

Sécurité applicative

partie 3 - bas niveau

présentation diffusée sous licence CC-BY-ND (nous citer @fimbault)

vulnérabilités classiques

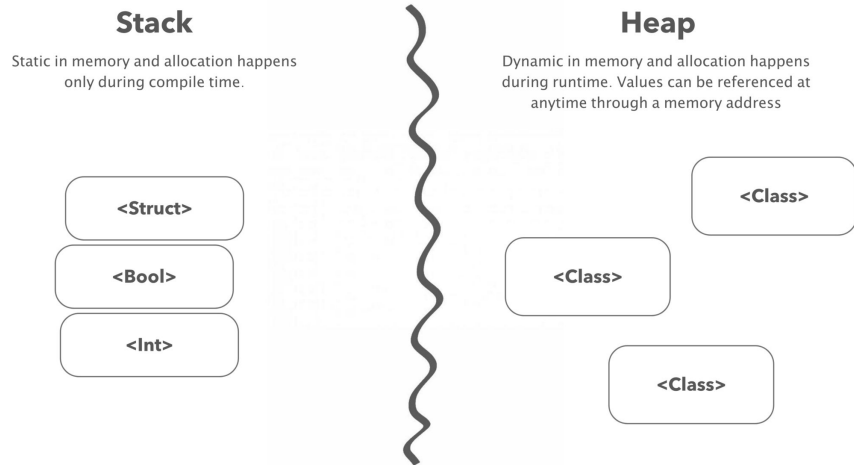
Rappel : vulnérabilité dans OpenSSL

- La sécurité black-box ne suffit pas :
 - Le problème n'est pas au niveau des appels système ou d'un dépôt de fichiers: ni le système d'exploitation ni l'antivirus ne peuvent bloquer une attaque
 - Les paquets suspects pourraient être bloqués par un IDS, mais le “packet chunking” peut contourner les règles de filtrage et la donnée sortie par le hacker est dans un flux crypté, invisible pour les outils d'analyse
- Il faut résoudre la problème à la source : le code (dans ce cas, C++)

Vulnérabilités bas niveau

Exemples de vulnérabilités en lien avec la gestion de la mémoire (les fameux pointeurs, cf vos cours de C/C++) :

- buffer overflows
- Format string
- Désallocation de pointeur



Mémoire et instructions

— — —

<https://beta.hackndo.com/stack-introduction/>

<https://www.youtube.com/watch?v=akCce7vSSfw&list=PLhixgUqwRTjxglIswKp9mpkfPNfHkzyeN&index=35>



Buffer overflow

Buffer = plage contigüe d'adresses associée à une variable

en C : les strings sont des tableaux de char (terminés par NUL)

user strings : text input, packets, env variable, file input, etc.

Overflow = accès d'un buffer en dehors de ses limites attribuées

- over-read / over-write, avant ou après le buffer
- à cause d'une boucle ("running of the end") ou par accès direct (arithmétique de pointeur)

strcpy

— — —

```
void func(char *arg1)
{
    int authenticated = 0;

    char buffer[4];

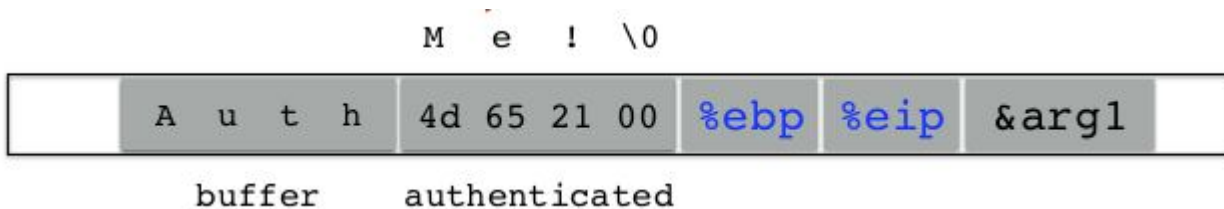
    strcpy(buffer, arg1);

    if(authenticated) { ... }
}
```

```
int main()
{
    char *mystr = "AuthMe!";

    func(mystr);

    ...
}
```



Code injection

On cherche à charger des instructions en code machine (assembly language, compilé pour l'architecture cible, ex: x86_64), sous forme d'un prompt en ligne de commande (le shell)

Note : ne peut pas contenir un octet à 0, sinon sprintf/strcpy/etc. arrête de copier

Le code pour lancer un shell s'appelle le shellcode

Shellcode

— — —

```
#include <stdio.h>
```

```
int main( ) {
```

```
char *name[2];
```

```
name[0] = "/bin/sh";
```

```
name[1] = NULL;
```

```
execve(name[0], name, NULL);
```

```
}
```

Assembly

```
xorl %eax, %eax
```

```
pushl %eax
```

```
pushl $0x68732f2f
```

```
pushl $0x6e69622f
```

```
movl %esp, %ebx
```

```
pushl %eax
```

```
...
```

Machine code

```
"\x31\xc0"
```

```
"\x50"
```

```
"\x68""//sh"
```

```
"\x68""/bin"
```

```
"\x89\xe3"
```

```
"\x50"
```

```
...
```

Challenge : lancer le code

Le challenge est ensuite de lancer ce code :

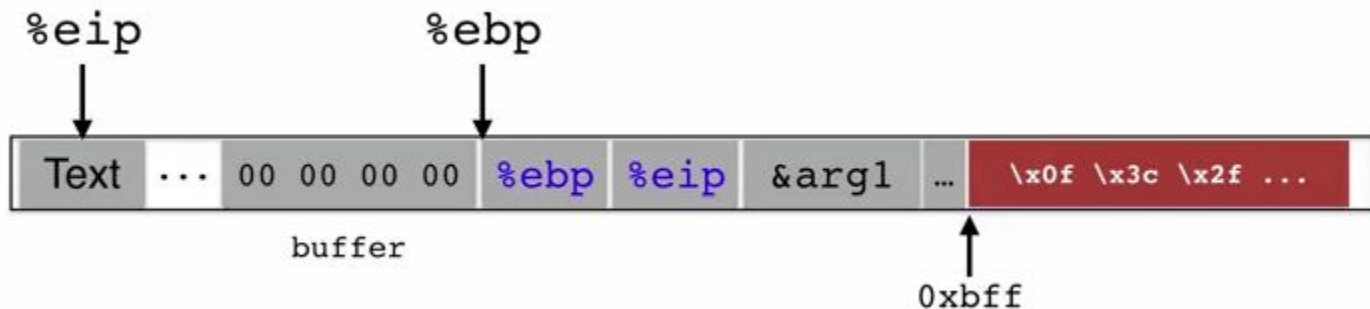
- on ne peut pas insérer une instruction “jump into my code”
- on ne sait pas précisément où est notre code par rapport à l'eip (instruction pointer)

Exemple: <https://www.youtube.com/watch?v=oS2O75H57qU&list=PLhixgUqwRTjxgllswKp9mpkfPNfHkzyeN&index=51>

(le hacker explique comment il procède en pratique pour résoudre ces difficultés)

