



Enum `std::result::Result`

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

`Result` is a type that represents either success (`Ok`) or failure (`Err`).

See the [module documentation](#) for details.

Variants

`Ok(T)`

Tuple Fields

0: T

Contains the success value

`Err(E)`

Tuple Fields

0: E

Contains the error value

Implementations

`impl<T, E> Result<T, E>` [src]

`pub const fn is_ok(&self) -> bool` 1.0.0 (const: 1.48.0) [src]

Returns `true` if the result is `Ok`.

Examples

Basic usage:

```

let x: Result<i32, &str> = Ok(-3);
assert_eq!(x.is_ok(), true);

```

Run

```

let x: Result<i32, &str> = Err("Some error message");
assert_eq!(x.is_ok(), false);

```

```

pub const fn is_err(&self) -> bool

```

1.0.0 (const: 1.48.0) [src]

Returns true if the result is `Err`.

Examples

Basic usage:

```

let x: Result<i32, &str> = Ok(-3);
assert_eq!(x.is_err(), false);

```

Run

```

let x: Result<i32, &str> = Err("Some error message");
assert_eq!(x.is_err(), true);

```

```

pub fn contains<U>(&self, x: &U) -> bool

```

[src]

where

`U: PartialEq<T>`,

 This is a nightly-only experimental API. (option_result_contains [#62358](#))

Returns true if the result is an `Ok` value containing the given value.

Examples

```

#![feature(option_result_contains)]

```

Run

```

let x: Result<u32, &str> = Ok(2);
assert_eq!(x.contains(&2), true);

```

```

let x: Result<u32, &str> = Ok(3);
assert_eq!(x.contains(&2), false);

```

```

let x: Result<u32, &str> = Err("Some error message");
assert_eq!(x.contains(&2), false);

```

```

pub fn contains_err<F>(&self, f: &F) -> bool

```

[src]

where

`F: PartialEq<E>`,

 This is a nightly-only experimental API. (result_contains_err [#62358](#))

Returns true if the result is an `Err` value containing the given value.



Examples

```
#![feature(result_contains_err)]
```

Run

```
let x: Result<u32, &str> = Ok(2);
assert_eq!(x.contains_err(&"Some error message"), false);
```

```
let x: Result<u32, &str> = Err("Some error message");
assert_eq!(x.contains_err(&"Some error message"), true);
```

```
let x: Result<u32, &str> = Err("Some other error message");
assert_eq!(x.contains_err(&"Some error message"), false);
```

```
pub fn ok(self) -> Option<T>
```

[src]

Converts from `Result<T, E>` to `Option<T>`.

Converts `self` into an `Option<T>`, consuming `self`, and discarding the error, if any.

Examples

Basic usage:

```
let x: Result<u32, &str> = Ok(2);
assert_eq!(x.ok(), Some(2));
```

Run

```
let x: Result<u32, &str> = Err("Nothing here");
assert_eq!(x.ok(), None);
```

```
pub fn err(self) -> Option<E>
```

[src]

Converts from `Result<T, E>` to `Option<E>`.

Converts `self` into an `Option<E>`, consuming `self`, and discarding the success value, if any.

Examples

Basic usage:

```
let x: Result<u32, &str> = Ok(2);
assert_eq!(x.err(), None);
```

Run

```
let x: Result<u32, &str> = Err("Nothing here");
assert_eq!(x.err(), Some("Nothing here"));
```

```
pub const fn as_ref(&self) -> Result<&T, &E>
```

1.0.0 (const: 1.48.0) [src]

Converts from `&Result<T, E>` to `Result<&T, &E>`.



Produces a new `Result`, containing a reference into the original, leaving the original in place.

Examples

Basic usage:

```
let x: Result<u32, &str> = Ok(2);
assert_eq!(x.as_ref(), Ok(&2));
```

Run

```
let x: Result<u32, &str> = Err("Error");
assert_eq!(x.as_ref(), Err(&"Error"));
```

```
pub fn as_mut(&mut self) -> Result<&mut T, &mut E> 1.0.0 (const: unstable) [src]
```

Converts from `&mut Result<T, E>` to `Result<&mut T, &mut E>`.

Examples

Basic usage:

```
fn mutate(r: &mut Result<i32, i32>) {
    match r.as_mut() {
        Ok(v) => *v = 42,
        Err(e) => *e = 0,
    }
}
```

Run

```
let mut x: Result<i32, i32> = Ok(2);
mutate(&mut x);
assert_eq!(x.unwrap(), 42);
```

```
let mut x: Result<i32, i32> = Err(13);
mutate(&mut x);
assert_eq!(x.unwrap_err(), 0);
```

```
pub fn map<U, F>(self, op: F) -> Result<U, E>
```

[src]

where


```
F: FnOnce(T) -> U,
```

Maps a `Result<T, E>` to `Result<U, E>` by applying a function to a contained `Ok` value, leaving an `Err` value untouched.

This function can be used to compose the results of two functions.

