Struct String

Methods

Struct std::string::String

A UTF-8 encoded, growable string.

The `String` type is the most common string type that has ownership over the contents of the string. It has a close relationship with its borrowed counterpart, the primitive `str`.

Examples

You can create a `String` from a literal string with `String::from`:

```
let hello = String::from("Hello, world!");
```
Run

You can append a `char` to a `String` with the `push` method, and append a `&str` with the `push_str` method:

```
let mut hello = String::from("Hello, ");

hello.push('w');
hello.push_str("orld!");
```
Run

If you have a vector of UTF-8 bytes, you can create a `String` from it with the `from_utf8` method:

```
// some bytes, in a vector
let sparkle_heart = vec![240, 159, 146, 150];

// We know these bytes are valid, so we'll use `unwrap()`.
let sparkle_heart = String::from_utf8(sparkle_heart).unwrap();

assert_eq!("💖", sparkle_heart);
```
Run

UTF-8

`String`s are always valid UTF-8. This has a few implications, the first of which is that if you need a non-UTF-8 string, consider `OsString`. It is similar, but without the UTF-8 constraint. The second implication is that you cannot index into a `String`:

```
let s = "hello";

println!("The first letter of s is {}", s[0]); // ERROR!!!
```
Run

Indexing is intended to be a constant-time operation, but UTF-8 encoding does not allow us to do this. Furthermore, it's not clear what sort of thing the index should return: a byte, a codepoint, or a grapheme cluster. The `bytes` and `chars` methods return iterators over the first two, respectively.

Deref

`String`s implement `Deref<Target=str>`, and so inherit all of `str`'s methods. In addition, this means that you can pass a `String` to a function which takes a `&str` by using an ampersand (`&`):

```
fn takes_str(s: &str) { }

let s = String::from("Hello");

takes_str(&s);
```
Run

This will create a `&str` from the `String` and pass it in. This conversion is very inexpensive, and so generally, functions will accept `&str`s as arguments unless they need a `String` for some specific reason.

In certain cases Rust doesn't have enough information to make this conversion, known as `Deref` coercion. In the following example a string slice `&'a str` implements the trait `TraitExample`, and the function `example_func` takes anything that implements the trait. In this case Rust would need to make two implicit conversions, which Rust doesn't have the means to do. For that reason, the following example will not compile.


```
trait TraitExample {}

impl<'a> TraitExample for &'a str {}

fn example_func<A: TraitExample>(example_arg: A) {}

fn main() {
    let example_string = String::from("example_string");
    example_func(&example_string);
}
```

Run

There are two options that would work instead. The first would be to change the line `example_func(&example_string);` to `example_func(example_string.as_str());`, using the method `as_str()` to explicitly extract the string slice containing the string. The second way changes `example_func(&example_string);` to `example_func(&*example_string);`. In this case we are dereferencing a `String` to a `str`, then referencing the `str` back to `&str`. The second way is more idiomatic, however both work to do the conversion explicitly rather than relying on the implicit conversion.

Representation

A `String` is made up of three components: a pointer to some bytes, a length, and a capacity. The pointer points to an internal buffer `String` uses to store its data. The length is the number of bytes currently stored in the buffer, and the capacity is the size of the buffer in bytes. As such, the length will always be less than or equal to the capacity.

This buffer is always stored on the heap.

You can look at these with the `as_ptr`, `len`, and `capacity` methods:

```
use std::mem;

let story = String::from("Once upon a time...");

let ptr = story.as_ptr();
let len = story.len();
let capacity = story.capacity();

// story has nineteen bytes
assert_eq!(19, len);

// Now that we have our parts, we throw the story away.
mem::forget(story);

// We can re-build a String out of ptr, len, and capacity. This is all
// unsafe because we are responsible for making sure the components are
// valid:
let s = unsafe { String::from_raw_parts(ptr as *mut _, len, capacity) } ;

assert_eq!(String::from("Once upon a time..."), s);
```

Run

If a `String` has enough capacity, adding elements to it will not re-allocate. For example, consider this program:

```
let mut s = String::new();

println!("{}", s.capacity());

for _ in 0..5 {
    s.push_str("hello");
    println!("{}", s.capacity());
}
```

Run

This will output the following:

```
0
5
10
```

Struct String

Methods


```
20
20
40
```

At first, we have no memory allocated at all, but as we append to the string, it increases its capacity appropriately. If we instead use the `with_capacity` method to allocate the correct capacity initially:

```
let mut s = String::with_capacity(25);

println!("{}", s.capacity());

for _ in 0..5 {
    s.push_str("hello");
    println!("{}", s.capacity());
}
```

Run

We end up with a different output:

```
25
25
25
25
25
25
```

Here, there's no need to allocate more memory inside the loop.

Methods

`impl String` [src]

`pub fn new() -> String` [src]

Creates a new empty `String`.

Given that the `String` is empty, this will not allocate any initial buffer. While that means that this initial operation is very inexpensive, it may cause excessive allocation later when you add data. If you have an idea of how much data the `String` will hold, consider the `with_capacity` method to prevent excessive re-allocation.

Examples

Basic usage:

```
let s = String::new();
```

Run

`pub fn with_capacity(capacity: usize) -> String` [src]

Creates a new empty `String` with a particular capacity.

`String`s have an internal buffer to hold their data. The capacity is the length of that buffer, and can be queried with the `capacity` method. This method creates an empty `String`, but one with an initial buffer that can hold `capacity` bytes. This is useful when you may be appending a bunch of data to the `String`, reducing the number of reallocations it needs to do.

If the given capacity is `0`, no allocation will occur, and this method is identical to the `new` method.

Examples

Basic usage:

```
let mut s = String::with_capacity(10);

// The String contains no chars, even though it has capacity for more
assert_eq!(s.len(), 0);

// These are all done without reallocating...
let cap = s.capacity();
```

Run

Struct String

Methods


```
for i in 0..10 {
    s.push('a');
}

assert_eq!(s.capacity(), cap);

// ...but this may make the vector reallocate
s.push('a');
```

```
pub fn from_utf8(vec: Vec<u8>) -> Result<String, FromUtf8Error>
```
[src]

Converts a vector of bytes to a `String`.

A string slice (`&str`) is made of bytes (`u8`), and a vector of bytes (`Vec<u8>`) is made of bytes, so this function converts between the two. Not all byte slices are valid `String`s, however: `String` requires that it is valid UTF-8. `from_utf8()` checks to ensure that the bytes are valid UTF-8, and then does the conversion.

If you are sure that the byte slice is valid UTF-8, and you don't want to incur the overhead of the validity check, there is an unsafe version of this function, `from_utf8_unchecked`, which has the same behavior but skips the check.

This method will take care to not copy the vector, for efficiency's sake.

If you need a `&str` instead of a `String`, consider `str::from_utf8`.

The inverse of this method is `as_bytes`.

Errors

Returns `Err` if the slice is not UTF-8 with a description as to why the provided bytes are not UTF-8. The vector you moved in is also included.

Examples

Basic usage:

Run

```
// some bytes, in a vector
let sparkle_heart = vec![240, 159, 146, 150];

// We know these bytes are valid, so we'll use `unwrap()`.
let sparkle_heart = String::from_utf8(sparkle_heart).unwrap();

assert_eq!("💖", sparkle_heart);
```

Incorrect bytes:

Run

```
// some invalid bytes, in a vector
let sparkle_heart = vec![0, 159, 146, 150];

assert!(String::from_utf8(sparkle_heart).is_err());
```

See the docs for `FromUtf8Error` for more details on what you can do with this error.

```
pub fn from_utf8_lossy(v: &'a [u8]) -> Cow<'a, str>
```
[src]

Converts a slice of bytes to a string, including invalid characters.

Strings are made of bytes (`u8`), and a slice of bytes (`&[u8]`) is made of bytes, so this function converts between the two. Not all byte slices are valid strings, however: strings are required to be valid UTF-8. During this conversion, `from_utf8_lossy()` will replace any invalid UTF-8 sequences with `U+FFFD REPLACEMENT CHARACTER`, which looks like this: `�`

If you are sure that the byte slice is valid UTF-8, and you don't want to incur the overhead of the conversion, there is an unsafe version of this function, `from_utf8_unchecked`, which has the same behavior but skips the checks.

This function returns a `Cow<'a, str>`. If our byte slice is invalid UTF-8, then we need to insert the replacement characters, which will change the size of the string, and hence, require a `String`. But if it's already valid UTF-8, we don't need a new allocation. This return type allows us to handle both cases.

Examples

Basic usage:

Struct String

Methods


```
// some bytes, in a vector
let sparkle_heart = vec![240, 159, 146, 150];

let sparkle_heart = String::from_utf8_lossy(&sparkle_heart);

assert_eq!("💖", sparkle_heart);
```

Run

Incorrect bytes:

```
// some invalid bytes
let input = b"Hello \xF0\x90\x80World";
let output = String::from_utf8_lossy(input);

assert_eq!("Hello 🌀World", output);
```

Run

```
pub fn from_utf16(v: &[u16]) -> Result<String, FromUtf16Error>
```

[src]

Decode a UTF-16 encoded vector `v` into a `String`, returning `Err` if `v` contains any invalid data.

Examples

Basic usage:

```
// 𝄞music
let v = &[0xD834, 0xDD1E, 0x006d, 0x0075,
          0x0073, 0x0069, 0x0063];
assert_eq!(String::from("𝄞music"),
           String::from_utf16(v).unwrap());

// 𝄞mu<invalid>ic
let v = &[0xD834, 0xDD1E, 0x006d, 0x0075,
          0xD800, 0x0069, 0x0063];
assert!(String::from_utf16(v).is_err());
```

Run

```
pub fn from_utf16_lossy(v: &[u16]) -> String
```

[src]

Decode a UTF-16 encoded slice `v` into a `String`, replacing invalid data with the replacement character (U+FFFD).

