Rust programming

Course – 3

Gavrilut Dragos

Agenda for today

1. Enums
2. Error management
   1. Panic
   2. Option
   3. Result
3. if let / let else / while let
4. Question mark operator

Enums

Enums

Enums in Rust are quite different that a classical concept of ***enums*** from C/C++.

The format (in terms of keywords) is however similar:

Rust

```
enum <name> {
    value_1,
    value_2
    ...
    value_n
}
```

Similarly, to access a value from the enumeration use **<enum_name>::value**

Enums

Let’s see some examples:

Rust

```rust
enum Color {
    Red,Green,Blue,White,Black,
}
fn main() {
    let mut c = Color::Red;
    if c == Color::Red {
        println!("Color is red");
    }
}
```

Error

```
error[E0369]: binary operation `==` cannot be applied to type `Color`
 --> src\main.rs:7:10
  |
7 |     if c == Color::Red {
  |        - ^^ ---------- Color
  |        |
  |        Color
  |
note: an implementation of `PartialEq<_>` might be missing for `Color`
 --> src\main.rs:2:1
  |
2 | enum Color {
  | ^^^^^^^^^^ must implement `PartialEq<_>`
help: consider annotating `Color` with `#[derive(PartialEq)]`
  |
2 | #[derive(PartialEq)]
  |
```

The code won’t compile
because unlike C/C++ an **enum** is not implicitly associated with an int value, and as such can not be compared with another type !

Enums

While initially the tendency is to consider these two pieces of code as equivalents, in reality, their behavior is quite different.

Rust

```rust
enum Color {
    Red,Green,Blue,White,Black,
}
fn main() {
    let mut c = Color::Red;
    if c == Color::Red {
        println!("Color is red");
    }
}
```

Compile error

binary operation `==` cannot be applied to type `Color`

C/C++

```cpp
enum class Color {
    Red, Green, Blue, White, Black
};
void main() {
    auto c = Color::Red;
    if (c == Color::Red)
        printf("Color is Red");
}
```

Compiles ok

Output

Color is Red

Enums

This is because in Rust, an enum is more similar to a C++ class than to C/C++ enum type.

Rust

```rust
enum Color {
    Red,Green,Blue,White,Black,
}
```

This is a better approximation of how Rust enums work.

C/C++

```cpp
class Color
{
    int value;
public:
    constexpr static int Red    = 0;
    constexpr static int Green  = 1;
    constexpr static int Blue   = 2;
    constexpr static int White  = 3;
    constexpr static int Black  = 4;
    Color(int v) : value(v) {}
};
```

Enums

Let’s see how the new example in C/C++ compiles.

C/C++

```cpp
class Color {
    int value;
public:
    constexpr static int Red    = 0;
    constexpr static int Green  = 1;
    constexpr static int Blue   = 2;
    constexpr static int White  = 3;
    constexpr static int Black  = 4;
    Color(int v) : value(v) {}
};
void main() {
    Color c = Color::Red;
    if (c == Color::Red)
        printf("Color is Red");
}
```

Error (MS compiler for C/C++)

```
Test.cpp(130,8): error C2676: binary '==': 'Color'
does not define this operator or a conversion to a
type acceptable to the predefined operator
```

So … what happened ?

Enums

Let's see how the new example in C/C++ compiles.

C/C++

```
class Color {
    int value;
public:
    constexpr static int Red    = 0;
    constexpr static int Green  = 1;
    constexpr static int Blue   = 2;
    constexpr static int White  = 3;
    constexpr static int Black  = 4;
    Color(int v) : value(v) {}
};
void main() {
    Color c = Color::Red;
    if (c == Color::Red)
        printf("Color is Red");
}
```

Error (MS compiler for C/C++)

```
Test.cpp(130,8): error C2676: binary '==': 'Color'
does not define this operator or a conversion to a
type acceptable to the predefined operator
```

So … what happened ?

This works because of Color ctor that receives an **int** as a parameter. Since Color::Red = 0 (is defined as an int), the expression is equivalent to Color c(Color::Red);

Enums

Let's see how the new example in C/C++ compiles.

C/C++

```
class Color {
    int value;
public:
    constexpr static int Red    = 0;
    constexpr static int Green  = 1;
    constexpr static int Blue   = 2;
    constexpr static int White  = 3;
    constexpr static int Black  = 4;
    Color(int v) : value(v) {}
};
void main() {
    Color c = Color::Red;
    if (c == Color::Red)
        printf("Color is Red");
}
```

Error (MS compiler for C/C++)

```
Test.cpp(130,8): error C2676: binary '==': 'Color'
does not define this operator or a conversion to a
type acceptable to the predefined operator
```

So … what happened ?

This, however, will not compile. "c" is of type Color, Color::Red is of type int, and there is no defined cast to convert from a Color to int, nor any operator to evaluate == between a Color and an int !

Enums

Let's see how the new example in C/C++ compiles.

C/C++

```
class Color {
    int value;
public:
    constexpr static int Red    = 0;
    constexpr static int Green  = 1;
    constexpr static int Blue   = 2;
    constexpr static int White  = 3;
    constexpr static int Black  = 4;
    Color(int v) : value(v) {}
    bool operator==(int v) { return value == v; }
};
void main() {
    Color c = Color::Red;
    if (c == Color::Red) printf("Color is Red");
}
```

Output
Color is Red

Now the code compiles and produces the expected output.

Enums

So, what does the previous example means for our Rust code:

Rust

```rust
enum Color {
    Red,Green,Blue,White,Black,
}
fn main() {
    let mut c = Color::Red;
    if c == Color::Red {
        println!("Color is red");
    }
}
```

It means that **we need to add a way to compare two Color objects,** if we want this program to compile and run as expected.

Enums

So, what does the previous example means for our Rust code:

Rust

```
#[derive(PartialEq)]
enum Color {
    Red,Green,Blue,White,Black,
}
fn main() {
    let mut c = Color::Red;
    if c == Color::Red {
        println!("Color is red");
    }
}
```

Output

```
Color is Red
```

Now the code runs and works as expected.
But what is that #[...] formula on top of the ***enum*** declaration ?

Enums

The pound sign (#) followed by [...] is the way Rust adds attributes for:

- Various declarations (structures, enums, etc)
- Methods or functions
- The entire program

Attributes will be studied in another course, but … they can be used for several things:

- To provide metadata about an object (version, name, docs, etc)
- To set up the configuration the compiler/linker should use when building an object
- To automatically generate code
- ….

Enums

#[derive(<name>)] means automatically implementing a trait called <name> into a structure / class / enum / etc. Automatically in this context means different things based on the trait.

In particular for an enum, we have used #[derive(PartialEq)] → meaning that we will implement the ***PartialEq*** trait for that enum.

For a type (let's call it ***SomeType***) a ***PartialEq*** implementation means adding to functions (eq ⇔ equality) and (ne ⇔ not equal)

Rust

```rust
impl PartialEq for SomeType {
    fn eq(&self, other: &SomeType) -> bool { … }
    fn ne(&self, other: &SomeType) -> bool { … }
}
```

Enums

The C++ approximation is using operator overloading to do the exact same thing:

Rust

```rust
impl PartialEq for SomeType {
    fn eq(&self, other: &SomeType) -> bool { … }
    fn ne(&self, other: &SomeType) -> bool { … }
}
```

↕

C++

```cpp
class SomeType
{
public:
    bool operator == (const SomeType& other) {...}
    bool operator != (const SomeType& other) {...}
}
```

Enums

So … why ***PartialEQ*** and not just ***EQ*** ?

Well -> lets start with what equality means (or more precisely equivalence).

We define a binary relation `ʘ` as an equivalence relation, if and only if it is:

a) Reflexive ➔ a ʘ a

b) Symmetric ➔ a ʘ b *if and only if* b ʘ a

c) Transitive ➔ *if* a ʘ b *and* b ʘ c *then* a ʘ c

In practice, not all binary relations reflect an equivalence relation (main due to the reflexive characteristics).

Enums

For example, if we are to look on 32 bytes floating value, and in particular to how NaN is represented on IEEE 754 format, then we can deduce the followings:

S	E	E	E	E	E	E	E	E	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c	c
?	1	1	1	1	1	1	1	1	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

- In reality, there are 2^{24} possibilities to write a NaN
- So … if you compare two NaN(s) via a bit check, they may be different, but they are both NaN
- As such, a bit-by-bit comparison between two numbers will not be reflexive (for floating values).
- While there are solutions (such as compare only some bits), adding this type of logic for every float will highly impact the performance.

Enums

Let's see how a simple enum looks like in memory ?

Rust

```rust
#[derive(PartialEq)]
enum Color {
    Red, Green, Blue, White, Black
}
fn main() {
    println!("size of Color = {}",std::mem::size_of::<Color>());
}
```

Output

Size of Color = 1

So … one difference from C/C++ is that outside any other specifications, a simple/small enum looks more like an u8/i8 value than an int (the way it is treated in C/C++).

Enums

Let’s see how a simple enum looks like in memory ?

Rust

```rust
#[derive(PartialEq)]
enum Color { Red, Green, Blue,
             White, Black }
fn main() {
    let mut c = Color::Red;
    if c == Color::White {
        println!("Color is white");
    }
}
```

```asm
sub     rsp,68h
mov     byte ptr [c],0
lea     rcx,[c]
lea     rdx,[offset to a Color::White object]
call    PartialEq::eq
mov     byte ptr [temp_value],al
mov     al,byte ptr [temp_value]
test    al,1
jne     print_something
jmp     end_program
```

- “c” object is in fact an u8 value (see the byte ptr from the assembly code), where Color::Red is associated with value 0

Enums

Let's see how a simple enum looks like in memory ?

Rust

```
#[derive(PartialEq)]
enum Color { Red, Green, Blue,
             White, Black }
fn main() {
    let mut c = Color::Red;
    if c == Color::White {
        println!("Color is white");
    }
}
```

```
sub     rsp,68h
mov     byte ptr [c],0
lea     rcx,[c]
lea     rdx,[offset to a Color::White object]
call    PartialEq::eq
mov     byte ptr [temp_value],al
mov     al,byte ptr [temp_value]
test    al,1
jne     print_something
jmp     end_program
```

- Next we need to call eq method from PartialEQ with two parameters (self – denoted by RCX register that holds the address of "c") and a reference (offset of another object of type Color to compare against)

Enums

Offset	Value
Addr of Color::White	3

Let’s see how a simple enum looks like in memory ?

Rust

```
#[derive(PartialEq)]
enum Color { Red, Green, Blue,
             White, Black }
fn main() {
    let mut c = Color::Red;
    if c == Color::White {
        println!("Color is white");
    }
}
```

```
sub     rsp,68h
mov     byte ptr [c],0
lea     rcx,[c]
lea     rdx,[offset to a Color::White object]
call    PartialEq::eq
mov     byte ptr [temp_value],al
mov     al,byte ptr [temp_value]
test    al,1
jne     print_something
jmp     end_program
```

- This offset points to a static address where the value “3” (u8) is located. Why 3? Well → Red = 0, Green = 1, Blue = 2, **White = 3** ... and since we compare “c” with Color::White, the object has to be a “3”

Enums

Let’s see how a simple enum looks like in memory ?

```
    sub       rsp,28h
    mov       qword ptr [rsp+8],rcx // self
    mov       qword ptr [rsp+10h],rdx // other
    movzx     eax,byte ptr [rcx]
    mov       qword ptr [rsp+18h],rax
    movzx     ecx,byte ptr [rdx]
    mov       qword ptr [rsp+20h],rcx
    cmp       rax,rcx
    je        if_then_part
    mov       byte ptr [return_value],0 // false
    jmp       end_if_label
if_then_part:
    mov       byte ptr [return_value],1 // true
end_if_label:
    mov       al,byte ptr [return_value]
    and       al,1
    movzx     eax,al
    add       rsp,28h
    ret
```

```
sub     rsp,68h
mov     byte ptr [c],0
lea     rcx,[c]
lea     rdx,[offset to a Color::White object]
call    PartialEq::eq
mov     byte ptr [temp_value],al
mov     al,byte ptr [temp_value]
test    al,1
jne     print_something
jmp     end_program
```

As we can see all this function is doing is to compare the first byte from the two objects.

