Introduction

This book is the primary reference for the Rust programming language. It provides three kinds of material:

- Chapters that informally describe each language construct and their use.
- Chapters that informally describe the memory model, concurrency model, runtime services, linkage model, and debugging facilities.
- Appendix chapters providing rationale and references to languages that influenced the design.

> ⚠️ Warning: This book is incomplete. Documenting everything takes a while. See the GitHub issues for what is not documented in this book.

Rust releases

Rust has a new language release every six weeks. The first stable release of the language was Rust 1.0.0, followed by Rust 1.1.0 and so on. Tools (`rustc` , `cargo` , etc.) and documentation (Standard library, this book, etc.) are released with the language release.

The latest release of this book, matching the latest Rust version, can always be found at https://doc.rust-lang.org/reference/. Prior versions can be found by adding the Rust version before the "reference" directory. For example, the Reference for Rust 1.49.0 is located at https://doc.rust-lang.org/1.49.0/reference/.

What *The Reference* is not

This book does not serve as an introduction to the language. Background familiarity with the language is assumed. A separate book is available to help acquire such background familiarity.

This book also does not serve as a reference to the standard library included in the language distribution. Those libraries are documented separately by extracting documentation attributes from their source code. Many of the features that one might expect to be language features are library features in Rust, so what you're looking for may be there, not here.

Similarly, this book does not usually document the specifics of `rustc` as a tool or of Cargo. `rustc` has its own book. Cargo has a book that contains a reference. There are a few pages such as linkage that still describe how `rustc` works.

This book also only serves as a reference to what is available in stable Rust. For unstable features being worked on, see the Unstable Book.

Rust compilers, including `rustc`, will perform optimizations. The reference does not specify what optimizations are allowed or disallowed. Instead, think of the compiled program as a black box. You can only probe by running it, feeding it input and observing its output. Everything that happens that way must conform to what the reference says.

Finally, this book is not normative. It may include details that are specific to `rustc` itself, and should not be taken as a specification for the Rust language. We intend to produce such a book someday, and until then, the reference is the closest thing we have to one.

How to use this book

This book does not assume you are reading this book sequentially. Each chapter generally can be read standalone, but will cross-link to other chapters for facets of the language they refer to, but do not discuss.

There are two main ways to read this document.

The first is to answer a specific question. If you know which chapter answers that question, you can jump to that chapter in the table of contents. Otherwise, you can press `s` or click the magnifying glass on the top bar to search for keywords related to your question. For example, say you wanted to know when a temporary value created in a let statement is dropped. If you didn't already know that the lifetime of temporaries is defined in the expressions chapter, you could search "temporary let" and the first search result will take you to that section.

The second is to generally improve your knowledge of a facet of the language. In that case, just browse the table of contents until you see something you want to know more about, and just start reading. If a link looks interesting, click it, and read about that section.

That said, there is no wrong way to read this book. Read it however you feel helps you best.

Conventions

Like all technical books, this book has certain conventions in how it displays information. These conventions are documented here.

- Statements that define a term contain that term in *italics*. Whenever that term is used outside of that chapter, it is usually a link to the section that has this definition.

  An *example term* is an example of a term being defined.

- Differences in the language by which edition the crate is compiled under are in a blockquote that start with the words "Edition Differences:" in **bold**.

  > **Edition Differences**: In the 2015 edition, this syntax is valid that is disallowed as of the 2018 edition.

- Notes that contain useful information about the state of the book or point out useful, but mostly out of scope, information are in blockquotes that start with the word "Note:" in **bold**.

  > **Note**: This is an example note.

- Warnings that show unsound behavior in the language or possibly confusing interactions of language features are in a special warning box.

  > ⚠️ Warning: This is an example warning.

- Code snippets inline in the text are inside `<code>` tags.

  Longer code examples are in a syntax highlighted box that has controls for copying, executing, and showing hidden lines in the top right corner.

  ```
  fn main() {
      println!("This is a code example");
  }
  ```

  All examples are written for the latest edition unless otherwise stated.

- The grammar and lexical structure is in blockquotes with either "Lexer" or "Syntax" in

bold superscript as the first line.

> **Syntax**
> *ExampleGrammar*:
>   `~` *Expression*
> | `box` *Expression*

See Notation for more detail.

Contributing

We welcome contributions of all kinds.

You can contribute to this book by opening an issue or sending a pull request to the Rust Reference repository. If this book does not answer your question, and you think its answer is in scope of it, please do not hesitate to file an issue or ask about it in the `t-lang/doc` stream on Zulip. Knowing what people use this book for the most helps direct our attention to making those sections the best that they can be. We also want the reference to be as normative as possible, so if you see anything that is wrong or is non-normative but not specifically called out, please also file an issue.

Notation

Grammar

The following notations are used by the *Lexer* and *Syntax* grammar snippets:

Notation	Examples	Meaning
CAPITAL	KW_IF, INTEGER_LITERAL	A token produced by the lexer
ItalicCamelCase	*LetStatement*, *Item*	A syntactical production
`string`	`x`, `while`, `*`	The exact character(s)
\x	\n, \r, \t, \0	The character represented by this escape
$x^?$	`pub` $^?$	An optional item
x^*	*OuterAttribute**	0 or more of x
x^+	*MacroMatch*$^+$	1 or more of x
$x^{a..b}$	HEX_DIGIT$^{1..6}$	a to b repetitions of x
|	`u8` | `u16`, Block | Item	Either one or another
[ ]	[`b` `B`]	Any of the characters listed
[-]	[`a`-`z`]	Any of the characters in the range
~[ ]	~[`b` `B`]	Any characters, except those listed
~ `string`	~`\n`, ~`*/`	Any characters, except this sequence
( )	(`,` *Parameter*)$^?$	Groups items

String table productions

Some rules in the grammar — notably unary operators, binary operators, and keywords — are given in a simplified form: as a listing of printable strings. These cases form a subset of the rules regarding the token rule, and are assumed to be the result of a lexical-analysis phase feeding the parser, driven by a DFA, operating over the disjunction of all such string table entries.

When such a string in `monospace` font occurs inside the grammar, it is an implicit reference to a single member of such a string table production. See tokens for more information.

Lexical structure

Input format

This chapter describes how a source file is interpreted as a sequence of tokens.

See Crates and source files for a description of how programs are organised into files.

Source encoding

Each source file is interpreted as a sequence of Unicode characters encoded in UTF-8. It is an error if the file is not valid UTF-8.

Byte order mark removal

If the first character in the sequence is `U+FEFF` (BYTE ORDER MARK), it is removed.

CRLF normalization

Each pair of characters `U+000D` (CR) immediately followed by `U+000A` (LF) is replaced by a single `U+000A` (LF).

Other occurrences of the character `U+000D` (CR) are left in place (they are treated as whitespace).

Shebang removal

If the remaining sequence begins with the characters `#!`, the characters up to and including the first `U+000A` (LF) are removed from the sequence.

For example, the first line of the following file would be ignored:

```
#!/usr/bin/env rustx

fn main() {
    println!("Hello!");
}
```

As an exception, if the `#!` characters are followed (ignoring intervening comments or whitespace) by a `[` token, nothing is removed. This prevents an inner attribute at the start of a source file being removed.

> **Note**: The standard library `include!` macro applies byte order mark removal, CRLF normalization, and shebang removal to the file it reads. The `include_str!` and `include_bytes!` macros do not.

Tokenization

The resulting sequence of characters is then converted into tokens as described in the remainder of this chapter.

Keywords

Rust divides keywords into three categories:

- strict
- reserved
- weak

Strict keywords

These keywords can only be used in their correct contexts. They cannot be used as the names of:

- Items
- Variables and function parameters
- Fields and variants
- Type parameters
- Lifetime parameters or loop labels
- Macros or attributes
- Macro placeholders
- Crates

Lexer:

KW_AS : `as`

KW_BREAK : `break`

KW_CONST : `const`

KW_CONTINUE : `continue`

KW_CRATE : `crate`

KW_ELSE : `else`

KW_ENUM : `enum`

KW_EXTERN : `extern`

KW_FALSE : `false`

KW_FN : `fn`

KW_FOR : `for`

KW_IF : `if`

KW_IMPL : `impl`

KW_IN : `in`

KW_LET : `let`
KW_LOOP : `loop`
KW_MATCH : `match`
KW_MOD : `mod`
KW_MOVE : `move`
KW_MUT : `mut`
KW_PUB : `pub`
KW_REF : `ref`
KW_RETURN : `return`
KW_SELFVALUE : `self`
KW_SELFTYPE : `Self`
KW_STATIC : `static`
KW_STRUCT : `struct`
KW_SUPER : `super`
KW_TRAIT : `trait`
KW_TRUE : `true`
KW_TYPE : `type`
KW_UNSAFE : `unsafe`
KW_USE : `use`
KW_WHERE : `where`
KW_WHILE : `while`

The following keywords were added beginning in the 2018 edition.

Lexer 2018+
KW_ASYNC : `async`
KW_AWAIT : `await`
KW_DYN : `dyn`

Reserved keywords

These keywords aren't used yet, but they are reserved for future use. They have the same restrictions as strict keywords. The reasoning behind this is to make current programs forward compatible with future versions of Rust by forbidding them to use these keywords.

Lexer

KW_ABSTRACT : `abstract`
KW_BECOME : `become`
KW_BOX : `box`
KW_DO : `do`
KW_FINAL : `final`
KW_MACRO : `macro`
KW_OVERRIDE : `override`
KW_PRIV : `priv`
KW_TYPEOF : `typeof`
KW_UNSIZED : `unsized`
KW_VIRTUAL : `virtual`
KW_YIELD : `yield`

The following keywords are reserved beginning in the 2018 edition.

Lexer 2018+

KW_TRY : `try`

Weak keywords

These keywords have special meaning only in certain contexts. For example, it is possible to declare a variable or method with the name `union`.

- `macro_rules` is used to create custom macros.
- `union` is used to declare a union and is only a keyword when used in a union declaration.
- `'static` is used for the static lifetime and cannot be used as a generic lifetime parameter or loop label

```
// error[E0262]: invalid lifetime parameter name: `'static`
fn invalid_lifetime_parameter<'static>(s: &'static str) -> &'static str { s
}
```

- In the 2015 edition, `dyn` is a keyword when used in a type position followed by a path that does not start with `::`.

  Beginning in the 2018 edition, `dyn` has been promoted to a strict keyword.

Lexer

KW_MACRO_RULES : `macro_rules`
KW_UNION : `union`
KW_STATICLIFETIME : `'static`

Lexer 2015

KW_DYN : `dyn`

Identifiers

> **Lexer:**
>
> IDENTIFIER_OR_KEYWORD :
>
> XID_Start XID_Continue*
>
> | _ XID_Continue+
>
> RAW_IDENTIFIER : `r#` IDENTIFIER_OR_KEYWORD *Except `crate`, `self`, `super`, `Self`*
>
> NON_KEYWORD_IDENTIFIER : IDENTIFIER_OR_KEYWORD *Except a strict or reserved keyword*
>
> IDENTIFIER :
> NON_KEYWORD_IDENTIFIER | RAW_IDENTIFIER

Identifiers follow the specification in Unicode Standard Annex #31 for Unicode version 15.0, with the additions described below. Some examples of identifiers:

- `foo`
- `_identifier`
- `r#true`
- `Москва`
- `東京`

The profile used from UAX #31 is:

- Start := `XID_Start` , plus the underscore character (U+005F)
- Continue := `XID_Continue`
- Medial := empty

with the additional constraint that a single underscore character is not an identifier.

> **Note**: Identifiers starting with an underscore are typically used to indicate an identifier that is intentionally unused, and will silence the unused warning in `rustc` .

Identifiers may not be a strict or reserved keyword without the `r#` prefix described below in raw identifiers.

Zero width non-joiner (ZWNJ U+200C) and zero width joiner (ZWJ U+200D) characters are not allowed in identifiers.

Identifiers are restricted to the ASCII subset of `XID_Start` and `XID_Continue` in the following situations:

- `extern crate` declarations
- External crate names referenced in a path
- Module names loaded from the filesystem without a `path` attribute
- `no_mangle` attributed items
- Item names in external blocks

Normalization

Identifiers are normalized using Normalization Form C (NFC) as defined in Unicode Standard Annex #15. Two identifiers are equal if their NFC forms are equal.

Procedural and declarative macros receive normalized identifiers in their input.

Raw identifiers

A raw identifier is like a normal identifier, but prefixed by `r#`. (Note that the `r#` prefix is not included as part of the actual identifier.) Unlike a normal identifier, a raw identifier may be any strict or reserved keyword except the ones listed above for `RAW_IDENTIFIER`.

Comments

Lexer

LINE_COMMENT :
  `//` (~[`/` `!` `\n`] | `//`) ~`\n`*
  | `//`

BLOCK_COMMENT :
  `/*` (~[`*` `!`] | `**` | *BlockCommentOrDoc*) (*BlockCommentOrDoc* | ~`*/`)* `*/`
  | `/**/`
  | `/***/`

INNER_LINE_DOC :
  `//!` ~[`\n` *IsolatedCR*]*

INNER_BLOCK_DOC :
  `/*!` (*BlockCommentOrDoc* | ~[`*/` *IsolatedCR*])* `*/`

OUTER_LINE_DOC :
  `///` (~ `/` ~[`\n` *IsolatedCR*]*)?

OUTER_BLOCK_DOC :
  `/**` (~`*` | *BlockCommentOrDoc*) (*BlockCommentOrDoc* | ~[`*/` *IsolatedCR*])* `*/`

BlockCommentOrDoc :
  BLOCK_COMMENT
  | OUTER_BLOCK_DOC
  | INNER_BLOCK_DOC

IsolatedCR :
  \r

Non-doc comments

Comments follow the general C++ style of line (`//`) and block (`/* ... */`) comment forms. Nested block comments are supported.

Non-doc comments are interpreted as a form of whitespace.

Doc comments

Line doc comments beginning with exactly *three* slashes (`///`), and block doc comments (`/** ... */`), both outer doc comments, are interpreted as a special syntax for `doc` attributes. That is, they are equivalent to writing `#[doc="..."]` around the body of the comment, i.e., `/// Foo` turns into `#[doc="Foo"]` and `/** Bar */` turns into `#[doc="Bar"]` .

Line comments beginning with `//!` and block comments `/*! ... */` are doc comments that apply to the parent of the comment, rather than the item that follows. That is, they are equivalent to writing `#![doc="..."]` around the body of the comment. `//!` comments are usually used to document modules that occupy a source file.

The character `U+000D` (CR) is not allowed in doc comments.

Note: The sequence `U+000D` (CR) immediately followed by `U+000A` (LF) would have been previously transformed into a single `U+000A` (LF).

Examples

```rust
//! A doc comment that applies to the implicit anonymous module of this crate

pub mod outer_module {

    //!  - Inner line doc
    //!! - Still an inner line doc (but with a bang at the beginning)

    /*!  - Inner block doc */
    /*!! - Still an inner block doc (but with a bang at the beginning) */

    //   - Only a comment
    ///  - Outer line doc (exactly 3 slashes)
    //// - Only a comment

    /*   - Only a comment */
    /**  - Outer block doc (exactly) 2 asterisks */
    /*** - Only a comment */

    pub mod inner_module {}

    pub mod nested_comments {
        /* In Rust /* we can /* nest comments */ */ */

        // All three types of block comments can contain or be nested inside
        // any other type:

        /*   /* */  /** */  /*! */  */
        /*!  /* */  /** */  /*! */  */
        /**  /* */  /** */  /*! */  */
        pub mod dummy_item {}
    }

    pub mod degenerate_cases {
        // empty inner line doc
        //!

        // empty inner block doc
        /*!*/

        // empty line comment
        //

        // empty outer line doc
        ///

        // empty block comment
        /**/

        pub mod dummy_item {}

        // empty 2-asterisk block isn't a doc block, it is a block comment
        /***/
```

```
    }

    /* The next one isn't allowed because outer doc comments
       require an item that will receive the doc */

    /// Where is my item?
}
```

Whitespace

Whitespace is any non-empty string containing only characters that have the `Pattern_White_Space` Unicode property, namely:

- `U+0009` (horizontal tab, `'\t'`)
- `U+000A` (line feed, `'\n'`)
- `U+000B` (vertical tab)
- `U+000C` (form feed)
- `U+000D` (carriage return, `'\r'`)
- `U+0020` (space, `' '`)
- `U+0085` (next line)
- `U+200E` (left-to-right mark)
- `U+200F` (right-to-left mark)
- `U+2028` (line separator)
- `U+2029` (paragraph separator)

Rust is a "free-form" language, meaning that all forms of whitespace serve only to separate *tokens* in the grammar, and have no semantic significance.

A Rust program has identical meaning if each whitespace element is replaced with any other legal whitespace element, such as a single space character.

Tokens

Tokens are primitive productions in the grammar defined by regular (non-recursive) languages. Rust source input can be broken down into the following kinds of tokens:

- Keywords
- Identifiers
- Literals
- Lifetimes
- Punctuation
- Delimiters

Within this documentation's grammar, "simple" tokens are given in string table production form, and appear in `monospace` font.

Literals

Literals are tokens used in literal expressions.

Examples

Characters and strings

	Example	`#` sets[1]	Characters	Escapes
Character	`'H'`	0	All Unicode	Quote & ASCII & Unicode
String	`"hello"`	0	All Unicode	Quote & ASCII & Unicode
Raw string	`r#"hello"#`	<256	All Unicode	`N/A`
Byte	`b'H'`	0	All ASCII	Quote & Byte
Byte string	`b"hello"`	0	All ASCII	Quote & Byte
Raw byte string	`br#"hello"#`	<256	All ASCII	`N/A`
C string	`c"hello"`	0	All Unicode	Quote & Byte & Unicode

	Example	`#` sets[1]	Characters	Escapes
Raw C string	`cr#"hello"#`	<256	All Unicode	`N/A`

[1] The number of `#` s on each side of the same literal must be equivalent.

Note: Character and string literal tokens never include the sequence of `U+000D` (CR) immediately followed by `U+000A` (LF): this pair would have been previously transformed into a single `U+000A` (LF).

ASCII escapes

	Name
`\x41`	7-bit character code (exactly 2 digits, up to 0x7F)
`\n`	Newline
`\r`	Carriage return
`\t`	Tab
`\\`	Backslash
`\0`	Null

Byte escapes

	Name
`\x7F`	8-bit character code (exactly 2 digits)
`\n`	Newline
`\r`	Carriage return
`\t`	Tab
`\\`	Backslash
`\0`	Null

Unicode escapes

	Name
`\u{7FFF}`	24-bit Unicode character code (up to 6 digits)

Quote escapes

	Name
`\'`	Single quote
`\"`	Double quote

Numbers

Number literals[2]	Example	Exponentiation
Decimal integer	`98_222`	`N/A`
Hex integer	`0xff`	`N/A`
Octal integer	`0o77`	`N/A`
Binary integer	`0b1111_0000`	`N/A`
Floating-point	`123.0E+77`	`Optional`

[2] All number literals allow `_` as a visual separator: `1_234.0E+18f64`

Suffixes

A suffix is a sequence of characters following the primary part of a literal (without intervening whitespace), of the same form as a non-raw identifier or keyword.

Lexer

SUFFIX : IDENTIFIER_OR_KEYWORD
SUFFIX_NO_E : SUFFIX *not beginning with `e` or `E`*

Any kind of literal (string, integer, etc) with any suffix is valid as a token.

A literal token with any suffix can be passed to a macro without producing an error. The macro itself will decide how to interpret such a token and whether to produce an error or not. In particular, the `literal` fragment specifier for by-example macros matches literal tokens with arbitrary suffixes.

```
macro_rules! blackhole { ($tt:tt) => () }
macro_rules! blackhole_lit { ($l:literal) => () }

blackhole!("string"suffix); // OK
blackhole_lit!(1suffix); // OK
```

However, suffixes on literal tokens which are interpreted as literal expressions or patterns are restricted. Any suffixes are rejected on non-numeric literal tokens, and numeric literal tokens are accepted only with suffixes from the list below.

Integer	Floating-point
`u8` , `i8` , `u16` , `i16` , `u32` , `i32` , `u64` , `i64` , `u128` , `i128` , `usize` , `isize`	`f32` , `f64`

Character and string literals

Character literals

Lexer

CHAR_LITERAL :
`'` (~[`'` `\` \n \r \t] | QUOTE_ESCAPE | ASCII_ESCAPE | UNICODE_ESCAPE) `'` SUFFIX?

QUOTE_ESCAPE :
`\'` | `\"`

ASCII_ESCAPE :
`\x` OCT_DIGIT HEX_DIGIT
| `\n` | `\r` | `\t` | `\\` | `\0`

UNICODE_ESCAPE :
`\u{` (HEX_DIGIT `_` *)1..6 `}`

A *character literal* is a single Unicode character enclosed within two `U+0027` (single-quote) characters, with the exception of `U+0027` itself, which must be *escaped* by a preceding `U+005C` character (`\`).

String literals

Lexer

STRING_LITERAL :
  `"` (
   ~[`"` `\` *IsolatedCR*]
   | QUOTE_ESCAPE
   | ASCII_ESCAPE
   | UNICODE_ESCAPE
   | STRING_CONTINUE
)* `"` SUFFIX?

STRING_CONTINUE :
  `\` *followed by* \n

A *string literal* is a sequence of any Unicode characters enclosed within two `U+0022` (double-quote) characters, with the exception of `U+0022` itself, which must be *escaped* by a preceding `U+005C` character (`\`).

Line-breaks, represented by the character `U+000A` (LF), are allowed in string literals. When an unescaped `U+005C` character (`\`) occurs immediately before a line break, the line break does not appear in the string represented by the token. See String continuation escapes for details. The character `U+000D` (CR) may not appear in a string literal other than as part of such a string continuation escape.

Character escapes

Some additional *escapes* are available in either character or non-raw string literals. An escape starts with a `U+005C` (`\`) and continues with one of the following forms:

- A *7-bit code point escape* starts with `U+0078` (`x`) and is followed by exactly two *hex digits* with value up to `0x7F`. It denotes the ASCII character with value equal to the provided hex value. Higher values are not permitted because it is ambiguous whether they mean Unicode code points or byte values.
- A *24-bit code point escape* starts with `U+0075` (`u`) and is followed by up to six *hex digits* surrounded by braces `U+007B` (`{`) and `U+007D` (`}`). It denotes the Unicode code point equal to the provided hex value.
- A *whitespace escape* is one of the characters `U+006E` (`n`), `U+0072` (`r`), or `U+0074` (`t`), denoting the Unicode values `U+000A` (LF), `U+000D` (CR) or `U+0009` (HT) respectively.
- The *null escape* is the character `U+0030` (`0`) and denotes the Unicode value `U+0000` (NUL).

- The *backslash escape* is the character `U+005C` (`\`) which must be escaped in order to denote itself.

Raw string literals

Lexer

RAW_STRING_LITERAL :
  `r` RAW_STRING_CONTENT SUFFIX?

RAW_STRING_CONTENT :
   `"` (~ *IsolatedCR*)* (non-greedy) `"`
  | `#` RAW_STRING_CONTENT `#`

Raw string literals do not process any escapes. They start with the character `U+0072` (`r`), followed by fewer than 256 of the character `U+0023` (`#`) and a `U+0022` (double-quote) character.

The *raw string body* can contain any sequence of Unicode characters other than `U+000D` (CR). It is terminated only by another `U+0022` (double-quote) character, followed by the same number of `U+0023` (`#`) characters that preceded the opening `U+0022` (double-quote) character.

All Unicode characters contained in the raw string body represent themselves, the characters `U+0022` (double-quote) (except when followed by at least as many `U+0023` (`#`) characters as were used to start the raw string literal) or `U+005C` (`\`) do not have any special meaning.

Examples for string literals:

```
"foo"; r"foo";                     // foo
"\"foo\""; r#""foo""#;             // "foo"

"foo #\"# bar";
r##"foo #"# bar"##;                // foo #"# bar

"\x52"; "R"; r"R";                 // R
"\\x52"; r"\x52";                  // \x52
```

Byte and byte string literals

Byte literals

> **Lexer**
>
> BYTE_LITERAL :
>   `b'` (ASCII_FOR_CHAR | BYTE_ESCAPE) `'` SUFFIX?
>
> ASCII_FOR_CHAR :
>   *any ASCII (i.e. 0x00 to 0x7F), except* `'`, `\`, \n, \r or \t
>
> BYTE_ESCAPE :
>   `\x` HEX_DIGIT HEX_DIGIT
> | `\n` | `\r` | `\t` | `\\` | `\0` | `\'` | `\"`

A *byte literal* is a single ASCII character (in the `U+0000` to `U+007F` range) or a single *escape* preceded by the characters `U+0062` (`b`) and `U+0027` (single-quote), and followed by the character `U+0027`. If the character `U+0027` is present within the literal, it must be *escaped* by a preceding `U+005C` (`\`) character. It is equivalent to a `u8` unsigned 8-bit integer *number literal*.

Byte string literals

> **Lexer**
>
> BYTE_STRING_LITERAL :
>   `b"` (ASCII_FOR_STRING | BYTE_ESCAPE | STRING_CONTINUE)* `"` SUFFIX?
>
> ASCII_FOR_STRING :
>   *any ASCII (i.e 0x00 to 0x7F), except* `"`, `\` *and IsolatedCR*

A non-raw *byte string literal* is a sequence of ASCII characters and *escapes*, preceded by the characters `U+0062` (`b`) and `U+0022` (double-quote), and followed by the character `U+0022`. If the character `U+0022` is present within the literal, it must be *escaped* by a preceding `U+005C` (`\`) character. Alternatively, a byte string literal can be a *raw byte string literal*, defined below.

Line-breaks, represented by the character `U+000A` (LF), are allowed in byte string literals. When an unescaped `U+005C` character (`\`) occurs immediately before a line break, the line break does not appear in the string represented by the token. See String continuation escapes for details. The character `U+000D` (CR) may not appear in a byte string literal other than as part of such a string continuation escape.

Some additional *escapes* are available in either byte or non-raw byte string literals. An escape starts with a `U+005C` (`\`) and continues with one of the following forms:

- A *byte escape* escape starts with `U+0078` (`x`) and is followed by exactly two *hex digits*. It denotes the byte equal to the provided hex value.
- A *whitespace escape* is one of the characters `U+006E` (`n`), `U+0072` (`r`), or `U+0074` (`t`), denoting the bytes values `0x0A` (ASCII LF), `0x0D` (ASCII CR) or `0x09` (ASCII HT) respectively.
- The *null escape* is the character `U+0030` (`0`) and denotes the byte value `0x00` (ASCII NUL).
- The *backslash escape* is the character `U+005C` (`\`) which must be escaped in order to denote its ASCII encoding `0x5C` .

Raw byte string literals

Lexer

RAW_BYTE_STRING_LITERAL :
  `br` RAW_BYTE_STRING_CONTENT SUFFIX?

RAW_BYTE_STRING_CONTENT :
   `"` ASCII_FOR_RAW* (non-greedy) `"`
  | `#` RAW_BYTE_STRING_CONTENT `#`

ASCII_FOR_RAW :
  any ASCII (i.e. 0x00 to 0x7F) except IsolatedCR

Raw byte string literals do not process any escapes. They start with the character `U+0062` (`b`), followed by `U+0072` (`r`), followed by fewer than 256 of the character `U+0023` (`#`), and a `U+0022` (double-quote) character.

The *raw string body* can contain any sequence of ASCII characters other than `U+000D` (CR). It is terminated only by another `U+0022` (double-quote) character, followed by the same number of `U+0023` (`#`) characters that preceded the opening `U+0022` (double-quote) character. A raw byte string literal can not contain any non-ASCII byte.

All characters contained in the raw string body represent their ASCII encoding, the characters `U+0022` (double-quote) (except when followed by at least as many `U+0023` (`#`) characters as were used to start the raw string literal) or `U+005C` (`\`) do not have any special meaning.

Examples for byte string literals:

```
b"foo"; br"foo";                     // foo
b"\"foo\""; br#""foo""#;             // "foo"

b"foo #\"# bar";
br##"foo #"# bar"##;                 // foo #"# bar

b"\x52"; b"R"; br"R";                // R
b"\\x52"; br"\x52";                  // \x52
```

C string and raw C string literals

C string literals

> **Lexer**
>
> C_STRING_LITERAL :
> `c"` (
> ~[`"` `\` *IsolatedCR NUL*]
> | BYTE_ESCAPE *except* `\0` *or* `\x00`
> | UNICODE_ESCAPE *except* `\u{0}`, `\u{00}`, …, `\u{000000}`
> | STRING_CONTINUE
>)* `"` SUFFIX?

A *C string literal* is a sequence of Unicode characters and *escapes*, preceded by the characters `U+0063` (`c`) and `U+0022` (double-quote), and followed by the character `U+0022`. If the character `U+0022` is present within the literal, it must be *escaped* by a preceding `U+005C` (`\`) character. Alternatively, a C string literal can be a *raw C string literal*, defined below.

C strings are implicitly terminated by byte `0x00`, so the C string literal `c""` is equivalent to manually constructing a `&CStr` from the byte string literal `b"\x00"`. Other than the implicit terminator, byte `0x00` is not permitted within a C string.

Line-breaks, represented by the character `U+000A` (LF), are allowed in C string literals. When an unescaped `U+005C` character (`\`) occurs immediately before a line break, the line break does not appear in the string represented by the token. See String continuation escapes for details. The character `U+000D` (CR) may not appear in a C string literal other than as part of such a string continuation escape.

Some additional *escapes* are available in non-raw C string literals. An escape starts with a `U+005C` (`\`) and continues with one of the following forms:

- A *byte escape* escape starts with `U+0078` (`x`) and is followed by exactly two *hex digits*. It denotes the byte equal to the provided hex value.
- A *24-bit code point escape* starts with `U+0075` (`u`) and is followed by up to six *hex digits* surrounded by braces `U+007B` (`{`) and `U+007D` (`}`). It denotes the Unicode code point equal to the provided hex value, encoded as UTF-8.
- A *whitespace escape* is one of the characters `U+006E` (`n`), `U+0072` (`r`), or `U+0074` (`t`), denoting the bytes values `0x0A` (ASCII LF), `0x0D` (ASCII CR) or `0x09` (ASCII HT) respectively.
- The *backslash escape* is the character `U+005C` (`\`) which must be escaped in order to denote its ASCII encoding `0x5C` .

A C string represents bytes with no defined encoding, but a C string literal may contain Unicode characters above `U+007F` . Such characters will be replaced with the bytes of that character's UTF-8 representation.

The following C string literals are equivalent:

```
c"æ";         // LATIN SMALL LETTER AE (U+00E6)
c"\u{00E6}";
c"\xC3\xA6";
```

> **Edition Differences**: C string literals are accepted in the 2021 edition or later. In earlier additions the token `c""` is lexed as `c ""` .

Raw C string literals

> **Lexer**
>
> RAW_C_STRING_LITERAL :
> `cr` RAW_C_STRING_CONTENT SUFFIX?
>
> RAW_C_STRING_CONTENT :
> `"` (~ *IsolatedCR NUL*)* (non-greedy) `"`
> | `#` RAW_C_STRING_CONTENT `#`

Raw C string literals do not process any escapes. They start with the character `U+0063` (`c`), followed by `U+0072` (`r`), followed by fewer than 256 of the character `U+0023` (`#`), and a `U+0022` (double-quote) character.

The *raw C string body* can contain any sequence of Unicode characters other than `U+0000` (NUL) and `U+000D` (CR). It is terminated only by another `U+0022` (double-quote) character, followed by the same number of `U+0023` (`#`) characters that preceded the opening `U+0022` (double-quote) character.

All characters contained in the raw C string body represent themselves in UTF-8 encoding. The characters `U+0022` (double-quote) (except when followed by at least as many `U+0023` (`#`) characters as were used to start the raw C string literal) or `U+005C` (`\`) do not have any special meaning.

> **Edition Differences**: Raw C string literals are accepted in the 2021 edition or later. In earlier additions the token `cr""` is lexed as `cr ""`, and `cr#""#` is lexed as `cr #""#` (which is non-grammatical).

Examples for C string and raw C string literals

```
c"foo"; cr"foo";                     // foo
c"\"foo\""; cr#""foo""#;             // "foo"

c"foo #\"# bar";
cr##"foo #"# bar"##;                 // foo #"# bar

c"\x52"; c"R"; cr"R";                // R
c"\\x52"; cr"\x52";                  // \x52
```

Number literals

A *number literal* is either an *integer literal* or a *floating-point literal*. The grammar for recognizing the two kinds of literals is mixed.

Integer literals

> **Lexer**
>
> INTEGER_LITERAL :
>
> (DEC_LITERAL | BIN_LITERAL | OCT_LITERAL | HEX_LITERAL) SUFFIX_NO_E?

DEC_LITERAL :
  DEC_DIGIT (DEC_DIGIT | _)*

BIN_LITERAL :
  `0b` (BIN_DIGIT | _)* BIN_DIGIT (BIN_DIGIT | _)*

OCT_LITERAL :
  `0o` (OCT_DIGIT | _)* OCT_DIGIT (OCT_DIGIT | _)*

HEX_LITERAL :
  `0x` (HEX_DIGIT | _)* HEX_DIGIT (HEX_DIGIT | _)*

BIN_DIGIT : [`0` - `1`]

OCT_DIGIT : [`0` - `7`]

DEC_DIGIT : [`0` - `9`]

HEX_DIGIT : [`0` - `9` `a` - `f` `A` - `F`]

An *integer literal* has one of four forms:

- A *decimal literal* starts with a *decimal digit* and continues with any mixture of *decimal digits* and *underscores*.
- A *hex literal* starts with the character sequence `U+0030` `U+0078` (`0x`) and continues as any mixture (with at least one digit) of hex digits and underscores.
- An *octal literal* starts with the character sequence `U+0030` `U+006F` (`0o`) and continues as any mixture (with at least one digit) of octal digits and underscores.
- A *binary literal* starts with the character sequence `U+0030` `U+0062` (`0b`) and continues as any mixture (with at least one digit) of binary digits and underscores.

Like any literal, an integer literal may be followed (immediately, without any spaces) by a suffix as described above. The suffix may not begin with `e` or `E` , as that would be interpreted as the exponent of a floating-point literal. See Integer literal expressions for the effect of these suffixes.

Examples of integer literals which are accepted as literal expressions:

```
123;
123i32;
123u32;
123_u32;

0xff;
0xff_u8;
0x01_f32; // integer 7986, not floating-point 1.0
0x01_e3;  // integer 483, not floating-point 1000.0

0o70;
0o70_i16;

0b1111_1111_1001_0000;
0b1111_1111_1001_0000i64;
0b________1;

0usize;

// These are too big for their type, but are accepted as literal expressions.
128_i8;
256_u8;

// This is an integer literal, accepted as a floating-point literal expression.
5f32;
```

Note that `-1i8`, for example, is analyzed as two tokens: `-` followed by `1i8`.

Examples of integer literals which are not accepted as literal expressions:

```
0invalidSuffix;
123AFB43;
0b010a;
0xAB_CD_EF_GH;
0b1111_f32;
```

Tuple index

Lexer

TUPLE_INDEX:
  INTEGER_LITERAL

A tuple index is used to refer to the fields of tuples, tuple structs, and tuple variants.

Tuple indices are compared with the literal token directly. Tuple indices start with `0` and each successive index increments the value by `1` as a decimal value. Thus, only decimal values will match, and the value must not have any extra `0` prefix characters.

```
let example = ("dog", "cat", "horse");
let dog = example.0;
let cat = example.1;
// The following examples are invalid.
let cat = example.01;  // ERROR no field named `01`
let horse = example.0b10;  // ERROR no field named `0b10`
```

> **Note**: Tuple indices may include certain suffixes, but this is not intended to be valid, and may be removed in a future version. See https://github.com/rust-lang/rust/issues/60210 for more information.

Floating-point literals

> **Lexer**
>
> FLOAT_LITERAL :
>   DEC_LITERAL `.` *(not immediately followed by `.`, `_` or an XID_Start character)*
> | DEC_LITERAL `.` DEC_LITERAL SUFFIX_NO_E?
> | DEC_LITERAL (`.` DEC_LITERAL)? FLOAT_EXPONENT SUFFIX?
>
> FLOAT_EXPONENT :
> (`e` | `E`) (`+` | `-`)? (DEC_DIGIT | `_`)* DEC_DIGIT (DEC_DIGIT | `_`)*

A *floating-point literal* has one of two forms:

- A *decimal literal* followed by a period character `U+002E` (`.`). This is optionally followed by another decimal literal, with an optional *exponent*.
- A single *decimal literal* followed by an *exponent*.

Like integer literals, a floating-point literal may be followed by a suffix, so long as the pre-suffix part does not end with `U+002E` (`.`). The suffix may not begin with `e` or `E` if the literal does not include an exponent. See Floating-point literal expressions for the effect of these suffixes.

Examples of floating-point literals which are accepted as literal expressions:

```
123.0f64;
0.1f64;
0.1f32;
12E+99_f64;
let x: f64 = 2.;
```

This last example is different because it is not possible to use the suffix syntax with a floating point literal ending in a period. `2.f64` would attempt to call a method named `f64` on `2`.

Note that `-1.0`, for example, is analyzed as two tokens: `-` followed by `1.0`.

Examples of floating-point literals which are not accepted as literal expressions:

```
2.0f80;
2e5f80;
2e5e6;
2.0e5e6;
1.3e10u64;
```

Reserved forms similar to number literals

Lexer

RESERVED_NUMBER :
  BIN_LITERAL [`2`-`9`]
  | OCT_LITERAL [`8`-`9`]
  | (BIN_LITERAL | OCT_LITERAL | HEX_LITERAL) `.`
    (not immediately followed by `.`, `_` or an XID_Start character)
  | (BIN_LITERAL | OCT_LITERAL) (`e` | `E`)
  | `0b` `_`* *end of input or not BIN_DIGIT*
  | `0o` `_`* *end of input or not OCT_DIGIT*
  | `0x` `_`* *end of input or not HEX_DIGIT*
  | DEC_LITERAL (. DEC_LITERAL)? (`e` | `E`) (`+` | `-`)? *end of input or not DEC_DIGIT*

The following lexical forms similar to number literals are *reserved forms*. Due to the possible ambiguity these raise, they are rejected by the tokenizer instead of being interpreted as separate tokens.

- An unsuffixed binary or octal literal followed, without intervening whitespace, by a decimal digit out of the range for its radix.

- An unsuffixed binary, octal, or hexadecimal literal followed, without intervening whitespace, by a period character (with the same restrictions on what follows the period as for floating-point literals).
- An unsuffixed binary or octal literal followed, without intervening whitespace, by the character `e` or `E`.
- Input which begins with one of the radix prefixes but is not a valid binary, octal, or hexadecimal literal (because it contains no digits).
- Input which has the form of a floating-point literal with no digits in the exponent.

Examples of reserved forms:

```
0b0102;  // this is not `0b010` followed by `2`
0o1279;  // this is not `0o127` followed by `9`
0x80.0;  // this is not `0x80` followed by `.` and `0`
0b101e;  // this is not a suffixed literal, or `0b101` followed by `e`
0b;      // this is not an integer literal, or `0` followed by `b`
0b_;     // this is not an integer literal, or `0` followed by `b_`
2e;      // this is not a floating-point literal, or `2` followed by `e`
2.0e;    // this is not a floating-point literal, or `2.0` followed by `e`
2em;     // this is not a suffixed literal, or `2` followed by `em`
2.0em;   // this is not a suffixed literal, or `2.0` followed by `em`
```

Lifetimes and loop labels

Lexer

LIFETIME_TOKEN :
   `'` IDENTIFIER_OR_KEYWORD *(not immediately followed by `'`)*
 | `'_` *(not immediately followed by `'`)*

LIFETIME_OR_LABEL :
   `'` NON_KEYWORD_IDENTIFIER *(not immediately followed by `'`)*

Lifetime parameters and loop labels use LIFETIME_OR_LABEL tokens. Any LIFETIME_TOKEN will be accepted by the lexer, and for example, can be used in macros.

Punctuation

Punctuation symbol tokens are listed here for completeness. Their individual usages and meanings are defined in the linked pages.

Symbol	Name	Usage
`+`	Plus	Addition, Trait Bounds, Macro Kleene Matcher
`-`	Minus	Subtraction, Negation
`*`	Star	Multiplication, Dereference, Raw Pointers, Macro Kleene Matcher, Use wildcards
`/`	Slash	Division
`%`	Percent	Remainder
`^`	Caret	Bitwise and Logical XOR
`!`	Not	Bitwise and Logical NOT, Macro Calls, Inner Attributes, Never Type, Negative impls
`&`	And	Bitwise and Logical AND, Borrow, References, Reference patterns
`|`	Or	Bitwise and Logical OR, Closures, Patterns in match, if let, and while let
`&&`	AndAnd	Lazy AND, Borrow, References, Reference patterns
`||`	OrOr	Lazy OR, Closures
`<<`	Shl	Shift Left, Nested Generics
`>>`	Shr	Shift Right, Nested Generics
`+=`	PlusEq	Addition assignment
`-=`	MinusEq	Subtraction assignment
`*=`	StarEq	Multiplication assignment
`/=`	SlashEq	Division assignment
`%=`	PercentEq	Remainder assignment
`^=`	CaretEq	Bitwise XOR assignment
`&=`	AndEq	Bitwise And assignment
`|=`	OrEq	Bitwise Or assignment
`<<=`	ShlEq	Shift Left assignment
`>>=`	ShrEq	Shift Right assignment, Nested Generics
`=`	Eq	Assignment, Attributes, Various type definitions

Symbol	Name	Usage
==	EqEq	Equal
!=	Ne	Not Equal
>	Gt	Greater than, Generics, Paths
<	Lt	Less than, Generics, Paths
>=	Ge	Greater than or equal to, Generics
<=	Le	Less than or equal to
@	At	Subpattern binding
_	Underscore	Wildcard patterns, Inferred types, Unnamed items in constants, extern crates, use declarations, and destructuring assignment
.	Dot	Field access, Tuple index
..	DotDot	Range, Struct expressions, Patterns, Range Patterns
...	DotDotDot	Variadic functions, Range patterns
..=	DotDotEq	Inclusive Range, Range patterns
,	Comma	Various separators
;	Semi	Terminator for various items and statements, Array types
:	Colon	Various separators
::	PathSep	Path separator
->	RArrow	Function return type, Closure return type, Function pointer type
=>	FatArrow	Match arms, Macros
<-	LArrow	The left arrow symbol has been unused since before Rust 1.0, but it is still treated as a single token
#	Pound	Attributes
$	Dollar	Macros
?	Question	Question mark operator, Questionably sized, Macro Kleene Matcher
~	Tilde	The tilde operator has been unused since before Rust 1.0, but its token may still be used

Delimiters

Bracket punctuation is used in various parts of the grammar. An open bracket must always be paired with a close bracket. Brackets and the tokens within them are referred to as "token trees" in macros. The three types of brackets are:

Bracket	Type
`{` `}`	Curly braces
`[` `]`	Square brackets
`(` `)`	Parentheses

Reserved prefixes

Lexer 2021+
RESERVED_TOKEN_DOUBLE_QUOTE : (IDENTIFIER_OR_KEYWORD *Except `b` or `c` or `r` or `br` or `cr`* | `_`) `"`
RESERVED_TOKEN_SINGLE_QUOTE : (IDENTIFIER_OR_KEYWORD *Except `b`* | `_`) `'`
RESERVED_TOKEN_POUND : (IDENTIFIER_OR_KEYWORD *Except `r` or `br` or `cr`* | `_`) `#`

Some lexical forms known as *reserved prefixes* are reserved for future use.