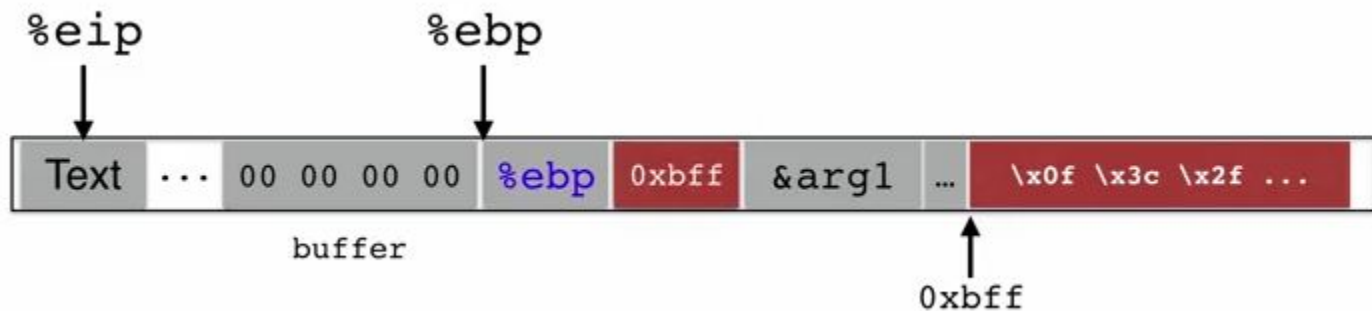
Comment positionner le code au niveau de %eip

L'objectif : charger l'adresse du code (ici 0xbff) dans la valeur de %eip



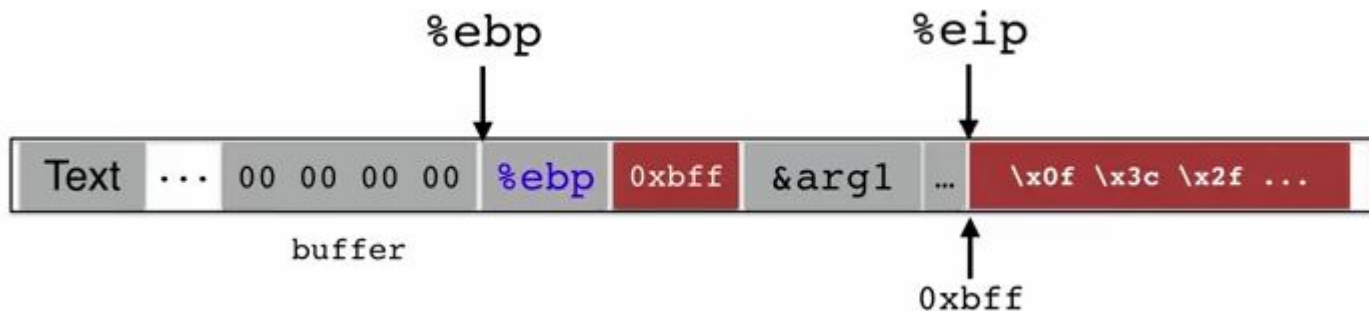
Comment positionner le code au niveau de %eip

L'adresse cible 0xbff est sauvé dans la stack



Comment positionner le code au niveau de %eip

Quand la fonction returns, on pointe vers %eip



Quelle adresse indiquer ?

Comment sait-on que l'adresse est 0xbff ?

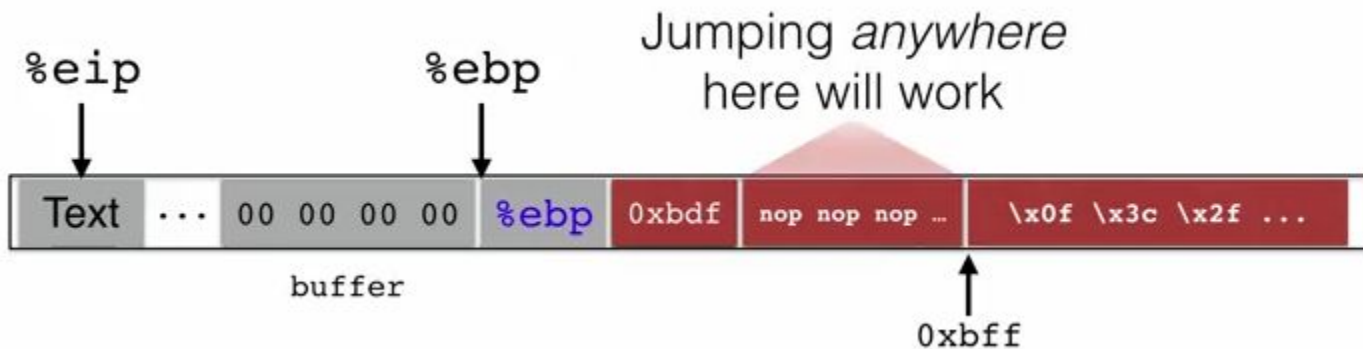
- Si l'attaquant a accès au code, il peut savoir à un moment précis où aller
- Sinon : possible de faire des essais, mais l'espace peut être très large (jusqu'à 2^{64})

Si on se trompe d'adresse :

- jump vers une instruction invalide (non compréhensible)
- crash le programme (CPU panic)

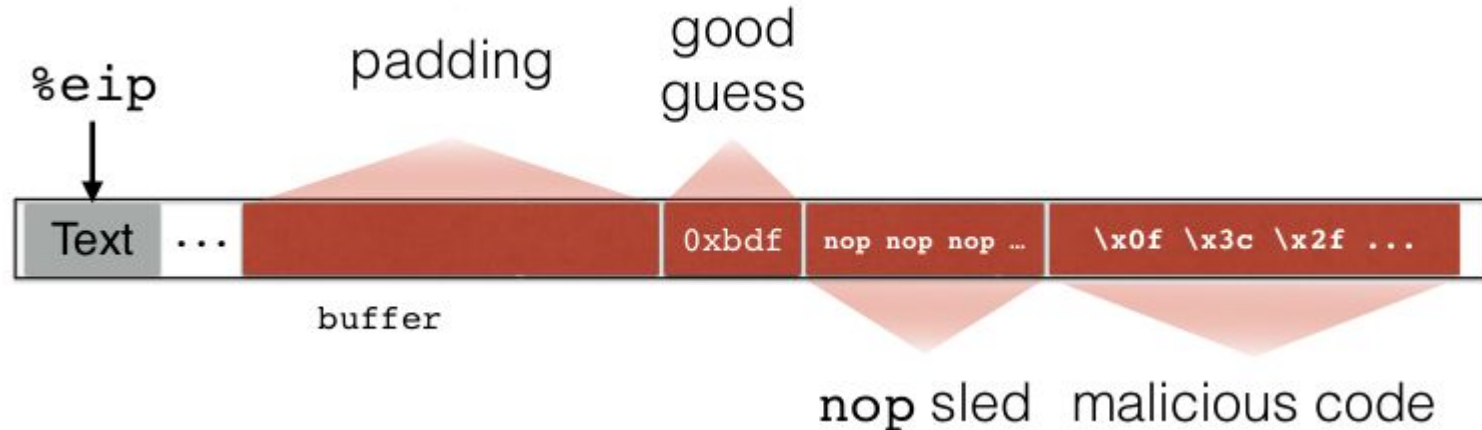
nop

`nop` = instruction d'1 seul octet qui renvoie à l'instruction suivante



Résumé de l'attaque "stack smashing" (1996)

On commence par quelques instructions (le buffer alloué dans la stack contient par exemple strcpy). Quand la fonction returns, on pointe vers %eip et on arrive au code malicieux.



Autres attaques : heap

Le buffer peut aussi être alloué dans la heap (via malloc).

```
typedef struct _vulnerable_struct {  
  
    char buff[MAX_LEN];  
  
    int (*cmp)(char*,char*);  
  
} vulnerable;
```

```
int foo(vulnerable* s,  
  
char* one, char* two)  
  
{  
  
    strcpy( s->buff, one );  
  
    strcat( s->buff, two );  
  
    return s->cmp( s->buff,  
"file://foobar" );  
  
}
```

Si $\text{strlen}(\text{one}) + \text{strlen}(\text{two}) \geq \text{MAX_LEN}$, overflow sur cmp

Autres attaques : integer overflow

Si nresp est large,
overflow sur response

```
void vulnerable()
{
    char *response;

    int nresp = packet_get_int();

    if (nresp > 0) {
        response = malloc(nresp*sizeof(char*));

        for (i = 0; i < nresp; i++)

            response[i] = packet_get_string(NULL);
    }
}
```

Autres attaques : corruption des données

Les précédentes attaques modifiaient du code.