Enums

Let's see how a simple enum looks like in memory ?

```
    sub       rsp,28h
    mov       qword ptr [rsp+8],rcx // self
    mov       qword ptr [rsp+10h],rdx // other
    movzx     eax,byte ptr [rcx]
    mov       qword ptr [rsp+18h],rax
    movzx     ecx,byte ptr [rdx]
    mov       qword ptr [rsp+20h],rcx
    cmp       rax,rcx
    je        if_then_part
    mov       byte ptr [return_value],0 // false
    jmp       end_if_label
if_then_part:
    mov       byte ptr [return_value],1 // true
end_if_label:
    mov       al,byte ptr [return_value]
    and       al,1
    movzx     eax,al
    add       rsp,28h
    ret
```

```
sub     rsp,68h
mov     byte ptr [c],0
lea     rcx,[c]
lea     rdx,[offset to a Color::White object]
call    PartialEq::eq
mov     byte ptr [temp_value],al
mov     al,byte ptr [temp_value]
test    al,1
jne     print_something
jmp     end_program
```

Notice that "and al,1". This instruction makes sure that the value of al is either 1 or 0. This is a clear indicator that the result of this function is a bool value (with 1=true and 0=false).

Enums

Let's see how a simple enum looks like in memory ?

Rust

```rust
#[derive(PartialEq)]
enum Color { Red, Green, Blue,
             White, Black }
fn main() {
    let mut c = Color::Red;
    if c == Color::White {
        println!("Color is white");
    }
}
```

```
sub     rsp,68h
mov     byte ptr [c],0
lea     rcx,[c]
lea     rdx,[offset to a Color::White object]
call    PartialEq::eq
mov     byte ptr [temp_value],al
mov     al,byte ptr [temp_value]
test    al,1
jne     print_something
jmp     end_program
```

- Finally, we check the bool value returned from the previous step to see if it is true (value 1) or not

Enums

This means that a proper C++ code that reflects this Rust code is:

C++ equivalent for Rust Color enum

```
class Color {
    uint8_t value;
    Color(uint8_t v) : value(v) {}
public:
    static const Color Red,Green,Blue,White,Black;
    bool operator==(const Color& v) { return value == v.value; }
};
const Color Color::Red  (0);
const Color Color::Green(1);
const Color Color::Blue (2);
const Color Color::White(3);
const Color Color::Black(4);

void main() {
    Color c = Color::Red;
    if (c == Color::White) printf("Color is White");
}
```

Enums

So … why are Rust ***enums*** built like this ? Is there a specific advantage they get by doing this ?

Well … yes → but first, let's see some examples (Rust/C++)

1. **Enum with just some variants**

Rust

```rust
#[derive(PartialEq)]
enum Color { Red, Green, Blue,
             White, Black }
```

C++

```cpp
enum class Color {
    Red,Green,Blue,White,Black
};
```

Enums

So … why are Rust ***enums*** build like this ? Is there a specific advantage they get by doing this ?

Well … yes → but first, let's see some examples (Rust/C++)

2. Enum with just some variants with specific values

Rust

```rust
#[derive(PartialEq)]
enum Color {
    Red = 2,
    Green = 10,
    Blue,
    White, Black
}
```

C++

```cpp
enum class Color {
        Red   = 2,
        Green = 10,
        Blue,
        White,
        Black
};
```

Enums

So … why are Rust ***enums*** build like this ? Is there a specific advantage they get by doing this ?

Well … yes → but first, let's see some examples (Rust/C++)

3. Enum with a specific type (e.g. int)

Rust

```
#[derive(PartialEq)]
#[repr(i32)]
enum Color {
    Red = 2,
    Green = 10,
    Blue, White, Black
}
```

C++

```
enum class Color : int {
        Red   = 2,
        Green = 10,
        Blue,
        White,
        Black
};
```

Enums

So … why are Rust ***enums*** build like this ? Is there a specific advantage they get by doing this ?

Well … yes → but first, let's see some examples (Rust/C++)

3. Enum with a specific type (e.g. int)

Rust

```rust
#[derive(PartialE
#[repr(i32)]
enum Color {
    Red = 2,
    Green = 10,
    Blue, White, Black
}
```

To specify a certain type/representation behind the discriminant of an enum, use the following format: **#[repr(type)]**
where type can be u8/i8, u16/i16, … u128/i128, usize/isize.
Currently, u128/i128 layout is unstable !

```cpp
        White,
        Black
};
```

Enums

So … why are Rust ***enums*** build like this ? Is there a specific advantage they get by doing this ?

Well … yes → but first, let's see some examples (Rust/C++)

3. Enum with a specific type (e.g. int)

Rust

```rust
#[repr(bool)]
enum Color {
    Red,
    Green,
}
```

Error

```
error[E0552]: unrecognized representation hint
 --> src\main.rs:1:8
  |
1 | #[repr(bool)]
  |        ^^^^
= help: valid reprs are `C`, `align`, `packed`, `transparent`, `simd`, `i8`, `u8`,
`i16`, `u16`, `i32`, `u32`, `i64`, `u64`, `i128`, `u128`, `isize`, `usize`
```

Not all representation are allowed ! Only numerical (integer) representation can be used for an ***enum*** discriminant.

Enums

So … why are Rust ***enums*** build like this ? Is there a specific advantage they get by doing this ?

Well … yes → but first, let's see some examples (Rust/C++)

3. Enum with a specific type (e.g. int)

Rust

```rust
#[repr(i32)]
enum Color {
    Red = 1,Green = 3,Blue = 15
}
fn main() {
    let c = Color::Green;
    let i = c as i32;
    println!("i={i}");
}
```

Output

```
i=3
```

You can also use ***as*** to convert an enum that has a numeric representation to its numerical value.

Enums

So … why are Rust ***enums*** build like this ? Is there a specific advantage they get by doing this ?

Well … yes → but first, let's see some examples (Rust/C++)

4. Bitflags

- Bitflags are NOT possible in Rust (with the standard library and functionality)
- There are however different crates (e.g EnumBitFlags , bitflags) that provides this functionality through some Rust macros
- In C++ bitflags over ***enums*** are easily implemented via **friend** functions that implement operators like ||, &&, !, etc.

Enums

So … why are Rust ***enums*** build like this ? Is there a specific advantage they get by doing this ?

This is the main case why enums are build like this (flexibility).

5. Multiple data member types *enums*

- Since an enum in Rust is more like a class than a classical enum from C, there is no reason to limit the variants to a specific type
- In C/C++, all variants from an ***enum*** have the same type (usually int if something else is not provided). This limitation can be overcome if we use classes with static const values instead of ***enums***.
- In Rust, however, we can create different variants of different types ➔

Rust

```
enum <Name> {
        Variant₁(type₁),
        Variant₂(type₂,…)
        …
        Variantₙ
}
```

Enums

Overview (Rust *enums* vs C/C++ *enums*)

	Rust	**C++**
Simple enums	Yes	Yes
Simple enums mapped to a specific type	Yes	Yes
Simple enums with different values (of the same type) associated to each variant	Yes	Yes
Enums that work as a bitflag	No*	Yes**
Enums with value of different types	Yes	No***

*   There are some crates such as ***EnumBitFlags***, ***bitflags*** *that solves this problem via macros*
**   Requires the usage of ***friend*** keyword do overwrite operators such as || , && , etc
*** Can not be done with classical enums, but fully supported through **std::variant**

Enums

Let's see some example of ***enums*** with variant of multiple types.

Rust

```
#[derive(PartialEq, Debug)]
enum Values {
    Integer(i32),
    Float(f32),
    Character(char)
}
fn main() {
    let i = Values::Integer(10);
    let f = Values::Float(1.2);
    let c = Values::Character('a');
    println!("{:?},{:?},{:?}",i,f,c);
}
```

Output

```
Integer(10),Float(1.2),Character('a')
```

The reason for the ***Debug*** derivation is to provide ***println!*** macro some sort of reflection that can be used to print ***enum*** values.

Enums

Let's see some example of ***enums*** with variant of multiple types.

Rust

```
enum Values {
    Integer(i32),
    Float(f32),
    Character(char),
}
fn main() {
    let i = Values::Integer(10);
    println!("i is {}", i as i32);
}
```

Error

```
error[E0605]: non-primitive cast: `Values` as `i32`
  --> src\main.rs:15:25
   |
15 |     println!("i is {}", i as i32);
   |                         ^^^^^^^^ an `as` expression can only
be used to convert between primitive types or to coerce to a
specific trait object
```

So … if this is not possible:
How can we tell if "i" is a Values::Integer, Values::Float or Values::Character ?

Enums

Let’s see some example of ***enums*** with variant of multiple types.

Rust

```
enum Values {
    Integer(i32),
    Float(f32),
    Character(char),
}
```

When we are trying to find the underlying type of one of the variants from an enum, we often use the term ***discriminant****.* The discriminant is often a numerical value that specifies the type (for example in this example the ***discriminant*** could be 0 for Integer, 1 for Float and 2 for Character).

Enums

The solution is to use match to validate the type of an object from Values:

Rust

```
enum Values {
    Integer(i32),
    Float(f32),
    Character(char),
}
fn extract_integer(v: &Values) -> i32 {
    match v {
        Values::Integer(ivalue) => return *ivalue,
        _ => return -1,
    }
}
fn main() {
    let i = Values::Integer(10);
    println!("i is {}", extract_integer(&i));
}
```

Output

```
i is 10
```

Enums

The same can be obtained by implementing a method into an enum:

Rust

```rust
enum Values {
    Integer(i32),
    Float(f32),
    Character(char),
}
impl Values {
    fn get_int(&self) -> i32 {
        match self {
            Values::Integer(ivalue) => return *ivalue,
            _ => return -1,
        }
    }
}
fn main() {
    let i = Values::Integer(10);
    println!("i is {}", i.get_int());
}
```

Output

```
i is 10
```

Enums

Alternatively, we can use std::mem::discriminant(...) to check if two values from the same enum have the same discriminant.

Rust

```
use std::mem;
enum Values {
    Integer(i32),
    Real(f64),
}
fn main() {
    let a = Values::Integer(10);
    let b = Values::Integer(20);
    let c = Values::Real(1.2);
    if mem::discriminant(&a) == mem::discriminant(&b) {
        println!("'a' and 'b' are of the same variant type !");
    }
    if mem::discriminant(&a) != mem::discriminant(&c) {
        println!("'a' and 'c' are not the same variant type !");
    }
}
```

Output

```
'a' and 'b' are of the same variant type !
'a' and 'c' are not the same variant type !
```

Enums

The previous example can be adjusted to find out if a value of an enum is of a specific type. Keep in mind that this method, while it works implies creating a temporary object to be used for comparison !