Source input which would otherwise be lexically interpreted as a non-raw identifier (or a keyword or `_`) which is immediately followed by a `#`, `'`, or `"` character (without intervening whitespace) is identified as a reserved prefix.

Note that raw identifiers, raw string literals, and raw byte string literals may contain a `#` character but are not interpreted as containing a reserved prefix.

Similarly the `r`, `b`, `br`, `c`, and `cr` prefixes used in raw string literals, byte literals, byte string literals, raw byte string literals, C string literals, and raw C string literals are not interpreted as reserved prefixes.

Edition Differences: Starting with the 2021 edition, reserved prefixes are reported as an error by the lexer (in particular, they cannot be passed to macros).

Before the 2021 edition, reserved prefixes are accepted by the lexer and interpreted as multiple tokens (for example, one token for the identifier or keyword, followed by a `#` token).

Examples accepted in all editions:

```
macro_rules! lexes {($($_:tt)*) => {}}
lexes!{a #foo}
lexes!{continue 'foo}
lexes!{match "..." {}}
lexes!{r#let#foo}         // three tokens: r#let # foo
```

Examples accepted before the 2021 edition but rejected later:

```
macro_rules! lexes {($($_:tt)*) => {}}
lexes!{a#foo}
lexes!{continue'foo}
lexes!{match"..." {}}
```

Macros

The functionality and syntax of Rust can be extended with custom definitions called macros. They are given names, and invoked through a consistent syntax: `some_extension!(...)`.

There are two ways to define new macros:

- Macros by Example define new syntax in a higher-level, declarative way.
- Procedural Macros define function-like macros, custom derives, and custom attributes using functions that operate on input tokens.

Macro Invocation

Syntax

MacroInvocation :
  SimplePath `!` *DelimTokenTree*

DelimTokenTree :
    `(` *TokenTree** `)`
  | `[` *TokenTree** `]`
  | `{` *TokenTree** `}`

TokenTree :
  *Token*_{*except delimiters*} | *DelimTokenTree*

MacroInvocationSemi :
    SimplePath `!` `(` *TokenTree** `)` `;`
  | *SimplePath* `!` `[` *TokenTree** `]` `;`
  | *SimplePath* `!` `{` *TokenTree** `}`

A macro invocation expands a macro at compile time and replaces the invocation with the result of the macro. Macros may be invoked in the following situations:

- Expressions and statements
- Patterns
- Types
- Items including associated items

- `macro_rules` transcribers
- External blocks

When used as an item or a statement, the *MacroInvocationSemi* form is used where a semicolon is required at the end when not using curly braces. Visibility qualifiers are never allowed before a macro invocation or `macro_rules` definition.

```
// Used as an expression.
let x = vec![1,2,3];

// Used as a statement.
println!("Hello!");

// Used in a pattern.
macro_rules! pat {
    ($i:ident) => (Some($i))
}

if let pat!(x) = Some(1) {
    assert_eq!(x, 1);
}

// Used in a type.
macro_rules! Tuple {
    { $A:ty, $B:ty } => { ($A, $B) };
}

type N2 = Tuple!(i32, i32);

// Used as an item.
thread_local!(static FOO: RefCell<u32> = RefCell::new(1));

// Used as an associated item.
macro_rules! const_maker {
    ($t:ty, $v:tt) => { const CONST: $t = $v; };
}
trait T {
    const_maker!{i32, 7}
}

// Macro calls within macros.
macro_rules! example {
    () => { println!("Macro call in a macro!") };
}
// Outer macro `example` is expanded, then inner macro `println` is expanded.
example!();
```

Macros By Example

> **Syntax**
>
> *MacroRulesDefinition* :
>   `macro_rules` `!` IDENTIFIER *MacroRulesDef*
>
> *MacroRulesDef* :
>   `(` *MacroRules* `)` `;`
> | `[` *MacroRules* `]` `;`
> | `{` *MacroRules* `}`
>
> *MacroRules* :
>   *MacroRule* (`;` *MacroRule*)* `;`?
>
> *MacroRule* :
>   *MacroMatcher* `=>` *MacroTranscriber*
>
> *MacroMatcher* :
>   `(` *MacroMatch** `)`
> | `[` *MacroMatch** `]`
> | `{` *MacroMatch** `}`
>
> *MacroMatch* :
>   *Token*except `$` and *delimiters*
> | *MacroMatcher*
> | `$` (IDENTIFIER_OR_KEYWORD except `crate` | RAW_IDENTIFIER | `_`) `:`
> *MacroFragSpec*
> | `$` `(` *MacroMatch*+ `)` *MacroRepSep*? *MacroRepOp*
>
> *MacroFragSpec* :
>   `block` | `expr` | `ident` | `item` | `lifetime` | `literal`
> | `meta` | `pat` | `pat_param` | `path` | `stmt` | `tt` | `ty` | `vis`
>
> *MacroRepSep* :
>   *Token*except *delimiters* and *MacroRepOp*
>
> *MacroRepOp* :
>   `*` | `+` | `?`
>
> *MacroTranscriber* :

DelimTokenTree

`macro_rules` allows users to define syntax extension in a declarative way. We call such extensions "macros by example" or simply "macros".

Each macro by example has a name, and one or more *rules*. Each rule has two parts: a *matcher*, describing the syntax that it matches, and a *transcriber*, describing the syntax that will replace a successfully matched invocation. Both the matcher and the transcriber must be surrounded by delimiters. Macros can expand to expressions, statements, items (including traits, impls, and foreign items), types, or patterns.

Transcribing

When a macro is invoked, the macro expander looks up macro invocations by name, and tries each macro rule in turn. It transcribes the first successful match; if this results in an error, then future matches are not tried. When matching, no lookahead is performed; if the compiler cannot unambiguously determine how to parse the macro invocation one token at a time, then it is an error. In the following example, the compiler does not look ahead past the identifier to see if the following token is a `)`, even though that would allow it to parse the invocation unambiguously:

```
macro_rules! ambiguity {
    ($($i:ident)* $j:ident) => { };
}

ambiguity!(error); // Error: local ambiguity
```

In both the matcher and the transcriber, the `$` token is used to invoke special behaviours from the macro engine (described below in Metavariables and Repetitions). Tokens that aren't part of such an invocation are matched and transcribed literally, with one exception. The exception is that the outer delimiters for the matcher will match any pair of delimiters. Thus, for instance, the matcher `(())` will match `{()}` but not `{{}}`. The character `$` cannot be matched or transcribed literally.

Forwarding a matched fragment

When forwarding a matched fragment to another macro-by-example, matchers in the second macro will see an opaque AST of the fragment type. The second macro can't use literal tokens to match the fragments in the matcher, only a fragment specifier of the same type. The `ident`, `lifetime`, and `tt` fragment types are an exception, and *can* be matched by literal tokens. The following illustrates this restriction:

```
macro_rules! foo {
    ($l:expr) => { bar!($l); }
// ERROR:               ^^ no rules expected this token in macro call
}

macro_rules! bar {
    (3) => {}
}

foo!(3);
```

The following illustrates how tokens can be directly matched after matching a `tt` fragment:

```
// compiles OK
macro_rules! foo {
    ($l:tt) => { bar!($l); }
}

macro_rules! bar {
    (3) => {}
}

foo!(3);
```

Metavariables

In the matcher, `$` *name* `:` *fragment-specifier* matches a Rust syntax fragment of the kind specified and binds it to the metavariable `$`*name*. Valid fragment specifiers are:

- `item`: an *Item*
- `block`: a *BlockExpression*
- `stmt`: a *Statement* without the trailing semicolon (except for item statements that require semicolons)
- `pat_param`: a *PatternNoTopAlt*
- `pat`: at least any *PatternNoTopAlt*, and possibly more depending on edition
- `expr`: an *Expression*
- `ty`: a *Type*
- `ident`: an IDENTIFIER_OR_KEYWORD or RAW_IDENTIFIER

- `path` : a *TypePath* style path
- `tt` : a *TokenTree* (a single token or tokens in matching delimiters `()` , `[]` , or `{}`)
- `meta` : an *Attr*, the contents of an attribute
- `lifetime` : a LIFETIME_TOKEN
- `vis` : a possibly empty *Visibility* qualifier
- `literal` : matches `-`?*LiteralExpression*

In the transcriber, metavariables are referred to simply by `$` *name*, since the fragment kind is specified in the matcher. Metavariables are replaced with the syntax element that matched them. The keyword metavariable `$crate` can be used to refer to the current crate; see Hygiene below. Metavariables can be transcribed more than once or not at all.

For reasons of backwards compatibility, though `_` is also an expression, a standalone underscore is not matched by the `expr` fragment specifier. However, `_` is matched by the `expr` fragment specifier when it appears as a subexpression.

> **Edition Differences**: Starting with the 2021 edition, `pat` fragment-specifiers match top-level or-patterns (that is, they accept *Pattern*).
>
> Before the 2021 edition, they match exactly the same fragments as `pat_param` (that is, they accept *PatternNoTopAlt*).
>
> The relevant edition is the one in effect for the `macro_rules!` definition.

Repetitions

In both the matcher and transcriber, repetitions are indicated by placing the tokens to be repeated inside `$(` … `)` , followed by a repetition operator, optionally with a separator token between. The separator token can be any token other than a delimiter or one of the repetition operators, but `;` and `,` are the most common. For instance, `$( $i:ident ),*` represents any number of identifiers separated by commas. Nested repetitions are permitted.

The repetition operators are:

- `*` — indicates any number of repetitions.
- `+` — indicates any number but at least one.
- `?` — indicates an optional fragment with zero or one occurrence.

Since `?` represents at most one occurrence, it cannot be used with a separator.

The repeated fragment both matches and transcribes to the specified number of the fragment, separated by the separator token. Metavariables are matched to every repetition of their corresponding fragment. For instance, the `$( $i:ident ),*` example above matches `$i` to all of the identifiers in the list.

During transcription, additional restrictions apply to repetitions so that the compiler knows how to expand them properly:

1. A metavariable must appear in exactly the same number, kind, and nesting order of repetitions in the transcriber as it did in the matcher. So for the matcher `$( $i:ident ),*`, the transcribers `=> { $i }`, `=> { $( $( $i)* )* }`, and `=> { $( $i )+ }` are all illegal, but `=> { $( $i );* }` is correct and replaces a comma-separated list of identifiers with a semicolon-separated list.
2. Each repetition in the transcriber must contain at least one metavariable to decide how many times to expand it. If multiple metavariables appear in the same repetition, they must be bound to the same number of fragments. For instance, `( $( $i:ident ),* ; $( $j:ident ),* ) => (( $( ($i,$j) ),* ))` must bind the same number of `$i` fragments as `$j` fragments. This means that invoking the macro with `(a, b, c; d, e, f)` is legal and expands to `((a,d), (b,e), (c,f))`, but `(a, b, c; d, e)` is illegal because it does not have the same number. This requirement applies to every layer of nested repetitions.

Scoping, Exporting, and Importing

For historical reasons, the scoping of macros by example does not work entirely like items. Macros have two forms of scope: textual scope, and path-based scope. Textual scope is based on the order that things appear in source files, or even across multiple files, and is the default scoping. It is explained further below. Path-based scope works exactly the same way that item scoping does. The scoping, exporting, and importing of macros is controlled largely by attributes.

When a macro is invoked by an unqualified identifier (not part of a multi-part path), it is first looked up in textual scoping. If this does not yield any results, then it is looked up in path-based scoping. If the macro's name is qualified with a path, then it is only looked up in path-based scoping.

```
use lazy_static::lazy_static; // Path-based import.

macro_rules! lazy_static { // Textual definition.
    (lazy) => {};
}

lazy_static!{lazy} // Textual lookup finds our macro first.
self::lazy_static!{} // Path-based lookup ignores our macro, finds imported
one.
```

Textual Scope

Textual scope is based largely on the order that things appear in source files, and works similarly to the scope of local variables declared with `let` except it also applies at the module level. When `macro_rules!` is used to define a macro, the macro enters the scope after the definition (note that it can still be used recursively, since names are looked up from the invocation site), up until its surrounding scope, typically a module, is closed. This can enter child modules and even span across multiple files:

```
//// src/lib.rs
mod has_macro {
    // m!{} // Error: m is not in scope.

    macro_rules! m {
        () => {};
    }
    m!{} // OK: appears after declaration of m.

    mod uses_macro;
}

// m!{} // Error: m is not in scope.

//// src/has_macro/uses_macro.rs

m!{} // OK: appears after declaration of m in src/lib.rs
```

It is not an error to define a macro multiple times; the most recent declaration will shadow the previous one unless it has gone out of scope.

```
macro_rules! m {
    (1) => {};
}

m!(1);

mod inner {
    m!(1);

    macro_rules! m {
        (2) => {};
    }
    // m!(1); // Error: no rule matches '1'
    m!(2);

    macro_rules! m {
        (3) => {};
    }
    m!(3);
}

m!(1);
```

Macros can be declared and used locally inside functions as well, and work similarly:

```
fn foo() {
    // m!(); // Error: m is not in scope.
    macro_rules! m {
        () => {};
    }
    m!();
}


// m!(); // Error: m is not in scope.
```

The `macro_use` attribute

The *`macro_use` attribute* has two purposes. First, it can be used to make a module's macro scope not end when the module is closed, by applying it to a module:

```
#[macro_use]
mod inner {
    macro_rules! m {
        () => {};
    }
}

m!();
```

Second, it can be used to import macros from another crate, by attaching it to an `extern crate` declaration appearing in the crate's root module. Macros imported this way are imported into the `macro_use` prelude, not textually, which means that they can be shadowed by any other name. While macros imported by `#[macro_use]` can be used before the import statement, in case of a conflict, the last macro imported wins. Optionally, a list of macros to import can be specified using the *MetaListIdents* syntax; this is not supported when `#[macro_use]` is applied to a module.

```
#[macro_use(lazy_static)] // Or #[macro_use] to import all macros.
extern crate lazy_static;

lazy_static!{}
// self::lazy_static!{} // Error: lazy_static is not defined in `self`
```

Macros to be imported with `#[macro_use]` must be exported with `#[macro_export]`, which is described below.

Path-Based Scope

By default, a macro has no path-based scope. However, if it has the `#[macro_export]` attribute, then it is declared in the crate root scope and can be referred to normally as such:

```
self::m!();
m!(); // OK: Path-based lookup finds m in the current module.

mod inner {
    super::m!();
    crate::m!();
}

mod mac {
    #[macro_export]
    macro_rules! m {
        () => {};
    }
}
```

Macros labeled with `#[macro_export]` are always `pub` and can be referred to by other crates, either by path or by `#[macro_use]` as described above.

Hygiene

By default, all identifiers referred to in a macro are expanded as-is, and are looked up at the macro's invocation site. This can lead to issues if a macro refers to an item or macro which isn't in scope at the invocation site. To alleviate this, the `$crate` metavariable can be used at the start of a path to force lookup to occur inside the crate defining the macro.

```
//// Definitions in the `helper_macro` crate.
#[macro_export]
macro_rules! helped {
    // () => { helper!() } // This might lead to an error due to 'helper' not
being in scope.
    () => { $crate::helper!() }
}

#[macro_export]
macro_rules! helper {
    () => { () }
}

//// Usage in another crate.
// Note that `helper_macro::helper` is not imported!
use helper_macro::helped;

fn unit() {
    helped!();
}
```

Note that, because `$crate` refers to the current crate, it must be used with a fully qualified module path when referring to non-macro items:

```
pub mod inner {
    #[macro_export]
    macro_rules! call_foo {
        () => { $crate::inner::foo() };
    }

    pub fn foo() {}
}
```

Additionally, even though `$crate` allows a macro to refer to items within its own crate when expanding, its use has no effect on visibility. An item or macro referred to must still be visible from the invocation site. In the following example, any attempt to invoke `call_foo!()` from outside its crate will fail because `foo()` is not public.

```
#[macro_export]
macro_rules! call_foo {
    () => { $crate::foo() };
}

fn foo() {}
```

> **Version & Edition Differences**: Prior to Rust 1.30, `$crate` and `local_inner_macros` (below) were unsupported. They were added alongside path-based imports of macros (described above), to ensure that helper macros did not need to be manually imported by users of a macro-exporting crate. Crates written for earlier versions of Rust that use helper macros need to be modified to use `$crate` or `local_inner_macros` to work well with path-based imports.

When a macro is exported, the `#[macro_export]` attribute can have the `local_inner_macros` keyword added to automatically prefix all contained macro invocations with `$crate::`. This is intended primarily as a tool to migrate code written before `$crate` was added to the language to work with Rust 2018's path-based imports of macros. Its use is discouraged in new code.

```
#[macro_export(local_inner_macros)]
macro_rules! helped {
    () => { helper!() } // Automatically converted to $crate::helper!().
}

#[macro_export]
macro_rules! helper {
    () => { () }
}
```

Follow-set Ambiguity Restrictions

The parser used by the macro system is reasonably powerful, but it is limited in order to prevent ambiguity in current or future versions of the language. In particular, in addition to the rule about ambiguous expansions, a nonterminal matched by a metavariable must be followed by a token which has been decided can be safely used after that kind of match.

As an example, a macro matcher like `$i:expr [ , ]` could in theory be accepted in Rust today, since `[,]` cannot be part of a legal expression and therefore the parse would always be unambiguous. However, because `[` can start trailing expressions, `[` is not a character which can safely be ruled out as coming after an expression. If `[,]` were accepted in a later version of Rust, this matcher would become ambiguous or would misparse, breaking working code. Matchers like `$i:expr,` or `$i:expr;` would be legal, however, because `,` and `;` are legal expression separators. The specific rules are:

- `expr` and `stmt` may only be followed by one of: `=>`, `,`, or `;`.
- `pat_param` may only be followed by one of: `=>`, `,`, `=`, `|`, `if`, or `in`.
- `pat` may only be followed by one of: `=>`, `,`, `=`, `if`, or `in`.
- `path` and `ty` may only be followed by one of: `=>`, `,`, `=`, `|`, `;`, `:`, `>`, `>>`, `[`, `{`, `as`, `where`, or a macro variable of `block` fragment specifier.
- `vis` may only be followed by one of: `,`, an identifier other than a non-raw `priv`, any token that can begin a type, or a metavariable with a `ident`, `ty`, or `path` fragment specifier.
- All other fragment specifiers have no restrictions.

Edition Differences: Before the 2021 edition, `pat` may also be followed by `|`.

When repetitions are involved, then the rules apply to every possible number of expansions, taking separators into account. This means:

- If the repetition includes a separator, that separator must be able to follow the contents of the repetition.
- If the repetition can repeat multiple times (`*` or `+`), then the contents must be able to follow themselves.
- The contents of the repetition must be able to follow whatever comes before, and whatever comes after must be able to follow the contents of the repetition.
- If the repetition can match zero times (`*` or `?`), then whatever comes after must be able to follow whatever comes before.

For more detail, see the formal specification.

Procedural Macros

Procedural macros allow creating syntax extensions as execution of a function. Procedural macros come in one of three flavors:

- Function-like macros - `custom!(...)`
- Derive macros - `#[derive(CustomDerive)]`
- Attribute macros - `#[CustomAttribute]`

Procedural macros allow you to run code at compile time that operates over Rust syntax, both consuming and producing Rust syntax. You can sort of think of procedural macros as functions from an AST to another AST.

Procedural macros must be defined in a crate with the crate type of `proc-macro`.

> **Note**: When using Cargo, Procedural macro crates are defined with the `proc-macro` key in your manifest:
>
> ```
> [lib]
> proc-macro = true
> ```

As functions, they must either return syntax, panic, or loop endlessly. Returned syntax either replaces or adds the syntax depending on the kind of procedural macro. Panics are caught by the compiler and are turned into a compiler error. Endless loops are not caught by the compiler which hangs the compiler.

Procedural macros run during compilation, and thus have the same resources that the compiler has. For example, standard input, error, and output are the same that the compiler has access to. Similarly, file access is the same. Because of this, procedural macros have the same security concerns that Cargo's build scripts have.

Procedural macros have two ways of reporting errors. The first is to panic. The second is to emit a `compile_error` macro invocation.

The `proc_macro` crate

Procedural macro crates almost always will link to the compiler-provided `proc_macro` crate. The `proc_macro` crate provides types required for writing procedural macros and facilities to make it easier.

This crate primarily contains a `TokenStream` type. Procedural macros operate over *token streams* instead of AST nodes, which is a far more stable interface over time for both the compiler and for procedural macros to target. A *token stream* is roughly equivalent to `Vec<TokenTree>` where a `TokenTree` can roughly be thought of as lexical token. For example `foo` is an `Ident` token, `.` is a `Punct` token, and `1.2` is a `Literal` token. The `TokenStream` type, unlike `Vec<TokenTree>`, is cheap to clone.

All tokens have an associated `Span`. A `Span` is an opaque value that cannot be modified but can be manufactured. `Span`s represent an extent of source code within a program and are primarily used for error reporting. While you cannot modify a `Span` itself, you can always change the `Span` *associated* with any token, such as through getting a `Span` from another token.

Procedural macro hygiene

Procedural macros are *unhygienic*. This means they behave as if the output token stream was simply written inline to the code it's next to. This means that it's affected by external items and also affects external imports.

Macro authors need to be careful to ensure their macros work in as many contexts as possible given this limitation. This often includes using absolute paths to items in libraries (for example, `::std::option::Option` instead of `Option`) or by ensuring that generated functions have names that are unlikely to clash with other functions (like `__internal_foo` instead of `foo`).

Function-like procedural macros

Function-like procedural macros are procedural macros that are invoked using the macro invocation operator (`!`).

These macros are defined by a public function with the `proc_macro` attribute and a signature of `(TokenStream) -> TokenStream`. The input `TokenStream` is what is inside the delimiters of the macro invocation and the output `TokenStream` replaces the entire macro invocation.

For example, the following macro definition ignores its input and outputs a function `answer` into its scope.

```
extern crate proc_macro;
use proc_macro::TokenStream;

#[proc_macro]
pub fn make_answer(_item: TokenStream) -> TokenStream {
    "fn answer() -> u32 { 42 }".parse().unwrap()
}
```

And then we use it in a binary crate to print "42" to standard output.

```
extern crate proc_macro_examples;
use proc_macro_examples::make_answer;

make_answer!();

fn main() {
    println!("{}", answer());
}
```

Function-like procedural macros may be invoked in any macro invocation position, which includes statements, expressions, patterns, type expressions, item positions, including items in `extern` blocks, inherent and trait implementations, and trait definitions.

Derive macros

Derive macros define new inputs for the `derive` attribute. These macros can create new items given the token stream of a struct, enum, or union. They can also define derive macro helper attributes.

Custom derive macros are defined by a public function with the `proc_macro_derive` attribute and a signature of `(TokenStream) -> TokenStream`.

The input `TokenStream` is the token stream of the item that has the `derive` attribute on it. The output `TokenStream` must be a set of items that are then appended to the module or block that the item from the input `TokenStream` is in.

The following is an example of a derive macro. Instead of doing anything useful with its input, it just appends a function `answer`.

```rust
extern crate proc_macro;
use proc_macro::TokenStream;

#[proc_macro_derive(AnswerFn)]
pub fn derive_answer_fn(_item: TokenStream) -> TokenStream {
    "fn answer() -> u32 { 42 }".parse().unwrap()
}
```

And then using said derive macro:

```rust
extern crate proc_macro_examples;
use proc_macro_examples::AnswerFn;

#[derive(AnswerFn)]
struct Struct;

fn main() {
    assert_eq!(42, answer());
}
```

Derive macro helper attributes

Derive macros can add additional attributes into the scope of the item they are on. Said attributes are called *derive macro helper attributes*. These attributes are inert, and their only purpose is to be fed into the derive macro that defined them. That said, they can be seen by all macros.

The way to define helper attributes is to put an `attributes` key in the `proc_macro_derive` macro with a comma separated list of identifiers that are the names of the helper attributes.

For example, the following derive macro defines a helper attribute `helper`, but ultimately doesn't do anything with it.

```rust
#[proc_macro_derive(HelperAttr, attributes(helper))]
pub fn derive_helper_attr(_item: TokenStream) -> TokenStream {
    TokenStream::new()
}
```

And then usage on the derive macro on a struct:

```rust
#[derive(HelperAttr)]
struct Struct {
    #[helper] field: ()
}
```

Attribute macros

Attribute macros define new outer attributes which can be attached to items, including items in `extern` blocks, inherent and trait implementations, and trait definitions.

Attribute macros are defined by a public function with the `proc_macro_attribute` attribute that has a signature of `(TokenStream, TokenStream) -> TokenStream`. The first `TokenStream` is the delimited token tree following the attribute's name, not including the outer delimiters. If the attribute is written as a bare attribute name, the attribute `TokenStream` is empty. The second `TokenStream` is the rest of the item including other attributes on the item. The returned `TokenStream` replaces the item with an arbitrary number of items.

For example, this attribute macro takes the input stream and returns it as is, effectively being the no-op of attributes.

```
#[proc_macro_attribute]
pub fn return_as_is(_attr: TokenStream, item: TokenStream) -> TokenStream {
    item
}
```

This following example shows the stringified `TokenStream`s that the attribute macros see. The output will show in the output of the compiler. The output is shown in the comments after the function prefixed with "out:".

```
// my-macro/src/lib.rs

#[proc_macro_attribute]
pub fn show_streams(attr: TokenStream, item: TokenStream) -> TokenStream {
    println!("attr: \"{}\"", attr.to_string());
    println!("item: \"{}\"", item.to_string());
    item
}
```

```
// src/lib.rs
extern crate my_macro;

use my_macro::show_streams;

// Example: Basic function
#[show_streams]
fn invoke1() {}
// out: attr: ""
// out: item: "fn invoke1() {}"

// Example: Attribute with input
#[show_streams(bar)]
fn invoke2() {}
// out: attr: "bar"
// out: item: "fn invoke2() {}"

// Example: Multiple tokens in the input
#[show_streams(multiple => tokens)]
fn invoke3() {}
// out: attr: "multiple => tokens"
// out: item: "fn invoke3() {}"

// Example:
#[show_streams { delimiters }]
fn invoke4() {}
// out: attr: "delimiters"
// out: item: "fn invoke4() {}"
```

Declarative macro tokens and procedural macro tokens

Declarative `macro_rules` macros and procedural macros use similar, but different definitions for tokens (or rather `TokenTree` s.)

Token trees in `macro_rules` (corresponding to `tt` matchers) are defined as

- Delimited groups (`(...)` , `{...}` , etc)
- All operators supported by the language, both single-character and multi-character ones (`+` , `+=`).
  - Note that this set doesn't include the single quote `'` .
- Literals (`"string"` , `1` , etc)
  - Note that negation (e.g. `-1`) is never a part of such literal tokens, but a separate operator token.
- Identifiers, including keywords (`ident` , `r#ident` , `fn`)
- Lifetimes (`'ident`)

- Metavariable substitutions in `macro_rules` (e.g. `$my_expr` in `macro_rules! mac { ($my_expr: expr) => { $my_expr } }` after the `mac` 's expansion, which will be considered a single token tree regardless of the passed expression)

Token trees in procedural macros are defined as

- Delimited groups (`(...)` , `{...}` , etc)
- All punctuation characters used in operators supported by the language (`+` , but not `+=`), and also the single quote `'` character (typically used in lifetimes, see below for lifetime splitting and joining behavior)
- Literals (`"string"` , `1` , etc)
  - Negation (e.g. `-1`) is supported as a part of integer and floating point literals.
- Identifiers, including keywords (`ident` , `r#ident` , `fn`)

Mismatches between these two definitions are accounted for when token streams are passed to and from procedural macros.
Note that the conversions below may happen lazily, so they might not happen if the tokens are not actually inspected.

When passed to a proc-macro

- All multi-character operators are broken into single characters.
- Lifetimes are broken into a `'` character and an identifier.
- All metavariable substitutions are represented as their underlying token streams.
  - Such token streams may be wrapped into delimited groups (`Group`) with implicit delimiters (`Delimiter::None`) when it's necessary for preserving parsing priorities.
  - `tt` and `ident` substitutions are never wrapped into such groups and always represented as their underlying token trees.

When emitted from a proc macro

- Punctuation characters are glued into multi-character operators when applicable.
- Single quotes `'` joined with identifiers are glued into lifetimes.
- Negative literals are converted into two tokens (the `-` and the literal) possibly wrapped into a delimited group (`Group`) with implicit delimiters (`Delimiter::None`) when it's necessary for preserving parsing priorities.

Note that neither declarative nor procedural macros support doc comment tokens (e.g. `/// Doc`), so they are always converted to token streams representing their equivalent `#[doc = r"str"]` attributes when passed to macros.

Crates and source files

> **Syntax**
> *Crate* :
>   *InnerAttribute**
>   *Item**

> Note: Although Rust, like any other language, can be implemented by an interpreter as well as a compiler, the only existing implementation is a compiler, and the language has always been designed to be compiled. For these reasons, this section assumes a compiler.

Rust's semantics obey a *phase distinction* between compile-time and run-time.[1] Semantic rules that have a *static interpretation* govern the success or failure of compilation, while semantic rules that have a *dynamic interpretation* govern the behavior of the program at run-time.

The compilation model centers on artifacts called *crates*. Each compilation processes a single crate in source form, and if successful, produces a single crate in binary form: either an executable or some sort of library.[2]

A *crate* is a unit of compilation and linking, as well as versioning, distribution, and runtime loading. A crate contains a *tree* of nested module scopes. The top level of this tree is a module that is anonymous (from the point of view of paths within the module) and any item within a crate has a canonical module path denoting its location within the crate's module tree.

The Rust compiler is always invoked with a single source file as input, and always produces a single output crate. The processing of that source file may result in other source files being loaded as modules. Source files have the extension `.rs`.

A Rust source file describes a module, the name and location of which — in the module tree of the current crate — are defined from outside the source file: either by an explicit *Module* item in a referencing source file, or by the name of the crate itself. Every source file is a module, but not every module needs its own source file: module definitions can be nested within one file.

Each source file contains a sequence of zero or more *Item* definitions, and may optionally begin with any number of attributes that apply to the containing module, most of which influence the behavior of the compiler. The anonymous crate module can have additional attributes that apply to the crate as a whole.

> **Note**: The file's contents may be preceded by a shebang.

```
// Specify the crate name.
#![crate_name = "projx"]

// Specify the type of output artifact.
#![crate_type = "lib"]

// Turn on a warning.
// This can be done in any module, not just the anonymous crate module.
#![warn(non_camel_case_types)]
```

Preludes and `no_std`

This section has been moved to the Preludes chapter.

Main Functions

A crate that contains a `main` function can be compiled to an executable. If a `main` function is present, it must take no arguments, must not declare any trait or lifetime bounds, must not have any where clauses, and its return type must implement the `Termination` trait.

```
fn main() {}
```

```
fn main() -> ! {
    std::process::exit(0);
}
```

```
fn main() -> impl std::process::Termination {
    std::process::ExitCode::SUCCESS
}
```

> **Note**: Types with implementations of `Termination` in the standard library include:
>
> - `()`
> - `!`
> - `Infallible`
> - `ExitCode`
> - `Result<T, E> where T: Termination, E: Debug`

The `no_main` attribute

The *`no_main`* *attribute* may be applied at the crate level to disable emitting the `main` symbol for an executable binary. This is useful when some other object being linked to defines `main`.

The `crate_name` attribute

The *`crate_name`* *attribute* may be applied at the crate level to specify the name of the crate with the *MetaNameValueStr* syntax.

```
#![crate_name = "mycrate"]
```

The crate name must not be empty, and must only contain Unicode alphanumeric or `_` (U+005F) characters.

[1] This distinction would also exist in an interpreter. Static checks like syntactic analysis, type checking, and lints should happen before the program is executed regardless of when it is executed.

[2] A crate is somewhat analogous to an *assembly* in the ECMA-335 CLI model, a *library* in the SML/NJ Compilation Manager, a *unit* in the Owens and Flatt module system, or a *configuration* in Mesa.

Conditional compilation

Syntax

ConfigurationPredicate :
  ConfigurationOption
 | *ConfigurationAll*
 | *ConfigurationAny*
 | *ConfigurationNot*

ConfigurationOption :
 IDENTIFIER (= (STRING_LITERAL | RAW_STRING_LITERAL))?

ConfigurationAll
 `all` `(` *ConfigurationPredicateList*? `)`

ConfigurationAny
 `any` `(` *ConfigurationPredicateList*? `)`

ConfigurationNot
 `not` `(` *ConfigurationPredicate* `)`

ConfigurationPredicateList
 ConfigurationPredicate (`,` *ConfigurationPredicate*)* `,`?

Conditionally compiled source code is source code that may or may not be considered a part of the source code depending on certain conditions. Source code can be conditionally compiled using the attributes `cfg` and `cfg_attr` and the built-in `cfg` macro. These conditions are based on the target architecture of the compiled crate, arbitrary values passed to the compiler, and a few other miscellaneous things further described below in detail.

Each form of conditional compilation takes a *configuration predicate* that evaluates to true or false. The predicate is one of the following:

- A configuration option. It is true if the option is set and false if it is unset.
- `all()` with a comma separated list of configuration predicates. It is false if at least one predicate is false. If there are no predicates, it is true.
- `any()` with a comma separated list of configuration predicates. It is true if at least one predicate is true. If there are no predicates, it is false.

- `not()` with a configuration predicate. It is true if its predicate is false and false if its predicate is true.

Configuration options are names and key-value pairs that are either set or unset. Names are written as a single identifier such as, for example, `unix`. Key-value pairs are written as an identifier, `=`, and then a string. For example, `target_arch = "x86_64"` is a configuration option.

Note: Whitespace around the `=` is ignored. `foo="bar"` and `foo = "bar"` are equivalent configuration options.

Keys are not unique in the set of key-value configuration options. For example, both `feature = "std"` and `feature = "serde"` can be set at the same time.

Set Configuration Options

Which configuration options are set is determined statically during the compilation of the crate. Certain options are *compiler-set* based on data about the compilation. Other options are *arbitrarily-set*, set based on input passed to the compiler outside of the code. It is not possible to set a configuration option from within the source code of the crate being compiled.

Note: For `rustc`, arbitrary-set configuration options are set using the `--cfg` flag.

Note: Configuration options with the key `feature` are a convention used by Cargo for specifying compile-time options and optional dependencies.

> ⚠️ Warning: It is possible for arbitrarily-set configuration options to have the same value as compiler-set configuration options. For example, it is possible to do `rustc --cfg "unix" program.rs` while compiling to a Windows target, and have both `unix` and `windows` configuration options set at the same time. It is unwise to actually do this.

`target_arch`

Key-value option set once with the target's CPU architecture. The value is similar to the first element of the platform's target triple, but not identical.

Example values:

- `"x86"`
- `"x86_64"`
- `"mips"`
- `"powerpc"`
- `"powerpc64"`
- `"arm"`
- `"aarch64"`

`target_feature`

Key-value option set for each platform feature available for the current compilation target.

Example values:

- `"avx"`
- `"avx2"`
- `"crt-static"`
- `"rdrand"`
- `"sse"`
- `"sse2"`
- `"sse4.1"`

See the `target_feature` attribute for more details on the available features. An additional feature of `crt-static` is available to the `target_feature` option to indicate that a static C runtime is available.

`target_os`

Key-value option set once with the target's operating system. This value is similar to the second and third element of the platform's target triple.

Example values:

- `"windows"`
- `"macos"`
- `"ios"`
- `"linux"`
- `"android"`
- `"freebsd"`
- `"dragonfly"`
- `"openbsd"`
- `"netbsd"`
- `"none"` (typical for embedded targets)

`target_family`

Key-value option providing a more generic description of a target, such as the family of the operating systems or architectures that the target generally falls into. Any number of `target_family` key-value pairs can be set.

Example values:

- `"unix"`
- `"windows"`
- `"wasm"`

`unix` and `windows`

`unix` is set if `target_family = "unix"` is set and `windows` is set if `target_family = "windows"` is set.

`target_env`

Key-value option set with further disambiguating information about the target platform with information about the ABI or `libc` used. For historical reasons, this value is only defined as not the empty-string when actually needed for disambiguation. Thus, for example, on many GNU platforms, this value will be empty. This value is similar to the fourth element of the platform's target triple. One difference is that embedded ABIs such as `gnueabihf` will simply define `target_env` as `"gnu"`.

Example values:

- `""`
- `"gnu"`
- `"msvc"`
- `"musl"`
- `"sgx"`

`target_abi`

Key-value option set to further disambiguate the `target_env` with information about the target ABI. For historical reasons, this value is only defined as not the empty-string when actually needed for disambiguation. Thus, for example, on many GNU platforms, this value will be empty.

Example values:

- `""`
- `"llvm"`
- `"eabihf"`
- `"abi64"`

`target_endian`

Key-value option set once with either a value of "little" or "big" depending on the endianness of the target's CPU.

`target_pointer_width`

Key-value option set once with the target's pointer width in bits.

Example values:

- `"16"`
- `"32"`
- `"64"`

`target_vendor`

Key-value option set once with the vendor of the target.

Example values:

- `"apple"`
- `"fortanix"`
- `"pc"`
- `"unknown"`

`target_has_atomic`

Key-value option set for each bit width that the target supports atomic loads, stores, and compare-and-swap operations.

When this cfg is present, all of the stable `core::sync::atomic` APIs are available for the relevant atomic width.

Possible values:

- `"8"`
- `"16"`
- `"32"`
- `"64"`
- `"128"`
- `"ptr"`

`test`

Enabled when compiling the test harness. Done with `rustc` by using the `--test` flag. See Testing for more on testing support.

`debug_assertions`

Enabled by default when compiling without optimizations. This can be used to enable extra debugging code in development but not in production. For example, it controls the behavior of the standard library's `debug_assert!` macro.

`proc_macro`

Set when the crate being compiled is being compiled with the `proc_macro` crate type.

`panic`

Key-value option set depending on the panic strategy. Note that more values may be added in the future.

Example values:

- `"abort"`
- `"unwind"`

Forms of conditional compilation

The `cfg` attribute

> **Syntax**
> *CfgAttrAttribute* :
>   `cfg` `(` *ConfigurationPredicate* `)`

The `cfg` attribute conditionally includes the thing it is attached to based on a configuration predicate.

It is written as `cfg`, `(`, a configuration predicate, and finally `)`.

If the predicate is true, the thing is rewritten to not have the `cfg` attribute on it. If the predicate is false, the thing is removed from the source code.

When a crate-level `cfg` has a false predicate, the behavior is slightly different: any crate attributes preceding the `cfg` are kept, and any crate attributes following the `cfg` are removed. This allows `#![no_std]` and `#![no_core]` crates to avoid linking `std` / `core` even if a `#![cfg(...)]` has removed the entire crate.

Some examples on functions:

```rust
// The function is only included in the build when compiling for macOS
#[cfg(target_os = "macos")]
fn macos_only() {
  // ...
}

// This function is only included when either foo or bar is defined
#[cfg(any(foo, bar))]
fn needs_foo_or_bar() {
  // ...
}

// This function is only included when compiling for a unixish OS with a 32-bit
// architecture
#[cfg(all(unix, target_pointer_width = "32"))]
fn on_32bit_unix() {
  // ...
}

// This function is only included when foo is not defined
#[cfg(not(foo))]
fn needs_not_foo() {
  // ...
}

// This function is only included when the panic strategy is set to unwind
#[cfg(panic = "unwind")]
fn when_unwinding() {
  // ...
}
```

The `cfg` attribute is allowed anywhere attributes are allowed.

The `cfg_attr` attribute

> **Syntax**
>
> *CfgAttrAttribute* :
>
> `cfg_attr` `(` *ConfigurationPredicate* `,` *CfgAttrs*? `)`
>
> *CfgAttrs* :
>
> *Attr* (`,` *Attr*)* `,`?

The `cfg_attr` attribute conditionally includes attributes based on a configuration predicate.

When the configuration predicate is true, this attribute expands out to the attributes listed after the predicate. For example, the following module will either be found at `linux.rs` or `windows.rs` based on the target.

```
#[cfg_attr(target_os = "linux", path = "linux.rs")]
#[cfg_attr(windows, path = "windows.rs")]
mod os;
```

Zero, one, or more attributes may be listed. Multiple attributes will each be expanded into separate attributes. For example:

```
#[cfg_attr(feature = "magic", sparkles, crackles)]
fn bewitched() {}

// When the `magic` feature flag is enabled, the above will expand to:
#[sparkles]
#[crackles]
fn bewitched() {}
```

> **Note**: The `cfg_attr` can expand to another `cfg_attr`. For example, `#[cfg_attr(target_os = "linux", cfg_attr(feature = "multithreaded", some_other_attribute))]` is valid. This example would be equivalent to `#[cfg_attr(all(target_os = "linux", feature ="multithreaded"), some_other_attribute)]`.

The `cfg_attr` attribute is allowed anywhere attributes are allowed.

The `cfg` macro

The built-in `cfg` macro takes in a single configuration predicate and evaluates to the `true` literal when the predicate is true and the `false` literal when it is false.

For example:

```rust
let machine_kind = if cfg!(unix) {
  "unix"
} else if cfg!(windows) {
  "windows"
} else {
  "unknown"
};

println!("I'm running on a {} machine!", machine_kind);
```

Items

Syntax:

Item:
  *OuterAttribute**
    VisItem
  | *MacroItem*

VisItem:
  Visibility?
  (
      Module
    | *ExternCrate*
    | *UseDeclaration*
    | *Function*
    | *TypeAlias*
    | *Struct*
    | *Enumeration*
    | *Union*
    | *ConstantItem*
    | *StaticItem*
    | *Trait*
    | *Implementation*
    | *ExternBlock*
)

MacroItem:
    MacroInvocationSemi
  | *MacroRulesDefinition*

An *item* is a component of a crate. Items are organized within a crate by a nested set of modules. Every crate has a single "outermost" anonymous module; all further items within the crate have paths within the module tree of the crate.

Items are entirely determined at compile-time, generally remain fixed during execution, and may reside in read-only memory.

There are several kinds of items:

- modules
- `extern crate` declarations
- `use` declarations
- function definitions
- type definitions
- struct definitions
- enumeration definitions
- union definitions
- constant items
- static items
- trait definitions
- implementations
- `extern` blocks

Some items form an implicit scope for the declaration of sub-items. In other words, within a function or module, declarations of items can (in many cases) be mixed with the statements, control blocks, and similar artifacts that otherwise compose the item body. The meaning of these scoped items is the same as if the item was declared outside the scope — it is still a static item — except that the item's *path name* within the module namespace is qualified by the name of the enclosing item, or is private to the enclosing item (in the case of functions). The grammar specifies the exact locations in which sub-item declarations may appear.

Modules

Syntax:

Module :
   `unsafe`? `mod` IDENTIFIER `;`
 | `unsafe`? `mod` IDENTIFIER `{`
    *InnerAttribute**
    *Item**
   `}`

A module is a container for zero or more items.

A *module item* is a module, surrounded in braces, named, and prefixed with the keyword `mod`. A module item introduces a new, named module into the tree of modules making up a crate. Modules can nest arbitrarily.

