Variable sized arrays / Vectors

```
let mut vec: Vec<T> = Vec::new();
= Vec::with_capacity();
= vec![];
= Vec::from(slice|str|VecDeque|CString)
= othervec.clone();                          ▼ Details
                                             if T:Clone
```

Accessing

```
vec[3];
                                             ▼ Details
                                             vec[1..3], vec[..3], vec[3..], vec[..]; vec[2] = a;
vec.len();
.is_empty();
.first_mut(); .last_mut();                   ▼ Details
                                             -> Option
.get_mut(index);                             ▼ Details
                                             -> Option
.contains(needle);                           ▼ Details
                                             -> bool
.iter().find(|&T| -> bool);                  ▼ Details
                                             -> Option
.binary_search(x:&T);
                                             ▼ Details
                                             -> Result<usize, usize>
                                             Ok(i): pos, Err(i): pos for insertion
```

Adding

```
.push(3);                                    ▼ Details
                                             to end
.insert(index, element);
.extend(iterable);
.extend_from_slice(&[T]);
.append(other : Vec);                        ▼ Details
                                             drains other
```

Removing

```
.pop();                                      ▼ Details
                                             removes last -> Option
.remove(index);                              ▼ Details
                                             -> el, shifts left
.swap_remove(index);                         ▼ Details
                                             -> el, fills with last
.drain(range);                               ▼ Details
                                             -> iter that drains
.clear();
.retain(|i| -> bool);                        ▼ Details
                                             in place
```

Manipulating

```
.sort();                                     ▼ Details
                                             in place
.sort_by(|&T|->Ordering);                    ▼ Details
                                             in place
.sort_by_key(|&T|->Key);                     ▼ Details
                                             Key:Ordering
.reverse();                                  ▼ Details
                                             in place
.swap(index1, index2);
```

Transforming (Iter, as_, to_)

```
.iter_mut();                                 ▼ Details
```

Transforming (Iter, as_, to_)

```
.iter_mut();                                 ▼ Details
                                             ->&mut T, keeps vector
.into_iter();                                ▼ Details
                                             ->T, consumes vector
.chunks_mut(cnk_sz);
                                             ▼ Details
                                             -> iter over a non overlapping slice at a time
.windows(wnd_sz);
                                             ▼ Details
                                             -> iter over an overlapping slice at a time
```

```
.as_ref();                                   ▼ Details
                                             -> &[T] or &Vec<T>
.as_mut_slice();                             ▼ Details
                                             -> &mut[T]
```

Memory

```
.reserve(100);                               ▼ Details
                                             in addition to .len() or more
.reserve_exact(100);                         ▼ Details
                                             in addition to .len()
```

Split

```
.split_at_mut(mid);                          ▼ Details
                                             -> (p1, p2), [mid] in 2nd part
.split_mut(|&T| -> bool);
.splitn_mut(n, |&T| -> bool); .rsplitn_mut(_);
                                             ▼ Details
                                             -> iter over mutable subslices,
                                             seperated by ||->true, at most n times
.split_off(mid);                             ▼ Details
                                             -> Vec; [mid] in 2nd part
```

Comparision

Traits

From<BinaryHeap> **from()** | **BorrowMut** **borrow_mut()** | **Clone** **clone/_from()** | **Hash** **hash/_slice()** | **IndexMut** **index/_mut()** **DerefMut** **deref/_mut()** | **FromIterator** **from_iter()** | **IntoIterator** **into_iter()** | **Extend** **extend()** | **PartialEq** **eq()** **ne()** | **PartialOrd** **partial_cmp()** **lt()** **le()** **gt()** **ge()** | **Eq** | **Ord** **cmp()** | **Drop** **drop()** | **Default** **default()** | **Debug (if T:Debug)** **fmt()** | **AsRef** **AsMut** **as_ref()** **as_mut()** | **From** **from()** | **Write** **write()** **write_all()** **flush()** **by_ref()** .. |

Hashmaps / Dicts

```
use std::collections::HashMap;               ▼ Details
let mut foo: HashMap<K, V> = HashMap::new();
= HashMap::with_capacity();                  ▼ Details
                                             K: Eq, Hash
= other.clone();                             ▼ Details
                                             if V,K:Clone
```

Access

```
foo[key];
foo.len();
.iter_mut();                                 ▼ Details
                                             -> iter over (&K, &mut V)
.into_iter();                                ▼ Details
                                             -> IterMut
.keys();                                     ▼ Details
                                             -> iter over keys
.values_mut();                               ▼ Details
                                             -> iter over values
.is_empty();                                 ▼ Details
                                             -> bool
.contains_key(k:Q);                          ▼ Details
                                             -> bool
```