Rust

```rust
enum Values {
    Integer(i32),
    Float(f32),
    Character(char),
}
impl Values {
    fn is_int(&self) -> bool {
        std::mem::discriminant(self) == std::mem::discriminant(&Values::Integer(0))
    }
}
fn main() {
    let i = Values::Integer(10);
    println!("i is int: {}", i.is_int());
}
```

Output

```
i is int: true
```

Enums

A variant from an enum can also be a set of values. The next example creates two version of an IpAddress (v4 and v6).

Rust

```
#[derive(Debug)]
enum IpAddress {
    v4(u8, u8, u8, u8),
    v6(u16, u16, u16, u16, u16, u16),
}
fn main() {
    let ip_1 = IpAddress::v4(192, 168, 0, 1);
    let ip_2 = IpAddress::v6(0x2010, 0x1234, 0x00FF, 0x0000, 0x0000, 0xFF12);
    println!("{:?}, {:?}", ip_1, ip_2);
}
```

Output

```
v4(192, 168, 0, 1), v6(8208, 4660, 255, 0, 0, 65298)
```

Enums

Let’s see some example of ***enums*** with variant of multiple types.

Rust

```
#[derive(PartialEq, Debug)]
enum Values {
    Integer(i32),
    Float(f32),
}
fn main() {
    let i1 = Values::Integer(10);
    let i2 = Values::Integer(20);
    if i1 == i2 {
        println!("Equal integers")
    } else {
        println!("Different integers");
    }
}
```

Output

```
Different integers
```

When comparing two ***enum*** variants, Rust will compare both their types and their value (if present).

Enums

Let's see some example of ***enums*** with variant of multiple types.

Rust

```
#[derive(PartialEq, Debug)]
enum Values {
    Integer(i32),
    Float(f32),
}
fn main() {
    let i1 = Values::Integer(10);
    let i2 = Values::Integer(10);
    if i1 == i2 {
        println!("Equal integers")
    } else {
        println!("Different integers");
    }
}
```

Output

```
Equal integers
```

When comparing two ***enum*** variants, Rust will compare both their types and their value (if present).

Enums

Let's see some example of ***enums*** with variant of multiple types.

Rust

```
#[derive(PartialEq, Debug)]
enum Values {
    Integer(i32),
    Float(f32),
}
fn main() {
    let i1 = Values::Integer(10);
    let i2 = Values::Float(2.0);
    if i1 == i2 {
        println!("Equal integers")
    } else {
        println!("Different integers");
    }
}
```

Output

```
Different integers
```

When comparing two ***enum*** variants, Rust will compare both their types and their value (if present).

Enums

Let's see some example of ***enums*** with variant of multiple types.

Rust

```
#[derive(PartialEq, Debug)]
enum Values {
    Integer(i32),
    Float(f32),
}
fn main() {
    let i1 = Values::Integer(10);
    let i2 = Values::Float(10.0);
    if i1 == i2 {
        println!("Equal integers")
    } else {
        println!("Different integers");
    }
}
```

Output

Different integers

Even if the value is the same (10) since there are different types (Integer and Float) they will not be equal.

Enums

Let's see some example of ***enums*** with variant of multiple types.

Rust

```
#[derive(PartialEq, Debug)]
enum Values {
    Integer(i32),
    Bool(bool),
    Real(f64),
}
fn main() {
    println!("Size of Values = {}", std::mem::size_of::<Values>());
}
```

Output

```
Size of Values = 16
```

So … why is the size of Values 16 bytes ?

- an **Integer** is 4 bytes
- a **bool** is one bytes
- a **float64** is 8 bytes

Enums

Let's see some example of ***enums*** with variant of multiple types.

Rust

```rust
#[derive(PartialEq, Debug)]
enum Values {
    Integer(i32),
    Bool(bool),
    Real(f64),
}
fn main() {
    let i = Values::Integer(10);
    let b = Values::Bool(true);
    let r = Values::Real(1.234);
}
```

Let's see how i, b, or r look in memory.

Enums

Let's see some example of ***enums*** with variant of multiple types.

Rust

```
#[derive(PartialEq, Debug)]
enum Values {
    Integer(i32),
    Bool(bool),
    Real(f64),
}
fn main() {
    let i = Values::Integer(10);
    let b = Values::Bool(true);
    let r = Values::Real(1.234);
}
```

Idx				Integer											
+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11	+12	+13	+14	+15
0	?	?	?	**10**	**0**	**0**	**0**	?	?	?	?	?	?	?	?

Let's see how i, b, or r look in memory.

Enums

Let's see some example of ***enums*** with variant of multiple types.

Rust

```
#[derive(PartialEq, Debug)]
enum Values {
    Integer(i32),
    Bool(bool),
    Real(f64),
}
fn main() {
    let i = Values::Integer(10);
    let b = Values::Bool(true);
    let r = Values::Real(1.234);
}
```

Idx	B														
+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11	+12	+13	+14	+15
1	**1**	?	?	?	?	?	?	?	?	?	?	?	?	?	?

Let's see how i, b, or r look in memory.

Enums

Let's see some example of ***enums*** with variant of multiple types.

Rust

```rust
#[derive(PartialEq, Debug)]
enum Values {
    Integer(i32),
    Bool(bool),
    Real(f64),
}
fn main() {
    let i = Values::Integer(10);
    let b = Values::Bool(true);
    let r = Values::Real(1.234);
}
```

Idx								Real							
+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11	+12	+13	+14	+15
2	?	?	?	?	?	?	?	**58**	**39**	**B4**	**C8**	**76**	**BE**	**F3**	**3F**

Let's see how i, b, or r look in memory.

Enums

This means that in reality, this is more like a union than multiple data members within the same class..

C++ aproximation

```
union Values
{
    uint8_t index;
    union {
        struct Integer {
            uint8_t index;
            int32_t value;
        };
        struct Bool {
            uint8_t index;
            bool value;
        };
        struct Real {
            uint8_t index;
            double value;
        };
    };
}
```

↔

C++ (2017 and beyond)

```
std::variant<int32_t,bool,double>
```

Where Values::index is:

0 → if the underline type from the union is **Integer**

1 → if the underline type from the union is **Bool**

2 → if the underline type from the union is **Real**

Error management

Error management

For every program (regardless of the language it is written in) there are three situations that require error management:

Type	Description	What to do
Compile Error	Usually when some semantics of the language are incorrect.	Repair the error and compile again
Run-time Error (manageable)	An error that can be managed by the program (e.g. we are trying to connect to a database but the internet connection is unavailable)	In this case, we need to have a logic within the program that treats this error (e.g. pops up a message and then wait for the internet connection to be available)
Run-time Error (critical)	An error that by its nature stops the execution of the program (e.g. a game can not start if the graphical driver is not working)	Nothing. These are the cases where the program just stops.

Error management

In reality, run-time errors can be:

1. **Treated** ➔ meaning that there is a specific code that treats an error (a specific execution flow that takes into consideration various cases where errors might occur)
2. **Un-treated** ➔ these are dangerous situation that might lead to program crashing or undefined behavior

A well written program falls into category 1 (meaning that the programmers of that program were very careful about various situation that might occur and can produce an error).

Error management

Let’s see a C/C++ example and discuss how an error might be treated:

C

```c
int div(const char * n1, const char* n2) {
    return atoi(n1)/atoi(n2);
}
void main(char** argv, int argc) {
    printf("Result is: %d",div(argv[1],argv[2]));
}
```

We will focus on div function, and not on the problems from the main function (e.g. not enough parameters).

Error management

What potential problems can we spot on div function ?

C

```c
int div(const char * n1, const char* n2) {
    return atoi(n1)/atoi(n2);
}
```

1. "n1" or "n2" can be null pointers (e.g. for example if the command line arguments are less than 2)
2. "n1" or "n2" can be invalid numbers (we are working with string, so there is no guarantee that either n1 or n2 respect a valid numerical format;
3. "n2" could be a valid number, but it is 0 and division by 0 will produce an error.

Error management

So how can we change function “div” to treat errors ?

1. Change the signature of function “div” to return either true (if the division was successful) or false otherwise and put the actual result in a reference or pointer.

C

```c
bool div(const char * n1, const char* n2, int* result) { … }
```

C

```c
bool div(const char * n1, const char* n2, int& result) { … }
```

It is recommended to use a reference as we don’t need to add an extra validation to check if the result pointer is valid.

Error management

So how can we change function “div” to treat errors ?

1. Change the signature of function “div” to return either true (if the division was successful) or false otherwise and put the actual result in a reference or pointer.
   The usage of such a function will be as follows:

C
```
void main(char** argv, int argc) {
    int result;
    if (div(argv[1],argv[2],result)==true) {
        printf("Result is: %d",result);
    } else {
        // error case
    }
}
```

Error management

So how can we change function “div” to treat errors ?

1. Change the signature of function “div” to return either true (if the division was successful) or false otherwise and put the actual result in a reference or pointer.

PRO	CONS
• Easy to write (regardless of the language)	• We need references (this means that every function call should be preceded by a variable definition where the result will be put) • Bool type is not necessarily associated with errors and as such some results might be misleading. • We don’t know the actual error (just that there is one).

Error management

So how can we change function "div" to treat errors ?

2. Change the signature of function "div" to return an error code (an ***int*** value that if set to 0 (or other constant) means no error, and otherwise means an error code). Similar to precedent case, the actual result should be put in a reference or pointer.

C

```c
int div(const char * n1, const char* n2, int* result) { … }
```

C

```c
int div(const char * n1, const char* n2, int& result) { … }
```

Error management

So how can we change function “div” to treat errors ?

2. Change the signature of function “div” to return an error code (an int value that if set to 0 (or other constant) means no error, and otherwise means an error code). One possible usage:

C

```c
void main(char** argv, int argc) {
    int result;
    int error = div(argv[1],argv[2],result);
    if (error == 0) {
        printf("Result is: %d",result);
    } else {
        switch (error) {
            case 1: printf("First parameter is null !"); break;
            case 2: printf("Second parameter is null !"); break;
            ….
        }
    }
}
```

Error management

So how can we change function “div” to treat errors ?

2. Change the signature of function “div” to return an error code (an int value that if set to 0 (or other constant) means no error, and otherwise means an error code). One possible usage:

PRO	CONS
• Easy to write (regardless of the language) • We know the error and we can act on it	• We need references (this means that every function call should be preceded by a variable definition where the result will be put) • Int (or numerical) types are not necessary associated with errors and as such some results might be misleading.

Error management

So how can we change function "div" to treat errors ?

3. Use exception (meaning that div function signature will not be changed). Instead, whenever an error occurs, an exception will be thrown.

C

```c
int div(const char * n1, const char* n2) { … }
```

This is a different approach that starts from the assumption that a function signature should reflect its purpose and not its error handling mechanisms.

Error management

So how can we change function “div” to treat errors ?

3. Use exception (meaning that div function signature will not be changed). Instead, whenever an error occurs, an exception will be thrown. Possible usage:

C

```
void main(char** argv, int argc) {
    try {
        printf("Result is: %d",div(argv[1],argv[2]));
    }
    catch (DivisionBy0Error)
    {
        printf("Division by 0");
    }
    catch (...)
    {
        printf("other error")
    }
}
```

Error management

So how can we change function “div” to treat errors ?