An example of a module:

```
mod math {
    type Complex = (f64, f64);
    fn sin(f: f64) -> f64 {
        /* ... */
    }
    fn cos(f: f64) -> f64 {
        /* ... */
    }
    fn tan(f: f64) -> f64 {
        /* ... */
    }
}
```

Modules and types share the same namespace. Declaring a named type with the same name as a module in scope is forbidden: that is, a type definition, trait, struct, enumeration, union, type parameter or crate can't shadow the name of a module in scope, or vice versa. Items brought into scope with `use` also have this restriction.

The `unsafe` keyword is syntactically allowed to appear before the `mod` keyword, but it is rejected at a semantic level. This allows macros to consume the syntax and make use of the `unsafe` keyword, before removing it from the token stream.

Module Source Filenames

A module without a body is loaded from an external file. When the module does not have a `path` attribute, the path to the file mirrors the logical module path. Ancestor module path components are directories, and the module's contents are in a file with the name of the module plus the `.rs` extension. For example, the following module structure can have this corresponding filesystem structure:

Module Path	Filesystem Path	File Contents
`crate`	`lib.rs`	`mod util;`
`crate::util`	`util.rs`	`mod config;`
`crate::util::config`	`util/config.rs`	

Module filenames may also be the name of the module as a directory with the contents in a file named `mod.rs` within that directory. The above example can alternately be expressed with `crate::util`'s contents in a file named `util/mod.rs`. It is not allowed to have both `util.rs` and `util/mod.rs`.

> **Note**: Prior to `rustc` 1.30, using `mod.rs` files was the way to load a module with nested children. It is encouraged to use the new naming convention as it is more consistent, and avoids having many files named `mod.rs` within a project.

The `path` attribute

The directories and files used for loading external file modules can be influenced with the `path` attribute.

For `path` attributes on modules not inside inline module blocks, the file path is relative to the directory the source file is located. For example, the following code snippet would use the paths shown based on where it is located:

```
#[path = "foo.rs"]
mod c;
```

Source File	`c`'s File Location	`c`'s Module Path
`src/a/b.rs`	`src/a/foo.rs`	`crate::a::b::c`
`src/a/mod.rs`	`src/a/foo.rs`	`crate::a::c`

For `path` attributes inside inline module blocks, the relative location of the file path depends on the kind of source file the `path` attribute is located in. "mod-rs" source files are root modules (such as `lib.rs` or `main.rs`) and modules with files named `mod.rs`. "non-mod-rs" source files are all other module files. Paths for `path` attributes inside inline module blocks in a mod-rs file are relative to the directory of the mod-rs file including the inline module components as directories. For non-mod-rs files, it is the same except the path starts with a directory with the name of the non-mod-rs module. For example, the following code snippet would use the paths shown based on where it is located:

```
mod inline {
    #[path = "other.rs"]
    mod inner;
}
```

Source File	`inner`'s File Location	`inner`'s Module Path
`src/a/b.rs`	`src/a/b/inline/other.rs`	`crate::a::b::inline::inner`
`src/a/mod.rs`	`src/a/inline/other.rs`	`crate::a::inline::inner`

An example of combining the above rules of `path` attributes on inline modules and nested modules within (applies to both mod-rs and non-mod-rs files):

```
#[path = "thread_files"]
mod thread {
    // Load the `local_data` module from `thread_files/tls.rs` relative to
    // this source file's directory.
    #[path = "tls.rs"]
    mod local_data;
}
```

Attributes on Modules

Modules, like all items, accept outer attributes. They also accept inner attributes: either after `{` for a module with a body, or at the beginning of the source file, after the optional BOM and shebang.

The built-in attributes that have meaning on a module are `cfg`, `deprecated`, `doc`, the lint check attributes, `path`, and `no_implicit_prelude`. Modules also accept macro attributes.

Extern crate declarations

Syntax:

ExternCrate :
  `extern` `crate` *CrateRef* *AsClause*? `;`

CrateRef :
  IDENTIFIER | `self`

AsClause :
  `as` (IDENTIFIER | `_`)

An *`extern crate` declaration* specifies a dependency on an external crate. The external crate is then bound into the declaring scope as the identifier provided in the `extern crate` declaration. Additionally, if the `extern crate` appears in the crate root, then the crate name is also added to the extern prelude, making it automatically in scope in all modules. The `as` clause can be used to bind the imported crate to a different name.

The external crate is resolved to a specific `soname` at compile time, and a runtime linkage requirement to that `soname` is passed to the linker for loading at runtime. The `soname` is resolved at compile time by scanning the compiler's library path and matching the optional `crate_name` provided against the `crate_name` attributes that were declared on the external crate when it was compiled. If no `crate_name` is provided, a default `name` attribute is assumed, equal to the identifier given in the `extern crate` declaration.

The `self` crate may be imported which creates a binding to the current crate. In this case the `as` clause must be used to specify the name to bind it to.

Three examples of `extern crate` declarations:

```
extern crate pcre;

extern crate std; // equivalent to: extern crate std as std;

extern crate std as ruststd; // linking to 'std' under another name
```

When naming Rust crates, hyphens are disallowed. However, Cargo packages may make use of them. In such case, when `Cargo.toml` doesn't specify a crate name, Cargo will transparently replace `-` with `_` (Refer to RFC 940 for more details).

Here is an example:

```
// Importing the Cargo package hello-world
extern crate hello_world; // hyphen replaced with an underscore
```

Extern Prelude

This section has been moved to Preludes — Extern Prelude.

Underscore Imports

An external crate dependency can be declared without binding its name in scope by using an underscore with the form `extern crate foo as _`. This may be useful for crates that only need to be linked, but are never referenced, and will avoid being reported as unused.

The `macro_use` attribute works as usual and imports the macro names into the `macro_use` prelude.

The `no_link` attribute

The *`no_link` attribute* may be specified on an `extern crate` item to prevent linking the crate into the output. This is commonly used to load a crate to access only its macros.

Use declarations

Syntax:

UseDeclaration :
  `use` *UseTree* `;`

UseTree :
  (*SimplePath*? `::`)? `*`
  | (*SimplePath*? `::`)? `{` (*UseTree* (`,` *UseTree*)* `,`?)? `}`
  | *SimplePath* (`as` (IDENTIFIER | `_`))?

A *use declaration* creates one or more local name bindings synonymous with some other path. Usually a `use` declaration is used to shorten the path required to refer to a module item. These declarations may appear in modules and blocks, usually at the top.

Use declarations support a number of convenient shortcuts:

- Simultaneously binding a list of paths with a common prefix, using the glob-like brace syntax `use a::b::{c, d, e::f, g::h::i};`
- Simultaneously binding a list of paths with a common prefix and their common parent module, using the `self` keyword, such as `use a::b::{self, c, d::e};`
- Rebinding the target name as a new local name, using the syntax `use p::q::r as x;`. This can also be used with the last two features: `use a::b::{self as ab, c as abc}`.
- Binding all paths matching a given prefix, using the asterisk wildcard syntax `use a::b::*;`.
- Nesting groups of the previous features multiple times, such as `use a::b::{self as ab, c, d::{*, e::f}};`

An example of `use` declarations:

```
use std::collections::hash_map::{self, HashMap};

fn foo<T>(_: T){}
fn bar(map1: HashMap<String, usize>, map2: hash_map::HashMap<String, usize>){}

fn main() {
    // use declarations can also exist inside of functions
    use std::option::Option::{Some, None};

    // Equivalent to 'foo(vec![std::option::Option::Some(1.0f64),
    // std::option::Option::None]);'
    foo(vec![Some(1.0f64), None]);

    // Both `hash_map` and `HashMap` are in scope.
    let map1 = HashMap::new();
    let map2 = hash_map::HashMap::new();
    bar(map1, map2);
}
```

`use` Visibility

Like items, `use` declarations are private to the containing module, by default. Also like items, a `use` declaration can be public, if qualified by the `pub` keyword. Such a `use` declaration serves to *re-export* a name. A public `use` declaration can therefore *redirect* some public name to a different target definition: even a definition with a private canonical path, inside a different module. If a sequence of such redirections form a cycle or cannot be resolved unambiguously, they represent a compile-time error.

An example of re-exporting:

```
mod quux {
    pub use self::foo::{bar, baz};
    pub mod foo {
        pub fn bar() {}
        pub fn baz() {}
    }
}

fn main() {
    quux::bar();
    quux::baz();
}
```

In this example, the module `quux` re-exports two public names defined in `foo` .

`use` Paths

> **Note**: This section is incomplete.

Some examples of what will and will not work for `use` items:

```
use std::path::{self, Path, PathBuf};  // good: std is a crate name
use crate::foo::baz::foobaz;    // good: foo is at the root of the crate

mod foo {

    pub mod example {
        pub mod iter {}
    }

    use crate::foo::example::iter; // good: foo is at crate root
//  use example::iter;      // bad in 2015 edition: relative paths are not
allowed without `self`; good in 2018 edition
    use self::baz::foobaz;  // good: self refers to module 'foo'
    use crate::foo::bar::foobar;   // good: foo is at crate root

    pub mod bar {
        pub fn foobar() { }
    }

    pub mod baz {
        use super::bar::foobar; // good: super refers to module 'foo'
        pub fn foobaz() { }
    }
}

fn main() {}
```

> **Edition Differences**: In the 2015 edition, `use` paths also allow accessing items in the crate root. Using the example above, the following `use` paths work in 2015 but not 2018:
>
> ```
> use foo::example::iter;
> use ::foo::baz::foobaz;
> ```
>
> The 2015 edition does not allow use declarations to reference the extern prelude. Thus `extern crate` declarations are still required in 2015 to reference an external crate in a use declaration. Beginning with the 2018 edition, use declarations can specify an external crate dependency the same way `extern crate` can.

In the 2018 edition, if an in-scope item has the same name as an external crate, then `use` of that crate name requires a leading `::` to unambiguously select the crate name. This is to retain compatibility with potential future changes.

```
// use std::fs; // Error, this is ambiguous.
use ::std::fs;  // Imports from the `std` crate, not the module below.
use self::std::fs as self_fs;  // Imports the module below.

mod std {
    pub mod fs {}
}
```

Underscore Imports

Items can be imported without binding to a name by using an underscore with the form `use path as _`. This is particularly useful to import a trait so that its methods may be used without importing the trait's symbol, for example if the trait's symbol may conflict with another symbol. Another example is to link an external crate without importing its name.

Asterisk glob imports will import items imported with `_` in their unnameable form.

```
mod foo {
    pub trait Zoo {
        fn zoo(&self) {}
    }

    impl<T> Zoo for T {}
}

use self::foo::Zoo as _;
struct Zoo;  // Underscore import avoids name conflict with this item.

fn main() {
    let z = Zoo;
    z.zoo();
}
```

The unique, unnameable symbols are created after macro expansion so that macros may safely emit multiple references to `_` imports. For example, the following should not produce an error:

```rust
macro_rules! m {
    ($item: item) => { $item $item }
}

m!(use std as _;);
// This expands to:
// use std as _;
// use std as _;
```

Functions

Syntax

Function :
  FunctionQualifiers `fn` IDENTIFIER *GenericParams*?
    (*FunctionParameters*?)
    FunctionReturnType? *WhereClause*?
    (*BlockExpression* | `;`)

FunctionQualifiers :
  `const`? `async`[1]? `unsafe`? (`extern` *Abi*?)?

Abi :
  STRING_LITERAL | RAW_STRING_LITERAL

FunctionParameters :
    SelfParam `,`?
  | (*SelfParam* `,`)? *FunctionParam* (`,` *FunctionParam*)* `,`?

SelfParam :
  *OuterAttribute** (*ShorthandSelf* | *TypedSelf*)

ShorthandSelf :
  (`&` | `&` *Lifetime*)? `mut`? `self`

TypedSelf :
  `mut`? `self` `:` *Type*

FunctionParam :
  *OuterAttribute** (*FunctionParamPattern* | `...` | *Type* [2])

FunctionParamPattern :
  PatternNoTopAlt `:` (*Type* | `...`)

FunctionReturnType :
  `->` *Type*

[1] The `async` qualifier is not allowed in the 2015 edition.

[2] Function parameters with only a type are only allowed in an associated function of a trait item

in the 2015 edition.

A *function* consists of a block, along with a name, a set of parameters, and an output type. Other than a name, all these are optional. Functions are declared with the keyword `fn`. Functions may declare a set of *input variables* as parameters, through which the caller passes arguments into the function, and the *output type* of the value the function will return to its caller on completion. If the output type is not explicitly stated, it is the unit type.

When referred to, a *function* yields a first-class *value* of the corresponding zero-sized *function item type*, which when called evaluates to a direct call to the function.

For example, this is a simple function:

```
fn answer_to_life_the_universe_and_everything() -> i32 {
    return 42;
}
```

Function parameters

Function parameters are irrefutable patterns, so any pattern that is valid in an else-less `let` binding is also valid as a parameter:

```
fn first((value, _): (i32, i32)) -> i32 { value }
```

If the first parameter is a *SelfParam*, this indicates that the function is a method. Functions with a self parameter may only appear as an associated function in a trait or implementation.

A parameter with the `...` token indicates a variadic function, and may only be used as the last parameter of an external block function. The variadic parameter may have an optional identifier, such as `args: ...`.

Function body

The block of a function is conceptually wrapped in a block that binds the argument patterns and then `return`s the value of the function's block. This means that the tail expression of the block, if evaluated, ends up being returned to the caller. As usual, an explicit return expression within the body of the function will short-cut that implicit return, if reached.

For example, the function above behaves as if it was written as:

```
// argument_0 is the actual first argument passed from the caller
let (value, _) = argument_0;
return {
    value
};
```

Functions without a body block are terminated with a semicolon. This form may only appear in a trait or external block.

Generic functions

A *generic function* allows one or more *parameterized types* to appear in its signature. Each type parameter must be explicitly declared in an angle-bracket-enclosed and comma-separated list, following the function name.

```
// foo is generic over A and B

fn foo<A, B>(x: A, y: B) {
```

Inside the function signature and body, the name of the type parameter can be used as a type name. Trait bounds can be specified for type parameters to allow methods with that trait to be called on values of that type. This is specified using the `where` syntax:

```
fn foo<T>(x: T) where T: Debug {
```

When a generic function is referenced, its type is instantiated based on the context of the reference. For example, calling the `foo` function here:

```
use std::fmt::Debug;

fn foo<T>(x: &[T]) where T: Debug {
    // details elided
}

foo(&[1, 2]);
```

will instantiate type parameter `T` with `i32`.

The type parameters can also be explicitly supplied in a trailing path component after the function name. This might be necessary if there is not sufficient context to determine the type parameters. For example, `mem::size_of::<u32>() == 4`.

Extern function qualifier

The `extern` function qualifier allows providing function *definitions* that can be called with a particular ABI:

```
extern "ABI" fn foo() { /* ... */ }
```

These are often used in combination with external block items which provide function *declarations* that can be used to call functions without providing their *definition*:

```
extern "ABI" {
  fn foo(); /* no body */
}
unsafe { foo() }
```

When `"extern" Abi?*` is omitted from `FunctionQualifiers` in function items, the ABI `"Rust"` is assigned. For example:

```
fn foo() {}
```

is equivalent to:

```
extern "Rust" fn foo() {}
```

Functions can be called by foreign code, and using an ABI that differs from Rust allows, for example, to provide functions that can be called from other programming languages like C:

```
// Declares a function with the "C" ABI
extern "C" fn new_i32() -> i32 { 0 }

// Declares a function with the "stdcall" ABI
extern "stdcall" fn new_i32_stdcall() -> i32 { 0 }
```

Just as with external block, when the `extern` keyword is used and the `"ABI"` is omitted, the ABI used defaults to `"C"`. That is, this:

```
extern fn new_i32() -> i32 { 0 }
let fptr: extern fn() -> i32 = new_i32;
```

is equivalent to:

```
extern "C" fn new_i32() -> i32 { 0 }
let fptr: extern "C" fn() -> i32 = new_i32;
```

Functions with an ABI that differs from `"Rust"` do not support unwinding in the exact same way that Rust does. Therefore, unwinding past the end of functions with such ABIs causes the process to abort.

> **Note**: The LLVM backend of the `rustc` implementation aborts the process by executing an illegal instruction.

Const functions

Functions qualified with the `const` keyword are const functions, as are tuple struct and tuple variant constructors. *Const functions* can be called from within const contexts.

Const functions may use the `extern` function qualifier, but only with the `"Rust"` and `"C"` ABIs.

Const functions are not allowed to be async.

Async functions

Functions may be qualified as async, and this can also be combined with the `unsafe` qualifier:

```
async fn regular_example() { }
async unsafe fn unsafe_example() { }
```

Async functions do no work when called: instead, they capture their arguments into a future. When polled, that future will execute the function's body.

An async function is roughly equivalent to a function that returns `impl Future` and with an `async move` block as its body:

```rust
// Source
async fn example(x: &str) -> usize {
    x.len()
}
```

is roughly equivalent to:

```rust
// Desugared
fn example<'a>(x: &'a str) -> impl Future<Output = usize> + 'a {
    async move { x.len() }
}
```

The actual desugaring is more complex:

- The return type in the desugaring is assumed to capture all lifetime parameters from the `async fn` declaration. This can be seen in the desugared example above, which explicitly outlives, and hence captures, `'a`.
- The `async move` block in the body captures all function parameters, including those that are unused or bound to a `_` pattern. This ensures that function parameters are dropped in the same order as they would be if the function were not async, except that the drop occurs when the returned future has been fully awaited.

For more information on the effect of async, see `async` blocks.

> **Edition differences**: Async functions are only available beginning with Rust 2018.

Combining `async` and `unsafe`

It is legal to declare a function that is both async and unsafe. The resulting function is unsafe to call and (like any async function) returns a future. This future is just an ordinary future and thus an `unsafe` context is not required to "await" it:

```
// Returns a future that, when awaited, dereferences `x`.
//
// Soundness condition: `x` must be safe to dereference until
// the resulting future is complete.
async unsafe fn unsafe_example(x: *const i32) -> i32 {
  *x
}

async fn safe_example() {
    // An `unsafe` block is required to invoke the function initially:
    let p = 22;
    let future = unsafe { unsafe_example(&p) };

    // But no `unsafe` block required here. This will
    // read the value of `p`:
    let q = future.await;
}
```

Note that this behavior is a consequence of the desugaring to a function that returns an `impl Future` -- in this case, the function we desugar to is an `unsafe` function, but the return value remains the same.

Unsafe is used on an async function in precisely the same way that it is used on other functions: it indicates that the function imposes some additional obligations on its caller to ensure soundness. As in any other unsafe function, these conditions may extend beyond the initial call itself -- in the snippet above, for example, the `unsafe_example` function took a pointer `x` as argument, and then (when awaited) dereferenced that pointer. This implies that `x` would have to be valid until the future is finished executing, and it is the caller's responsibility to ensure that.

Attributes on functions

Outer attributes are allowed on functions. Inner attributes are allowed directly after the `{` inside its block.

This example shows an inner attribute on a function. The function is documented with just the word "Example".

```
fn documented() {
    #![doc = "Example"]
}
```

> Note: Except for lints, it is idiomatic to only use outer attributes on function items.

The attributes that have meaning on a function are `cfg`, `cfg_attr`, `deprecated`, `doc`, `export_name`, `link_section`, `no_mangle`, the lint check attributes, `must_use`, the procedural macro attributes, the testing attributes, and the optimization hint attributes. Functions also accept attributes macros.

Attributes on function parameters

Outer attributes are allowed on function parameters and the permitted built-in attributes are restricted to `cfg`, `cfg_attr`, `allow`, `warn`, `deny`, and `forbid`.

```
fn len(
    #[cfg(windows)] slice: &[u16],
    #[cfg(not(windows))] slice: &[u8],
) -> usize {
    slice.len()
}
```

Inert helper attributes used by procedural macro attributes applied to items are also allowed but be careful to not include these inert attributes in your final `TokenStream`.

For example, the following code defines an inert `some_inert_attribute` attribute that is not formally defined anywhere and the `some_proc_macro_attribute` procedural macro is responsible for detecting its presence and removing it from the output token stream.

```
#[some_proc_macro_attribute]
fn foo_oof(#[some_inert_attribute] arg: u8) {
}
```

Type aliases

Syntax
TypeAlias :
`type` IDENTIFIER *GenericParams*? (`:` *TypeParamBounds*)? *WhereClause*? (`=` *Type* *WhereClause*?)? `;`

A *type alias* defines a new name for an existing type. Type aliases are declared with the keyword `type`. Every value has a single, specific type, but may implement several different traits, or be compatible with several different type constraints.

For example, the following defines the type `Point` as a synonym for the type `(u8, u8)`, the type of pairs of unsigned 8 bit integers:

```
type Point = (u8, u8);
let p: Point = (41, 68);
```

A type alias to a tuple-struct or unit-struct cannot be used to qualify that type's constructor:

```
struct MyStruct(u32);

use MyStruct as UseAlias;
type TypeAlias = MyStruct;

let _ = UseAlias(5); // OK
let _ = TypeAlias(5); // Doesn't work
```

A type alias, when not used as an associated type, must include a *Type* and may not include *TypeParamBounds*.

A type alias, when used as an associated type in a trait, must not include a *Type* specification but may include *TypeParamBounds*.

A type alias, when used as an associated type in a trait impl, must include a *Type* specification and may not include *TypeParamBounds*.

Where clauses before the equals sign on a type alias in a trait impl (like `type TypeAlias<T> where T: Foo = Bar<T>`) are deprecated. Where clauses after the equals sign (like `type TypeAlias<T> = Bar<T> where T: Foo`) are preferred.

Structs

> **Syntax**
>
> *Struct* :
>   *StructStruct*
> | *TupleStruct*
>
> *StructStruct* :
>   `struct` IDENTIFIER *GenericParams*? *WhereClause*? (`{` *StructFields*? `}` | `;`)
>
> *TupleStruct* :
>   `struct` IDENTIFIER *GenericParams*? `(` *TupleFields*? `)` *WhereClause*? `;`
>
> *StructFields* :
>   *StructField* (`,` *StructField*)* `,`?
>
> *StructField* :
>   *OuterAttribute**
>   *Visibility*?
>   IDENTIFIER `:` *Type*
>
> *TupleFields* :
>   *TupleField* (`,` *TupleField*)* `,`?
>
> *TupleField* :
>   *OuterAttribute**
>   *Visibility*?
>   *Type*

A *struct* is a nominal struct type defined with the keyword `struct`.

An example of a `struct` item and its use:

```
struct Point {x: i32, y: i32}
let p = Point {x: 10, y: 11};
let px: i32 = p.x;
```

A *tuple struct* is a nominal tuple type, also defined with the keyword `struct`. For example:

```
struct Point(i32, i32);
let p = Point(10, 11);
let px: i32 = match p { Point(x, _) => x };
```

A *unit-like struct* is a struct without any fields, defined by leaving off the list of fields entirely. Such a struct implicitly defines a constant of its type with the same name. For example:

```
struct Cookie;
let c = [Cookie, Cookie {}, Cookie, Cookie {}];
```

is equivalent to

```
struct Cookie {}
const Cookie: Cookie = Cookie {};
let c = [Cookie, Cookie {}, Cookie, Cookie {}];
```

The precise memory layout of a struct is not specified. One can specify a particular layout using the `repr` attribute.

Enumerations

Syntax

Enumeration :
`enum` IDENTIFIER *GenericParams*? *WhereClause*? `{` *EnumItems*? `}`

EnumItems :
EnumItem (`,` *EnumItem*)* `,` ?

EnumItem :
*OuterAttribute** *Visibility*?
IDENTIFIER (*EnumItemTuple* | *EnumItemStruct*)? *EnumItemDiscriminant*?

EnumItemTuple :
`(` *TupleFields*? `)`

EnumItemStruct :
`{` *StructFields*? `}`

EnumItemDiscriminant :
`=` *Expression*

An *enumeration*, also referred to as an *enum*, is a simultaneous definition of a nominal enumerated type as well as a set of *constructors*, that can be used to create or pattern-match values of the corresponding enumerated type.

Enumerations are declared with the keyword `enum`.

An example of an `enum` item and its use:

```
enum Animal {
    Dog,
    Cat,
}

let mut a: Animal = Animal::Dog;
a = Animal::Cat;
```

Enum constructors can have either named or unnamed fields:

```
enum Animal {
    Dog(String, f64),
    Cat { name: String, weight: f64 },
}

let mut a: Animal = Animal::Dog("Cocoa".to_string(), 37.2);
a = Animal::Cat { name: "Spotty".to_string(), weight: 2.7 };
```

In this example, `Cat` is a *struct-like enum variant*, whereas `Dog` is simply called an enum variant.

An enum where no constructors contain fields are called a *field-less enum*. For example, this is a fieldless enum:

```
enum Fieldless {
    Tuple(),
    Struct{},
    Unit,
}
```

If a field-less enum only contains unit variants, the enum is called an *unit-only enum*. For example:

```
enum Enum {
    Foo = 3,
    Bar = 2,
    Baz = 1,
}
```

Discriminants

Each enum instance has a *discriminant*: an integer logically associated to it that is used to determine which variant it holds.

Under the default representation, the discriminant is interpreted as an `isize` value. However, the compiler is allowed to use a smaller type (or another means of distinguishing variants) in its actual memory layout.

Assigning discriminant values

Explicit discriminants

In two circumstances, the discriminant of a variant may be explicitly set by following the variant name with `=` and a constant expression:

1. if the enumeration is "unit-only".

2. if a primitive representation is used. For example:

```
#[repr(u8)]
enum Enum {
    Unit = 3,
    Tuple(u16),
    Struct {
        a: u8,
        b: u16,
    } = 1,
}
```

Implicit discriminants

If a discriminant for a variant is not specified, then it is set to one higher than the discriminant of the previous variant in the declaration. If the discriminant of the first variant in the declaration is unspecified, then it is set to zero.

```
enum Foo {
    Bar,            // 0
    Baz = 123,      // 123
    Quux,           // 124
}

let baz_discriminant = Foo::Baz as u32;
assert_eq!(baz_discriminant, 123);
```

Restrictions

It is an error when two variants share the same discriminant.

```
enum SharedDiscriminantError {
    SharedA = 1,
    SharedB = 1
}

enum SharedDiscriminantError2 {
    Zero,       // 0
    One,        // 1
    OneToo = 1  // 1 (collision with previous!)
}
```

It is also an error to have an unspecified discriminant where the previous discriminant is the maximum value for the size of the discriminant.

```
#[repr(u8)]
enum OverflowingDiscriminantError {
    Max = 255,
    MaxPlusOne // Would be 256, but that overflows the enum.
}

#[repr(u8)]
enum OverflowingDiscriminantError2 {
    MaxMinusOne = 254, // 254
    Max,               // 255
    MaxPlusOne         // Would be 256, but that overflows the enum.
}
```

Accessing discriminant

Via `mem::discriminant`

`mem::discriminant` returns an opaque reference to the discriminant of an enum value which can be compared. This cannot be used to get the value of the discriminant.

Casting

If an enumeration is unit-only (with no tuple and struct variants), then its discriminant can be directly accessed with a numeric cast; e.g.:

```rust
enum Enum {
    Foo,
    Bar,
    Baz,
}

assert_eq!(0, Enum::Foo as isize);
assert_eq!(1, Enum::Bar as isize);
assert_eq!(2, Enum::Baz as isize);
```

Field-less enums can be casted if they do not have explicit discriminants, or where only unit variants are explicit.

```rust
enum Fieldless {
    Tuple(),
    Struct{},
    Unit,
}

assert_eq!(0, Fieldless::Tuple() as isize);
assert_eq!(1, Fieldless::Struct{} as isize);
assert_eq!(2, Fieldless::Unit as isize);

#[repr(u8)]
enum FieldlessWithDiscrimants {
    First = 10,
    Tuple(),
    Second = 20,
    Struct{},
    Unit,
}

assert_eq!(10, FieldlessWithDiscrimants::First as u8);
assert_eq!(11, FieldlessWithDiscrimants::Tuple() as u8);
assert_eq!(20, FieldlessWithDiscrimants::Second as u8);
assert_eq!(21, FieldlessWithDiscrimants::Struct{} as u8);
assert_eq!(22, FieldlessWithDiscrimants::Unit as u8);
```

Pointer casting

If the enumeration specifies a primitive representation, then the discriminant may be reliably accessed via unsafe pointer casting:

```
#[repr(u8)]
enum Enum {
    Unit,
    Tuple(bool),
    Struct{a: bool},
}

impl Enum {
    fn discriminant(&self) -> u8 {
        unsafe { *(self as *const Self as *const u8) }
    }
}

let unit_like = Enum::Unit;
let tuple_like = Enum::Tuple(true);
let struct_like = Enum::Struct{a: false};

assert_eq!(0, unit_like.discriminant());
assert_eq!(1, tuple_like.discriminant());
assert_eq!(2, struct_like.discriminant());
```

Zero-variant enums

Enums with zero variants are known as *zero-variant enums*. As they have no valid values, they cannot be instantiated.

```
enum ZeroVariants {}
```

Zero-variant enums are equivalent to the never type, but they cannot be coerced into other types.

```
let x: ZeroVariants = panic!();
let y: u32 = x; // mismatched type error
```

Variant visibility

Enum variants syntactically allow a *Visibility* annotation, but this is rejected when the enum is validated. This allows items to be parsed with a unified syntax across different contexts where they are used.

```
macro_rules! mac_variant {
    ($vis:vis $name:ident) => {
        enum $name {
            $vis Unit,

            $vis Tuple(u8, u16),

            $vis Struct { f: u8 },
        }
    }
}

// Empty `vis` is allowed.
mac_variant! { E }

// This is allowed, since it is removed before being validated.
#[cfg(FALSE)]
enum E {
    pub U,
    pub(crate) T(u8),
    pub(super) T { f: String }
}
```

Unions

Syntax

Union :

`union` IDENTIFIER *GenericParams*? *WhereClause*? `{` *StructFields* `}`

A union declaration uses the same syntax as a struct declaration, except with `union` in place of `struct`.

```
#[repr(C)]
union MyUnion {
    f1: u32,
    f2: f32,
}
```

The key property of unions is that all fields of a union share common storage. As a result, writes to one field of a union can overwrite its other fields, and size of a union is determined by the size of its largest field.

Union field types are restricted to the following subset of types:

- `Copy` types
- References (`&T` and `&mut T` for arbitrary `T`)
- `ManuallyDrop<T>` (for arbitrary `T`)
- Tuples and arrays containing only allowed union field types

This restriction ensures, in particular, that union fields never need to be dropped. Like for structs and enums, it is possible to `impl Drop` for a union to manually define what happens when it gets dropped.

Initialization of a union

A value of a union type can be created using the same syntax that is used for struct types, except that it must specify exactly one field:

```
let u = MyUnion { f1: 1 };
```

The expression above creates a value of type `MyUnion` and initializes the storage using field `f1`. The union can be accessed using the same syntax as struct fields:

```
let f = unsafe { u.f1 };
```

Reading and writing union fields

Unions have no notion of an "active field". Instead, every union access just interprets the storage as the type of the field used for the access. Reading a union field reads the bits of the union at the field's type. Fields might have a non-zero offset (except when the C representation is used); in that case the bits starting at the offset of the fields are read. It is the programmer's responsibility to make sure that the data is valid at the field's type. Failing to do so results in undefined behavior. For example, reading the value `3` from a field of the boolean type is undefined behavior. Effectively, writing to and then reading from a union with the C representation is analogous to a `transmute` from the type used for writing to the type used for reading.

Consequently, all reads of union fields have to be placed in `unsafe` blocks:

```
unsafe {
    let f = u.f1;
}
```

Commonly, code using unions will provide safe wrappers around unsafe union field accesses.

In contrast, writes to union fields are safe, since they just overwrite arbitrary data, but cannot cause undefined behavior. (Note that union field types can never have drop glue, so a union field write will never implicitly drop anything.)

Pattern matching on unions

Another way to access union fields is to use pattern matching. Pattern matching on union fields uses the same syntax as struct patterns, except that the pattern must specify exactly one field. Since pattern matching is like reading the union with a particular field, it has to be placed in `unsafe` blocks as well.

```
fn f(u: MyUnion) {
    unsafe {
        match u {
            MyUnion { f1: 10 } => { println!("ten"); }
            MyUnion { f2 } => { println!("{}", f2); }
        }
    }
}
```

Pattern matching may match a union as a field of a larger structure. In particular, when using a Rust union to implement a C tagged union via FFI, this allows matching on the tag and the corresponding field simultaneously:

```
#[repr(u32)]
enum Tag { I, F }

#[repr(C)]
union U {
    i: i32,
    f: f32,
}

#[repr(C)]
struct Value {
    tag: Tag,
    u: U,
}

fn is_zero(v: Value) -> bool {
    unsafe {
        match v {
            Value { tag: Tag::I, u: U { i: 0 } } => true,
            Value { tag: Tag::F, u: U { f: num } } if num == 0.0 => true,
            _ => false,
        }
    }
}
```

References to union fields

Since union fields share common storage, gaining write access to one field of a union can give write access to all its remaining fields. Borrow checking rules have to be adjusted to account for this fact. As a result, if one field of a union is borrowed, all its remaining fields are borrowed as well for the same lifetime.

```
// ERROR: cannot borrow `u` (via `u.f2`) as mutable more than once at a time
fn test() {
    let mut u = MyUnion { f1: 1 };
    unsafe {
        let b1 = &mut u.f1;
//                    ---- first mutable borrow occurs here (via `u.f1`)
        let b2 = &mut u.f2;
//                    ^^^^ second mutable borrow occurs here (via `u.f2`)
        *b1 = 5;
    }
//  - first borrow ends here
    assert_eq!(unsafe { u.f1 }, 5);
}
```

As you could see, in many aspects (except for layouts, safety, and ownership) unions behave exactly like structs, largely as a consequence of inheriting their syntactic shape from structs. This is also true for many unmentioned aspects of Rust language (such as privacy, name resolution, type inference, generics, trait implementations, inherent implementations, coherence, pattern checking, etc etc etc).

Constant items

> **Syntax**
>
> *ConstantItem* :
>
> `const` (IDENTIFIER | `_`) `:` *Type* (`=` *Expression*)? `;`

A *constant item* is an optionally named *constant value* which is not associated with a specific memory location in the program. Constants are essentially inlined wherever they are used, meaning that they are copied directly into the relevant context when used. This includes usage of constants from external crates, and non- `Copy` types. References to the same constant are not necessarily guaranteed to refer to the same memory address.

Constants must be explicitly typed. The type must have a `'static` lifetime: any references in the initializer must have `'static` lifetimes.

Constants may refer to the address of other constants, in which case the address will have elided lifetimes where applicable, otherwise – in most cases – defaulting to the `static` lifetime. (See static lifetime elision.) The compiler is, however, still at liberty to translate the constant many times, so the address referred to may not be stable.

```rust
const BIT1: u32 = 1 << 0;
const BIT2: u32 = 1 << 1;

const BITS: [u32; 2] = [BIT1, BIT2];
const STRING: &'static str = "bitstring";

struct BitsNStrings<'a> {
    mybits: [u32; 2],
    mystring: &'a str,
}

const BITS_N_STRINGS: BitsNStrings<'static> = BitsNStrings {
    mybits: BITS,
    mystring: STRING,
};
```

The constant expression may only be omitted in a trait definition.

Constants with Destructors

Constants can contain destructors. Destructors are run when the value goes out of scope.

```
struct TypeWithDestructor(i32);

impl Drop for TypeWithDestructor {
    fn drop(&mut self) {
        println!("Dropped. Held {}.", self.0);
    }
}

const ZERO_WITH_DESTRUCTOR: TypeWithDestructor = TypeWithDestructor(0);

fn create_and_drop_zero_with_destructor() {
    let x = ZERO_WITH_DESTRUCTOR;
    // x gets dropped at end of function, calling drop.
    // prints "Dropped. Held 0.".
}
```

Unnamed constant

Unlike an associated constant, a free constant may be unnamed by using an underscore instead of the name. For example:

```
const _: () =  { struct _SameNameTwice; };

// OK although it is the same name as above:
const _: () =  { struct _SameNameTwice; };
```

As with underscore imports, macros may safely emit the same unnamed constant in the same scope more than once. For example, the following should not produce an error:

```
macro_rules! m {
    ($item: item) => { $item $item }
}

m!(const _: () = (););
// This expands to:
// const _: () = ();
// const _: () = ();
```

Evaluation

Free constants are always evaluated at compile-time to surface panics. This happens even within an unused function:

```
// Compile-time panic
const PANIC: () = std::unimplemented!();

fn unused_generic_function<T>() {
    // A failing compile-time assertion
    const _: () = assert!(usize::BITS == 0);
}
```

Static items

> **Syntax**
>
> *StaticItem* :
>
> `static` `mut`? IDENTIFIER `:` *Type* (`=` *Expression*)? `;`

A *static item* is similar to a constant, except that it represents a precise memory location in the program. All references to the static refer to the same memory location. Static items have the `static` lifetime, which outlives all other lifetimes in a Rust program. Static items do not call `drop` at the end of the program.

The static initializer is a constant expression evaluated at compile time. Static initializers may refer to other statics.

Non-`mut` static items that contain a type that is not interior mutable may be placed in read-only memory.

All access to a static is safe, but there are a number of restrictions on statics:

- The type must have the `Sync` trait bound to allow thread-safe access.
- Constants cannot refer to statics.

The initializer expression must be omitted in an external block, and must be provided for free static items.

Statics & generics

A static item defined in a generic scope (for example in a blanket or default implementation) will result in exactly one static item being defined, as if the static definition was pulled out of the current scope into the module. There will *not* be one item per monomorphization.

This code:

```
use std::sync::atomic::{AtomicUsize, Ordering};

trait Tr {
    fn default_impl() {
        static COUNTER: AtomicUsize = AtomicUsize::new(0);
        println!("default_impl: counter was {}", COUNTER.fetch_add(1,
Ordering::Relaxed));
    }

    fn blanket_impl();
}

struct Ty1 {}
struct Ty2 {}

impl<T> Tr for T {
    fn blanket_impl() {
        static COUNTER: AtomicUsize = AtomicUsize::new(0);
        println!("blanket_impl: counter was {}", COUNTER.fetch_add(1,
Ordering::Relaxed));
    }
}

fn main() {
    <Ty1 as Tr>::default_impl();
    <Ty2 as Tr>::default_impl();
    <Ty1 as Tr>::blanket_impl();
    <Ty2 as Tr>::blanket_impl();
}
```

prints

```
default_impl: counter was 0
default_impl: counter was 1
blanket_impl: counter was 0
blanket_impl: counter was 1
```

Mutable statics

If a static item is declared with the `mut` keyword, then it is allowed to be modified by the program. One of Rust's goals is to make concurrency bugs hard to run into, and this is obviously a very large source of race conditions or other bugs. For this reason, an `unsafe` block is required when either reading or writing a mutable static variable. Care should be taken to ensure that modifications to a mutable static are safe with respect to other threads running in the same process.

Mutable statics are still very useful, however. They can be used with C libraries and can also be bound from C libraries in an `extern` block.

```
static mut LEVELS: u32 = 0;

// This violates the idea of no shared state, and this doesn't internally
// protect against races, so this function is `unsafe`
unsafe fn bump_levels_unsafe1() -> u32 {
    let ret = LEVELS;
    LEVELS += 1;
    return ret;
}

// Assuming that we have an atomic_add function which returns the old value,
// this function is "safe" but the meaning of the return value may not be what
// callers expect, so it's still marked as `unsafe`
unsafe fn bump_levels_unsafe2() -> u32 {
    return atomic_add(&mut LEVELS, 1);
}
```

Mutable statics have the same restrictions as normal statics, except that the type does not have to implement the `Sync` trait.

Using Statics or Consts

It can be confusing whether or not you should use a constant item or a static item. Constants should, in general, be preferred over statics unless one of the following are true:

- Large amounts of data are being stored
- The single-address property of statics is required.
- Interior mutability is required.

Traits

> **Syntax**
>
> *Trait* :
>
> `unsafe`? `trait` IDENTIFIER *GenericParams*? (`:` *TypeParamBounds*?)? *WhereClause*? `{`
>
> *InnerAttribute**
>
> *AssociatedItem**
>
> `}`

A *trait* describes an abstract interface that types can implement. This interface consists of associated items, which come in three varieties:

- functions
- types
- constants

All traits define an implicit type parameter `Self` that refers to "the type that is implementing this interface". Traits may also contain additional type parameters. These type parameters, including `Self`, may be constrained by other traits and so forth as usual.

Traits are implemented for specific types through separate implementations.

Trait functions may omit the function body by replacing it with a semicolon. This indicates that the implementation must define the function. If the trait function defines a body, this definition acts as a default for any implementation which does not override it. Similarly, associated constants may omit the equals sign and expression to indicate implementations must define the constant value. Associated types must never define the type, the type may only be specified in an implementation.

```
// Examples of associated trait items with and without definitions.
trait Example {
    const CONST_NO_DEFAULT: i32;
    const CONST_WITH_DEFAULT: i32 = 99;
    type TypeNoDefault;
    fn method_without_default(&self);
    fn method_with_default(&self) {}
}
```

Trait functions are not allowed to be `const`.

Trait bounds

Generic items may use traits as bounds on their type parameters.