On peut aussi modifier de la donnée :

- changer une clé secrète (utile pour décrypter d'autres données)
- modifier des variables d'état (ex: authenticated flag)
- modifier des strings (ex: injection SQL)

Autres attaques : read overflow

On lit le message (a priori de taille = len).

Si len est trop grand, on peut lire plus que ce qui est censé être autorisé.

-> bug HEARTBLEED

```
int main() {  
  
    char buf[100], *p;  
  
    int i, len;  
  
    while (1) {  
  
        p = fgets(buf, sizeof(buf), stdin );  
  
        if (p == NULL) return 0;  
  
        len = atoi(p);  
  
        p = fgets(buf, sizeof(buf), stdin );  
  
        if (p == NULL) return 0;  
  
        for (i=0; i<len; i++)  
  
            if (!isctrl(buf[i])) putchar(buf[i]);  
  
        else putchar('.');  
  
        printf("\n");  
  
    }  
}
```

Impacts potentiels

Normalement un bug mémoire fait crasher le programme, mais un attaquant peut aussi :

- voler des informations (ex: heartbleed)
- corrompre une information (ex: paramètres d'une machine)
- lancer du code

Attacks

- *Stack smashing*
- *Format string attack*
- *Stale memory access*
- *Return-oriented Programming (ROP)*

De nombreux systèmes critiques codés en C/C++

Quelques exemples :

- OS Kernels
- Browsers (chromium)
- Serveurs (nginx, nodejs) et bases de données (redis, pg)
- Systèmes embarqués

Jan 2020	Jan 2019	Change	Programming Language	Ratings	Change
1	1		Java	16.896%	-0.01%
2	2		C	15.773%	+2.44%
3	3		Python	9.704%	+1.41%
4	4		C++	5.574%	-2.58%

Source : TIOBE index, C est le langage de l'année 2019 !

Protection de la mémoire (en C++)

Diverses techniques :

- sanitizers (<https://github.com/google/sanitizers>)
- fuzzing <https://blog.hboeck.de/archives/868-How-Heartbleed-couldve-been-found.html>
- Address space layout randomization
- SafeStack (<http://clang.llvm.org/docs/SafeStack.html>)
- Control Flow Integrity (<http://clang.llvm.org/docs/ControlFlowIntegrity.html>)
- etc.

Gérer la mémoire est un problème compliqué

Simplification pour le développeur :

- garbage collector (java, go, etc.), avec un langage supportant les types si possible (js -> typescript)
- essayer de mieux gérer la mémoire et les race conditions, de manière déterministe (safe rust)
<https://hacks.mozilla.org/2019/02/rewriting-a-browser-component-in-rust/>
- mais ca n'empêche pas tous les problèmes de corruption !

Ce que vous avez appris

— — —

- Comment un attaquant peut exploiter la mémoire pour lancer son code ou modifier des données

Introduction à rust

mémoire, erreurs

gestion de la mémoire

Revoir les principes

— — —

<https://programmingisterrible.com/post/42215715657/postels-principle-is-a-bad-idea>

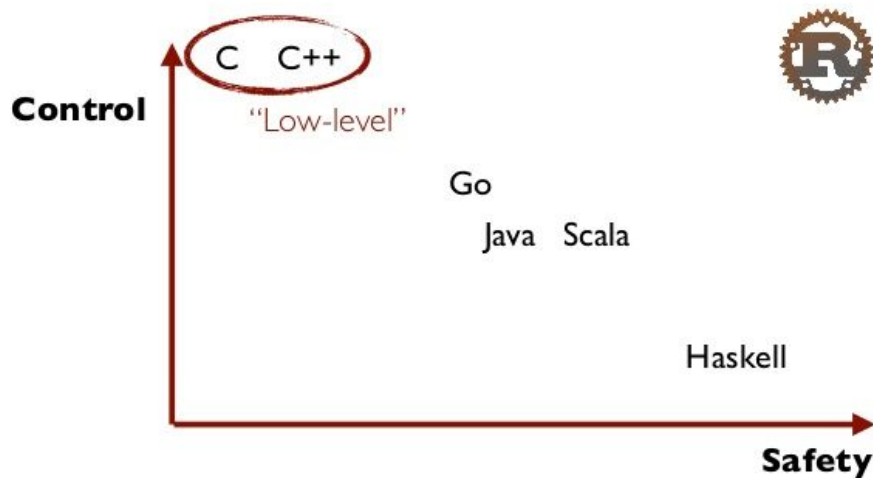
<https://www.cs.dartmouth.edu/~sergey/langsec/papers/postel-patch.pdf>

The Postel's Principle Patch

Here's our proposed patch:

- Be *definite* about what you accept.
- Treat valid or expected inputs as formal languages, accept them with a matching computational power, and generate their recognizer from their grammar.
- Treat input-handling computational power as a privilege, and reduce it whenever possible.

Pourquoi rust ?



Total					
	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

Pourquoi rust ?

Orienté system programming :

ex : browser mozilla

Until late 2018	From that point onwards
Rust is a systems programming language that runs blazingly fast, prevents segfaults and guarantees thread safety.	Empowering everyone to build reliable and efficient software.

Critique de rust

— — —

- complexe à apprendre (c'est plutôt vrai, il faut parfois se battre avec le compilateur, mais c'est sans doute mieux qu'un bug difficile à reproduire au runtime)
- ne remplacera pas tous les projets C/C++ (so what ?)

Peut-on faire plus simple ?

— — —

<https://without.boats/blog/notes-on-a-smaller-rust/> et <https://without.boats/blog/revisiting-a-smaller-rust/>

Comparaison des techniques https://aardappel.github.io/lobster/memory_management.html

Tout dépend de ce dont on a besoin. Ex ziglang (C amélioré, mais pas memory safe)

<https://drewdevault.com/2019/03/25/Rust-is-not-a-good-C-replacement.html>

<https://www.youtube.com/watch?v=Gv2l7qTux7g>

Ecosystème

— — —

<https://www.jetbrains.com/idea/devecosystem-2019/rust/>

29e du classement TIOBE

1er most loved language (stackoverflow) depuis 4 ans

bonne intégration avec le web (<https://rustwasm.github.io/wasm-bindgen/web-sys/index.html>)
et avec C (pas de problème de perf comme cgo)



Problèmes évités

— — —

- “dangling pointers” et “buffer overflow” : cf cours
- “data races” : typiquement plusieurs threads qui essaient d’accéder à la même variable globale en même temps
- “iterator invalidation”

```
for(iterator it = map.begin(); it != map.end(); ++it)
```

```
{ map.erase(it->first);
```

```
// whoops, now map has been restructured and iterator still thinks itself is healthy }
```

Problèmes évités : gestion de la mémoire

— — —

- compile time checks

```
fn for_each(v: &Vec<T>) {    // read only vector
    for item in 0..v.len() { // will never reach invalid memory
        work(v[things]);
    }
}
```

Mais pas dans tous les cas

— — —

- équivalent de heartbleed : si on choisit d'utiliser un seul buffer tout au long du programme et qu'on ne le réinitialise pas avant de le réutiliser, les informations des appels précédents sont à risque.

```
let buffer = &mut[0u8; 1024];
```

```
read_secrets(&user1, buffer);
```

```
store_secrets(buffer);
```

```
read_secrets(&user2, buffer);
```

```
store_secrets(buffer);
```

Conclusion par rapport à la sécurité

à ce jour, rust est bien meilleur que la plupart des autres langages mais ne permet pas d'éviter toutes les failles de sécurité :