3. Use exception (meaning that div function signature will not be changed). Instead, whenever an error occurs, an exception will be thrown.

PRO	CONS
• Easy to write (regardless of the language) • We know the error and we can act on it	• Not really linear in terms of code execution • Memory allocation might not be cleared • Can’t really be enforced (someone can decide not to use it, because a try...catch block is not necessary to read the result of a function.

Error management

So how can we change function “div” to treat errors ?

4. Use an error specific type that holds both the value and the error/error code. This is a more modern approach of the error management problem. A definition (for C++ language) looks like this:

C++17 and beyond

```
std::optional<int> div(const char * n1, const char* n2) { … }
```

This type was introduced in C++ with the 2017 standard.

Error management

So how can we change function "div" to treat errors ?

4. Use an error specific type that holds both the value and the error/error code. This is a more modern approach of the error management problem. A possible usage:

C++ (2017+ standard)

```
void main(char** argv, int argc) {
    auto res = div(argv[1],argv[2]);
    if (res.has_value()){
        printf("Result is: %d",res.value());
    } else {
        // process error
    }
}
```

Notice that the code is quite small and the res variable incapsulates both the value and the error.

Error management

So how can we change function “div” to treat errors ?

4. Use an error specific type that holds both the value and the error/error code. This is a more modern approach of the error management problem.

PRO	CONS
• Easy to write (regardless of the language) • We know the error and we can act on it • Linear programming • Enforceable (you can not get the result without knowing the error as well) • This is a type designed for error management so it has no double interpretation	• Might require some adjustments in how someone programs if he/she are used with an error management similar to cases 1,2 or 3

Error management

A general observation on these four cases:

- Older languages (e.g. C) usually use cases 1 or 2 (e.g. Windows API (case 1), Linux API (case 2))
- Newer languages (C++, Java, C#, Python, etc) usually support cases 1 to 3. The potential risk here is that none of these cases are enforceable (meaning that someone might write a program and use all 3 techniques to propagate errors)
- Modern languages (e.g. C++17/C++20/C++23, Rust) support the 4^{th} method as well.

Error management

Type	C	C++	Rust
Case 1 (return True/False)	Yes	Yes	Yes
Case 2 (return error code)	Yes	Yes	Yes
Case 3 (exceptions)	-	Yes	-
Case 4 (return True+value for success, False otherwise)	-	std::optional	Option
Case 4 (return value for success, error information otherwise)	-	std:expected std::variant	Result

Error management

- What differentiate Rust from the rest of the languages that implement the 4th method is that Rust **does not implement exceptions.**

- This means that a programmer can decide to use either case 1,2 or 4 in Rust if he/she wants to return an error.

Error management

Error management in Rust is done via:

1. panic macro (if we want to immediately exit a program)
2. Option template/generic type (if we want to return a value or nothing – the latter meaning that an error has occurred)
3. Result template/generic type (if we want to return a value or an error that explains what happened).

Error management (panic)

Error management (panic)

A "**panic**" is a critical runtime-error that you can not recover from.

In Rust, these situation can be encountered in two scenarios:

1. The execution flow reach a point where the outcome cannot be computed in a deterministic way (e.g. a possible undefined behavior). Stopping the execution at this point will provide more information for a developer to fix the actual issue (e.g. a heap overflow). In Rust this is done at thread level (meaning it will stop the current thread, not the entire process).
2. The logic of the problem / its purpose can not be served anymore, and the programmer decides to stop the problem at the current point of the execution.

Error management (panic)

Rust provides a macro (called **panic!**) that can be used to abord a program immediately. ***panic!*** macro has two forms:

```
panic! ();
panic! (message);
```

Rust

```rust
fn main() {
    let r1 = 20;
    let r2 = 10;
    if r1 > r2 {
        panic!("Expecting r1={r1} to be smaller than r2={r2}");
    }
}
```

Panic (runtime – v 1.61.0)

```
thread 'main' panicked at 'Expecting r1=20 to be smaller than r2=10', src\main.rs:7:9
stack backtrace:
   0: std::panicking::begin_panic_handler
             at /rustc/fe5b13d681f25ee6474be29d748c65adcd91f69e\/library\std\src\panicking.rs:584
   1: core::panicking::panic_fmt
             at /rustc/fe5b13d681f25ee6474be29d748c65adcd91f69e\/library\core\src\panicking.rs:143
   2: first::main
             at .\src\main.rs:7
   …
```

Error management (panic)

Panic errors can also be triggered if the programmer is trying to perform an operation with an undefined result:

Rust

```rust
fn main() {
    let v = [10,20,30];
    let x = v[v[0]];
    println!("x={x}");
}
```

Panic (runtime) – for Rust 1.61.0

```
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 10', src\main.rs:3:13
stack backtrace:
   0: std::panicking::begin_panic_handler
             at /rustc/fe5b13d681f25ee6474be29d748c65adcd91f69e\/library\std\src\panicking.rs:584
   1: core::panicking::panic_fmt
             at /rustc/fe5b13d681f25ee6474be29d748c65adcd91f69e\/library\core\src\panicking.rs:143
   2: core::panicking::panic_bounds_check
   ……………………
```

In this case, there is an attempt to read a value from a vector outside its bounds. While it is possible that the memory from the offset “**10**” (the value of v[0]) is accessible (e.g. a value on the stack) the outcome is ***undetermined*** and it is better to cause an abord at this point that try to understand an error thrown by an incorrect value of “x” later.

Error management (panic)

Its also important to notice that Rust tries to identify this kind of error from the compile phase. Let's compare the next three cases:

Rust

```rust
fn main() {
    let v = [10,20,30];
    let x = v[10];
    println!("x={x}");
}
```

Error

```
error: this operation will panic
at runtime
 --> src\main.rs:3:13
  |
3 |     let x = v[10];
  |             ^^^^^ index out
of bounds: the length is 3 but
the index is 10
```

That's obvious (v[10] is clearly out of bounds for an array of 3 elements).

Error management (panic)

Its also important to notice that Rust tries to identify this kind of error from the compile phase. Let's compare the next three cases:

Rust

```rust
fn main() {
    let v = [10,20,30];
    let x = v[10];
    println!("x={x}");
}
```

Rust (1.61.0)

```rust
fn main() {
    let v = [10,20,30];
    let x = v[v[0]];
    println!("x={x}");
}
```

Error

```
error: this operation will panic
at runtime
 --> src\main.rs:3:13
  |
3 |     let x = v[10];
  |             ^^^^^ index out
of bounds: the length is 3 but
the index is 10
```

Panic (runtime – 1.61.0)

```
thread 'main' panicked at 'index
out of bounds: the len is 3 but
the index is 10',
src\main.rs:3:13
```

In this case, Rust **1.61.0** crashes at runtime (it is unable to identify that v[v[0]] = v[10] that is clearly out of bounds)

Error management (panic)

Its also important to notice that Rust tries to identify this kind of error from the compile phase. Let's compare the next three cases:

Rust

```rust
fn main() {
    let v = [10,20,30];
    let x = v[10];
    println!("x={x}");
}
```

Rust (1.61.0)

```rust
fn main() {
    let v = [10,20,30];
    let x = v[v[0]];
    println!("x={x}");
}
```

Rust (1.71.0)

```rust
fn main() {
    let v = [10,20,30];
    let x = v[v[0]];
    println!("x={x}");
}
```

Error

```
the index is 10
```

Panic (runtime – 1.61.0)

Panic (1.71.0)

```
error: this operation will panic
at runtime
 --> src\main.rs:3:13
  |
3 |     let x = v[v[0]];
  |             ^^^^^^^ index out
of bounds: the length is 3 but
the index is 10
```

Notice that this is the exact same code but tested with a different (newer) version of Rust. As it turns out, Rust constantly improves its detection for out of boundery cases. This is why the same case that will trigger a runtime panic for version 1.61.0 will be identified as a compile error for version 1.71.0

Error management (panic)

Rust 1.71.0 seems to be able to identify even more complicated cases (for example when we use more complex equations)

Rust (1.71.0)

```rust
fn main() {
    let v = [10, 20, 30];
    let x = v[v[0]*v[0]/v[1]];
    println!("x={x}");
}
```

Error

```
error: this operation will panic at runtime
 --> src\main.rs:3:13
  |
3 |     let x = v[v[0]*v[0]/v[1]];
  |             ^^^^^^^^^^^^^^^^^ index out of
bounds: the length is 3 but the index is 5
```

The compile makes the correct inference (v[0] = 10, v[1] = 20, v[0] * v[0] / v[1] = 10*10/20 = 100/20 = 5

Error management (panic)

A solution to "trick" rust compiler and make it not detect an out of boundary case is the following:

Rust (1.71.0)

```rust
use std::{time::{SystemTime, Duration}, thread::sleep};
fn main() {
    let start = SystemTime::now();
    sleep(Duration::from_secs(5));
    let dif = start.elapsed().unwrap().as_secs();
    let v = [10, 20, 30];
    let x = v[dif as usize];
    println!("x={x}");
}
```

Error

```
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 5',
src\main.rs:7:13
stack backtrace:
   0: std::panicking::begin_panic_handler
             at
/rustc/8ede3aae28fe6e4d52b38157d7bfe0d3bceef225/library\std\src\panicking.rs:593
   1: core::panicking::panic_fmt
             at
/rustc/8ede3aae28fe6e4d52b38157d7bfe0d3bceef225/library\core\src\panicking.rs:67
```

Error management (panic)

A solution to "trick" rust compiler and make it not detect an out of boundary case is the following:

Rust (1.71.0)

```rust
use std::{time::{SystemTime, Duration}, thread::sleep};
fn main() {
    let start = SystemTime::now();
    sleep(Duration::from_secs(5));
    let dif = start.elapsed().unwrap().as_secs();
    let v = [10, 20, 30];
    let x = v[dif as usize];
    println!("x={x}");
}
```

Notice that we did the following steps:

1. Get the current time
2. Wait (sleep) for 5 seconds
3. Compute the elapsed time

It is obvious that the elapsed time should be 5 and if we use it as index access on a 3 elements array it will produce a panic.

Keep in mind that this solution was tested with Rust 1.71.0/1.80.0

It is possible that future version of Rust might detect this behavior from the compile time and trigger a compile error instead of runtime panic.

Error management (panic)

Rust also has a method (**catch_unwind**) that can be used to capture a panic (similar to what ***try…catch*** mechanism is doing).

However, it is not recommended and if used with C++ exceptions from an exported function, the behavior is undefined.

Error management (panic)

Rust also has a method (**catch_unwind**) that can be used to capture a panic (similar to what ***try…catch*** mechanism is doing).

Rust

```rust
use std::{
    thread::sleep,
    time::{Duration, SystemTime},
};
fn some_function() -> i32 {
    let start = SystemTime::now();
    sleep(Duration::from_secs(5));
    let dif = start.elapsed().unwrap().
    let v = [10, 20, 30];
    let x = v[dif as usize];
    println!("x={x}");
    return x as i32;
}
fn main() {
    let r = std::panic::catch_unwind(some_function);
    if r.is_ok() {
        println!("Function was successful -> return value is: {}", r.unwrap());
    } else {
        println!("Function failed with panic");
    }
}
```

Error

```
thread 'main' panicked at 'index out of bounds: the len is 3 but the index is 5',
src\main.rs:4:13
stack backtrace:
....
Function failed with panic
```

Error management (panic)

Rust also has a method (catch_unwind) that can be used to capture a panic (similar to what try…catch mechanism is doing).