Unlike `from_utf8_lossy` which returns a `Cow<'a, str>`, `from_utf16_lossy` returns a `String` since the UTF-16 to UTF-8 conversion requires a memory allocation.

Examples

Basic usage:

```
// 𝄞mus<invalid>ic<invalid>
let v = &[0xD834, 0xDD1E, 0x006d, 0x0075,
          0x0073, 0xDD1E, 0x0069, 0x0063,
          0xD834];

assert_eq!(String::from("𝄞mus\u{FFFD}ic\u{FFFD}"),
           String::from_utf16_lossy(v));
```

Run

```
pub unsafe fn from_raw_parts(
    buf: *mut u8,
    length: usize,
    capacity: usize
) -> String
```

[src]

Creates a new `String` from a length, capacity, and pointer.

Safety

This is highly unsafe, due to the number of invariants that aren't checked:

- The memory at `ptr` needs to have been previously allocated by the same allocator the standard library uses.

Struct String

Methods

new
with_capacity
from_utf8
from_utf8_lossy
from_utf16
from_utf16_lossy
from_raw_parts
from_utf8_unchecked
into_bytes
as_str
as_mut_str
push_str
capacity
reserve
reserve_exact
try_reserve
try_reserve_exact
shrink_to_fit
shrink_to
push
as_bytes
truncate
pop
remove
retain
insert
insert_str
as_mut_vec
len
is_empty
split_off

- `length` needs to be less than or equal to `capacity`.
- `capacity` needs to be the correct value.

Violating these may cause problems like corrupting the allocator's internal data structures.

The ownership of `ptr` is effectively transferred to the `String` which may then deallocate, reallocate or change the contents of memory pointed to by the pointer at will. Ensure that nothing else uses the pointer after calling this function.

Examples

Basic usage:

```
use std::mem;

unsafe {
    let s = String::from("hello");
    let ptr = s.as_ptr();
    let len = s.len();
    let capacity = s.capacity();

    mem::forget(s);

    let s = String::from_raw_parts(ptr as *mut _, len, capacity);

    assert_eq!(String::from("hello"), s);
}
```

Run

```
pub unsafe fn from_utf8_unchecked(bytes: Vec<u8>) -> String
```

[src]

Converts a vector of bytes to a `String` without checking that the string contains valid UTF-8.

See the safe version, `from_utf8`, for more details.

Safety

This function is unsafe because it does not check that the bytes passed to it are valid UTF-8. If this constraint is violated, it may cause memory unsafety issues with future users of the `String`, as the rest of the standard library assumes that `String`s are valid UTF-8.

Examples

Basic usage:

```
// some bytes, in a vector
let sparkle_heart = vec![240, 159, 146, 150];

let sparkle_heart = unsafe {
    String::from_utf8_unchecked(sparkle_heart)
};

assert_eq!("💖", sparkle_heart);
```

Run

```
pub fn into_bytes(self) -> Vec<u8>
```

[src]

Converts a `String` into a byte vector.

This consumes the `String`, so we do not need to copy its contents.

Examples

Basic usage:

```
let s = String::from("hello");
let bytes = s.into_bytes();

assert_eq!(&[104, 101, 108, 108, 111][..], &bytes[..]);
```

Run

```
pub fn as_str(&self) -> &str
```

1.7.0 [src]

Extracts a string slice containing the entire string.

Examples

Basic usage:

```
let s = String::from("foo");

assert_eq!("foo", s.as_str());
```
Run

`pub fn as_mut_str(&mut self) -> &mut str` 1.7.0 [src]

Converts a `String` into a mutable string slice.

Examples

Basic usage:

```
let mut s = String::from("foobar");
let s_mut_str = s.as_mut_str();

s_mut_str.make_ascii_uppercase();

assert_eq!("FOOBAR", s_mut_str);
```
Run

`pub fn push_str(&mut self, string: &str)` [src]

Appends a given string slice onto the end of this `String`.

Examples

Basic usage:

```
let mut s = String::from("foo");

s.push_str("bar");

assert_eq!("foobar", s);
```
Run

`pub fn capacity(&self) -> usize` [src]

Returns this `String`'s capacity, in bytes.

Examples

Basic usage:

```
let s = String::with_capacity(10);

assert!(s.capacity() >= 10);
```
Run

`pub fn reserve(&mut self, additional: usize)` [src]

Ensures that this `String`'s capacity is at least `additional` bytes larger than its length.

The capacity may be increased by more than `additional` bytes if it chooses, to prevent frequent reallocations.

If you do not want this "at least" behavior, see the `reserve_exact` method.

Panics

Panics if the new capacity overflows `usize`.

Examples

Basic usage:

```
let mut s = String::new();
```
Run


```
s.reserve(10);

assert!(s.capacity() >= 10);
```

This may not actually increase the capacity:

```
let mut s = String::with_capacity(10);
s.push('a');
s.push('b');

// s now has a length of 2 and a capacity of 10
assert_eq!(2, s.len());
assert_eq!(10, s.capacity());

// Since we already have an extra 8 capacity, calling this...
s.reserve(8);

// ... doesn't actually increase.
assert_eq!(10, s.capacity());
```

Run

```
pub fn reserve_exact(&mut self, additional: usize)
```

[src]

Ensures that this `String`'s capacity is `additional` bytes larger than its length.

Consider using the `reserve` method unless you absolutely know better than the allocator.

Panics

Panics if the new capacity overflows `usize`.

Examples

Basic usage:

```
let mut s = String::new();

s.reserve_exact(10);

assert!(s.capacity() >= 10);
```

Run

This may not actually increase the capacity:

```
let mut s = String::with_capacity(10);
s.push('a');
s.push('b');

// s now has a length of 2 and a capacity of 10
assert_eq!(2, s.len());
assert_eq!(10, s.capacity());

// Since we already have an extra 8 capacity, calling this...
s.reserve_exact(8);

// ... doesn't actually increase.
assert_eq!(10, s.capacity());
```

Run

```
pub fn try_reserve(
    &mut self,
    additional: usize
) -> Result<(), CollectionAllocErr>
```

[src]

▸ This is a nightly-only experimental API. (`try_reserve` #48043)

Tries to reserve capacity for at least `additional` more elements to be inserted in the given `String`. The collection may reserve more space to avoid frequent reallocations. After calling `reserve`, capacity will be greater than or equal to `self.len() + additional`. Does nothing if capacity is already sufficient.

Errors

If the capacity overflows, or the allocator reports a failure, then an error is returned.

Examples

Run

```
#![feature(try_reserve)]
use std::collections::CollectionAllocErr;

fn process_data(data: &str) -> Result<String, CollectionAllocErr> {
    let mut output = String::new();

    // Pre-reserve the memory, exiting if we can't
    output.try_reserve(data.len())?;

    // Now we know this can't OOM in the middle of our complex work
    output.push_str(data);

    Ok(output)
}
```

[src]

```
pub fn try_reserve_exact(
    &mut self,
    additional: usize
) -> Result<(), CollectionAllocErr>
```

▸ This is a nightly-only experimental API. (`try_reserve` #48043)

Tries to reserves the minimum capacity for exactly `additional` more elements to be inserted in the given `String`. After calling `reserve_exact`, capacity will be greater than or equal to `self.len() + additional`. Does nothing if the capacity is already sufficient.

Note that the allocator may give the collection more space than it requests. Therefore capacity can not be relied upon to be precisely minimal. Prefer `reserve` if future insertions are expected.

Errors

If the capacity overflows, or the allocator reports a failure, then an error is returned.

Examples

Run

```
#![feature(try_reserve)]
use std::collections::CollectionAllocErr;

fn process_data(data: &str) -> Result<String, CollectionAllocErr> {
    let mut output = String::new();

    // Pre-reserve the memory, exiting if we can't
    output.try_reserve(data.len())?;

    // Now we know this can't OOM in the middle of our complex work
    output.push_str(data);

    Ok(output)
}
```

[src]

```
pub fn shrink_to_fit(&mut self)
```

Shrinks the capacity of this `String` to match its length.

Examples

Basic usage:

Run

```
let mut s = String::from("foo");

s.reserve(100);
```

Struct String

Methods


```
assert!(s.capacity() >= 100);

s.shrink_to_fit();
assert_eq!(3, s.capacity());
```

```
pub fn shrink_to(&mut self, min_capacity: usize)
```
[src]

▸ This is a nightly-only experimental API. (`shrink_to`)

Shrinks the capacity of this `String` with a lower bound.

The capacity will remain at least as large as both the length and the supplied value.

Panics if the current capacity is smaller than the supplied minimum capacity.

Examples

Run

```
#![feature(shrink_to)]
let mut s = String::from("foo");

s.reserve(100);
assert!(s.capacity() >= 100);

s.shrink_to(10);
assert!(s.capacity() >= 10);
s.shrink_to(0);
assert!(s.capacity() >= 3);
```

```
pub fn push(&mut self, ch: char)
```
[src]

Appends the given `char` to the end of this `String`.

Examples

Basic usage:

Run

```
let mut s = String::from("abc");

s.push('1');
s.push('2');
s.push('3');

assert_eq!("abc123", s);
```

ⓘ
```
pub fn as_bytes(&self) -> &[u8]
```
[src]

Returns a byte slice of this `String`'s contents.

The inverse of this method is `from_utf8`.

Examples

Basic usage:

Run

```
let s = String::from("hello");

assert_eq!(&[104, 101, 108, 108, 111], s.as_bytes());
```

```
pub fn truncate(&mut self, new_len: usize)
```
[src]

Shortens this `String` to the specified length.

If `new_len` is greater than the string's current length, this has no effect.

Note that this method has no effect on the allocated capacity of the string

Panics

Panics if `new_len` does not lie on a `char` boundary.

