Rust: into the danger zone

SSP-RS — Safe System Programming (in Rust)

Samuel Tardieu Stefano Zacchiroli

2024-11-05

The danger zone

What is the danger zone?

Contrary to what beginners may assume, Rust gives developpers as much control over the environment as other languages such as C would:

- You may write arbitrary data anywhere in memory at any time.
- You may call arbitrary code with any argument you wish to give it.

Even more, when building an application:

- You can execute arbitrary code on the system the Rust compiler runs on at compile time.
- You can use this capability to generate code to be compiled.

Those are the topics we will study in today's class.

Unsafe programming

Rust and safety

Safety in Rust is a major asset of the language and is often emphasized when comparing Rust with C or C++:

- It is possible to write safe C or C++. However, the compiler will not prevent you from building a `NULL`, misaligned, or otherwise invalid pointer.
- By contrast, in Rust, as long as you don't use the `unsafe` keyword (staying within the *safe Rust* domain), you receive strong guarantees enforced by the compiler. However, this comes with additional constraints such as lifetimes, `Send`/`Sync`, …

In this class, we will study why and how to disable certain Rust safeguards. We will always do this locally, ensuring that the global safety properties are not compromised.

First of all, let's check what those safety properties look like.

Rust safety guarantees

In *safe Rust* (the default mode), Rust guarantees that some properties always hold. Those guarantees are enforced by the compiler and do not cause any overhead at run time. Of course, the compiler works with what it knows: it assumes that any use of the `unsafe` keyword preserves those guarantees.

They are, for example:

- Only the owner of an object can build an initial reference to this object. Such a reference can then be transmitted, lent, or copied.
- No reference will ever survive the object it is referencing.
- No mutable reference to an object can exist at the same time as another reference.
- No object can be modified while a reference (mutable or not) to it exist.
- No object can be transferred to another thread unless its type implements the `Send` trait.
- No object can be referenced from another thread unless its type implements the `Sync` trait.
- No object contains data outside the scope of its type (`bool` are stored as 0 or 1 in a byte).
- A `String` (or a `&str`) always contains valid UTF-8 data.
- No object with a destructor will ever contain an invalid value.

The `unsafe` keyword

An `unsafe` block allows the programmer to call functions marked as `unsafe`:

```
1 fn main() {
2   let pi: f32 = 3.141592653589;
3   let x: u32 = unsafe { std::mem::transmute(pi) };
4   println!("pi = {pi:.02}, x = {x}");     // pi = 3.14, x = 1078530011 on a x86-64 computer
5 }
```

`std::mem::transmute()` is an unsafe function: it can convert any type into another type provided they have the same size. Its signature looks like:

```
pub unsafe fn transmute<S, D>(src: S) -> D
```

If we hadn't used `unsafe` around the call to `std::mem::transmute()` on line 3, the compiler would have issued an error message:

```
error[E0133]: call to unsafe function is unsafe and requires unsafe function or block
 --> t.rs:3:16
  |
3 |   let x: u32 = std::mem::transmute(pi);
  |                ^^^^^^^^^^^^^^^^^^^^^^^ call to unsafe function
```

Why are some functions unsafe?

Converting a `f32` to a `u32` using the unsafe `std::mem::transmute()` function looks innocuous: it allows us to look at the internal representation of a floating-point number and no harm is done.

However, do you think the following code, which compiles without any warning, is safe?

```
1  fn main() {
2    let bytes: [usize; 3] = [1, 2, 3];
3    let s: String = unsafe { std::mem::transmute(bytes) };  // Kids, do not try this at home
4    println!("s = {s}");
5  }
```

Hint: it is not, and executing this code triggers a segmentation fault on Unix systems. Why is that?

We build a `String` by filling its pointer, capacity, and size (the order does not matter, it is wrong anyway) with values 1, 2, and 3. We certainly have not reserved heap memory at any of those addresses. This program can fail in two ways:

- printing the string content will likely fail;
- freeing the supposedly reserved memory in the destructor will most certainly fail.

So is unsafe really usable?