Manipulate

```
.get_mut(k:&Q);                              ▼ Details
                                             -> Option<&V>, K:Borrow<Q>
.entry(key);                                 ▼ Details
                                             in place manipulation
.drain();                                    ▼ Details
                                             -> iter that drains
.clear();
.extend(iter : <Item=(&K,&V)>);
.insert(k,v);                                ▼ Details
                                             -> Option<&V>, None on success.
.remove(k:&Q);                               ▼ Details
                                             -> Option<&V>
.from_iter(iter : <Item=(K,V)>);             ▼ Details
                                             -> HashMap
```

Manage

```
.capacity();
.reserve(additional);
.shrink_to_fit();
.clone_from(source);                         ▼ Details
                                             overrides self
```

Comparision

```
.eq() .ne();                                 ▼ Details
                                             T: PartialEq
```

Special Hasher

```
let hm = HashMap::with_hasher(b);
= HashMap::with_capacity_and_hasher(b);
hm.hasher(b);                                ▼ Details
                                             -> &BuildHasher
```

Traits

Clone **clone()** **clone_from()** | **PartialEq** **eq()** **ne()** | **Eq** | **Debug** **fmt()** | **Default** **default()** | **Index** **index()** | **IntoIterator** **into_iter()** | **FromIterator** **from_iter()** | **Extend** **extend()** |

Option

```
let foo : Option = Some(T::new());
= None;
```

If

```
.is_some();
.is_none();                                  ▼ Details
```

&

```
.as_ref();                                   ▼ Details
                                             -> Option<&T>
.as_mut();                                   ▼ Details
                                             -> Option<&mut T>
.cloned();                                   ▼ Details
                                             Option<&T> -> Option<T> if T:Clone
.iter_mut();                                 ▼ Details
                                             Iter with 1 or 0 elements
```

Retrieve T

```
.unwrap();                                   ▼ Details
                                             -> T or panic
.expect(msg);                                ▼ Details
                                             -> T or panic(msg)
.unwrap_or(default:T);                       ▼ Details
                                             -> T
.unwrap_or_default();                        ▼ Details
                                             -> T, if T:Default
.unwrap_or_else(|| -> T);                    ▼ Details
                                             -> T
mutableopt.take();
                                             ▼ Details
                                             -> Option<T>, moves T out of mutableopt
```

Manipulate (map)

```
.map(|t| -> U);                              ▼ Details
                                             -> Option<U>
.map_or(default:U, |t| -> U);                ▼ Details
                                             -> Option<U>
.map_or_else(|| default -> U, |t| -> U);
                                             ▼ Details
                                             -> Option<U>
```

to Result<>

```
.ok_or(err:E);                               ▼ Details
                                             -> Result<T,E>
.ok_or_else(|| err -> E);                    ▼ Details
                                             -> Result<T,E>
```

Boolean Combinations

```
a.and(b : Option<U>);                        ▼ Details
                                             b if a && b
a.and_then(|| b -> Option<U>);               ▼ Details
                                             b if a && b
a.or(b : Option<T>);                         ▼ Details
                                             a if a else b
a.or_else(|| b -> Option<T>);                ▼ Details
                                             a if a else b
```

Traits

Hash **hash()** | **Debug** **fmt()** | **Ord** **cmp()** | **Eq** | **PartialOrd** **partial_cmp()** **lt()** **le()** **gt()** **ge()** | **PartialEq** **eq()** **ne()** | **Copy** | **Clone** **clone()** **clone_from()** | **Default** **default()** | **IntoIterator** **into_iter()** | **FromIterator** **from_iter()** |

Result

```
let foo : Result = Ok(T::new());
= Err(E::new());
```

If

```
.is_ok();
.is_err();                                   ▼ Details
```

&

```
.as_ref();                                   ▼ Details
                                             -> Result<&T, &E>
.as_mut();                                   ▼ Details
                                             -> Option<&mut T, &mut E>
.iter_mut();                                 ▼ Details
                                             Iter with 1 or 0 elements
```

Retrieve T

```
.unwrap();                                   ▼ Details
                                             -> T or panic; if E:Debug
.expect(msg);                                ▼ Details
                                             -> T or panic(msg); if E:Debug
.unwrap_or(default:T);                       ▼ Details
                                             -> T
.unwrap_or_else(|err| default -> T);         ▼ Details
                                             -> T
```