Generic Traits

Type parameters can be specified for a trait to make it generic. These appear after the trait name, using the same syntax used in generic functions.

```
trait Seq<T> {
    fn len(&self) -> u32;
    fn elt_at(&self, n: u32) -> T;
    fn iter<F>(&self, f: F) where F: Fn(T);
}
```

Object Safety

Object safe traits can be the base trait of a trait object. A trait is *object safe* if it has the following qualities (defined in RFC 255):

- All supertraits must also be object safe.
- `Sized` must not be a supertrait. In other words, it must not require `Self: Sized`.
- It must not have any associated constants.
- It must not have any associated types with generics.
- All associated functions must either be dispatchable from a trait object or be explicitly non-dispatchable:
    - Dispatchable functions must:
        - Not have any type parameters (although lifetime parameters are allowed).
        - Be a method that does not use `Self` except in the type of the receiver.
        - Have a receiver with one of the following types:
            - `&Self` (i.e. `&self`)
            - `&mut Self` (i.e `&mut self`)
            - `Box<Self>`
            - `Rc<Self>`
            - `Arc<Self>`
            - `Pin<P>` where `P` is one of the types above
        - Not have an opaque return type; that is,

            - Not be an `async fn` (which has a hidden `Future` type).
            - Not have a return position `impl Trait` type (`fn example(&self) -> impl Trait`).
        - Not have a `where Self: Sized` bound (receiver type of `Self` (i.e. `self`) implies this).
    - Explicitly non-dispatchable functions require:
        - Have a `where Self: Sized` bound (receiver type of `Self` (i.e. `self`) implies this).

```
// Examples of object safe methods.
trait TraitMethods {
    fn by_ref(self: &Self) {}
    fn by_ref_mut(self: &mut Self) {}
    fn by_box(self: Box<Self>) {}
    fn by_rc(self: Rc<Self>) {}
    fn by_arc(self: Arc<Self>) {}
    fn by_pin(self: Pin<&Self>) {}
    fn with_lifetime<'a>(self: &'a Self) {}
    fn nested_pin(self: Pin<Arc<Self>>) {}
}
```

```
// This trait is object-safe, but these methods cannot be dispatched on a trait
object.
trait NonDispatchable {
    // Non-methods cannot be dispatched.
    fn foo() where Self: Sized {}
    // Self type isn't known until runtime.
    fn returns(&self) -> Self where Self: Sized;
    // `other` may be a different concrete type of the receiver.
    fn param(&self, other: Self) where Self: Sized {}
    // Generics are not compatible with vtables.
    fn typed<T>(&self, x: T) where Self: Sized {}
}

struct S;
impl NonDispatchable for S {
    fn returns(&self) -> Self where Self: Sized { S }
}
let obj: Box<dyn NonDispatchable> = Box::new(S);
obj.returns(); // ERROR: cannot call with Self return
obj.param(S);  // ERROR: cannot call with Self parameter
obj.typed(1);  // ERROR: cannot call with generic type
```

```
// Examples of non-object safe traits.
trait NotObjectSafe {
    const CONST: i32 = 1;  // ERROR: cannot have associated const

    fn foo() {}  // ERROR: associated function without Sized
    fn returns(&self) -> Self; // ERROR: Self in return type
    fn typed<T>(&self, x: T) {} // ERROR: has generic type parameters
    fn nested(self: Rc<Box<Self>>) {} // ERROR: nested receiver not yet
supported
}

struct S;
impl NotObjectSafe for S {
    fn returns(&self) -> Self { S }
}
let obj: Box<dyn NotObjectSafe> = Box::new(S); // ERROR
```

```
// Self: Sized traits are not object-safe.
trait TraitWithSize where Self: Sized {}

struct S;
impl TraitWithSize for S {}
let obj: Box<dyn TraitWithSize> = Box::new(S); // ERROR
```

```
// Not object safe if `Self` is a type argument.
trait Super<A> {}
trait WithSelf: Super<Self> where Self: Sized {}

struct S;
impl<A> Super<A> for S {}
impl WithSelf for S {}
let obj: Box<dyn WithSelf> = Box::new(S); // ERROR: cannot use `Self` type
parameter
```

Supertraits

Supertraits are traits that are required to be implemented for a type to implement a specific trait. Furthermore, anywhere a generic or trait object is bounded by a trait, it has access to the associated items of its supertraits.

Supertraits are declared by trait bounds on the `Self` type of a trait and transitively the supertraits of the traits declared in those trait bounds. It is an error for a trait to be its own supertrait.

The trait with a supertrait is called a **subtrait** of its supertrait.

The following is an example of declaring `Shape` to be a supertrait of `Circle`.

```
trait Shape { fn area(&self) -> f64; }
trait Circle : Shape { fn radius(&self) -> f64; }
```

And the following is the same example, except using where clauses.

```
trait Shape { fn area(&self) -> f64; }
trait Circle where Self: Shape { fn radius(&self) -> f64; }
```

This next example gives `radius` a default implementation using the `area` function from `Shape`.

```
trait Circle where Self: Shape {
    fn radius(&self) -> f64 {
        // A = pi * r^2
        // so algebraically,
        // r = sqrt(A / pi)
        (self.area() /std::f64::consts::PI).sqrt()
    }
}
```

This next example calls a supertrait method on a generic parameter.

```
fn print_area_and_radius<C: Circle>(c: C) {
    // Here we call the area method from the supertrait `Shape` of `Circle`.
    println!("Area: {}", c.area());
    println!("Radius: {}", c.radius());
}
```

Similarly, here is an example of calling supertrait methods on trait objects.

```
let circle = Box::new(circle) as Box<dyn Circle>;
let nonsense = circle.radius() * circle.area();
```

Unsafe traits

Traits items that begin with the `unsafe` keyword indicate that *implementing* the trait may be unsafe. It is safe to use a correctly implemented unsafe trait. The trait implementation must also begin with the `unsafe` keyword.

`Sync` and `Send` are examples of unsafe traits.

Parameter patterns

Function or method declarations without a body only allow IDENTIFIER or `_` wild card patterns. `mut` IDENTIFIER is currently allowed, but it is deprecated and will become a hard error in the future.

In the 2015 edition, the pattern for a trait function or method parameter is optional:

```
// 2015 Edition
trait T {
    fn f(i32);  // Parameter identifiers are not required.
}
```

The kinds of patterns for parameters is limited to one of the following:

- IDENTIFIER
- `mut` IDENTIFIER
- _
- `&` IDENTIFIER
- `&&` IDENTIFIER

Beginning in the 2018 edition, function or method parameter patterns are no longer optional. Also, all irrefutable patterns are allowed as long as there is a body. Without a body, the limitations listed above are still in effect.

```
trait T {
    fn f1((a, b): (i32, i32)) {}
    fn f2(_: (i32, i32));  // Cannot use tuple pattern without a body.
}
```

Item visibility

Trait items syntactically allow a *Visibility* annotation, but this is rejected when the trait is validated. This allows items to be parsed with a unified syntax across different contexts where they are used. As an example, an empty `vis` macro fragment specifier can be used for trait items, where the macro rule may be used in other situations where visibility is allowed.

```rust
macro_rules! create_method {
    ($vis:vis $name:ident) => {
        $vis fn $name(&self) {}
    };
}

trait T1 {
    // Empty `vis` is allowed.
    create_method! { method_of_t1 }
}

struct S;

impl S {
    // Visibility is allowed here.
    create_method! { pub method_of_s }
}

impl T1 for S {}

fn main() {
    let s = S;
    s.method_of_t1();
    s.method_of_s();
}
```

Implementations

> **Syntax**
>
> *Implementation* :
>   *InherentImpl* | *TraitImpl*
>
> *InherentImpl* :
>   `impl` *GenericParams*? *Type* *WhereClause*? `{`
>   *InnerAttribute**
>   *AssociatedItem**
>   `}`
>
> *TraitImpl* :
>   `unsafe`? `impl` *GenericParams*? `!`? *TypePath* `for` *Type*
>   *WhereClause*?
>   `{`
>   *InnerAttribute**
>   *AssociatedItem**
>   `}`

An *implementation* is an item that associates items with an *implementing type*. Implementations are defined with the keyword `impl` and contain functions that belong to an instance of the type that is being implemented or to the type statically.

There are two types of implementations:

- inherent implementations
- trait implementations

Inherent Implementations

An inherent implementation is defined as the sequence of the `impl` keyword, generic type declarations, a path to a nominal type, a where clause, and a bracketed set of associable items.

The nominal type is called the *implementing type* and the associable items are the *associated* *items* to the implementing type.

Inherent implementations associate the contained items to the implementing type. Inherent implementations can contain associated functions (including methods) and associated constants. They cannot contain associated type aliases.

The path to an associated item is any path to the implementing type, followed by the associated item's identifier as the final path component.

A type can also have multiple inherent implementations. An implementing type must be defined within the same crate as the original type definition.

```
pub mod color {
    pub struct Color(pub u8, pub u8, pub u8);

    impl Color {
        pub const WHITE: Color = Color(255, 255, 255);
    }
}

mod values {
    use super::color::Color;
    impl Color {
        pub fn red() -> Color {
            Color(255, 0, 0)
        }
    }
}

pub use self::color::Color;
fn main() {
    // Actual path to the implementing type and impl in the same module.
    color::Color::WHITE;

    // Impl blocks in different modules are still accessed through a path to
the type.
    color::Color::red();

    // Re-exported paths to the implementing type also work.
    Color::red();

    // Does not work, because use in `values` is not pub.
    // values::Color::red();
}
```

Trait Implementations

A *trait implementation* is defined like an inherent implementation except that the optional generic type declarations are followed by a trait, followed by the keyword `for`, followed by a path to a nominal type.

The trait is known as the *implemented trait*. The implementing type implements the implemented trait.

A trait implementation must define all non-default associated items declared by the implemented trait, may redefine default associated items defined by the implemented trait, and cannot define any other items.

The path to the associated items is `<` followed by a path to the implementing type followed by `as` followed by a path to the trait followed by `>` as a path component followed by the associated item's path component.

Unsafe traits require the trait implementation to begin with the `unsafe` keyword.

```rust
struct Circle {
    radius: f64,
    center: Point,
}

impl Copy for Circle {}

impl Clone for Circle {
    fn clone(&self) -> Circle { *self }
}

impl Shape for Circle {
    fn draw(&self, s: Surface) { do_draw_circle(s, *self); }
    fn bounding_box(&self) -> BoundingBox {
        let r = self.radius;
        BoundingBox {
            x: self.center.x - r,
            y: self.center.y - r,
            width: 2.0 * r,
            height: 2.0 * r,
        }
    }
}
```

Trait Implementation Coherence

A trait implementation is considered incoherent if either the orphan rules check fails or there are overlapping implementation instances.

Two trait implementations overlap when there is a non-empty intersection of the traits the implementation is for, the implementations can be instantiated with the same type.

Orphan rules

Given `impl<P1..=Pn> Trait<T1..=Tn> for T0`, an `impl` is valid only if at least one of the following is true:

- `Trait` is a local trait
- All of
  - At least one of the types `T0..=Tn` must be a local type. Let `Ti` be the first such type.
  - No uncovered type parameters `P1..=Pn` may appear in `T0..Ti` (excluding `Ti`)

Only the appearance of *uncovered* type parameters is restricted. Note that for the purposes of coherence, fundamental types are special. The `T` in `Box<T>` is not considered covered, and `Box<LocalType>` is considered local.

Generic Implementations

An implementation can take generic parameters, which can be used in the rest of the implementation. Implementation parameters are written directly after the `impl` keyword.

```
impl<T> Seq<T> for Vec<T> {
    /* ... */
}
impl Seq<bool> for u32 {
    /* Treat the integer as a sequence of bits */
}
```

Generic parameters *constrain* an implementation if the parameter appears at least once in one of:

- The implemented trait, if it has one
- The implementing type
- As an associated type in the bounds of a type that contains another parameter that constrains the implementation

Type and const parameters must always constrain the implementation. Lifetimes must constrain the implementation if the lifetime is used in an associated type.

Examples of constraining situations:

```
// T constrains by being an argument to GenericTrait.
impl<T> GenericTrait<T> for i32 { /* ... */ }

// T constrains by being an argument to GenericStruct
impl<T> Trait for GenericStruct<T> { /* ... */ }

// Likewise, N constrains by being an argument to ConstGenericStruct
impl<const N: usize> Trait for ConstGenericStruct<N> { /* ... */ }

// T constrains by being in an associated type in a bound for type `U` which is
// itself a generic parameter constraining the trait.
impl<T, U> GenericTrait<U> for u32 where U: HasAssocType<Ty = T> { /* ... */ }

// Like previous, except the type is `(U, isize)`. `U` appears inside the type
// that includes `T`, and is not the type itself.
impl<T, U> GenericStruct<U> where (U, isize): HasAssocType<Ty = T> { /* ... */
}
```

Examples of non-constraining situations:

```
// The rest of these are errors, since they have type or const parameters that
// do not constrain.

// T does not constrain since it does not appear at all.
impl<T> Struct { /* ... */ }

// N does not constrain for the same reason.
impl<const N: usize> Struct { /* ... */ }

// Usage of T inside the implementation does not constrain the impl.
impl<T> Struct {
    fn uses_t(t: &T) { /* ... */ }
}

// T is used as an associated type in the bounds for U, but U does not
constrain.
impl<T, U> Struct where U: HasAssocType<Ty = T> { /* ... */ }

// T is used in the bounds, but not as an associated type, so it does not
constrain.
impl<T, U> GenericTrait<U> for u32 where U: GenericTrait<T> {}
```

Example of an allowed unconstraining lifetime parameter:

```
impl<'a> Struct {}
```

Example of a disallowed unconstraining lifetime parameter:

```
impl<'a> HasAssocType for Struct {
    type Ty = &'a Struct;
}
```

Attributes on Implementations

Implementations may contain outer attributes before the `impl` keyword and inner attributes inside the brackets that contain the associated items. Inner attributes must come before any associated items. The attributes that have meaning here are `cfg`, `deprecated`, `doc`, and the lint check attributes.

External blocks

> **Syntax**
>
> *ExternBlock* :
>
> `unsafe`? `extern` *Abi*? `{`
>
> *InnerAttribute**
>
> *ExternalItem**
>
> `}`
>
> *ExternalItem* :
>
> *OuterAttribute** (
>
> *MacroInvocationSemi*
>
> | (*Visibility*? (*StaticItem* | *Function*))
>
>)

External blocks provide *declarations* of items that are not *defined* in the current crate and are the basis of Rust's foreign function interface. These are akin to unchecked imports.

Two kinds of item *declarations* are allowed in external blocks: functions and statics. Calling functions or accessing statics that are declared in external blocks is only allowed in an `unsafe` context.

The `unsafe` keyword is syntactically allowed to appear before the `extern` keyword, but it is rejected at a semantic level. This allows macros to consume the syntax and make use of the `unsafe` keyword, before removing it from the token stream.

Functions

Functions within external blocks are declared in the same way as other Rust functions, with the exception that they must not have a body and are instead terminated by a semicolon. Patterns are not allowed in parameters, only IDENTIFIER or `_` may be used. Function qualifiers (`const` , `async` , `unsafe` , and `extern`) are not allowed.

Functions within external blocks may be called by Rust code, just like functions defined in Rust. The Rust compiler automatically translates between the Rust ABI and the foreign ABI.

A function declared in an extern block is implicitly `unsafe` . When coerced to a function pointer, a function declared in an extern block has type `unsafe extern "abi" for<'l1, ..., 'lm> fn(A1, ..., An) -> R`, where `'l1`, ... `'lm` are its lifetime parameters, `A1`, ..., `An` are the declared types of its parameters and `R` is the declared return type.

Statics

Statics within external blocks are declared in the same way as statics outside of external blocks, except that they do not have an expression initializing their value. It is `unsafe` to access a static item declared in an extern block, whether or not it's mutable, because there is nothing guaranteeing that the bit pattern at the static's memory is valid for the type it is declared with, since some arbitrary (e.g. C) code is in charge of initializing the static.

Extern statics can be either immutable or mutable just like statics outside of external blocks. An immutable static *must* be initialized before any Rust code is executed. It is not enough for the static to be initialized before Rust code reads from it.

ABI

By default external blocks assume that the library they are calling uses the standard C ABI on the specific platform. Other ABIs may be specified using an `abi` string, as shown here:

```
// Interface to the Windows API
extern "stdcall" { }
```

There are three ABI strings which are cross-platform, and which all compilers are guaranteed to support:

- `extern "Rust"` -- The default ABI when you write a normal `fn foo()` in any Rust code.
- `extern "C"` -- This is the same as `extern fn foo()`; whatever the default your C compiler supports.
- `extern "system"` -- Usually the same as `extern "C"`, except on Win32, in which case it's `"stdcall"`, or what you should use to link to the Windows API itself

There are also some platform-specific ABI strings:

- `extern "cdecl"` -- The default for x86_32 C code.
- `extern "stdcall"` -- The default for the Win32 API on x86_32.
- `extern "win64"` -- The default for C code on x86_64 Windows.

- `extern "sysv64"` -- The default for C code on non-Windows x86_64.
- `extern "aapcs"` -- The default for ARM.
- `extern "fastcall"` -- The `fastcall` ABI -- corresponds to MSVC's `__fastcall` and GCC and clang's `__attribute__((fastcall))`
- `extern "vectorcall"` -- The `vectorcall` ABI -- corresponds to MSVC's `__vectorcall` and clang's `__attribute__((vectorcall))`
- `extern "thiscall"` -- The default for C++ member functions on MSVC -- corresponds to MSVC's `__thiscall` and GCC and clang's `__attribute__((thiscall))`
- `extern "efiapi"` -- The ABI used for UEFI functions.

Variadic functions

Functions within external blocks may be variadic by specifying `...` as the last argument. There must be at least one parameter before the variadic parameter. The variadic parameter may optionally be specified with an identifier.

```
extern "C" {
    fn foo(x: i32, ...);
    fn with_name(format: *const u8, args: ...);
}
```

Attributes on extern blocks

The following attributes control the behavior of external blocks.

The `link` attribute

The *`link` attribute* specifies the name of a native library that the compiler should link with for the items within an `extern` block. It uses the *MetaListNameValueStr* syntax to specify its inputs. The `name` key is the name of the native library to link. The `kind` key is an optional value which specifies the kind of library with the following possible values:

- `dylib` — Indicates a dynamic library. This is the default if `kind` is not specified.
- `static` — Indicates a static library.
- `framework` — Indicates a macOS framework. This is only valid for macOS targets.
- `raw-dylib` — Indicates a dynamic library where the compiler will generate an import

library to link against (see `dylib` versus `raw-dylib` below for details). This is only valid for Windows targets.

The `name` key must be included if `kind` is specified.

The optional `modifiers` argument is a way to specify linking modifiers for the library to link. Modifiers are specified as a comma-delimited string with each modifier prefixed with either a `+` or `-` to indicate that the modifier is enabled or disabled, respectively. Specifying multiple `modifiers` arguments in a single `link` attribute, or multiple identical modifiers in the same `modifiers` argument is not currently supported.
Example: `#[link(name = "mylib", kind = "static", modifiers = "+whole-archive")]`.

The `wasm_import_module` key may be used to specify the WebAssembly module name for the items within an `extern` block when importing symbols from the host environment. The default module name is `env` if `wasm_import_module` is not specified.

```
#[link(name = "crypto")]
extern {
    // …
}

#[link(name = "CoreFoundation", kind = "framework")]
extern {
    // …
}

#[link(wasm_import_module = "foo")]
extern {
    // …
}
```

It is valid to add the `link` attribute on an empty extern block. You can use this to satisfy the linking requirements of extern blocks elsewhere in your code (including upstream crates) instead of adding the attribute to each extern block.

Linking modifiers: `bundle`

This modifier is only compatible with the `static` linking kind. Using any other kind will result in a compiler error.

When building a rlib or staticlib `+bundle` means that the native static library will be packed into the rlib or staticlib archive, and then retrieved from there during linking of the final binary.

When building a rlib `-bundle` means that the native static library is registered as a dependency of that rlib "by name", and object files from it are included only during linking of the final binary, the file search by that name is also performed during final linking.
When building a staticlib `-bundle` means that the native static library is simply not included into the archive and some higher level build system will need to add it later during linking of the final binary.

This modifier has no effect when building other targets like executables or dynamic libraries.

The default for this modifier is `+bundle`.

More implementation details about this modifier can be found in `bundle` documentation for rustc.

Linking modifiers: `whole-archive`

This modifier is only compatible with the `static` linking kind. Using any other kind will result in a compiler error.

`+whole-archive` means that the static library is linked as a whole archive without throwing any object files away.

The default for this modifier is `-whole-archive`.

More implementation details about this modifier can be found in `whole-archive` documentation for rustc.

Linking modifiers: `verbatim`

This modifier is compatible with all linking kinds.

`+verbatim` means that rustc itself won't add any target-specified library prefixes or suffixes (like `lib` or `.a`) to the library name, and will try its best to ask for the same thing from the linker.

`-verbatim` means that rustc will either add a target-specific prefix and suffix to the library name before passing it to linker, or won't prevent linker from implicitly adding it.

The default for this modifier is `-verbatim`.

More implementation details about this modifier can be found in `verbatim` documentation for rustc.

`dylib` versus `raw-dylib`

On Windows, linking against a dynamic library requires that an import library is provided to the linker: this is a special static library that declares all of the symbols exported by the dynamic library in such a way that the linker knows that they have to be dynamically loaded at runtime.

Specifying `kind = "dylib"` instructs the Rust compiler to link an import library based on the `name` key. The linker will then use its normal library resolution logic to find that import library. Alternatively, specifying `kind = "raw-dylib"` instructs the compiler to generate an import library during compilation and provide that to the linker instead.

`raw-dylib` is only supported on Windows. Using it when targeting other platforms will result in a compiler error.

The `import_name_type` key

On x86 Windows, names of functions are "decorated" (i.e., have a specific prefix and/or suffix added) to indicate their calling convention. For example, a `stdcall` calling convention function with the name `fn1` that has no arguments would be decorated as `_fn1@0`. However, the PE Format does also permit names to have no prefix or be undecorated. Additionally, the MSVC and GNU toolchains use different decorations for the same calling conventions which means, by default, some Win32 functions cannot be called using the `raw-dylib` link kind via the GNU toolchain.

To allow for these differences, when using the `raw-dylib` link kind you may also specify the `import_name_type` key with one of the following values to change how functions are named in the generated import library:

- `decorated`: The function name will be fully-decorated using the MSVC toolchain format.
- `noprefix`: The function name will be decorated using the MSVC toolchain format, but skipping the leading `?`, `@`, or optionally `_`.
- `undecorated`: The function name will not be decorated.

If the `import_name_type` key is not specified, then the function name will be fully-decorated using the target toolchain's format.

Variables are never decorated and so the `import_name_type` key has no effect on how they are named in the generated import library.

The `import_name_type` key is only supported on x86 Windows. Using it when targeting other platforms will result in a compiler error.

The `link_name` attribute

The *`link_name` attribute* may be specified on declarations inside an `extern` block to indicate the symbol to import for the given function or static. It uses the *MetaNameValueStr* syntax to specify the name of the symbol.

```
extern {
    #[link_name = "actual_symbol_name"]
    fn name_in_rust();
}
```

Using this attribute with the `link_ordinal` attribute will result in a compiler error.

The `link_ordinal` attribute

The *`link_ordinal` attribute* can be applied on declarations inside an `extern` block to indicate the numeric ordinal to use when generating the import library to link against. An ordinal is a unique number per symbol exported by a dynamic library on Windows and can be used when the library is being loaded to find that symbol rather than having to look it up by name.

> ⚠️ Warning: `link_ordinal` should only be used in cases where the ordinal of the symbol is known to be stable: if the ordinal of a symbol is not explicitly set when its containing binary is built then one will be automatically assigned to it, and that assigned ordinal may change between builds of the binary.

```
#[link(name = "exporter", kind = "raw-dylib")]
extern "stdcall" {
    #[link_ordinal(15)]
    fn imported_function_stdcall(i: i32);
}
```

This attribute is only used with the `raw-dylib` linking kind. Using any other kind will result in a compiler error.

Using this attribute with the `link_name` attribute will result in a compiler error.

Attributes on function parameters

Attributes on extern function parameters follow the same rules and restrictions as regular function parameters.

Generic parameters

Syntax

GenericParams :
`<` `>`
| `<` (*GenericParam* `,`)* *GenericParam* `,` ? `>`

GenericParam :
*OuterAttribute** (*LifetimeParam* | *TypeParam* | *ConstParam*)

LifetimeParam :
LIFETIME_OR_LABEL (`:` *LifetimeBounds*)?

TypeParam :
IDENTIFIER(`:` *TypeParamBounds*?)? (`=` *Type*)?

ConstParam:
`const` IDENTIFIER `:` *Type* (`=` *Block* | IDENTIFIER | -?LITERAL)?

Functions, type aliases, structs, enumerations, unions, traits, and implementations may be *parameterized* by types, constants, and lifetimes. These parameters are listed in angle brackets (`<...>`), usually immediately after the name of the item and before its definition. For implementations, which don't have a name, they come directly after `impl` . The order of generic parameters is restricted to lifetime parameters and then type and const parameters intermixed.

Some examples of items with type, const, and lifetime parameters:

```
fn foo<'a, T>() {}
trait A<U> {}
struct Ref<'a, T> where T: 'a { r: &'a T }
struct InnerArray<T, const N: usize>([T; N]);
struct EitherOrderWorks<const N: bool, U>(U);
```

Generic parameters are in scope within the item definition where they are declared. They are not in scope for items declared within the body of a function as described in item declarations.

References, raw pointers, arrays, slices, tuples, and function pointers have lifetime or type parameters as well, but are not referred to with path syntax.

Const generics

Const generic parameters allow items to be generic over constant values. The const identifier introduces a name for the constant parameter, and all instances of the item must be instantiated with a value of the given type.

The only allowed types of const parameters are `u8`, `u16`, `u32`, `u64`, `u128`, `usize`, `i8`, `i16`, `i32`, `i64`, `i128`, `isize`, `char` and `bool`.

Const parameters can be used anywhere a const item can be used, with the exception that when used in a type or array repeat expression, it must be standalone (as described below). That is, they are allowed in the following places:

1. As an applied const to any type which forms a part of the signature of the item in question.
2. As part of a const expression used to define an associated const, or as a parameter to an associated type.
3. As a value in any runtime expression in the body of any functions in the item.
4. As a parameter to any type used in the body of any functions in the item.
5. As a part of the type of any fields in the item.

```
// Examples where const generic parameters can be used.

// Used in the signature of the item itself.
fn foo<const N: usize>(arr: [i32; N]) {
    // Used as a type within a function body.
    let x: [i32; N];
    // Used as an expression.
    println!("{}", N * 2);
}

// Used as a field of a struct.
struct Foo<const N: usize>([i32; N]);

impl<const N: usize> Foo<N> {
    // Used as an associated constant.
    const CONST: usize = N * 4;
}

trait Trait {
    type Output;
}

impl<const N: usize> Trait for Foo<N> {
    // Used as an associated type.
    type Output = [i32; N];
}
```

```
// Examples where const generic parameters cannot be used.
fn foo<const N: usize>() {
    // Cannot use in item definitions within a function body.
    const BAD_CONST: [usize; N] = [1; N];
    static BAD_STATIC: [usize; N] = [1; N];
    fn inner(bad_arg: [usize; N]) {
        let bad_value = N * 2;
    }
    type BadAlias = [usize; N];
    struct BadStruct([usize; N]);
}
```

As a further restriction, const parameters may only appear as a standalone argument inside of a type or array repeat expression. In those contexts, they may only be used as a single segment path expression, possibly inside a block (such as `N` or `{N}`). That is, they cannot be combined with other expressions.

```
// Examples where const parameters may not be used.

// Not allowed to combine in other expressions in types, such as the
// arithmetic expression in the return type here.
fn bad_function<const N: usize>() -> [u8; {N + 1}] {
    // Similarly not allowed for array repeat expressions.
    [1; {N + 1}]
}
```

A const argument in a path specifies the const value to use for that item. The argument must be a const expression of the type ascribed to the const parameter. The const expression must be a block expression (surrounded with braces) unless it is a single path segment (an IDENTIFIER) or a literal (with a possibly leading `-` token).

Note: This syntactic restriction is necessary to avoid requiring infinite lookahead when parsing an expression inside of a type.

```
fn double<const N: i32>() {
    println!("doubled: {}", N * 2);
}

const SOME_CONST: i32 = 12;

fn example() {
    // Example usage of a const argument.
    double::<9>();
    double::<-123>();
    double::<{7 + 8}>();
    double::<SOME_CONST>();
    double::<{ SOME_CONST + 5 }>();
}
```

When there is ambiguity if a generic argument could be resolved as either a type or const argument, it is always resolved as a type. Placing the argument in a block expression can force it to be interpreted as a const argument.

```
type N = u32;
struct Foo<const N: usize>;
// The following is an error, because `N` is interpreted as the type alias `N`.
fn foo<const N: usize>() -> Foo<N> { todo!() } // ERROR
// Can be fixed by wrapping in braces to force it to be interpreted as the `N`
// const parameter:
fn bar<const N: usize>() -> Foo<{ N }> { todo!() } // ok
```

Unlike type and lifetime parameters, const parameters can be declared without being used inside of a parameterized item, with the exception of implementations as described in generic implementations:

```
// ok
struct Foo<const N: usize>;
enum Bar<const M: usize> { A, B }

// ERROR: unused parameter
struct Baz<T>;
struct Biz<'a>;
struct Unconstrained;
impl<const N: usize> Unconstrained {}
```

When resolving a trait bound obligation, the exhaustiveness of all implementations of const parameters is not considered when determining if the bound is satisfied. For example, in the following, even though all possible const values for the `bool` type are implemented, it is still an error that the trait bound is not satisfied:

```
struct Foo<const B: bool>;
trait Bar {}
impl Bar for Foo<true> {}
impl Bar for Foo<false> {}

fn needs_bar(_: impl Bar) {}
fn generic<const B: bool>() {
    let v = Foo::<B>;
    needs_bar(v); // ERROR: trait bound `Foo<B>: Bar` is not satisfied
}
```

Where clauses

Syntax

WhereClause :
   `where` (*WhereClauseItem* `,`)* *WhereClauseItem* ?

WhereClauseItem :
      LifetimeWhereClauseItem
   | *TypeBoundWhereClauseItem*

LifetimeWhereClauseItem :
   Lifetime `:` *LifetimeBounds*

TypeBoundWhereClauseItem :
   ForLifetimes? *Type* `:` *TypeParamBounds*?

Where clauses provide another way to specify bounds on type and lifetime parameters as well as a way to specify bounds on types that aren't type parameters.

The `for` keyword can be used to introduce higher-ranked lifetimes. It only allows *LifetimeParam* parameters.

```
struct A<T>
where
    T: Iterator,            // Could use A<T: Iterator> instead
    T::Item: Copy,          // Bound on an associated type
    String: PartialEq<T>,   // Bound on `String`, using the type parameter
    i32: Default,           // Allowed, but not useful
{
    f: T,
}
```

Attributes

Generic lifetime and type parameters allow attributes on them. There are no built-in attributes that do anything in this position, although custom derive attributes may give meaning to it.

This example shows using a custom derive attribute to modify the meaning of a generic parameter.

```
// Assume that the derive for MyFlexibleClone declared `my_flexible_clone` as
// an attribute it understands.
#[derive(MyFlexibleClone)]
struct Foo<#[my_flexible_clone(unbounded)] H> {
    a: *const H
}
```

Associated Items

Syntax

AssociatedItem :
  *OuterAttribute** (
    MacroInvocationSemi
   | (*Visibility*? (*TypeAlias* | *ConstantItem* | *Function*))
)

Associated Items are the items declared in traits or defined in implementations. They are called this because they are defined on an associate type — the type in the implementation. They are a subset of the kinds of items you can declare in a module. Specifically, there are associated functions (including methods), associated types, and associated constants.

Associated items are useful when the associated item logically is related to the associating item. For example, the `is_some` method on `Option` is intrinsically related to Options, so should be associated.

Every associated item kind comes in two varieties: definitions that contain the actual implementation and declarations that declare signatures for definitions.

It is the declarations that make up the contract of traits and what is available on generic types.

Associated functions and methods

Associated functions are functions associated with a type.

An *associated function declaration* declares a signature for an associated function definition. It is written as a function item, except the function body is replaced with a `;`.

The identifier is the name of the function. The generics, parameter list, return type, and where clause of the associated function must be the same as the associated function declarations's.

An *associated function definition* defines a function associated with another type. It is written the same as a function item.

An example of a common associated function is a `new` function that returns a value of the type the associated function is associated with.

```
struct Struct {
    field: i32
}

impl Struct {
    fn new() -> Struct {
        Struct {
            field: 0i32
        }
    }
}

fn main () {
    let _struct = Struct::new();
}
```

When the associated function is declared on a trait, the function can also be called with a path that is a path to the trait appended by the name of the trait. When this happens, it is substituted for `<_ as Trait>::function_name`.

```
trait Num {
    fn from_i32(n: i32) -> Self;
}

impl Num for f64 {
    fn from_i32(n: i32) -> f64 { n as f64 }
}

// These 4 are all equivalent in this case.
let _: f64 = Num::from_i32(42);
let _: f64 = <_ as Num>::from_i32(42);
let _: f64 = <f64 as Num>::from_i32(42);
let _: f64 = f64::from_i32(42);
```

Methods

Associated functions whose first parameter is named `self` are called *methods* and may be invoked using the method call operator, for example, `x.foo()`, as well as the usual function call notation.

If the type of the `self` parameter is specified, it is limited to types resolving to one generated by the following grammar (where `'lt` denotes some arbitrary lifetime):

```
P = &'lt S | &'lt mut S | Box<S> | Rc<S> | Arc<S> | Pin<P>
S = Self | P
```

The `Self` terminal in this grammar denotes a type resolving to the implementing type. This can also include the contextual type alias `Self`, other type aliases, or associated type projections resolving to the implementing type.

```
// Examples of methods implemented on struct `Example`.
struct Example;
type Alias = Example;
trait Trait { type Output; }
impl Trait for Example { type Output = Example; }
impl Example {
    fn by_value(self: Self) {}
    fn by_ref(self: &Self) {}
    fn by_ref_mut(self: &mut Self) {}
    fn by_box(self: Box<Self>) {}
    fn by_rc(self: Rc<Self>) {}
    fn by_arc(self: Arc<Self>) {}
    fn by_pin(self: Pin<&Self>) {}
    fn explicit_type(self: Arc<Example>) {}
    fn with_lifetime<'a>(self: &'a Self) {}
    fn nested<'a>(self: &mut &'a Arc<Rc<Box<Alias>>>) {}
    fn via_projection(self: <Example as Trait>::Output) {}
}
```

Shorthand syntax can be used without specifying a type, which have the following equivalents:

Shorthand	Equivalent
`self`	`self: Self`
`&'lifetime self`	`self: &'lifetime Self`
`&'lifetime mut self`	`self: &'lifetime mut Self`

> **Note**: Lifetimes can be, and usually are, elided with this shorthand.

If the `self` parameter is prefixed with `mut`, it becomes a mutable variable, similar to regular parameters using a `mut` identifier pattern. For example:

```
trait Changer: Sized {
    fn change(mut self) {}
    fn modify(mut self: Box<Self>) {}
}
```

As an example of methods on a trait, consider the following:

```
trait Shape {
    fn draw(&self, surface: Surface);
    fn bounding_box(&self) -> BoundingBox;
}
```

This defines a trait with two methods. All values that have implementations of this trait while the trait is in scope can have their `draw` and `bounding_box` methods called.

```
struct Circle {
    // ...
}

impl Shape for Circle {
    // ...
}

let circle_shape = Circle::new();
let bounding_box = circle_shape.bounding_box();
```

> **Edition Differences**: In the 2015 edition, it is possible to declare trait methods with anonymous parameters (e.g. `fn foo(u8)`). This is deprecated and an error as of the 2018 edition. All parameters must have an argument name.

Attributes on method parameters

Attributes on method parameters follow the same rules and restrictions as regular function parameters.

Associated Types

Associated types are type aliases associated with another type. Associated types cannot be defined in inherent implementations nor can they be given a default implementation in traits.

An *associated type declaration* declares a signature for associated type definitions. It is written in one of the following forms, where `Assoc` is the name of the associated type, `Params` is a comma-separated list of type, lifetime or const parameters, `Bounds` is a plus-separated list of trait bounds that the associated type must meet, and `WhereBounds` is a comma-separated list of bounds that the parameters must meet:

```
type Assoc;
type Assoc: Bounds;
type Assoc<Params>;
type Assoc<Params>: Bounds;
type Assoc<Params> where WhereBounds;
type Assoc<Params>: Bounds where WhereBounds;
```

The identifier is the name of the declared type alias. The optional trait bounds must be fulfilled by the implementations of the type alias. There is an implicit `Sized` bound on associated types that can be relaxed using the special `?Sized` bound.

An *associated type definition* defines a type alias for the implementation of a trait on a type. They are written similarly to an *associated type declaration*, but cannot contain `Bounds`, but instead must contain a `Type`:

```
type Assoc = Type;
type Assoc<Params> = Type; // the type `Type` here may reference `Params`
type Assoc<Params> = Type where WhereBounds;
type Assoc<Params> where WhereBounds = Type; // deprecated, prefer the form
above
```

If a type `Item` has an associated type `Assoc` from a trait `Trait`, then `<Item as Trait>::Assoc` is a type that is an alias of the type specified in the associated type definition. Furthermore, if `Item` is a type parameter, then `Item::Assoc` can be used in type parameters.

Associated types may include generic parameters and where clauses; these are often referred to as *generic associated types*, or *GATs*. If the type `Thing` has an associated type `Item` from a trait `Trait` with the generics `<'a>`, the type can be named like `<Thing as Trait>::Item<'x>`, where `'x` is some lifetime in scope. In this case, `'x` will be used wherever `'a` appears in the associated type definitions on impls.

```rust
trait AssociatedType {
    // Associated type declaration
    type Assoc;
}

struct Struct;

struct OtherStruct;

impl AssociatedType for Struct {
    // Associated type definition
    type Assoc = OtherStruct;
}

impl OtherStruct {
    fn new() -> OtherStruct {
        OtherStruct
    }
}

fn main() {
    // Usage of the associated type to refer to OtherStruct as <Struct as
AssociatedType>::Assoc
    let _other_struct: OtherStruct = <Struct as AssociatedType>::Assoc::new();
}
```

An example of associated types with generics and where clauses:

```
struct ArrayLender<'a, T>(&'a mut [T; 16]);

trait Lend {
    // Generic associated type declaration
    type Lender<'a> where Self: 'a;
    fn lend<'a>(&'a mut self) -> Self::Lender<'a>;
}

impl<T> Lend for [T; 16] {
    // Generic associated type definition
    type Lender<'a> = ArrayLender<'a, T> where Self: 'a;

    fn lend<'a>(&'a mut self) -> Self::Lender<'a> {
        ArrayLender(self)
    }
}

fn borrow<'a, T: Lend>(array: &'a mut T) -> <T as Lend>::Lender<'a> {
    array.lend()
}


fn main() {
    let mut array = [0usize; 16];
    let lender = borrow(&mut array);
}
```

Associated Types Container Example

Consider the following example of a `Container` trait. Notice that the type is available for use in the method signatures:

```
trait Container {
    type E;
    fn empty() -> Self;
    fn insert(&mut self, elem: Self::E);
}
```

In order for a type to implement this trait, it must not only provide implementations for every method, but it must specify the type `E`. Here's an implementation of `Container` for the standard library type `Vec`:

```
impl<T> Container for Vec<T> {
    type E = T;
    fn empty() -> Vec<T> { Vec::new() }
    fn insert(&mut self, x: T) { self.push(x); }
}
```

Relationship between `Bounds` and `WhereBounds`

In this example:

```
trait Example {
    type Output<T>: Ord where T: Debug;
}
```

Given a reference to the associated type like `<X as Example>::Output<Y>`, the associated type itself must be `Ord`, and the type `Y` must be `Debug`.

Required where clauses on generic associated types

Generic associated type declarations on traits currently may require a list of where clauses, dependent on functions in the trait and how the GAT is used. These rules may be loosened in the future; updates can be found on the generic associated types initiative repository.

In a few words, these where clauses are required in order to maximize the allowed definitions of the associated type in impls. To do this, any clauses that *can be proven to hold* on functions (using the parameters of the function or trait) where a GAT appears as an input or output must also be written on the GAT itself.

```
trait LendingIterator {
    type Item<'x> where Self: 'x;
    fn next<'a>(&'a mut self) -> Self::Item<'a>;
}
```

In the above, on the `next` function, we can prove that `Self: 'a`, because of the implied bounds from `&'a mut self`; therefore, we must write the equivalent bound on the GAT itself: `where Self: 'x`.

When there are multiple functions in a trait that use the GAT, then the *intersection* of the bounds from the different functions are used, rather than the union.

```
trait Check<T> {
    type Checker<'x>;
    fn create_checker<'a>(item: &'a T) -> Self::Checker<'a>;
    fn do_check(checker: Self::Checker<'_>);
}
```

In this example, no bounds are required on the `type Checker<'a>;`. While we know that `T: 'a` on `create_checker`, we do not know that on `do_check`. However, if `do_check` was commented out, then the `where T: 'x` bound would be required on `Checker`.

The bounds on associated types also propagate required where clauses.

```
trait Iterable {
    type Item<'a> where Self: 'a;
    type Iterator<'a>: Iterator<Item = Self::Item<'a>> where Self: 'a;
    fn iter<'a>(&'a self) -> Self::Iterator<'a>;
}
```

Here, `where Self: 'a` is required on `Item` because of `iter`. However, `Item` is used in the bounds of `Iterator`, the `where Self: 'a` clause is also required there.

Finally, any explicit uses of `'static` on GATs in the trait do not count towards the required bounds.

```
trait StaticReturn {
    type Y<'a>;
    fn foo(&self) -> Self::Y<'static>;
}
```

Associated Constants

Associated constants are constants associated with a type.

An *associated constant declaration* declares a signature for associated constant definitions. It is written as `const`, then an identifier, then `:`, then a type, finished by a `;`.

The identifier is the name of the constant used in the path. The type is the type that the definition has to implement.

An *associated constant definition* defines a constant associated with a type. It is written the same as a constant item.

Associated constant definitions undergo constant evaluation only when referenced. Further, definitions that include generic parameters are evaluated after monomorphization.

```
struct Struct;
struct GenericStruct<const ID: i32>;

impl Struct {
    // Definition not immediately evaluated
    const PANIC: () = panic!("compile-time panic");
}

impl<const ID: i32> GenericStruct<ID> {
    // Definition not immediately evaluated
    const NON_ZERO: () = if ID == 0 {
        panic!("contradiction")
    };
}

fn main() {
    // Referencing Struct::PANIC causes compilation error
    let _ = Struct::PANIC;

    // Fine, ID is not 0
    let _ = GenericStruct::<1>::NON_ZERO;

    // Compilation error from evaluating NON_ZERO with ID=0
    let _ = GenericStruct::<0>::NON_ZERO;
}
```

Associated Constants Examples

A basic example:

```
trait ConstantId {
    const ID: i32;
}

struct Struct;

impl ConstantId for Struct {
    const ID: i32 = 1;
}

fn main() {
    assert_eq!(1, Struct::ID);
}
```

Using default values:

```
trait ConstantIdDefault {
    const ID: i32 = 1;
}

struct Struct;
struct OtherStruct;

impl ConstantIdDefault for Struct {}

impl ConstantIdDefault for OtherStruct {
    const ID: i32 = 5;
}

fn main() {
    assert_eq!(1, Struct::ID);
    assert_eq!(5, OtherStruct::ID);
}
```

Attributes

> **Syntax**
>
> *InnerAttribute* :
>   `#` `!` `[` *Attr* `]`
>
> *OuterAttribute* :
>   `#` `[` *Attr* `]`
>
> *Attr* :
>   *SimplePath* *AttrInput*?
>
> *AttrInput* :
>   *DelimTokenTree*
> | `=` *Expression*

An *attribute* is a general, free-form metadatum that is interpreted according to name, convention, language, and compiler version. Attributes are modeled on Attributes in ECMA-335, with the syntax coming from ECMA-334 (C#).

Inner attributes, written with a bang (`!`) after the hash (`#`), apply to the item that the attribute is declared within. *Outer attributes*, written without the bang after the hash, apply to the thing that follows the attribute.

The attribute consists of a path to the attribute, followed by an optional delimited token tree whose interpretation is defined by the attribute. Attributes other than macro attributes also allow the input to be an equals sign (`=`) followed by an expression. See the meta item syntax below for more details.

Attributes can be classified into the following kinds:

- Built-in attributes
- Macro attributes
- Derive macro helper attributes
- Tool attributes

Attributes may be applied to many things in the language:

- All item declarations accept outer attributes while external blocks, functions, implementations, and modules accept inner attributes.
- Most statements accept outer attributes (see Expression Attributes for limitations on

expression statements).
- Block expressions accept outer and inner attributes, but only when they are the outer expression of an expression statement or the final expression of another block expression.
- Enum variants and struct and union fields accept outer attributes.
- Match expression arms accept outer attributes.
- Generic lifetime or type parameter accept outer attributes.
- Expressions accept outer attributes in limited situations, see Expression Attributes for details.
- Function, closure and function pointer parameters accept outer attributes. This includes attributes on variadic parameters denoted with `...` in function pointers and external blocks.