Yes. The following code is safe even though it uses the `unsafe` keyword twice. Why?

```
1  struct RawData([u8; 24]);   // On a 64 bit computer
2
3  impl RawData {
4    pub fn new(s: String) -> Self {
5      RawData(unsafe { std::mem::transmute(s) })
6    }
7
8    pub fn restore(self) -> String {
9      unsafe { std::mem::transmute(self.0) }
10   }
11 }
12
13 fn main() {
14   let r = RawData::new(String::from("it works!"));
15   println!("Let's convert it back: {}", r.restore());
16 }
```

- The only way to build a `RawData` is by giving a valid `String` to `RawData::new()`.
- The `RawData` object can be transformed back into a `String` only once.

Evolution of unsafe

We will mention Rust releases (such as Rust 1.82) and Rust editions (such as Rust 2024):

- A new release appears every 6 weeks, after a 6 weeks beta testing phase). For example, Rust 1.82 was out on October 17, 2024, and Rust 1.83 will be out on November 28, 2024.
- A new edition appears approximately every 3 years. Rust 2024 will correspond to Rust 1.85, which will be released in February 20, 2025. Previous editions were Rust 2015, 2018, and 2021.
- A new edition usually introduces backward incompatible syntax changes (for example, `gen` will be a reserved keyword in Rust 2024). Tools (`cargo fix --edition`) help migrate code automatically (`gen` → `r#gen`).
- A project indicates the revision it uses in `Cargo.toml`:

```
[package]
edition = "2021"
```

- Rust 2024 will introduce some changes in the handling of `unsafe`, indicated in the next slides

Exposing an unsafe function

An `unsafe` function forces the user to call it in an `unsafe` context. Inside the `unsafe` function itself, you do not need to use the `unsafe` keyword (but you will get a warning starting from Rust 2024 if you don't):

It is considered good practice when writing an unsafe function to document why it is unsafe:

```
1  /// Convert a sequence of bytes into a String. The bytes must
2  /// have been obtained from a reverse conversion.
3  ///
4  /// # Safety
5  ///
6  /// This function must not be called twice with the same sequence
7  /// of bytes as that would return strings pointing to the same
8  /// heap memory area.
9  pub unsafe fn from_bytes(bytes: [u8; 24]) -> String {
10   unsafe { std::mem::transmute(bytes) }
11 }
```

⚠ Even after an `unsafe` function has been called, Rust invariants must be preserved, provided that the function has been used properly.

Exposing an unsafe function (cnt'd)

An `unsafe` function does not necessarily mean that its body calls unsafe functions. The author of the function may want to force the programmer to take responsibility:

```
1  /// Turn the laser on.
2  ///
3  /// # Safety
4  ///
5  /// The laser must be turned on only if no other laser points to
6  /// the same area at the same time, otherwise a human standing
7  /// there could suffer from eye damage or skin burns.
8  pub unsafe fn turn_laser_on(laser: &Laser) {
9    laser.set_power_to(Power::UltraMaxPower);
10 }
```

In general, an `unsafe` function is exposed when neither the compiler nor the function author can check that the necessary conditions for using this function safely are being met. The burden is transferred to the user of the function.

Most of the time however, `unsafe` functions manipulate references and pointers.

Transmuting and references

`std::mem::transmute()` allows one to build references in a most unsafe (and wrong) way. What does the following code print?

```
1  fn add(a: &u32, b: &u32, c: &mut u32) {
2    *c = *a;
3    *c += *b;
4  }
5
6  fn main() {
7    let a: u32 = 1;
8    let mut b: u32 = 20;
9    let c: &mut u32 = unsafe { std::mem::transmute(&mut b) };  // c and b points to the same u32
10   add(&a, &b, c);
11   println!("a = {a}, b = {b}, c = {c}");
12 }
```

Without optimizations: a = 1, b = 2, c = 2. With optimizations: a = 1, b = 21, c = 21

On lines 2 and 3, the compiler may assume that `c` doesn't point at the same location as `a` or `b` since a mutable reference to an object cannot exist at the same type as a non-mutable reference.

Limitations of references

Sometimes, references, as powerful as they are, can be seen as an obstacle:

- We may want to interface with other programming languages which are not as cautious as Rust. They will manipulate pointers pointing to the same data in a mutable way.
- We may need to temporarily store a reference to some data even though its lifetime would not allow us to do so. If we know that this is safe to do because the data will not be destroyed or moved before the container is, we should be able to do this.

Even though you might not need it in your day-to-day use, Rust is perfectly able to do the things described above. Let's meet the raw pointers.

