Crate std

Since 1.0.0 ·

Settings Help Summary

The Rust Standard Library

The Rust Standard Library is the foundation of portable Rust software, a set of minimal and battle-tested shared abstractions for the broader Rust ecosystem. It offers core types, like `Vec<T>` and `Option<T>`, library-defined operations on language primitives, standard macros, I/O and multithreading, among many other things.

`std` is available to all Rust crates by default. Therefore, the standard library can be accessed in `use` statements through the path `std`, as in `use std::env`.

How to read this documentation

If you already know the name of what you are looking for, the fastest way to find it is to use the search bar at the top of the page.

Otherwise, you may want to jump to one of these useful sections:

- `std::*` modules
- Primitive types
- Standard macros
- The Rust Prelude

If this is your first time, the documentation for the standard library is written to be casually perused. Clicking on interesting things should generally lead you to interesting places. Still, there are important bits you don't want to miss, so read on for a tour of the standard library and its documentation!

Once you are familiar with the contents of the standard library you may begin to find the verbosity of the prose distracting. At this stage in your development you may want to press the "» Summary" button near the top of the page to collapse it into a more skimmable view.

While you are looking at the top of the page, also notice the "Source" link. Rust's API documentation comes with the source code and you are encouraged to read it. The standard library source is generally high quality and a peek behind the curtains is often enlightening.

What is in the standard library documentation?

First of all, The Rust Standard Library is divided into a number of focused modules, all listed further down this page. These modules are the bedrock upon which all of Rust is forged, and they

have mighty names like `std::slice` and `std::cmp`. Modules' documentation typically includes an overview of the module along with examples, and are a smart place to start familiarizing yourself with the library.

Second, implicit methods on primitive types are documented here. This can be a source of confusion for two reasons:

1. While primitives are implemented by the compiler, the standard library implements methods directly on the primitive types (and it is the only library that does so), which are documented in the section on primitives.
2. The standard library exports many modules *with the same name as primitive types*. These define additional items related to the primitive type, but not the all-important methods.

So for example there is a page for the primitive type `i32` that lists all the methods that can be called on 32-bit integers (very useful), and there is a page for the module `std::i32` that documents the constant values `MIN` and `MAX` (rarely useful).

Note the documentation for the primitives `str` and `[T]` (also called 'slice'). Many method calls on `String` and `Vec<T>` are actually calls to methods on `str` and `[T]` respectively, via deref coercions.

Third, the standard library defines The Rust Prelude, a small collection of items - mostly traits - that are imported into every module of every crate. The traits in the prelude are pervasive, making the prelude documentation a good entry point to learning about the library.

And finally, the standard library exports a number of standard macros, and lists them on this page (technically, not all of the standard macros are defined by the standard library - some are defined by the compiler - but they are documented here the same). Like the prelude, the standard macros are imported by default into all crates.

Contributing changes to the documentation

Check out the Rust contribution guidelines here. The source for this documentation can be found on GitHub in the 'library/std/' directory. To contribute changes, make sure you read the guidelines first, then submit pull-requests for your suggested changes.

Contributions are appreciated! If you see a part of the docs that can be improved, submit a PR, or chat with us first on Discord #docs.

A Tour of The Rust Standard Library

The rest of this crate documentation is dedicated to pointing out notable features of The Rust Standard Library.

Containers and collections

The `option` and `result` modules define optional and error-handling types, `Option<T>` and `Result<T, E>`. The `iter` module defines Rust's iterator trait, `Iterator`, which works with the `for` loop to access collections.

The standard library exposes three common ways to deal with contiguous regions of memory:

- `Vec<T>` - A heap-allocated *vector* that is resizable at runtime.
- `[T; N]` - An inline *array* with a fixed size at compile time.
- `[T]` - A dynamically sized *slice* into any other kind of contiguous storage, whether heap-allocated or not.

Slices can only be handled through some kind of *pointer*, and as such come in many flavors such as:

- `&[T]` - *shared slice*
- `&mut [T]` - *mutable slice*
- `Box<[T]>` - *owned slice*

`str`, a UTF-8 string slice, is a primitive type, and the standard library defines many methods for it. Rust `str`s are typically accessed as immutable references: `&str`. Use the owned `String` for building and mutating strings.