- rust garantit que vos données ne seront jamais en mesure d'être écrites en deux endroits en même temps, mais ne protège pas contre les erreurs de logique
- il existe des projets visant à développer des preuves formelles :
 - <http://plv.mpi-sws.org/rustbelt/> (attention très complexe)
 - de ce point de vue, pas encore aussi avancé qu'un langage comme Ada (avec spark) ou Ocaml (coq)

Installation et hello world

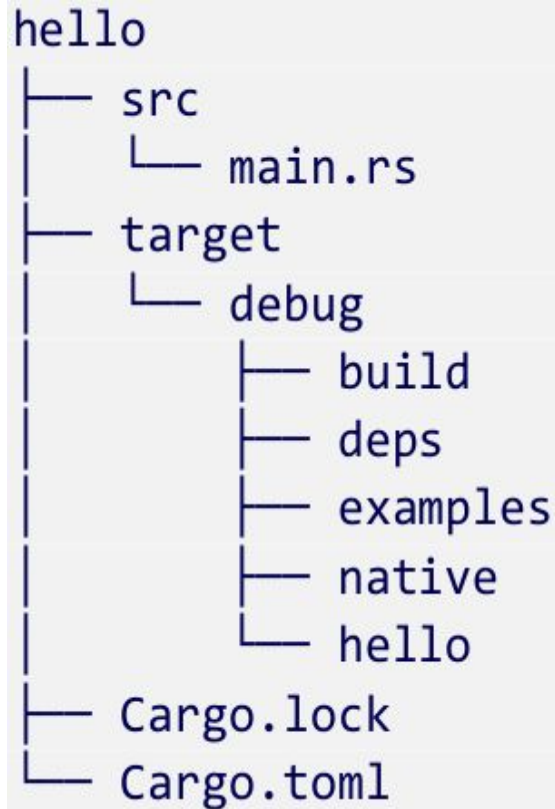
— — —

- Rustup : <https://www.rust-lang.org/tools/install>
- Tester : `cargo --version`
- Create project : `cargo new --bin hello (ou --lib)`

`cargo build (ou cargo build --release)`

`./target/debug/hello`

`cargo run`



Simplifier le développement

— — —

- Documentation :

<https://www.rust-lang.org/learn>

<https://doc.rust-lang.org/1.30.0/book/index.html>

y compris async : https://rust-lang.github.io/async-book/01_getting_started/04_async_await_primer.html

- IDE : VSCode + rust rls extension

Exemple de programme

— — —

```
fn main() {  
  
  let a = 10;  
  
  let b: i32 = 20;  
  
  let c = add(a, b);  
  
  println!("a + b = {}", c);  
  
}  
  
fn add(i: i32, j: i32) -> i32 {  
  
  i + j  
  
}
```

types numériques possibles :

i8 (8 bits) , i16 , i32 , i64 - signed integers

u8 , u16 , u32 , u64 - unsigned integers

f32 , f64 - floating point numbers

isize , usize - CPU's "native" width

pas besoin de ";" ni de return

Hello world ++

— — —

<https://www.notion.so/Chapter-1-Let-s-do-this-47d8f5a8f66a43a09a6e7ed4a328e79a>



Tester un programme existant

— — —

https://github.com/fimbault/rust_tuto/tree/master/basic_syntax



Résumé sur les types et les fonctions



Ownership

- stratégie spécifique à rust pour éviter les problèmes mémoire
- Chaque donnée doit avoir un owner, responsable de la désallocation de cette donnée
 - automatique quand la variable est “out of scope”
- Le owner décide aussi de la mutabilité

Notion de scope

- début : quand une valeur est créée ou bougée (move) dans une partie de la mémoire
- Fini si move ou drop

```
fn main() {  
    let list = vec![1, 2, 3];  
  
    let list_first = list.first();  
    let list_last = list.last();  
  
    println!(  
        "The first element is {:?} and the last is {:?}" ,  
        list_first,  
        list_last,  
    );  
}
```

Différence avec C



- En C, le code doit faire explicitement l'allocation `malloc(sizeof())` et la désallocation `free()`
- Avantages : rapide, utilise le moins de mémoire
- Inconvénients :
 - Use after free : équivalent d'un serveur qui vous enlève le plat avant que vous ayez fini de manger
 - Memory leaks : le serveur ne vient jamais nettoyer la table
 - Double free : le serveur et le client essaient tous les deux de nettoyer la table en même temps

Différence avec un garbage collector (GC)



- Un GC évite d'avoir à penser à nettoyer la mémoire
 - Ex: java, ruby, go
 - exemple de gestion de la mémoire en go <https://spacetime.dev/memory-security-go>
- Avantages :
 - plus simple pour le développeur
 - évite les oublis
- Inconvénients :
 - moins de contrôle sur les ressources
 - utilise plus de mémoire que nécessaire (mais ça peut être très optimisé)

Exemple d'algorithme GC : mark and sweep

— — —

Phase 1 : marquage

```
Mark(root)
```

```
If markedBit(root) = false then
```

```
markedBit(root) = true
```

```
For each v referenced by root
```

```
Mark(v)
```

root est une variable qui référence un objet est est directement accessible par une variable locale.

Quand un objet est créé, son bit de marquage est mis à 0.

Dans la phase de marquage, on set le bit à 1 pour tous les objets auquel un utilisateur peut se référer (via un graph traversal).

-> on trace tous les objets qui sont accessibles directement ou indirectement par le programme.

Exemple d'algorithme GC : mark and sweep

— — —

Phase 2 : sweep

For each object `p` in heap

 If `markedBit(p) = true` then

`markedBit(p) = false`

 else

`heap.release(p)`

vide la mémoire heap pour tous les objets non atteignables (cad avec un bit de marquage à 0).

En rust : pas seulement la mémoire + socket, mutex, file

sockets : éviter les problèmes

- use after close
- closing twice
- socket leaks

Un garbage collector n'aide pas à résoudre ces problèmes.

-> killer feature de rust

Scope d'une variable

— — —

```
fn say(s: String) {  
  
    println!("I say, {}!", s);  
  
}
```

```
fn main() {  
  
    let a = String::from("hello");  
  
    say(a);  
  
    println!("Using a again: {}", a);  
  
}
```



```
error[E0382]: borrow of moved value: `a`
```

```
--> src/main.rs:8:35
```

```
6 |     let a = String::from("hello");  
  |         - move occurs because `a` has type `std::string::String`, which does not implement the `Copy` trait  
7 |     say(a);  
  |         - value moved here  
8 |     println!("Using a again: {}", a);  
  |                                   ^ value borrowed here after move
```

```
error: aborting due to previous error
```

```
For more information about this error, try `rustc --explain E0382`.
```

```
error: could not compile `ownership`.
```

```
To learn more, run the command again with --verbose.
```

Garder l'ownership sur une donnée : cloning

- Cloning fait une deep copy, chaque data a son propre owner
 - une des façons de gérer les problèmes de “moved values”

```
main.rs
1  fn say(s: String) {
2      println!("I say, {}!", s);
3  }
4
5  fn main() {
6      let a = String::from("hello");
7      say(a.clone());
8      println!("Using a again: {}", a);
9  }
10
```

Exercise : pluralize (30 min)



— — —

```
fn main() {  
  
    let s = String::from("item");  
  
    // Add code here that calls the pluralize function  
  
    println!("I have one {}, you have two {}",s,you_add_something_here,);  
  
}  
  
// Add appropriate parameters, return values, and implementation to this function  
  
fn pluralize() {}
```