Raw pointers

In addition to references, which are similar to a pointer to some valid date associated with a consistent lifetime, Rust has *raw pointers*, similar to C pointers:

- A pointer to non-modifiable data of type `T` has type `*const T`.
- A pointer to modifiable data of type `T` has type `*mut T`.
- Those pointers can contain `NULL` or may be invalid, they are not Rust references.

It is very easy to build a raw pointer:

- `&raw const expr` and `&raw mut expr` since Rust 1.82
- `&T` and `&mut T` can be cast into `*const T` without using `unsafe`
- `&mut T` can be cast into `*mut T` without using `unsafe`
- `*const T` and `*mut T` can also be built from an integer without using `unsafe`

However, *using* a raw pointer is an unsafe operation:

- `*const T` can be converted to `Option<&T>` using the unsafe `as_ref()` method
- `*mut T` can be converted to `Option<&mut T>` using the unsafe `as_mut()` method

When are raw pointers used?

Raw pointers are used in several contexts.

High-level data structures:

- Some structures such as `Vec<T>` or `String` allocate memory on the heap and manipulate the reserved area directly.
- `Arc<T>` is a clonable smart pointer which counts the references made to the object it holds and destroys it when the reference count drop to 0.

Interface with other programming languages:

- You can call a function written in C from Rust. All pointer types on the C side will likely be represented as `*const T` or `*mut T` on the Rust side.
- You can call a function written in Rust from C. The function signature may use raw pointers to give the C side something it understands.

We will now focus on interfacing with C-like languages.

Foreign Function Interface

FFI: introduction

FFI (*foreign function interface*) is a mechanism to interface a programming language (Rust) with others (C, C++) by using the other language ABI (*application binary interface*) instead of the native one.

The ABI describes:

- how data get stored in memory
- how functions are called, where arguments and return values are located
- which hardware registers must be preserved by the caller and the callee accross a function call

For a given hardware architecture, several ABI may exist: Microsoft Windows doesn't use the same calling conventions on x86_64 as Linux or OS X.

When calling C functions, or when generating code for functions exported to the C world, Rust will use the appropriate C ABI.

Representing compound types

Rust does not specify how the fields of a `struct` or an `enum` are organized in memory. However, the compiler can be directed to use a particular representation:

```
#[repr(u8)]                  // Fixed integer representation on enums only
enum E { A, B, C = 3 }       // Represented on a byte with values 0, 1, 3

#[repr(C)]                   // Be compatible with C
#[repr(align(8))]            // Force alignment on 8 bytes (must be 2^n)
struct S { a: u8, b: u32 }  // Inserts 3 bytes of padding after a for proper alignment of b

#[repr(transparent)]
struct T {
  a: (),
  b: i32,                    // Same layout as i32, sole non-ZST field
  c: PhantomData<S>,         // (ZST: zero-sized type with no alignment constraint)
```

You can use `std::mem::transmute()` to convert data of type `E` and `u8` in both directions, as well as data of type `T` and `i32`. Of course this is an `unsafe` operation, as it might for example build a invalid `E` value (*e.g*, by transmuting 2).

C compatible memory layout with `repr(C)`

`repr(C)` enforces C ABI compatibility for the target architecture and operating system:

- Integer types `u*` and `i*` are compatible with their C counterpart `uint*_t` and `int*_t`.
- Floating point types `f32` and `f64` are mapped to C `float` and `double`.
- `usize` and `isize` correspond to C `size_t` and `ssize_t`.
- Raw pointer types `*const T` and `*mut T` are compatible with C `const T *` and `T *`.
- Empty types or zero-sized types will still occupy 0 bytes although this is forbidden by the C standard.
- Fat pointers, such as `&str` or `&[u8]`, are not available in C and should never be used with FFI.
- Tuples are represented as structures, with fields keeping their ordering and their respective properties.
- Rust `enum` type with simple variants are mapped to C `enum` – usually the same size as an `int`.