Examples

Basic usage:

```
let mut s = String::from("hello");

s.truncate(2);

assert_eq!("he", s);
```
Run

```
pub fn pop(&mut self) -> Option<char>
```
[src]

Removes the last character from the string buffer and returns it.

Returns `None` if this `String` is empty.

Examples

Basic usage:

```
let mut s = String::from("foo");

assert_eq!(s.pop(), Some('o'));
assert_eq!(s.pop(), Some('o'));
assert_eq!(s.pop(), Some('f'));

assert_eq!(s.pop(), None);
```
Run

```
pub fn remove(&mut self, idx: usize) -> char
```
[src]

Removes a `char` from this `String` at a byte position and returns it.

This is an `O(n)` operation, as it requires copying every element in the buffer.

Panics

Panics if `idx` is larger than or equal to the `String`'s length, or if it does not lie on a `char` boundary.

Examples

Basic usage:

```
let mut s = String::from("foo");

assert_eq!(s.remove(0), 'f');
assert_eq!(s.remove(1), 'o');
assert_eq!(s.remove(0), 'o');
```
Run

```
pub fn retain<F>(&mut self, f: F)
where
    F: FnMut(char) -> bool,
```
1.26.0 [src]

Retains only the characters specified by the predicate.

In other words, remove all characters `c` such that `f(c)` returns `false`. This method operates in place and preserves the order of the retained characters.

Examples

```
let mut s = String::from("f_o_ob_ar");

s.retain(|c| c != '_');

assert_eq!(s, "foobar");
```
Run

```
pub fn insert(&mut self, idx: usize, ch: char)
```
[src]

Inserts a character into this `String` at a byte position.

This is an `O(n)` operation as it requires copying every element in the buffer.

Struct String

Methods

Panics

Panics if `idx` is larger than the `String`'s length, or if it does not lie on a `char` boundary.

Examples

Basic usage:

```
let mut s = String::with_capacity(3);

s.insert(0, 'f');
s.insert(1, 'o');
s.insert(2, 'o');

assert_eq!("foo", s);
```

Run

`pub fn insert_str(&mut self, idx: usize, string: &str)` 1.16.0 [src]

Inserts a string slice into this `String` at a byte position.

This is an `O(n)` operation as it requires copying every element in the buffer.

Panics

Panics if `idx` is larger than the `String`'s length, or if it does not lie on a `char` boundary.

Examples

Basic usage:

```
let mut s = String::from("bar");

s.insert_str(0, "foo");

assert_eq!("foobar", s);
```

Run

`pub unsafe fn as_mut_vec(&mut self) -> &mut Vec<u8>` [src]

Returns a mutable reference to the contents of this `String`.

Safety

This function is unsafe because it does not check that the bytes passed to it are valid UTF-8. If this constraint is violated, it may cause memory unsafety issues with future users of the `String`, as the rest of the standard library assumes that `String`s are valid UTF-8.

Examples

Basic usage:

```
let mut s = String::from("hello");

unsafe {
    let vec = s.as_mut_vec();
    assert_eq!(&[104, 101, 108, 108, 111][..], &vec[..]);

    vec.reverse();
}
assert_eq!(s, "olleh");
```

Run

`pub fn len(&self) -> usize` [src]

Returns the length of this `String`, in bytes.

Examples

Basic usage:

```
let a = String::from("foo");
```

Run


```
assert_eq!(a.len(), 3);
```

```
pub fn is_empty(&self) -> bool
```
[src]

Returns `true` if this `String` has a length of zero.

Returns `false` otherwise.

Examples

Basic usage:

```
let mut v = String::new();
assert!(v.is_empty());

v.push('a');
assert!(!v.is_empty());
```
Run

```
pub fn split_off(&mut self, at: usize) -> String
```
1.16.0 [src]

Splits the string into two at the given index.

Returns a newly allocated `String`. `self` contains bytes `[0, at)`, and the returned `String` contains bytes `[at, len)`. `at` must be on the boundary of a UTF-8 code point.

Note that the capacity of `self` does not change.

Panics

Panics if `at` is not on a `UTF-8` code point boundary, or if it is beyond the last code point of the string.

Examples

```
let mut hello = String::from("Hello, World!");
let world = hello.split_off(7);
assert_eq!(hello, "Hello, ");
assert_eq!(world, "World!");
```
Run

```
pub fn clear(&mut self)
```
[src]

Truncates this `String`, removing all contents.

While this means the `String` will have a length of zero, it does not touch its capacity.

Examples

Basic usage:

```
let mut s = String::from("foo");

s.clear();

assert!(s.is_empty());
assert_eq!(0, s.len());
assert_eq!(3, s.capacity());
```
Run

```
pub fn drain<R>(&mut self, range: R) -> Drain
where
    R: RangeBounds<usize>,
```
1.6.0 [src]

Creates a draining iterator that removes the specified range in the string and yields the removed chars.

Note: The element range is removed even if the iterator is not consumed until the end.

Panics

Panics if the starting point or end point do not lie on a `char` boundary, or if they're out of bounds.

Examples

Basic usage:

```
let mut s = String::from("α is alpha, β is beta");
let beta_offset = s.find('β').unwrap_or(s.len());

// Remove the range up until the β from the string
let t: String = s.drain(..beta_offset).collect();
assert_eq!(t, "α is alpha, ");
assert_eq!(s, "β is beta");

// A full range clears the string
s.drain(..);
assert_eq!(s, "");
```

Run

```
pub fn splice<R>(&mut self, range: R, replace_with: &str)
where
    R: RangeBounds<usize>,
```

[src]

▸ This is a nightly-only experimental API. (`splice` #44643)

Creates a splicing iterator that removes the specified range in the string, and replaces it with the given string. The given string doesn't need to be the same length as the range.

Note: Unlike `Vec::splice`, the replacement happens eagerly, and this method does not return the removed chars.

Panics

Panics if the starting point or end point do not lie on a `char` boundary, or if they're out of bounds.

Examples

Basic usage:

```
#![feature(splice)]
let mut s = String::from("α is alpha, β is beta");
let beta_offset = s.find('β').unwrap_or(s.len());

// Replace the range up until the β from the string
s.splice(..beta_offset, "A is capital alpha; ");
assert_eq!(s, "A is capital alpha; β is beta");
```

Run

```
pub fn into_boxed_str(self) -> Box<str>
```

1.4.0 [src]

Converts this `String` into a `Box<str>`.

This will drop any excess capacity.

Examples

Basic usage:

```
let s = String::from("hello");

let b = s.into_boxed_str();
```

Run

Methods from Deref<Target = str>

```
pub fn len(&self) -> usize
```

[src]

Returns the length of `self`.

This length is in bytes, not `char`s or graphemes. In other words, it may not be what a human considers the length of the string.

Examples

Basic usage:

Struct String

Methods

- new
- with_capacity
- from_utf8
- from_utf8_lossy
- from_utf16
- from_utf16_lossy
- from_raw_parts
- from_utf8_unchecked
- into_bytes
- as_str
- as_mut_str
- push_str
- capacity
- reserve
- reserve_exact
- try_reserve
- try_reserve_exact
- shrink_to_fit
- shrink_to
- push
- as_bytes
- truncate
- pop
- remove
- retain
- insert
- insert_str
- as_mut_vec
- len
- is_empty
- split_off

```
let len = "foo".len();
assert_eq!(3, len);

let len = "foo".len(); // fancy f!
assert_eq!(4, len);
```

Run

`pub fn is_empty(&self) -> bool` [src]

Returns `true` if `self` has a length of zero bytes.

Examples

Basic usage:

```
let s = "";
assert!(s.is_empty());

let s = "not empty";
assert!(!s.is_empty());
```

Run

`pub fn is_char_boundary(&self, index: usize) -> bool` 1.9.0 [src]

Checks that `index`-th byte lies at the start and/or end of a UTF-8 code point sequence.

The start and end of the string (when `index == self.len()`) are considered to be boundaries.

Returns `false` if `index` is greater than `self.len()`.

Examples

```
let s = "Löwe 老虎 Léopard";
assert!(s.is_char_boundary(0));
// start of `老`
assert!(s.is_char_boundary(6));
assert!(s.is_char_boundary(s.len()));

// second byte of `ö`
assert!(!s.is_char_boundary(2));

// third byte of `老`
assert!(!s.is_char_boundary(8));
```

Run

ⓘ `pub fn as_bytes(&self) -> &[u8]` [src]

Converts a string slice to a byte slice. To convert the byte slice back into a string slice, use the `str::from_utf8` function.

Examples

Basic usage:

```
let bytes = "bors".as_bytes();
assert_eq!(b"bors", bytes);
```

Run

ⓘ `pub unsafe fn as_bytes_mut(&mut self) -> &mut [u8]` 1.20.0 [src]

Converts a mutable string slice to a mutable byte slice. To convert the mutable byte slice back into a mutable string slice, use the `str::from_utf8_mut` function.

Examples

Basic usage:

```
let mut s = String::from("Hello");
let bytes = unsafe { s.as_bytes_mut() };

assert_eq!(b"Hello", bytes);
```

Run

Struct String

Methods

Mutability:

```
let mut s = String::from("🗻∈🌏");

unsafe {
    let bytes = s.as_bytes_mut();

    bytes[0] = 0xF0;
    bytes[1] = 0x9F;
    bytes[2] = 0x8D;
    bytes[3] = 0x94;
}

assert_eq!("🍔∈🌏", s);
```

Run

```
pub fn as_ptr(&self) -> *const u8
```

[src]

Converts a string slice to a raw pointer.

As string slices are a slice of bytes, the raw pointer points to a `u8`. This pointer will be pointing to the first byte of the string slice.

