut les copains.
```

```
Bienvenue aux autres.
```

Lecture et écriture de fichiers

```
use std::io::*;
use std::fs::*;

fn main() {
  if let Ok(ouverture) =  File::open("bienvenue.txt") {
    let tampon_lecture = BufReader::new(ouverture);
    if let Ok(sauvegarde) = File::create("bonjour.txt") {
      let mut tampon_sauvegarde = BufWriter::new(sauvegarde);
      if let Ok(lignes) = tampon_lecture.lines().collect::<Result<Vec<String>>>() {
        for ligne in lignes {
          writeln!(stdout(), "{}", ligne);
          writeln!(tampon_sauvegarde, "{}", ligne);
        }
      }
    }
  }
}
```

Résultat

```
Bonjour à tous!
Salut les copains.
Bienvenue aux autres.
```

bonjour.txt

```
Bonjour à tous!
Salut les copains.
Bienvenue aux autres.
```

L'objectif de ce programme est de lire le contenu d'un fichier pour en créer un autre en plaçant le même contenu avec pour intermédiaire des tampons de lecture et d'écriture. Il s'agit d'une simple copie de fichier. Il s'agit d'un cas d'école, les tampons ici ne sont absolument pas nécessaires.

Accès fichiers

Il existe deux moyens d'ouvrir un fichier. Précisons que le type File est défini dans le module du système de fichiers **std::fs** et non dans le module **std::io**.

*Plusieurs méthodes, notamment **append()**, **write(), read(), create()**, **create_new()**, et bien d'autres puisque chacune renvoie **self**. Ce patron de conception par enchaînement de chaînes est suffisamment répandu dans **Rust** pour porter un nom spécifique **« bâtisseur »** (builder).*

*Une fois que vous avez ouvert un fichier **File**, vous pouvez vous en servir comme tout autre lecteur ou scripteur et lui associer un tampon si nécessaire. Le fichier sera automatiquement refermé lorsque vous le larguerez.*

- ***File::open(nomFichier)** : ouvre un fichier existant en lecture et renvoie un **io::Result<File>** qui dénote une erreur si le fichier n'existe pas.*
- ***File::create(nomFichier)** : crée un fichier pour écriture. Si un fichier homonyme existe, son contenu est complètement tronqué.*

*Si aucun des deux modes d'ouverture ne convient, vous pouvez décider du comportement exact grâce à **OpenOptions**.*

Création d'un fichier de sauvegarde et ajout de texte saisie au clavier au fur et à mesure du lancement du programme

```
use std::io::*;
use std::fs::*;

fn main() {
  let clavier = stdin();
  let fichier = match OpenOptions::new()
                    .create(true) // Crée le fichier s'il n'existe pas
                    .append(true) // Ajoute le nouveau contenu en fin de fichier
                    .open("sauvegarde.txt") {
    Err(erreur) => panic!("{}", erreur),
    Ok(fichier) => fichier
  };
  let mut sauvegarde = BufWriter::new(fichier);
  for ligne in clavier.lock().lines() {
    if let Ok(saisie) = ligne {
      if saisie.contains("fin") { break; }
      writeln!(&mut sauvegarde, "{}", saisie);
    }
  }
}
```

Console 1ère fois