Some examples of attributes:

```rust
// General metadata applied to the enclosing module or crate.
#![crate_type = "lib"]

// A function marked as a unit test
#[test]
fn test_foo() {
    /* ... */
}

// A conditionally-compiled module
#[cfg(target_os = "linux")]
mod bar {
    /* ... */
}

// A lint attribute used to suppress a warning/error
#[allow(non_camel_case_types)]
type int8_t = i8;

// Inner attribute applies to the entire function.
fn some_unused_variables() {
  #![allow(unused_variables)]

  let x = ();
  let y = ();
  let z = ();
}
```

Meta Item Attribute Syntax

A "meta item" is the syntax used for the *Attr* rule by most built-in attributes. It has the following grammar:

Syntax

MetaItem :
  SimplePath
  | *SimplePath* `=` *Expression*
  | *SimplePath* `(` *MetaSeq*? `)`

MetaSeq :
  MetaItemInner (`,` MetaItemInner)* `,` ?

MetaItemInner :
  MetaItem
  | *Expression*

Expressions in meta items must macro-expand to literal expressions, which must not include integer or float type suffixes. Expressions which are not literal expressions will be syntactically accepted (and can be passed to proc-macros), but will be rejected after parsing.

Note that if the attribute appears within another macro, it will be expanded after that outer macro. For example, the following code will expand the `Serialize` proc-macro first, which must preserve the `include_str!` call in order for it to be expanded:

```
#[derive(Serialize)]
struct Foo {
    #[doc = include_str!("x.md")]
    x: u32
}
```

Additionally, macros in attributes will be expanded only after all other attributes applied to the item:

```
#[macro_attr1] // expanded first
#[doc = mac!()] // `mac!` is expanded fourth.
#[macro_attr2] // expanded second
#[derive(MacroDerive1, MacroDerive2)] // expanded third
fn foo() {}
```

Various built-in attributes use different subsets of the meta item syntax to specify their inputs. The following grammar rules show some commonly used forms:

Syntax

MetaWord:
  IDENTIFIER

MetaNameValueStr:
  IDENTIFIER `=` (STRING_LITERAL | RAW_STRING_LITERAL)

MetaListPaths:
  IDENTIFIER `(` (*SimplePath* (`,` *SimplePath*)* `,`?)? `)`

MetaListIdents:
  IDENTIFIER `(` (IDENTIFIER (`,` IDENTIFIER)* `,`?)? `)`

MetaListNameValueStr:
  IDENTIFIER `(` (*MetaNameValueStr* (`,` *MetaNameValueStr*)* `,`?)? `)`

Some examples of meta items are:

Style	Example
MetaWord	`no_std`
MetaNameValueStr	`doc = "example"`
MetaListPaths	`allow(unused, clippy::inline_always)`
MetaListIdents	`macro_use(foo, bar)`
MetaListNameValueStr	`link(name = "CoreFoundation", kind = "framework")`

Active and inert attributes

An attribute is either active or inert. During attribute processing, *active attributes* remove themselves from the thing they are on while *inert attributes* stay on.

The `cfg` and `cfg_attr` attributes are active. The `test` attribute is inert when compiling for tests and active otherwise. Attribute macros are active. All other attributes are inert.

Tool attributes

The compiler may allow attributes for external tools where each tool resides in its own namespace in the tool prelude. The first segment of the attribute path is the name of the tool, with one or more additional segments whose interpretation is up to the tool.

When a tool is not in use, the tool's attributes are accepted without a warning. When the tool is in use, the tool is responsible for processing and interpretation of its attributes.

Tool attributes are not available if the `no_implicit_prelude` attribute is used.

```
// Tells the rustfmt tool to not format the following element.
#[rustfmt::skip]
struct S {
}

// Controls the "cyclomatic complexity" threshold for the clippy tool.
#[clippy::cyclomatic_complexity = "100"]
pub fn f() {}
```

> Note: `rustc` currently recognizes the tools "clippy" and "rustfmt".

Built-in attributes index

The following is an index of all built-in attributes.

- Conditional compilation
  - `cfg` — Controls conditional compilation.
  - `cfg_attr` — Conditionally includes attributes.
- Testing
  - `test` — Marks a function as a test.
  - `ignore` — Disables a test function.
  - `should_panic` — Indicates a test should generate a panic.
- Derive
  - `derive` — Automatic trait implementations.
  - `automatically_derived` — Marker for implementations created by `derive`.
- Macros
  - `macro_export` — Exports a `macro_rules` macro for cross-crate usage.
  - `macro_use` — Expands macro visibility, or imports macros from other crates.
  - `proc_macro` — Defines a function-like macro.
  - `proc_macro_derive` — Defines a derive macro.
  - `proc_macro_attribute` — Defines an attribute macro.

- Diagnostics
  - `allow`, `warn`, `deny`, `forbid` — Alters the default lint level.
  - `deprecated` — Generates deprecation notices.
  - `must_use` — Generates a lint for unused values.
- ABI, linking, symbols, and FFI
  - `link` — Specifies a native library to link with an `extern` block.
  - `link_name` — Specifies the name of the symbol for functions or statics in an `extern` block.
  - `link_ordinal` — Specifies the ordinal of the symbol for functions or statics in an `extern` block.
  - `no_link` — Prevents linking an extern crate.
  - `repr` — Controls type layout.
  - `crate_type` — Specifies the type of crate (library, executable, etc.).
  - `no_main` — Disables emitting the `main` symbol.
  - `export_name` — Specifies the exported symbol name for a function or static.
  - `link_section` — Specifies the section of an object file to use for a function or static.
  - `no_mangle` — Disables symbol name encoding.
  - `used` — Forces the compiler to keep a static item in the output object file.
  - `crate_name` — Specifies the crate name.
- Code generation
  - `inline` — Hint to inline code.
  - `cold` — Hint that a function is unlikely to be called.
  - `no_builtins` — Disables use of certain built-in functions.
  - `target_feature` — Configure platform-specific code generation.
  - `track_caller` - Pass the parent call location to `std::panic::Location::caller()`.
  - `instruction_set` - Specify the instruction set used to generate a functions code
- Documentation
  - `doc` — Specifies documentation. See The Rustdoc Book for more information. Doc comments are transformed into `doc` attributes.
- Preludes
  - `no_std` — Removes std from the prelude.
  - `no_implicit_prelude` — Disables prelude lookups within a module.
- Modules
  - `path` — Specifies the filename for a module.
- Limits
  - `recursion_limit` — Sets the maximum recursion limit for certain compile-time operations.

    - `type_length_limit` — Sets the maximum size of a polymorphic type.
- Runtime
    - `panic_handler` — Sets the function to handle panics.
    - `global_allocator` — Sets the global memory allocator.
    - `windows_subsystem` — Specifies the windows subsystem to link with.
- Features
    - `feature` — Used to enable unstable or experimental compiler features. See The Unstable Book for features implemented in `rustc`.
- Type System
    - `non_exhaustive` — Indicate that a type will have more fields/variants added in future.
- Debugger
    - `debugger_visualizer` — Embeds a file that specifies debugger output for a type.

Testing attributes

The following attributes are used for specifying functions for performing tests. Compiling a crate in "test" mode enables building the test functions along with a test harness for executing the tests. Enabling the test mode also enables the `test` conditional compilation option.

The `test` attribute

The *`test` attribute* marks a function to be executed as a test. These functions are only compiled when in test mode. Test functions must be free, monomorphic functions that take no arguments, and the return type must implement the `Termination` trait, for example:

- `()`
- `Result<T, E> where T: Termination, E: Debug`
- `!`

> Note: The test mode is enabled by passing the `--test` argument to `rustc` or using `cargo test`.

The test harness calls the returned value's `report` method, and classifies the test as passed or failed depending on whether the resulting `ExitCode` represents successful termination. In particular:

- Tests that return `()` pass as long as they terminate and do not panic.
- Tests that return a `Result<(), E>` pass as long as they return `Ok(())`.
- Tests that return `ExitCode::SUCCESS` pass, and tests that return `ExitCode::FAILURE` fail.
- Tests that do not terminate neither pass nor fail.

```
#[test]
fn test_the_thing() -> io::Result<()> {
    let state = setup_the_thing()?; // expected to succeed
    do_the_thing(&state)?;          // expected to succeed
    Ok(())
}
```

The `ignore` attribute

A function annotated with the `test` attribute can also be annotated with the `ignore` attribute. The *`ignore` attribute* tells the test harness to not execute that function as a test. It will still be compiled when in test mode.

The `ignore` attribute may optionally be written with the *MetaNameValueStr* syntax to specify a reason why the test is ignored.

```
#[test]
#[ignore = "not yet implemented"]
fn mytest() {
    // …
}
```

> **Note**: The `rustc` test harness supports the `--include-ignored` flag to force ignored tests to be run.

The `should_panic` attribute

A function annotated with the `test` attribute that returns `()` can also be annotated with the `should_panic` attribute. The *`should_panic` attribute* makes the test only pass if it actually panics.

The `should_panic` attribute may optionally take an input string that must appear within the panic message. If the string is not found in the message, then the test will fail. The string may be passed using the *MetaNameValueStr* syntax or the *MetaListNameValueStr* syntax with an `expected` field.

```
#[test]
#[should_panic(expected = "values don't match")]
fn mytest() {
    assert_eq!(1, 2, "values don't match");
}
```

Derive

The *`derive` attribute* allows new items to be automatically generated for data structures. It uses the *MetaListPaths* syntax to specify a list of traits to implement or paths to derive macros to process.

For example, the following will create an `impl` item for the `PartialEq` and `Clone` traits for `Foo`, and the type parameter `T` will be given the `PartialEq` or `Clone` constraints for the appropriate `impl`:

```
#[derive(PartialEq, Clone)]
struct Foo<T> {
    a: i32,
    b: T,
}
```

The generated `impl` for `PartialEq` is equivalent to

```
impl<T: PartialEq> PartialEq for Foo<T> {
    fn eq(&self, other: &Foo<T>) -> bool {
        self.a == other.a && self.b == other.b
    }
}
```

You can implement `derive` for your own traits through procedural macros.

The `automatically_derived` attribute

The *`automatically_derived` attribute* is automatically added to implementations created by the `derive` attribute for built-in traits. It has no direct effect, but it may be used by tools and diagnostic lints to detect these automatically generated implementations.

Diagnostic attributes

The following attributes are used for controlling or generating diagnostic messages during compilation.

Lint check attributes

A lint check names a potentially undesirable coding pattern, such as unreachable code or omitted documentation. The lint attributes `allow`, `warn`, `deny`, and `forbid` use the *MetaListPaths* syntax to specify a list of lint names to change the lint level for the entity to which the attribute applies.

For any lint check `C`:

- `allow(C)` overrides the check for `C` so that violations will go unreported,
- `warn(C)` warns about violations of `C` but continues compilation.
- `deny(C)` signals an error after encountering a violation of `C`,
- `forbid(C)` is the same as `deny(C)`, but also forbids changing the lint level afterwards,

---

> Note: The lint checks supported by `rustc` can be found via `rustc -W help`, along with their default settings and are documented in the rustc book.

```
pub mod m1 {
    // Missing documentation is ignored here
    #[allow(missing_docs)]
    pub fn undocumented_one() -> i32 { 1 }

    // Missing documentation signals a warning here
    #[warn(missing_docs)]
    pub fn undocumented_too() -> i32 { 2 }

    // Missing documentation signals an error here
    #[deny(missing_docs)]
    pub fn undocumented_end() -> i32 { 3 }
}
```

Lint attributes can override the level specified from a previous attribute, as long as the level does not attempt to change a forbidden lint. Previous attributes are those from a higher level in the syntax tree, or from a previous attribute on the same entity as listed in left-to-right source order.

This example shows how one can use `allow` and `warn` to toggle a particular check on and off:

```
#[warn(missing_docs)]
pub mod m2 {
    #[allow(missing_docs)]
    pub mod nested {
        // Missing documentation is ignored here
        pub fn undocumented_one() -> i32 { 1 }

        // Missing documentation signals a warning here,
        // despite the allow above.
        #[warn(missing_docs)]
        pub fn undocumented_two() -> i32 { 2 }
    }

    // Missing documentation signals a warning here
    pub fn undocumented_too() -> i32 { 3 }
}
```

This example shows how one can use `forbid` to disallow uses of `allow` for that lint check:

```
#[forbid(missing_docs)]
pub mod m3 {
    // Attempting to toggle warning signals an error here
    #[allow(missing_docs)]
    /// Returns 2.
    pub fn undocumented_too() -> i32 { 2 }
}
```

> Note: `rustc` allows setting lint levels on the command-line, and also supports setting caps on the lints that are reported.

Lint groups

Lints may be organized into named groups so that the level of related lints can be adjusted together. Using a named group is equivalent to listing out the lints within that group.

```
// This allows all lints in the "unused" group.
#[allow(unused)]
// This overrides the "unused_must_use" lint from the "unused"
// group to deny.
#[deny(unused_must_use)]
fn example() {
    // This does not generate a warning because the "unused_variables"
    // lint is in the "unused" group.
    let x = 1;
    // This generates an error because the result is unused and
    // "unused_must_use" is marked as "deny".
    std::fs::remove_file("some_file"); // ERROR: unused `Result` that must be
used
}
```

There is a special group named "warnings" which includes all lints at the "warn" level. The "warnings" group ignores attribute order and applies to all lints that would otherwise warn within the entity.

```
// The order of these two attributes does not matter.
#[deny(warnings)]
// The unsafe_code lint is normally "allow" by default.
#[warn(unsafe_code)]
fn example_err() {
    // This is an error because the `unsafe_code` warning has
    // been lifted to "deny".
    unsafe { an_unsafe_fn() } // ERROR: usage of `unsafe` block
}
```

Tool lint attributes

Tool lints allows using scoped lints, to `allow`, `warn`, `deny` or `forbid` lints of certain tools.

Tool lints only get checked when the associated tool is active. If a lint attribute, such as `allow`, references a nonexistent tool lint, the compiler will not warn about the nonexistent lint until you use the tool.

Otherwise, they work just like regular lint attributes:

```
// set the entire `pedantic` clippy lint group to warn
#![warn(clippy::pedantic)]
// silence warnings from the `filter_map` clippy lint
#![allow(clippy::filter_map)]

fn main() {
    // ...
}

// silence the `cmp_nan` clippy lint just for this function
#[allow(clippy::cmp_nan)]
fn foo() {
    // ...
}
```

Note: `rustc` currently recognizes the tool lints for "clippy" and "rustdoc".

The `deprecated` attribute

The *`deprecated` attribute* marks an item as deprecated. `rustc` will issue warnings on usage of `#[deprecated]` items. `rustdoc` will show item deprecation, including the `since` version and `note`, if available.

The `deprecated` attribute has several forms:

- `deprecated` — Issues a generic message.
- `deprecated = "message"` — Includes the given string in the deprecation message.
- *MetaListNameValueStr* syntax with two optional fields:
    - `since` — Specifies a version number when the item was deprecated. `rustc` does not currently interpret the string, but external tools like Clippy may check the validity of the value.
    - `note` — Specifies a string that should be included in the deprecation message. This is typically used to provide an explanation about the deprecation and preferred alternatives.

The `deprecated` attribute may be applied to any item, trait item, enum variant, struct field, external block item, or macro definition. It cannot be applied to trait implementation items. When applied to an item containing other items, such as a module or implementation, all child items inherit the deprecation attribute.

Here is an example:

```
#[deprecated(since = "5.2.0", note = "foo was rarely used. Users should instead
use bar")]
pub fn foo() {}

pub fn bar() {}
```

The RFC contains motivations and more details.

The `must_use` attribute

The *`must_use` attribute* is used to issue a diagnostic warning when a value is not "used". It can be applied to user-defined composite types (`struct` s, `enum` s, and `union` s), functions, and traits.

The `must_use` attribute may include a message by using the *MetaNameValueStr* syntax such as `#[must_use = "example message"]` . The message will be given alongside the warning.

When used on user-defined composite types, if the expression of an expression statement has that type, then the `unused_must_use` lint is violated.

```
#[must_use]
struct MustUse {
    // some fields
}

// Violates the `unused_must_use` lint.
MustUse::new();
```

When used on a function, if the expression of an expression statement is a call expression to that function, then the `unused_must_use` lint is violated.

```
#[must_use]
fn five() -> i32 { 5i32 }

// Violates the unused_must_use lint.
five();
```

When used on a trait declaration, a call expression of an expression statement to a function that returns an impl trait or a dyn trait of that trait violates the `unused_must_use` lint.

```
#[must_use]
trait Critical {}
impl Critical for i32 {}

fn get_critical() -> impl Critical {
    4i32
}

// Violates the `unused_must_use` lint.
get_critical();
```

When used on a function in a trait declaration, then the behavior also applies when the call expression is a function from an implementation of the trait.

```
trait Trait {
    #[must_use]
    fn use_me(&self) -> i32;
}

impl Trait for i32 {
    fn use_me(&self) -> i32 { 0i32 }
}

// Violates the `unused_must_use` lint.
5i32.use_me();
```

When used on a function in a trait implementation, the attribute does nothing.

> Note: Trivial no-op expressions containing the value will not violate the lint. Examples include wrapping the value in a type that does not implement `Drop` and then not using that type and being the final expression of a block expression that is not used.
>
> ```
> #[must_use]
> fn five() -> i32 { 5i32 }
>
> // None of these violate the unused_must_use lint.
> (five(),);
> Some(five());
> { five() };
> if true { five() } else { 0i32 };
> match true {
>     _ => five()
> };
> ```

> Note: It is idiomatic to use a let statement with a pattern of `_` when a must-used value

is purposely discarded.

```
#[must_use]
fn five() -> i32 { 5i32 }

// Does not violate the unused_must_use lint.
let _ = five();
```

Code generation attributes

The following attributes are used for controlling code generation.

Optimization hints

The `cold` and `inline` attributes give suggestions to generate code in a way that may be faster than what it would do without the hint. The attributes are only hints, and may be ignored.

Both attributes can be used on functions. When applied to a function in a trait, they apply only to that function when used as a default function for a trait implementation and not to all trait implementations. The attributes have no effect on a trait function without a body.

The `inline` attribute

The ***`inline`*** ***attribute*** suggests that a copy of the attributed function should be placed in the caller, rather than generating code to call the function where it is defined.

> ***Note***: The `rustc` compiler automatically inlines functions based on internal heuristics. Incorrectly inlining functions can make the program slower, so this attribute should be used with care.

There are three ways to use the inline attribute:

- `#[inline]` *suggests* performing an inline expansion.
- `#[inline(always)]` *suggests* that an inline expansion should always be performed.
- `#[inline(never)]` *suggests* that an inline expansion should never be performed.

> ***Note***: `#[inline]` in every form is a hint, with no *requirements* on the language to place a copy of the attributed function in the caller.

The `cold` attribute

The *`cold`* *attribute* suggests that the attributed function is unlikely to be called.

The `no_builtins` attribute

The *`no_builtins`* *attribute* may be applied at the crate level to disable optimizing certain code patterns to invocations of library functions that are assumed to exist.

The `target_feature` attribute

The *`target_feature`* *attribute* may be applied to a function to enable code generation of that function for specific platform architecture features. It uses the *MetaListNameValueStr* syntax with a single key of `enable` whose value is a string of comma-separated feature names to enable.

```
#[target_feature(enable = "avx2")]
unsafe fn foo_avx2() {}
```

Each target architecture has a set of features that may be enabled. It is an error to specify a feature for a target architecture that the crate is not being compiled for.

It is undefined behavior to call a function that is compiled with a feature that is not supported on the current platform the code is running on, *except* if the platform explicitly documents this to be safe.

Functions marked with `target_feature` are not inlined into a context that does not support the given features. The `#[inline(always)]` attribute may not be used with a `target_feature` attribute.

Available features

The following is a list of the available feature names.

`x86` or `x86_64`

Executing code with unsupported features is undefined behavior on this platform. Hence this platform requires that `#[target_feature]` is only applied to `unsafe` functions.

Feature	Implicitly Enables	Description
`adx`		ADX — Multi-Precision Add-Carry Instruction Extensions
`aes`	`sse2`	AES — Advanced Encryption Standard
`avx`	`sse4.2`	AVX — Advanced Vector Extensions
`avx2`	`avx`	AVX2 — Advanced Vector Extensions 2
`bmi1`		BMI1 — Bit Manipulation Instruction Sets
`bmi2`		BMI2 — Bit Manipulation Instruction Sets 2
`cmpxchg16b`		`cmpxchg16b` - Compares and exchange 16 bytes (128 bits) of data atomically
`f16c`	`avx`	F16C — 16-bit floating point conversion instructions
`fma`	`avx`	FMA3 — Three-operand fused multiply-add
`fxsr`		`fxsave` and `fxrstor` — Save and restore x87 FPU, MMX Technology, and SSE State
`lzcnt`		`lzcnt` — Leading zeros count
`movbe`		`movbe` - Move data after swapping bytes
`pclmulqdq`	`sse2`	`pclmulqdq` — Packed carry-less multiplication quadword
`popcnt`		`popcnt` — Count of bits set to 1
`rdrand`		`rdrand` — Read random number
`rdseed`		`rdseed` — Read random seed
`sha`	`sse2`	SHA — Secure Hash Algorithm
`sse`		SSE — Streaming SIMD Extensions
`sse2`	`sse`	SSE2 — Streaming SIMD Extensions 2
`sse3`	`sse2`	SSE3 — Streaming SIMD Extensions 3
`sse4.1`	`ssse3`	SSE4.1 — Streaming SIMD Extensions 4.1
`sse4.2`	`sse4.1`	SSE4.2 — Streaming SIMD Extensions 4.2
`ssse3`	`sse3`	SSSE3 — Supplemental Streaming SIMD Extensions 3
`xsave`		`xsave` — Save processor extended states
`xsavec`		`xsavec` — Save processor extended states with compaction

Feature	Implicitly Enables	Description
`xsaveopt`		`xsaveopt` — Save processor extended states optimized
`xsaves`		`xsaves` — Save processor extended states supervisor

`aarch64`

This platform requires that `#[target_feature]` is only applied to `unsafe` functions.

Further documentation on these features can be found in the ARM Architecture Reference Manual, or elsewhere on developer.arm.com.

Note: The following pairs of features should both be marked as enabled or disabled together if used:

- `paca` and `pacg`, which LLVM currently implements as one feature.

Feature	Implicitly Enables	Feature Name
`aes`	`neon`	FEAT_AES & FEAT_PMULL - Advanced SIMD AES & PMULL instructions
`bf16`		FEAT_BF16 - BFloat16 instructions
`bti`		FEAT_BTI - Branch Target Identification
`crc`		FEAT_CRC - CRC32 checksum instructions
`dit`		FEAT_DIT - Data Independent Timing instructions
`dotprod`		FEAT_DotProd - Advanced SIMD Int8 dot product instructions
`dpb`		FEAT_DPB - Data cache clean to point of persistence
`dpb2`		FEAT_DPB2 - Data cache clean to point of deep persistence
`f32mm`	`sve`	FEAT_F32MM - SVE single-precision FP matrix multiply instruction

Feature	Implicitly Enables	Feature Name
`f64mm`	`sve`	FEAT_F64MM - SVE double-precision FP matrix multiply instruction
`fcma`	`neon`	FEAT_FCMA - Floating point complex number support
`fhm`	`fp16`	FEAT_FHM - Half-precision FP FMLAL instructions
`flagm`		FEAT_FlagM - Conditional flag manipulation
`fp16`	`neon`	FEAT_FP16 - Half-precision FP data processing
`frintts`		FEAT_FRINTTS - Floating-point to int helper instructions
`i8mm`		FEAT_I8MM - Int8 Matrix Multiplication
`jsconv`	`neon`	FEAT_JSCVT - JavaScript conversion instruction
`lse`		FEAT_LSE - Large System Extension
`lor`		FEAT_LOR - Limited Ordering Regions extension
`mte`		FEAT_MTE & FEAT_MTE2 - Memory Tagging Extension
`neon`		FEAT_FP & FEAT_AdvSIMD - Floating Point and Advanced SIMD extension
`pan`		FEAT_PAN - Privileged Access-Never extension
`paca`		FEAT_PAuth - Pointer Authentication (address authentication)
`pacg`		FEAT_PAuth - Pointer Authentication (generic authentication)
`pmuv3`		FEAT_PMUv3 - Performance Monitors extension (v3)
`rand`		FEAT_RNG - Random Number Generator
`ras`		FEAT_RAS & FEAT_RASv1p1 - Reliability, Availability and Serviceability extension
`rcpc`		FEAT_LRCPC - Release consistent Processor Consistent
`rcpc2`	`rcpc`	FEAT_LRCPC2 - RcPc with immediate offsets
`rdm`		FEAT_RDM - Rounding Double Multiply accumulate
`sb`		FEAT_SB - Speculation Barrier

Feature	Implicitly Enables	Feature Name
`sha2`	`neon`	FEAT_SHA1 & FEAT_SHA256 - Advanced SIMD SHA instructions
`sha3`	`sha2`	FEAT_SHA512 & FEAT_SHA3 - Advanced SIMD SHA instructions
`sm4`	`neon`	FEAT_SM3 & FEAT_SM4 - Advanced SIMD SM3/4 instructions
`spe`		FEAT_SPE - Statistical Profiling Extension
`ssbs`		FEAT_SSBS & FEAT_SSBS2 - Speculative Store Bypass Safe
`sve`	`fp16`	FEAT_SVE - Scalable Vector Extension
`sve2`	`sve`	FEAT_SVE2 - Scalable Vector Extension 2
`sve2-aes`	`sve2` , `aes`	FEAT_SVE_AES - SVE AES instructions
`sve2-sm4`	`sve2` , `sm4`	FEAT_SVE_SM4 - SVE SM4 instructions
`sve2-sha3`	`sve2` , `sha3`	FEAT_SVE_SHA3 - SVE SHA3 instructions
`sve2-bitperm`	`sve2`	FEAT_SVE_BitPerm - SVE Bit Permute
`tme`		FEAT_TME - Transactional Memory Extension
`vh`		FEAT_VHE - Virtualization Host Extensions

`riscv32` or `riscv64`

This platform requires that `#[target_feature]` is only applied to `unsafe` functions.

Further documentation on these features can be found in their respective specification. Many specifications are described in the RISC-V ISA Manual or in another manual hosted on the RISC-V GitHub Account.

Feature	Implicitly Enables	Description
`a`		A — Atomic instructions
`c`		C — Compressed instructions
`m`		M — Integer Multiplication and Division instructions
`zb`	`zba` , `zbc` , `zbs`	Zb — Bit Manipulation instructions

Feature	Implicitly Enables	Description
zba		Zba — Address Generation instructions
zbb		Zbb — Basic bit-manipulation
zbc		Zbc — Carry-less multiplication
zbkb		Zbkb — Bit Manipulation Instructions for Cryptography
zbkc		Zbkc — Carry-less multiplication for Cryptography
zbkx		Zbkx — Crossbar permutations
zbs		Zbs — Single-bit instructions
zk	zkn, zkr, zks, zkt, zbkb, zbkc, zkbx	Zk — Scalar Cryptography
zkn	zknd, zkne, zknh, zbkb, zbkc, zkbx	Zkn — NIST Algorithm suite extension
zknd		Zknd — NIST Suite: AES Decryption
zkne		Zkne — NIST Suite: AES Encryption
zknh		Zknh — NIST Suite: Hash Function Instructions
zkr		Zkr — Entropy Source Extension
zks	zksed, zksh, zbkb, zbkc, zkbx	Zks — ShangMi Algorithm Suite
zksed		Zksed — ShangMi Suite: SM4 Block Cipher Instructions
zksh		Zksh — ShangMi Suite: SM3 Hash Function Instructions
zkt		Zkt — Data Independent Execution Latency Subset

`wasm32` or `wasm64`

`#[target_feature]` may be used with both safe and `unsafe` functions on Wasm platforms. It is impossible to cause undefined behavior via the `#[target_feature]` attribute because attempting to use instructions unsupported by the Wasm engine will fail at load time without the risk of being interpreted in a way different from what the compiler expected.

Feature	Description
`simd128`	WebAssembly simd proposal

Additional information

See the `target_feature` conditional compilation option for selectively enabling or disabling compilation of code based on compile-time settings. Note that this option is not affected by the `target_feature` attribute, and is only driven by the features enabled for the entire crate.

See the `is_x86_feature_detected` or `is_aarch64_feature_detected` macros in the standard library for runtime feature detection on these platforms.

> Note: `rustc` has a default set of features enabled for each target and CPU. The CPU may be chosen with the `-C target-cpu` flag. Individual features may be enabled or disabled for an entire crate with the `-C target-feature` flag.

The `track_caller` attribute

The `track_caller` attribute may be applied to any function with `"Rust"` ABI with the exception of the entry point `fn main`. When applied to functions and methods in trait declarations, the attribute applies to all implementations. If the trait provides a default implementation with the attribute, then the attribute also applies to override implementations.

When applied to a function in an `extern` block the attribute must also be applied to any linked implementations, otherwise undefined behavior results. When applied to a function which is made available to an `extern` block, the declaration in the `extern` block must also have the attribute, otherwise undefined behavior results.

Behavior

Applying the attribute to a function `f` allows code within `f` to get a hint of the `Location` of the "topmost" tracked call that led to `f`'s invocation. At the point of observation, an implementation behaves as if it walks up the stack from `f`'s frame to find the nearest frame of an *unattributed* function `outer`, and it returns the `Location` of the tracked call in `outer`.

```
#[track_caller]
fn f() {
    println!("{}", std::panic::Location::caller());
}
```

> Note: `core` provides `core::panic::Location::caller` for observing caller locations. It wraps the `core::intrinsics::caller_location` intrinsic implemented by `rustc`.

> Note: because the resulting `Location` is a hint, an implementation may halt its walk up the stack early. See Limitations for important caveats.

Examples

When `f` is called directly by `calls_f`, code in `f` observes its callsite within `calls_f`:

```
fn calls_f() {
    f(); // <-- f() prints this location
}
```

When `f` is called by another attributed function `g` which is in turn called by `calls_g`, code in both `f` and `g` observes `g`'s callsite within `calls_g`:

```
#[track_caller]
fn g() {
    println!("{}", std::panic::Location::caller());
    f();
}

fn calls_g() {
    g(); // <-- g() prints this location twice, once itself and once from f()
}
```

When `g` is called by another attributed function `h` which is in turn called by `calls_h`, all code in `f`, `g`, and `h` observes `h`'s callsite within `calls_h`:

```
#[track_caller]
fn h() {
    println!("{}", std::panic::Location::caller());
    g();
}

fn calls_h() {
    h(); // <-- prints this location three times, once itself, once from g(),
once from f()
}
```

And so on.

Limitations

This information is a hint and implementations are not required to preserve it.

In particular, coercing a function with `#[track_caller]` to a function pointer creates a shim which appears to observers to have been called at the attributed function's definition site, losing actual caller information across virtual calls. A common example of this coercion is the creation of a trait object whose methods are attributed.

> Note: The aforementioned shim for function pointers is necessary because `rustc` implements `track_caller` in a codegen context by appending an implicit parameter to the function ABI, but this would be unsound for an indirect call because the parameter is not a part of the function's type and a given function pointer type may or may not refer to a function with the attribute. The creation of a shim hides the implicit parameter from callers of the function pointer, preserving soundness.

The `instruction_set` attribute

The *`instruction_set`* *attribute* may be applied to a function to control which instruction set the function will be generated for. This allows mixing more than one instruction set in a single program on CPU architectures that support it. It uses the *MetaListPath* syntax, and a path comprised of the architecture family name and instruction set name.

It is a compilation error to use the `instruction_set` attribute on a target that does not support it.

On ARM

For the `ARMv4T` and `ARMv5te` architectures, the following are supported:

- `arm::a32` - Generate the function as A32 "ARM" code.
- `arm::t32` - Generate the function as T32 "Thumb" code.

```
#[instruction_set(arm::a32)]
fn foo_arm_code() {}

#[instruction_set(arm::t32)]
fn bar_thumb_code() {}
```

Using the `instruction_set` attribute has the following effects:

- If the address of the function is taken as a function pointer, the low bit of the address will be set to 0 (arm) or 1 (thumb) depending on the instruction set.
- Any inline assembly in the function must use the specified instruction set instead of the target default.

Limits

The following attributes affect compile-time limits.

The `recursion_limit` attribute

The *`recursion_limit` attribute* may be applied at the crate level to set the maximum depth for potentially infinitely-recursive compile-time operations like macro expansion or auto-dereference. It uses the *MetaNameValueStr* syntax to specify the recursion depth.

> Note: The default in `rustc` is 128.

```
#![recursion_limit = "4"]

macro_rules! a {
    () => { a!(1); };
    (1) => { a!(2); };
    (2) => { a!(3); };
    (3) => { a!(4); };
    (4) => { };
}

// This fails to expand because it requires a recursion depth greater than 4.
a!{}
```

```
#![recursion_limit = "1"]

// This fails because it requires two recursive steps to auto-dereference.
(|_: &u8| {})(&&&1);
```

The `type_length_limit` attribute

The *`type_length_limit` attribute* limits the maximum number of type substitutions made when constructing a concrete type during monomorphization. It is applied at the crate level, and uses the *MetaNameValueStr* syntax to set the limit based on the number of type substitutions.

> Note: The default in `rustc` is 1048576.

```
#![type_length_limit = "4"]

fn f<T>(x: T) {}

// This fails to compile because monomorphizing to
// `f::<((((i32,), i32), i32), i32)>` requires more than 4 type elements.
f(((((1,), 2), 3), 4));
```

Type system attributes

The following attributes are used for changing how a type can be used.

The `non_exhaustive` attribute

The *`non_exhaustive` attribute* indicates that a type or variant may have more fields or variants added in the future. It can be applied to `struct`s, `enum`s, and `enum` variants.

The `non_exhaustive` attribute uses the *MetaWord* syntax and thus does not take any inputs.

Within the defining crate, `non_exhaustive` has no effect.

```
#[non_exhaustive]
pub struct Config {
    pub window_width: u16,
    pub window_height: u16,
}

#[non_exhaustive]
pub enum Error {
    Message(String),
    Other,
}

pub enum Message {
    #[non_exhaustive] Send { from: u32, to: u32, contents: String },
    #[non_exhaustive] Reaction(u32),
    #[non_exhaustive] Quit,
}

// Non-exhaustive structs can be constructed as normal within the defining
crate.
let config = Config { window_width: 640, window_height: 480 };

// Non-exhaustive structs can be matched on exhaustively within the defining
crate.
if let Config { window_width, window_height } = config {
    // ...
}

let error = Error::Other;
let message = Message::Reaction(3);

// Non-exhaustive enums can be matched on exhaustively within the defining
crate.
match error {
    Error::Message(ref s) => { },
    Error::Other => { },
}

match message {
    // Non-exhaustive variants can be matched on exhaustively within the
defining crate.
    Message::Send { from, to, contents } => { },
    Message::Reaction(id) => { },
    Message::Quit => { },
}
```

Outside of the defining crate, types annotated with `non_exhaustive` have limitations that preserve backwards compatibility when new fields or variants are added.

Non-exhaustive types cannot be constructed outside of the defining crate:

- Non-exhaustive variants (`struct` or `enum` variant) cannot be constructed with a *StructExpression* (including with functional update syntax).
- `enum` instances can be constructed.

```
// `Config`, `Error`, and `Message` are types defined in an upstream crate that
have been
// annotated as `#[non_exhaustive]`.
use upstream::{Config, Error, Message};

// Cannot construct an instance of `Config`, if new fields were added in
// a new version of `upstream` then this would fail to compile, so it is
// disallowed.
let config = Config { window_width: 640, window_height: 480 };

// Can construct an instance of `Error`, new variants being introduced would
// not result in this failing to compile.
let error = Error::Message("foo".to_string());

// Cannot construct an instance of `Message::Send` or `Message::Reaction`,
// if new fields were added in a new version of `upstream` then this would
// fail to compile, so it is disallowed.
let message = Message::Send { from: 0, to: 1, contents: "foo".to_string(), };
let message = Message::Reaction(0);

// Cannot construct an instance of `Message::Quit`, if this were converted to
// a tuple-variant `upstream` then this would fail to compile.
let message = Message::Quit;
```

There are limitations when matching on non-exhaustive types outside of the defining crate:

- When pattern matching on a non-exhaustive variant (`struct` or `enum` variant), a *StructPattern* must be used which must include a `..` . Tuple variant constructor visibility is lowered to `min($vis, pub(crate))` .
- When pattern matching on a non-exhaustive `enum` , matching on a variant does not contribute towards the exhaustiveness of the arms.

```
// `Config`, `Error`, and `Message` are types defined in an upstream crate that
have been
// annotated as `#[non_exhaustive]`.
use upstream::{Config, Error, Message};

// Cannot match on a non-exhaustive enum without including a wildcard arm.
match error {
  Error::Message(ref s) => { },
  Error::Other => { },
  // would compile with: `_ => {},`
}

// Cannot match on a non-exhaustive struct without a wildcard.
if let Ok(Config { window_width, window_height }) = config {
    // would compile with: `..`
}

match message {
  // Cannot match on a non-exhaustive struct enum variant without including a
wildcard.
  Message::Send { from, to, contents } => { },
  // Cannot match on a non-exhaustive tuple or unit enum variant.
  Message::Reaction(type) => { },
  Message::Quit => { },
}
```

It's also not allowed to cast non-exhaustive types from foreign crates.

```
use othercrate::NonExhaustiveEnum;

// Cannot cast a non-exhaustive enum outside of its defining crate.
let _ = NonExhaustiveEnum::default() as u8;
```

Non-exhaustive types are always considered inhabited in downstream crates.

Debugger attributes

The following attributes are used for enhancing the debugging experience when using third-party debuggers like GDB or WinDbg.

The `debugger_visualizer` attribute

The *`debugger_visualizer` attribute* can be used to embed a debugger visualizer file into the debug information. This enables an improved debugger experience for displaying values in the debugger. It uses the *MetaListNameValueStr* syntax to specify its inputs, and must be specified as a crate attribute.

Using `debugger_visualizer` with Natvis

Natvis is an XML-based framework for Microsoft debuggers (such as Visual Studio and WinDbg) that uses declarative rules to customize the display of types. For detailed information on the Natvis format, refer to Microsoft's Natvis documentation.

This attribute only supports embedding Natvis files on `-windows-msvc` targets.

The path to the Natvis file is specified with the `natvis_file` key, which is a path relative to the crate source file:

```
#![debugger_visualizer(natvis_file = "Rectangle.natvis")]

struct FancyRect {
    x: f32,
    y: f32,
    dx: f32,
    dy: f32,
}

fn main() {
    let fancy_rect = FancyRect { x: 10.0, y: 10.0, dx: 5.0, dy: 5.0 };
    println!("set breakpoint here");
}
```

and `Rectangle.natvis` contains:

```xml
<?xml version="1.0" encoding="utf-8"?>
<AutoVisualizer xmlns="http://schemas.microsoft.com/vstudio/debugger/natvis/
2010">
    <Type Name="foo::FancyRect">
      <DisplayString>({x},{y}) + ({dx}, {dy})</DisplayString>
      <Expand>
        <Synthetic Name="LowerLeft">
          <DisplayString>({x}, {y})</DisplayString>
        </Synthetic>
        <Synthetic Name="UpperLeft">
          <DisplayString>({x}, {y + dy})</DisplayString>
        </Synthetic>
        <Synthetic Name="UpperRight">
          <DisplayString>({x + dx}, {y + dy})</DisplayString>
        </Synthetic>
        <Synthetic Name="LowerRight">
          <DisplayString>({x + dx}, {y})</DisplayString>
        </Synthetic>
      </Expand>
    </Type>
</AutoVisualizer>
```

When viewed under WinDbg, the `fancy_rect` variable would be shown as follows:

```
> Variables:
  > fancy_rect: (10.0, 10.0) + (5.0, 5.0)
    > LowerLeft: (10.0, 10.0)
    > UpperLeft: (10.0, 15.0)
    > UpperRight: (15.0, 15.0)
    > LowerRight: (15.0, 10.0)
```

Using `debugger_visualizer` with GDB

GDB supports the use of a structured Python script, called a *pretty printer*, that describes how a type should be visualized in the debugger view. For detailed information on pretty printers, refer to GDB's pretty printing documentation.

Embedded pretty printers are not automatically loaded when debugging a binary under GDB. There are two ways to enable auto-loading embedded pretty printers:

1. Launch GDB with extra arguments to explicitly add a directory or binary to the auto-load safe path: `gdb -iex "add-auto-load-safe-path safe-path path/to/binary" path/to/binary` For more information, see GDB's auto-loading documentation.
2. Create a file named `gdbinit` under `$HOME/.config/gdb` (you may need to create the directory if it doesn't already exist). Add the following line to that file: `add-auto-load-`

safe-path path/to/binary .

These scripts are embedded using the `gdb_script_file` key, which is a path relative to the crate source file.

```rust
#![debugger_visualizer(gdb_script_file = "printer.py")]

struct Person {
    name: String,
    age: i32,
}

fn main() {
    let bob = Person { name: String::from("Bob"), age: 10 };
    println!("set breakpoint here");
}
```

and `printer.py` contains:

```python
import gdb

class PersonPrinter:
    "Print a Person"

    def __init__(self, val):
        self.val = val
        self.name = val["name"]
        self.age = int(val["age"])

    def to_string(self):
        return "{} is {} years old.".format(self.name, self.age)

def lookup(val):
    lookup_tag = val.type.tag
    if lookup_tag is None:
        return None
    if "foo::Person" == lookup_tag:
        return PersonPrinter(val)

    return None

gdb.current_objfile().pretty_printers.append(lookup)
```

When the crate's debug executable is passed into GDB[1], `print bob` will display:

```
"Bob" is 10 years old.
```

[1] Note: This assumes you are using the `rust-gdb` script which configures pretty-printers for standard library types like `String`.

Statements and expressions

Rust is *primarily* an expression language. This means that most forms of value-producing or effect-causing evaluation are directed by the uniform syntax category of *expressions*. Each kind of expression can typically *nest* within each other kind of expression, and rules for evaluation of expressions involve specifying both the value produced by the expression and the order in which its sub-expressions are themselves evaluated.

In contrast, statements serve *mostly* to contain and explicitly sequence expression evaluation.

Statements

Syntax

Statement :
  ;
  | *Item*
  | *LetStatement*
  | *ExpressionStatement*
  | *MacroInvocationSemi*

A *statement* is a component of a block, which is in turn a component of an outer expression or function.

Rust has two kinds of statement: declaration statements and expression statements.

Declaration statements

A *declaration statement* is one that introduces one or more *names* into the enclosing statement block. The declared names may denote new variables or new items.

The two kinds of declaration statements are item declarations and `let` statements.

Item declarations

An *item declaration statement* has a syntactic form identical to an item declaration within a module. Declaring an item within a statement block restricts its scope to the block containing the statement. The item is not given a canonical path nor are any sub-items it may declare. The exception to this is that associated items defined by implementations are still accessible in outer scopes as long as the item and, if applicable, trait are accessible. It is otherwise identical in meaning to declaring the item inside a module.