1 solution : pluralize

— — —



```
1 fn main() {
2     let s = String::from("item");
3
4     // Add code here that calls the pluralize function
5     let pl = pluralize(s.clone());
6
7     println!(
8         "I have one {}, you have two {}",
9         s,
10        pl,
11    );
12 }
13
14 // Add appropriate parameters, return values, and implementation to this function
15 fn pluralize(singular: String) -> String {
16     singular + "s"
17 }
```

Variantes : pluralize



— — —

- Que se passe-t-il si on ne clone pas s ?
- Que se passe-t-il si on clone s seulement dans println ?
- Que se passe-t-il si on fait

```
println!(  
    "I have one {}, you have two {}",  
    s,  
    pluralize(s),  
    );
```

Conclusion : pluralize



Exercice sur le ownership avec des strings et des fonctions

Solution avec `clone()` pour faire un move de la data

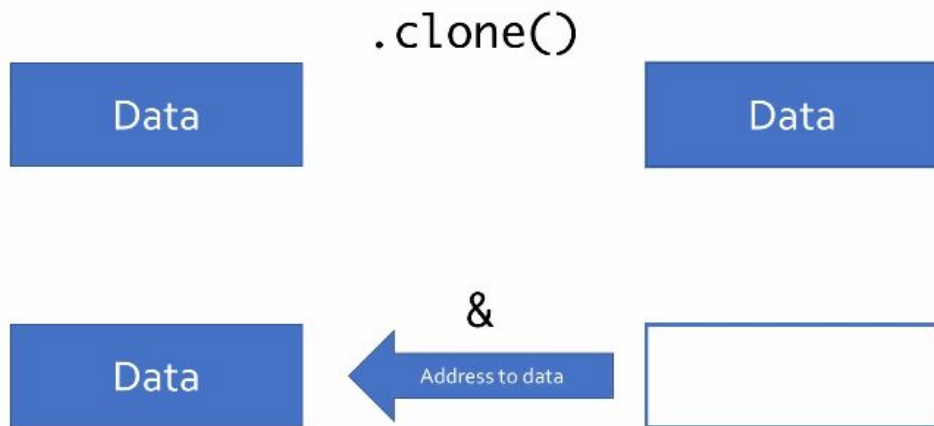
Ce qui ne marche pas : lorsque le move ne se fait pas dans le bon ordre

Pourquoi est-ce plus dur que dans d'autres langages ?

- On n'a pas de GC, et la gestion manuelle de la mémoire génère trop d'erreurs
- L'approche de rust se base sur l'ownership :
 - cleanup quand la variable est out of scope
 - les opérations qui prennent du temps (clone) sont explicites
- clone() est pas idiomatique
 - la solution est le **borrowing**

Une solution idiomatique : borrowing

- Permet d'utiliser à du code d'utiliser une valeur sans bouger l'ownership (donc plus performant)
- On donne une référence (&) à la valeur d'origine



Borrow : pluralize

— — —

```
fn main() {  
  
    let s = String::from("item");  
  
    let pl = pluralize(&s);  
  
    println!(  
        "I have one {}, you have two {}",  
        s,  
        pl,  
    );  
}
```

Borrow : pluralize

— — —

- On n'y est presque ... “binary operation + cannot be applied to &str”

```
error[E0369]: binary operation '+' cannot be applied to type '&str'
--> src/main.rs:16:14
```

```
16 | singular + "s"
    |          ^ -- &str
    |          |
    |          + cannot be used to concatenate two '&str' strings
    |          &str
```

help: `to_owned()` can be used to create an owned `String` from a string reference. String concatenation appends the string on the right to the string on the left and may require reallocation. This requires ownership of the string on the left

```
16 | singular.to_owned() + "s"
```

- Le compilateur nous aide !

Borrow : pluralize

— — —

- Il faut donc modifier légèrement la fonction pluralize

```
fn pluralize(singular: &str) -> String {  
  
    singular.to_owned() + "s"  
  
}
```

Comparaison clone et borrow

— — —

```
1 fn main() {
2     let s = String::from("item");
3
4     let pl = pluralize(s.clone());
5
6     println!(
7         "I have one {}, you have two {}",
8         s,
9         pl,
10    );
11 }
12 |
13 fn pluralize(singular: String) -> String {
14     singular + "s"
15 }
16
```

```
1 fn main() {
2     let s = String::from("item");
3
4     let pl = pluralize(&s);
5
6     println!(
7         "I have one {}, you have two {}",
8         s,
9         pl,
10    );
11 }
12 |
13 fn pluralize(singular: &str) -> String {
14     singular.to_owned() + "s"
15 }
16 |
```

Exemple : push_str

- la signature d'une fonction qui fait un borrow sur ses paramètres indique que la fonction ne gère pas l'allocation ou la deallocation de la mémoire

```
[_] pub fn push_str(&mut self, string: &str)
```

Appends a given string slice onto the end of this String.

Examples

Basic usage:

```
let mut s = String::from("foo");  
  
s.push_str("bar");  
  
assert_eq!("foobar", s);
```

Borrowing vs Pointeur

- Proche des pointeurs en C/C++
- Mais une grosse différence : **le borrow checker assure à la compilation qu'on n'aura jamais une référence invalide**
- Permet d'éviter des erreurs au runtime (avec les difficultés potentielles de reproduction):
 - segmentation fault (en C)
 - undefined is not a function (js)

Résumé ownership et borrowing

- borrowing permet d'emprunter une valeur au lieu de transférer l'ownership
- Permet de réduire les allocations et d'améliorer la performance
- rust s'assurer que les références sont toujours valides



String vs str

String et &str sont des groupements de u8

Il existe 2 types différents, du fait de la gestion mémoire :

- String (owned) est mutable, alloué sur la heap
- str (borrowed) est immutable, c'est une vue

Exemple :

```
let x = "config"; // de type &str
```

```
let x = *"config"; error: the trait `core::marker::Sized` is not implemented for the type `str` [E0277]
```

Passer de String à str

str contient une adresse mémoire et une longueur, String a en plus une capacité qui représente la quantité de mémoire réservée (mais pas nécessairement utilisée).

```
let x: &str = "a";
```

```
let y: String = x.to_owned(); // on aurait aussi pu utiliser `String::from` ou `str::into`
```

```
let z: &str = &y;
```

Autres types de données

- `array ([T; N])`, ex: `[1, 2, 3]`, ou repeat expression `[0; 100]`
 - contrairement à C, un tableau est alloué sur la stack, donc accessible directement (sans pointeur)
- `Vec<T>` : liste de T qui peut augmenter ou diminuer en taille
- `slice ([T])` : équivalent d'un tableau mais sa taille est dynamique et il est plus facile d'implémenter des traits (méthodes)

```
let mut v = vec![1,2,3];
```

```
v.push(4);
```

```
let v_slice = &v[1..2];
```

Vec vs slice

le type `Vec` permet de modifier le contenu d'une vue (non constante) représentée par les slice :

```
let x: &[i32] = &[0, 1, 2];
```

```
let y: Vec<i32> = x.to_vec();
```

```
let z: &[i32] = &y;
```

Slice

- indices vérifiés au runtime (ex : éviter out of bounds)

```
let mut v = vec![1,2,3];
```

```
let v_slice = &v[..9];
```

index 9 out of range for slice of length 3

let v =

Vec<i32>	
data	
length	3
capacity	3

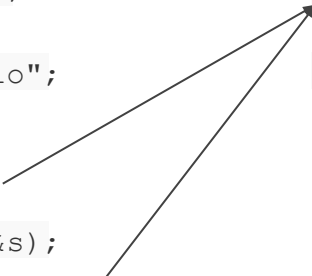
slice	
pointer	
length	9

10	0
20	1
30	2
	3
NOT	4
	5
YOUR	6
	7
MEMORY!	8
	9

Utilisation des slices en paramètres

- donne plus de flexibilité que des arrays ou vectors
- convertit `&String` en `&str` (pour faciliter)


```
fn main() {  
    let s = String::from("hi");  
    let string_literal = "hello";  
  
    either_string_or_literal(&s);  
    either_string_or_literal(&string_literal);  
}  
  
fn either_string_or_literal(param: &str) {  
    println!("this is a string slice: {:?}", param);  
}
```