Examples

Print the numbers on each line of a string multiplied by two.



```
let line = "1\n2\n3\n4\n";

for num in line.lines() {
    match num.parse::<i32>().map(|i| i * 2) {
        Ok(n) => println!("{}", n),
        Err(..) => {}
    }
}
```

Run

```
pub fn map_or<U, F>(self, default: U, f: F) -> U 1.41.0 [src]
```

where

F: **FnOnce**(T) -> U,

Returns the provided default (if **Err**), or applies a function to the contained value (if **Ok**),

Arguments passed to `map_or` are eagerly evaluated; if you are passing the result of a function call, it is recommended to use `map_or_else`, which is lazily evaluated.

Examples

```
let x: Result<_, &str> = Ok("foo");
assert_eq!(x.map_or(42, |v| v.len()), 3);

let x: Result<&str, _> = Err("bar");
assert_eq!(x.map_or(42, |v| v.len()), 42);
```

Run

```
pub fn map_or_else<U, D, F>(self, default: D, f: F) -> U 1.41.0 [src]
```

where

D: **FnOnce**(E) -> U,

F: **FnOnce**(T) -> U,

Maps a `Result<T, E>` to `U` by applying fallback function `default` to a contained **Err** value, or function `f` to a contained **Ok** value.

This function can be used to unpack a successful result while handling an error.

Examples

Basic usage:

```
let k = 21;

let x : Result<_, &str> = Ok("foo");
assert_eq!(x.map_or_else(|e| k * 2, |v| v.len()), 3);

let x : Result<&str, _> = Err("bar");
assert_eq!(x.map_or_else(|e| k * 2, |v| v.len()), 42);
```

Run

```
fn map_err<F, O>(self, op: O) -> Result<T, F> [src]
    e
    O: FnOnce(E) -> F,
```

Maps a `Result<T, E>` to `Result<T, F>` by applying a function to a contained `Err` value, leaving an `Ok` value untouched.

This function can be used to pass through a successful result while handling an error.

Examples

Basic usage:

```
fn stringify(x: u32) -> String { format!("error code: {}", x) } Run

let x: Result<u32, u32> = Ok(2);
assert_eq!(x.map_err(stringify), Ok(2));

let x: Result<u32, u32> = Err(13);
assert_eq!(x.map_err(stringify), Err("error code: 13".to_string()));
```

```
pub fn iter(&self) -> Iter<'_, T> [src]
```



Returns an iterator over the possibly contained value.

The iterator yields one value if the result is `Result::Ok`, otherwise none.

Examples

Basic usage:

```
let x: Result<u32, &str> = Ok(7); Run
assert_eq!(x.iter().next(), Some(&7));

let x: Result<u32, &str> = Err("nothing!");
assert_eq!(x.iter().next(), None);
```

```
pub fn iter_mut(&mut self) -> IterMut<'_, T> [src]
```



Returns a mutable iterator over the possibly contained value.

The iterator yields one value if the result is `Result::Ok`, otherwise none.

Examples

Basic usage:



```
let mut x: Result<u32, &str> = Ok(7);
match x.iter_mut().next() {
    Some(v) => *v = 40,
    None => {},
}
assert_eq!(x, Ok(40));

let mut x: Result<u32, &str> = Err("nothing!");
assert_eq!(x.iter_mut().next(), None);
```

Run

```
pub fn and<U>(self, res: Result<U, E>) -> Result<U, E>
```

[src]

Returns `res` if the result is `Ok`, otherwise returns the `Err` value of `self`.

Examples

Basic usage:

```
let x: Result<u32, &str> = Ok(2);
let y: Result<&str, &str> = Err("late error");
assert_eq!(x.and(y), Err("late error"));

let x: Result<u32, &str> = Err("early error");
let y: Result<&str, &str> = Ok("foo");
assert_eq!(x.and(y), Err("early error"));

let x: Result<u32, &str> = Err("not a 2");
let y: Result<&str, &str> = Err("late error");
assert_eq!(x.and(y), Err("not a 2"));

let x: Result<u32, &str> = Ok(2);
let y: Result<&str, &str> = Ok("different result type");
assert_eq!(x.and(y), Ok("different result type"));
```

Run

```
pub fn and_then<U, F>(self, op: F) -> Result<U, E>
```

[src]

where

```
F: FnOnce(T) -> Result<U, E>,
```

Calls `op` if the result is `Ok`, otherwise returns the `Err` value of `self`.

This function can be used for control flow based on `Result` values.

Examples

Basic usage:

```

fn sq(x: u32) -> Result<u32, u32> { Ok(x * x) }
fn err(x: u32) -> Result<u32, u32> { Err(x) }

```

Run

```

assert_eq!(Ok(2).and_then(sq).and_then(sq), Ok(16));
assert_eq!(Ok(2).and_then(sq).and_then(err), Err(4));
assert_eq!(Ok(2).and_then(err).and_then(sq), Err(2));
assert_eq!(Err(3).and_then(sq).and_then(sq), Err(3));

```

```
pub fn or<F>(self, res: Result<T, F>) -> Result<T, F> [src]
```

Returns `res` if the result is `Err`, otherwise returns the `Ok` value of `self`.