Retrieve E

```
.unwrap_err();                               ▼ Details
                                             -> E; if T:Debug
```

Manipulate (map)

```
.map(|t| -> U);                              ▼ Details
                                             -> Result<U,E>
.map_err(|e| -> F);                          ▼ Details
                                             -> Result<T,F>
```

to Option<>

```
.ok();                                       ▼ Details
                                             -> Option<T>
.err();                                      ▼ Details
                                             -> Option<E>
```

Boolean Combinations

```
a.and(b : Result<U,E>);                      ▼ Details
                                             b if a && b else first err
a.and_then(|| b -> Result<U,E>);
                                             ▼ Details
                                             b if a && b else first error
a.or(b : Result<T,E>);                       ▼ Details
                                             a if a else b
a.or_else(|| b -> Result<T,E>);              ▼ Details
                                             a if a else b
```

Traits

Hash **hash()** | **Debug** **fmt()** | **Ord** **cmp()** | **Eq** | **PartialOrd** **partial_cmp()** **lt()** **le()** **gt()** **ge()** | **PartialEq** **eq()** **ne()** | **Copy** | **Clone** **clone()** **clone_from()** | **IntoIterator** **into_iter()** | **FromIterator** **from_iter()** |

Contribute at github.com/phaiax/rust-cheatsheet

▼ Details

The Periodic Table of Rust Types

	Immutable Pointer	Mutable Pointer	Owned Pointer	Bare	Unsized
Raw	`*const T` Immutable raw pointer	`*mut T` Mutable raw pointer	*Raw pointers do not have ownership, `*const T` or `*mut T` should be used as appropriate. See also `Unique<T>`.*		
Simple	`&'r T` Immutable borrowed reference	`&'r mut T` Mutable borrowed reference	`Box<T>` Owned pointer	`T` Primitive type, struct, enum and so on	
Trait	`&'r (Trait + K)` Immutable borrowed trait object	`&'r mut (Trait + K)` Mutable borrowed trait object	`Box<Trait + K>` Owned trait object		`Trait + K` Unsized trait type
Array	`&'r [T]` Immutable borrowed slice	`&'r mut [T]` Mutable borrowed slice	`Box<[T]>` Owned array `Vec<T>` Owned growable vector	`[T; n]` Fixed-size array	`[T]` Unsized array type
String	`&'r str` Immutable borrowed string slice	`&'r mut str` Mutable borrowed string slice (not that useful)	`Box<str>` Owned string (theoretical) `String` Owned growable string	*Fixed sized storage is impractical for the variable length UTF-8 encoding.*	`str` Unsized string type
Callable	`Fn(T…) -> U + K` Closure with immutable environment	`FnMut(T…) -> U + K` Closure with mutable environment	`FnOnce(T…) -> U + K` Closure with owned environment	`fn(T…) -> U` Bare function type	

☐ Supported as of 1.0.0-alpha
☐ Provided by the standard library
■ Impossible

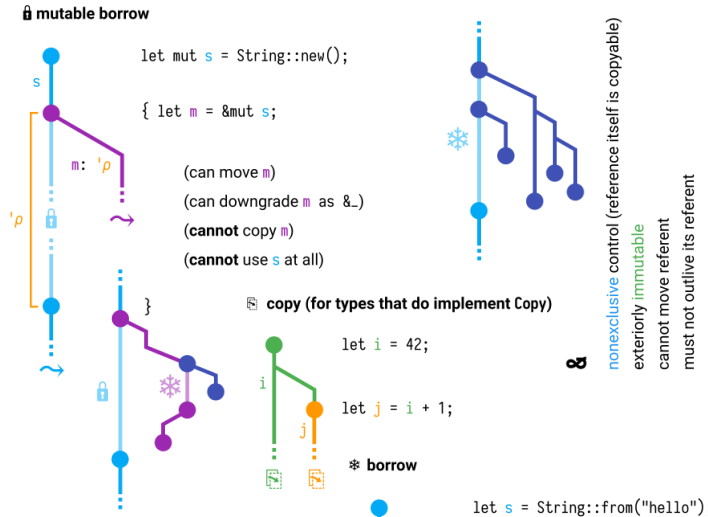
☐ `'r` Lifetime
☐ `K` Trait bounds
☐ `T…` Function arguments
☐ `-> U` Function return
☐ `extern "ABI"` ABI definition