Basic C interfacing example

Let's call the C standard library function "powf" which is defined in the math (`libm`) library as:

```
float powf(float x, float y);
```

Executing the following Rust code:

```
1 #[link(name = "m")]                    // Link with the C math library "libm"
2 extern "C" {
3   fn powf(x: f32, y: f32) -> f32;      // Raise x to the power of y
4 }
5
6 fn main() {
7   let result = unsafe { powf(2.0, 10.0) };
8   println!("2^10 = {result}");
9 }
```

displays:

```
2^10 = 1024
```

Other standard C types

The libraries we want to interface with might use other C types. The `std::ffi` module defines the following equivalent types:

- `c_void`: used as `*const c_void` or `*mut c_void` to match C `void *` or `const void *` types.
- `c_schar`, `c_short`, `c_int`, `c_long`, `c_longlong`: equivalent to C `signed char`, `short`, etc.
- `c_uchar`, `c_ushort`, `c_uint`, `c_ulong`, `c_ulonglong`: equivalent to C `unsigned char`, `unsigned short`, etc.
- `c_char`, `c_float`, `c_double`: equivalent to C `char`, `float` and `double`.
- C `long double` has no equivalent as Rust does not have a `f128` type. However, if one needs to be exchanged without being interpreted, any 128 bit type with equivalent alignment constraints (such as `u128`) can be used.

Representing null-terminated strings

Rust strings are not null-terminated: they do not have an extra `'\0'` (`NUL`) character at the end. Moreover, `'\0'` is a perfectly valid character to store in a Rust string.

For interfacing with foreign languages using null-terminated strings such as C, Rust has the following types:

- `std::ffi::CString`: a owned null-terminated string whose bytes are stored on the heap. It implements `Drop`.
- `std::ffi::CStr`: a non-owned null-terminated string. `CStr` is to `CString` what `str` is to `String` (except that `CStr` is not a fat pointer).

`CString` and `CStr` can be built from `*const c_char`. The user needs to know if they should own the string data or not.

A `*const c_char` can be obtained by calling the `as_ptr()` method on a `CString` or a `CStr`.

Importing functions defined in a foreign language

In Rust, you can declare that imported functions use a specific ABI, for example “C”, corresponding to the C language. You can indicate that a libary must be linked in, and you can give a different link name for every imported function (*e.g.,* if such link name would conflict with a Rust keyword):

```
1  use std::ffi::{c_char, c_int};
2
3  #[link(name = "c")]
4  extern "C" {
5    // "..." denotes a variable number of arguments
6    fn open(path: *const c_char, oflag: c_int, ...) -> c_int;
7    fn read(fd: c_int, buf: *mut u8, count: c_int) -> c_int;
8
9    #[link_name = "close"]  // e.g., to prevent a name clash with an existing function
10   fn c_close(fd: c_int) -> c_int;
11 }
```

All imported functions are considered *unsafe* by Rust, as the compiler is unable to check that they do not violate Rust safety guarantees.

Exporting Rust functions to a foreign language

Using a similar mechanism, you can export Rust functions to other languages:

```
1 extern "C" fn compute_cost(a: u32, b: u32) -> u32 {
2   a + 3 * b
3 }
```

Such a function can be passed to C code as a function pointer.

It is also possible for C code to call those functions directly by name, provided that they are `pub`. However by default the external name of a Rust function is mangled (i.e., renamed by the compiler), because different modules may export functions with similar names.

Two (unsafe since Rust 1.82) attributes let you control what the external name of the function will be:

- `#[unsafe(no_mangle)]` exports the raw function name (here “compute_cost”).
- `#[unsafe(export_name = "some_name")]` lets you choose a different name for the exported function.

What about variables?

Variables can also be imported in an `extern` block, or exported with a chosen name:

```
1  extern "C" {
2    static mut errno: std::ffi::c_int;
3  }
4
5  #[repr(u32)]
6  enum RustError { NoError, Error1, Error2 };
7
8  #[export_name = "current_rust_error"]
9  static mut CURRENT_ERROR: RustError = RustError::NoError;
```

Note that accessing a `static mut` variable is always an `unsafe` operation, regardless of whether the variable is defined in Rust or imported.

Taking advantage of the null-pointer optimization

Reminder:

- NPO concerns enumerations with an empty variant (*e.g.*, `None`) and one with a value (*e.g.*, `Some`). At least one bit-pattern must not represent a valid value.
- The empty variant (`None`) will use one of the unused bit-pattern.
- The non-empty variant (`Some(T)`) will be bit-compatible with its value (`T`).