```
Bienvenue
fin
```

Console 2ème fois

```
Bonjour
fin
```

sauvegarde.txt

```
Bienvenue
Bonjour
```

Autres types lecteurs et scripteurs

Tout au long de cette étude, nous avons utilisé quelques types autres que **File** implémentant **Read** et **Write**. Le moment est venu de donner quelques détails à leur sujet.

- ***io::stdin()*** *: renvoie un lecteur pour le flux d'entrée standard du système. Le type correspondant est* ***io::Stdin****. Ce flux est partagé par tous les* ***exétrons*** *; chaque lecture suppose la pose et la libération d'un verrou* ***mutex****.*

  Stdin *est doté d'une méthode* ***lock()*** *qui obtient le verrou et renvoie* ***io::StdinLock****, un lecteur tamponné qui conserve le verrou* ***mutex*** *jusqu'à ce qu'il soit largué. Les opérations individuelles réalisées avec* ***StdinLock*** *n'imposent plus le surcoût de gestion du* ***mutex****.*

  Des raisons techniques empêchent ***io::stdin().lock()*** *de fonctionner. En effet, le verrou possède une référence sur la valeur* ***Stdin****, ce qui signifie que cette valeur doit être stockée de sorte à survivre suffisamment longtemps ;*
- ***io::stdout()*** *et* ***io::stderr()*** *: renvoient un scripteur vers la sortie standard et la sortie d'erreur standard. Les deux méthodes utilisent elles aussi des verrous* ***mutex*** *et des méthodes* ***lock()****.*
- ***Vec<u8>*** *: implémente le trait* ***Write*** *et le fait d'écrire vers un* ***Vec<u8>*** *ajoute les nouvelles données au vecteur. (****String*** *n'implémente pas* ***Write*** *et pour écrire une chaîne avec* ***Write****, il faut d'abord écrire vers un vecteur* ***Vec<u8>*** *puis utiliser* ***String::from_utf8(vec)*** *pour convertir le vecteur en chaîne.)*
- ***Cursor::new(tampon)*** *: crée un scripteur tamponné* ***Cursor*** *qui lit depuis* ***tampon****. C'est ce lecteur que vous utiliserez pour lire depuis une chaîne* ***String****. Le paramètre tampon doit être d'un type qui implémente* ***AsRef<[u8]>****, ce qui permet donc de fournir en entrée* ***&[u8], &st*** *ou un* ***Vec<u8>****.*

  Le lecteur ***Cursor*** *se résume à deux champs : le tampon et une valeur entière qui est le décalage dans le tampon de la position de la prochaine lecture. Au départ, la position est à* ***()****.*

  Les curseurs implémentent ***Read****,* ***BufRead*** *et* ***Seek****. Si le tampon est de type* ***&tampon [u8]*** *ou* ***Vec<u8>****, alors* ***Cursor*** *implémente aussi* ***Write****. Le fait d'écrire vers un curseur écrase les octets trouvés dans tampon à partir de la position courante. Si vous tentez d'écrire après la fin de* ***&mut [u8]****, vous obtenez une écriture partielle ou bien une erreur* ***io::Error****.*

  Dans le cas d'un ***Vec<u8>****, si vous tentez d'écrire après la fin, cela ne fait qu'agrandir le vecteur.* ***Cursor<&mut [u8]>*** *et* ***Cursor<Vec<u8>>*** *implémentent les quatre traits de* ***std::io::prelude****.*
- ***std::net::TcpStream*** *: incarne une connexion réseau* ***TCP****. Puisque* ***TCP*** *est bidirectionnel, il s'agit d'un lecteur et d'un scripteur. La méthode classique* ***TcpStream::connect((« hostname », PORT))*** *tente de vous connecter à un serveur pour obtenir en retour un* ***io::Result<TcpStream>****.*
- ***std::process::Command*** *: permet de lancer un processus enfant et d'alimenter son entrée standard en donnée par un canal «* ***pipe*** *».*

Le module ***std::io*** *offre enfin une série de fonctions pour disposer de certains lecteurs et scripteurs spécifiques :*

- ***io::sink()*** *: est le scripteur de drainage sans fond. Toutes ses méthodes d'écriture renvoient* ***Ok*** *et les données sont perdues.*
- ***io::empty()*** *: est le lecteur de drainage. Les lectures réussissent toujours, mais la fonction renvoie un signal de fin de données d'entrée.*
- ***io::repeat()*** *: renvoie un lecteur qui répète sans cesse le même octet.*

Dans ce premier exemple, je vous propose de réaliser une communication réseau en passant par une connexion par socket en utilisant un service et un protocole HTTP (sans passer par des structures spécifiques pour le protocole HTTP)

Connexion à mes cours en ligne, récupération du document HTML correspondant avec l'en-tête HTPP

```
use std::io::*;
use std::net::*;

fn main() {
  let mut service = match TcpStream::connect(("remy-manu.no-ip.biz", 80)) {
    Err(_) => panic!("Impossible de se connecter au service!"),
    Ok(service) => service
  };
  let mut html = String::new();
  service.write(b"GET HTTP/1.0 /\n\n").unwrap(); // Envoi d'octets (double validation pour le protocole HTTP)
```