For converting to strings use the `format!` macro, and for converting from strings use the `FromStr` trait.

Data may be shared by placing it in a reference-counted box or the `Rc` type, and if further contained in a `Cell` or `RefCell`, may be mutated as well as shared. Likewise, in a concurrent setting it is common to pair an atomically-reference-counted box, `Arc`, with a `Mutex` to get the same effect.

The `collections` module defines maps, sets, linked lists and other typical collection types, including the common `HashMap<K, V>`.

Platform abstractions and I/O

Besides basic data types, the standard library is largely concerned with abstracting over differences in common platforms, most notably Windows and Unix derivatives.

Common types of I/O, including files, TCP, and UDP, are defined in the `io`, `fs`, and `net` modules.

The `thread` module contains Rust's threading abstractions. `sync` contains further primitive shared memory types, including `atomic`, `mpmc` and `mpsc`, which contains the channel types for message passing.

Use before and after `main()`

Many parts of the standard library are expected to work before and after `main()`; but this is not guaranteed or ensured by tests. It is recommended that you write your own tests and run them on each platform you wish to support. This means that use of `std` before/after main, especially of features that interact with the OS or global state, is exempted from stability and portability guarantees and instead only provided on a best-effort basis. Nevertheless bug reports are appreciated.

On the other hand `core` and `alloc` are most likely to work in such environments with the caveat that any hookable behavior such as panics, oom handling or allocators will also depend on the compatibility of the hooks.

Some features may also behave differently outside main, e.g. stdio could become unbuffered, some panics might turn into aborts, backtraces might not get symbolicated or similar.

Non-exhaustive list of known limitations:

- after-main use of thread-locals, which also affects additional features:
  - `thread::current()`
- under UNIX, before main, file descriptors 0, 1, and 2 may be unchanged (they are guaranteed to be open during main, and are opened to /dev/null O_RDWR if they weren't open on program start)

Primitive Types

array	A fixed-size array, denoted `[T; N]`, for the element type, `T`, and the non-negative compile-time constant size, `N`.
bool	The boolean type.
char	A character type.
f32	A 32-bit floating-point type (specifically, the "binary32" type defined in IEEE 754-2008).
f64	A 64-bit floating-point type (specifically, the "binary64" type defined in IEEE 754-2008).
fn	Function pointers, like `fn(usize) -> bool`.
i8	The 8-bit signed integer type.
i16	The 16-bit signed integer type.
i32	The 32-bit signed integer type.
i64	The 64-bit signed integer type.
i128	The 128-bit signed integer type.
isize	The pointer-sized signed integer type.
pointer	Raw, unsafe pointers, `*const T`, and `*mut T`.
reference	References, `&T` and `&mut T`.
slice	A dynamically-sized view into a contiguous sequence, `[T]`.
str	String slices.

tuple	A finite heterogeneous sequence, `(T, U, ..)`.
u8	The 8-bit unsigned integer type.
u16	The 16-bit unsigned integer type.
u32	The 32-bit unsigned integer type.
u64	The 64-bit unsigned integer type.
u128	The 128-bit unsigned integer type.
unit	The `()` type, also called “unit”.
usize	The pointer-sized unsigned integer type.
f16 Experimental	A 16-bit floating-point type (specifically, the “binary16” type defined in IEEE 754-2008).
f128 Experimental	A 128-bit floating-point type (specifically, the “binary128” type defined in IEEE 754-2008).
never Experimental	The `!` type, also called “never”.