There is no implicit capture of the containing function's generic parameters, parameters, and local variables. For example, `inner` may not access `outer_var`.

```
fn outer() {
  let outer_var = true;

  fn inner() { /* outer_var is not in scope here */ }

  inner();
}
```

`let` statements

> **Syntax**
>
> *LetStatement* :
>
> *OuterAttribute** `let` *PatternNoTopAlt* (`:` *Type*)? (`=` *Expression* † (`else` *BlockExpression*) ?) ? `;`
>
> † When an `else` block is specified, the *Expression* must not be a *LazyBooleanExpression*, or end with a `}`.

A ***`let` statement*** introduces a new set of variables, given by a pattern. The pattern is followed optionally by a type annotation and then either ends, or is followed by an initializer expression plus an optional `else` block. When no type annotation is given, the compiler will infer the type, or signal an error if insufficient type information is available for definite inference. Any variables introduced by a variable declaration are visible from the point of declaration until the end of the enclosing block scope, except when they are shadowed by another variable declaration.

If an `else` block is not present, the pattern must be irrefutable. If an `else` block is present, the pattern may be refutable. If the pattern does not match (this requires it to be refutable), the `else` block is executed. The `else` block must always diverge (evaluate to the never type).

```
let (mut v, w) = (vec![1, 2, 3], 42); // The bindings may be mut or const
let Some(t) = v.pop() else { // Refutable patterns require an else block
    panic!(); // The else block must diverge
};
let [u, v] = [v[0], v[1]] else { // This pattern is irrefutable, so the
compiler
                                  // will lint as the else block is redundant.
    panic!();
};
```

Expression statements

Syntax

ExpressionStatement :
    ExpressionWithoutBlock `;`
  | *ExpressionWithBlock* `;`?

An *expression statement* is one that evaluates an expression and ignores its result. As a rule, an expression statement's purpose is to trigger the effects of evaluating its expression.

An expression that consists of only a block expression or control flow expression, if used in a context where a statement is permitted, can omit the trailing semicolon. This can cause an ambiguity between it being parsed as a standalone statement and as a part of another expression; in this case, it is parsed as a statement. The type of *ExpressionWithBlock* expressions when used as statements must be the unit type.

```
v.pop();          // Ignore the element returned from pop
if v.is_empty() {
    v.push(5);
} else {
    v.remove(0);
}                 // Semicolon can be omitted.
[1];              // Separate expression statement, not an indexing expression.
```

When the trailing semicolon is omitted, the result must be type `()`.

```
// bad: the block's type is i32, not ()
// Error: expected `()` because of default return type
// if true {
//   1
// }

// good: the block's type is i32
if true {
  1
} else {
  2
};
```

Attributes on Statements

Statements accept outer attributes. The attributes that have meaning on a statement are `cfg`, and the lint check attributes.

Expressions

Syntax

Expression :
  ExpressionWithoutBlock
 | *ExpressionWithBlock*

ExpressionWithoutBlock :
 *OuterAttribute**†
 (
   LiteralExpression
  | *PathExpression*
  | *OperatorExpression*
  | *GroupedExpression*
  | *ArrayExpression*
  | *AwaitExpression*
  | *IndexExpression*
  | *TupleExpression*
  | *TupleIndexingExpression*
  | *StructExpression*
  | *CallExpression*
  | *MethodCallExpression*
  | *FieldExpression*
  | *ClosureExpression*
  | *AsyncBlockExpression*
  | *ContinueExpression*
  | *BreakExpression*
  | *RangeExpression*
  | *ReturnExpression*
  | *UnderscoreExpression*
  | *MacroInvocation*
)

ExpressionWithBlock :
 *OuterAttribute**†
 (
   BlockExpression
  | *UnsafeBlockExpression*
  | *LoopExpression*

| *IfExpression*
| *IfLetExpression*
| *MatchExpression*
)

An expression may have two roles: it always produces a *value*, and it may have *effects* (otherwise known as "side effects"). An expression *evaluates to* a value, and has effects during *evaluation*. Many expressions contain sub-expressions, called the *operands* of the expression. The meaning of each kind of expression dictates several things:

- Whether or not to evaluate the operands when evaluating the expression
- The order in which to evaluate the operands
- How to combine the operands' values to obtain the value of the expression

In this way, the structure of expressions dictates the structure of execution. Blocks are just another kind of expression, so blocks, statements, expressions, and blocks again can recursively nest inside each other to an arbitrary depth.

> **Note**: We give names to the operands of expressions so that we may discuss them, but these names are not stable and may be changed.

Expression precedence

The precedence of Rust operators and expressions is ordered as follows, going from strong to weak. Binary Operators at the same precedence level are grouped in the order given by their associativity.

Operator/Expression	Associativity
Paths	
Method calls	
Field expressions	left to right
Function calls, array indexing	
`?`	
Unary `-` `*` `!` `&` `&mut`	
`as`	left to right
`*` `/` `%`	left to right

Operator/Expression	Associativity
`+` `-`	left to right
`<<` `>>`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`==` `!=` `<` `>` `<=` `>=`	Require parentheses
`&&`	left to right
`||`	left to right
`..` `..=`	Require parentheses
`=` `+=` `-=` `*=` `/=` `%=` `&=` `|=` `^=` `<<=` `>>=`	right to left
`return` `break` closures	

Evaluation order of operands

The following list of expressions all evaluate their operands the same way, as described after the list. Other expressions either don't take operands or evaluate them conditionally as described on their respective pages.

- Dereference expression
- Error propagation expression
- Negation expression
- Arithmetic and logical binary operators
- Comparison operators
- Type cast expression
- Grouped expression
- Array expression
- Await expression
- Index expression
- Tuple expression
- Tuple index expression
- Struct expression
- Call expression
- Method call expression
- Field expression

- Break expression
- Range expression
- Return expression

The operands of these expressions are evaluated prior to applying the effects of the expression. Expressions taking multiple operands are evaluated left to right as written in the source code.

> **Note**: Which subexpressions are the operands of an expression is determined by expression precedence as per the previous section.

For example, the two `next` method calls will always be called in the same order:

```
let mut one_two = vec![1, 2].into_iter();
assert_eq!(
    (1, 2),
    (one_two.next().unwrap(), one_two.next().unwrap())
);
```

> **Note**: Since this is applied recursively, these expressions are also evaluated from innermost to outermost, ignoring siblings until there are no inner subexpressions.

Place Expressions and Value Expressions

Expressions are divided into two main categories: place expressions and value expressions; there is also a third, minor category of expressions called assignee expressions. Within each expression, operands may likewise occur in either place context or value context. The evaluation of an expression depends both on its own category and the context it occurs within.

A *place expression* is an expression that represents a memory location. These expressions are paths which refer to local variables, static variables, dereferences (`*expr`), array indexing expressions (`expr[expr]`), field references (`expr.f`) and parenthesized place expressions. All other expressions are value expressions.

A *value expression* is an expression that represents an actual value.

The following contexts are *place expression* contexts:

- The left operand of a compound assignment expression.
- The operand of a unary borrow, address-of or dereference operator.
- The operand of a field expression.
- The indexed operand of an array indexing expression.
- The operand of any implicit borrow.
- The initializer of a let statement.
- The scrutinee of an `if let`, `match`, or `while let` expression.
- The base of a functional update struct expression.

Note: Historically, place expressions were called *lvalues* and value expressions were called *rvalues*.

An *assignee expression* is an expression that appears in the left operand of an assignment expression. Explicitly, the assignee expressions are:

- Place expressions.
- Underscores.
- Tuples of assignee expressions.
- Slices of assignee expressions.
- Tuple structs of assignee expressions.
- Structs of assignee expressions (with optionally named fields).
- Unit structs.

Arbitrary parenthesisation is permitted inside assignee expressions.

Moved and copied types

When a place expression is evaluated in a value expression context, or is bound by value in a pattern, it denotes the value held *in* that memory location. If the type of that value implements `Copy`, then the value will be copied. In the remaining situations, if that type is `Sized`, then it may be possible to move the value. Only the following place expressions may be moved out of:

- Variables which are not currently borrowed.
- Temporary values.
- Fields of a place expression which can be moved out of and don't implement `Drop`.
- The result of dereferencing an expression with type `Box<T>` and that can also be moved out of.

After moving out of a place expression that evaluates to a local variable, the location is deinitialized and cannot be read from again until it is reinitialized. In all other cases, trying to use a place expression in a value expression context is an error.

Mutability

For a place expression to be assigned to, mutably borrowed, implicitly mutably borrowed, or bound to a pattern containing `ref mut`, it must be *mutable*. We call these *mutable place expressions*. In contrast, other place expressions are called *immutable place expressions*.

The following expressions can be mutable place expression contexts:

- Mutable variables which are not currently borrowed.
- Mutable `static` items.
- Temporary values.
- Fields: this evaluates the subexpression in a mutable place expression context.
- Dereferences of a `*mut T` pointer.
- Dereference of a variable, or field of a variable, with type `&mut T`. Note: This is an exception to the requirement of the next rule.
- Dereferences of a type that implements `DerefMut`: this then requires that the value being dereferenced is evaluated in a mutable place expression context.
- Array indexing of a type that implements `IndexMut`: this then evaluates the value being indexed, but not the index, in mutable place expression context.

Temporaries

When using a value expression in most place expression contexts, a temporary unnamed memory location is created and initialized to that value. The expression evaluates to that location instead, except if promoted to a `static`. The drop scope of the temporary is usually the end of the enclosing statement.

Implicit Borrows

Certain expressions will treat an expression as a place expression by implicitly borrowing it. For example, it is possible to compare two unsized slices for equality directly, because the `==` operator implicitly borrows its operands:

```rust
let a: &[i32];
let b: &[i32];
// ...
*a == *b;
// Equivalent form:
::std::cmp::PartialEq::eq(&*a, &*b);
```

Implicit borrows may be taken in the following expressions:

- Left operand in method-call expressions.
- Left operand in field expressions.
- Left operand in call expressions.
- Left operand in array indexing expressions.
- Operand of the dereference operator (`*`).
- Operands of comparison.
- Left operands of the compound assignment.

Overloading Traits

Many of the following operators and expressions can also be overloaded for other types using traits in `std::ops` or `std::cmp`. These traits also exist in `core::ops` and `core::cmp` with the same names.

Expression Attributes

Outer attributes before an expression are allowed only in a few specific cases:

- Before an expression used as a statement.
- Elements of array expressions, tuple expressions, call expressions, and tuple-style struct expressions.
- The tail expression of block expressions.

They are never allowed before:

- Range expressions.
- Binary operator expressions (*ArithmeticOrLogicalExpression*, *ComparisonExpression*, *LazyBooleanExpression*, *TypeCastExpression*, *AssignmentExpression*, *CompoundAssignmentExpression*).

Literal expressions

Syntax

LiteralExpression :
   CHAR_LITERAL
 | STRING_LITERAL
 | RAW_STRING_LITERAL
 | BYTE_LITERAL
 | BYTE_STRING_LITERAL
 | RAW_BYTE_STRING_LITERAL
 | C_STRING_LITERAL
 | RAW_C_STRING_LITERAL
 | INTEGER_LITERAL
 | FLOAT_LITERAL
 | `true` | `false`

A *literal expression* is an expression consisting of a single token, rather than a sequence of tokens, that immediately and directly denotes the value it evaluates to, rather than referring to it by name or some other evaluation rule.

A literal is a form of constant expression, so is evaluated (primarily) at compile time.

Each of the lexical literal forms described earlier can make up a literal expression, as can the keywords `true` and `false`.

```
"hello";   // string type
'5';       // character type
5;         // integer type
```

In the descriptions below, the *string representation* of a token is the sequence of characters from the input which matched the token's production in a *Lexer* grammar snippet.

> **Note**: this string representation never includes a character `U+000D` (CR) immediately followed by `U+000A` (LF): this pair would have been previously transformed into a single `U+000A` (LF).

Escapes

The descriptions of textual literal expressions below make use of several forms of *escape*.

Each form of escape is characterised by:

- an *escape sequence*: a sequence of characters, which always begins with `U+005C` (`\`)
- an *escaped value*: either a single character or an empty sequence of characters

In the definitions of escapes below:

- An *octal digit* is any of the characters in the range [`0` - `7`].
- A *hexadecimal digit* is any of the characters in the ranges [`0` - `9`], [`a` - `f`], or [`A` - `F`].

Simple escapes

Each sequence of characters occurring in the first column of the following table is an escape sequence.

In each case, the escaped value is the character given in the corresponding entry in the second column.

Escape sequence	Escaped value
`\0`	U+0000 (NUL)
`\t`	U+0009 (HT)
`\n`	U+000A (LF)
`\r`	U+000D (CR)
`\"`	U+0022 (QUOTATION MARK)
`\'`	U+0027 (APOSTROPHE)
`\\`	U+005C (REVERSE SOLIDUS)

8-bit escapes

The escape sequence consists of `\x` followed by two hexadecimal digits.

The escaped value is the character whose Unicode scalar value is the result of interpreting the final two characters in the escape sequence as a hexadecimal integer, as if by `u8::from_str_radix` with radix 16.

> **Note**: the escaped value therefore has a Unicode scalar value in the range of `u8`.

7-bit escapes

The escape sequence consists of `\x` followed by an octal digit then a hexadecimal digit.

The escaped value is the character whose Unicode scalar value is the result of interpreting the final two characters in the escape sequence as a hexadecimal integer, as if by `u8::from_str_radix` with radix 16.

Unicode escapes

The escape sequence consists of `\u{`, followed by a sequence of characters each of which is a hexadecimal digit or `_`, followed by `}`.

The escaped value is the character whose Unicode scalar value is the result of interpreting the hexadecimal digits contained in the escape sequence as a hexadecimal integer, as if by `u8::from_str_radix` with radix 16.

> **Note**: the permitted forms of a CHAR_LITERAL or STRING_LITERAL token ensure that there is such a character.

String continuation escapes

The escape sequence consists of `\` followed immediately by `U+000A` (LF), and all following whitespace characters before the next non-whitespace character. For this purpose, the whitespace characters are `U+0009` (HT), `U+000A` (LF), `U+000D` (CR), and `U+0020` (SPACE).

The escaped value is an empty sequence of characters.

> **Note**: The effect of this form of escape is that a string continuation skips following whitespace, including additional newlines. Thus `a`, `b` and `c` are equal:

```
let a = "foobar";
let b = "foo\
         bar";
let c = "foo\

     bar";

assert_eq!(a, b);
assert_eq!(b, c);
```

Skipping additional newlines (as in example c) is potentially confusing and unexpected. This behavior may be adjusted in the future. Until a decision is made, it is recommended to avoid relying on skipping multiple newlines with line continuations. See this issue for more information.

Character literal expressions

A character literal expression consists of a single CHAR_LITERAL token.

The expression's type is the primitive `char` type.

The token must not have a suffix.

The token's *literal content* is the sequence of characters following the first `U+0027` (`'`) and preceding the last `U+0027` (`'`) in the string representation of the token.

The literal expression's *represented character* is derived from the literal content as follows:

- If the literal content is one of the following forms of escape sequence, the represented character is the escape sequence's escaped value:
  - Simple escapes
  - 7-bit escapes
  - Unicode escapes
- Otherwise the represented character is the single character that makes up the literal content.

The expression's value is the `char` corresponding to the represented character's Unicode scalar value.

> **Note**: the permitted forms of a CHAR_LITERAL token ensure that these rules always produce a single character.

Examples of character literal expressions:

```
'R';                               // R
'\'';                              // '
'\x52';                            // R
'\u{00E6}';                        // LATIN SMALL LETTER AE (U+00E6)
```

String literal expressions

A string literal expression consists of a single STRING_LITERAL or RAW_STRING_LITERAL token.

The expression's type is a shared reference (with `static` lifetime) to the primitive `str` type. That is, the type is `&'static str`.

The token must not have a suffix.

The token's *literal content* is the sequence of characters following the first `U+0022` (`"`) and preceding the last `U+0022` (`"`) in the string representation of the token.

The literal expression's *represented string* is a sequence of characters derived from the literal content as follows:

- If the token is a STRING_LITERAL, each escape sequence of any of the following forms occurring in the literal content is replaced by the escape sequence's escaped value.
  - Simple escapes
  - 7-bit escapes
  - Unicode escapes
  - String continuation escapes

  These replacements take place in left-to-right order. For example, the token `"\\x41"` is converted to the characters `\` `x` `4` `1`.

- If the token is a RAW_STRING_LITERAL, the represented string is identical to the literal content.

The expression's value is a reference to a statically allocated `str` containing the UTF-8 encoding of the represented string.

Examples of string literal expressions:

```
"foo"; r"foo";                     // foo
"\"foo\""; r#""foo""#;             // "foo"

"foo #\"# bar";
r##"foo #"# bar"##;                // foo #"# bar

"\x52"; "R"; r"R";                 // R
"\\x52"; r"\x52";                  // \x52
```

Byte literal expressions

A byte literal expression consists of a single BYTE_LITERAL token.

The expression's type is the primitive `u8` type.

The token must not have a suffix.

The token's *literal content* is the sequence of characters following the first `U+0027` (`'`) and preceding the last `U+0027` (`'`) in the string representation of the token.

The literal expression's *represented character* is derived from the literal content as follows:

- If the literal content is one of the following forms of escape sequence, the represented character is the escape sequence's escaped value:
  - Simple escapes
  - 8-bit escapes
- Otherwise the represented character is the single character that makes up the literal content.

The expression's value is the represented character's Unicode scalar value.

> **Note**: the permitted forms of a BYTE_LITERAL token ensure that these rules always produce a single character, whose Unicode scalar value is in the range of `u8`.

Examples of byte literal expressions:

```
b'R';                              // 82
b'\'';                             // 39
b'\x52';                           // 82
b'\xA0';                           // 160
```

Byte string literal expressions

A byte string literal expression consists of a single BYTE_STRING_LITERAL or RAW_BYTE_STRING_LITERAL token.

The expression's type is a shared reference (with `static` lifetime) to an array whose element type is `u8`. That is, the type is `&'static [u8; N]`, where `N` is the number of bytes in the represented string described below.

The token must not have a suffix.

The token's *literal content* is the sequence of characters following the first `U+0022` (`"`) and preceding the last `U+0022` (`"`) in the string representation of the token.

The literal expression's *represented string* is a sequence of characters derived from the literal content as follows:

- If the token is a BYTE_STRING_LITERAL, each escape sequence of any of the following forms occurring in the literal content is replaced by the escape sequence's escaped value.
    - Simple escapes
    - 8-bit escapes
    - String continuation escapes

  These replacements take place in left-to-right order. For example, the token `b"\\x41"` is converted to the characters `\` `x` `4` `1`.
- If the token is a RAW_BYTE_STRING_LITERAL, the represented string is identical to the literal content.

The expression's value is a reference to a statically allocated array containing the Unicode scalar values of the characters in the represented string, in the same order.

> **Note**: the permitted forms of BYTE_STRING_LITERAL and RAW_BYTE_STRING_LITERAL tokens ensure that these rules always produce array element values in the range of

`u8` .

Examples of byte string literal expressions:

```
b"foo"; br"foo";                     // foo
b"\"foo\""; br#""foo""#;             // "foo"

b"foo #\"# bar";
br##"foo #"# bar"##;                 // foo #"# bar

b"\x52"; b"R"; br"R";                // R
b"\\x52"; br"\x52";                  // \x52
```

C string literal expressions

A C string literal expression consists of a single C_STRING_LITERAL or RAW_C_STRING_LITERAL token.

The expression's type is a shared reference (with `static` lifetime) to the standard library CStr type. That is, the type is `&'static core::ffi::CStr` .

The token must not have a suffix.

The token's *literal content* is the sequence of characters following the first `"` and preceding the last `"` in the string representation of the token.

The literal expression's *represented bytes* are a sequence of bytes derived from the literal content as follows:

- If the token is a C_STRING_LITERAL, the literal content is treated as a sequence of items, each of which is either a single Unicode character other than `\` or an escape. The sequence of items is converted to a sequence of bytes as follows:
  - Each single Unicode character contributes its UTF-8 representation.
  - Each simple escape contributes the Unicode scalar value of its escaped value.
  - Each 8-bit escape contributes a single byte containing the Unicode scalar value of its escaped value.
  - Each unicode escape contributes the UTF-8 representation of its escaped value.
  - Each string continuation escape contributes no bytes.
- If the token is a RAW_C_STRING_LITERAL, the represented bytes are the UTF-8 encoding

of the literal content.

> **Note**: the permitted forms of C_STRING_LITERAL and RAW_C_STRING_LITERAL tokens ensure that the represented bytes never include a null byte.

The expression's value is a reference to a statically allocated CStr whose array of bytes contains the represented bytes followed by a null byte.

Examples of C string literal expressions:

```
c"foo"; cr"foo";                     // foo
c"\"foo\""; cr#""foo""#;             // "foo"

c"foo #\"# bar";
cr##"foo #"# bar"##;                 // foo #"# bar

c"\x52"; c"R"; cr"R";                // R
c"\\x52"; cr"\x52";                  // \x52

c"æ";                                // LATIN SMALL LETTER AE (U+00E6)
c"\u{00E6}";                         // LATIN SMALL LETTER AE (U+00E6)
c"\xC3\xA6";                         // LATIN SMALL LETTER AE (U+00E6)

c"\xE6".to_bytes();                  // [230]
c"\u{00E6}".to_bytes();              // [195, 166]
```

Integer literal expressions

An integer literal expression consists of a single INTEGER_LITERAL token.

If the token has a suffix, the suffix must be the name of one of the primitive integer types: `u8`, `i8`, `u16`, `i16`, `u32`, `i32`, `u64`, `i64`, `u128`, `i128`, `usize`, or `isize`, and the expression has that type.

If the token has no suffix, the expression's type is determined by type inference:

- If an integer type can be *uniquely* determined from the surrounding program context, the expression has that type.
- If the program context under-constrains the type, it defaults to the signed 32-bit integer `i32`.

- If the program context over-constrains the type, it is considered a static type error.

Examples of integer literal expressions:

```
123;                              // type i32
123i32;                           // type i32
123u32;                           // type u32
123_u32;                          // type u32
let a: u64 = 123;                 // type u64

0xff;                             // type i32
0xff_u8;                          // type u8

0o70;                             // type i32
0o70_i16;                         // type i16

0b1111_1111_1001_0000;            // type i32
0b1111_1111_1001_0000i64;         // type i64

0usize;                           // type usize
```

The value of the expression is determined from the string representation of the token as follows:

- An integer radix is chosen by inspecting the first two characters of the string, as follows:
  - `0b` indicates radix 2
  - `0o` indicates radix 8
  - `0x` indicates radix 16
  - otherwise the radix is 10.
- If the radix is not 10, the first two characters are removed from the string.
- Any suffix is removed from the string.
- Any underscores are removed from the string.
- The string is converted to a `u128` value as if by `u128::from_str_radix` with the chosen radix. If the value does not fit in `u128`, it is a compiler error.
- The `u128` value is converted to the expression's type via a numeric cast.

Note: The final cast will truncate the value of the literal if it does not fit in the expression's type. `rustc` includes a lint check named `overflowing_literals`, defaulting to `deny`, which rejects expressions where this occurs.

> **Note**: `-1i8`, for example, is an application of the negation operator to the literal expression `1i8`, not a single integer literal expression. See Overflow for notes on representing the most negative value for a signed type.

Floating-point literal expressions

A floating-point literal expression has one of two forms:

- a single FLOAT_LITERAL token
- a single INTEGER_LITERAL token which has a suffix and no radix indicator

If the token has a suffix, the suffix must be the name of one of the primitive floating-point types: `f32` or `f64`, and the expression has that type.

If the token has no suffix, the expression's type is determined by type inference:

- If a floating-point type can be *uniquely* determined from the surrounding program context, the expression has that type.
- If the program context under-constrains the type, it defaults to `f64`.
- If the program context over-constrains the type, it is considered a static type error.

Examples of floating-point literal expressions:

```
123.0f64;        // type f64
0.1f64;          // type f64
0.1f32;          // type f32
12E+99_f64;      // type f64
5f32;            // type f32
let x: f64 = 2.; // type f64
```

The value of the expression is determined from the string representation of the token as follows:

- Any suffix is removed from the string.
- Any underscores are removed from the string.
- The string is converted to the expression's type as if by `f32::from_str` or `f64::from_str`.

> **Note**: `-1.0`, for example, is an application of the negation operator to the literal expression `1.0`, not a single floating-point literal expression.

> **Note**: `inf` and `NaN` are not literal tokens. The `f32::INFINITY`, `f64::INFINITY`, `f32::NAN`, and `f64::NAN` constants can be used instead of literal expressions. In `rustc`, a literal large enough to be evaluated as infinite will trigger the `overflowing_literals` lint check.

Boolean literal expressions

A boolean literal expression consists of one of the keywords `true` or `false`.

The expression's type is the primitive boolean type, and its value is:

- true if the keyword is `true`
- false if the keyword is `false`

Path expressions

Syntax

PathExpression :
  PathInExpression
  | *QualifiedPathInExpression*

A path used as an expression context denotes either a local variable or an item. Path expressions that resolve to local or static variables are place expressions, other paths are value expressions. Using a `static mut` variable requires an `unsafe` block.

```
local_var;
globals::STATIC_VAR;
unsafe { globals::STATIC_MUT_VAR };
let some_constructor = Some::<i32>;
let push_integer = Vec::<i32>::push;
let slice_reverse = <[i32]>::reverse;
```

Block expressions

> **Syntax**
>
> *BlockExpression* :
>   `{`
>   *InnerAttribute**
>   *Statements*?
>   `}`
>
> *Statements* :
>   *Statement*+
> | *Statement*+ *ExpressionWithoutBlock*
> | *ExpressionWithoutBlock*

A *block expression*, or *block*, is a control flow expression and anonymous namespace scope for items and variable declarations. As a control flow expression, a block sequentially executes its component non-item declaration statements and then its final optional expression. As an anonymous namespace scope, item declarations are only in scope inside the block itself and variables declared by `let` statements are in scope from the next statement until the end of the block.

The syntax for a block is `{`, then any inner attributes, then any number of statements, then an optional expression, called the final operand, and finally a `}`.

Statements are usually required to be followed by a semicolon, with two exceptions:

1. Item declaration statements do not need to be followed by a semicolon.
2. Expression statements usually require a following semicolon except if its outer expression is a flow control expression.

Furthermore, extra semicolons between statements are allowed, but these semicolons do not affect semantics.

When evaluating a block expression, each statement, except for item declaration statements, is executed sequentially. Then the final operand is executed, if given.

The type of a block is the type of the final operand, or `()` if the final operand is omitted.

```
let _: () = {
    fn_call();
};

let five: i32 = {
    fn_call();
    5
};

assert_eq!(5, five);
```

> Note: As a control flow expression, if a block expression is the outer expression of an expression statement, the expected type is `()` unless it is followed immediately by a semicolon.

Blocks are always value expressions and evaluate the last operand in value expression context.

> **Note**: This can be used to force moving a value if really needed. For example, the following example fails on the call to `consume_self` because the struct was moved out of `s` in the block expression.
>
> ```
> struct Struct;
>
> impl Struct {
>     fn consume_self(self) {}
>     fn borrow_self(&self) {}
> }
>
> fn move_by_block_expression() {
>     let s = Struct;
>
>     // Move the value out of `s` in the block expression.
>     (&{ s }).borrow_self();
>
>     // Fails to execute because `s` is moved out of.
>     s.consume_self();
> }
> ```

`async` blocks

Syntax

AsyncBlockExpression :

`async` `move`? *BlockExpression*

An *async block* is a variant of a block expression which evaluates to a future. The final expression of the block, if present, determines the result value of the future.

Executing an async block is similar to executing a closure expression: its immediate effect is to produce and return an anonymous type. Whereas closures return a type that implements one or more of the `std::ops::Fn` traits, however, the type returned for an async block implements the `std::future::Future` trait. The actual data format for this type is unspecified.

> **Note:** The future type that rustc generates is roughly equivalent to an enum with one variant per `await` point, where each variant stores the data needed to resume from its corresponding point.

> **Edition differences**: Async blocks are only available beginning with Rust 2018.

Capture modes

Async blocks capture variables from their environment using the same capture modes as closures. Like closures, when written `async { .. }` the capture mode for each variable will be inferred from the content of the block. `async move { .. }` blocks however will move all referenced variables into the resulting future.

Async context

Because async blocks construct a future, they define an **async context** which can in turn contain `await` expressions. Async contexts are established by async blocks as well as the bodies of async functions, whose semantics are defined in terms of async blocks.

Control-flow operators

Async blocks act like a function boundary, much like closures. Therefore, the `?` operator and `return` expressions both affect the output of the future, not the enclosing function or other context. That is, `return <expr>` from within an async block will return the result of `<expr>` as the output of the future. Similarly, if `<expr>?` propagates an error, that error is propagated as the result of the future.

Finally, the `break` and `continue` keywords cannot be used to branch out from an async block. Therefore the following is illegal:

```
loop {
    async move {
        break; // error[E0267]: `break` inside of an `async` block
    }
}
```

`unsafe` blocks

> **Syntax**
> *UnsafeBlockExpression* :
>   `unsafe` *BlockExpression*

See [`unsafe` block] for more information on when to use `unsafe`

A block of code can be prefixed with the `unsafe` keyword to permit unsafe operations. Examples:

```
unsafe {
    let b = [13u8, 17u8];
    let a = &b[0] as *const u8;
    assert_eq!(*a, 13);
    assert_eq!(*a.offset(1), 17);
}

let a = unsafe { an_unsafe_fn() };
```

Labelled block expressions

Labelled block expressions are documented in the Loops and other breakable expressions section.

Attributes on block expressions

Inner attributes are allowed directly after the opening brace of a block expression in the following situations:

- Function and method bodies.
- Loop bodies (`loop` , `while` , `while let` , and `for`).
- Block expressions used as a statement.
- Block expressions as elements of array expressions, tuple expressions, call expressions, and tuple-style struct expressions.
- A block expression as the tail expression of another block expression.

The attributes that have meaning on a block expression are `cfg` and the lint check attributes.

For example, this function returns `true` on unix platforms and `false` on other platforms.

```
fn is_unix_platform() -> bool {
    #[cfg(unix)] { true }
    #[cfg(not(unix))] { false }
}
```

Operator expressions

Syntax

OperatorExpression :
  BorrowExpression
 | *DereferenceExpression*
 | *ErrorPropagationExpression*
 | *NegationExpression*
 | *ArithmeticOrLogicalExpression*
 | *ComparisonExpression*
 | *LazyBooleanExpression*
 | *TypeCastExpression*
 | *AssignmentExpression*
 | *CompoundAssignmentExpression*

Operators are defined for built in types by the Rust language. Many of the following operators can also be overloaded using traits in `std::ops` or `std::cmp`.

Overflow

Integer operators will panic when they overflow when compiled in debug mode. The `-C debug-assertions` and `-C overflow-checks` compiler flags can be used to control this more directly. The following things are considered to be overflow:

- When `+`, `*` or binary `-` create a value greater than the maximum value, or less than the minimum value that can be stored.
- Applying unary `-` to the most negative value of any signed integer type, unless the operand is a literal expression (or a literal expression standing alone inside one or more grouped expressions).
- Using `/` or `%`, where the left-hand argument is the smallest integer of a signed integer type and the right-hand argument is `-1`. These checks occur even when `-C overflow-checks` is disabled, for legacy reasons.
- Using `<<` or `>>` where the right-hand argument is greater than or equal to the number of bits in the type of the left-hand argument, or is negative.

Note: The exception for literal expressions behind unary `-` means that forms such as

`-128_i8` or `let j: i8 = -(128)` never cause a panic and have the expected value of -128.

In these cases, the literal expression already has the most negative value for its type (for example, `128_i8` has the value -128) because integer literals are truncated to their type per the description in Integer literal expressions.

Negation of these most negative values leaves the value unchanged due to two's complement overflow conventions.

In `rustc`, these most negative expressions are also ignored by the `overflowing_literals` lint check.

Borrow operators

Syntax

BorrowExpression :
   (`&` | `&&`) *Expression*
 | (`&` | `&&`) `mut` *Expression*

The `&` (shared borrow) and `&mut` (mutable borrow) operators are unary prefix operators. When applied to a place expression, this expressions produces a reference (pointer) to the location that the value refers to. The memory location is also placed into a borrowed state for the duration of the reference. For a shared borrow (`&`), this implies that the place may not be mutated, but it may be read or shared again. For a mutable borrow (`&mut`), the place may not be accessed in any way until the borrow expires. `&mut` evaluates its operand in a mutable place expression context. If the `&` or `&mut` operators are applied to a value expression, then a temporary value is created.

These operators cannot be overloaded.

```
{
    // a temporary with value 7 is created that lasts for this scope.
    let shared_reference = &7;
}
let mut array = [-2, 3, 9];
{
    // Mutably borrows `array` for this scope.
    // `array` may only be used through `mutable_reference`.
    let mutable_reference = &mut array;
}
```

Even though `&&` is a single token (the lazy 'and' operator), when used in the context of borrow expressions it works as two borrows:

```
// same meanings:
let a = &&  10;
let a = & & 10;

// same meanings:
let a = &&&&  mut 10;
let a = && && mut 10;
let a = & & & & mut 10;
```

Raw address-of operators

Related to the borrow operators are the *raw address-of operators*, which do not have first-class syntax, but are exposed via the macros `ptr::addr_of!(expr)` and `ptr::addr_of_mut!(expr)`. The expression `expr` is evaluated in place expression context. `ptr::addr_of!(expr)` then creates a const raw pointer of type `*const T` to the given place, and `ptr::addr_of_mut!(expr)` creates a mutable raw pointer of type `*mut T`.

The raw address-of operators must be used instead of a borrow operator whenever the place expression could evaluate to a place that is not properly aligned or does not store a valid value as determined by its type, or whenever creating a reference would introduce incorrect aliasing assumptions. In those situations, using a borrow operator would cause undefined behavior by creating an invalid reference, but a raw pointer may still be constructed using an address-of operator.

The following is an example of creating a raw pointer to an unaligned place through a `packed` struct:

```
use std::ptr;

#[repr(packed)]
struct Packed {
    f1: u8,
    f2: u16,
}

let packed = Packed { f1: 1, f2: 2 };
// `&packed.f2` would create an unaligned reference, and thus be Undefined
Behavior!
let raw_f2 = ptr::addr_of!(packed.f2);
assert_eq!(unsafe { raw_f2.read_unaligned() }, 2);
```

The following is an example of creating a raw pointer to a place that does not contain a valid value:

```
use std::{ptr, mem::MaybeUninit};

struct Demo {
    field: bool,
}

let mut uninit = MaybeUninit::<Demo>::uninit();
// `&uninit.as_mut().field` would create a reference to an uninitialized
`bool`,
// and thus be Undefined Behavior!
let f1_ptr = unsafe { ptr::addr_of_mut!((*uninit.as_mut_ptr()).field) };
unsafe { f1_ptr.write(true); }
let init = unsafe { uninit.assume_init() };
```

The dereference operator

> **Syntax**
>
> *DereferenceExpression* :
>   `*` *Expression*

The `*` (dereference) operator is also a unary prefix operator. When applied to a pointer it denotes the pointed-to location. If the expression is of type `&mut T` or `*mut T`, and is either a local variable, a (nested) field of a local variable or is a mutable place expression, then the resulting memory location can be assigned to. Dereferencing a raw pointer requires `unsafe`.

On non-pointer types `*x` is equivalent to `*std::ops::Deref::deref(&x)` in an immutable place expression context and `*std::ops::DerefMut::deref_mut(&mut x)` in a mutable place expression context.

```
let x = &7;
assert_eq!(*x, 7);
let y = &mut 9;
*y = 11;
assert_eq!(*y, 11);
```

The question mark operator

Syntax

ErrorPropagationExpression :
  Expression `?`

The question mark operator (`?`) unwraps valid values or returns erroneous values, propagating them to the calling function. It is a unary postfix operator that can only be applied to the types `Result<T, E>` and `Option<T>` .

When applied to values of the `Result<T, E>` type, it propagates errors. If the value is `Err(e)` , then it will return `Err(From::from(e))` from the enclosing function or closure. If applied to `Ok(x)` , then it will unwrap the value to evaluate to `x` .

```
fn try_to_parse() -> Result<i32, ParseIntError> {
    let x: i32 = "123".parse()?; // x = 123
    let y: i32 = "24a".parse()?; // returns an Err() immediately
    Ok(x + y)                    // Doesn't run.
}

let res = try_to_parse();
println!("{:?}", res);
```

When applied to values of the `Option<T>` type, it propagates `None` s. If the value is `None` , then it will return `None` . If applied to `Some(x)` , then it will unwrap the value to evaluate to `x` .

```
fn try_option_some() -> Option<u8> {
    let val = Some(1)?;
    Some(val)
}
assert_eq!(try_option_some(), Some(1));

fn try_option_none() -> Option<u8> {
    let val = None?;
    Some(val)
}
assert_eq!(try_option_none(), None);
```

`?` cannot be overloaded.

Negation operators

Syntax

NegationExpression :
  `-` *Expression*
| `!` *Expression*

These are the last two unary operators. This table summarizes the behavior of them on primitive types and which traits are used to overload these operators for other types. Remember that signed integers are always represented using two's complement. The operands of all of these operators are evaluated in value expression context so are moved or copied.

Symbol	Integer	`bool`	Floating Point	Overloading Trait
`-`	Negation*		Negation	`std::ops::Neg`
`!`	Bitwise NOT	Logical NOT		`std::ops::Not`

* Only for signed integer types.

Here are some example of these operators

```
let x = 6;
assert_eq!(-x, -6);
assert_eq!(!x, -7);
assert_eq!(true, !false);
```

Arithmetic and Logical Binary Operators

Syntax

ArithmeticOrLogicalExpression :
  Expression `+` *Expression*
 | *Expression* `-` *Expression*
 | *Expression* `*` *Expression*
 | *Expression* `/` *Expression*
 | *Expression* `%` *Expression*
 | *Expression* `&` *Expression*
 | *Expression* `|` *Expression*
 | *Expression* `^` *Expression*
 | *Expression* `<<` *Expression*
 | *Expression* `>>` *Expression*

Binary operators expressions are all written with infix notation. This table summarizes the behavior of arithmetic and logical binary operators on primitive types and which traits are used to overload these operators for other types. Remember that signed integers are always represented using two's complement. The operands of all of these operators are evaluated in value expression context so are moved or copied.

Symbol	Integer	`bool`	Floating Point	Overloading Trait
`+`	Addition		Addition	`std::ops::Add`
`-`	Subtraction		Subtraction	`std::ops::Sub`
`*`	Multiplication		Multiplication	`std::ops::Mul`
`/`	Division*†		Division	`std::ops::Div`
`%`	Remainder**†		Remainder	`std::ops::Rem`
`&`	Bitwise AND	Logical AND		`std::ops::BitAnd`
`|`	Bitwise OR	Logical OR		`std::ops::BitOr`
`^`	Bitwise XOR	Logical XOR		`std::ops::BitXor`
`<<`	Left Shift			`std::ops::Shl`
`>>`	Right Shift***			`std::ops::Shr`

* Integer division rounds towards zero.

** Rust uses a remainder defined with truncating division. Given `remainder = dividend % divisor`, the remainder will have the same sign as the dividend.

*** Arithmetic right shift on signed integer types, logical right shift on unsigned integer types.

† For integer types, division by zero panics.

Here are examples of these operators being used.

```
assert_eq!(3 + 6, 9);
assert_eq!(5.5 - 1.25, 4.25);
assert_eq!(-5 * 14, -70);
assert_eq!(14 / 3, 4);
assert_eq!(100 % 7, 2);
assert_eq!(0b1010 & 0b1100, 0b1000);
assert_eq!(0b1010 | 0b1100, 0b1110);
assert_eq!(0b1010 ^ 0b1100, 0b110);
assert_eq!(13 << 3, 104);
assert_eq!(-10 >> 2, -3);
```

Comparison Operators

Syntax

ComparisonExpression :
  Expression `==` *Expression*
  | *Expression* `!=` *Expression*
  | *Expression* `>` *Expression*
  | *Expression* `<` *Expression*
  | *Expression* `>=` *Expression*
  | *Expression* `<=` *Expression*

Comparison operators are also defined both for primitive types and many types in the standard library. Parentheses are required when chaining comparison operators. For example, the expression `a == b == c` is invalid and may be written as `(a == b) == c`.

Unlike arithmetic and logical operators, the traits for overloading these operators are used more generally to show how a type may be compared and will likely be assumed to define actual comparisons by functions that use these traits as bounds. Many functions and macros in the standard library can then use that assumption (although not to ensure safety). Unlike the arithmetic and logical operators above, these operators implicitly take shared borrows of their operands, evaluating them in place expression context:

```
a == b;
// is equivalent to
::std::cmp::PartialEq::eq(&a, &b);
```

This means that the operands don't have to be moved out of.

Symbol	Meaning	Overloading method
`==`	Equal	`std::cmp::PartialEq::eq`
`!=`	Not equal	`std::cmp::PartialEq::ne`
`>`	Greater than	`std::cmp::PartialOrd::gt`
`<`	Less than	`std::cmp::PartialOrd::lt`
`>=`	Greater than or equal to	`std::cmp::PartialOrd::ge`
`<=`	Less than or equal to	`std::cmp::PartialOrd::le`

Here are examples of the comparison operators being used.

```
assert!(123 == 123);
assert!(23 != -12);
assert!(12.5 > 12.2);
assert!([1, 2, 3] < [1, 3, 4]);
assert!('A' <= 'B');
assert!("World" >= "Hello");
```

Lazy boolean operators

Syntax

LazyBooleanExpression :
   Expression `||` *Expression*
 | *Expression* `&&` *Expression*

The operators `||` and `&&` may be applied to operands of boolean type. The `||` operator denotes logical 'or', and the `&&` operator denotes logical 'and'. They differ from `|` and `&` in that the right-hand operand is only evaluated when the left-hand operand does not already determine the result of the expression. That is, `||` only evaluates its right-hand operand when the left-hand operand evaluates to `false`, and `&&` only when it evaluates to `true`.

```
let x = false || true; // true
let y = false && panic!(); // false, doesn't evaluate `panic!()`
```

Type cast expressions

Syntax

TypeCastExpression :
  Expression `as` *TypeNoBounds*

A type cast expression is denoted with the binary operator `as`.

Executing an `as` expression casts the value on the left-hand side to the type on the right-hand side.

An example of an `as` expression:

```
fn average(values: &[f64]) -> f64 {
    let sum: f64 = sum(values);
    let size: f64 = len(values) as f64;
    sum / size
}
```

`as` can be used to explicitly perform coercions, as well as the following additional casts. Any cast that does not fit either a coercion rule or an entry in the table is a compiler error. Here `*T` means either `*const T` or `*mut T`. `m` stands for optional `mut` in reference types and `mut` or `const` in pointer types.

Type of `e`	`U`	Cast performed by `e as U`
Integer or Float type	Integer or Float type	Numeric cast
Enumeration	Integer type	Enum cast
`bool` or `char`	Integer type	Primitive to integer cast
`u8`	`char`	`u8` to `char` cast

Type of `e`	`U`	Cast performed by `e as U`
`*T`	`*V` where `V: Sized` *	Pointer to pointer cast
`*T` where `T: Sized`	Integer type	Pointer to address cast
Integer type	`*V` where `V: Sized`	Address to pointer cast
`&m₁ T`	`*m₂ T` **	Reference to pointer cast
`&m₁ [T; n]`	`*m₂ T` **	Array to pointer cast
Function item	Function pointer	Function item to function pointer cast
Function item	`*V` where `V: Sized`	Function item to pointer cast
Function item	Integer	Function item to address cast
Function pointer	`*V` where `V: Sized`	Function pointer to pointer cast
Function pointer	Integer	Function pointer to address cast
Closure ***	Function pointer	Closure to function pointer cast

* or `T` and `V` are compatible unsized types, e.g., both slices, both the same trait object.