Rust references (`&`, `&mut`, `Box`, function pointers) can never contain `NULL`. We can use `Option<T>` where `T` is a reference type in a FFI definition:

```
1 extern "C" {
2   // Set the new callback and return the previous one
3   fn exchange_callback(callback: Option<extern "C" fn()>) -> Option<extern "C" fn()>;
4 }
```

Here, we will have the bidirectional mapping:

- Rust `None` with C `NULL`
- Rust `Some(fn())` to C non-null `*(void fn())`

A complete example

The following program, when run, will print “Received signal 2” when control-C is pressed (2 is `SIGINT` on Linux), then quit when control-C is pressed again:

```rust
use std::ffi::c_int;

type SigHandler = extern "C" fn(c_int);

#[link(name = "c")]
extern "C" {
  fn signal(signum: c_int, handler: Option<SigHandler>) -> Option<SigHandler>;
}

extern "C" fn print_signal(signum: c_int) {
  println!("Received signal {signum}");
  unsafe { signal(2, None) };
}

fn main() {
  unsafe { signal(2, Some(print_signal)) };
  loop {}
}
```

FFI limitations

All language run time executives[1] do not behave the same way. The interactions between languages should be limited to what is defined by the FFI. For example:

- If C++ calls Rust which calls C++, a C++ exception will have a hard-time propagating across the Rust code and reaching the upper C++ layer.
- A Rust `panic!()` will have the same problem going through a foreign language.

In Rust, you may use `std::panic::catch_unwind()` to prevent a `panic!()` from propagating to the caller, but this is more a stopgap solution than a well-thought design, as panics should be avoided in the first place.

[1] The "executive" here is the part which sets up the memory, prepares the arguments to give to the main function, propagates the exception, …

Beyond C

In addition to C, other ABIs are recognized by Rust:

- “Rust”: rarely useful in Rust code…
- “stdcall”: used by Microsoft Win32 ABI
- “system”: selects the right ABI depending on the Windows version
- “x86-interrupt” (experimental): used to write processor interrupt handlers in Rust
- “rust-intrinsic” (experimental): internal

Now that we have all the tools to do so, we can generate bindings for libraries as a whole and use them from Rust.

Binding generation

Generating bindings between programming languages is a tedious job. Tools exist to ease this task:

- bindgen generates Rust code from C and C++ header files. It can be used from the command line or as a library.
- cbindgen generates C, C++, and Python (cython) from Rust code.

Example of a bindgen translation from C to Rust

Starting from the C header to the left, bindgen generates the Rust file shown to the right:

```c
#include <stdint.h>

typedef struct mystruct_s {
  uint8_t c;
  uint32_t i;
  struct mystruct_s *next;
} mystruct_t;

uint8_t process(const mystruct_t *s);
```

```rust
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct mystruct_s {
  pub c: u8,
  pub i: u32,
  pub next: *mut mystruct_s,
}
pub type mystruct_t = mystruct_s;
extern "C" {
  pub fn process(s: *const mystruct_t) -> u8;
}
```

In C, a `struct` can be copied by assignment or when passed to, or returned from, a function. bindgen has automatically derived the `Copy` trait: the Rust `struct` can be used as a function parameter, or stored, without losing ownership of the original value, as a copy will be made automatically.

Consistency between header file and library

When distributing a library, you might want to simultaneously attain the following goals:

- You want the user to type `cargo build` or `cargo install`, you don't want to force them to install bindgen and type an extra `bindgen` command.
- You don't want to distribute a pre-computed bindgen output, because the user might not have the same version of the library as you do, and some layout details may be different.

Fortunately, `cargo` allows the use of a *build script* which will be compiled and executed before your crate is compiled. The script is located in `build.rs`, at the root of the project. It can control various aspects of the build process.

To use `bindgen` when building your crate, you must add it as a *build dependency* to `Cargo.toml`:

```
1 [build-dependencies]
2 bindgen = "0.69.5"
```

Build-scripts

Buid-script can influence:

- what options will be given to the compiler and to the linker;
- the libraries to link along with the binaries in your project or in the projects that use yours;
- the files and environment variables to watch and determine if the build-script needs to be executed again;
- warnings given to the user;
- environment variables accessible throughout the compilation process.

Communication happens through the script's standard output, *e.g.,* by using `println!()`.