Arguments passed to `or` are eagerly evaluated; if you are passing the result of a function call, it is recommended to use `or_else`, which is lazily evaluated.

Examples

Basic usage:

```

let x: Result<u32, &str> = Ok(2);
let y: Result<u32, &str> = Err("late error");
assert_eq!(x.or(y), Ok(2));

let x: Result<u32, &str> = Err("early error");
let y: Result<u32, &str> = Ok(2);
assert_eq!(x.or(y), Ok(2));

let x: Result<u32, &str> = Err("not a 2");
let y: Result<u32, &str> = Err("late error");
assert_eq!(x.or(y), Err("late error"));

let x: Result<u32, &str> = Ok(2);
let y: Result<u32, &str> = Ok(100);
assert_eq!(x.or(y), Ok(2));

```

Run

```
pub fn or_else<F, O>(self, op: O) -> Result<T, F> [src]
where
    O: FnOnce(E) -> Result<T, F>,
```

Calls `op` if the result is `Err`, otherwise returns the `Ok` value of `self`.

This function can be used for control flow based on result values.

Examples

Basic usage:


```

fn sq(x: u32) -> Result<u32, u32> { Ok(x * x) }
fn err(x: u32) -> Result<u32, u32> { Err(x) }

```

Run

```

assert_eq!(Ok(2).or_else(sq).or_else(sq), Ok(2));
assert_eq!(Ok(2).or_else(err).or_else(sq), Ok(2));
assert_eq!(Err(3).or_else(sq).or_else(err), Ok(9));
assert_eq!(Err(3).or_else(err).or_else(err), Err(3));

```

```
pub fn unwrap_or(self, default: T) -> T
```

[src]

Returns the contained **Ok** value or a provided default.

Arguments passed to `unwrap_or` are eagerly evaluated; if you are passing the result of a function call, it is recommended to use `unwrap_or_else`, which is lazily evaluated.

Examples

Basic usage:

```

let default = 2;
let x: Result<u32, &str> = Ok(9);
assert_eq!(x.unwrap_or(default), 9);

let x: Result<u32, &str> = Err("error");
assert_eq!(x.unwrap_or(default), default);

```

Run

```
pub fn unwrap_or_else<F>(self, op: F) -> T
```

[src]

where

`F: FnOnce(E) -> T,`

Returns the contained **Ok** value or computes it from a closure.

Examples

Basic usage:

```

fn count(x: &str) -> usize { x.len() }

assert_eq!(Ok(2).unwrap_or_else(count), 2);
assert_eq!(Err("foo").unwrap_or_else(count), 3);

```

Run

```
pub unsafe fn unwrap_unchecked(self) -> T
```

1.58.0 [src]

Returns the contained **Ok** value, consuming the `self` value, without checking that the value is not an **Err**.

Safety

Calling this method on an **Err** is *undefined behavior*.



Examples

```
let x: Result<u32, &str> = Ok(2);
assert_eq!(unsafe { x.unwrap_unchecked() }, 2);
```

Run

```
let x: Result<u32, &str> = Err("emergency failure");
unsafe { x.unwrap_unchecked(); } // Undefined behavior!
```

Run

```
pub unsafe fn unwrap_err_unchecked(self) -> E
```

1.58.0 [src]

Returns the contained `Err` value, consuming the `self` value, without checking that the value is not an `Ok`.

Safety

Calling this method on an `Ok` is *undefined behavior*.

Examples

```
let x: Result<u32, &str> = Ok(2);
unsafe { x.unwrap_err_unchecked(); } // Undefined behavior!
```

Run

```
let x: Result<u32, &str> = Err("emergency failure");
assert_eq!(unsafe { x.unwrap_err_unchecked() }, "emergency failure")
```

Run

```
impl<'_, T, E> Result<&'_ T, E>
```

[src]

where

`T`: `Copy`,

```
pub fn copied(self) -> Result<T, E>
```

[src]



This is a nightly-only experimental API. ([result_copied #63168](#))

Maps a `Result<&T, E>` to a `Result<T, E>` by copying the contents of the `Ok` part.

Examples

```
#![feature(result_copied)]
let val = 12;
let x: Result<&i32, i32> = Ok(&val);
assert_eq!(x, Ok(&12));
let copied = x.copied();
assert_eq!(copied, Ok(12));
```

Run

```
impl<'_, T, E> Result<&'_ mut T, E>
```

[src]

☰
T: Copy,

```
pub fn copied(self) -> Result<T, E>
```

[src]

🔍 This is a nightly-only experimental API. ([result_copied #63168](#))

Maps a `Result<&mut T, E>` to a `Result<T, E>` by copying the contents of the `Ok` part.