** only when `m₁` is `mut` or `m₂` is `const`. Casting `mut` reference to `const` pointer is allowed.

*** only for closures that do not capture (close over) any local variables

Semantics

Numeric cast

- Casting between two integers of the same size (e.g. i32 -> u32) is a no-op (Rust uses 2's complement for negative values of fixed integers)
- Casting from a larger integer to a smaller integer (e.g. u32 -> u8) will truncate
- Casting from a smaller integer to a larger integer (e.g. u8 -> u32) will
  - zero-extend if the source is unsigned
  - sign-extend if the source is signed
- Casting from a float to an integer will round the float towards zero
  - `NaN` will return `0`
  - Values larger than the maximum integer value, including `INFINITY`, will saturate to the maximum value of the integer type.

    - Values smaller than the minimum integer value, including `NEG_INFINITY`, will saturate to the minimum value of the integer type.
- Casting from an integer to float will produce the closest possible float *
    - if necessary, rounding is according to `roundTiesToEven` mode ***
    - on overflow, infinity (of the same sign as the input) is produced
    - note: with the current set of numeric types, overflow can only happen on `u128 as f32` for values greater or equal to `f32::MAX + (0.5 ULP)`
- Casting from an f32 to an f64 is perfect and lossless
- Casting from an f64 to an f32 will produce the closest possible f32 **
    - if necessary, rounding is according to `roundTiesToEven` mode ***
    - on overflow, infinity (of the same sign as the input) is produced

* if integer-to-float casts with this rounding mode and overflow behavior are not supported natively by the hardware, these casts will likely be slower than expected.

** if f64-to-f32 casts with this rounding mode and overflow behavior are not supported natively by the hardware, these casts will likely be slower than expected.

*** as defined in IEEE 754-2008 §4.3.1: pick the nearest floating point number, preferring the one with an even least significant digit if exactly halfway between two floating point numbers.

Enum cast

Casts an enum to its discriminant, then uses a numeric cast if needed. Casting is limited to the following kinds of enumerations:

- Unit-only enums
- Field-less enums without explicit discriminants, or where only unit-variants have explicit discriminants

Primitive to integer cast

- `false` casts to `0`, `true` casts to `1`
- `char` casts to the value of the code point, then uses a numeric cast if needed.

`u8` to `char` cast

Casts to the `char` with the corresponding code point.

Pointer to address cast

Casting from a raw pointer to an integer produces the machine address of the referenced memory. If the integer type is smaller than the pointer type, the address may be truncated; using `usize` avoids this.

Address to pointer cast

Casting from an integer to a raw pointer interprets the integer as a memory address and produces a pointer referencing that memory.

> ⚠️ Warning: This interacts with the Rust memory model, which is still under development. A pointer obtained from this cast may suffer additional restrictions even if it is bitwise equal to a valid pointer. Dereferencing such a pointer may be undefined behavior if aliasing rules are not followed.

A trivial example of sound address arithmetic:

```
let mut values: [i32; 2] = [1, 2];
let p1: *mut i32 = values.as_mut_ptr();
let first_address = p1 as usize;
let second_address = first_address + 4; // 4 == size_of::<i32>()
let p2 = second_address as *mut i32;
unsafe {
    *p2 += 1;
}
assert_eq!(values[1], 3);
```

Pointer-to-pointer cast

`*const T` / `*mut T` can be cast to `*const U` / `*mut U` with the following behavior:

- If `T` and `U` are both sized, the pointer is returned unchanged.
- If `T` and `U` are both unsized, the pointer is also returned unchanged. In particular, the metadata is preserved exactly.

  For instance, a cast from `*const [T]` to `*const [U]` preserves the number of elements. Note that, as a consequence, such casts do not necessarily preserve the size of the pointer's referent (e.g., casting `*const [u16]` to `*const [u8]` will result in a raw pointer which refers to an object of half the size of the original). The same holds for `str` and any compound type whose unsized tail is a slice type, such as `struct`

`Foo(i32, [u8])` or `(u64, Foo)`.

- If `T` is unsized and `U` is sized, the cast discards all metadata that completes the wide pointer `T` and produces a thin pointer `U` consisting of the data part of the unsized pointer.

Assignment expressions

> **Syntax**
>
> *AssignmentExpression* :
>   *Expression* `=` *Expression*

An *assignment expression* moves a value into a specified place.

An assignment expression consists of a mutable assignee expression, the *assignee operand*, followed by an equals sign (`=`) and a value expression, the *assigned value operand*. In its most basic form, an assignee expression is a place expression, and we discuss this case first. The more general case of destructuring assignment is discussed below, but this case always decomposes into sequential assignments to place expressions, which may be considered the more fundamental case.

Basic assignments

Evaluating assignment expressions begins by evaluating its operands. The assigned value operand is evaluated first, followed by the assignee expression. For destructuring assignment, subexpressions of the assignee expression are evaluated left-to-right.

> **Note**: This is different than other expressions in that the right operand is evaluated before the left one.

It then has the effect of first dropping the value at the assigned place, unless the place is an uninitialized local variable or an uninitialized field of a local variable. Next it either copies or moves the assigned value to the assigned place.

An assignment expression always produces the unit value.

Example:

```
let mut x = 0;
let y = 0;
x = y;
```

Destructuring assignments

Destructuring assignment is a counterpart to destructuring pattern matches for variable declaration, permitting assignment to complex values, such as tuples or structs. For instance, we may swap two mutable variables:

```
let (mut a, mut b) = (0, 1);
// Swap `a` and `b` using destructuring assignment.
(b, a) = (a, b);
```

In contrast to destructuring declarations using `let`, patterns may not appear on the left-hand side of an assignment due to syntactic ambiguities. Instead, a group of expressions that correspond to patterns are designated to be assignee expressions, and permitted on the left-hand side of an assignment. Assignee expressions are then desugared to pattern matches followed by sequential assignment. The desugared patterns must be irrefutable: in particular, this means that only slice patterns whose length is known at compile-time, and the trivial slice `[..]`, are permitted for destructuring assignment.

The desugaring method is straightforward, and is illustrated best by example.

```
(a, b) = (3, 4);

[a, b] = [3, 4];

Struct { x: a, y: b } = Struct { x: 3, y: 4};

// desugars to:

{
    let (_a, _b) = (3, 4);
    a = _a;
    b = _b;
}

{
    let [_a, _b] = [3, 4];
    a = _a;
    b = _b;
}

{
    let Struct { x: _a, y: _b } = Struct { x: 3, y: 4};
    a = _a;
    b = _b;
}
```

Identifiers are not forbidden from being used multiple times in a single assignee expression.

Underscore expressions and empty range expressions may be used to ignore certain values, without binding them.

Note that default binding modes do not apply for the desugared expression.

Compound assignment expressions

Syntax

CompoundAssignmentExpression :
  Expression `+=` *Expression*
  | *Expression* `-=` *Expression*
  | *Expression* `*=` *Expression*
  | *Expression* `/=` *Expression*
  | *Expression* `%=` *Expression*
  | *Expression* `&=` *Expression*

| *Expression* `|=` *Expression*
| *Expression* `^=` *Expression*
| *Expression* `<<=` *Expression*
| *Expression* `>>=` *Expression*

Compound assignment expressions combine arithmetic and logical binary operators with assignment expressions.

For example:

```
let mut x = 5;
x += 1;
assert!(x == 6);
```

The syntax of compound assignment is a mutable place expression, the *assigned operand*, then one of the operators followed by an `=` as a single token (no whitespace), and then a value expression, the *modifying operand*.

Unlike other place operands, the assigned place operand must be a place expression. Attempting to use a value expression is a compiler error rather than promoting it to a temporary.

Evaluation of compound assignment expressions depends on the types of the operators.

If both types are primitives, then the modifying operand will be evaluated first followed by the assigned operand. It will then set the value of the assigned operand's place to the value of performing the operation of the operator with the values of the assigned operand and modifying operand.

> **Note**: This is different than other expressions in that the right operand is evaluated before the left one.

Otherwise, this expression is syntactic sugar for calling the function of the overloading compound assignment trait of the operator (see the table earlier in this chapter). A mutable borrow of the assigned operand is automatically taken.

For example, the following expression statements in `example` are equivalent:

```
impl AddAssign<Addable> for Addable {
    /* */
}

fn example() {
  a1 += a2;

  AddAssign::add_assign(&mut a1, a2);
}
```

Like assignment expressions, compound assignment expressions always produce the unit value.

> ⚠️ Warning: The evaluation order of operands swaps depending on the types of the operands: with primitive types the right-hand side will get evaluated first, while with non-primitive types the left-hand side will get evaluated first. Try not to write code that depends on the evaluation order of operands in compound assignment expressions. See this test for an example of using this dependency.

Grouped expressions

> **Syntax**
>
> *GroupedExpression* :
>   `(` *Expression* `)`

A *parenthesized expression* wraps a single expression, evaluating to that expression. The syntax for a parenthesized expression is a `(`, then an expression, called the *enclosed operand*, and then a `)`.

Parenthesized expressions evaluate to the value of the enclosed operand. Unlike other expressions, parenthesized expressions are both place expressions and value expressions. When the enclosed operand is a place expression, it is a place expression and when the enclosed operand is a value expression, it is a value expression.

Parentheses can be used to explicitly modify the precedence order of subexpressions within an expression.

An example of a parenthesized expression:

```
let x: i32 = 2 + 3 * 4; // not parenthesized
let y: i32 = (2 + 3) * 4; // parenthesized
assert_eq!(x, 14);
assert_eq!(y, 20);
```

An example of a necessary use of parentheses is when calling a function pointer that is a member of a struct:

```
assert_eq!( a.f (), "The method f");
assert_eq!((a.f)(), "The field f");
```

Array and array index expressions

Array expressions

Syntax

ArrayExpression :
  `[` *ArrayElements*? `]`

ArrayElements :
  Expression (`,` *Expression*)* `,` ?
  | *Expression* `;` *Expression*

Array expressions construct arrays. Array expressions come in two forms.

The first form lists out every value in the array. The syntax for this form is a comma-separated list of expressions of uniform type enclosed in square brackets. This produces an array containing each of these values in the order they are written.

The syntax for the second form is two expressions separated by a semicolon (`;`) enclosed in square brackets. The expression before the `;` is called the *repeat operand*. The expression after the `;` is called the *length operand*. It must have type `usize` and be a constant expression, such as a literal or a constant item. An array expression of this form creates an array with the length of the value of the length operand with each element being a copy of the repeat operand. That is, `[a; b]` creates an array containing `b` copies of the value of `a` . If the length operand has a value greater than 1 then this requires that the type of the repeat operand is `Copy` or that it must be a path to a constant item.

When the repeat operand is a constant item, it is evaluated the length operand's value times. If that value is `0` , then the constant item is not evaluated at all. For expressions that are not a constant item, it is evaluated exactly once, and then the result is copied the length operand's value times.

```
[1, 2, 3, 4];
["a", "b", "c", "d"];
[0; 128];              // array with 128 zeros
[0u8, 0u8, 0u8, 0u8,];
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]; // 2D array
const EMPTY: Vec<i32> = Vec::new();
[EMPTY; 2];
```

Array and slice indexing expressions

Syntax

IndexExpression :
  Expression `[` *Expression* `]`

Array and slice-typed values can be indexed by writing a square-bracket-enclosed expression of type `usize` (the index) after them. When the array is mutable, the resulting memory location can be assigned to.

For other types an index expression `a[b]` is equivalent to `*std::ops::Index::index(&a, b)`, or `*std::ops::IndexMut::index_mut(&mut a, b)` in a mutable place expression context. Just as with methods, Rust will also insert dereference operations on `a` repeatedly to find an implementation.

Indices are zero-based for arrays and slices. Array access is a constant expression, so bounds can be checked at compile-time with a constant index value. Otherwise a check will be performed at run-time that will put the thread in a *panicked state* if it fails.

```
// lint is deny by default.
#![warn(unconditional_panic)]

([1, 2, 3, 4])[2];        // Evaluates to 3

let b = [[1, 0, 0], [0, 1, 0], [0, 0, 1]];
b[1][2];                  // multidimensional array indexing

let x = (["a", "b"])[10]; // warning: index out of bounds

let n = 10;
let y = (["a", "b"])[n];  // panics

let arr = ["a", "b"];
arr[10];                  // warning: index out of bounds
```

The array index expression can be implemented for types other than arrays and slices by implementing the Index and IndexMut traits.

Tuple and tuple indexing expressions

Tuple expressions

> **Syntax**
>
> *TupleExpression* :
>
> `(` *TupleElements*? `)`
>
> *TupleElements* :
>
> (*Expression* `,`)+ *Expression*?

A *tuple expression* constructs tuple values.

The syntax for tuple expressions is a parenthesized, comma separated list of expressions, called the *tuple initializer operands*. 1-ary tuple expressions require a comma after their tuple initializer operand to be disambiguated with a parenthetical expression.

Tuple expressions are a value expression that evaluate into a newly constructed value of a tuple type. The number of tuple initializer operands is the arity of the constructed tuple. Tuple expressions without any tuple initializer operands produce the unit tuple. For other tuple expressions, the first written tuple initializer operand initializes the field `0` and subsequent operands initializes the next highest field. For example, in the tuple expression `('a', 'b', 'c')`, `'a'` initializes the value of the field `0`, `'b'` field `1`, and `'c'` field `2`.

Examples of tuple expressions and their types:

Expression	Type
`()`	`()` (unit)
`(0.0, 4.5)`	`(f64, f64)`
`("x".to_string(), )`	`(String, )`
`("a", 4usize, true)`	`(&'static str, usize, bool)`

Tuple indexing expressions

Syntax

TupleIndexingExpression :
  Expression `.` TUPLE_INDEX

A *tuple indexing expression* accesses fields of tuples and tuple structs.

The syntax for a tuple index expression is an expression, called the *tuple operand*, then a `.`, then finally a tuple index. The syntax for the *tuple index* is a decimal literal with no leading zeros, underscores, or suffix. For example `0` and `2` are valid tuple indices but not `01`, `0_`, nor `0i32`.

The type of the tuple operand must be a tuple type or a tuple struct. The tuple index must be a name of a field of the type of the tuple operand.

Evaluation of tuple index expressions has no side effects beyond evaluation of its tuple operand. As a place expression, it evaluates to the location of the field of the tuple operand with the same name as the tuple index.

Examples of tuple indexing expressions:

```
// Indexing a tuple
let pair = ("a string", 2);
assert_eq!(pair.1, 2);

// Indexing a tuple struct
let point = Point(1.0, 0.0);
assert_eq!(point.0, 1.0);
assert_eq!(point.1, 0.0);
```

Note: Unlike field access expressions, tuple index expressions can be the function operand of a call expression as it cannot be confused with a method call since method names cannot be numbers.

Note: Although arrays and slices also have elements, you must use an array or slice indexing expression or a slice pattern to access their elements.

Struct expressions

> **Syntax**
>
> *StructExpression* :
>   *StructExprStruct*
> | *StructExprTuple*
> | *StructExprUnit*
>
> *StructExprStruct* :
>   *PathInExpression* `{` (*StructExprFields* | *StructBase*)? `}`
>
> *StructExprFields* :
>   *StructExprField* (`,` *StructExprField*)* (`,` *StructBase* | `,`?)
>
> *StructExprField* :
>   *OuterAttribute* *
>   (
>   IDENTIFIER
> | (IDENTIFIER | TUPLE_INDEX) `:` *Expression*
>   )
>
> *StructBase* :
>   `..` *Expression*
>
> *StructExprTuple* :
>   *PathInExpression* `(`
>   (*Expression* (`,` *Expression*)* `,`?)?
>   `)`
>
> *StructExprUnit* : *PathInExpression*

A *struct expression* creates a struct, enum, or union value. It consists of a path to a struct, enum variant, or union item followed by the values for the fields of the item. There are three forms of struct expressions: struct, tuple, and unit.

The following are examples of struct expressions:

```
Point {x: 10.0, y: 20.0};
NothingInMe {};
TuplePoint(10.0, 20.0);
TuplePoint { 0: 10.0, 1: 20.0 }; // Results in the same value as the above line
let u = game::User {name: "Joe", age: 35, score: 100_000};
some_fn::<Cookie>(Cookie);
```

Field struct expression

A struct expression with fields enclosed in curly braces allows you to specify the value for each individual field in any order. The field name is separated from its value with a colon.

A value of a union type can only be created using this syntax, and it must specify exactly one field.

Functional update syntax

A struct expression that constructs a value of a struct type can terminate with the syntax `..` followed by an expression to denote a functional update. The expression following `..` (the base) must have the same struct type as the new struct type being formed.

The entire expression uses the given values for the fields that were specified and moves or copies the remaining fields from the base expression. As with all struct expressions, all of the fields of the struct must be visible, even those not explicitly named.

```
let mut base = Point3d {x: 1, y: 2, z: 3};
let y_ref = &mut base.y;
Point3d {y: 0, z: 10, .. base}; // OK, only base.x is accessed
drop(y_ref);
```

Struct expressions with curly braces can't be used directly in a loop or if expression's head, or in the scrutinee of an if let or match expression. However, struct expressions can be used in these situations if they are within another expression, for example inside parentheses.

The field names can be decimal integer values to specify indices for constructing tuple structs. This can be used with base structs to fill out the remaining indices not specified:

```
struct Color(u8, u8, u8);
let c1 = Color(0, 0, 0);  // Typical way of creating a tuple struct.
let c2 = Color{0: 255, 1: 127, 2: 0};  // Specifying fields by index.
let c3 = Color{1: 0, ..c2};  // Fill out all other fields using a base struct.
```

Struct field init shorthand

When initializing a data structure (struct, enum, union) with named (but not numbered) fields, it is allowed to write `fieldname` as a shorthand for `fieldname: fieldname`. This allows a compact syntax with less duplication. For example:

```
Point3d { x: x, y: y_value, z: z };
Point3d { x, y: y_value, z };
```

Tuple struct expression

A struct expression with fields enclosed in parentheses constructs a tuple struct. Though it is listed here as a specific expression for completeness, it is equivalent to a call expression to the tuple struct's constructor. For example:

```
struct Position(i32, i32, i32);
Position(0, 0, 0);  // Typical way of creating a tuple struct.
let c = Position;  // `c` is a function that takes 3 arguments.
let pos = c(8, 6, 7);  // Creates a `Position` value.
```

Unit struct expression

A unit struct expression is just the path to a unit struct item. This refers to the unit struct's implicit constant of its value. The unit struct value can also be constructed with a fieldless struct expression. For example:

```
struct Gamma;
let a = Gamma;  // Gamma unit value.
let b = Gamma{};  // Exact same value as `a`.
```

Call expressions

Syntax

CallExpression :
  Expression `(` *CallParams*? `)`

CallParams :
  Expression (`,` *Expression*)* `,`?

A *call expression* calls a function. The syntax of a call expression is an expression, called the *function operand*, followed by a parenthesized comma-separated list of expression, called the *argument operands*. If the function eventually returns, then the expression completes. For non-function types, the expression `f(...)` uses the method on one of the `std::ops::Fn`, `std::ops::FnMut` or `std::ops::FnOnce` traits, which differ in whether they take the type by reference, mutable reference, or take ownership respectively. An automatic borrow will be taken if needed. The function operand will also be automatically dereferenced as required.

Some examples of call expressions:

```
let three: i32 = add(1i32, 2i32);
let name: &'static str = (|| "Rust")();
```

Disambiguating Function Calls

All function calls are sugar for a more explicit fully-qualified syntax. Function calls may need to be fully qualified, depending on the ambiguity of a call in light of in-scope items.

Note: In the past, the terms "Unambiguous Function Call Syntax", "Universal Function Call Syntax", or "UFCS", have been used in documentation, issues, RFCs, and other community writings. However, these terms lack descriptive power and potentially confuse the issue at hand. We mention them here for searchability's sake.

Several situations often occur which result in ambiguities about the receiver or referent of method or associated function calls. These situations may include:

- Multiple in-scope traits define methods with the same name for the same types

- Auto- `deref` is undesirable; for example, distinguishing between methods on a smart pointer itself and the pointer's referent
- Methods which take no arguments, like `default()`, and return properties of a type, like `size_of()`

To resolve the ambiguity, the programmer may refer to their desired method or function using more specific paths, types, or traits.

For example,

```
trait Pretty {
    fn print(&self);
}

trait Ugly {
    fn print(&self);
}

struct Foo;
impl Pretty for Foo {
    fn print(&self) {}
}

struct Bar;
impl Pretty for Bar {
    fn print(&self) {}
}
impl Ugly for Bar {
    fn print(&self) {}
}

fn main() {
    let f = Foo;
    let b = Bar;

    // we can do this because we only have one item called `print` for `Foo`s
    f.print();
    // more explicit, and, in the case of `Foo`, not necessary
    Foo::print(&f);
    // if you're not into the whole brevity thing
    <Foo as Pretty>::print(&f);

    // b.print(); // Error: multiple 'print' found
    // Bar::print(&b); // Still an error: multiple `print` found

    // necessary because of in-scope items defining `print`
    <Bar as Pretty>::print(&b);
}
```

Refer to RFC 132 for further details and motivations.

Method-call expressions

> **Syntax**
>
> *MethodCallExpression* :
>
> *Expression* `.` *PathExprSegment* `(` *CallParams*? `)`

A *method call* consists of an expression (the *receiver*) followed by a single dot, an expression path segment, and a parenthesized expression-list. Method calls are resolved to associated methods on specific traits, either statically dispatching to a method if the exact `self` -type of the left-hand-side is known, or dynamically dispatching if the left-hand-side expression is an indirect trait object.

```
let pi: Result<f32, _> = "3.14".parse();
let log_pi = pi.unwrap_or(1.0).log(2.72);
```

When looking up a method call, the receiver may be automatically dereferenced or borrowed in order to call a method. This requires a more complex lookup process than for other functions, since there may be a number of possible methods to call. The following procedure is used:

The first step is to build a list of candidate receiver types. Obtain these by repeatedly dereferencing the receiver expression's type, adding each type encountered to the list, then finally attempting an unsized coercion at the end, and adding the result type if that is successful. Then, for each candidate `T` , add `&T` and `&mut T` to the list immediately after `T` .

For instance, if the receiver has type `Box<[i32;2]>` , then the candidate types will be `Box<[i32;2]>` , `&Box<[i32;2]>` , `&mut Box<[i32;2]>` , `[i32; 2]` (by dereferencing), `&[i32; 2]` , `&mut [i32; 2]` , `[i32]` (by unsized coercion), `&[i32]` , and finally `&mut [i32]` .

Then, for each candidate type `T` , search for a visible method with a receiver of that type in the following places:

1. `T` 's inherent methods (methods implemented directly on `T`).
2. Any of the methods provided by a visible trait implemented by `T` . If `T` is a type parameter, methods provided by trait bounds on `T` are looked up first. Then all remaining methods in scope are looked up.

> Note: the lookup is done for each type in order, which can occasionally lead to surprising results. The below code will print "In trait impl!", because `&self` methods

are looked up first, the trait method is found before the struct's `&mut self` method is found.

```
struct Foo {}

trait Bar {
  fn bar(&self);
}

impl Foo {
  fn bar(&mut self) {
    println!("In struct impl!")
  }
}

impl Bar for Foo {
  fn bar(&self) {
    println!("In trait impl!")
  }
}

fn main() {
  let mut f = Foo{};
  f.bar();
}
```

If this results in multiple possible candidates, then it is an error, and the receiver must be converted to an appropriate receiver type to make the method call.

This process does not take into account the mutability or lifetime of the receiver, or whether a method is `unsafe`. Once a method is looked up, if it can't be called for one (or more) of those reasons, the result is a compiler error.

If a step is reached where there is more than one possible method, such as where generic methods or traits are considered the same, then it is a compiler error. These cases require a disambiguating function call syntax for method and function invocation.

Edition Differences: Before the 2021 edition, during the search for visible methods, if the candidate receiver type is an array type, methods provided by the standard library `IntoIterator` trait are ignored.

The edition used for this purpose is determined by the token representing the method name.

This special case may be removed in the future.

> ⚠️ ***Warning:*** For trait objects, if there is an inherent method of the same name as a trait method, it will give a compiler error when trying to call the method in a method call expression. Instead, you can call the method using disambiguating function call syntax, in which case it calls the trait method, not the inherent method. There is no way to call the inherent method. Just don't define inherent methods on trait objects with the same name as a trait method and you'll be fine.

Field access expressions

> **Syntax**
>
> *FieldExpression* :
>   *Expression* `.` IDENTIFIER

A *field expression* is a place expression that evaluates to the location of a field of a struct or union. When the operand is mutable, the field expression is also mutable.

The syntax for a field expression is an expression, called the *container operand*, then a `.`, and finally an identifier. Field expressions cannot be followed by a parenthetical comma-separated list of expressions, as that is instead parsed as a method call expression. That is, they cannot be the function operand of a call expression.

> **Note**: Wrap the field expression in a parenthesized expression to use it in a call expression.
>
> ```
> let holds_callable = HoldsCallable { callable: || () };
>
> // Invalid: Parsed as calling the method "callable"
> // holds_callable.callable();
>
> // Valid
> (holds_callable.callable)();
> ```

Examples:

```
mystruct.myfield;
foo().x;
(Struct {a: 10, b: 20}).a;
(mystruct.function_field)() // Call expression containing a field expression
```

Automatic dereferencing

If the type of the container operand implements `Deref` or `DerefMut` depending on whether the operand is mutable, it is *automatically dereferenced* as many times as necessary to make the field access possible. This process is also called *autoderef* for short.

Borrowing

The fields of a struct or a reference to a struct are treated as separate entities when borrowing. If the struct does not implement `Drop` and is stored in a local variable, this also applies to moving out of each of its fields. This also does not apply if automatic dereferencing is done though user-defined types other than `Box` .

```
struct A { f1: String, f2: String, f3: String }
let mut x: A;
let a: &mut String = &mut x.f1; // x.f1 borrowed mutably
let b: &String = &x.f2;         // x.f2 borrowed immutably
let c: &String = &x.f2;         // Can borrow again
let d: String = x.f3;           // Move out of x.f3
```

Closure expressions

Syntax

ClosureExpression :
  `move` ?
  (`||` | `|` *ClosureParameters*? `|`)
  (*Expression* | `->` *TypeNoBounds BlockExpression*)

ClosureParameters :
  ClosureParam (`,` *ClosureParam*)* `,` ?

ClosureParam :
  *OuterAttribute** *PatternNoTopAlt* (`:` *Type*)?

A *closure expression*, also known as a lambda expression or a lambda, defines a closure type and evaluates to a value of that type. The syntax for a closure expression is an optional `move` keyword, then a pipe-symbol-delimited (`|`) comma-separated list of patterns, called the *closure parameters* each optionally followed by a `:` and a type, then an optional `->` and type, called the *return type*, and then an expression, called the *closure body operand*. The optional type after each pattern is a type annotation for the pattern. If there is a return type, the closure body must be a block.

A closure expression denotes a function that maps a list of parameters onto the expression that follows the parameters. Just like a `let` binding, the closure parameters are irrefutable patterns, whose type annotation is optional and will be inferred from context if not given. Each closure expression has a unique, anonymous type.

Significantly, closure expressions *capture their environment*, which regular function definitions do not. Without the `move` keyword, the closure expression infers how it captures each variable from its environment, preferring to capture by shared reference, effectively borrowing all outer variables mentioned inside the closure's body. If needed the compiler will infer that instead mutable references should be taken, or that the values should be moved or copied (depending on their type) from the environment. A closure can be forced to capture its environment by copying or moving values by prefixing it with the `move` keyword. This is often used to ensure that the closure's lifetime is `'static`.

Closure trait implementations

Which traits the closure type implement depends on how variables are captured and the types of the captured variables. See the call traits and coercions chapter for how and when a closure implements `Fn`, `FnMut`, and `FnOnce`. The closure type implements `Send` and `Sync` if the type of every captured variable also implements the trait.

Example

In this example, we define a function `ten_times` that takes a higher-order function argument, and we then call it with a closure expression as an argument, followed by a closure expression that moves values from its environment.

```
fn ten_times<F>(f: F) where F: Fn(i32) {
    for index in 0..10 {
        f(index);
    }
}

ten_times(|j| println!("hello, {}", j));
// With type annotations
ten_times(|j: i32| -> () { println!("hello, {}", j) });

let word = "konnichiwa".to_owned();
ten_times(move |j| println!("{}, {}", word, j));
```

Attributes on closure parameters

Attributes on closure parameters follow the same rules and restrictions as regular function parameters.

Loops and other breakable expressions

Syntax

LoopExpression :
  LoopLabel? (
    InfiniteLoopExpression
   | *PredicateLoopExpression*
   | *PredicatePatternLoopExpression*
   | *IteratorLoopExpression*
   | *LabelBlockExpression*
)

Rust supports five loop expressions:

- A `loop` expression denotes an infinite loop.
- A `while` expression loops until a predicate is false.
- A `while let` expression tests a pattern.
- A `for` expression extracts values from an iterator, looping until the iterator is empty.
- A labelled block expression runs a loop exactly once, but allows exiting the loop early with `break`.

All five types of loop support `break` expressions, and labels. All except labelled block expressions support `continue` expressions. Only `loop` and labelled block expressions support evaluation to non-trivial values.

Infinite loops

Syntax

InfiniteLoopExpression :
  `loop` *BlockExpression*

A `loop` expression repeats execution of its body continuously: `loop { println!("I live."); }`.

A `loop` expression without an associated `break` expression is diverging and has type `!`. A `loop` expression containing associated `break` expression(s) may terminate, and must have type compatible with the value of the `break` expression(s).

Predicate loops

> **Syntax**
>
> *PredicateLoopExpression* :
>
> `while` *Expression*except struct expression *BlockExpression*

A `while` loop begins by evaluating the boolean loop conditional operand. If the loop conditional operand evaluates to `true`, the loop body block executes, then control returns to the loop conditional operand. If the loop conditional expression evaluates to `false`, the `while` expression completes.

An example:

```
let mut i = 0;

while i < 10 {
    println!("hello");
    i = i + 1;
}
```

Predicate pattern loops

> **Syntax**
>
> *PredicatePatternLoopExpression* :
>
> `while` `let` *Pattern* `=` *Scrutinee*except lazy boolean operator expression *BlockExpression*

A `while let` loop is semantically similar to a `while` loop but in place of a condition expression it expects the keyword `let` followed by a pattern, an `=`, a scrutinee expression and a block expression. If the value of the scrutinee matches the pattern, the loop body block executes then control returns to the pattern matching statement. Otherwise, the while expression completes.

```
let mut x = vec![1, 2, 3];

while let Some(y) = x.pop() {
    println!("y = {}", y);
}

while let _ = 5 {
    println!("Irrefutable patterns are always true");
    break;
}
```

A `while let` loop is equivalent to a `loop` expression containing a `match` expression as follows.

```
'label: while let PATS = EXPR {
    /* loop body */
}
```

is equivalent to

```
'label: loop {
    match EXPR {
        PATS => { /* loop body */ },
        _ => break,
    }
}
```

Multiple patterns may be specified with the `|` operator. This has the same semantics as with `|` in `match` expressions:

```
let mut vals = vec![2, 3, 1, 2, 2];
while let Some(v @ 1) | Some(v @ 2) = vals.pop() {
    // Prints 2, 2, then 1
    println!("{}", v);
}
```

As is the case in `if let` expressions, the scrutinee cannot be a lazy boolean operator expression.

Iterator loops

Syntax

IteratorLoopExpression :
`for` *Pattern* `in` *Expression*$_{\textit{except struct expression}}$ *BlockExpression*

A `for` expression is a syntactic construct for looping over elements provided by an implementation of `std::iter::IntoIterator` . If the iterator yields a value, that value is matched against the irrefutable pattern, the body of the loop is executed, and then control returns to the head of the `for` loop. If the iterator is empty, the `for` expression completes.

An example of a `for` loop over the contents of an array:

```
let v = &["apples", "cake", "coffee"];

for text in v {
    println!("I like {}.", text);
}
```

An example of a for loop over a series of integers:

```
let mut sum = 0;
for n in 1..11 {
    sum += n;
}
assert_eq!(sum, 55);
```

A `for` loop is equivalent to a `loop` expression containing a `match` expression as follows:

```
'label: for PATTERN in iter_expr {
    /* loop body */
}
```

is equivalent to

```
{
    let result = match IntoIterator::into_iter(iter_expr) {
        mut iter => 'label: loop {
            let mut next;
            match Iterator::next(&mut iter) {
                Option::Some(val) => next = val,
                Option::None => break,
            };
            let PATTERN = next;
            let () = { /* loop body */ };
        },
    };
    result
}
```

`IntoIterator`, `Iterator`, and `Option` are always the standard library items here, not whatever those names resolve to in the current scope. The variable names `next`, `iter`, and `val` are for exposition only, they do not actually have names the user can type.

> **Note**: that the outer `match` is used to ensure that any temporary values in `iter_expr` don't get dropped before the loop is finished. `next` is declared before being assigned because it results in types being inferred correctly more often.

Loop labels

> **Syntax**
>
> *LoopLabel* :
>   LIFETIME_OR_LABEL `:`

A loop expression may optionally have a *label*. The label is written as a lifetime preceding the loop expression, as in `'foo: loop { break 'foo; }`, `'bar: while false {}`, `'humbug: for _ in 0..0 {}`. If a label is present, then labeled `break` and `continue` expressions nested within this loop may exit out of this loop or return control to its head. See break expressions and continue expressions.

Labels follow the hygiene and shadowing rules of local variables. For example, this code will print "outer loop":

```
'a: loop {
    'a: loop {
        break 'a;
    }
    print!("outer loop");
    break 'a;
}
```

`break` expressions

> **Syntax**
>
> *BreakExpression* :
>
> `break` LIFETIME_OR_LABEL? *Expression*?

When `break` is encountered, execution of the associated loop body is immediately terminated, for example:

```
let mut last = 0;
for x in 1..100 {
    if x > 12 {
        break;
    }
    last = x;
}
assert_eq!(last, 12);
```

A `break` expression is normally associated with the innermost `loop`, `for` or `while` loop enclosing the `break` expression, but a label can be used to specify which enclosing loop is affected. Example:

```
'outer: loop {
    while true {
        break 'outer;
    }
}
```

A `break` expression is only permitted in the body of a loop, and has one of the forms `break`, `break 'label` or (see below) `break EXPR` or `break 'label EXPR`.

Labelled block expressions

> **Syntax**
>
> *LabelBlockExpression* :
>   *BlockExpression*

Labelled block expressions are exactly like block expressions, except that they allow using `break` expressions within the block. Unlike loops, `break` expressions within a labelled block expression *must* have a label (i.e. the label is not optional). Similarly, labelled block expressions *must* begin with a label.

```
let result = 'block: {
    do_thing();
    if condition_not_met() {
        break 'block 1;
    }
    do_next_thing();
    if condition_not_met() {
        break 'block 2;
    }
    do_last_thing();
    3
};
```

`continue` expressions

> **Syntax**
>
> *ContinueExpression* :
>   `continue` LIFETIME_OR_LABEL?

When `continue` is encountered, the current iteration of the associated loop body is immediately terminated, returning control to the loop *head*. In the case of a `while` loop, the head is the conditional expression controlling the loop. In the case of a `for` loop, the head is the call-expression controlling the loop.

Like `break`, `continue` is normally associated with the innermost enclosing loop, but `continue 'label` may be used to specify the loop affected. A `continue` expression is only permitted in the body of a loop.

`break` and loop values

When associated with a `loop`, a break expression may be used to return a value from that loop, via one of the forms `break EXPR` or `break 'label EXPR`, where `EXPR` is an expression whose result is returned from the `loop`. For example:

```
let (mut a, mut b) = (1, 1);
let result = loop {
    if b > 10 {
        break b;
    }
    let c = a + b;
    a = b;
    b = c;
};
// first number in Fibonacci sequence over 10:
assert_eq!(result, 13);
```

In the case a `loop` has an associated `break`, it is not considered diverging, and the `loop` must have a type compatible with each `break` expression. `break` without an expression is considered identical to `break` with expression `()`.

Range expressions

Syntax

RangeExpression :
   RangeExpr
  | *RangeFromExpr*
  | *RangeToExpr*
  | *RangeFullExpr*
  | *RangeInclusiveExpr*
  | *RangeToInclusiveExpr*

RangeExpr :
  Expression `..` *Expression*

RangeFromExpr :
  Expression `..`

RangeToExpr :
  `..` *Expression*

RangeFullExpr :
  `..`

RangeInclusiveExpr :
  Expression `..=` *Expression*

RangeToInclusiveExpr :
  `..=` *Expression*

The `..` and `..=` operators will construct an object of one of the `std::ops::Range` (or `core::ops::Range`) variants, according to the following table:

Production	Syntax	Type	Range
RangeExpr	start `..` end	std::ops::Range	start ≤ x < end
RangeFromExpr	start `..`	std::ops::RangeFrom	start ≤ x
RangeToExpr	`..` end	std::ops::RangeTo	x < end
RangeFullExpr	`..`	std::ops::RangeFull	-

Production	Syntax	Type	Range
RangeInclusiveExpr	start `..=` end	std::ops::RangeInclusive	start ≤ x ≤ end
RangeToInclusiveExpr	`..=` end	std::ops::RangeToInclusive	x ≤ end

Examples:

```
1..2;   // std::ops::Range
3..;    // std::ops::RangeFrom
..4;    // std::ops::RangeTo
..;     // std::ops::RangeFull
5..=6;  // std::ops::RangeInclusive
..=7;   // std::ops::RangeToInclusive
```

The following expressions are equivalent.

```
let x = std::ops::Range {start: 0, end: 10};
let y = 0..10;

assert_eq!(x, y);
```

Ranges can be used in `for` loops:

```
for i in 1..11 {
    println!("{}", i);
}
```

`if` and `if let` expressions

`if` expressions

> **Syntax**
>
> *IfExpression* :
>
> `if` *Expression*$_{except\ struct\ expression}$ *BlockExpression*
>
> (`else` (*BlockExpression* | *IfExpression* | *IfLetExpression*))$^?$

An `if` expression is a conditional branch in program control. The syntax of an `if` expression is a condition operand, followed by a consequent block, any number of `else if` conditions and blocks, and an optional trailing `else` block. The condition operands must have the boolean type. If a condition operand evaluates to `true`, the consequent block is executed and any subsequent `else if` or `else` block is skipped. If a condition operand evaluates to `false`, the consequent block is skipped and any subsequent `else if` condition is evaluated. If all `if` and `else if` conditions evaluate to `false` then any `else` block is executed. An if expression evaluates to the same value as the executed block, or `()` if no block is evaluated. An `if` expression must have the same type in all situations.

```
if x == 4 {
    println!("x is four");
} else if x == 3 {
    println!("x is three");
} else {
    println!("x is something else");
}

let y = if 12 * 15 > 150 {
    "Bigger"
} else {
    "Smaller"
};
assert_eq!(y, "Bigger");
```

`if let` expressions

Syntax

IfLetExpression :
   `if` `let` *Pattern* `=` *Scrutinee*except lazy boolean operator expression *BlockExpression*
   (`else` (*BlockExpression* | *IfExpression* | *IfLetExpression*))?

An `if let` expression is semantically similar to an `if` expression but in place of a condition operand it expects the keyword `let` followed by a pattern, an `=` and a scrutinee operand. If the value of the scrutinee matches the pattern, the corresponding block will execute. Otherwise, flow proceeds to the following `else` block if it exists. Like `if` expressions, `if let` expressions have a value determined by the block that is evaluated.

```
let dish = ("Ham", "Eggs");

// this body will be skipped because the pattern is refuted
if let ("Bacon", b) = dish {
    println!("Bacon is served with {}", b);
} else {
    // This block is evaluated instead.
    println!("No bacon will be served");
}

// this body will execute
if let ("Ham", b) = dish {
    println!("Ham is served with {}", b);
}

if let _ = 5 {
    println!("Irrefutable patterns are always true");
}
```

`if` and `if let` expressions can be intermixed:

```
let x = Some(3);
let a = if let Some(1) = x {
    1
} else if x == Some(2) {
    2
} else if let Some(y) = x {
    y
} else {
    -1
};
assert_eq!(a, 3);
```

An `if let` expression is equivalent to a `match` expression as follows:

```
if let PATS = EXPR {
    /* body */
} else {
    /*else */
}
```

is equivalent to

```
match EXPR {
    PATS => { /* body */ },
    _ => { /* else */ },    // () if there is no else
}
```

Multiple patterns may be specified with the `|` operator. This has the same semantics as with `|` in `match` expressions:

```
enum E {
    X(u8),
    Y(u8),
    Z(u8),
}
let v = E::Y(12);
if let E::X(n) | E::Y(n) = v {
    assert_eq!(n, 12);
}
```

The expression cannot be a lazy boolean operator expression. Use of a lazy boolean operator is ambiguous with a planned feature change of the language (the implementation of if-let chains - see eRFC 2947). When lazy boolean operator expression is desired, this can be achieved by using parenthesis as below:

```
// Before...
if let PAT = EXPR && EXPR { .. }

// After...
if let PAT = ( EXPR && EXPR ) { .. }

// Before...
if let PAT = EXPR || EXPR { .. }

// After...
if let PAT = ( EXPR || EXPR ) { .. }
```

`match` expressions

Syntax

MatchExpression :
  `match` *Scrutinee* `{`
    *InnerAttribute**
    MatchArms?
  `}`

Scrutinee :
  *Expression*except struct expression

MatchArms :
  (*MatchArm* `=>` (*ExpressionWithoutBlock* `,` | *ExpressionWithBlock* `,`?))*
  MatchArm `=>` *Expression* `,`?

MatchArm :
  *OuterAttribute** *Pattern* *MatchArmGuard*?

MatchArmGuard :
  `if` *Expression*

A ***`match`* expression** branches on a pattern. The exact form of matching that occurs depends on the pattern. A `match` expression has a *scrutinee expression*, which is the value to compare to the patterns. The scrutinee expression and the patterns must have the same type.

A `match` behaves differently depending on whether or not the scrutinee expression is a place expression or value expression. If the scrutinee expression is a value expression, it is first evaluated into a temporary location, and the resulting value is sequentially compared to the patterns in the arms until a match is found. The first arm with a matching pattern is chosen as the branch target of the `match`, any variables bound by the pattern are assigned to local variables in the arm's block, and control enters the block.

When the scrutinee expression is a place expression, the match does not allocate a temporary location; however, a by-value binding may copy or move from the memory location. When possible, it is preferable to match on place expressions, as the lifetime of these matches inherits the lifetime of the place expression rather than being restricted to the inside of the match.

An example of a `match` expression:

```
let x = 1;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    4 => println!("four"),
    5 => println!("five"),
    _ => println!("something else"),
}
```

Variables bound within the pattern are scoped to the match guard and the arm's expression. The binding mode (move, copy, or reference) depends on the pattern.