Build script example: interfacing with an ILDA decoder (laser show)

After installing the ilda-decoder library, we will buid a `ilda-decoder-sys` crate. Here is the content of its `build.rs`:

```rust
use std::{env, path::PathBuf};

fn main() {
    println!("cargo:rustc-link-lib=ilda-decoder");
    println!("cargo:rerun-if-changed=src/header.h");
    let bindings = bindgen::Builder::default()
        .header("src/header.h")       // Includes <ilda-decoder.h>
        .parse_callbacks(Box::new(bindgen::CargoCallbacks))
        .allowlist_type("ilda.*")     // Do not generate non-necessary types
        .allowlist_function("ilda.*") // Same for non-necessary functions
        .generate()
        .expect("Unable to generate bindings");
    let out_path = PathBuf::from(env::var("OUT_DIR").unwrap());
    bindings.write_to_file(out_path.join("bindings.rs"))
        .expect("Couldn't write bindings!");
}
```

Explaining the build script

The following choices have been made in the `ilda-decoder-sys` crate:

- A header `src/header.h` which includes `<ilda-decoder.h>` has been created. This way, we do not have to guess the path of `ilda-decoder.h` and we can let the C compiler (through bindgen) do its job.
- Callbacks used in the `bindgen` invocation tell `cargo` to watch any file included through `src/header.h`, transitively.
- Types and functions produced by bindgen will b restricted to those starting with “ilda”, as well as the types used transitively. If we didn’t restrict this list, all items present in referenced header files (such as `<stdlib.h>`) would have been translated.
- bindgen will write the bindings in the directory referenced by the `OUT_DIR` environment variable set by `cargo`. This directory is located under `target/` and will not pollute the source directories.

In addition, bindgen will generate layout tests. Those tests will check that the corresponding Rust structures have the same size and right alignment constraints as expected by bindgen when doing the translation.

ILDA: integrating bindgen output in the crate source code

This `ilda-decoder-sys` crate containg low-level bindings to the `ilda-decoder` C library. Here is the content of `src/lib.rs`:

```
1  #![allow(non_camel_case_types)]
2  #![allow(unused)]
3  #![cfg_attr(test, allow(deref_nullptr))]   // Necessary for the layout tests
4
5  include!(concat!(env!("OUT_DIR"), "/bindings.rs"));
```

You may want to use `cargo expand` (install it with `cargo install cargo-expand` first) to see the result of the `include!` macro.

Typically, another crate called `ilda-decoder` will add a higher-level interface to `ilda-decoder-sys`. Ideally, this new crate will contain no exported unsafe functions.

Summary: `unsafe` and FFI

- `unsafe` lets you manipulate entities in a way that the compiler is not able to check.
- Using `unsafe` does not allow you to violate Rust safety guarantees. For example, a reference must never be `NULL` or point to deallocated data.
- You can mark a function `unsafe` to make the caller aware of possible guarantee violations if they do not respect the stated conditions.
- You must use `unsafe` when calling a function defined in another language as the compiler.
- You can easily interface with C and C++ code using `bindgen`.
- Build scripts allow you to call bindgen during the package building process.
- When interfacing with a library written in another language, you may want to split the bindings into two crates. One crate will contain the low-level bindings and its name will be suffixed by “-sys”, and one crate will contain more idiomatic and safe Rust bindings by calling the first one behind the scenes.

What other powerful and dangerous tool could be used in Rust code? Let’s meet procedural macros.

Procedural macros in Rust

Macros

Rust has two families of macros:

- Pattern-based macros, in which the macro arguments are matched against a pattern and expanded. They do not constitute any particular security risk:

```
1  macro_rules! my_double_vec {
2      [$($x: expr),*] => {{            // Accept zero or more expressions separated by a comma
3          let mut v = vec![];
4          $(v.push($x); v.push($x);)*
5          v
6      }}
7  }
8
9  fn main() {
10     let v = my_double_vec![10, 20, 30];
11     println!("v = {v:?}");           // v = [10, 10, 20, 20, 30, 30]
12 }
```

- Procedural macros, which take a piece of source code and return another to be compiled, after processing it at compile time.

What is a procedural macro?

A procedural macro roughly looks like:

```
1 pub fn code_of_the_macro(item: TokenStream) -> TokenStream {
2   …
3 }
```

A `TokenStream` is a list of `TokenTree`, and each `TokenTree` value is either:

- an `Ident` — identifier (`foobar`, `fn`, `use`),
- or a `Literal` – literal (`42`, `"Hello, world!"`),
- or a `Punct` – single punctuation character (`.`, `;`, `-`, `>`),
- or a `Group` – a `TokenStream` surrounded by bracket delimiters (`()`, `[]` or `{}`).