Mutable reference

— — —

```
#[derive(Debug)]  
struct Account {  
    balance: u32,  
}
```



permit 1'inspection {:?}

```
fn transfer(source: &Account, target: &Account, amount: u32)  
{  
    source.balance -= amount;  
    target.balance += amount;  
}
```

```
fn main() {  
    let acct1 = Account { balance: 20 };  
    let acct2 = Account { balance: 10 };  
  
    transfer(&acct1, &acct2, 3);  
  
    println!("Account 1: {:?}", acct1);  
    println!("Account 2: {:?}", acct2);  
}
```


Mutable reference

— — —

```
error[E0594]: cannot assign to `source.balance` which is behind a `&` reference
--> src/main.rs:7:5
6 | fn transfer(source: &Account, target: &Account, amount: u32) {
  | ----- help: consider changing this to be a mutable reference: `&mut Account`
7 |     source.balance -= amount;
  |     ~~~~~ source is a `&` reference, so the data it refers to cannot be written
```

Mutable reference

— — —

```
fn transfer(source: &mut Account, target: &mut Account, amount: u32) {  
    source.balance -= amount;  
    target.balance += amount;  
}
```

```
fn main() {  
    let mut acct1 = Account { balance: 20 };  
    let mut acct2 = Account { balance: 10 };  
  
    transfer(&mut acct1, &mut acct2, 3);  
}
```

&mut self

— — —

changer les champs de l'instance courante (en premier paramètre d'une méthode)

```
impl Account {  
    // earn money  
    fn earn(&mut self, amount: u32) {  
        self.balance += amount;  
    }  
}  
  
fn main() {  
    let mut acct3 = Account { balance : 0 };  
    acct3.earn( 10);  
}
```

Règles de borrowing liée à la mutabilité

plusieurs immutable references

OU 1 seule mutable reference



Exemple d'erreur

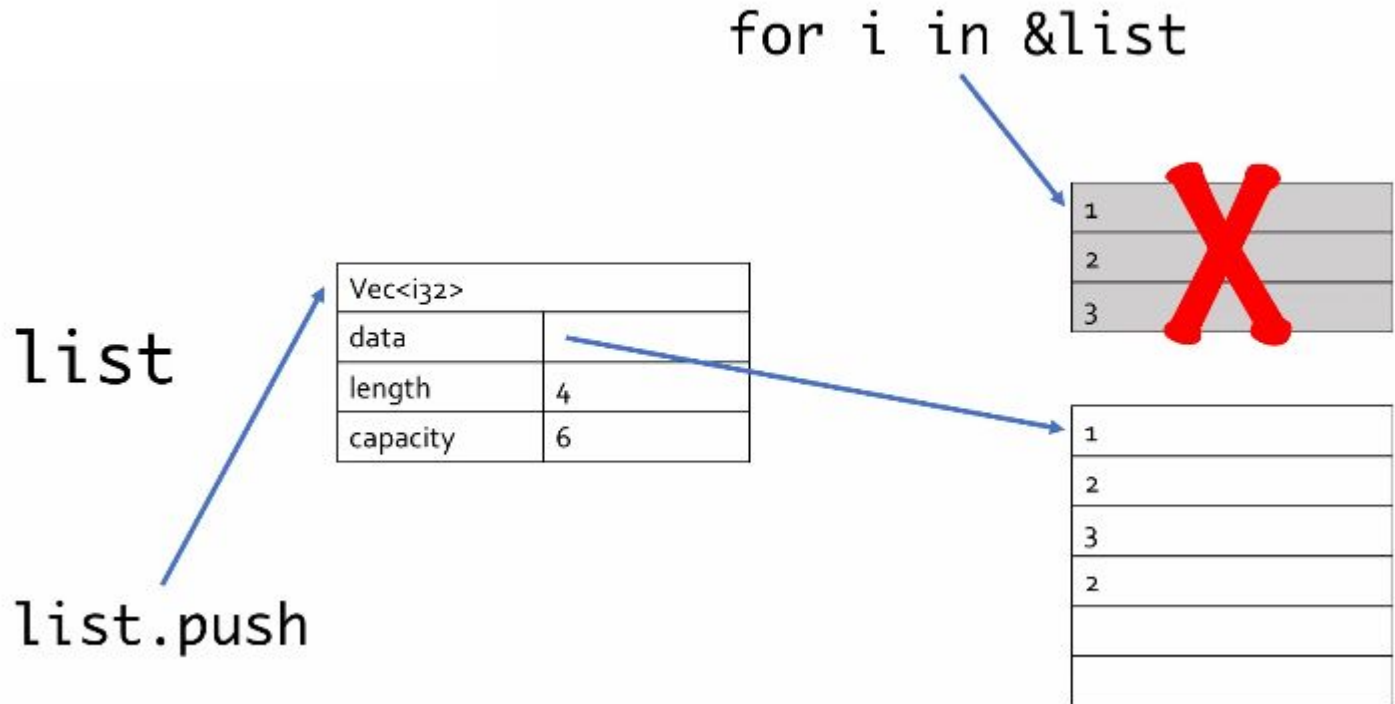
```
fn main() {  
    let mut list = vec![1, 2, 3];  
    for i in &list {  
        println!("i is {}", i);  
        list.push(i + 1);  
    }  
}
```

```
pub fn push(&mut self ch: char)
```

Appends the given `char` to the end of this `String`.

```
error[E0502]: cannot borrow `list` as mutable because it is also borrowed as immutable  
--> src/main.rs:5:9  
3 |     for i in &list {  
    |               -----  
    |               |  
    |               immutable borrow occurs here  
    |               immutable borrow later used here  
4 |         println!("i is {}", i);  
5 |         list.push(i + 1);  
    |         ^^^^^^^^^^^^^^^ mutable borrow occurs here
```

Le compilateur détecte un iterator invalidation



Electric boogaloo

— — —



<https://www.notion.so/Chapter-2-Electric-Boogaloo-9949ba6bcc154246b1348366c086f4c5>

gestion des erreurs

Des erreurs vont se produire

Exemples :

- vérification d'un user input
- accès à des ressources tierces (ex: timeout sur requête réseau)
- ou tout simplement une erreur de programmation

En python : exception

Python permet d'attraper les erreurs via un `catch` n'importe où dans la `call stack`.

Le problème est qu'il faut connaître toutes les exceptions qui peuvent être levées (`throw`) par une fonction et tous (!!) ses appels internes.

En pratique, on oublie souvent ...

En python : try

— — —

```
while True:
```

```
try:
```

```
    x = int(input("Please enter a number: "))
```

```
    break
```

```
except ValueError:
```

```
    print("Oops! That was no valid number. Try again...")
```

Pour en savoir plus : <https://docs.python.org/fr/3.5/tutorial/errors.html>

En rust : enum types

Result peut retourner :

- un résultat : Ok / Err
- une option : Some / None

https://learning-rust.github.io/docs/e3.option_and_result.html

On va voir comment ça fonctionne et pourquoi c'est utile.