```
  service.read_to_string(&mut html).unwrap();    // Sans prise en compte des erreurs éventuelles
  println!("{}", html);
}
```

Resultat

```
HTTP/1.1 400 Bad Request                                    // En-tête de la réponse du protocole HTTP
Server: nginx
Date: Sun, 15 Aug 2021 12:20:13 GMT
Content-Type: text/html
Content-Length: 150
Connection: close

<html>
<head><title>400 Bad Request</title></head>            // Document HTML
<body>
<center><h1>400 Bad Request</h1></center>
<hr><center>nginx</center>
</body>
</html>
```

*La méthode statique **connect()** de la structure **TcpStream** peut aussi s'utiliser de la façon suivante pour le même résultat :*

```
let mut service = match TcpStream::connect("remy-manu.no-ip.biz:80") {
```

Le deuxième exemple que je propose, permet de lancer une commande du système d'exploitation depuis votre programme et de récupérer le résultat dans un fichier spécifique.

Pour cela, nous utilisons le type Command avec les méthodes status() pour obtenir le résultat directement dans la console de sortie, et la méthode outpu() pour récupérer le flux de sortie pour l'associer à un fichier par exemple.

Lancer une commande du système d'exploitation

```
use std::process::Command;
use std::io::Write;
use std::fs::File;

fn main() {
  Command::new("ls").arg("-al").status().expect("ls non exécuté");
  let commande = Command::new("ls").arg("-al").output().expect("ls non exécuté");
  if let Ok(mut sauvegarde) = File::create("sauvegarde.txt") {
    sauvegarde.write_all(&commande.stdout).expect("Impossible d'écrire dans le fichier");
  }
}
```

Résultat

```
total 44
drwxr-xr-x  5 manu manu 4096 août  16 16:13 .
drwxr-xr-x  3 manu manu 4096 avril  6 13:43 ..
-rw-r--r--  1 manu manu  153 août   7 09:26 Cargo.lock
-rw-r--r--  1 manu manu  196 août   7 09:26 Cargo.toml
drwxrwxr-x 3 manu manu 4096 août  16 17:48 .idea
-rw-rw-r--  1 manu manu  589 août  16 17:47 sauvegarde.txt
drwxr-xr-x  2 manu manu 4096 août  16 17:51 src
drwxr-xr-x  3 manu manu 4096 avril  6 15:52 target
-rw-r--r--  1 manu manu  480 mars  18 10:48 test-rust.iml
```

sauvegarde.txt