For example, `vec![10, 20]` is represented as `Ident("vec")`, `Punct('!')`, `Group(g)` where `g` is a group with `[]` delimiters containing `Literal(10)`, `Punct(',')`, `Literal(20)`.

Every `TokenTree` holds location (*span*) information. However, no semantic data (“Is the identifier a keyword? If it is a variable, where is it defined?”) are attached.

Procedural macro example

The following (silly) procedural macro defined in the `powerful_macros` crate:

```rust
use proc_macro::TokenStream;

#[proc_macro]
pub fn rev(t: TokenStream) -> TokenStream {
    let v = t.into_iter().collect::<Vec<_>>();
    v.into_iter().rev().collect()
}
```

can be used in:

```rust
use power_macros::rev;

fn main() {
    rev!(("world!")!println;("Hello, ")!print);
}
```

and will print

```
$ cargo -q run
Hello, world!
```

Three kinds of procedural macros

- The “regular” macros, called as `rev!(…)`:

```rust
#[proc_macro]
pub fn rev(t: TokenStream) -> TokenStream { … }
```

- The “attribute” macros:

```rust
#[proc_macro_attr]
pub fn something(t: TokenStream, args: TokenStream) -> TokenStream { … }

#[something(some, extra, args => free form here)]
pub fn function_on_which_something_is_applied() { … }
```

- The “derive” macros. While other procedural macros replace the `TokenStream` they receive by the one they return, derive macros will only generate additional code (*e.g,* new `impl` blocks):

```rust
#[proc_macro_derive]
pub fn SomeProperty(t: TokenStream) -> TokenStream { … }   // Note the unusual name casing

#[derive(SomeProperty)]
pub struct Foobar { … }
```

Analyzing a `TokenStream` is tedious

There are crates available to parse and generate `TokenStream` more easily, notably:

- `proc_macro2`: define the same types (`TokenStream`, `TokenTree`) as `proc_macro` (with methods to convert between them), but in a real crate, usable in other contexts such as `build.rs` scripts
- `syn`: parse a `TokenStream` (or `proc_macro2::TokenStream`) into a AST[2] representing a Rust entity, into a custom type implementing a parser, or token by token according to a grammar defined by hand
- `quote`: generate a `proc_macro2::TokenStream` from a template, with powerful expansion capabilities

[2] Abstract Syntax Tree

Using `syn` and `quote`

This example uses `syn` to parse the macro argument as a `LitInt` (literal integer), extracts its value and returns a new stream with the multiples fitting in a `u8`.

```
/// Generate a vector of multiples of its arguments that fit into a u8
#[proc_macro]
pub fn small_multiples(t: proc_macro::TokenStream) -> proc_macro::TokenStream {
    let xs: u8 = syn::parse_macro_input!(t as syn::LitInt).base10_parse().unwrap();
    let multiples = std::iter::successors(Some(xs), |p| p.checked_add(xs)).collect::<Vec<_>>();
    quote::quote! { vec![#(#multiples),*] }.into()     // Generate a comma separated list
}
```

```
fn main() {
    println!("Multiples of 17: {:?}", power_macros::small_multiples!(17));
}
```

```
$ cargo run -q
Multiples of 17: [17, 34, 51, 68, 85, 102, 119, 136, 153, 170, 187, 204, 221, 238, 255]
```

💡 Note that in `syn::parse_macro_input!(t as syn::LitInt)`, the `as` does not denote a type conversion, this is part of a mini-language created for `parse_macro_input!()`.

syn/quote: let async

Let's say we want to parse and transform the following constructs:

```
let async v = expr;      // let v = async { expr };
let async mut v = expr; // let mut v = async { expr };
```

and

```
let async v: T = expr;      // let v = { async fn func() -> T = { expr }; func() };
let async mut v: T = expr; // let mut v = { async fn func() -> T = { expr }; func() };
```

by parsing the input with `syn` and generating the output with `quote`, using the following intermediate structure:

```
struct LetAsync {      // let async [mut] ident [: type] = expr;
  is_mut: bool,
  ident: Ident,
  typ: Option<Type>,
  expr: Expr,
}
```