Examples

```
#![feature(result_copied)]
let mut val = 12;
let x: Result<&mut i32, i32> = Ok(&mut val);
assert_eq!(x, Ok(&mut 12));
let copied = x.copied();
assert_eq!(copied, Ok(12));
```

Run

```
impl<'_, T, E> Result<&'_ T, E>
where
    T: Clone,
```

[src]

```
pub fn cloned(self) -> Result<T, E>
```

[src]

🔍 This is a nightly-only experimental API. ([result_cloned #63168](#))

Maps a `Result<&T, E>` to a `Result<T, E>` by cloning the contents of the `Ok` part.

Examples

```
#![feature(result_cloned)]
let val = 12;
let x: Result<&i32, i32> = Ok(&val);
assert_eq!(x, Ok(&12));
let cloned = x.cloned();
assert_eq!(cloned, Ok(12));
```

Run

```
impl<'_, T, E> Result<&'_ mut T, E>
where
    T: Clone,
```

[src]

```
pub fn cloned(self) -> Result<T, E>
```

[src]

🔍 This is a nightly-only experimental API. ([result_cloned #63168](#))

Maps a `Result<&mut T, E>` to a `Result<T, E>` by cloning the contents of the `Ok` part.

Examples

≡

Run

```
#![feature(result_cloned)]
let mut val = 12;
let x: Result<&mut i32, i32> = Ok(&mut val);
assert_eq!(x, Ok(&mut 12));
let cloned = x.cloned();
assert_eq!(cloned, Ok(12));
```

impl<T, E> Result<T, E>

[src]

where

E: Debug,

pub fn expect(self, msg: &str) -> T

1.4.0 [src]

Returns the contained **Ok** value, consuming the `self` value.

Panics

Panics if the value is an **Err**, with a panic message including the passed message, and the content of the **Err**.

Examples

Basic usage:

i>

```
let x: Result<u32, &str> = Err("emergency failure");
x.expect("Testing expect"); // panics with `Testing expect: emergency failure`
```

Run

pub fn unwrap(self) -> T

[src]

Returns the contained **Ok** value, consuming the `self` value.

Because this function may panic, its use is generally discouraged. Instead, prefer to use pattern matching and handle the **Err** case explicitly, or call `unwrap_or`, `unwrap_or_else`, or `unwrap_or_default`.

Panics

Panics if the value is an **Err**, with a panic message provided by the **Err**'s value.

Examples

Basic usage:

```
let x: Result<u32, &str> = Ok(2);
assert_eq!(x.unwrap(), 2);
```

Run

i>

```
let x: Result<u32, &str> = Err("emergency failure");
x.unwrap(); // panics with `emergency failure`
```

Run

l<T, E> Result<T, E>

e

T: Debug,

[src]

pub fn expect_err(self, msg: &str) -> E

1.17.0 [src]

Returns the contained **Err** value, consuming the `self` value.

Panics

Panics if the value is an **Ok**, with a panic message including the passed message, and the content of the **Ok**.

Examples

Basic usage:

i

```
let x: Result<u32, &str> = Ok(10);
x.expect_err("Testing expect_err"); // panics with `Testing expect_err`
```

Run

pub fn unwrap_err(self) -> E

[src]

Returns the contained **Err** value, consuming the `self` value.

Panics

Panics if the value is an **Ok**, with a custom panic message provided by the **Ok**'s value.

Examples

i

```
let x: Result<u32, &str> = Ok(2);
x.unwrap_err(); // panics with `2`
```

Run

```
let x: Result<u32, &str> = Err("emergency failure");
assert_eq!(x.unwrap_err(), "emergency failure");
```

Run

impl<T, E> Result<T, E>

[src]

where

T: Default,

pub fn unwrap_or_default(self) -> T

1.16.0 [src]

Returns the contained **Ok** value or a default

Consumes the `self` argument then, if **Ok**, returns the contained value, otherwise if **Err**, returns the default value for that type.

Examples

Converts a string to an integer, turning poorly-formed strings into 0 (the default value for integers). **parse** converts a string to any other type that implements **FromStr**, returning an



Err on error.

```

let good_year_from_input = "1909";
let bad_year_from_input = "190blarg";
let good_year = good_year_from_input.parse().unwrap_or_default();
let bad_year = bad_year_from_input.parse().unwrap_or_default();

assert_eq!(1909, good_year);
assert_eq!(0, bad_year);

```

Run

```

impl<T, E> Result<T, E>
where
    E: Into<!>,

```

[src]

```

pub fn into_ok(self) -> T

```

[src]



This is a nightly-only experimental API. (unwrap_infallible [#61695](#))

Returns the contained **Ok** value, but never panics.

Unlike **unwrap**, this method is known to never panic on the result types it is implemented for. Therefore, it can be used instead of **unwrap** as a maintainability safeguard that will fail to compile if the error type of the **Result** is later changed to an error that can actually occur.

