Rust Type Layout

Size, Alignment, Endianess, and Representations
vs
Input, Output, and Foreign Functions

Objective: Binary Compatible

I want to be able to exchange data structures between

- multiple instances of my application running on different processor architectures like x86-64, ARM, Power, RISC-V, and s390x (https://doc.rust-lang.org/rustc/platform-support.html)
- my application and other applications – possibly written in other programming languages.

Size depends on the Platform

Fixed-Size Types

Some primitive types have fixed sizes.

Type	Size in 8-bit bytes	Description
i8 u8 bool	1	8-bit signed two's complement binary integer 8-bit unsigned binary integer Boolean true or false
i16 u16	2	16-bit signed two's complement binary integer 16-bit unsigned binary integer
i32 u32 f32 char	4	32-bit signed two's complement binary integer 32-bit unsigned binary integer 32-bit binary IEEE 754-2008 floating-point number UTF-32 Unicode character
i64 u64 f64	8	64-bit signed two's complement binary integer 64-bit unsigned binary integer 64-bit binary IEEE 754-2008 floating-point number
i128 u128	16	128-bit signed two's complement binary integer 128-bit unsigned binary integer
()	0	Unit type

Varying-Size Types

For other primitive types their sizes depend on the platform.

Type	Size in 8-bit bytes	Description
isize	?	signed binary integer size
usize	?	unsigned binary integer size
fn	?	function pointer
*T	?	pointer
&T	?	reference
Box<T>	?	box
Option<T>	?	option
Option<Box<T>>	?	optional box

Size Conclusion

Use only fixed-size types in external data structures.

Get their sizes via std::mem::size_of<T>().

While size_of<T>() is a constant function, for varying-size types it can return different results on different architectures and theoretically even between different Rust releases.

Alignment depends on the Platform

Alignment depends on Platform

The alignment depends entirely on the platform.

The function std::mem::align_of<T>() returns the minimum required alignment in structs for the type T.

- This can differ from the preferred alignment.
- There is no function in std::mem to obtain this preferred alignment.
- See https://doc.rust-lang.org/std/mem/fn.align_of.html

For example, on the x86-64 architecture the 128-bit binary integer types are aligned on 64 bits.

std::alloc::Layout supports alignment on powers of 2.

Alignment Conclusions

You can determine the minimum required alignment simply and quickly by running std::mem::align_of<T>() for each primitive type on each platform that is relevant for you.

std::mem::align_of<T>() <= std::mem::size_of<T>() holds true because of the definition of size_of. Therefore it is safe to align types on their size.

To determine the preferred alignment, you have to read the platform specification.

Endianess depends on the Platform

Endianess depends on Platform

Integer primitive types longer than 1 byte can be big or little endian. So far Rust does not list any mixed-endian platforms.

The methods from_be(x), from_le(x), to_be(), and to_le() make the endianess explicit, but work on the same integer type only.

The methods from_be_bytes(x), from_le_bytes(x), to_be_bytes(), and to_le_bytes() are defined for byte arrays ([u8; n]) only.

There are no endianess functions that work on byte slices ([u8]).

Endianess Conclusions

Explicitly specify the endianess of external integer fields via the methods from_be(x), from_le(x), to_be(), and to_le().

Explicitly specify the endianess of external integer fields in byte buffers via the methods from_be_bytes(bytes), from_le_bytes(bytes), to_be_bytes(), and to_le_bytes().

Representations

Representations

The standard Rust representation guarantees only soundness regarding the type layout of user defined composite types (structs, enums, and unions).

The C representation guarantees platform-independent ordering and alignment, but the size of enums is excepted.

The packed representation has defined alignments, but can create unaligned items.

See https://doc.rust-lang.org/reference/type-layout.html

Representation Conclusions

Use the C representation repr(c) to enforce the ordering and alignment of fields within structures.

Explicitly specify reserved fields instead of relying on the insertion of padding bytes to properly align fields.

Input and Output of Bytes only

Input and Output Traits

std::io::Read reads into byte slices and byte vectors.

std::io::Write writes from byte slices.

std::fs::File implements Read and Write.

std::net::TcpStream implements Read and Write.

Byte slices have an alignment of 1.

Input and Output Conclusions

There seem to be no safe interfaces in std::io for reading into or writing from slices of primitive types other than u8.

Safely reading other primitive types requires constructing them from bytes after reading these.

Safely writing other primitive types requires converting them to bytes before writing these.

Foreign Function Interface

External Functions

Calls from Rust to non-Rust call functions declared in "extern" blocks with the specified ABI (e.g. "C"). Functions in "extern" blocks are always "unsafe", because the Rust compiler could not check them.

Calls from non-Rust to Rust call "extern" functions with the specified ABI. Extern functions are safe by default, but require the attribute #[no_mangle] to get stable external symbols.

See https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html

External Function Conclusion

Non-Rust code is always unsafe from the perspective of Rust.

You can avoid unsafe code by using IO instead of calls.

IO comes with the overhead of system calls to the OS kernel.

Also the OS kernel is potentially unsafe, unless the kernel is implemented in Rust as well, like in the Redox OS.

On architectures with memory protection you can isolate unsafe code to separate processes.

End of Story

Comments or Questions?