Rust

```rust
fn some_function() -> i32 {
    let v = [1, 2, 3];
    let x = v[v[0]];
    println!("x={x}");
    return x as i32;
}
fn main() {
    let r = std::panic::catch_unwind(some_function);
    if r.is_ok() {
        println!("Function was successful -> return value is: {}", r.unwrap());
    } else {
        println!("Function failed with panic");
    }
}
```

Output

```
x=2
Function was successful -> return value is: 2
```

Error management (Option type)

Error management (Option type)

Rust Option type allows a function to return two scenarios:

- A **false** case (something is not OK) → and no value associated
- A **true** case → the requested value is returned.

Rust (Source: option.rs from core library)

```
pub enum Option<T> {
    #[lang = "None"]
    #[stable(feature = "rust1", since = "1.0.0")]
    None,

    #[lang = "Some"]
    #[stable(feature = "rust1", since = "1.0.0")]
    Some(#[stable(feature = "rust1", since = "1.0.0")] T),
}
```

Simplified view

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

Error management (Option type)

Let's consider the following problem → we would like to write a function that returns a number only if the parameter is odd, or no number if the parameters is even.

Rust

```
fn validate_odd(n: i32) -> Option<i32>
{
    if n % 2 == 1 {
        return Some(n);
    } else {
        return None;
    }
}
```

or

Rust

```
fn validate_odd(n: i32) -> Option<i32>
{
    if n % 2 == 1 {
        Some(n)
    } else {
        None
    }
}
```

Error management (Option type)

Option type has the following methods:

Method	Usage
`fn unwrap(self) -> T`	Returns the value if no error is present or panics otherwise
`fn expect(self, msg: &str) -> T`	Returns the value if no error is present or panics with a specific message otherwise
`fn is_some(&self) -> bool`	True if no error is present, false otherwise
`fn is_none(&self) -> bool`	True if error is present, false otherwise
`fn unwrap_or(self, default: T) -> T`	Returns the value if no error is present or a default value in case of error
`fn unwrap_or_else<F>(self, f: F) -> T`	Returns the value if no error is present or the result of a function in case of error

Error management (Option type)

Let's see some possible usage of validate_odd(...) function:

Rust

```
fn validate_odd(n: i32) -> Option<i32> { ... }
fn main() {
    let r = validate_odd(5);
    if r.is_some() {
        println!("Number is odd: {}", r.unwrap());
    } else {
        println!("Error");
    }
}
```

Output

Number is odd: 5

Error management (Option type)

The same example can be written using the **match** keyword in the following way (this is actual recommended way to check the value of an Option).

Rust

```rust
fn validate_odd(n: i32) -> Option<i32> { … }
fn main() {
    let r = validate_odd(5);
    match r {
        Some(value) => println!("Odd value: {value}"),
        None => println!("Not an odd number"),
    }
}
```

Output

```
Odd value: 5
```

Error management (Option type)

Let’s see some possible usage of validate_odd(…) function:

Rust

```
fn validate_odd(n: i32) -> Option<i32> { … }
fn main() {
    let r = validate_odd(5).unwrap();
    println!("Number is odd: {}",r);
}
```

Output

Number is odd: 5

In this case we expect validate_odd function to work correctly.

Variable **“r”** is of type i32.

Error management (Option type)

Let’s see some possible usage of validate_odd(…) function:

Rust

```
fn validate_odd(n: i32) -> Option<i32> { … }
fn main() {
    let r = validate_odd(4).unwrap();
    println!("Number is odd: {}",r);
}
```

In this case, since 4 is not odd, a panic (runtime) error will be triggered.

Panic (runtime)

```
thread 'main' panicked at 'called `Option::unwrap()` on a `None` value', src\main.rs:11:28
```

Error management (Option type)

Let’s see some possible usage of validate_odd(…) function:

Rust

```
fn validate_odd(n: i32) -> Option<i32> { … }
fn main() {
    let r = validate_odd(4).expect("Expecting a valid odd number");
    println!("Number is odd: {}",r);
}
```

In this case, since 4 is not odd, a panic (runtime) error will be triggered.

Panic (runtime)

```
thread 'main' panicked at 'Expecting a valid odd number', src\main.rs:11:28
```

Error management (Option type)

Let's see some possible usage of validate_odd(…) function:

Rust

```
fn validate_odd(n: i32) -> Option<i32> { … }
fn main() {
    let r = validate_odd(4).unwrap_or(-1);
    println!("Number is odd: {}",r);
}
```

Output

Number is odd: -1

In this case since we have used unwrap_or method, the Option<i32> value is evaluated. Since, 4 is not an odd number, the error case will be triggered and the default value will be returned (in this case, it is -1)

Error management (Option type)

Let’s see some possible usage of validate_odd(…) function:

Rust

```rust
fn validate_odd(n: i32) -> Option<i32> { … }
fn response_function() -> i32 {
    println!("There is an error !");
    return -1;
}
fn main() {
    let r = validate_odd(4).unwrap_or_else(response_function);
    println!("Number is odd: {}",r);
}
```

Output

```
There is an error !
Number is odd: -1
```

Similarly, a function that returns the value in case of error can be used via .unwrap_or_else method.

Error management (Option type)

Let’s consider the following code:

Rust

```
fn main() {
    let mut x: Option<String> = Some(String::from("my string"));
    let mut y: Option<String> = None;
    println!("x={:?}, y={:?}",x,y);
    y = x;
    println!("x={:?}, y={:?}",x,y);
}
```

What is the issue with this piece of code ?

Error management (Option type)

Let’s consider the following code:

Rust

```
fn main() {
    let mut x: Option<String> = Some(String::from("my string"));
    let mut y: Option<String> = None;
    println!("x={:?}, y={:?}",x,y);
    y = x;
    println!("x={:?}, y={:?}",x,y);
}
```

Compile Error

```
error[E0382]: borrow of moved value: `x`
 --> src\main.rs:7:31
  |
3 |     let mut x: Option<String> = Some(String::from("my string"));
  |         ----- move occurs because `x` has type `Option<String>`, which does not implement the `Copy` trait
...
6 |     y = x;
  |         - value moved here
7 |     println!("x={:?}, y={:?}",x,y);
```

Error management (Option type)

Assignments between enums follow the same rules as for other data types. In particular for Option, since String does not implement the Copy trait the ownership of that string is transferred entirely to variable “y” making variable “x” useful.

But what if want to transfer just the String (not the entire variable) and keep the variable “x” but with the variant “None” ? This feature is in particular useful is an Option<T> is part of a structure from where its more complicated to remove it ?

The solution is to use the method .take(&mut self) defined as following: lets assume variable “s” is of type Option<T>; then let s2 = s.take() will have the following effect:

- If s is of type Some(T), then make s2 of type Some(T) and move the T value from s to s2. Then s becomes None
- If s is of type None, then make s2 of type None and do nothing to s

Error management (Option type)

Let’s consider the following code:

Rust

```rust
fn main() {
    let mut x: Option<String> = Some(String::from("my string"));
    let mut y: Option<String> = None;
    println!("x={:?}, y={:?}",x,y);
    y = x.take();
    println!("x={:?}, y={:?}",x,y);
}
```

Output

```
x=Some("my string"), y=None
x=None, y=Some("my string")
```

Notice that after the call y = x.take(), x becomes **None**, and the String from x is transferred to y.

Error management (Option type)

Let’s consider the following code:

Rust

```
fn main() {
    let mut x: Option<String> = Some(String::from("my string"));
    let mut y: Option<String> = None;
    println!("x={:?}, y={:?}",x,y);
    y = x.take();
    println!("x={:?}, y={:?}",x,y);
    y = x.take();
    println!("x={:?}, y={:?}",x,y);
}
```

Output

```
x=Some("my string"), y=None
x=None, y=Some("my string")
x=None, y=None
```

In this case:

- First `y = x.take();` moves “my string” from “x” to “y” and makes “x” None
- Second `y = x.take();` has nothing to move from “x” (as it is already None) and as such becomes None as well.

Error management (Option type)

How does an ***Option<T>*** looks in memory ?

The simplest way is to consider it as a structure with two fields (a bool one and one of type T):

Field	Type	Usage
ok	bool (or an aligned number)	If this field is 1 (true) than the field value is correct and available, otherwise it is not
value	T (the type from the template/generics)	The actual value (only if field ok is 1 (true))

However, for some cases this template only contains the value. Since a reference is never null, an Option<reference> in Rust does not need the ok field, and as such it is guaranteed to be of the same size as the size of a reference !

Error management (Option type)

Let's see the sizes of an Option in memory. For that we will use the following standard command (std::mem::size_of) , that is an equivalent of ***sizeof*** keyword from C/C++.

Rust

```
fn main() {
    println!("Size of Option<usize> = {}", std::mem::size_of::<usize>());
    println!("Size of Option<i32> = {}", std::mem::size_of::<Option<i32>>());
    println!("Size of Option<i64> = {}", std::mem::size_of::<Option<i64>>());
    println!("Size of Option<&str> = {}", std::mem::size_of::<Option<&str>>());
    println!("Size of Option<Box<i32>> = {}", std::mem::size_of::<Box<i32>>());
}
```

Output

```
Size of Option<usize> = 8
Size of Option<i32> = 8
Size of Option<i64> = 16
Size of Option<&str> = 16
Size of Option<Box<i32>> = 8
```

Error management (Option type)

Let's discuss the previous results.

Type	Size	Explanation
usize	8	It can be either 2,4 or 8 (depending on the architecture). In this case it is a x64 architecture (meaning the size of 8).
Option<i32>	8	Two field member (first field → 4 bytes (with alignment) is the **ok** part, second field is an actual i32 value)
Option<i64>	16	Two field member (first field → 8 bytes (with alignment) is the **ok** part, second field is an actual i64 value)
Option<&str>	16	One field (the actual str value). If null than None value is considered. Keep in mind that size_of(str) = 16 (pointer+size), both 8 bytes
Option<Box<i32>>	8	One field (the pointer to an i32 value). If null, than None value is considered.

Error management (Option type)

Now let’s see how Option handles memory for an enumeration:

Rust

```rust
enum Color {
    Red, Green, Blue
}
fn main() {
    println!("Size of Color         = {}", std::mem::size_of::<Color>());
    println!("Size of Option<Color> = {}", std::mem::size_of::<Option<Color>>());
}
```

Output

```
Size of Color         = 1
Size of Option<Color> = 1
```

Notice that ***Option<Color>*** and ***Color*** have the same memory size.

Let’s see why this happens.

Error management (Option type)

Let's evaluate how memory looks like in the following cases:

Rust

```rust
enum Color {
    Red = 2, Green = 4, Blue = 10
}
fn main() {
    let c = Color::Red;
    let o1 = Some(Color::Green);
    let mut o2: Option<Color> = None;
    o2 = Some(Color::Blue);
}
```

Variable	Type	Value	Memory (Hex)

Error management (Option type)

Let's evaluate how memory looks like in the following cases:

Rust

```
enum Color {
    Red = 2, Green = 4, Blue = 10
}
fn main() {
    let c = Color::Red;
    let o1 = Some(Color::Green);
    let mut o2: Option<Color> = None;
    o2 = Some(Color::Blue);
}
```

Variable	Type	Value	Memory (Hex)
c	Color	Color::Red	02

As Color is an ***enum***, its size is considered one byte and values Red, Green and Blue are mapped in this byte.

