Module std::io

1.0.0 [−][src]

[−] Traits, helpers, and type definitions for core I/O functionality.

The `std::io` module contains a number of common things you'll need when doing input and output. The most core part of this module is the `Read` and `Write` traits, which provide the most general interface for reading and writing input and output.

Read and Write

Because they are traits, `Read` and `Write` are implemented by a number of other types, and you can implement them for your types too. As such, you'll see a few different types of I/O throughout the documentation in this module: `File`s, `TcpStream`s, and sometimes even `Vec<T>`s. For example, `Read` adds a `read` method, which we can use on `File`s:

```
use std::io;
use std::io::prelude::*;
use std::fs::File;

fn main() -> io::Result<()> {
    let mut f = File::open("foo.txt")?;
    let mut buffer = [0; 10];

    // read up to 10 bytes
    let n = f.read(&mut buffer)?;

    println!("The bytes: {:?}", &buffer[..n]);
    Ok(())
}
```

Run

`Read` and `Write` are so important, implementors of the two traits have a nickname: readers and writers. So you'll sometimes see 'a reader' instead of 'a type that implements the `Read` trait'. Much easier!

Seek and BufRead

Beyond that, there are two important traits that are provided: `Seek` and `BufRead`. Both of these build on top of a reader to control how the reading happens. `Seek` lets you control where the next byte is coming from:

```
use std::io;
use std::io::prelude::*;
use std::io::SeekFrom;
use std::fs::File;

fn main() -> io::Result<()> {
    let mut f = File::open("foo.txt")?;
    let mut buffer = [0; 10];

    // skip to the last 10 bytes of the file
    f.seek(SeekFrom::End(-10))?;

    // read up to 10 bytes
    let n = f.read(&mut buffer)?;

    println!("The bytes: {:?}", &buffer[..n]);
    Ok(())
}
```

Run

`BufRead` uses an internal buffer to provide a number of other ways to read, but to show it off, we'll need to talk about buffers in general. Keep reading!

BufReader and BufWriter

Byte-based interfaces are unwieldy and can be inefficient, as we'd need to be making near-constant calls to the operating system. To help with this, `std::io` comes with two structs, `BufReader` and `BufWriter`, which wrap readers and writers. The wrapper uses a buffer, reducing the number of calls and providing nicer methods for accessing exactly what you want.

For example, `BufReader` works with the `BufRead` trait to add extra methods to any reader:

```
use std::io;
use std::io::prelude::*;
use std::io::BufReader;
use std::fs::File;

fn main() -> io::Result<()> {
    let f = File::open("foo.txt")?;
    let mut reader = BufReader::new(f);
    let mut buffer = String::new();

    // read a line into buffer
    reader.read_line(&mut buffer)?;

    println!("{}", buffer);
    Ok(())
}
```

Run

`BufWriter` doesn't add any new ways of writing; it just buffers every call to `write`:

```
use std::io;
use std::io::prelude::*;
use std::io::BufWriter;
use std::fs::File;

fn main() -> io::Result<()> {
    let f = File::create("foo.txt")?;
    {
        let mut writer = BufWriter::new(f);

        // write a byte to the buffer
        writer.write(&[42])?;

    } // the buffer is flushed once writer goes out of scope

    Ok(())
}
```

Run

Standard input and output

A very common source of input is standard input:

```
use std::io;

fn main() -> io::Result<()> {
    let mut input = String::new();

    io::stdin().read_line(&mut input)?;

    println!("You typed: {}", input.trim());
    Ok(())
}
```

Run

Note that you cannot use the `?` operator in functions that do not return a `Result<T, E>`. Instead, you can call `.unwrap()` or `match` on the return value to catch any possible errors:

```
use std::io;

let mut input = String::new();

io::stdin().read_line(&mut input).unwrap();
```

Run

And a very common source of output is standard output:

```
use std::io;
use std::io::prelude::*;

fn main() -> io::Result<()> {
    io::stdout().write(&[42])?;
    Ok(())
}
```

Run

Of course, using `io::stdout` directly is less common than something like `println!`.

Iterator types

A large number of the structures provided by `std::io` are for various ways of iterating over I/O. For example, `Lines` is used to split over lines:

```
use std::io;
use std::io::prelude::*;
use std::io::BufReader;
use std::fs::File;

fn main() -> io::Result<()> {
    let f = File::open("foo.txt")?;
    let reader = BufReader::new(f);

    for line in reader.lines() {
        println!("{}", line?);
    }
    Ok(())
}
```

Run

Functions

There are a number of functions that offer access to various features. For example, we can use three of these functions to copy everything from standard input to standard output:

```
use std::io;

fn main() -> io::Result<()> {
    io::copy(&mut io::stdin(), &mut io::stdout())?;
    Ok(())
}
```

Run

io::Result

Last, but certainly not least, is `io::Result`. This type is used as the return type of many `std::io` functions that can cause an error, and can be returned from your own functions as well. Many of the examples in this module use the `?` operator:

```
use std::io;

fn read_input() -> io::Result<()> {
    let mut input = String::new();

    io::stdin().read_line(&mut input)?;

    println!("You typed: {}", input.trim());

    Ok(())
}
```

Run

The return type of `read_input()`, `io::Result<()>`, is a very common type for functions which don't have a 'real' return value, but do want to return errors if they happen. In this case, the only purpose of this function is to read the line and print it, so we use `()`.