Examples

Basic usage:

```
let s = "Hello";
let ptr = s.as_ptr();
```

Run

```
pub fn get<I>(&self, i: I) -> Option<&<I as SliceIndex<str>>::Output>
where
    I: SliceIndex<str>,
```

1.20.0 [src]

Returns a subslice of `str`.

This is the non-panicking alternative to indexing the `str`. Returns `None` whenever equivalent indexing operation would panic.

Examples

```
let v = String::from("🗻∈🌏");

assert_eq!(Some("🗻"), v.get(0..4));

// indices not on UTF-8 sequence boundaries
assert!(v.get(1..).is_none());
assert!(v.get(..8).is_none());

// out of bounds
assert!(v.get(..42).is_none());
```

Run

```
pub fn get_mut<I>(
    &mut self,
    i: I
) -> Option<&mut <I as SliceIndex<str>>::Output>
where
    I: SliceIndex<str>,
```

1.20.0 [src]

Returns a mutable subslice of `str`.

This is the non-panicking alternative to indexing the `str`. Returns `None` whenever equivalent indexing operation would panic.

Examples

```
let mut v = String::from("hello");
// correct length
assert!(v.get_mut(0..5).is_some());
// out of bounds
```

Run


```
assert!(v.get_mut(..42).is_none());
assert_eq!(Some("he"), v.get_mut(0..2).map(|v| &*v));

assert_eq!("hello", v);
{
    let s = v.get_mut(0..2);
    let s = s.map(|s| {
        s.make_ascii_uppercase();
        &*s
    });
    assert_eq!(Some("HE"), s);
}
assert_eq!("HEllo", v);
```

```
pub unsafe fn get_unchecked<I>(&self, i: I) -> &<I as SliceIndex<str>>::Output
where
    I: SliceIndex<str>,
```

1.20.0 [src]

Returns a unchecked subslice of `str`.

This is the unchecked alternative to indexing the `str`.

Safety

Callers of this function are responsible that these preconditions are satisfied:

- The starting index must come before the ending index;
- Indexes must be within bounds of the original slice;
- Indexes must lie on UTF-8 sequence boundaries.

Failing that, the returned string slice may reference invalid memory or violate the invariants communicated by the `str` type.

Examples

Run

```
let v = "🗻∈🌏";
unsafe {
    assert_eq!("🗻", v.get_unchecked(0..4));
    assert_eq!("∈", v.get_unchecked(4..7));
    assert_eq!("🌏", v.get_unchecked(7..11));
}
```

```
pub unsafe fn get_unchecked_mut<I>(
    &mut self,
    i: I
) -> &mut <I as SliceIndex<str>>::Output
where
    I: SliceIndex<str>,
```

1.20.0 [src]

Returns a mutable, unchecked subslice of `str`.

This is the unchecked alternative to indexing the `str`.

Safety

Callers of this function are responsible that these preconditions are satisfied:

- The starting index must come before the ending index;
- Indexes must be within bounds of the original slice;
- Indexes must lie on UTF-8 sequence boundaries.

Failing that, the returned string slice may reference invalid memory or violate the invariants communicated by the `str` type.

Examples

Run

```
let mut v = String::from("🗻∈🌏");
unsafe {
    assert_eq!("🗻", v.get_unchecked_mut(0..4));
```

Struct String

Methods


```
    assert_eq!("Ö", v.get_unchecked_mut(4..7));
    assert_eq!("🌍", v.get_unchecked_mut(7..11));
}
```

```
pub unsafe fn slice_unchecked(&self, begin: usize, end: usize) -> &str
```
[src]

Creates a string slice from another string slice, bypassing safety checks.

This is generally not recommended, use with caution! For a safe alternative see `str` and `Index`.

This new slice goes from `begin` to `end`, including `begin` but excluding `end`.

To get a mutable string slice instead, see the `slice_mut_unchecked` method.

Safety

Callers of this function are responsible that three preconditions are satisfied:

- `begin` must come before `end`.
- `begin` and `end` must be byte positions within the string slice.
- `begin` and `end` must lie on UTF-8 sequence boundaries.

Examples

Basic usage:

```
let s = "Löwe 老虎 Léopard";

unsafe {
    assert_eq!("Löwe 老虎 Léopard", s.slice_unchecked(0, 21));
}

let s = "Hello, world!";

unsafe {
    assert_eq!("world", s.slice_unchecked(7, 12));
}
```
Run

```
pub unsafe fn slice_mut_unchecked(
    &mut self,
    begin: usize,
    end: usize
) -> &mut str
```
1.5.0 [src]

Creates a string slice from another string slice, bypassing safety checks. This is generally not recommended, use with caution! For a safe alternative see `str` and `IndexMut`.

This new slice goes from `begin` to `end`, including `begin` but excluding `end`.

To get an immutable string slice instead, see the `slice_unchecked` method.

Safety

Callers of this function are responsible that three preconditions are satisfied:

- `begin` must come before `end`.
- `begin` and `end` must be byte positions within the string slice.
- `begin` and `end` must lie on UTF-8 sequence boundaries.

```
pub fn split_at(&self, mid: usize) -> (&str, &str)
```
1.4.0 [src]

Divide one string slice into two at an index.

The argument, `mid`, should be a byte offset from the start of the string. It must also be on the boundary of a UTF-8 code point.

The two slices returned go from the start of the string slice to `mid`, and from `mid` to the end of the string slice.

To get mutable string slices instead, see the `split_at_mut` method.

Panics

Panics if `mid` is not on a UTF-8 code point boundary, or if it is beyond the last code point of the string slice.

Struct String

Methods

new
with_capacity
from_utf8
from_utf8_lossy
from_utf16
from_utf16_lossy
from_raw_parts
from_utf8_unchecked
into_bytes
as_str
as_mut_str
push_str
capacity
reserve
reserve_exact
try_reserve
try_reserve_exact
shrink_to_fit
shrink_to
push
as_bytes
truncate
pop
remove
retain
insert
insert_str
as_mut_vec
len
is_empty
split_off

Examples

Basic usage:

```
let s = "Per Martin-Löf";

let (first, last) = s.split_at(3);

assert_eq!("Per", first);
assert_eq!(" Martin-Löf", last);
```

Run

```
pub fn split_at_mut(&mut self, mid: usize) -> (&mut str, &mut str)
```

1.4.0 [src]

Divide one mutable string slice into two at an index.

The argument, `mid`, should be a byte offset from the start of the string. It must also be on the boundary of a UTF-8 code point.

The two slices returned go from the start of the string slice to `mid`, and from `mid` to the end of the string slice.

To get immutable string slices instead, see the `split_at` method.

Panics

Panics if `mid` is not on a UTF-8 code point boundary, or if it is beyond the last code point of the string slice.

Examples

Basic usage:

```
let mut s = "Per Martin-Löf".to_string();
{
    let (first, last) = s.split_at_mut(3);
    first.make_ascii_uppercase();
    assert_eq!("PER", first);
    assert_eq!(" Martin-Löf", last);
}
assert_eq!("PER Martin-Löf", s);
```

Run

```
pub fn chars(&self) -> Chars
```

[src]

Returns an iterator over the `char`s of a string slice.

As a string slice consists of valid UTF-8, we can iterate through a string slice by `char`. This method returns such an iterator.

It's important to remember that `char` represents a Unicode Scalar Value, and may not match your idea of what a 'character' is. Iteration over grapheme clusters may be what you actually want.

Examples

Basic usage:

```
let word = "goodbye";

let count = word.chars().count();
assert_eq!(7, count);

let mut chars = word.chars();

assert_eq!(Some('g'), chars.next());
assert_eq!(Some('o'), chars.next());
assert_eq!(Some('o'), chars.next());
assert_eq!(Some('d'), chars.next());
assert_eq!(Some('b'), chars.next());
assert_eq!(Some('y'), chars.next());
assert_eq!(Some('e'), chars.next());

assert_eq!(None, chars.next());
```

Run

Remember, `char`s may not match your human intuition about characters:

Struct String

Methods

new
with_capacity
from_utf8
from_utf8_lossy
from_utf16
from_utf16_lossy
from_raw_parts
from_utf8_unchecked
into_bytes
as_str
as_mut_str
push_str
capacity
reserve
reserve_exact
try_reserve
try_reserve_exact
shrink_to_fit
shrink_to
push
as_bytes
truncate
pop
remove
retain
insert
insert_str
as_mut_vec
len
is_empty
split_off

```
let y = "y̆";

let mut chars = y.chars();

assert_eq!(Some('y'), chars.next()); // not 'y̆'
assert_eq!(Some('\u{0306}'), chars.next());

assert_eq!(None, chars.next());
```

Run

ⓘ `pub fn char_indices(&self) -> CharIndices` [src]

Returns an iterator over the `char`s of a string slice, and their positions.

As a string slice consists of valid UTF-8, we can iterate through a string slice by `char`. This method returns an iterator of both these `char`s, as well as their byte positions.

The iterator yields tuples. The position is first, the `char` is second.

Examples

Basic usage:

```
let word = "goodbye";

let count = word.char_indices().count();
assert_eq!(7, count);

let mut char_indices = word.char_indices();

assert_eq!(Some((0, 'g')), char_indices.next());
assert_eq!(Some((1, 'o')), char_indices.next());
assert_eq!(Some((2, 'o')), char_indices.next());
assert_eq!(Some((3, 'd')), char_indices.next());
assert_eq!(Some((4, 'b')), char_indices.next());
assert_eq!(Some((5, 'y')), char_indices.next());
assert_eq!(Some((6, 'e')), char_indices.next());

assert_eq!(None, char_indices.next());
```

Run

Remember, `char`s may not match your human intuition about characters:

```
let yes = "y̆es";

let mut char_indices = yes.char_indices();

assert_eq!(Some((0, 'y')), char_indices.next()); // not (0, 'y̆')
assert_eq!(Some((1, '\u{0306}')), char_indices.next());

// note the 3 here - the last character took up two bytes
assert_eq!(Some((3, 'e')), char_indices.next());
assert_eq!(Some((4, 's')), char_indices.next());

assert_eq!(None, char_indices.next());
```

Run

ⓘ `pub fn bytes(&self) -> Bytes` [src]

An iterator over the bytes of a string slice.