Error management (Option type)

Let's evaluate how memory looks like in the following cases:

Rust

```rust
enum Color {
    Red = 2, Green = 4, Blue = 10
}
fn main() {
    let c = Color::Red;
    let o1 = Some(Color::Green);
    let mut o2: Option<Color> = None;
    o2 = Some(Color::Blue);
}
```

Variable	Type	Value	Memory (Hex)
c	Color	Color::Red	02
o1	Option<Color>	Some(Color::Green)	04

Notice that "o1" has the same memory representation as with what Color type has (meaning that Color::Green looks identical in memory with Some(Color::Green)).

In this case, the question is how do we differentiate between None and Some (in case of Color) ?

Error management (Option type)

Let's evaluate how memory looks like in the following cases:

Rust

```
enum Color {
    Red = 2, Green = 4, Blue = 10
}
fn main() {
    let c = Color::Red;
    let o1 = Some(Color::Green);
    let mut o2: Option<Color> = None;
    o2 = Some(Color::Blue);
}
```

Variable	Type	Value	Memory (Hex)
c	Color	Color::Red	02
o1	Option<Color>	Some(Color::Green)	04
o2	Option<Color>	None	**01**

As it turns out, Rust searches an invalid value that can be store on a byte (the same size as what Color has) and uses that value to represent None. Since ***Color*** is using 2, 4 and 10, then value 1 (Hex: 01) is unused and as such Rust can use this invalid value to represent None. **As such, the size of an enum and of an Option<enum> will be the same.**

Error management (Option type)

The same logic applies for enums with multiple types:

Rust

```
#[derive(Debug)]
enum Values {
    Integer(i32),
    Float(f32),
    Bool(bool),
    Double(f64)
}
fn main() {
    let i = Values::Integer(10);
    let d = Values::Double(1.5);
    let n: Option<Values> = None;
    println!("{:?},{:?},{:?}",i,d,n);
}
```

Output

Integer(10),Double(1.5),None

In this case:

- **Values::Integer** → will use the discriminant 0
- **Values::Float** → will use the discriminant 1
- **Values::Bool** → will use the discriminant 2
- **Values::Double** → will use the discriminant 3

As such, the first free (invalid) value is 4. As a result, "n" will have the same size as Values, but its discriminant will be 4 (an invalid value to represent ***None***).

Error management (Option type)

There are some exception cases to this type of optimization:

Rust

```
#[derive(Debug)]
enum MultiValueEnum {
    Value_0,
    //Value_1 … Value_252
    Value_253,
    Value_254,
}
fn main() {
    println!("Size of MultiValueEnum = {}", std::mem::size_of::<MultiValueEnum>());
    println!("Size of Option<MultiValueEnum> = {}", std::mem::size_of::<Option<MultiValueEnum>>());
}
```

Output

```
Size of MultiValueEnum = 1
Size of Option<MultiValueEnum> = 1
```

In this case, MultiValueEnum has ***254*** values. As such it is still possible for Rust to find an invalid value (**255**) to be used for **None** cases. As such the size of the enum and Option<…> are the same (one byte).

Error management (Option type)

There are some exception cases to this type of optimization:

Rust

```
#[derive(Debug)]
enum MultiValueEnum {
    Value_0,
    //Value_1 … Value_252
    Value_253,
    Value_254,
    Value_255,
}
fn main() {
    println!("Size of MultiValueEnum = {}", std::mem::size_of::<MultiValueEnum>());
    println!("Size of Option<MultiValueEnum> = {}", std::mem::size_of::<Option<MultiValueEnum>>());
}
```

Output

```
Size of MultiValueEnum = 1
Size of Option<MultiValueEnum> = 2
```

However, if an ***enum*** fills up the entire space of possible value, this will force the Option to use an additional byte to represent the discriminant. As such, the size of the Option<…> will be higher.

Error management (Option type)

It is also worth mention that Rust has several types that have similar optimization if used with an Option:

- **NonNull** (a raw pointer that can not be Null). In this case, the value Null (since it is an impossible value) will be used to describe the None case from an Option
- **NonZero{*numeric type*}** (an integer that can not be 0). The following types are allowed: NonZeroI8, NonZeroI16, NonZeroI32, NonZeroI64, NonZeroI128, NonZeroIsize, NonZeroU8, NonZeroU16, NonZeroU32, NonZeroU64, NonZeroU128, NonZeroUsize. These types are in fact wrappers around the basic integer types that make sure that the value is not 0. As such, they can be used within an Option and keep the same size in memory.

Error management (Option type)

Now … let's see how a C++ representation of a Rust Option looks like.
We will try to represent an Option<i32> in Rust.

C++ (possible representation for Option<i32>)

```
class OptionalInt {
    bool ok;
    int value;
public:
    OptionalInt(): ok(false), value(0) {}
    OptionalInt(int v): ok(true), value(v) {}
    inline bool has_value() const { return ok; }
    inline int value() const { if (!ok) throw "error"; return value; }
}
```

Error management (Option type)

And the usage within our function for odd numbers will look like this:

C++

```
class OptionalInt {
    bool ok;
    int value;
public:
    ..................................
}

OptionalInt validate_odd(int value) {
    if (value % 2 == 1)
        return OptionalInt(value);
    else
        return OptionalInt();
}
```

Error management (Option type)

Similarly, for the case of a reference (denoted by a non-null pointer in C++), consider the following possible implementation:

C++

```cpp
class OptionalReferenceToInt
{
    int *value;
public:
    OptionalReferenceToInt() : value(nullptr) {}
    OptionalReferenceToInt(int *v) : value(v) {}
    inline bool has_value() const { return value != nullptr; }
    inline int& value() const
    {
        if (value == nullptr) throw "error";
        return *value;
    }
};
```

Error management (Option type)

C++ comparison:

Rust (Option)	C++ (std::optional)
`fn unwrap(self) -> T`	`T& value();`
`fn expect(self, msg: &str) -> T`	N/A
`fn is_some(&self) -> bool`	`bool has_value()`
`fn is_none(&self) -> bool`	N/A
`fn unwrap_or(self, default: T) -> T`	`T value_or(T&& default)`
`fn unwrap_or_else<F>(self, f: F) -> T`	`T or_else(F&& function)` *[from C++23]*
`fn and_then<F>(self, f: F) -> Option<T>`	`T and_then(F&& function)` *[from C++23]*
`None`	`std::nullopt`

Obs: `.is_none()` method is the opposite of `.is_some()` (so it is not necessary).

C++ also supports a lot of operators on top of std::optional. For example (.has_value() method is also called via a cast operator to bool).

Error management (Option type)

C++ comparison:

Rust

```
fn validate_odd(n: i32) -> Option<i32> {
    if n % 2 == 1 {
        return Some(n);
    } else {
        return None;
    }
}
fn main() {
    let r = validate_odd(5);
    if r.is_some() {
        println!("Number is odd: {}",
r.unwrap());
    } else {
        println!("Error");
    }
}
```

C++17 and beyond

```
std::optional<int> validate_odd(int value) {
    if (value % 2 == 1)
        return value;
    else
        return std::nullopt;
}
void main() {
    auto r = validate_odd(5);
    if (r.has_value())
        printf("Number is odd %d", r.value());
    else
        printf("Error");
}
```

Error management (Option type)

As a general overview, use Option in the following cases:

1. You have a function that might or might not return a value of some type (one good example will be something like a String to Number function that might be able to convert a parameter into a valid number or it might be not).

```
Option<i32> StringToNumber(s: &str)
```

instead of

```
bool StringToNumber(s: &str, result: &mut i32)
```

2. You need to return an error code from a function (e.g., use the Option<T> to include the error code, or None for no error code). This a less utilized case, but it is still possible.

Error management (Result type)

Error management (Result type)

Rust has a special generics/template type (declared as an enum) named Result that is used for these cases:

Rust (Source: result.rs from core library)

```
pub enum Result<T, E> {
    #[lang = "Ok"]
    #[stable(feature = "rust1", since = "1.0.0")]
    Ok(#[stable(feature = "rust1", since = "1.0.0")] T),

    #[lang = "Err"]
    #[stable(feature = "rust1", since = "1.0.0")]
    Err(#[stable(feature = "rust1", since = "1.0.0")] E),
}
```

Simplified view

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

Error management (Result type)

Let’s see some examples:

Rust

```
fn division(n1: i32, n2: i32) -> Result<i32, &'static str> {
    if n2 == 0 {
        return Err("Division by zero");
    } else {
        return Ok(n1 / n2);
    }
}
fn main() {
    let r1 = division(5, 0);
    let r2 = division(5, 1);
    print!("{:?},{:?}", r1, r2);
}
```

Output

```
Err("Division by zero"),Ok(5)
```

Error management (Result type)

Note that division function can be simplified by removing the return keyword and the final semicolon:

Rust

```
fn division(n1: i32, n2: i32) -> Result<i32, &'static str> {
    if n2 == 0 {
        Err("Division by zero")
    } else {
        Ok(n1 / n2)
    }
}
```

To provide even more clarity, Err and Ok can be preceded by the ***enum*** name:

Rust

```
fn division(n1: i32, n2: i32) -> Result<i32, &'static str> {
    if n2 == 0 {
        Result::Err("Division by zero")
    } else {
        Result::Ok(n1 / n2)
    }
}
```

Error management (Result type)

To check if a ***Result*** contains an error or an ok value, use the methods:
is_err() / .err() / .is_ok() / . ok()

Rust

```
fn division(n1: i32, n2: i32) -> Result<i32, &'static str> { ... }
fn main() {
    let r = division(5, 0);
    if r.is_err() {
        println!("Error found: {}", r.err().unwrap());
    } else {
        println!("Success, result is {}", r.ok().unwrap());
    }
}
```

Output

Error found: Division by zero

Error management (Result type)

To check if a ***Result*** contains an error or an ok value, use the methods:

is_err() / .err() / .is_ok() / . ok()

Rust

```
fn division(n1: i32, n2: i32) -> Result<i32, &'static str> { ... }
fn main() {
    let r = division(5, 0);
    if r.is_err() {
        println!("Error found: {}", r.err().unwrap());
    } else {
        println!("Success, result is {}", r.ok().unwrap());
    }
}
```

Output

Success, result is 2

… and a success case (e.g. a division between 5 and 2)

Error management (Result type)

Why do we need that .unwrap() to get the err or ok value ?

Rust

```
fn division(n1: i32, n2: i32) -> Result<i32, &'static str> { ... }
fn main() {
    let r = division(5, 2);
    if r.is_err() {
        println!("Error found: {}", r.err().unwrap());
    } else {
        println!("Success, result is {}", r.ok().unwrap());
    }
}
```

That's because both .err() and .ok() methods return an Option and not the actual value: `pub const fn err(self) -> Option<E>` and `pub const fn ok(self) -> Option<T>`

Where T and E are template/generics types define in Result enum.