Examples

Basic usage:

```

fn only_good_news() -> Result<String, !> {
    Ok("this is fine".into())
}

let s: String = only_good_news().into_ok();
println!("{}", s);

```

Run

```

impl<T, E> Result<T, E>
where
    T: Into<!>,

```

[src]

```

pub fn into_err(self) -> E

```

[src]



This is a nightly-only experimental API. (unwrap_infallible [#61695](#))

Returns the contained **Err** value, but never panics.

Unlike **unwrap_err**, this method is known to never panic on the result types it is implemented for. Therefore, it can be used instead of **unwrap_err** as a maintainability

afeguard that will fail to compile if the `Ok` type of the `Result` is later changed to a type that can actually occur.

Examples

Basic usage:

Run

```
fn only_bad_news() -> Result<!, String> {
    Err("Oops, it failed".into())
}

let error: String = only_bad_news().into_err();
println!("{}", error);
```

```
impl<T, E> Result<T, E> [src]
where
    T: Deref,
```

```
pub fn as_deref(&self) -> Result<&<T as Deref>::Target, &E> 1.47.0 [src]
```

Converts from `Result<T, E>` (or `&Result<T, E>`) to `Result<&<T as Deref>::Target, &E>`.

Coerces the `Ok` variant of the original `Result` via `Deref` and returns the new `Result`.

Examples

Run

```
let x: Result<String, u32> = Ok("hello".to_string());
let y: Result<&str, &u32> = Ok("hello");
assert_eq!(x.as_deref(), y);

let x: Result<String, u32> = Err(42);
let y: Result<&str, &u32> = Err(&42);
assert_eq!(x.as_deref(), y);
```

```
impl<T, E> Result<T, E> [src]
where
    T: DerefMut,
```

```
pub fn as_deref_mut(&mut self) -> Result<&mut <T as DerefMut>::Target, &mut E> 1.47.0 [src]
```

Converts from `Result<T, E>` (or `&mut Result<T, E>`) to `Result<&mut <T as DerefMut>::Target, &mut E>`.

Coerces the `Ok` variant of the original `Result` via `DerefMut` and returns the new `Result`.

Examples



```

let mut s = "HELLO".to_string();
let mut x: Result<String, u32> = Ok("hello".to_string());
let y: Result<&mut str, &mut u32> = Ok(&mut s);
assert_eq!(x.as_deref_mut().map(|x| { x.make_ascii_uppercase(); x })

let mut i = 42;
let mut x: Result<String, u32> = Err(42);
let y: Result<&mut str, &mut u32> = Err(&mut i);
assert_eq!(x.as_deref_mut().map(|x| { x.make_ascii_uppercase(); x })

```

Run

```
impl<T, E> Result<Option<T>, E> [src]
```

```
pub fn transpose(self) -> Option<Result<T, E>> 1.33.0 (const: unstable) [src]
```

Transposes a Result of an Option into an Option of a Result.

Ok(None) will be mapped to None. Ok(Some(_)) and Err(_) will be mapped to Some(Ok(_)) and Some(Err(_)).

Examples

```
#[derive(Debug, Eq, PartialEq)]
struct SomeErr;
```

Run

```

let x: Result<Option<i32>, SomeErr> = Ok(Some(5));
let y: Option<Result<i32, SomeErr>> = Some(Ok(5));
assert_eq!(x.transpose(), y);

```

```
impl<T, E> Result<Result<T, E>, E> [src]
```

```
pub fn flatten(self) -> Result<T, E> [src]
```



This is a nightly-only experimental API. (result_flattening [#70142](#))

Converts from Result<Result<T, E>, E> to Result<T, E>

Examples

Basic usage:


```

#![feature(result_flattening)]
let x: Result<Result<&'static str, u32>, u32> = Ok(Ok("hello"));
assert_eq!(Ok("hello"), x.flatten());

let x: Result<Result<&'static str, u32>, u32> = Ok(Err(6));
assert_eq!(Err(6), x.flatten());

let x: Result<Result<&'static str, u32>, u32> = Err(6);
assert_eq!(Err(6), x.flatten());

```

Run

Flattening only removes one level of nesting at a time:

```


#![feature(result_flattening)]
let x: Result<Result<Result<&'static str, u32>, u32>, u32> = Ok(Ok(Ok("hello")));
assert_eq!(Ok(Ok("hello")), x.flatten());
assert_eq!(Ok("hello"), x.flatten().flatten());

```

Run

```
impl<T> Result<T, T> [src]
```

```
pub const fn into_ok_or_err(self) -> T [src]
```

 This is a nightly-only experimental API. (result_into_ok_or_err [#82223](#))

Returns the **Ok** value if `self` is **Ok**, and the **Err** value if `self` is **Err**.

In other words, this function returns the value (the `T`) of a `Result<T, T>`, regardless of whether or not that result is **Ok** or **Err**.

This can be useful in conjunction with APIs such as `Atomic*::compare_exchange`, or `slice::binary_search`, but only in cases where you don't care if the result was **Ok** or not.