As a string slice consists of a sequence of bytes, we can iterate through a string slice by byte. This method returns such an iterator.

Examples

Basic usage:

```
let mut bytes = "bors".bytes();

assert_eq!(Some(b'b'), bytes.next());
```

Run

Struct String

Methods

new
with_capacity
from_utf8
from_utf8_lossy
from_utf16
from_utf16_lossy
from_raw_parts
from_utf8_unchecked
into_bytes
as_str
as_mut_str
push_str
capacity
reserve
reserve_exact
try_reserve
try_reserve_exact
shrink_to_fit
shrink_to
push
as_bytes
truncate
pop
remove
retain
insert
insert_str
as_mut_vec
len
is_empty
split_off

```
assert_eq!(Some(b'o'), bytes.next());
assert_eq!(Some(b'r'), bytes.next());
assert_eq!(Some(b's'), bytes.next());

assert_eq!(None, bytes.next());
```

```
pub fn split_whitespace(&self) -> SplitWhitespace
```

1.1.0 [src]

Split a string slice by whitespace.

The iterator returned will return string slices that are sub-slices of the original string slice, separated by any amount of whitespace.

'Whitespace' is defined according to the terms of the Unicode Derived Core Property `White_Space`.

Examples

Basic usage:

```
let mut iter = "A few words".split_whitespace();

assert_eq!(Some("A"), iter.next());
assert_eq!(Some("few"), iter.next());
assert_eq!(Some("words"), iter.next());

assert_eq!(None, iter.next());
```

Run

All kinds of whitespace are considered:

```
let mut iter = " Mary   had\ta\u{2009}little  \n\t lamb".split_whitespace()
assert_eq!(Some("Mary"), iter.next());
assert_eq!(Some("had"), iter.next());
assert_eq!(Some("a"), iter.next());
assert_eq!(Some("little"), iter.next());
assert_eq!(Some("lamb"), iter.next());

assert_eq!(None, iter.next());
```

Run

```
pub fn lines(&self) -> Lines
```

[src]

An iterator over the lines of a string, as string slices.

Lines are ended with either a newline (`\n`) or a carriage return with a line feed (`\r\n`).

The final line ending is optional.

Examples

Basic usage:

```
let text = "foo\r\nbar\n\nbaz\n";
let mut lines = text.lines();

assert_eq!(Some("foo"), lines.next());
assert_eq!(Some("bar"), lines.next());
assert_eq!(Some(""), lines.next());
assert_eq!(Some("baz"), lines.next());

assert_eq!(None, lines.next());
```

Run

The final line ending isn't required:

```
let text = "foo\nbar\n\r\nbaz";
let mut lines = text.lines();

assert_eq!(Some("foo"), lines.next());
assert_eq!(Some("bar"), lines.next());
assert_eq!(Some(""), lines.next());
assert_eq!(Some("baz"), lines.next());
```

Run

Struct String

Methods

- new
- with_capacity
- from_utf8
- from_utf8_lossy
- from_utf16
- from_utf16_lossy
- from_raw_parts
- from_utf8_unchecked
- into_bytes
- as_str
- as_mut_str
- push_str
- capacity
- reserve
- reserve_exact
- try_reserve
- try_reserve_exact
- shrink_to_fit
- shrink_to
- push
- as_bytes
- truncate
- pop
- remove
- retain
- insert
- insert_str
- as_mut_vec
- len
- is_empty
- split_off

```
    assert_eq!(None, lines.next());
```

```
pub fn lines_any(&self) -> LinesAny
```

[src]

Deprecated since 1.4.0: use lines() instead now

An iterator over the lines of a string.

```
pub fn encode_utf16(&self) -> EncodeUtf16
```

1.8.0 [src]

Returns an iterator of `u16` over the string encoded as UTF-16.

Examples

Basic usage:

```
let text = "Zażółć gęślą jaźń";

let utf8_len = text.len();
let utf16_len = text.encode_utf16().count();

assert!(utf16_len <= utf8_len);
```

Run

```
pub fn contains<'a, P>(&'a self, pat: P) -> bool
where
    P: Pattern<'a>,
```

[src]

Returns `true` if the given pattern matches a sub-slice of this string slice.

Returns `false` if it does not.

Examples

Basic usage:

```
let bananas = "bananas";

assert!(bananas.contains("nana"));
assert!(!bananas.contains("apples"));
```

Run

```
pub fn starts_with<'a, P>(&'a self, pat: P) -> bool
where
    P: Pattern<'a>,
```

[src]

Returns `true` if the given pattern matches a prefix of this string slice.

Returns `false` if it does not.

Examples

Basic usage:

```
let bananas = "bananas";

assert!(bananas.starts_with("bana"));
assert!(!bananas.starts_with("nana"));
```

Run

```
pub fn ends_with<'a, P>(&'a self, pat: P) -> bool
where
    P: Pattern<'a>,
    <P as Pattern<'a>>::Searcher: ReverseSearcher<'a>,
```

[src]

Returns `true` if the given pattern matches a suffix of this string slice.

Returns `false` if it does not.

Examples

Basic usage:

```
let bananas = "bananas";
```

Run

Struct String

Methods


```
assert!(bananas.ends_with("anas"));
assert!(!bananas.ends_with("nana"));
```

```
pub fn find<'a, P>(&'a self, pat: P) -> Option<usize>
where
    P: Pattern<'a>,
```
[src]

Returns the byte index of the first character of this string slice that matches the pattern.

Returns `None` if the pattern doesn't match.

The pattern can be a `&str`, `char`, or a closure that determines if a character matches.

Examples

Simple patterns:

```
let s = "Löwe 老虎 Léopard";

assert_eq!(s.find('L'), Some(0));
assert_eq!(s.find('é'), Some(14));
assert_eq!(s.find("Léopard"), Some(13));
```
Run

More complex patterns using point-free style and closures:

```
let s = "Löwe 老虎 Léopard";

assert_eq!(s.find(char::is_whitespace), Some(5));
assert_eq!(s.find(char::is_lowercase), Some(1));
assert_eq!(s.find(|c: char| c.is_whitespace() || c.is_lowercase()), Some(1));
assert_eq!(s.find(|c: char| (c < 'o') && (c > 'a')), Some(4));
```
Run

Not finding the pattern:

```
let s = "Löwe 老虎 Léopard";
let x: &[_] = &['1', '2'];

assert_eq!(s.find(x), None);
```
Run

```
pub fn rfind<'a, P>(&'a self, pat: P) -> Option<usize>
where
    P: Pattern<'a>,
    <P as Pattern<'a>>::Searcher: ReverseSearcher<'a>,
```
[src]

Returns the byte index of the last character of this string slice that matches the pattern.

Returns `None` if the pattern doesn't match.

The pattern can be a `&str`, `char`, or a closure that determines if a character matches.

Examples

Simple patterns:

```
let s = "Löwe 老虎 Léopard";

assert_eq!(s.rfind('L'), Some(13));
assert_eq!(s.rfind('é'), Some(14));
```
Run

More complex patterns with closures:

```
let s = "Löwe 老虎 Léopard";

assert_eq!(s.rfind(char::is_whitespace), Some(12));
assert_eq!(s.rfind(char::is_lowercase), Some(20));
```
Run

Not finding the pattern:


```
let s = "Löwe 老虎 Léopard";
let x: &[_] = &['1', '2'];

assert_eq!(s.rfind(x), None);
```
Run

```
pub fn split<'a, P>(&'a self, pat: P) -> Split<'a, P>
where
    P: Pattern<'a>,
```
[src]

An iterator over substrings of this string slice, separated by characters matched by a pattern.

The pattern can be a `&str`, `char`, or a closure that determines the split.

Iterator behavior

The returned iterator will be a `DoubleEndedIterator` if the pattern allows a reverse search and forward/reverse search yields the same elements. This is true for, eg, `char` but not for `&str`.

If the pattern allows a reverse search but its results might differ from a forward search, the `rsplit` method can be used.

Examples

Simple patterns:

```
let v: Vec<&str> = "Mary had a little lamb".split(' ').collect();
assert_eq!(v, ["Mary", "had", "a", "little", "lamb"]);

let v: Vec<&str> = "".split('X').collect();
assert_eq!(v, [""]);

let v: Vec<&str> = "lionXXtigerXleopard".split('X').collect();
assert_eq!(v, ["lion", "", "tiger", "leopard"]);

let v: Vec<&str> = "lion::tiger::leopard".split("::").collect();
assert_eq!(v, ["lion", "tiger", "leopard"]);

let v: Vec<&str> = "abc1def2ghi".split(char::is_numeric).collect();
assert_eq!(v, ["abc", "def", "ghi"]);

let v: Vec<&str> = "lionXtigerXleopard".split(char::is_uppercase).collect();
assert_eq!(v, ["lion", "tiger", "leopard"]);
```
Run

A more complex pattern, using a closure:

```
let v: Vec<&str> = "abc1defXghi".split(|c| c == '1' || c == 'X').collect();
assert_eq!(v, ["abc", "def", "ghi"]);
```
Run

If a string contains multiple contiguous separators, you will end up with empty strings in the output:

```
let x = "||||a||b|c".to_string();
let d: Vec<_> = x.split('|').collect();

assert_eq!(d, &["", "", "", "", "a", "", "b", "c"]);
```
Run

Contiguous separators are separated by the empty string.

```
let x = "((((()".to_string();
let d: Vec<_> = x.split('/').collect();

assert_eq!(d, &["(", "", "", ")"]);
```
Run

Separators at the start or end of a string are neighbored by empty strings.

```
let d: Vec<_> = "010".split("0").collect();
assert_eq!(d, &["", "1", ""]);
```
Run

Struct String

Methods

When the empty string is used as a separator, it separates every character in the string, along with the beginning and end of the string.

```
let f: Vec<_> = "rust".split("").collect();
assert_eq!(f, &["", "r", "u", "s", "t", ""]);
```
Run

Contiguous separators can lead to possibly surprising behavior when whitespace is used as the separator. This code is correct:

```
let x = "    a  b c".to_string();
let d: Vec<_> = x.split(' ').collect();

assert_eq!(d, &["", "", "", "", "a", "", "b", "c"]);
```
Run

It does *not* give you:

```
assert_eq!(d, &["a", "b", "c"]);
```
Run

Use `split_whitespace` for this behavior.