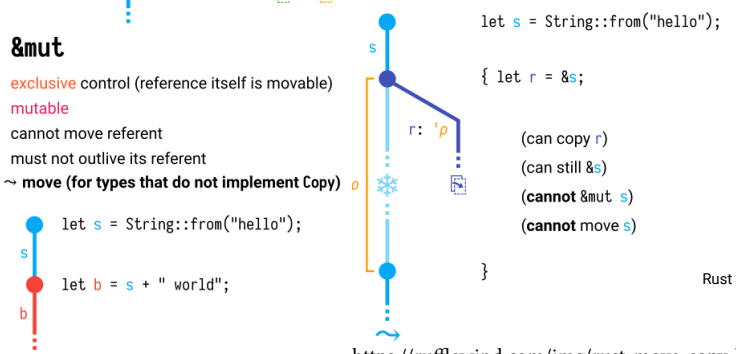
Lifetime Reference

Sat, Oct 31, 2015 http://www.charlesetc.com/lifetime-reference/

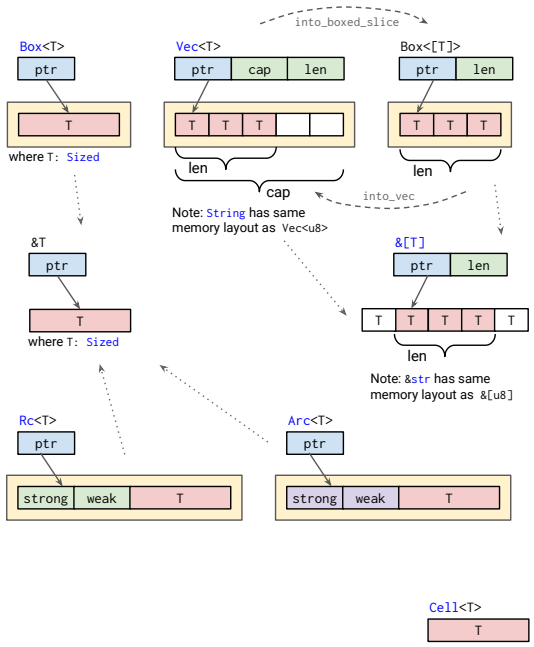
Places in Rust where you use any lifetime syntax will fall into two categories:

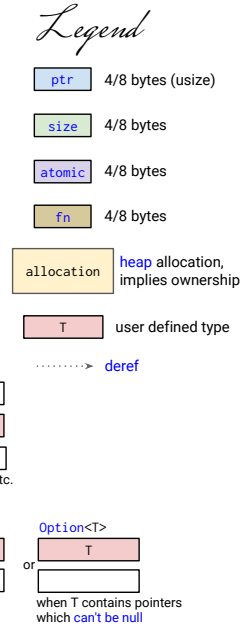
Concept	Category	Usage
`fn`	creation	`fn example_function<'a>()`
`struct`	creation	`struct Example<'a>`
`enum`	creation	`enum Test<'a>`
`impl`	creation	`impl<'a> Example<'a>`
`struct`	reference	`some_field: Example<'a>`
`enum`	reference	`some_field: Test<'a>`
`&`	reference	`next_field: &'a i32`
`&mut`	reference	`next_field: &'a mut i32`
`Box`	reference	`last_field: Box<i32 + 'a>`





https://rufflewind.com/img/rust-move-copy-borrow.png





References

	Can `Copy`?	Can mutate through?
`&T`	yes	no
`&mut T`	no	yes

This demonstrates that neither `&T` nor `&mut T` is a subtype of the other, in the Liskov sense.

Ownership and mutability

Ownership controls when a value is destroyed. A value can have either a unique owner, or a number of references which collectively share ownership. The latter case usually involves reference counting.

Interior mutability refers to any wrapper type `Wrapper` such that we can go from `&Wrapper<T>` to `&mut T`, or at least have some of the capabilities of `&mut T`.

The column headings here refer (more or less) to the point in time at which Rust's safety invariants are checked. Note that no unsafety can occur due to sharing the thread-unsafe structures between threads. The compiler will simply reject your code, through the magic of the `Sync` trait.

	Static	Dynamic	Dynamic, thread-safe
Direct ownership	`T`		
Ownership via heap	`Box<T>`		
Shared ownership		`Rc<T>`	`Arc<T>`
Get, set, compare & swap, etc.	`&mut T`	`Cell<T>`	`AtomicFoo`
Borrow immutably	`&T`		
Borrow mutably, or single reader			`Mutex<T>`
Borrow mutably, or multiple readers	`&mut T`	`RefCell<T>`	`RwLock<T>`
Borrow mutably, unsafe	`static mut`	`UnsafeCell<T>`	`UnsafeCell<T>`

Numeric types

	Unsigned integer	Signed integer	Floating-point
8 bits	`u8`	`i8`	
16 bits	`u16`	`i16`	
32 bits	`u32`	`i32`	`f32`
64 bits	`u64`	`i64`	`f64`
128 bits	`u128` [α]	`i128` [α]	
Pointer-sized	`usize`	`isize`	

[α] Nightly-only, as of rustc 1.18.