Gestion des erreurs

— — —

On peut :

- arrêter le programme (panic)
- essayer de trouver une solution (recover)
- utiliser des valeurs par défaut
- propager une erreur
- notifier l'utilisateur

Exemple : command line

- Peut traiter des données pour différentes plateformes (windows ou linux, le reste n'est pas supporté)
- Spécifié avec un flag -p

```
./cmdline -p linux
```

- Que se passe-t-il si on passe une valeur qui n'est pas attendue ?

```
panic $ cargo run
  Compiling panic v0.1.0 (file:///private/tmp/panic)
  Finished dev [unoptimized + debuginfo] target(s) in 1.8 secs
  Running `target/debug/panic`
thread 'main' panicked at 'explicit panic', src/main.rs:2:5
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

panic macro

— — —

```
fn main() {  
  
    let arg = 0;  
  
    panic!("Oops got value {}", arg);  
  
}
```

écrire l'application cmdline

— — —

```
#[derive(Debug)]
```

```
enum Platform {
```

```
    Windows,
```

```
    Linux,
```

```
}
```

```
impl Platform {
```

```
    fn parse(platform_arg: &str) -> Platform {
```

```
        // set value or panic
```

```
}
```

```
fn main() {
```

```
    let platform_arg = "linux";
```

```
    let platform = Platform::parse(platform_arg);
```

```
    println!("Producing output for {:?}", platform);
```

```
}
```


panic macro

— — —

```
fn parse(platform_arg: &str) -> Platform {  
  
    if platform_arg == "windows" {  
  
        Platform::Windows  
  
    } else if platform_arg == "linux" {  
  
        Platform::Linux  
  
    } else {  
  
        panic!("Unknown platform: {}. Valid values: windows or linux", platform_arg);  
  
    }  
  
}
```

quand paniquer ?

— — —

1) continuer serait incorrect

- ex de l'appli cmdline si on tourne sur macos)
- out of bound memory access -> index out of bound panic

```
fn main() {  
  
    let v = vec![1, 2, 3];  
  
    let index = 10;  
  
    println!("The value is {}", v[index]);  
  
}
```

quand paniquer ?

— — —

2) pas de possibilité de recover

3) quand le problème doit être corrigé dans le code (bug)

```
extern crate http;
```

```
use http::header::HeaderName;
```

```
fn main() {
```

```
    let _h = HeaderName::from_static("special! characters!");
```

```
}
```

quand ne pas paniquer

— — —

L'erreur est une occurrence récurrente.

Ex : dans un browser, du HTML malformé.

macros qui appellent panic!

— — —

unreachable! impossible d'arriver à cet endroit du code

```
enum DoorState {  
    Opened,  
    Closed,  
}  
  
enum DoorAction {  
    Open,  
    Close,  
}  
  
fn take_action(current_state: DoorState, action: DoorAction) {  
    match (current_state, action) {  
        (DoorState::Opened, DoorAction::Close) => {  
            // code to close the door goes here  
        },  
        (DoorState::Closed, DoorAction::Open) => {  
            // code to open the door goes here  
        },  
        // If you get here, a programming mistake has been made  
        _ => unreachable!(),  
    }  
}
```

macros qui appellent panic!

— — —

`unimplemented!`

`assert!`, `assert_eq!`, `assert_ne!` : pour vérifier des préconditions ou dans des tests

Recap : panic!

— — —

On a vu quand paniquer est une bonne idée

On va maintenant voir comment gérer les résultats d'une manière plus générale.

Gérer les retours

Exemple : parser du JSON

Input string	Return value
<code>{"k":"v"}</code>	<code>serde_json::Object({ "k":serde_json::String("v") })</code>
<code>{1:2}</code>	key must be a string
<code>{"k":"v", }</code>	trailing comma

Cas de succès

`Ok(value)`

`Some(value)`

`!= value (type mismatch)`

Exemple

— — —

```
extern crate serde;
```

```
extern crate serde_json;
```

```
#[macro_use]
```

```
extern crate serde_derive;
```

```
#[derive(Deserialize, Debug)]
```

```
struct Person {
```

```
    name: String,
```

```
}
```

```
fn main() {
```

```
    let first = serde_json::from_str::<Person>(r#"{
```

```
        "name": "John Doe"
```

```
    }"#);
```

```
    println!("first's name = {:?}", first.name);
```

```
}
```

```
no field `name` on type `std::result::Result<Person, serde_json::Error>`
```

Mapping Ok / Err

— — —

```
let first_inner = match first {  
  
    Ok(inner) => inner,  
  
    Err(_) => Person {  
  
        name: String::from("unknown"),  
  
        },  
  
};  
  
println!("first's name = {:?}", first_inner.name);
```

Plusieurs types d'erreurs

— — —

```
let first_inner = match first {  
  
    Ok(inner) => inner,  
  
    Err(error) => {  
  
        if error.is_eof() {  
  
            // process  
  
        }  
  
    }  
  
};
```

Struct `serde_json::Error`

[+] Show declaration

[-] This type represents all possible errors that can occur when serializing or deserializing JSON data.

Methods

-] `impl Error`

[+] `pub fn line(&self) -> usize`

[+] `pub fn column(&self) -> usize`

[+] `pub fn classify(&self) -> Category`

[+] `pub fn is_io(&self) -> bool`

[+] `pub fn is_syntax(&self) -> bool`

[+] `pub fn is_data(&self) -> bool`

[-] `pub fn is_eof(&self) -> bool`

Returns true if this error was caused by prematurely reaching the end of the input data.

Callers that process streaming input may be interested in retrying the deserialization once more data is available.

Recoverable => Unrecoverable

— — —

```
let first_inner = match first {  
  
    Ok(inner) => inner,  
  
    Err(_) => panic!("couldn't parse JSON into Person!"),  
  
};
```

Note : Unrecoverable => Recoverable n'est pas possible (sauf pour quelques cas très spécifiques)

Exemple : save

Input = texte

Validate < 200 bytes

Save to the database

Returns :

- Success : Id of the new database record
- Failure : error with info

Exemple : save

Success case : ID

Error case : hardcoded string litteral

— — —

```
fn save(text: &str) -> Result<i64, &static str> {
```

```
    if text.len() > 200 {
```

```
        return Err("text is too long");
```

```
    }
```

```
}
```

Note : `'static` est un exemple de lifetime (la référence est valide pendant tout le programme)

Sauvegarde en base de données

— — —

Suppose it fails

```
struct StatusRecord {  
  
    id: i64,  
  
    // whatever  
  
}  
  
fn save_to_database(text: &str) -> Result<StatusRecord, &'static str> {  
  
    // fake implementation that always fails  
  
    Err("database unavailable")  
  
}
```


Sauvegarde en base de données

— — —

```
fn save_status(text: &str) -> Result<i64, &'static str> {  
  
    if text.len() > 200 {  
  
        return Err("status text is too long");  
  
    }  
  
    let record = match save_to_database(text) {  
  
        // write the Ok and Err cases  
  
    };  
  
}
```

Sauvegarde en base de données

— — —

```
let record = match save_to_database(text) {  
  
    Ok(rec) => rec,  
  
    Err(e) => return Err(e),  
  
};
```

C'est tellement commun que l'opérateur ? peut remplacer cette implémentation :

```
let record = match save_to_database(text)?;
```

Où peut-on utiliser l'opérateur '?'

Seulement dans une fonction qui retourne `Result` ou `Option` (depuis `rust 1.22`)

Modifier le code suivant pour permettre d'utiliser l'opérateur '?'