Examples

```

#![feature(result_into_ok_or_err)]
let ok: Result<u32, u32> = Ok(3);
let err: Result<u32, u32> = Err(4);

assert_eq!(ok.into_ok_or_err(), 3);
assert_eq!(err.into_ok_or_err(), 4);

```

Run

Trait Implementations

```
impl<T, E> Clone for Result<T, E> [src]
```

```

    e
    T: Clone,
    E: Clone,

```

```
pub fn clone(&self) -> Result<T, E> [src]
```

Returns a copy of the value. [Read more](#)

```
pub fn clone_from(&mut self, source: &Result<T, E>) [src]
```

Performs copy-assignment from source. [Read more](#)

```
impl<T, E> Debug for Result<T, E> [src]
```

where

```

    T: Debug,
    E: Debug,

```

```
pub fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error> [src]
```

Formats the value using the given formatter. [Read more](#)

```
impl<'_> From<&'_ StreamResult> for Result<MZStatus, MZError> [src]
```

```
pub fn from(res: &StreamResult) -> Result<MZStatus, MZError> [src]
```

Performs the conversion.

```
impl From<StreamResult> for Result<MZStatus, MZError> [src]
```

```
pub fn from(res: StreamResult) -> Result<MZStatus, MZError> [src]
```

Performs the conversion.

```
impl<A, E, V> FromIterator<Result<A, E>> for Result<V, E> [src]
```

where

```
V: FromIterator<A>,
```

```
pub fn from_iter<I>(iter: I) -> Result<V, E> [src]
```

where

```
I: IntoIterator<Item = Result<A, E>>,
```

Takes each element in the `Iterator`: if it is an `Err`, no further elements are taken, and the `Err` is returned. Should no `Err` occur, a container with the values of each `Result` is returned.

Here is an example which increments every integer in a vector, checking for overflow:

```

let v = vec![1, 2];
let res: Result<Vec<u32>, &'static str> = v.iter().map(|x: &u32|
    x.checked_add(1).ok_or("Overflow!"))
    .collect();
assert_eq!(res, Ok(vec![2, 3]));

```

Run



Here is another example that tries to subtract one from another list of integers, this time checking for underflow:

```
let v = vec![1, 2, 0];
let res: Result<Vec<u32>, &'static str> = v.iter().map(|x: &u32|
    x.checked_sub(1).ok_or("Underflow!")
).collect();
assert_eq!(res, Err("Underflow!"));
```

Run

Here is a variation on the previous example, showing that no further elements are taken from iter after the first Err.

```
let v = vec![3, 2, 1, 10];
let mut shared = 0;
let res: Result<Vec<u32>, &'static str> = v.iter().map(|x: &u32| {
    shared += x;
    x.checked_sub(2).ok_or("Underflow!")
}).collect();
assert_eq!(res, Err("Underflow!"));
assert_eq!(shared, 6);
```

Run

Since the third element caused an underflow, no further elements were taken, so the final value of shared is 6 (= 3 + 2 + 1), not 16.

```
impl<T, E, F> FromResidual<Result<Infallible, E>> for
Poll<Option<Result<T, F>>>
where
    F: From<E>,
```

[src]

```
pub fn from_residual(x: Result<Infallible, E>) ->
Poll<Option<Result<T, F>>>
```

[src]

This is a nightly-only experimental API. (try_trait_v2 [#84277](#))

Constructs the type from a compatible Residual type. [Read more](#)

```
impl<T, E, F> FromResidual<Result<Infallible, E>> for
Result<T, F>
where
    F: From<E>,
```

[src]

```
pub fn from_residual(residual: Result<Infallible, E>) -> Result<T,
F>
```

[src]

This is a nightly-only experimental API. (try_trait_v2 [#84277](#))

Constructs the type from a compatible Residual type. [Read more](#)

```
impl<T, E, F> FromResidual<Result<Infallible, E>> for
    Result<T, F>>
```

[\[src\]](#)

where

F: From<E>,

```
pub fn from_residual(x: Result<Infallible, E>) -> Poll<Result<T,
    F>>
```

[\[src\]](#)

This is a nightly-only experimental API. ([try_trait_v2 #84277](#))

Constructs the type from a compatible Residual type. [Read more](#)

```
impl<T, E> Hash for Result<T, E>
```

[\[src\]](#)

where

T: Hash,

E: Hash,

```
pub fn hash<__H>(&self, state: &mut __H)
```

[\[src\]](#)

where

__H: Hasher,

Feeds this value into the given Hasher. [Read more](#)

```
fn hash_slice<H>(data: &[Self], state: &mut H)
```

1.3.0 [\[src\]](#)

where

H: Hasher,

Feeds a slice of this type into the given Hasher. [Read more](#)

```
impl<T, E> IntoIterator for Result<T, E>
```

[\[src\]](#)

```
pub fn into_iter(self) -> IntoIter<T>
```

[\[src\]](#)

Returns a consuming iterator over the possibly contained value.