Platform-specific behavior

Many I/O functions throughout the standard library are documented to indicate what various library or syscalls they are delegated to. This is done to help applications both understand what's happening under the hood as well as investigate any possibly unclear semantics. Note, however, that this is informative, not a binding contract. The implementation of many of these functions are subject to change over time and may call fewer or more syscalls/library functions.

Modules

prelude	The I/O Prelude

Structs

Initializer	Experimental A type used to conditionally initialize buffers passed to `Read` methods.
BufReader	The `BufReader<R>` struct adds buffering to any reader.
BufWriter	Wraps a writer and buffers its output.
Bytes	An iterator over `u8` values of a reader.
Chain	Adaptor to chain together two readers.
Cursor	A `Cursor` wraps an in-memory buffer and provides it with a `Seek` implementation.
Empty	A reader which is always at EOF.
Error	The error type for I/O operations of the `Read`, `Write`, `Seek`, and associated traits.
IntoInnerError	An error returned by `BufWriter::into_inner` which combines an error that happened while writing out the buffer, and the buffered writer object which may be used to recover from the condition.
IoSlice	A buffer type used with `Write::write_vectored`.
IoSliceMut	A buffer type used with `Read::read_vectored`.
LineWriter	Wraps a writer and buffers output to it, flushing whenever a newline (`0x0a`, `'\n'`) is detected.
Lines	An iterator over the lines of an instance of `BufRead`.
Repeat	A reader which yields one byte over and over and over and over and over and...
Sink	A writer which will move data into the void.
Split	An iterator over the contents of an instance of `BufRead` split on a particular byte.
Stderr	A handle to the standard error stream of a process.
StderrLock	A locked reference to the `Stderr` handle.
Stdin	A handle to the standard input stream of a process.
StdinLock	A locked reference to the `Stdin` handle.
Stdout	A handle to the global standard output stream of the current process.
StdoutLock	A locked reference to the `Stdout` handle.
Take	Reader adaptor which limits the bytes read from an underlying reader.

Enums

ErrorKind	A list specifying general categories of I/O error.
SeekFrom	Enumeration of possible methods to seek within an I/O object.

Traits

BufRead	A `BufRead` is a type of `Reader` which has an internal buffer, allowing it to perform extra ways of reading.
Read	The `Read` trait allows for reading bytes from a source.
Seek	The `Seek` trait provides a cursor which can be moved within a stream of bytes.
Write	A trait for objects which are byte-oriented sinks.

Functions

copy	Copies the entire contents of a reader into a writer.
empty	Constructs a new handle to an empty reader.
repeat	Creates an instance of a reader that infinitely repeats one byte.
sink	Creates an instance of a writer which will successfully consume all data.
stderr	Constructs a new handle to the standard error of the current process.
stdin	Constructs a new handle to the standard input of the current process.
stdout	Constructs a new handle to the standard output of the current process.

Type Definitions

Result	A specialized `Result` type for I/O operations.

Module io
Modules
Structs
Enums
Traits
Functions
Type Definitions
std
Primitive Types
array
bool
char
f32
f64
fn
i128
i16
i32
i64
i8
isize
never
pointer
reference
slice
str
tuple
u128
u16
u32
u64
u8
unit
usize
Modules
alloc
any
array
ascii
backtrace
borrow
boxed
cell
char
clone
cmp
collections
convert
default
env
error
f32
f64
ffi
fmt
fs
future
hash
hint
i128
i16
i32
i64
i8
intrinsics
io
isize
iter
lazy
marker
mem
net
num
ops
option
os
panic
path
pin
prelude
primitive
process
ptr
raw
rc
result
slice
str
string
sync
task
thread
time
u128
u16
u32
u64
u8
usize
vec
Macros
asm
assert
assert_eq
assert_ne
cfg
column
compile_error
concat
concat_idents
dbg
debug_assert
debug_assert_eq
debug_assert_ne
env
eprint
eprintln
file
format
format_args
format_args_nl
global_asm
include
include_bytes
include_str
is_aarch64_feature_dete...
is_arm_feature_detected
is_mips64_feature_detec...
is_mips_feature_detected
is_powerpc64_feature_d...
is_powerpc_feature_det...
is_x86_feature_detected
line
llvm_asm
log_syntax
matches
module_path
option_env
panic
print
println
stringify
thread_local
todo
trace_macros
try
unimplemented
unreachable
vec
write
writeln
Keywords
Self
as
async
await
break
const
continue
crate
dyn
else
enum
extern
false
fn
for
if
impl
in
let
loop
match
mod
move
mut
pub
ref
return
self
static
struct
super
trait
true
type
union
unsafe
use
where
while