Strings

Format	Borrow[β]	Borrow substr?	Mutate	Copy on write	Owned, in heap
Any bytes	`&[u8]`	yes	`&mut [u8]`	`Cow<[u8]>`	`Vec<u8>`
UTF-8	`&str`	yes	`&mut str` [α]	`Cow<str>`	`String`
Platform-dependent	`&OsStr`	no		`Cow<OsStr>`	`OsString`
Filesystem path	`&Path`	no		`Cow<Path>`	`PathBuf`
`NUL`-terminated, safe	`&CStr`	no		`Cow<CStr>`	`CString`
`NUL`-terminated, raw	`*const c_char` [γ]	yes[δ]	`*mut c_char` [γ]		`*mut c_char` [γε]

[α] Nearly useless, because most mutations could change the length of a UTF-8 codepoint. One exception is ASCII-only case conversion.

[β] In most cases, you can borrow static memory (e.g. a string literal) with a type like `&'static str`.

[γ] With raw pointers, you are on your own regarding ownership / borrowing semantics. Any good C library will document its expectations.

[δ] You can slice off the front of a `NUL`-terminated string, but not the end.

[ε] On the general principle that if you own something you can mutate it. But you could use `*const c_char` instead.

https://github.com/kmcallister/rustic-symmetries

The first size is the size of the value itself: the stuff that ends up on the stack if you put it in a `let` variable. The second size is the size of any *owned* data in the heap.

We assume `T`, `A`, `B`, `C` are `Sized`.

Type	Value size	Contents	Heap size
`bool`	1 byte	0 or 1	
`()`	empty!		
`(A, B, C)` `struct`	sum of `A`, `B`, `C` + pad / align	values of type `A`, `B`, `C`	anything owned by `A`, `B`, or `C`
`enum`	size of tag + max of variants + pad / align	tag + one variant	anything owned by variant
`[T; n]`	`n` × size of `T`	`n` elements of type `T`	anything owned by `T`
`&T`, `&mut T` `*const T`, `*mut T`	1 word	pointer	
`Box<T>`	1 word	pointer	size of `T`
`Option<T>`	1 word + size of `T` + pad / align (but see below)	tag + optionally `T`	anything owned by `T`, if `Some`
`Option<&T>` `Option<&mut T>`	1 word[β]	pointer or `NULL`	
`Option<Box<T>>`	1 word[β]	pointer or `NULL`	size of `T`, if `Some`
`[T]`, `str`	dynamic size	elements or codepoints	
`&[T]`	2 words	pointer, length (in elements)	
`&str`	2 words	pointer, length (in bytes)	
`Box<[T]>`	2 words	pointer, length (in elements)	length × size of `T`
`Box<str>`	2 words	pointer, length (in bytes)	length (bytes)
`Vec<T>`	3 words	pointer, length, capacity	capacity × size of `T`
`String`	3 words	pointer, length, capacity	capacity (bytes)
`Trait`	dynamic size	fields of concrete type	anything owned by fields
`&Trait`	2 words	pointer to concrete value, pointer to vtable	
`Box<Trait>`	2 words	pointer to concrete value, pointer to vtable	size of concrete value
Specific `fn` used as a value[α]	empty!		
Specific lambda	depends on captures, but known statically	captures	anything owned by captures

`fn(A) -> B`
`unsafe fn(A) -> B`
`extern fn(A) -> B`
	1 word[α]	pointer to code	

`PhantomData<T>`	empty!		
`Rc<T>` `Arc<T>`	1 word	pointer	2 words + size of `T` + pad / align
`Cell<T>`	size of `T`	`T`	anything owned by `T`
`AtomicT`	size of `T`	`T`	
`RefCell<T>`	1 word + size of `T` + pad / align	borrow flag, `T`	anything owned by `T`
`Mutex<T>` `RwLock<T>`	2 words + size of `T` + pad / align	poison flag, pointer to OS mutex, `T`	anything owned by `T` + OS mutex

[α] These are function pointers. Technically, they can have a different size from a data pointer, but this does not happen on common architectures.

[β] This optimization actually applies to any `Option`-shaped enum which contains, somewhere, a field which cannot be 0.