```
total 44
drwxr-xr-x  5 manu manu 4096 août  16 16:13 .
drwxr-xr-x  3 manu manu 4096 avril  6 13:43 ..
-rw-r--r--  1 manu manu  153 août   7 09:26 Cargo.lock
-rw-r--r--  1 manu manu  196 août   7 09:26 Cargo.toml
drwxrwxr-x 3 manu manu 4096 août  16 17:48 .idea
-rw-rw-r--  1 manu manu  589 août  16 17:47 sauvegarde.txt
drwxr-xr-x  2 manu manu 4096 août  16 17:51 src
drwxr-xr-x  3 manu manu 4096 avril  6 15:52 target
-rw-r--r--  1 manu manu  480 mars  18 10:48 test-rust.iml
```

Fichiers et répertoires

Découvrons maintenant comment **Rust** propose de manipuler les fichiers et répertoires avec des fonctions qui correspondent aux deux modules **std::path** et **std::fs**. Puisque toutes ces caractéristiques ont recours à des noms de fichiers, nous allons commencer par découvrir les types de noms de fichiers.

OsStr et Path

Les versions actuelles des systèmes d'exploitation n'imposent hélas pas encore que les noms de fichiers soient toujours codés en **Unicode** bien formé. Les chaînes **Rust** contiennent obligatoirement du code **Unicode** valide. Dans la pratique,

les noms des fichiers sont presque toujours **Unicode**, mais **Rust** doit gérer les cas rare d'une chaîne qui n'est pas conforme et c'est pour cette raison que **Rust** définit les deux types **std::ffi::OsStr** et **OsString**.

*Les chaînes **Rust** contiennent obligatoirement du code **Unicode** valide. Dans la pratique, les noms de fichiers sont presque toujours en **Unicode**, mais **Rust** doit gérer les cas rare d'une chaîne qui n'est pas conforme et c'est pour cette raison que **Rust** définit les deux types **std::ffi::OsStr** et **OsString**.*

Le type OsStr est un sur-ensemble d'UTF-8 dont le but est de pouvoir incarner n'importe quel nom de fichier, paramètre de ligne de commande ou variable d'environnement du système d'exploitation concerné, que ce soit du code Unicode ou pas. Sous Unix, OsStr peut contenir n'importe quelle séquence d'octets. Sous Windows, il peut contenir une extension d'UTF-8 correspondant à n'importe quelle séquence de valeurs de 16 bits, y compris les paires de substitution sans correspondance.

*Nous disposons ainsi de deux types chaînes : **str** pour les vrais chaînes Unicode et **OsStr** pour tout ce que le système d'exploitation pourrait produire sans garantie de qualité. Nous y ajoutons le type **std::path::Path** pour les noms de fichiers, pour plus de confort.*

*En effet **Path** est exactement identique à **OsStr** mais il dispose d'un bon nombre de méthodes bien pratiques pour les noms de fichiers. Vous vous servirez de **Path** pour les chemins d'accès absolus et relatifs. Vous vous servirez de **OsStr** pour les éléments individuels qui constituent un chemin d'accès.*

Pour chacun de ces types chaînes, vous disposez d'un type possédant correspondant : le type String possède une chaîne str sur le tas, le type std::ffi::OsString possède une chaîne OsStr sur le tas et std::path::PathBuf possède un Path sur le tas.

Possibilté des types miés aux noms de fichiers et chemins

	Str	OsStr	Path
Type sans taille, toujours transmis par référence	Oui	Oui	Oui
Accepte n'importe quel texte Unicode	Oui	Oui	Oui
Correspond normalement à du UTF-8	Oui	Oui	Oui
Peut contenir des données non Unicode	Non	Oui	Oui
Méthode de traitement de texte	Oui	Non	Non
Méthode de gestion de nom de fichier	Non	Non	Oui
Équivalent possédé, retaillable et réservé sur le tas	String	OsString	PathBuf
Conversion vers type possédé	to_string()	to_os_string()	to_path_buf()

*Ces trois types implémentent le même trait **AsRef<Path>**, ce qui permet de déclarer très facilement une fonction générique pouvant accepter n'importe quel type de nom de fichier en paramètre. Toutes les fonctions et méthodes qui attendent en entrée un paramètre de type **Path** se servent de cette technique et vous pouvez sans problème leur transmettre un littéral chaîne.*

Méthodes de Path et PathBuf

Le trait de gestion des chemins d'accès **Path** propose notamment les méthodes suivantes. Toutes ces méthodes traitent des chaînes stockées en mémoire. Il est également possible d'utiliser **Path** par interrogation directe du système de fichiers avec **exists()**, **is_file()**, **is_dir()**, **read_dir()**, **canonicalize()** et quelques autres.

- ***Path::new(str)** : convertie une chaîne **&str** ou **&OsStr** vers **&Path** sans copier le contenu de la chaîne. Le nouvel élément **&Path** pointe sur les mêmes octets que l'original **&str** ou **&OsStr**. (La méthode apparentée **OsStr::new(str)** convertit une **&str** vers une **&OsStr**).*
- ***path.parent()** : renvoie s'il existe le nom du répertoire parent du chemin avec une valeur renvoyée du type **Option<&Path>**. Le texte du chemin d'accès n'est pas copié et le répertoire parent de **path** reste une sous-chaîne du chemin **path**.*
- ***path.file_name()** : renvoie le nom du fichier, c'est-à-dire le dernier composant du chemin **path** s'il existe. Le type renvoyé est **Option<&OsStr>**. En général, lorsque le chemin d'accès **path** comporte un nom de répertoire puis une barre oblique et enfin le nom de fichier, la méthode renvoie le nom du fichier.*
- ***path.is_absolute()** et **path.is_relative()** : permettent de savoir si le chemin d'accès au fichier est absolu comme dans le chemin Unix **« /usr/bin/advent »**, ou s'il est relatif, comme dans **« src/main.rs »**.*
- ***path1.join(chem2)** : permet de fusionner deux chemins en renvoyant un nouveau chemin **PathBuf**. Si le **chem2** est un chemin absolu, la méthode renvoie une copie de **chem2**. Elle permet donc de convertir un chemin vers un chemin absolu.*
- ***path.components()** : renvoie un itérateur sur les différents composants du chemin, de gauche à droite. Le type de l'itérateur est **std::path::Component**, qui correspond à une énumération permettant d'incarner les différents composants qui constituent un nom de fichier complet.*