Modules

alloc	Memory allocation APIs.
any	Utilities for dynamic typing or type reflection.
arch	SIMD and vendor intrinsics module.
array	Utilities for the array primitive type.
ascii	Operations on ASCII strings and characters.
backtrace	Support for capturing a stack backtrace of an OS thread
borrow	A module for working with borrowed data.
boxed	The `Box<T>` type for heap allocation.
cell	Shareable mutable containers.
char	Utilities for the `char` primitive type.
clone	The `Clone` trait for types that cannot be ‘implicitly copied’.
cmp	Utilities for comparing and ordering values.
collections	Collection types.
convert	Traits for conversions between types.
default	The `Default` trait for types with a default value.
env	Inspection and manipulation of the process’s environment.
error	Interfaces for working with Errors.
f32	Constants for the `f32` single-precision floating point type.
f64	Constants for the `f64` double-precision floating point type.
ffi	Utilities related to FFI bindings.
fmt	Utilities for formatting and printing `Strings`.
fs	Filesystem manipulation operations.
future	Asynchronous basic functionality.
hash	Generic hashing support.

hint	Hints to compiler that affects how code should be emitted or optimized.
i8 Deprecation planned	Redundant constants module for the `i8` primitive type.
i16 Deprecation planned	Redundant constants module for the `i16` primitive type.
i32 Deprecation planned	Redundant constants module for the `i32` primitive type.
i64 Deprecation planned	Redundant constants module for the `i64` primitive type.
i128 Deprecation planned	Redundant constants module for the `i128` primitive type.
io	Traits, helpers, and type definitions for core I/O functionality.
isize Deprecation planned	Redundant constants module for the `isize` primitive type.
iter	Composable external iteration.
marker	Primitive traits and types representing basic properties of types.
mem	Basic functions for dealing with memory.
net	Networking primitives for TCP/UDP communication.
num	Additional functionality for numerics.
ops	Overloadable operators.
option	Optional values.
os	OS-specific functionality.
panic	Panic support in the standard library.
path	Cross-platform path manipulation.
pin	Types that pin data to a location in memory.
prelude	The Rust Prelude
primitive	This module reexports the primitive types to allow usage that is not possibly shadowed by other declared types.
process	A module for working with processes.
ptr	Manually manage memory through raw pointers.
rc	Single-threaded reference-counting pointers. 'Rc' stands for 'Reference Counted'.
result	Error handling with the `Result` type.
slice	Utilities for the slice primitive type.
str	Utilities for the `str` primitive type.
string	A UTF-8–encoded, growable string.
sync	Useful synchronization primitives.
task	Types and Traits for working with asynchronous tasks.
thread	Native threads.
time	Temporal quantification.
u8 Deprecation planned	Redundant constants module for the `u8` primitive type.
u16 Deprecation planned	Redundant constants module for the `u16` primitive type.
u32 Deprecation planned	Redundant constants module for the `u32` primitive type.
u64 Deprecation planned	Redundant constants module for the `u64` primitive type.
u128 Deprecation planned	Redundant constants module for the `u128` primitive type.
usize Deprecation planned	Redundant constants module for the `usize` primitive type.

vec	A contiguous growable array type with heap-allocated contents, written `Vec<T>`.
assert_matches Experimental	Unstable module containing the unstable `assert_matches` macro.
async_iter Experimental	Composable asynchronous iteration.
autodiff Experimental	This module provides support for automatic differentiation.
bstr Experimental	The `ByteStr` and `ByteString` types and trait implementations.
f16 Experimental	Constants for the `f16` half-precision floating point type.
f128 Experimental	Constants for the `f128` quadruple-precision floating point type.
intrinsics Experimental	Compiler intrinsics.
pat Experimental	Helper module for exporting the `pattern_type` macro
random Experimental	Random value generation.
range Experimental	Experimental replacement range types
simd Experimental	Portable SIMD module.
unsafe_binder Experimental	Operators used to turn types into unsafe binders and back.