ⓘ
```
pub fn rsplit<'a, P>(&'a self, pat: P) -> RSplit<'a, P>
where
    P: Pattern<'a>,
    <P as Pattern<'a>>::Searcher: ReverseSearcher<'a>,
```
[src]

An iterator over substrings of the given string slice, separated by characters matched by a pattern and yielded in reverse order.

The pattern can be a `&str`, `char`, or a closure that determines the split.

Iterator behavior

The returned iterator requires that the pattern supports a reverse search, and it will be a `DoubleEndedIterator` if a forward/reverse search yields the same elements.

For iterating from the front, the `split` method can be used.

Examples

Simple patterns:

```
let v: Vec<&str> = "Mary had a little lamb".rsplit(' ').collect();
assert_eq!(v, ["lamb", "little", "a", "had", "Mary"]);

let v: Vec<&str> = "".rsplit('X').collect();
assert_eq!(v, [""]);

let v: Vec<&str> = "lionXXtigerXleopard".rsplit('X').collect();
assert_eq!(v, ["leopard", "tiger", "", "lion"]);

let v: Vec<&str> = "lion::tiger::leopard".rsplit("::").collect();
assert_eq!(v, ["leopard", "tiger", "lion"]);
```
Run

A more complex pattern, using a closure:

```
let v: Vec<&str> = "abc1defXghi".rsplit(|c| c == '1' || c == 'X').collect()
assert_eq!(v, ["ghi", "def", "abc"]);
```
Run

ⓘ
```
pub fn split_terminator<'a, P>(&'a self, pat: P) -> SplitTerminator<'a, P>
where
    P: Pattern<'a>,
```
[src]

An iterator over substrings of the given string slice, separated by characters matched by a pattern.

The pattern can be a `&str`, `char`, or a closure that determines the split.

Equivalent to `split`, except that the trailing substring is skipped if empty.

This method can be used for string data that is *terminated*, rather than *separated* by a pattern.

Iterator behavior

Struct String

Methods

The returned iterator will be a `DoubleEndedIterator` if the pattern allows a reverse search and forward/reverse search yields the same elements. This is true for, eg, `char` but not for `&str`.

If the pattern allows a reverse search but its results might differ from a forward search, the `rsplit_terminator` method can be used.

Examples

Basic usage:

```
let v: Vec<&str> = "A.B.".split_terminator('.').collect();
assert_eq!(v, ["A", "B"]);

let v: Vec<&str> = "A..B..".split_terminator(".").collect();
assert_eq!(v, ["A", "", "B", ""]);
```

Run

ⓘ `pub fn rsplit_terminator<'a, P>(&'a self, pat: P) -> RSplitTerminator<'a, P>` [src]

```
where
    P: Pattern<'a>,
    <P as Pattern<'a>>::Searcher: ReverseSearcher<'a>,
```

An iterator over substrings of `self`, separated by characters matched by a pattern and yielded in reverse order.

The pattern can be a simple `&str`, `char`, or a closure that determines the split. Additional libraries might provide more complex patterns like regular expressions.

Equivalent to `split`, except that the trailing substring is skipped if empty.

This method can be used for string data that is *terminated*, rather than *separated* by a pattern.

Iterator behavior

The returned iterator requires that the pattern supports a reverse search, and it will be double ended if a forward/reverse search yields the same elements.

For iterating from the front, the `split_terminator` method can be used.

Examples

```
let v: Vec<&str> = "A.B.".rsplit_terminator('.').collect();
assert_eq!(v, ["B", "A"]);

let v: Vec<&str> = "A..B..".rsplit_terminator(".").collect();
assert_eq!(v, ["", "B", "", "A"]);
```

Run

ⓘ `pub fn splitn<'a, P>(&'a self, n: usize, pat: P) -> SplitN<'a, P>` [src]

```
where
    P: Pattern<'a>,
```

An iterator over substrings of the given string slice, separated by a pattern, restricted to returning at most `n` items.

If `n` substrings are returned, the last substring (the `n`th substring) will contain the remainder of the string.

The pattern can be a `&str`, `char`, or a closure that determines the split.

Iterator behavior

The returned iterator will not be double ended, because it is not efficient to support.

If the pattern allows a reverse search, the `rsplitn` method can be used.

Examples

Simple patterns:

```
let v: Vec<&str> = "Mary had a little lambda".splitn(3, ' ').collect();
assert_eq!(v, ["Mary", "had", "a little lambda"]);

let v: Vec<&str> = "lionXXtigerXleopard".splitn(3, "X").collect();
assert_eq!(v, ["lion", "", "tigerXleopard"]);

let v: Vec<&str> = "abcXdef".splitn(1, 'X').collect();
```

Run


```
assert_eq!(v, ["abcXdef"]);

let v: Vec<&str> = "".splitn(1, 'X').collect();
assert_eq!(v, [""]);
```

A more complex pattern, using a closure:

```
let v: Vec<&str> = "abc1defXghi".splitn(2, |c| c == '1' || c == 'X').collec Run
assert_eq!(v, ["abc", "defXghi"]);
```

ⓘ `pub fn rsplitn<'a, P>(&'a self, n: usize, pat: P) -> RSplitN<'a, P>` [src]

```
where
    P: Pattern<'a>,
    <P as Pattern<'a>>::Searcher: ReverseSearcher<'a>,
```

An iterator over substrings of this string slice, separated by a pattern, starting from the end of the string, restricted to returning at most `n` items.

If `n` substrings are returned, the last substring (the `n`th substring) will contain the remainder of the string.

The pattern can be a `&str`, `char`, or a closure that determines the split.

Iterator behavior

The returned iterator will not be double ended, because it is not efficient to support.

For splitting from the front, the `splitn` method can be used.

Examples

Simple patterns:

```
let v: Vec<&str> = "Mary had a little lamb".rsplitn(3, ' ').collect();      Run
assert_eq!(v, ["lamb", "little", "Mary had a"]);

let v: Vec<&str> = "lionXXtigerXleopard".rsplitn(3, 'X').collect();
assert_eq!(v, ["leopard", "tiger", "lionX"]);

let v: Vec<&str> = "lion::tiger::leopard".rsplitn(2, "::").collect();
assert_eq!(v, ["leopard", "lion::tiger"]);
```

A more complex pattern, using a closure:

```
let v: Vec<&str> = "abc1defXghi".rsplitn(2, |c| c == '1' || c == 'X').colle Run
assert_eq!(v, ["ghi", "abc1def"]);
```

ⓘ `pub fn matches<'a, P>(&'a self, pat: P) -> Matches<'a, P>` 1.2.0 [src]

```
where
    P: Pattern<'a>,
```

An iterator over the disjoint matches of a pattern within the given string slice.

The pattern can be a `&str`, `char`, or a closure that determines if a character matches.

Iterator behavior

The returned iterator will be a `DoubleEndedIterator` if the pattern allows a reverse search and forward/reverse search yields the same elements. This is true for, eg, `char` but not for `&str`.

If the pattern allows a reverse search but its results might differ from a forward search, the `rmatches` method can be used.

Examples

Basic usage:

```
let v: Vec<&str> = "abcXXXabcYYYabc".matches("abc").collect();              Run
assert_eq!(v, ["abc", "abc", "abc"]);

let v: Vec<&str> = "1abc2abc3".matches(char::is_numeric).collect();
assert_eq!(v, ["1", "2", "3"]);
```

Struct String

Methods


```
pub fn rmatches<'a, P>(&'a self, pat: P) -> RMatches<'a, P>
where
    P: Pattern<'a>,
    <P as Pattern<'a>>::Searcher: ReverseSearcher<'a>,
```

1.2.0 [src]

An iterator over the disjoint matches of a pattern within this string slice, yielded in reverse order.

The pattern can be a `&str`, `char`, or a closure that determines if a character matches.

Iterator behavior

The returned iterator requires that the pattern supports a reverse search, and it will be a `DoubleEndedIterator` if a forward/reverse search yields the same elements.

For iterating from the front, the `matches` method can be used.

Examples

Basic usage:

Run

```
let v: Vec<&str> = "abcXXXabcYYYabc".rmatches("abc").collect();
assert_eq!(v, ["abc", "abc", "abc"]);

let v: Vec<&str> = "1abc2abc3".rmatches(char::is_numeric).collect();
assert_eq!(v, ["3", "2", "1"]);
```

```
pub fn match_indices<'a, P>(&'a self, pat: P) -> MatchIndices<'a, P>
where
    P: Pattern<'a>,
```

1.5.0 [src]

An iterator over the disjoint matches of a pattern within this string slice as well as the index that the match starts at.

For matches of `pat` within `self` that overlap, only the indices corresponding to the first match are returned.

The pattern can be a `&str`, `char`, or a closure that determines if a character matches.

Iterator behavior

The returned iterator will be a `DoubleEndedIterator` if the pattern allows a reverse search and forward/reverse search yields the same elements. This is true for, eg, `char` but not for `&str`.