```
pub enum Component<'a> {
  Prefix(PrefixComponent<'a>),  // Windows seul : lettre lecteur ou de partage
  RootDir,                      // Racine, '/' ou '\'
  CurDir,                // Entrée spéciale '.' (répertoire courant)
  ParentDir,             // Entrée spéciale '..' (répertoire parent)
  Normal(&'a OsStr)      // Nom de fichier ou de répertoire
}
```

*Vous disposez également de trois méthode supplémentaires pour convertir un élément **Path** en une chaîne. Aucune des trois ne protège contre l'apparition de code **UTF-8** invalide dans **Path**.*

- ***path.to_string()** : convertit un **Path** en une chaîne de type **Option<&str>**. Si le chemin n'est pas du code **UTF-8** valide renvoie **None**.*
- ***path.to_string_lossy()** : s'apparente à la précédente, sauf qu'elle cherche à renvoyer une chaîne dans tous les cas. Si le chemin ne contient pas du code **UTF-8** valide, la méthode réalise une copie en remplaçant chaque séquence d'octets invalide par le caractère de remplacement Unicode **U+FFFD**.*

  *Le type de valeur renvoyée est **std::borrow::Com<str>** qui est une chaîne soit empruntée, soit possédée. Pour obtenir une chaîne **String** à partir de cette valeur, vous vous servez de la méthode **to_owned()**.*
- ***path.display()** : sert à afficher un chemin d'accès. La méthode ne renvoie pas une chaîne, mais elle implémente le trait **std::fmt::Display**, ce qui permet de l'utiliser avec **format!()**, **println!()** et d'autres macros. Si le chemin ne contient pas du code **UTF-8** valide, il est possible que le résultat affiché comporte le caractère de remplacement, le fameux **U+FFFD**.*

Méthodes associées aux répertoires

```
use std::path::Path;

fn main() {
  let home = Path::new("/home/manu");
  println!("{:?}", home.parent());
  println!("{:?}", home.file_name());
  println!("{:?}", home.is_absolute());
  println!("{:?}", home.is_relative());
  println!("{:?}", home.components());
  println!("{:?}", home.to_str());
  println!("{:?}", home.to_string_lossy());
  println!("{:?}", home.display());
}
```

Résultat