```
fn add_one(n: &str) -> i32 {  
  
    let num: i32 = n.parse()?;  
  
    num + 1  
  
}
```

Où peut-on utiliser l'opérateur '?'

— — —

```
fn add_one(n: &str) -> Result<i32, std::num::ParseIntError> {  
  
    let num: i32 = n.parse()?;  
  
    Ok(num + 1)  
  
}
```

Struct `std::num::ParseIntError`

1.0.0 [-][src]

[+] Show declaration

[-] An error which can be returned when parsing an integer.

This error is used as the error type for the `from_str_radix()` functions on the primitive integer types, such as `i8::from_str_radix`.

Potential causes

Among other causes, `ParseIntError` can be thrown because of leading or trailing whitespace in the string e.g., when it is obtained from the standard input. Using the `str.trim()` method ensures that no whitespace remains before parsing.

Option

— — —

```
fn add_one_to_last(n: &[i32]) -> i32 { // à modifier comme retournant une Option
```

```
    let num = n.last()?;
```

```
    num + 1
```

```
}
```

```
fn main() {
```

```
    let list = vec![1, 2, 3];
```

```
    println!("add_one_to_last on list: {:?}", add_one_to_last(&list));
```

```
    // essayer aussi sur un vecteur vide
```

```
}
```

Option

— — —

```
fn add_one_to_last(n: &[i32]) -> Option<i32> {
```

```
    let num = n.last()?;
```

```
    Some(num + 1)
```

```
}
```

Option

— — —

```
fn main() {  
  
    let list = vec![1, 2, 3];  
  
    println!("add_one_to_last on list: {:?}", add_one_to_last(&list)); // type Some(8)  
  
    let empty_list = vec![];  
  
    println!("add_one_to_last on empty list: {:?}", add_one_to_last(&empty_list)); // type None  
  
}
```

conversions Result/Option

Il existe des méthodes de conversion :

Result -> Option

Option -> Result

Il existe aussi des possibilités de fallback, ou de transformation de certaines valeurs Ok/Some seulement

Cas spécifique de main et des tests

Depuis rust 1.28, c'est possible aussi

```
#[test]

fn parsing_works() -> Result<(), Box<dyn std::error::Error>>{

    let x: i32 = "1".parse()?;

    Ok(())

}

fn main() {

    println!("Hello, world!");

}
```

Testing method

Result : is_ok, is_err

Option : is_some, is_none

<https://doc.rust-lang.org/std/result/>

```
let good_result: Result<i32, i32> = Ok(10);
let bad_result: Result<i32, i32> = Err(10);

// The `is_ok` and `is_err` methods do what they say.
assert!(good_result.is_ok() && !good_result.is_err());
assert!(bad_result.is_err() && !bad_result.is_ok());
```

Plusieurs types d'erreur : Box<Error>

— — —

```
fn num_threads() -> Result<usize, ???> {  
  
    let s = env::var("NUM_THREADS")?; // env::VarError  
  
    let n: usize = s.parse()?; // num::ParseIntError  
  
    Ok(n+1) // quand tout se passe bien  
  
}
```

Box est un objet trait, composé d'un pointeur (Box) et d'un trait (std::error::Error) : le type exact de l'erreur est déterminé au runtime (NB: Error nécessite Debug et Display)

Box<Error>

— — —

```
use std::env;
```

```
use std::error::Error;
```

```
fn num_threads() -> Result<usize, Box<Error>> {
```

```
    let s = env::var("NUM_THREADS")?;
```

```
    let n: usize = s.parse()?;
```

```
    Ok(n + 1)
```

```
}
```

```
fn run_application() -> Result<(), Box<Error>> {
```

```
    let num = num_threads()?;
```

```
    println!("the number of threads is {}", num);
```

```
    // Rest of the program's functionality
```

```
    Ok(())
```

```
}
```

```
fn main() {
```

```
    if let Err(e) = run_application() {
```

```
        panic!("An error happened: {}", e);
```

```
    }
```

```
}
```

Inconvénients de Box<Error>

- On ne peut pas inspecter le type de l'erreur dans le code
- On ne peut pas décider de gérer différentes erreurs de manière différente

On verra plus tard une 2e solution : custom error type

Avantage de la gestion d'erreur dans rust

On distingue les bugs des erreurs récupérables

- bugs : panic! proche de la cause
- erreur récupérable : la possibilité d'erreurs est explicite

Rust vs C : buffer overread

— — —

- Ecrire le même programme en rust

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
int main() {
```

```
    int* accts = calloc(2,  
sizeof(int));
```

```
    accts[0] = 50298;
```

```
    accts[1] = 7423;
```

```
    int bal = accts[11];
```

```
    printf("Balance: %d\n", bal);
```

```
    free(accts);
```

```
    return 0;
```

```
}
```

compile et runs avec
un résultat faux

Your balance is: 0

Rust vs C : buffer overread

— — —

```
fn main() {  
  
    let accts = vec![  
  
        50298,  
  
        7423  
  
    ];  
  
    let bal = accts[11];  
  
    println!("Balance: {}",bal);  
  
}
```

```
#include <stdio.h>  
  
#include <stdlib.h>  
  
int main() {  
  
    int* accts = calloc(2,  
sizeof(int));  
  
    accts[0] = 50298;  
  
    accts[1] = 7423;  
  
    int bal = accts[11];  
  
    printf("Balance: %d\n", bal);  
  
    free(accts);  
  
    return 0;  
  
}
```


Rust vs C : buffer overread

— — —

```
fn main() {
```

```
    let accts = vec![
```

```
        50298,
```

```
        7423
```


```
    ];
```

```
    let bal = accts[11];
```

```
    println!("Balance: {}", bal);
```

```
}
```

compiles but fails
loudly and faster



```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
int main() {
```

```
    int* accts = calloc(2,  
sizeof(int));
```

```
    accts[0] = 50298;
```

```
    accts[1] = 7423;
```

```
    int bal = accts[11];
```

```
    printf("Balance: %d\n", bal);
```

```
    free(accts);
```

```
    return 0;
```

thread 'main' panicked at 'index out of bounds: the len is 2 but the index is 11'

expect

— — —

```
let num : i32 = "1".parse()
```

```
.expect("parsing failed");
```

- Success : num vaudra 1
- Failure : panic! et affiche le message

```
let num : i32 = "NotNum".parse()
```

```
.expect("parsing failed");
```

expect : cas d'un fichier de config

— — —

```
let config = File::open("config.toml")
```

```
.expect("Config file was created");
```

- En général, on ouvre le fichier :
 - `File::open` retourne un `Result` car l'opération peut ne pas fonctionner
- S'il n'existe pas, on le crée

Result : savoir qu'une fonction peut ne pas marcher

Function `serde_json::from_str`

```
pub fn from_str<'a, T>(s: &'a str) -> Result<T>
where
    T: Deserialize<'a>,
```

Result indique qu'il est possible que la fonction ne fonctionne pas (même sans regarder l'implémentation)

En python, aucune indication d'un problème potentiel dans la signature des fonctions, donc difficile de savoir quelle erreur on peut rencontrer : <https://docs.python.org/fr/3/library/json.html>

le compilateur rust oblige à gérer les erreurs

Il n'est pas possible d'utiliser directement un résultat Ok

error[E0308]: mismatched types

```
let mut num: i32 = "10".parse();  
num += 1;
```

^^^^^^^^^^^^^^^^ expected i32, found
enum `std::result::Result`

En pratique : on implémente expect et on peut rechercher dans le code pour savoir où le changer

comparaison avec golang

En go, on retourne un tuple (success value, error value)

Convention : vérifier sur l'erreur est nil

Mais aucune vérification par le compilateur



My point isn't that exceptions are bad. My point is that exceptions are too hard and I'm not smart enough to handle them.

[https://rauljordan.com/2020/07/06/why-go-error-handling-is-a
awesome.html](https://rauljordan.com/2020/07/06/why-go-error-handling-is-a-awesome.html)

désavantage de la gestion d'erreur avec rust

Quand on commence à développer, gérer les erreurs n'est pas la priorité

-> utiliser des expect

-> mieux gérer les cas d'erreur plus tard si besoin

aller plus loin sur les erreurs

— — —

Des librairies existent pour simplifier le travail

<https://blog.yoshuawuyts.com/error-handling-survey/>

examples

— — —



<http://www.sheshbabu.com/posts/rust-error-handling/>

<https://github.com/sheshbabu/rust-error-handling-examples>