The iterator yields one value if the result is `Result::Ok`, otherwise none.

Examples

Basic usage:

```
let x: Result<u32, &str> = Ok(5);
let v: Vec<u32> = x.into_iter().collect();
assert_eq!(v, [5]);
```

Run

```
let x: Result<u32, &str> = Err("nothing!");
let v: Vec<u32> = x.into_iter().collect();
assert_eq!(v, []);
```

```
type Item = T
```

The type of the elements being iterated over.

```
impl IntoIter = IntoIter<T>
```

Which kind of iterator are we turning this into?

```
impl<'a, T, E> IntoIterator for &'a Result<T, E> 1.4.0 [src]
```

```
type Item = &'a T
```

The type of the elements being iterated over.

```
type IntoIter = Iter<'a, T>
```

Which kind of iterator are we turning this into?

```
pub fn into_iter(self) -> Iter<'a, T> [src]
```

 Creates an iterator from a value. [Read more](#)

```
impl<'a, T, E> IntoIterator for &'a mut Result<T, E> 1.4.0 [src]
```

```
type Item = &'a mut T
```

The type of the elements being iterated over.

```
type IntoIter = IterMut<'a, T>
```

Which kind of iterator are we turning this into?

```
pub fn into_iter(self) -> IterMut<'a, T> [src]
```

 Creates an iterator from a value. [Read more](#)

```
impl<T, E> Ord for Result<T, E> [src]
```

where

```
T: Ord,
```

```
E: Ord,
```

```
pub fn cmp(&self, other: &Result<T, E>) -> Ordering [src]
```

This method returns an [Ordering](#) between self and other. [Read more](#)

```
fn max(self, other: Self) -> Self 1.21.0 [src]
```

Compares and returns the maximum of two values. [Read more](#)

```
fn min(self, other: Self) -> Self 1.21.0 [src]
```

Compares and returns the minimum of two values. [Read more](#)

```
fn clamp(self, min: Self, max: Self) -> Self 1.50.0 [src]
```

Restrict a value to a certain interval. [Read more](#)

```
impl<T, E> PartialEq<Result<T, E>> for Result<T, E> [src]
```

where

```
T: PartialEq<T>,
```

```
E: PartialEq<E>,
```

```
fn eq(&self, other: &Result<T, E>) -> bool [src]
```

This method tests for self and other values to be equal, and is used by ==. [Read more](#)

```
pub fn ne(&self, other: &Result<T, E>) -> bool [src]
```

This method tests for !=.

```
impl<T, E> PartialOrd<Result<T, E>> for Result<T, E> [src]
```

where

```
T: PartialOrd<T>,
E: PartialOrd<E>,
```

```
pub fn partial_cmp(&self, other: &Result<T, E>) -> Option<Ordering> [src]
```

This method returns an ordering between self and other values if one exists. [Read more](#)

```
fn lt(&self, other: &Rhs) -> bool [src]
```

This method tests less than (for self and other) and is used by the < operator. [Read more](#)

```
fn le(&self, other: &Rhs) -> bool [src]
```

This method tests less than or equal to (for self and other) and is used by the <= operator. [Read more](#)

```
fn gt(&self, other: &Rhs) -> bool [src]
```

This method tests greater than (for self and other) and is used by the > operator. [Read more](#)

```
fn ge(&self, other: &Rhs) -> bool [src]
```

This method tests greater than or equal to (for self and other) and is used by the >= operator. [Read more](#)

```
impl<T, U, E> Product<Result<U, E>> for Result<T, E> 1.16.0 [src]
```

where

```
T: Product<U>,
```

```
pub fn product<I>(iter: I) -> Result<T, E> [src]
```

where

```
I: Iterator<Item = Result<U, E>>,
```

Takes each element in the `Iterator`: if it is an `Err`, no further elements are taken, and the `Err` is returned. Should no `Err` occur, the product of all elements is returned.

```
impl<T, U, E> Sum<Result<U, E>> for Result<T, E> 1.16.0 [src]
```


where

```
T: Sum<U>,
```

```
pub fn sum<I>(iter: I) -> Result<T, E> [src]
```

where

```
I: Iterator<Item = Result<U, E>>,
```


 Takes each element in the `Iterator`: if it is an `Err`, no further elements are taken, and the `Err` is returned. Should no `Err` occur, the sum of all elements is returned.