```
Some("/home")
Some("manu")
true
false
Components([RootDir, Normal("home"), Normal("manu")])
Some("/home/manu")
"/home/manu"
"/home/manu"
```

Fonctions d'accès au système de fichiers

Le tableau suivant présente les principales fonctions définies dans **std::fs** avec les équivalents approchés sous Unix et sous Windows. Toutes les fonctions renvoient une valeur de type **io::Result**. Si rien n'est précisé dans la colonne de la fonction **Rust**, le type est **Result<()>**.

Actions	Fonction Rust	Unix	Windows
Création et suppression	create_dir(chemin)	mkdir	CreateDirectory
	create_dir_all(chemin)	mkdir -p	mkdir
	remove_dir(chemin)	rmdir	RemoveDirectory
	remove_dir_all(chemin)	rm -r	rmdir /s
	remove_file(chemin)	unlink, delete	DeleteFile
Copie Déplacement et liaison	copy(chmsrc, chmdest) → Result<u64>	cp -p	CopyFileEx
	rename(chmsrc, chmdest)	rename	MoveFileEx
	hard_link(chmsrc, chmdest)	link	CreateHardLink
Inspection	canonicalize(chemin) → Result<PathBuf>	realpath	GetFinalPathNameByHendle
	metadata(chemin) → Result<MetaData>	stat	GetFileInformationByHandle
	symlink_metadata(chemin) → Result<MetaData>	lstat	
	read_dir(chemin) → Result<ReadDir>	opendir	FindFirstFile
	read_link(chemin) → Result<PathBuf>	readlink	FSCTL_GET_REPARSE_POINT
Gestion des permissions	set_permission(chemin, perm)	chmod	SetFileAttributes

*La fonction **copy()** renvoie la taille en octets du fichier qui a été copié. Cette liste permet de voir que **Rust** a cherché à proposer des fonctions portables qui opèrent de façon prévisible sur les différents systèmes **Windows**, **MacOs**, **Linux** et autres **Unix**.*

*Toutes ces fonctions sont implémentées par un appel au système d'exploitation. C'est ainsi que la fonction **std::fs::canonicalize(chemin)** n'applique pas un traitement de chaîne pour supprimer les deux entrées **« . »** et **« .. »** dans le chemin. Elle se sert du chemin relatif à partir du répertoire courant puis recherche les liens symboliques. Notez que si le chemin n'existe pas, cela provoque une erreur.*

*Notez que le type **Metadata** qui est produit par les deux méthodes **std::fs::metada(chemin)** et **std::fs::symlink_metadata(chemin)** contient par exemple le type de fichier ainsi que sa taille, ses droits d'accès et son horodateur.*

*Le type **std::fs::DirEntry** est une structure dotée de quelques méthode utiles :*

- ***entry.file_name()** : renvoie le nom du fichier ou du répertoire de type **OsString**.*
- ***entry.path()** : renvoie la même chose en y ajoutant le chemin de départ, ce qui produit un nouveau **PathBuf**. Par exemple, Si nous étions en train d'interroger le répertoire « **/home/manu** » et si **entry.file_name()** valait **« bonjour.txt »** alors **entry.path()** renverrait **PathBuf::from(« /home/manu/bonjour.txt »**.*

  *Le type de valeur renvoyée est **std::borrow::Com<str>** qui est une chaîne soit empruntée, soit possédée. Pour obtenir une chaîne **String** à partir de cette valeur, vous vous servez de la méthode **to_owned()**.*
- ***entry.file_type()** : renvoie un type **io::Result<FileType>**. Le type **FileType** dispose des méthodes **is_file()**, **is_dir()** et **is_symlink()**.*
- ***entry.metadata()** : récupère le reste des métadonnées au sujet de l'entrée.*

*Sachez que les deux entrées spéciales **« . »** et **« .. »** ne font pas partie du résultat lors de la lecture du contenu d'un répertoire.*

Méthodes associées aux répertoires

```
use std::path::Path;
use std::fs::*;

fn main() {
  create_dir("./nouveau").unwrap();
  hard_link("bienvenue.txt", "bonjour.txt");
  if let Ok(fichiers) = Path::new(".").read_dir() {
    for liste in fichiers {
      if let Ok(fichier) = liste {
        let nom = fichier.file_name().to_string_lossy().to_string();
        let donnees = fichier.metadata().unwrap();
        if let Ok(type_fichier) = fichier.file_type() {
          if type_fichier.is_dir()  { println!("(Répertoire) {:15} {}", &nom, &donnees.len()); }
          else                     { println!("(Fichier)    {:15} {:?}", &nom, &donnees.len()); }
        }
      }
    }
  }
  remove_dir_all("./nouveau").unwrap();
  remove_file("bonjour.txt").unwrap();
}
```

Résultat

```
(Répertoire)  target          4096
(Fichier)     Cargo.toml      196
(Répertoire)  .idea           4096
(Fichier)     Cargo.lock      153
(Fichier)     test-rust.iml   480
(Répertoire)  nouveau         4096
(Fichier)     bienvenue.txt   58
(Fichier)     bonjour.txt     58
(Répertoire)  src             4096
```