If the pattern allows a reverse search but its results might differ from a forward search, the `rmatch_indices` method can be used.

Examples

Basic usage:

Run

```
let v: Vec<_> = "abcXXXabcYYYabc".match_indices("abc").collect();
assert_eq!(v, [(0, "abc"), (6, "abc"), (12, "abc")]);

let v: Vec<_> = "1abcabc2".match_indices("abc").collect();
assert_eq!(v, [(1, "abc"), (4, "abc")]);

let v: Vec<_> = "ababa".match_indices("aba").collect();
assert_eq!(v, [(0, "aba")]); // only the first `aba`
```

```
pub fn rmatch_indices<'a, P>(&'a self, pat: P) -> RMatchIndices<'a, P>
where
    P: Pattern<'a>,
    <P as Pattern<'a>>::Searcher: ReverseSearcher<'a>,
```

1.5.0 [src]

An iterator over the disjoint matches of a pattern within `self`, yielded in reverse order along with the index of the match.

For matches of `pat` within `self` that overlap, only the indices corresponding to the last match are returned.

The pattern can be a `&str`, `char`, or a closure that determines if a character matches.

Iterator behavior

The returned iterator requires that the pattern supports a reverse search, and it will be a `DoubleEndedIterator` if a forward/reverse search yields the same elements.

Struct String

Methods

new
with_capacity
from_utf8
from_utf8_lossy
from_utf16
from_utf16_lossy
from_raw_parts
from_utf8_unchecked
into_bytes
as_str
as_mut_str
push_str
capacity
reserve
reserve_exact
try_reserve
try_reserve_exact
shrink_to_fit
shrink_to
push
as_bytes
truncate
pop
remove
retain
insert
insert_str
as_mut_vec
len
is_empty
split_off

For iterating from the front, the `match_indices` method can be used.

Examples

Basic usage:

```
let v: Vec<_> = "abcXXXabcYYYabc".rmatch_indices("abc").collect();
assert_eq!(v, [(12, "abc"), (6, "abc"), (0, "abc")]);

let v: Vec<_> = "1abcabc2".rmatch_indices("abc").collect();
assert_eq!(v, [(4, "abc"), (1, "abc")]);

let v: Vec<_> = "ababa".rmatch_indices("aba").collect();
assert_eq!(v, [(2, "aba")]); // only the last `aba`
```

Run

```
pub fn trim(&self) -> &str
```

[src]

Returns a string slice with leading and trailing whitespace removed.

'Whitespace' is defined according to the terms of the Unicode Derived Core Property `White_Space`.

Examples

Basic usage:

```
let s = " Hello\tworld\t";

assert_eq!("Hello\tworld", s.trim());
```

Run

```
pub fn trim_left(&self) -> &str
```

[src]

Returns a string slice with leading whitespace removed.

'Whitespace' is defined according to the terms of the Unicode Derived Core Property `White_Space`.

Text directionality

A string is a sequence of bytes. 'Left' in this context means the first position of that byte string; for a language like Arabic or Hebrew which are 'right to left' rather than 'left to right', this will be the *right* side, not the left.

Examples

Basic usage:

```
let s = " Hello\tworld\t";

assert_eq!("Hello\tworld\t", s.trim_left());
```

Run

Directionality:

```
let s = "  English";
assert!(Some('E') == s.trim_left().chars().next());

let s = "  עברית";
assert!(Some('ע') == s.trim_left().chars().next());
```

Run

```
pub fn trim_right(&self) -> &str
```

[src]

Returns a string slice with trailing whitespace removed.

'Whitespace' is defined according to the terms of the Unicode Derived Core Property `White_Space`.

Text directionality

A string is a sequence of bytes. 'Right' in this context means the last position of that byte string; for a language like Arabic or Hebrew which are 'right to left' rather than 'left to right', this will be the *left* side, not the right.

Examples

Basic usage:

Struct String

Methods


```
let s = " Hello\tworld\t";

assert_eq!(" Hello\tworld", s.trim_right());
```

Run

Directionality:

```
let s = "English  ";
assert!(Some('h') == s.trim_right().chars().rev().next());

let s = "עברית  ";
assert!(Some('ת') == s.trim_right().chars().rev().next());
```

Run

```
pub fn trim_matches<'a, P>(&'a self, pat: P) -> &'a str
where
    P: Pattern<'a>,
    <P as Pattern<'a>>::Searcher: DoubleEndedSearcher<'a>,
```

[src]

Returns a string slice with all prefixes and suffixes that match a pattern repeatedly removed.

The pattern can be a `char` or a closure that determines if a character matches.

Examples

Simple patterns:

```
assert_eq!("11foo1bar11".trim_matches('1'), "foo1bar");
assert_eq!("123foo1bar123".trim_matches(char::is_numeric), "foo1bar");

let x: &[_] = &['1', '2'];
assert_eq!("12foo1bar12".trim_matches(x), "foo1bar");
```

Run

A more complex pattern, using a closure:

```
assert_eq!("1foo1barXX".trim_matches(|c| c == '1' || c == 'X'), "foo1bar");
```

Run

```
pub fn trim_left_matches<'a, P>(&'a self, pat: P) -> &'a str
where
    P: Pattern<'a>,
```

[src]

Returns a string slice with all prefixes that match a pattern repeatedly removed.

The pattern can be a `&str`, `char`, or a closure that determines if a character matches.

Text directionality

A string is a sequence of bytes. 'Left' in this context means the first position of that byte string; for a language like Arabic or Hebrew which are 'right to left' rather than 'left to right', this will be the *right* side, not the left.

Examples

Basic usage:

```
assert_eq!("11foo1bar11".trim_left_matches('1'), "foo1bar11");
assert_eq!("123foo1bar123".trim_left_matches(char::is_numeric), "foo1bar123");

let x: &[_] = &['1', '2'];
assert_eq!("12foo1bar12".trim_left_matches(x), "foo1bar12");
```

Run

```
pub fn trim_right_matches<'a, P>(&'a self, pat: P) -> &'a str
where
    P: Pattern<'a>,
    <P as Pattern<'a>>::Searcher: ReverseSearcher<'a>,
```

[src]

Returns a string slice with all suffixes that match a pattern repeatedly removed.

The pattern can be a `&str`, `char`, or a closure that determines if a character matches.

Text directionality

A string is a sequence of bytes. 'Right' in this context means the last position of that byte string; for a language

Struct String

Methods

like Arabic or Hebrew which are 'right to left' rather than 'left to right', this will be the *left* side, not the right.

Examples

Simple patterns:

```
assert_eq!("11foo1bar11".trim_right_matches('1'), "11foo1bar");
assert_eq!("123foo1bar123".trim_right_matches(char::is_numeric), "123foo1bar");

let x: &[_] = &['1', '2'];
assert_eq!("12foo1bar12".trim_right_matches(x), "12foo1bar");
```

Run

A more complex pattern, using a closure:

```
assert_eq!("1fooX".trim_right_matches(|c| c == '1' || c == 'X'), "1foo");
```

Run

```
pub fn parse<F>(&self) -> Result<F, <F as FromStr>::Err>
where
    F: FromStr,
```

[src]

Parses this string slice into another type.

Because `parse` is so general, it can cause problems with type inference. As such, `parse` is one of the few times you'll see the syntax affectionately known as the 'turbofish': `::<>`. This helps the inference algorithm understand specifically which type you're trying to parse into.

`parse` can parse any type that implements the `FromStr` trait.

Errors

Will return `Err` if it's not possible to parse this string slice into the desired type.

Examples

Basic usage

```
let four: u32 = "4".parse().unwrap();

assert_eq!(4, four);
```

Run

Using the 'turbofish' instead of annotating `four`:

```
let four = "4".parse::<u32>();

assert_eq!(Ok(4), four);
```

Run

Failing to parse:

```
let nope = "j".parse::<u32>();

assert!(nope.is_err());
```

Run

```
pub fn replace<'a, P>(&'a self, from: P, to: &str) -> String
where
    P: Pattern<'a>,
```

[src]

Replaces all matches of a pattern with another string.

`replace` creates a new `String`, and copies the data from this string slice into it. While doing so, it attempts to find matches of a pattern. If it finds any, it replaces them with the replacement string slice.

Examples

Basic usage:

```
let s = "this is old";

assert_eq!("this is new", s.replace("old", "new"));
```

Run

When the pattern doesn't match:

```
let s = "this is old";
assert_eq!(s, s.replace("cookie monster", "little lamb"));
```
Run

```
pub fn replacen<'a, P>(&'a self, pat: P, to: &str, count: usize) -> String
where
    P: Pattern<'a>,
```
1.16.0 [src]

Replaces first N matches of a pattern with another string.

`replacen` creates a new `String`, and copies the data from this string slice into it. While doing so, it attempts to find matches of a pattern. If it finds any, it replaces them with the replacement string slice at most `count` times.

Examples

Basic usage:

```
let s = "foo foo 123 foo";
assert_eq!("new new 123 foo", s.replacen("foo", "new", 2));
assert_eq!("faa fao 123 foo", s.replacen('o', "a", 3));
assert_eq!("foo foo new23 foo", s.replacen(char::is_numeric, "new", 1));
```
Run

When the pattern doesn't match:

```
let s = "this is old";
assert_eq!(s, s.replacen("cookie monster", "little lamb", 10));
```
Run

```
pub fn to_lowercase(&self) -> String
```
1.2.0 [src]

Returns the lowercase equivalent of this string slice, as a new `String`.

'Lowercase' is defined according to the terms of the Unicode Derived Core Property `Lowercase`.

Since some characters can expand into multiple characters when changing the case, this function returns a `String` instead of modifying the parameter in-place.