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

Error management (Result type)

Result type has the following methods:

Method	Usage
`fn unwrap(self) -> T`	Returns the value if no error is present or panics otherwise
`fn expect(self, msg: &str) -> T`	Returns the value if no error is present or panics with a specific message otherwise
`fn expect_err(self, msg: &str) -> E`	Returns the error if present, else panics with a specific message
`fn is_ok(&self) -> bool`	True if no error is present, false otherwise
`fn is_err(&self) -> bool`	True if error is present, false otherwise
`fn unwrap_or(self, default: T) -> T`	Returns the value if no error is present or a default value in case of error
`fn unwrap_or_else<F>(self, f: F) -> T`	Returns the value if no error is present or the result of a function in case of error
`fn err(self) -> Option<E>`	Returns an Option over an error
`fn ok(self) -> Option<T>`	Returns an Option over the value

Error management (Result type)

Let's see how Rust result type is stored in memory:

Rust

```
fn i32_or_i8(value: i32) -> Result<i32, i8>
{
    if value < 255 {
        Err(value as i8)
    } else {
        Ok(value as i32)
    }
}
fn main() {
    let mut r = i32_or_i8(10);
    r = i32_or_i8(1000);
    if r.is_ok()
    {
        println!("value = {}", r.unwrap());
    }
}
```

Error management (Result type)

Let’s see how Rust result type is stored in memory:

Rust

```rust
fn i32_or_i8(value: i32) -> Result<i32, i8>
{
    if value < 255 {
        Err(value as i8)
    } else {
        Ok(value as i32)
    }
}
fn main() {
    let mut r = i32_or_i8(10);
    r = i32_or_i8(1000);
    if r.is_ok()
    {
        println!("value = {}", r.unwrap());
    }
}
```

Memory layout for variable “r”

+0	+1	+2	+3	+4	+5	+6	+7
01	0A	00	00	00	00	00	00

0A (hex) = 10 => the i8 value

01 = ERROR (we are on the error case)

<u>Possible structure (C representation)</u>

struct ErrorCase {
unsigned char index; *// always 1 (Err case)*
signed char value;
}

Error management (Result type)

Let's see how Rust result type is stored in memory:

Rust

```
fn i32_or_i8(value: i32) -> Result<i32, i8>
{
    if value < 255 {
        Err(value as i8)
    } else {
        Ok(value as i32)
    }
}
fn main() {
    let mut r = i32_or_i8(10);
    r = i32_or_i8(1000);
    if r.is_ok()
    {
        println!("value = {}", r.unwrap());
    }
}
```

Memory layout for variable "r"

+0	+1	+2	+3	+4	+5	+6	+7
00	0A	00	00	E8	03	00	00

E8 03 00 00 (hex) = 1000 (0xE8+0x03*256)

00 = OK (we are on the ok case)

Possible structure (C representation)

```
struct OkCase {
    unsigned char index;   // always 0 (Ok case)
    unsigned int   value;
}
```

3 bytes alignment

Error management (Result type)

Let’s see how Rust result type is stored in memory:

Rust

```rust
fn i32_or_i8(value: i32) -> Result<i32, i8>
{
    if value < 255 {
        Err(value as i8)
    } else {
        Ok(value as i32)
    }
}
fn main() {
    let mut r = i32_or_i8(10);
    r = i32_or_i8(1000);
    if r.is_ok()
    {
        println!("value = {}", r.unwrap());
    }
}
```

Possible structure (C representation)

```c
union Result_i32_i8 {
  unsigned char index;

  struct {
    unsigned char index;   // always 0
    unsigned int   value;
  } Ok;

  struct {
    unsigned char index;   // always 1
    signed char     value;
  } Err;
}
```

Error management (Result type)

Let's see how Rust result type is stored in memory:

Rust

```rust
fn i32_or_i8(value: i32) -> Result<i32, i8>
{
    if value < 255 {
        Err(value as i8)
    } else {
        Ok(value as i32)
    }
}
fn main() {
    let mut r = i32_or_i8(10);
    r = i32_or_i8(1000);
    if r.is_ok()
    {
        println!("value = {}", r.unwrap());
    }
}
```

Caller:

```
lea     rcx,[r]
call    Result::is_ok
```

Notice that RCX holds the address of **"r"** !!!

Error management (Result type)

Let’s see how Rust result type is stored in memory:

Rust

```rust
fn i32_or_i8(value: i32) -> Result<i32, i8>
{
    if value < 255 {
        Err(value as i8)
    } else {
        Ok(value as i32)
    }
}
fn main() {
    let mut r = i32_or_i8(10);
    r = i32_or_i8(1000);
    if r.is_ok()
    {
        println!("value = {}", r.unwrap());
    }
}
```

Indeed, the first byte from the address of “r” is being checked to see if it is 1 or 0

Result::is_ok:

```asm
Result::is_ok:
        sub     rsp,10h
        mov     qword ptr [rsp+8],rcx
        mov     al,byte ptr [rcx]
        and     al,1
        movzx   eax,al
        cmp     rax,0
        jne     index_is_0
        mov     byte ptr [rsp+7],1
        jmp     end_if
index_is_0:
        mov     byte ptr [rsp+7],0
end_if:
        mov     al,byte ptr [rsp+7]
        and     al,1
        movzx   eax,al
        add     rsp,10h
        ret
```

Error management (Result type)

Rust ***Result*** type has two similarities in C++:

1. **std::variant** (available from C++17) → more generic, and can be adjusted to reflect a Result
2. **std::expected** (available from C++23) → this is the closest template that mimics the way Rust Result works.

Note that ***std::variant*** is not designed for error management (but it can be used for this purpose). std::variant can contain multiple different types while Result only has two types.

Error management (Result type)

C++ comparison:

Rust (Result)	C++ (std::expected)	C++ (std::variant)
fn unwrap(self) -> T	T& value();	T& std::get<T> ();
fn expect(self, msg: &str) -> T	N/A	N/A
fn expect_err(self, msg: &str) -> E	N/A	N/A
fn is_ok(&self) -> bool	bool has_value()	bool std::holds_alternative<T> ();
fn is_err(&self) -> bool	N/A	bool std::holds_alternative<E> ();
fn unwrap_or(self, default: T) -> T	N/A	N/A
fn unwrap_or_else<F>(self, f: F) -> T	N/A	N/A
fn err(self) -> Option<E>	E& error();	T& std::get<T> ();
fn ok(self) -> Option<T>	T& value();	T& std::get<E> ();

Obs: .is_ok() method is the opposite of .is_err() (so an equivalent is not necessary).

Error management (Result type)

C++ comparison:

Rust

```rust
fn i32_or_i8(value: i32) -> Result<i32, i8>
{
    if value < 255 {
        Err(value as i8)
    } else {
        Ok(value as i32)
    }
}
fn main() {
    let mut r = i32_or_i8(10);
    r = i32_or_i8(1000);
    if r.is_ok()
    {
        println!("value = {}", r.unwrap());
    }
}
```

C++17

```cpp
std::variant<int32_t,int8_t> i32_or_i8(int value)
{
    if (value < 255)
        return (int8_t)value;
    else
        return (int32_t)value;
}
void main()
{
    auto r = i32_or_i8(10);
    r = i32_or_i8(1000);
    if (std::holds_alternative<int32_t>(r))
        printf("Value = %d",std::get<int32_t>(r));
}
```

Error management (Result type)

C++ comparison:

Rust

```rust
fn i32_or_i8(value: i32) -> Result<i32, i8>
{
    if value < 255 {
        Err(value as i8)
    } else {
        Ok(value as i32)
    }
}
fn main() {
    let mut r = i32_or_i8(10);
    r = i32_or_i8(1000);
    if r.is_ok()
    {
        println!("value = {}", r.unwrap());
    }
}
```

C++23

```cpp
std::expected<int32_t,int8_t> i32_or_i8(int value)
{
    if (value < 255)
        return (int8_t)value;
    else
        return (int32_t)value;
}
void main()
{
    auto r = i32_or_i8(10);
    r = i32_or_i8(1000);
    if (r.has_value())
        printf("Value = %d", r.value());
}
```

Error management (Result type)

It is worth mention that Result can be used as a returned type for main function, Option however can not. In order to use a Result as a return type for the main function, the Ok type has to implement a special trait called ***Termination***. For example, the following types implement this trait:

- std::process::ExitCode
- Unit type “()” or void
- Never type “!”

Error management (Result type)

Let's see some examples:

Rust

```
fn main() -> Result<(), i32> {
    let x = 10;
    println!("x = {:?}", x);
    Ok(())
}
```

Output

```
x = 10
```

or with

Rust

```
use std::process::ExitCode;
fn main() -> Result<ExitCode, i32> {
    let x = 10;
    println!("x = {:?}", x);
    Ok(ExitCode::SUCCESS)
}
```

Output

```
x = 10
```

Error management (Result type)

Let's see some examples:

Rust

```
fn main() -> Result<i32, bool> {
    let x = 10;
    println!("x = {:?}", x);
    Ok(10)
}
```

Compiler error

```
error[E0277]: the trait bound `i32: Termination` is not satisfied
 --> src\main.rs:1:14
  |
1 | fn main() -> Result<i32, bool> {
  |              ^^^^^^^^^^^^^^^^^ the trait `Termination` is not implemented for `i32`
  |
  = note: required for `Result<i32, bool>` to implement `Termination`
```

Since "i32" does not implement the trait Termination, it can not be used as valid type for the Ok variant of an Result from main function.

If let
let else
while let

If let / let else / while let

When used with enums, if let and while let have a special syntax that allows de-structuring the enum and copy the value associated with it into a variable that will further be used in the next expression block.

Format:

- ***if let EnumVariant(variable) = Expression { … }***
- ***while let EnumVariant(variable) = Expression { … }***

Where:

- Expression returns an Enum object. One of the variants of that Enum has to be of type EnumVariant
- If variable is of type EnumVariant, the variable is initialized and the if condition is considered to be true
- This also mean that while in complex expression from if let or while let form, that variable is initialized and can be used.

If let / let else / while let

Let's see a simple example:

Rust

```rust
enum Values {
    Bool(bool),
    Integer(i32),
    Float(f32),
}
fn main() {
    let i = Values::Integer(10);
    if let Values::Integer(v) = i {
        println!("i is Integer and has value: {v}");
    }
}
```

Output

```
i is Integer and has value: 10
```

If let / let else / while let

Let’s see a simple example:

Rust

```
enum Values {
    Bool(bool),
    Integer(i32),
    Float(f32),
}
fn main() {
    let i = Values::Integer(10);
    if let Values::Integer(v) = i {
        println!("i is Integer and has va
    }
}
```

This translates in the following way:

- if “I” is of variant ***Integer,*** that copy the value of “I” (of type i32) into a newly created variable (v) of type i32 and run the **THEN** block of the if instruction.

If let / let else / while let

Let’s see a simple example:

Rust

```
enum Values {
    Bool(bool),
    Integer(i32),
    Float(f32),
}
use Values::*;
fn main() {
    let i = Values::Integer(10);
    if let Integer(v) = i {
        println!("i is Integer an
    }
}
```

Notice that importing all values from values: `use Values::*;` allows us to use the name of the variant directly in an if let / while let structure.