Examples

This sums up every integer in a vector, rejecting the sum if a negative element is encountered:

```
let v = vec![1, 2];
let res: Result<i32, &'static str> = v.iter().map(|&x: &i32|
    if x < 0 { Err("Negative element found") }
    else { Ok(x) }
).sum();
assert_eq!(res, Ok(3));
```

Run

 `impl<E: Debug> Termination for Result<(), E>` [src]

 `fn report(self) -> i32` [src]

 This is a nightly-only experimental API. (termination_trait_lib #43301)

Is called to get the representation of the value as status code. This status code is returned to the operating system. [Read more](#)

 `impl<E: Debug> Termination for Result<!, E>` [src]

 `fn report(self) -> i32` [src]

 This is a nightly-only experimental API. (termination_trait_lib #43301)

Is called to get the representation of the value as status code. This status code is returned to the operating system. [Read more](#)

 `impl<E: Debug> Termination for Result<Infallible, E>` [src]

 `fn report(self) -> i32` [src]

 This is a nightly-only experimental API. (termination_trait_lib #43301)

Is called to get the representation of the value as status code. This status code is returned to the operating system. [Read more](#)

 `impl<T, E> Try for Result<T, E>` [src]

 `type Output = T`

 This is a nightly-only experimental API. (try_trait_v2 #84277)

The type of the value produced by `?` when *not* short-circuiting.

 `type Residual = Result<Infallible, E>`



This is a nightly-only experimental API. (try_trait_v2 [#84277](#))

The type of the value passed to `FromResidual::from_residual` as part of `?` when short-circuiting. [Read more](#)

```
pub fn from_output(output: <Result<T, E> as Try>::Output) -> Result<T, E> [src]
```



This is a nightly-only experimental API. (try_trait_v2 [#84277](#))

Constructs the type from its `Output` type. [Read more](#)

```
pub fn branch(
    self
) -> ControlFlow<<Result<T, E> as Try>::Residual, <Result<T, E> as Try>::Output> [src]
```



This is a nightly-only experimental API. (try_trait_v2 [#84277](#))

Used in `?` to decide whether the operator should produce a value (because this returned `ControlFlow::Continue`) or propagate a value back to the caller (because this returned `ControlFlow::Break`). [Read more](#)

```
impl<T, E> Copy for Result<T, E> [src]
```

where

`T`: `Copy`,
`E`: `Copy`,

```
impl<T, E> Eq for Result<T, E> [src]
```

where

`T`: `Eq`,
`E`: `Eq`,

```
impl<T, E> StructuralEq for Result<T, E> [src]
```

```
impl<T, E> StructuralPartialEq for Result<T, E> [src]
```

Auto Trait Implementations

```
impl<T, E> RefUnwindSafe for Result<T, E>
```

where

`E`: `RefUnwindSafe`,
`T`: `RefUnwindSafe`,

```
impl<T, E> Send for Result<T, E>
```

where

`E`: `Send`,
`T`: `Send`,

```
impl<T, E> Sync for Result<T, E>
```



```

    E: Sync,
    T: Sync,

```

```
impl<T, E> Unpin for Result<T, E>
```

```
where
```

```

    E: Unpin,
    T: Unpin,

```

```
impl<T, E> UnwindSafe for Result<T, E>
```

```
where
```

```

    E: UnwindSafe,
    T: UnwindSafe,

```

Blanket Implementations

■ `impl<T> Any for T` [\[src\]](#)

```
where
```

```
T: 'static + ?Sized,
```

■ `pub fn type_id(&self) -> TypeId` [\[src\]](#)

Gets the TypeId of self. [Read more](#)

■ `impl<T> Borrow<T> for T` [\[src\]](#)

```
where
```

```
T: ?Sized,
```

■ `pub fn borrow(&self) -> &T` [\[src\]](#)

Immutably borrows from an owned value. [Read more](#)

■ `impl<T> BorrowMut<T> for T` [\[src\]](#)

```
where
```

```
T: ?Sized,
```

■ `pub fn borrow_mut(&mut self) -> &mut T` [\[src\]](#)

Mutably borrows from an owned value. [Read more](#)

■ `impl<T> From<T> for T` [\[src\]](#)

■ `pub fn from(t: T) -> T` [\[src\]](#)

Performs the conversion.

■ `impl<T, U> Into<U> for T` [\[src\]](#)

```
where
```

```
U: From<T>,
```

■ `pub fn into(self) -> U` [\[src\]](#)

Performs the conversion.



`impl<T> ToOwned for T`

[src]

where

`T: Clone,`

`type Owned = T`

The resulting type after obtaining ownership.

`pub fn to_owned(&self) -> T`

[src]

Creates owned data from borrowed data, usually by cloning. [Read more](#)

`pub fn clone_into(&self, target: &mut T)`

[src]



This is a nightly-only experimental API. (toowned_clone_into [#41263](#))

Uses borrowed data to replace owned data, usually by cloning. [Read more](#)

`impl<T, U> TryFrom<U> for T`

[src]

where

`U: Into<T>,`

`type Error = Infallible`

The type returned in the event of a conversion error.

`pub fn try_from(value: U) -> Result<T, <T as TryFrom<U>>::Error>`

[src]

Performs the conversion.

`impl<T, U> TryInto<U> for T`

[src]

where

`U: TryFrom<T>,`

`type Error = <U as TryFrom<T>>::Error`

The type returned in the event of a conversion error.

`pub fn try_into(self) -> Result<U, <U as TryFrom<T>>::Error>`

[src]

Performs the conversion.