Macros

assert	Asserts that a boolean expression is `true` at runtime.
assert_eq	Asserts that two expressions are equal to each other (using `PartialEq`).
assert_ne	Asserts that two expressions are not equal to each other (using `PartialEq`).
cfg	Evaluates boolean combinations of configuration flags at compile-time.
column	Expands to the column number at which it was invoked.
compile_error	Causes compilation to fail with the given error message when encountered.
concat	Concatenates literals into a static string slice.
dbg	Prints and returns the value of a given expression for quick and dirty debugging.
debug_assert	Asserts that a boolean expression is `true` at runtime.
debug_assert_eq	Asserts that two expressions are equal to each other.
debug_assert_ne	Asserts that two expressions are not equal to each other.
env	Inspects an environment variable at compile time.
eprint	Prints to the standard error.
eprintln	Prints to the standard error, with a newline.
file	Expands to the file name in which it was invoked.
format	Creates a `String` using interpolation of runtime expressions.
format_args	Constructs parameters for the other string-formatting macros.
include	Parses a file as an expression or an item according to the context.

include_bytes	Includes a file as a reference to a byte array.
include_str	Includes a UTF-8 encoded file as a string.
is_x86_feature_detected	A macro to test at *runtime* whether a CPU feature is available on x86/x86-64 platforms.
line	Expands to the line number on which it was invoked.
matches	Returns whether the given expression matches the provided pattern.
module_path	Expands to a string that represents the current module path.
option_env	Optionally inspects an environment variable at compile time.
panic	Panics the current thread.
print	Prints to the standard output.
println	Prints to the standard output, with a newline.
stringify	Stringifies its arguments.
thread_local	Declare a new thread local storage key of type `std::thread::LocalKey`.
todo	Indicates unfinished code.
try Deprecated	Unwraps a result or propagates its error.
unimplemented	Indicates unimplemented code by panicking with a message of "not implemented".
unreachable	Indicates unreachable code.
vec	Creates a `Vec` containing the arguments.
write	Writes formatted data into a buffer.
writeln	Writes formatted data into a buffer, with a newline appended.
cfg_match Experimental	A macro for defining `#[cfg]` match-like statements.
concat_bytes Experimental	Concatenates literals into a byte slice.
concat_idents Experimental	Concatenates identifiers into one identifier.
const_format_args Experimental	Same as `format_args`, but can be used in some const contexts.
format_args_nl Experimental	Same as `format_args`, but adds a newline in the end.
log_syntax Experimental	Prints passed tokens into the standard output.
trace_macros Experimental	Enables or disables tracing functionality used for debugging other macros.

Keywords

SelfTy	The implementing type within a `trait` or `impl` block, or the current type within a type definition.
as	Cast between types, or rename an import.
async	Returns a `Future` instead of blocking the current thread.
await	Suspend execution until the result of a `Future` is ready.
break	Exit early from a loop or labelled block.

const	Compile-time constants, compile-time evaluable functions, and raw pointers.
continue	Skip to the next iteration of a loop.
crate	A Rust binary or library.
dyn	`dyn` is a prefix of a trait object's type.
else	What expression to evaluate when an `if` condition evaluates to `false`.
enum	A type that can be any one of several variants.
extern	Link to or import external code.
false	A value of type `bool` representing logical **false**.
fn	A function or function pointer.
for	Iteration with `in`, trait implementation with `impl`, or higher-ranked trait bounds (`for<'a>`).
if	Evaluate a block if a condition holds.
impl	Implementations of functionality for a type, or a type implementing some functionality.
in	Iterate over a series of values with `for`.
let	Bind a value to a variable.
loop	Loop indefinitely.
match	Control flow based on pattern matching.
mod	Organize code into modules.
move	Capture a closure's environment by value.
mut	A mutable variable, reference, or pointer.
pub	Make an item visible to others.
ref	Bind by reference during pattern matching.
return	Returns a value from a function.
self	The receiver of a method, or the current module.
static	A static item is a value which is valid for the entire duration of your program (a `'static` lifetime).
struct	A type that is composed of other types.
super	The parent of the current module.
trait	A common interface for a group of types.
true	A value of type `bool` representing logical **true**.
type	Define an alias for an existing type.
union	The Rust equivalent of a C-style union.
unsafe	Code or interfaces whose memory safety cannot be verified by the type system.
use	Import or rename items from other crates or modules, use values under ergonomic clones semantic, or specify precise capturing with `use<..>`.
where	Add constraints that must be upheld to use an item.
while	Loop while a condition is upheld.