If let / let else / while let

Let's see a simple example:

Rust

```
enum Values {
    Bool(bool),
    Integer(i32),
    Float(f32),
}
use Values::*;
fn main() {
    let i = Values::Integer(10);
    let f = Values::Float(1.2);
    if let (Integer(v1),Float(v2)) = (i,f) {
        println!("i is Integer and has value: {v1}");
        println!("f is Float and has value  : {v2}");
    }
}
```

Output

```
i is Integer and has value: 10
f is Float and has value  : 1.2
```

If let / let else / while let

Let's see a simple example:

Rust

```
enum Values {
    Bool(bool),
    Integer(i32),
    Float(f32),
}
    let i = Values::Integer(10);
    let f = Values::Float(1.2);
    if let (Integer(v1),Float(v2)) = (i,f) {
        println!("i is Integer and has value: {v1}");
        println!("f is Float and has value  : {v2}");
    }
}
```

This actually translates into:
If "I" is of type Integer and "f" is of type Float then
copy the value of "I" into "v1" and the value of "f" into "v2"

If let / let else / while let

Let's see a simple example:

Rust

```
enum Values { Bool(bool), Integer(i32), Float(f32) }
use Values::*;
fn main() {
    let i = Values::Integer(10);
    let f = Values::Bool(true);
    if let (Integer(v1),Float(v2)) = (i,f) {
        println!("i is Integer and has value: {v1}");
        println!("f is Float and has value  : {v2}");
    } else {
        println!("No match");
    }
}
```

Output
No match

If let / let else / while let

Let's see a simple example:

Rust

```
enum Values { Bool(bool), Integer(i32), Float(f32) }
use Values::*;
fn main() {
    let i = Values::Integer(10);
    let f = Values::Bool(true);
    if let (Integer(v1),Float(v2)) = (i,f) {
        println!("i is Integer and has value: {v1}");
        println!("f is Float and has value  : {v2}");
    } else {
        println!("No match");
    }
}
```

Notice that "f" is of type Bool. As such the condition from if let statement is false as "f" is not of type Float

If let / let else / while let

This feature is in particular useful for usage with Option. For example:

Rust

```
fn smaller_than_5(value: i32) -> Option<i32> {
    if value < 5 {
        Some(value)
    } else {
        None
    }
}
fn main()
{
    let mut x = 0;
    while let Some(i) = smaller_than_5(x) {
        println!("x = {i}");
        x += 1;
    }
}
```

Output

```
x = 0
x = 1
x = 2
x = 3
x = 4
```

If let / let else / while let

The same logic with multiple variants can be used for while let as well:

Rust

```
fn smaller_than(x: i32, value: i32) -> Option<i32> {
    if x < value {
        Some(x)
    } else {
        None
    }
}
fn main() {
    let mut x = 0;
    while let (Some(i), Some(j)) = (smaller_than(x, 5), smaller_than(x * 3, 8)) {
        println!("i = {i}, j = {j}");
        x += 1;
    }
}
```

Output

```
i = 0, j = 0
i = 1, j = 3
i = 2, j = 6
```

If let / let else / while let

The same logic with multiple variants can be used for while let as well:

Rust

```
fn smaller_than(x: i32, value: i32) -> Option<i32> {
    if x < value {
        Some(x)
    } else {

    }
}
fn main() {
    let mut x = 0;
    while let (Some(i), Some(j)) = (smaller_than(x, 5), smaller_than(x * 3, 8)) {
        println!("i = {i}, j = {j}");
        x += 1;
    }
}
```

This translate as follows: while `smaller_than(x, 5)` returns a variant type of ***Some***, and `smaller_than(x * 3, 8)` also return a variant type of ***Some***, copy the resulted values into variable "i" and "j" and enter the while loop.

If let / let else / while let

let else is also a special syntax that allows direct initialization of a variable from an expression that results in an enum (for example an Option)

Format:

- ***let EnumVariant(variable) = Expression else { <error code> }***

The error code is usually a panic macro , or a return values (if this is called from a function). It should be noticed that this is a sugar syntax for the following:

- let x = if let EnumVariant(variable) { variable } else { <error code> }

If let / let else / while let

Example:

Rust

```
fn smaller_than(x: i32, value: i32) -> Option<i32> {
    if x < value {
        Some(x)
    } else {
        None
    }
}
fn main() {
    let Some(x) = smaller_than(2, 3) else { panic!("Fail to initialize x"); };
    println!("x={x}");
}
```

Output
X=2

Notice that “x” is of type i32 ! (and not Option<i32>)

If let / let else / while let

Example:

Rust

```rust
fn smaller_than(x: i32, value
    if x < value {
        Some(x)
    } else {
        None
    }
}
fn main() {
    let Some(x) = smaller_than(4, 3) else { panic!("Fail to initialize x"); };
    println!("x={x}");
}
```

Panic (runtime)

```
thread 'main' panicked at 'Fail to initialize x', src\main.rs:9:44
stack backtrace:
   0: std::panicking::begin_panic_handler
             at
/rustc/8ede3aae28fe6e4d52b38157d7bfe0d3bceef225/library\std\src\panicking.rs:593
   1: core::panicking::panic_fmt
             at
/rustc/8ede3aae28fe6e4d52b38157d7bfe0d3bceef225/library\core\src\panicking.rs:67
   ...
```

If let / let else / while let

You can also use let…else syntax to return something from a function:

Rust

```
fn smaller_than(x: i32, value: i32) -> Option<i32> {
    if x < value {
        Some(x)
    } else {
        None
    }
}
fn get_x(value: i32)->i32 {
    let Some(x) = smaller_than(value, 3) else { return -1; };
    x*2
}
fn main() {
    println!("get_x(2)={}",get_x(2));
    println!("get_x(4)={}",get_x(4));
}
```

Output

```
get_x(2)=4
get_x(4)=-1
```

Question mark operator (?)

Question mark operator (“?”)

Rust has a special unary operator “?” that works with Option and Result generics in the following way:

- Let’s consider these expression: “x = a?”, where “a” is of type ***Option<…>*** or ***Result<…>***
- The “?” performs the following actions:
    1. If the value of “a” is Some (for Option) or Ok (for Result), then “a” gets unwrapped and assigned it to “x”
    2. If the value of “a” is None (for Option) or Err (for Result) and the function where “x = a?” operation is located returns an Option or Result, then the function returns immediately the value of “a” → a None or Err

OBS: “?” operator can only be used in a function that has a return type of Option<…> or Result<…>

Question mark operator ("?")

Let's see some examples:

Rust

```
fn sum(v1: Option<i32>, v2: Option<i32>) -> Option<i32> {
    let x = v1?;
    let y = v2?;
    Some(x + y)
}
fn main() {
    let result = sum(Some(10), Some(15));
    println!("{:?}", result);
}
```

Output

Some(25)

In this case since both Some(10) and Some(15) are valid, the result returns the expected sum.

Question mark operator (“?”)

If however, we change one of the parameters to **None,** the sum is not possible anymore, but we don’t need to change the code as the execution will stop when evaluating “y”.

Rust

```
fn sum(v1: Option<i32>, v2: Option<i32>) -> Option<i32> {
    let x = v1?;
    let y = v2?;
    Some(x + y)
}
fn main() {
    let result = sum(Some(10), None);
    println!("{:?}", result);
}
```

Output

```
None
```

Question mark operator ("?")

In reality, ***sum*** function can be written in two ways:

A) Using question mark operator ?

Rust

```
fn sum(v1: Option<i32>, v2: Option<i32>) -> Option<i32> {
    let x = v1?;
    let y = v2?;
    Some(x + y)
}
```

B) Using if expressions

Rust

```
fn sum(v1: Option<i32>, v2:Option<i32>) -> Option<i32> {
    let mut x = 0;
    if v1.is_none() { return None; } else { x = v1.unwrap(); }
    let mut y = 0;
    if v2.is_none() { return None; } else { y = v2.unwrap(); }
    Some(x + y)
}
```

Equivalent code

Question mark operator ("?")

Its important to notice that the operator "?" does not require the same type to be used (but rather the same type for the error case).

Rust

```
fn foo()->Option<i32> {
    return None
}
fn goo()->Option<f64> {
    let x = foo()?;
    Some(1.234)
}
fn main() {
    let x = goo();
    println!("x = {:?}",x);
}
```

Output

x = None

Question mark operator ("?")

Its important to notice that the operator "?" does not require the same type to be used (but rather the same type for the error case).

Rust

```
fn foo()->Option<i32> {
    return None
}
fn goo()->Option<f64> {
    let x = foo()?;
    Some(1.234)
}
fn main() {
    let x = goo();
    println!("x = {:?}",x);
}
```

Notice that foo() and goo() function have different result types (one returns an Option<i32> and the other one an Option<f64>)

Question mark operator (“?”)

Its important to notice that the operator “?” does not require the same type to be used (but rather the same type for the error case).

Rust

```
fn foo()->Option<i32> {
    return None
}
fn goo()->Option<f64> {
    let x = foo()?;
    Some(1.234)
}
fn main() {
    let x = goo();
    println!("x = {:?}",x);
}
```

However, what's important is that the error type should be similar (in our case, the error type of an Option is None and as such is the same for both Option<i32> and Option<f64>). **Or to be more precisely, Option<i32>::None can be converted into Option<f64>::None**

Question mark operator ("?")

The same logic applies for Result as well. In this case it is important for a conversion between the error type of different Results to be possible.

Rust

```rust
fn foo()->Result<String,i32> {
    return Err(10)
}
fn goo()->Result<f64,i32> {
    let x = foo()?;
    Ok(1.234)
}
fn main() {
    let x = goo();
    println!("x = {:?}",x);
}
```

Output

```
x = Err(10)
```

Question mark operator (“?”)

The same logic applies for Result as well. In this case it is important for a conversion between the error type of different Results to be possible.

Rust

```rust
fn foo()->Result<String,i32> {
    return Err(10)
}
fn goo()->Result<f64,i32> {
    let x = foo()?;
    Ok(1.234)
}
fn main() {
    let x = goo();
    println!("x = {:?}",x);
}
```

In this case, the error case of foo is of type Err(i32). Similarly, the error case of goo is of type Err(i32). This means that even if the Ok cases for this two cases are different, we can use the “?” operator as there is obviously a conversion possible from Err(i32) to Err(i32)

Question mark operator ("?")

The same logic applies for Result as well. In this case it is important for a conversion between the error type of different Results to be possible.

Rust

```
fn foo()->Result<String,i8> {
    return Err(10)
}
fn goo()->Result<f64,i32> {
    let x = foo()?;
    Ok(1.234)
}
fn main() {
    let x = goo();
    println!("x = {:?}",x);
}
```

Notice that the Error types don't have to be identical. There just has to be a conversion possible between them. In this case there is one between i8 and i32

Output

x = Err(10)

Question mark operator ("?")

The same logic applies for Result as well. In this case it is important for a conversion between the error type of different Results to be possible.

Rust

```rust
fn foo()->Result<String,f64> {
    return Err(10.2)
}
fn goo()->Result<f64,i32> {
    let x = foo()?;
    Ok(1.234)
}
fn main() {
    let x = goo();
    println!("x = {:?}",x);
}
```

Error

```
error[E0277]: `?` couldn't convert the error to `i32`
 --> src\main.rs:5:18
  |
4 | fn goo()->Result<f64,i32> {
  |           --------------- expected `i32` because of this
5 |     let x = foo()?;
  |                  ^ the trait `From<f64>` is not implemented for `i32`
  |
  = note: the question mark operation (`?`) implicitly performs a conversion on
          the error value using the `From` trait
  = help: the following other types implement trait `From<T>`:
```

In this case, Err(f64) can not be converted to Err(i32) so this type of error can not be propagated.

Q & A