syn: Using parsers

```rust
struct LetAsync {     // let async [mut] ident [: type] = expr;
  is_mut: bool,
  ident: Ident,
  typ: Option<Type>,
  expr: Expr,
}

impl Parse for LetAsync {
  fn parse(input: ParseStream) -> syn::Result<Self> {
    input.parse::<Token![let]>()?;
    input.parse::<Token![async]>()?;
    let is_mut = input.parse::<Option<Token![mut]>>()?.is_some();
    let ident: Ident = input.parse()?;
    let typ: Option<Type> = input.parse::<Option<Token![:]>>()?
      .map(|_| input.parse()).transpose()?;
    input.parse::<Token![=]>()?;
    let expr: Expr = input.parse()?;
    input.parse::<Token![;]>()?;
    Ok(LetAsync { is_mut, ident, typ, expr, })
  }
}
```

quote: Implementing ToTokens

```rust
struct LetAsync {      // let async [mut] ident [: type] = expr;
  is_mut: bool,
  ident: Ident,
  typ: Option<Type>,
  expr: Expr,
}

impl ToTokens for LetAsync {
  fn to_tokens(&self, tokens: &mut proc_macro2::TokenStream) {
    let expr = &self.expr;
    let expr = if let Some(typ) = &self.typ {
      quote! { { async fn func() -> #typ { #expr }; func() }}
    } else {
      quote! { async { #expr } }
    };
    let maybe_mut = self.is_mut.then(|| quote!(mut));
    let ident = &self.ident;
    quote! { let #maybe_mut #ident = #expr; }.to_tokens(tokens);
  }
}
```

syn: using visitors

The `syn::visit::Visit` and `syn::visit_mut::VisitMut` visitor traits define default methods for all nodes in the AST that delegate to public utility functions which by default traverse the tree:

```
1  trait Visit<'ast> {
2    fn visit_expr_if(&mut self, i: &'ast ExprIf) {
3      visit_expr_if(self, i);
4    }
5  }
6
7  pub fn visit_expr_if<'ast, V>(v: &mut V, node: &'ast ExprIf) where
8      V: Visit<'ast> + ?Sized
9  {
10   for it in &node.attrs { v.visit_attribute(it); }
11   v.visit_expr(&*node.cond);
12   v.visit_block(&node.then_branch);
13   if let Some(it) = &node.else_branch { v.visit_expr(&*(it).1); }
14 }
```

syn: visitor example 1/2

```rust
struct Increment;

impl VisitMut for Increment {
  fn visit_lit_int_mut(&mut self, lit: &mut syn::LitInt) {
    let value = lit.base10_parse::<i128>().unwrap();
    lit.span().unwrap().warning(format!("Trouvé la constante {}",
      french_number(&value))).emit();
    *lit = syn::LitInt::new(&format!("{}", value + 1), lit.span());
  }
}

#[proc_macro_attribute]
/// Increment any literal integers by 1 and issue a warning
pub fn increment(_attr: TokenStream, tokens: TokenStream) -> TokenStream {
  let mut item = parse_macro_input!(tokens as Item);
  Increment.visit_item_mut(&mut item);
  quote!(#item).into()
}
```

syn: visitor example 2/2

```rust
#[increment]
fn add_three(x: i32) -> i32 { x + 3 }

fn main() { println!("add_three(10) = {}", add_three(10)); }
```

generates the following warning at compilation:

```
warning: Trouvé la constante 3
 --> src/main.rs:2:35
  |
2 | fn add_three(x: i32) -> i32 { x + 3 }
  |                                   ^
```

and at runtime

```
add_three(10) = 14
```

If this attribute is applied to a module, all integers in that module will be incremented.

Procedural macros and build scripts security

- Procedural macros and build scripts have a complete access to the compilation environment and can execute arbitrary commands.
- They can inject code into the library or the executable being compiled.
- They are however pervasive:
  - Almost all crates implementing bindings use `bindgen` through a build script.
  - Almost all crates implementing procedural macros depend on `syn` and `quote` which themselves use procedural macros heavily.
  - The omnipresent serialization framework `serde` works by tagging types with `#[derive(Serialize)]` and `#[derive(Unserialize)]` (procedural macros).

For example, in August 2023, the `serde` maintainer has covertly modified `serde`'s build script so that a portion of the `serde` library was retrieved as a binary blob from one of their servers instead of being compiled locally. It took six weeks before this was noticed.

We will discuss these risks further in a few weeks during the “Open Source Software Security” lecture.