Examples

Basic usage:

```
let s = "HELLO";

assert_eq!("hello", s.to_lowercase());
```
Run

A tricky example, with sigma:

```
let sigma = "Σ";

assert_eq!("σ", sigma.to_lowercase());

// but at the end of a word, it's ς, not σ:
let odysseus = "ὈΔΥΣΣΕΥΣ";

assert_eq!("ὀδυσσεύς", odysseus.to_lowercase());
```
Run

Languages without case are not changed:

```
let new_year = "农历新年";

assert_eq!(new_year, new_year.to_lowercase());
```
Run

```
pub fn to_uppercase(&self) -> String
```
1.2.0 [src]

Returns the uppercase equivalent of this string slice, as a new `String`.

'Uppercase' is defined according to the terms of the Unicode Derived Core Property `Uppercase`.

Since some characters can expand into multiple characters when changing the case, this function returns a `String` instead of modifying the parameter in-place.

Examples

Basic usage:

```
let s = "hello";

assert_eq!("HELLO", s.to_uppercase());
```
Run

Scripts without case are not changed:

```
let new_year = "农历新年";

assert_eq!(new_year, new_year.to_uppercase());
```
Run

```
pub fn escape_debug(&self) -> String
```
[src]

This is a nightly-only experimental API. (`str_escape` #27791)

Escapes each char in `s` with `char::escape_debug`.

```
pub fn escape_default(&self) -> String
```
[src]

This is a nightly-only experimental API. (`str_escape` #27791)

Escapes each char in `s` with `char::escape_default`.

```
pub fn escape_unicode(&self) -> String
```
[src]

This is a nightly-only experimental API. (`str_escape` #27791)

Escapes each char in `s` with `char::escape_unicode`.

```
pub fn repeat(&self, n: usize) -> String
```
1.16.0 [src]

Create a `String` by repeating a string `n` times.

Examples

Basic usage:

```
assert_eq!("abc".repeat(4), String::from("abcabcabcabc"));
```
Run

```
pub fn is_ascii(&self) -> bool
```
1.23.0 [src]

Checks if all characters in this string are within the ASCII range.

Examples

```
let ascii = "hello!\n";
let non_ascii = "Grüße, Jürgen ❤";

assert!(ascii.is_ascii());
assert!(!non_ascii.is_ascii());
```
Run

```
pub fn to_ascii_uppercase(&self) -> String
```
1.23.0 [src]

Returns a copy of this string where each character is mapped to its ASCII upper case equivalent.

ASCII letters 'a' to 'z' are mapped to 'A' to 'Z', but non-ASCII letters are unchanged.

To uppercase the value in-place, use `make_ascii_uppercase`.

To uppercase ASCII characters in addition to non-ASCII characters, use `to_uppercase`.

Examples

```
let s = "Grüße, Jürgen ❤";

assert_eq!("GRüßE, JüRGEN ❤", s.to_ascii_uppercase());
```

Run

```
pub fn to_ascii_lowercase(&self) -> String
```

1.23.0 [src]

Returns a copy of this string where each character is mapped to its ASCII lower case equivalent.

ASCII letters 'A' to 'Z' are mapped to 'a' to 'z', but non-ASCII letters are unchanged.

To lowercase the value in-place, use `make_ascii_lowercase`.

To lowercase ASCII characters in addition to non-ASCII characters, use `to_lowercase`.

Examples

```
let s = "Grüße, Jürgen ❤";

assert_eq!("grüße, jürgen ❤", s.to_ascii_lowercase());
```

Run

```
pub fn eq_ignore_ascii_case(&self, other: &str) -> bool
```

1.23.0 [src]

Checks that two strings are an ASCII case-insensitive match.

Same as `to_ascii_lowercase(a) == to_ascii_lowercase(b)`, but without allocating and copying temporaries.

Examples

```
assert!("Ferris".eq_ignore_ascii_case("FERRIS"));
assert!("Ferrös".eq_ignore_ascii_case("FERRöS"));
assert!(!"Ferrös".eq_ignore_ascii_case("FERRÖS"));
```

Run

```
pub fn make_ascii_uppercase(&mut self)
```

1.23.0 [src]

Converts this string to its ASCII upper case equivalent in-place.

ASCII letters 'a' to 'z' are mapped to 'A' to 'Z', but non-ASCII letters are unchanged.

To return a new uppercased value without modifying the existing one, use `to_ascii_uppercase`.

```
pub fn make_ascii_lowercase(&mut self)
```

1.23.0 [src]

Converts this string to its ASCII lower case equivalent in-place.

ASCII letters 'A' to 'Z' are mapped to 'a' to 'z', but non-ASCII letters are unchanged.

To return a new lowercased value without modifying the existing one, use `to_ascii_lowercase`.

Trait Implementations

```
impl<'a> FromIterator<String> for Cow<'a, str>
```

1.12.0 [src]

```
impl<'a> FromIterator<&'a char> for String
```

1.17.0 [src]

```
impl<'a> FromIterator<&'a str> for String
```

[src]

```
impl<'a> FromIterator<Cow<'a, str>> for String
```

1.19.0 [src]

```
impl FromIterator<String> for String
```

1.4.0 [src]

```
impl FromIterator<char> for String
```

[src]

```
impl FromStr for String
```

[src]

```
impl Borrow<str> for String
```

[src]

```
impl Display for String
```

[src]

```
impl Ord for String
```

[src]

```
impl Index<RangeFull> for String
```

[src]

```
impl Index<RangeToInclusive<usize>> for String
```

1.26.0 [src]

```
impl Index<RangeInclusive<usize>> for String
```

1.26.0 [src]

Struct String

Methods

- new
- with_capacity
- from_utf8
- from_utf8_lossy
- from_utf16
- from_utf16_lossy
- from_raw_parts
- from_utf8_unchecked
- into_bytes
- as_str
- as_mut_str
- push_str
- capacity
- reserve
- reserve_exact
- try_reserve
- try_reserve_exact
- shrink_to_fit
- shrink_to
- push
- as_bytes
- truncate
- pop
- remove
- retain
- insert
- insert_str
- as_mut_vec
- len
- is_empty
- split_off

```
impl Index<RangeFrom<usize>> for String
```
```
impl Index<Range<usize>> for String [src]
```
```
impl Index<RangeTo<usize>> for String [src]
```
```
impl DerefMut for String 1.3.0 [src]
```
```
impl From<Box<str>> for String 1.18.0 [src]
```
```
impl From<String> for Arc<str> 1.21.0 [src]
```
```
impl From<String> for Box<str> 1.20.0 [src]
```
```
impl From<String> for Rc<str> 1.21.0 [src]
```
```
impl From<String> for Vec<u8> 1.14.0 [src]
```
```
impl<'a> From<String> for Cow<'a, str> [src]
```
```
impl<'a> From<&'a str> for String [src]
```
```
impl<'a> From<Cow<'a, str>> for String 1.14.0 [src]
```
```
impl AsRef<[u8]> for String [src]
```
```
impl AsRef<str> for String [src]
```
```
impl Eq for String [src]
```
```
impl Hash for String [src]
```
```
impl Clone for String [src]
```
```
impl IndexMut<Range<usize>> for String 1.3.0 [src]
```
```
impl IndexMut<RangeToInclusive<usize>> for String 1.26.0 [src]
```
```
impl IndexMut<RangeFull> for String 1.3.0 [src]
```
```
impl IndexMut<RangeTo<usize>> for String 1.3.0 [src]
```
```
impl IndexMut<RangeFrom<usize>> for String 1.3.0 [src]
```
```
impl IndexMut<RangeInclusive<usize>> for String 1.26.0 [src]
```
```
impl<'a> AddAssign<&'a str> for String 1.12.0 [src]
```
```
impl Default for String [src]
```
```
impl Deref for String [src]
```
```
impl Write for String [src]
```
```
impl ToString for String 1.17.0 [src]
```
```
impl<'a, 'b> Pattern<'a> for &'b String [src]
```
```
type Searcher = <&'b str as Pattern<'a>>::Searcher
```
```
impl<'a, 'b> PartialEq<str> for String [src]
```
```
impl<'a, 'b> PartialEq<Cow<'a, str>> for String [src]
```
```
impl<'a, 'b> PartialEq<&'a str> for String [src]
```
```
impl<'a, 'b> PartialEq<String> for str [src]
```
```
impl<'a, 'b> PartialEq<String> for Cow<'a, str> [src]
```
```
impl PartialEq<String> for String [src]
```
```
impl<'a, 'b> PartialEq<String> for &'a str [src]
```
```
impl Extend<String> for String 1.4.0 [src]
```
```
impl<'a> Extend<Cow<'a, str>> for String 1.19.0 [src]
```
```
impl<'a> Extend<&'a char> for String 1.2.0 [src]
```
```
impl Extend<char> for String [src]
```
```
impl<'a> Extend<&'a str> for String [src]
```
```
impl<'a> Add<&'a str> for String [src]
```

Struct String

Methods

- new
- with_capacity
- from_utf8
- from_utf8_lossy
- from_utf16
- from_utf16_lossy
- from_raw_parts
- from_utf8_unchecked
- into_bytes
- as_str
- as_mut_str
- push_str
- capacity
- reserve
- reserve_exact
- try_reserve
- try_reserve_exact
- shrink_to_fit
- shrink_to
- push
- as_bytes
- truncate
- pop
- remove
- retain
- insert
- insert_str
- as_mut_vec
- len
- is_empty
- split_off

```
type Output = String
```

```
impl PartialOrd<String> for String [src]
impl Debug for String [src]
impl From<String> for Box<Error + Send + Sync> [src]
impl From<String> for Box<Error> 1.6.0 [src]
impl From<String> for OsString [src]
impl AsRef<OsStr> for String [src]
impl ToSocketAddrs for String 1.16.0 [src]
impl From<String> for PathBuf [src]
impl AsRef<Path> for String [src]
```

Auto Trait Implementations

```
impl Send for String
impl Sync for String
```