Multiple match patterns may be joined with the `|` operator. Each pattern will be tested in left-to-right sequence until a successful match is found.

```
let x = 9;
let message = match x {
    0 | 1  => "not many",
    2 ..= 9 => "a few",
    _      => "lots"
};

assert_eq!(message, "a few");

// Demonstration of pattern match order.
struct S(i32, i32);

match S(1, 2) {
    S(z @ 1, _) | S(_, z @ 2) => assert_eq!(z, 1),
    _ => panic!(),
}
```

> Note: The `2..=9` is a Range Pattern, not a Range Expression. Thus, only those types of ranges supported by range patterns can be used in match arms.

Every binding in each `|` separated pattern must appear in all of the patterns in the arm. Every binding of the same name must have the same type, and have the same binding mode.

Match guards

Match arms can accept *match guards* to further refine the criteria for matching a case. Pattern guards appear after the pattern and consist of a `bool` -typed expression following the `if` keyword.

When the pattern matches successfully, the pattern guard expression is executed. If the expression evaluates to true, the pattern is successfully matched against. Otherwise, the next pattern, including other matches with the `|` operator in the same arm, is tested.

```
let message = match maybe_digit {
    Some(x) if x < 10 => process_digit(x),
    Some(x) => process_other(x),
    None => panic!(),
};
```

> Note: Multiple matches using the `|` operator can cause the pattern guard and the side effects it has to execute multiple times. For example:
>
> ```
> let i : Cell<i32> = Cell::new(0);
> match 1 {
>     1 | _ if { i.set(i.get() + 1); false } => {}
>     _ => {}
> }
> assert_eq!(i.get(), 2);
> ```

A pattern guard may refer to the variables bound within the pattern they follow. Before evaluating the guard, a shared reference is taken to the part of the scrutinee the variable matches on. While evaluating the guard, this shared reference is then used when accessing the variable. Only when the guard evaluates to true is the value moved, or copied, from the scrutinee into the variable. This allows shared borrows to be used inside guards without moving out of the scrutinee in case guard fails to match. Moreover, by holding a shared reference while evaluating the guard, mutation inside guards is also prevented.

Attributes on match arms

Outer attributes are allowed on match arms. The only attributes that have meaning on match arms are `cfg` and the lint check attributes.

Inner attributes are allowed directly after the opening brace of the match expression in the same expression contexts as attributes on block expressions.

`return` expressions

> **Syntax**
>
> *ReturnExpression* :
>
> `return` *Expression*?

Return expressions are denoted with the keyword `return`. Evaluating a `return` expression moves its argument into the designated output location for the current function call, destroys the current function activation frame, and transfers control to the caller frame.

An example of a `return` expression:

```
fn max(a: i32, b: i32) -> i32 {
    if a > b {
        return a;
    }
    return b;
}
```

Await expressions

> **Syntax**
>
> *AwaitExpression* :
>   *Expression* `.` `await`

An `await` expression is a syntactic construct for suspending a computation provided by an implementation of `std::future::IntoFuture` until the given future is ready to produce a value. The syntax for an await expression is an expression with a type that implements the `IntoFuture` trait, called the *future operand*, then the token `.`, and then the `await` keyword. Await expressions are legal only within an async context, like an `async fn` or an `async block`.

More specifically, an await expression has the following effect.

1. Create a future by calling `IntoFuture::into_future` on the future operand.
2. Evaluate the future to a future `tmp`;
3. Pin `tmp` using `Pin::new_unchecked`;
4. This pinned future is then polled by calling the `Future::poll` method and passing it the current task context;
5. If the call to `poll` returns `Poll::Pending`, then the future returns `Poll::Pending`, suspending its state so that, when the surrounding async context is re-polled,execution returns to step 3;
6. Otherwise the call to `poll` must have returned `Poll::Ready`, in which case the value contained in the `Poll::Ready` variant is used as the result of the `await` expression itself.

> **Edition differences**: Await expressions are only available beginning with Rust 2018.

Task context

The task context refers to the `Context` which was supplied to the current async context when the async context itself was polled. Because `await` expressions are only legal in an async context, there must be some task context available.

Approximate desugaring

Effectively, an await expression is roughly equivalent to the following non-normative desugaring:

```
match operand.into_future() {
    mut pinned => loop {
        let mut pin = unsafe { Pin::new_unchecked(&mut pinned) };
        match Pin::future::poll(Pin::borrow(&mut pin), &mut current_context) {
            Poll::Ready(r) => break r,
            Poll::Pending => yield Poll::Pending,
        }
    }
}
```

where the `yield` pseudo-code returns `Poll::Pending` and, when re-invoked, resumes execution from that point. The variable `current_context` refers to the context taken from the async environment.

_ expressions

Syntax

UnderscoreExpression :
  `_`

Underscore expressions, denoted with the symbol `_`, are used to signify a placeholder in a destructuring assignment. They may only appear in the left-hand side of an assignment.

Note that this is distinct from the wildcard pattern.

An example of an `_` expression:

```
let p = (1, 2);
let mut a = 0;
(_, a) = p;
```

Patterns

> **Syntax**
>
> *Pattern* :
>
> | ? *PatternNoTopAlt* (| *PatternNoTopAlt*)*
>
> *PatternNoTopAlt* :
>
> *PatternWithoutRange*
>
> | *RangePattern*
>
> *PatternWithoutRange* :
>
> *LiteralPattern*
>
> | *IdentifierPattern*
>
> | *WildcardPattern*
>
> | *RestPattern*
>
> | *ReferencePattern*
>
> | *StructPattern*
>
> | *TupleStructPattern*
>
> | *TuplePattern*
>
> | *GroupedPattern*
>
> | *SlicePattern*
>
> | *PathPattern*
>
> | *MacroInvocation*

Patterns are used to match values against structures and to, optionally, bind variables to values inside these structures. They are also used in variable declarations and parameters for functions and closures.

The pattern in the following example does four things:

- Tests if `person` has the `car` field filled with something.
- Tests if the person's `age` field is between 13 and 19, and binds its value to the `person_age` variable.
- Binds a reference to the `name` field to the variable `person_name`.
- Ignores the rest of the fields of `person`. The remaining fields can have any value and are not bound to any variables.

```
if let
    Person {
        car: Some(_),
        age: person_age @ 13..=19,
        name: ref person_name,
        ..
    } = person
{
    println!("{} has a car and is {} years old.", person_name, person_age);
}
```

Patterns are used in:

- `let` declarations
- Function and closure parameters
- `match` expressions
- `if let` expressions
- `while let` expressions
- `for` expressions

Destructuring

Patterns can be used to *destructure* structs, enums, and tuples. Destructuring breaks up a value into its component pieces. The syntax used is almost the same as when creating such values. In a pattern whose scrutinee expression has a `struct`, `enum` or `tuple` type, a placeholder (`_`) stands in for a *single* data field, whereas a wildcard `..` stands in for *all* the remaining fields of a particular variant. When destructuring a data structure with named (but not numbered) fields, it is allowed to write `fieldname` as a shorthand for `fieldname: fieldname`.

```
match message {
    Message::Quit => println!("Quit"),
    Message::WriteString(write) => println!("{}", &write),
    Message::Move{ x, y: 0 } => println!("move {} horizontally", x),
    Message::Move{ .. } => println!("other move"),
    Message::ChangeColor { 0: red, 1: green, 2: _ } => {
        println!("color change, red: {}, green: {}", red, green);
    }
};
```

Refutability

A pattern is said to be *refutable* when it has the possibility of not being matched by the value it is being matched against. *Irrefutable* patterns, on the other hand, always match the value they are being matched against. Examples:

```
let (x, y) = (1, 2);               // "(x, y)" is an irrefutable pattern

if let (a, 3) = (1, 2) {           // "(a, 3)" is refutable, and will not match
    panic!("Shouldn't reach here");
} else if let (a, 4) = (3, 4) {    // "(a, 4)" is refutable, and will match
    println!("Matched ({}, 4)", a);
}
```

Literal patterns

> **Syntax**
>
> *LiteralPattern* :
>
> `true` | `false`
>
> | CHAR_LITERAL
>
> | BYTE_LITERAL
>
> | STRING_LITERAL
>
> | RAW_STRING_LITERAL
>
> | BYTE_STRING_LITERAL
>
> | RAW_BYTE_STRING_LITERAL
>
> | C_STRING_LITERAL
>
> | RAW_C_STRING_LITERAL
>
> | `-`? INTEGER_LITERAL
>
> | `-`? FLOAT_LITERAL

Literal patterns match exactly the same value as what is created by the literal. Since negative numbers are not literals, literal patterns also accept an optional minus sign before the literal, which acts like the negation operator.

> ⚠️ Floating-point literals are currently accepted, but due to the complexity of comparing them, they are going to be forbidden on literal patterns in a future version of Rust (see issue #41620).

> ⚠️ C string and raw C string literals are accepted in literal patterns, but `&CStr` doesn't implement structural equality (`#[derive(Eq, PartialEq)]`) and therefore any such `match` on a `&CStr` will be rejected with a type error.

Literal patterns are always refutable.

Examples:

```
for i in -2..5 {
    match i {
        -1 => println!("It's minus one"),
        1 => println!("It's a one"),
        2|4 => println!("It's either a two or a four"),
        _ => println!("Matched none of the arms"),
    }
}
```

Identifier patterns

> **Syntax**
> *IdentifierPattern* :
> `ref`? `mut`? IDENTIFIER (`@` *PatternNoTopAlt*) ?

Identifier patterns bind the value they match to a variable. The identifier must be unique within the pattern. The variable will shadow any variables of the same name in scope. The scope of the new binding depends on the context of where the pattern is used (such as a `let` binding or a `match` arm).

Patterns that consist of only an identifier, possibly with a `mut`, match any value and bind it to that identifier. This is the most commonly used pattern in variable declarations and parameters for functions and closures.

```
let mut variable = 10;
fn sum(x: i32, y: i32) -> i32 {
```

To bind the matched value of a pattern to a variable, use the syntax `variable @ subpattern`. For example, the following binds the value 2 to `e` (not the entire range: the range here is a range subpattern).

```
let x = 2;

match x {
    e @ 1 ..= 5 => println!("got a range element {}", e),
    _ => println!("anything"),
}
```

By default, identifier patterns bind a variable to a copy of or move from the matched value depending on whether the matched value implements `Copy` . This can be changed to bind to a reference by using the `ref` keyword, or to a mutable reference using `ref mut` . For example:

```
match a {
    None => (),
    Some(value) => (),
}

match a {
    None => (),
    Some(ref value) => (),
}
```

In the first match expression, the value is copied (or moved). In the second match, a reference to the same memory location is bound to the variable value. This syntax is needed because in destructuring subpatterns the `&` operator can't be applied to the value's fields. For example, the following is not valid:

```
if let Person { name: &person_name, age: 18..=150 } = value { }
```

To make it valid, write the following:

```
if let Person {name: ref person_name, age: 18..=150 } = value { }
```

Thus, `ref` is not something that is being matched against. Its objective is exclusively to make the matched binding a reference, instead of potentially copying or moving what was matched.

Path patterns take precedence over identifier patterns. It is an error if `ref` or `ref mut` is specified and the identifier shadows a constant.

Identifier patterns are irrefutable if the `@` subpattern is irrefutable or the subpattern is not specified.

Binding modes

To service better ergonomics, patterns operate in different *binding modes* in order to make it easier to bind references to values. When a reference value is matched by a non-reference pattern, it will be automatically treated as a `ref` or `ref mut` binding. Example:

```
let x: &Option<i32> = &Some(3);
if let Some(y) = x {
    // y was converted to `ref y` and its type is &i32
}
```

Non-reference patterns include all patterns except bindings, wildcard patterns (`_`), `const` patterns of reference types, and reference patterns.

If a binding pattern does not explicitly have `ref` , `ref mut` , or `mut` , then it uses the *default binding mode* to determine how the variable is bound. The default binding mode starts in "move" mode which uses move semantics. When matching a pattern, the compiler starts from the outside of the pattern and works inwards. Each time a reference is matched using a non-reference pattern, it will automatically dereference the value and update the default binding mode. References will set the default binding mode to `ref` . Mutable references will set the mode to `ref mut` unless the mode is already `ref` in which case it remains `ref` . If the automatically dereferenced value is still a reference, it is dereferenced and this process repeats.

Move bindings and reference bindings can be mixed together in the same pattern. Doing so will result in partial move of the object bound to and the object cannot be used afterwards. This applies only if the type cannot be copied.

In the example below, `name` is moved out of `person` . Trying to use `person` as a whole or `person.name` would result in an error because of *partial move*.

Example:

```
// `name` is moved from person and `age` referenced
let Person { name, ref age } = person;
```

Wildcard pattern

Syntax

WildcardPattern :

_

The *wildcard pattern* (an underscore symbol) matches any value. It is used to ignore values when they don't matter. Inside other patterns it matches a single data field (as opposed to the `..` which matches the remaining fields). Unlike identifier patterns, it does not copy, move or borrow the value it matches.

Examples:

```
let (a, _) = (10, x);   // the x is always matched by _

// ignore a function/closure param
let real_part = |a: f64, _: f64| { a };

// ignore a field from a struct
let RGBA{r: red, g: green, b: blue, a: _} = color;

// accept any Some, with any value
if let Some(_) = x {}
```

The wildcard pattern is always irrefutable.

Rest patterns

Syntax

RestPattern :
   `..`

The *rest pattern* (the `..` token) acts as a variable-length pattern which matches zero or more elements that haven't been matched already before and after. It may only be used in tuple, tuple struct, and slice patterns, and may only appear once as one of the elements in those patterns. It is also allowed in an identifier pattern for slice patterns only.

The rest pattern is always irrefutable.

Examples:

```rust
match slice {
    [] => println!("slice is empty"),
    [one] => println!("single element {}", one),
    [head, tail @ ..] => println!("head={} tail={:?}", head, tail),
}

match slice {
    // Ignore everything but the last element, which must be "!".
    [.., "!"] => println!("!!!"),

    // `start` is a slice of everything except the last element, which must be
"z".
    [start @ .., "z"] => println!("starts with: {:?}", start),

    // `end` is a slice of everything but the first element, which must be "a".
    ["a", end @ ..] => println!("ends with: {:?}", end),

    // 'whole' is the entire slice and `last` is the final element
    whole @ [.., last] => println!("the last element of {:?} is {}", whole,
last),

    rest => println!("{:?}", rest),
}

if let [.., penultimate, _] = slice {
    println!("next to last is {}", penultimate);
}

// Rest patterns may also be used in tuple and tuple struct patterns.
match tuple {
    (1, .., y, z) => println!("y={} z={}", y, z),
    (.., 5) => println!("tail must be 5"),
    (..) => println!("matches everything else"),
}
```

Range patterns

> **Syntax**
>
> *RangePattern* :
>   *RangeInclusivePattern*
> | *RangeFromPattern*
> | *RangeToInclusivePattern*
> | *ObsoleteRangePattern*
>
> *RangeInclusivePattern* :

RangePatternBound `..=` *RangePatternBound*

RangeFromPattern :
RangePatternBound `..`

RangeToInclusivePattern :
`..=` *RangePatternBound*

ObsoleteRangePattern :
RangePatternBound `...` *RangePatternBound*

RangePatternBound :
CHAR_LITERAL
| BYTE_LITERAL
| `-`? INTEGER_LITERAL
| `-`? FLOAT_LITERAL
| *PathExpression*

Range patterns match scalar values within the range defined by their bounds. They comprise a *sigil* (one of `..` , `..=` , or `...`) and a bound on one or both sides. A bound on the left of the sigil is a *lower bound*. A bound on the right is an *upper bound*.

A range pattern with both a lower and upper bound will match all values between and including both of its bounds. It is written as its lower bound, followed by `..=` , followed by its upper bound. The type of the range pattern is the type unification of its upper and lower bounds.

For example, a pattern `'m'..='p'` will match only the values `'m'` , `'n'` , `'o'` , and `'p'` .

The lower bound cannot be greater than the upper bound. That is, in `a..=b` , a ≤ b must be the case. For example, it is an error to have a range pattern `10..=0` .

A range pattern with only a lower bound will match any value greater than or equal to the lower bound. It is written as its lower bound followed by `..` , and has the same type as its lower bound. For example, `1..` will match 1, 9, or 9001, or 9007199254740991 (if it is of an appropriate size), but not 0, and not negative numbers for signed integers.

A range pattern with only an upper bound matches any value less than or equal to the upper bound. It is written as `..=` followed by its upper bound, and has the same type as its upper bound. For example, `..=10` will match 10, 1, 0, and for signed integer types, all negative values.

Range patterns with only one bound cannot be used as the top-level pattern for subpatterns in slice patterns.

The bounds is written as one of:

- A character, byte, integer, or float literal.
- A `-` followed by an integer or float literal.
- A path

If the bounds is written as a path, after macro resolution, the path must resolve to a constant item of the type `char`, an integer type, or a float type.

The type and value of the bounds is dependent upon how it is written out. If the bounds is a path, the pattern has the type and value of the constant the path resolves to. If it is a literal, it has the type and value of the corresponding literal expression. If is a literal preceded by a `-`, it has the same type as the corresponding literal expression and the value of negating the value of the corresponding literal expression.

Examples:

```
let valid_variable = match c {
    'a'..='z' => true,
    'A'..='Z' => true,
    'α'..='ω' => true,
    _ => false,
};

println!("{}", match ph {
    0..=6 => "acid",
    7 => "neutral",
    8..=14 => "base",
    _ => unreachable!(),
});

match uint {
    0 => "zero!",
    1.. => "positive number!",
};

// using paths to constants:
println!("{}", match altitude {
    TROPOSPHERE_MIN..=TROPOSPHERE_MAX => "troposphere",
    STRATOSPHERE_MIN..=STRATOSPHERE_MAX => "stratosphere",
    MESOSPHERE_MIN..=MESOSPHERE_MAX => "mesosphere",
    _ => "outer space, maybe",
});

if let size @ binary::MEGA..=binary::GIGA = n_items * bytes_per_item {
    println!("It fits and occupies {} bytes", size);
}

// using qualified paths:
println!("{}", match 0xfacade {
    0 ..= <u8 as MaxValue>::MAX => "fits in a u8",
    0 ..= <u16 as MaxValue>::MAX => "fits in a u16",
    0 ..= <u32 as MaxValue>::MAX => "fits in a u32",
    _ => "too big",
});
```

Range patterns for fix-width integer and `char` types are irrefutable when they span the entire set of possible values of a type. For example, `0u8..=255u8` is irrefutable. The range of values for an integer type is the closed range from its minimum to maximum value. The range of values for a `char` type are precisely those ranges containing all Unicode Scalar Values: `'\u{0000}'..='\u{D7FF}'` and `'\u{E000}'..='\u{10FFFF}'`.

Floating point range patterns are deprecated and may be removed in a future Rust release. See issue #41620 for more information.

Edition Differences: Before the 2021 edition, range patterns with both a lower and upper bound may also be written using `...` in place of `..=`, with the same meaning.

Note: Although range patterns use the same syntax as range expressions, there are no exclusive range patterns. That is, neither `x .. y` nor `.. x` are valid range patterns.

Reference patterns

> **Syntax**
> *ReferencePattern* :
>   (`&` | `&&`) `mut` ? *PatternWithoutRange*

Reference patterns dereference the pointers that are being matched and, thus, borrow them.

For example, these two matches on `x: &i32` are equivalent:

```
let int_reference = &3;

let a = match *int_reference { 0 => "zero", _ => "some" };
let b = match int_reference { &0 => "zero", _ => "some" };

assert_eq!(a, b);
```

The grammar production for reference patterns has to match the token `&&` to match a reference to a reference because it is a token by itself, not two `&` tokens.

Adding the `mut` keyword dereferences a mutable reference. The mutability must match the mutability of the reference.

Reference patterns are always irrefutable.

Struct patterns

Syntax

StructPattern :
  PathInExpression `{`
    StructPatternElements ?
  `}`

StructPatternElements :
    StructPatternFields (`,` | `,` *StructPatternEtCetera*)?
  | *StructPatternEtCetera*

StructPatternFields :
  StructPatternField (`,` *StructPatternField*) *

StructPatternField :
  OuterAttribute *
  (
      TUPLE_INDEX `:` *Pattern*
    | IDENTIFIER `:` *Pattern*
    | `ref` ? `mut` ? IDENTIFIER
)

StructPatternEtCetera :
  OuterAttribute *
  `..`

Struct patterns match struct, enum, and union values that match all criteria defined by its subpatterns. They are also used to destructure a struct, enum, or union value.

On a struct pattern, the fields are referenced by name, index (in the case of tuple structs) or ignored by use of `..` :

```
match s {
    Point {x: 10, y: 20} => (),
    Point {y: 10, x: 20} => (),    // order doesn't matter
    Point {x: 10, ..} => (),
    Point {..} => (),
}

match t {
    PointTuple {0: 10, 1: 20} => (),
    PointTuple {1: 10, 0: 20} => (),   // order doesn't matter
    PointTuple {0: 10, ..} => (),
    PointTuple {..} => (),
}

match m {
    Message::Quit => (),
    Message::Move {x: 10, y: 20} => (),
    Message::Move {..} => (),
}
```

If `..` is not used, a struct pattern used to match a struct is required to specify all fields:

```
match struct_value {
    Struct{a: 10, b: 'X', c: false} => (),
    Struct{a: 10, b: 'X', ref c} => (),
    Struct{a: 10, b: 'X', ref mut c} => (),
    Struct{a: 10, b: 'X', c: _} => (),
    Struct{a: _, b: _, c: _} => (),
}
```

A struct pattern used to match a union must specify exactly one field (see Pattern matching on unions).

The `ref` and/or `mut` *IDENTIFIER* syntax matches any value and binds it to a variable with the same name as the given field.

```
let Struct{a: x, b: y, c: z} = struct_value;          // destructure all fields
```

A struct pattern is refutable if the *PathInExpression* resolves to a constructor of an enum with more than one variant, or one of its subpatterns is refutable.

Tuple struct patterns

> **Syntax**
> *TupleStructPattern* :
> *PathInExpression* `(` *TupleStructItems*? `)`
>
> *TupleStructItems* :
> *Pattern* (`,` *Pattern*)* `,` ?

Tuple struct patterns match tuple struct and enum values that match all criteria defined by its subpatterns. They are also used to destructure a tuple struct or enum value.

A tuple struct pattern is refutable if the *PathInExpression* resolves to a constructor of an enum with more than one variant, or one of its subpatterns is refutable.

Tuple patterns

> **Syntax**
> *TuplePattern* :
> `(` *TuplePatternItems*? `)`
>
> *TuplePatternItems* :
> *Pattern* `,`
> | *RestPattern*
> | *Pattern* (`,` *Pattern*)+ `,` ?

Tuple patterns match tuple values that match all criteria defined by its subpatterns. They are also used to destructure a tuple.

The form `(..)` with a single *RestPattern* is a special form that does not require a comma, and matches a tuple of any size.

The tuple pattern is refutable when one of its subpatterns is refutable.

An example of using tuple patterns:

```
let pair = (10, "ten");
let (a, b) = pair;

assert_eq!(a, 10);
assert_eq!(b, "ten");
```

Grouped patterns

> **Syntax**
>
> *GroupedPattern* :
>
> `(` *Pattern* `)`

Enclosing a pattern in parentheses can be used to explicitly control the precedence of compound patterns. For example, a reference pattern next to a range pattern such as `&0..=5` is ambiguous and is not allowed, but can be expressed with parentheses.

```
let int_reference = &3;
match int_reference {
    &(0..=5) => (),
    _ => (),
}
```

Slice patterns

> **Syntax**
>
> *SlicePattern* :
>
> `[` *SlicePatternItems*? `]`
>
> *SlicePatternItems* :
>
> *Pattern* (`,` *Pattern*)* `,`?

Slice patterns can match both arrays of fixed size and slices of dynamic size.

```
// Fixed size
let arr = [1, 2, 3];
match arr {
    [1, _, _] => "starts with one",
    [a, b, c] => "starts with something else",
};

// Dynamic size
let v = vec![1, 2, 3];
match v[..] {
    [a, b] => { /* this arm will not apply because the length doesn't match */
}
    [a, b, c] => { /* this arm will apply */ }
    _ => { /* this wildcard is required, since the length is not known
statically */ }
};
```

Slice patterns are irrefutable when matching an array as long as each element is irrefutable. When matching a slice, it is irrefutable only in the form with a single `..` rest pattern or identifier pattern with the `..` rest pattern as a subpattern.

Within a slice, a range pattern without both lower and upper bound must be enclosed in parentheses, as in `(a..)`, to clarify it is intended to match against a single slice element. A range pattern with both lower and upper bound, like `a..=b`, is not required to be enclosed in parentheses.

Path patterns

Syntax

PathPattern :
    PathExpression

Path patterns are patterns that refer either to constant values or to structs or enum variants that have no fields.

Unqualified path patterns can refer to:

- enum variants
- structs
- constants

- associated constants

Qualified path patterns can only refer to associated constants.

Constants cannot be a union type. Struct and enum constants must have `#[derive(PartialEq, Eq)]` (not merely implemented).

Path patterns are irrefutable when they refer to structs or an enum variant when the enum has only one variant or a constant whose type is irrefutable. They are refutable when they refer to refutable constants or enum variants for enums with multiple variants.

Or-patterns

Or-patterns are patterns that match on one of two or more sub-patterns (for example `A | B | C`). They can nest arbitrarily. Syntactically, or-patterns are allowed in any of the places where other patterns are allowed (represented by the *Pattern* production), with the exceptions of `let`-bindings and function and closure arguments (represented by the *PatternNoTopAlt* production).

Static semantics

1. Given a pattern `p | q` at some depth for some arbitrary patterns `p` and `q`, the pattern is considered ill-formed if:

    - the type inferred for `p` does not unify with the type inferred for `q`, or
    - the same set of bindings are not introduced in `p` and `q`, or
    - the type of any two bindings with the same name in `p` and `q` do not unify with respect to types or binding modes.

    Unification of types is in all instances aforementioned exact and implicit type coercions do not apply.

2. When type checking an expression `match e_s { a_1 => e_1, ... a_n => e_n }`, for each match arm `a_i` which contains a pattern of form `p_i | q_i`, the pattern `p_i | q_i` is considered ill formed if, at the depth `d` where it exists the fragment of `e_s` at depth `d`, the type of the expression fragment does not unify with `p_i | q_i`.

3. With respect to exhaustiveness checking, a pattern `p | q` is considered to cover `p` as well as `q`. For some constructor `c(x, ..)` the distributive law applies such that `c(p`

`| q, ..rest)` covers the same set of value as `c(p, ..rest) | c(q, ..rest)` does. This can be applied recursively until there are no more nested patterns of form `p | q` other than those that exist at the top level.

Note that by *"constructor"* we do not refer to tuple struct patterns, but rather we refer to a pattern for any product type. This includes enum variants, tuple structs, structs with named fields, arrays, tuples, and slices.

Dynamic semantics

1. The dynamic semantics of pattern matching a scrutinee expression `e_s` against a pattern `c(p | q, ..rest)` at depth `d` where `c` is some constructor, `p` and `q` are arbitrary patterns, and `rest` is optionally any remaining potential factors in `c`, is defined as being the same as that of `c(p, ..rest) | c(q, ..rest)`.

Precedence with other undelimited patterns

As shown elsewhere in this chapter, there are several types of patterns that are syntactically undelimited, including identifier patterns, reference patterns, and or-patterns. Or-patterns always have the lowest-precedence. This allows us to reserve syntactic space for a possible future type ascription feature and also to reduce ambiguity. For example, `x @ A(..) | B(..)` will result in an error that `x` is not bound in all patterns. `&A(x) | B(x)` will result in a type mismatch between `x` in the different subpatterns.

Type system

Types

Every variable, item, and value in a Rust program has a type. The *type* of a *value* defines the interpretation of the memory holding it and the operations that may be performed on the value.

Built-in types are tightly integrated into the language, in nontrivial ways that are not possible to emulate in user-defined types. User-defined types have limited capabilities.

The list of types is:

- Primitive types:
  - Boolean — `bool`
  - Numeric — integer and float
  - Textual — `char` and `str`
  - Never — `!` — a type with no values
- Sequence types:
  - Tuple
  - Array
  - Slice
- User-defined types:
  - Struct
  - Enum
  - Union
- Function types:
  - Functions
  - Closures
- Pointer types:
  - References
  - Raw pointers
  - Function pointers
- Trait types:
  - Trait objects
  - Impl trait

Type expressions

Syntax

Type :
  TypeNoBounds
  | *ImplTraitType*
  | *TraitObjectType*

TypeNoBounds :
  ParenthesizedType
  | *ImplTraitTypeOneBound*
  | *TraitObjectTypeOneBound*
  | *TypePath*
  | *TupleType*
  | *NeverType*
  | *RawPointerType*
  | *ReferenceType*
  | *ArrayType*
  | *SliceType*
  | *InferredType*
  | *QualifiedPathInType*
  | *BareFunctionType*
  | *MacroInvocation*

A *type expression* as defined in the *Type* grammar rule above is the syntax for referring to a type. It may refer to:

- Sequence types (tuple, array, slice).
- Type paths which can reference:
  - Primitive types (boolean, numeric, textual).
  - Paths to an item (struct, enum, union, type alias, trait).
  - `Self` path where `Self` is the implementing type.
  - Generic type parameters.
- Pointer types (reference, raw pointer, function pointer).
- The inferred type which asks the compiler to determine the type.
- Parentheses which are used for disambiguation.
- Trait types: Trait objects and impl trait.
- The never type.
- Macros which expand to a type expression.

Parenthesized types

ParenthesizedType :
`(` *Type* `)`

In some situations the combination of types may be ambiguous. Use parentheses around a type to avoid ambiguity. For example, the `+` operator for type boundaries within a reference type is unclear where the boundary applies, so the use of parentheses is required. Grammar rules that require this disambiguation use the *TypeNoBounds* rule instead of *Type*.

```
type T<'a> = &'a (dyn Any + Send);
```

Recursive types

Nominal types — structs, enumerations, and unions — may be recursive. That is, each `enum` variant or `struct` or `union` field may refer, directly or indirectly, to the enclosing `enum` or `struct` type itself. Such recursion has restrictions:

- Recursive types must include a nominal type in the recursion (not mere type aliases, or other structural types such as arrays or tuples). So `type Rec = &'static [Rec]` is not allowed.
- The size of a recursive type must be finite; in other words the recursive fields of the type must be pointer types.

An example of a *recursive* type and its use:

```
enum List<T> {
    Nil,
    Cons(T, Box<List<T>>)
}

let a: List<i32> = List::Cons(7, Box::new(List::Cons(13,
Box::new(List::Nil))));
```

Boolean type

```
let b: bool = true;
```

The *boolean type* or *bool* is a primitive data type that can take on one of two values, called *true* and *false*.

Values of this type may be created using a literal expression using the keywords `true` and `false` corresponding to the value of the same name.

This type is a part of the language prelude with the name `bool`.

An object with the boolean type has a size and alignment of 1 each. The value false has the bit pattern `0x00` and the value true has the bit pattern `0x01`. It is undefined behavior for an object with the boolean type to have any other bit pattern.

The boolean type is the type of many operands in various expressions:

- The condition operand in if expressions and while expressions
- The operands in lazy boolean operator expressions

Note: The boolean type acts similarly to but is not an enumerated type. In practice, this mostly means that constructors are not associated to the type (e.g. `bool::true`).

Like all primitives, the boolean type implements the traits `Clone`, `Copy`, `Sized`, `Send`, and `Sync`.

Note: See the standard library docs for library operations.

Operations on boolean values

When using certain operator expressions with a

boolean type for its operands, they evaluate using the rules of boolean logic.

Logical not

b	!b
true	false
false	true

Logical or

a	b	a | b
true	true	true
true	false	true
false	true	true
false	false	false

Logical and

a	b	a & b
true	true	true
true	false	false
false	true	false
false	false	false

Logical xor

a	b	a ^ b
true	true	false
true	false	true
false	true	true
false	false	false

Comparisons

a	b	a == b
true	true	true

a	b	`a == b`
`true`	`false`	`false`
`false`	`true`	`false`
`false`	`false`	`true`

a	b	`a > b`
`true`	`true`	`false`
`true`	`false`	`true`
`false`	`true`	`false`
`false`	`false`	`false`

- `a != b` is the same as `!(a == b)`
- `a >= b` is the same as `a == b | a > b`
- `a < b` is the same as `!(a >= b)`
- `a <= b` is the same as `a == b | a < b`

Bit validity

The single byte of a `bool` is guaranteed to be initialized (in other words, `transmute::<bool, u8>(...)` is always sound -- but since some bit patterns are invalid `bool` s, the inverse is not always sound).

Numeric types

Integer types

The unsigned integer types consist of:

Type	Minimum	Maximum
`u8`	0	2^8-1
`u16`	0	2^{16}-1
`u32`	0	2^{32}-1
`u64`	0	2^{64}-1
`u128`	0	2^{128}-1

The signed two's complement integer types consist of:

Type	Minimum	Maximum
`i8`	-(2^7)	2^7-1
`i16`	-(2^{15})	2^{15}-1
`i32`	-(2^{31})	2^{31}-1
`i64`	-(2^{63})	2^{63}-1
`i128`	-(2^{127})	2^{127}-1

Floating-point types

The IEEE 754-2008 "binary32" and "binary64" floating-point types are `f32` and `f64`, respectively.

Machine-dependent integer types

The `usize` type is an unsigned integer type with the same number of bits as the platform's pointer type. It can represent every memory address in the process.

The `isize` type is a signed integer type with the same number of bits as the platform's pointer type. The theoretical upper bound on object and array size is the maximum `isize` value. This ensures that `isize` can be used to calculate differences between pointers into an object or array and can address every byte within an object along with one byte past the end.

`usize` and `isize` are at least 16-bits wide.

> **Note**: Many pieces of Rust code may assume that pointers, `usize`, and `isize` are either 32-bit or 64-bit. As a consequence, 16-bit pointer support is limited and may require explicit care and acknowledgment from a library to support.

Bit validity

For every numeric type, `T`, the bit validity of `T` is equivalent to the bit validity of `[u8; size_of::<T>()]`. An uninitialized byte is not a valid `u8`.

Textual types

The types `char` and `str` hold textual data.

A value of type `char` is a Unicode scalar value (i.e. a code point that is not a surrogate), represented as a 32-bit unsigned word in the 0x0000 to 0xD7FF or 0xE000 to 0x10FFFF range. It is immediate Undefined Behavior to create a `char` that falls outside this range. A `[char]` is effectively a UCS-4 / UTF-32 string of length 1.

A value of type `str` is represented the same way as `[u8]`, a slice of 8-bit unsigned bytes. However, the Rust standard library makes extra assumptions about `str`: methods working on `str` assume and ensure that the data in there is valid UTF-8. Calling a `str` method with a non-UTF-8 buffer can cause Undefined Behavior now or in the future.

Since `str` is a dynamically sized type, it can only be instantiated through a pointer type, such as `&str`.

Layout and bit validity

`char` is guaranteed to have the same size and alignment as `u32` on all platforms.

Every byte of a `char` is guaranteed to be initialized (in other words, `transmute::<char, [u8; size_of::<char>()]>(...)` is always sound -- but since some bit patterns are invalid `char`s, the inverse is not always sound).

Never type

Syntax

NeverType : `!`

The never type `!` is a type with no values, representing the result of computations that never complete. Expressions of type `!` can be coerced into any other type.

The `!` type can **only** appear in function return types presently, indicating it is a diverging function that never returns.

```
fn foo() -> ! {
    panic!("This call never returns.");
}
```

```
extern "C" {
    pub fn no_return_extern_func() -> !;
}
```

Tuple types

Syntax

TupleType :
`( )`
| `(` (*Type* `,`)+ *Type*? `)`

Tuple types are a family of structural types[1] for heterogeneous lists of other types.

The syntax for a tuple type is a parenthesized, comma-separated list of types. 1-ary tuples require a comma after their element type to be disambiguated with a parenthesized type.

A tuple type has a number of fields equal to the length of the list of types. This number of fields determines the *arity* of the tuple. A tuple with `n` fields is called an *n-ary tuple*. For example, a tuple with 2 fields is a 2-ary tuple.

Fields of tuples are named using increasing numeric names matching their position in the list of types. The first field is `0`. The second field is `1`. And so on. The type of each field is the type of the same position in the tuple's list of types.

For convenience and historical reasons, the tuple type with no fields (`()`) is often called *unit* or *the unit type*. Its one value is also called *unit* or *the unit value*.

Some examples of tuple types:

- `()` (unit)
- `(f64, f64)`
- `(String, i32)`
- `(i32, String)` (different type from the previous example)
- `(i32, f64, Vec<String>, Option<bool>)`

Values of this type are constructed using a tuple expression. Furthermore, various expressions will produce the unit value if there is no other meaningful value for it to evaluate to. Tuple fields can be accessed by either a tuple index expression or pattern matching.

[1] Structural types are always equivalent if their internal types are equivalent. For a nominal version of tuples, see tuple structs.

Array types

> **Syntax**
>
> *ArrayType* :
>
> `[` *Type* `;` *Expression* `]`

An array is a fixed-size sequence of `N` elements of type `T`. The array type is written as `[T; N]`. The size is a constant expression that evaluates to a `usize`.

Examples:

```
// A stack-allocated array
let array: [i32; 3] = [1, 2, 3];

// A heap-allocated array, coerced to a slice
let boxed_array: Box<[i32]> = Box::new([1, 2, 3]);
```

All elements of arrays are always initialized, and access to an array is always bounds-checked in safe methods and operators.

> Note: The `Vec<T>` standard library type provides a heap-allocated resizable array type.

Slice types

> **Syntax**
> *SliceType* :
>   `[` *Type* `]`

A slice is a dynamically sized type representing a 'view' into a sequence of elements of type `T`. The slice type is written as `[T]`.

Slice types are generally used through pointer types. For example:

- `&[T]` : a 'shared slice', often just called a 'slice'. It doesn't own the data it points to; it borrows it.
- `&mut [T]` : a 'mutable slice'. It mutably borrows the data it points to.
- `Box<[T]>` : a 'boxed slice'

Examples:

```
// A heap-allocated array, coerced to a slice
let boxed_array: Box<[i32]> = Box::new([1, 2, 3]);

// A (shared) slice into an array
let slice: &[i32] = &boxed_array[..];
```

All elements of slices are always initialized, and access to a slice is always bounds-checked in safe methods and operators.

Struct types

A `struct` *type* is a heterogeneous product of other types, called the *fields* of the type.[1]

New instances of a `struct` can be constructed with a struct expression.

The memory layout of a `struct` is undefined by default to allow for compiler optimizations like field reordering, but it can be fixed with the `repr` attribute. In either case, fields may be given in any order in a corresponding struct *expression*; the resulting `struct` value will always have the same memory layout.

The fields of a `struct` may be qualified by visibility modifiers, to allow access to data in a struct outside a module.

A *tuple struct* type is just like a struct type, except that the fields are anonymous.

A *unit-like struct* type is like a struct type, except that it has no fields. The one value constructed by the associated struct expression is the only value that inhabits such a type.

[1] `struct` types are analogous to `struct` types in C, the *record* types of the ML family, or the *struct* types of the Lisp family.

Enumerated types

An *enumerated type* is a nominal, heterogeneous disjoint union type, denoted by the name of an `enum` item. [1]

An `enum` item declares both the type and a number of *variants*, each of which is independently named and has the syntax of a struct, tuple struct or unit-like struct.

New instances of an `enum` can be constructed with a struct expression.

Any `enum` value consumes as much memory as the largest variant for its corresponding `enum` type, as well as the size needed to store a discriminant.

Enum types cannot be denoted *structurally* as types, but must be denoted by named reference to an `enum` item.

[1] The `enum` type is analogous to a `data` constructor declaration in ML, or a *pick ADT* in Limbo.

Union types

A *union type* is a nominal, heterogeneous C-like union, denoted by the name of a `union` item.

Unions have no notion of an "active field". Instead, every union access transmutes parts of the content of the union to the type of the accessed field. Since transmutes can cause unexpected or undefined behaviour, `unsafe` is required to read from a union field. Union field types are also restricted to a subset of types which ensures that they never need dropping. See the item documentation for further details.

The memory layout of a `union` is undefined by default (in particular, fields do *not* have to be at offset 0), but the `#[repr(...)]` attribute can be used to fix a layout.

Function item types

When referred to, a function item, or the constructor of a tuple-like struct or enum variant, yields a zero-sized value of its *function item type*. That type explicitly identifies the function - its name, its type arguments, and its early-bound lifetime arguments (but not its late-bound lifetime arguments, which are only assigned when the function is called) - so the value does not need to contain an actual function pointer, and no indirection is needed when the function is called.

There is no syntax that directly refers to a function item type, but the compiler will display the type as something like `fn(u32) -> i32 {fn_name}` in error messages.

Because the function item type explicitly identifies the function, the item types of different functions - different items, or the same item with different generics - are distinct, and mixing them will create a type error:

```
fn foo<T>() { }
let x = &mut foo::<i32>;
*x = foo::<u32>; //~ ERROR mismatched types
```

However, there is a coercion from function items to function pointers with the same signature, which is triggered not only when a function item is used when a function pointer is directly expected, but also when different function item types with the same signature meet in different arms of the same `if` or `match`:

```
// `foo_ptr_1` has function pointer type `fn()` here
let foo_ptr_1: fn() = foo::<i32>;

// ... and so does `foo_ptr_2` - this type-checks.
let foo_ptr_2 = if want_i32 {
    foo::<i32>
} else {
    foo::<u32>
};
```

All function items implement `Fn`, `FnMut`, `FnOnce`, `Copy`, `Clone`, `Send`, and `Sync`.

Closure types

A closure expression produces a closure value with a unique, anonymous type that cannot be written out. A closure type is approximately equivalent to a struct which contains the captured variables. For instance, the following closure:

```
fn f<F : FnOnce() -> String> (g: F) {
    println!("{}", g());
}

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

f(|| {
    s += &t;
    s
});
// Prints "foobar".
```

generates a closure type roughly like the following:

```
struct Closure<'a> {
    s : String,
    t : &'a String,
}

impl<'a> FnOnce<()> for Closure<'a> {
    type Output = String;
    fn call_once(self) -> String {
        self.s += &*self.t;
        self.s
    }
}
```

so that the call to `f` works as if it were:

```
f(Closure{s: s, t: &t});
```

Capture modes

The compiler prefers to capture a closed-over variable by immutable borrow, followed by unique immutable borrow (see below), by mutable borrow, and finally by move. It will pick the first choice of these that is compatible with how the captured variable is used inside the closure body. The compiler does not take surrounding code into account, such as the lifetimes of involved variables, or of the closure itself.

If the `move` keyword is used, then all captures are by move or, for `Copy` types, by copy, regardless of whether a borrow would work. The `move` keyword is usually used to allow the closure to outlive the captured values, such as if the closure is being returned or used to spawn a new thread.

Composite types such as structs, tuples, and enums are always captured entirely, not by individual fields. It may be necessary to borrow into a local variable in order to capture a single field:

```
struct SetVec {
    set: HashSet<u32>,
    vec: Vec<u32>
}

impl SetVec {
    fn populate(&mut self) {
        let vec = &mut self.vec;
        self.set.iter().for_each(|&n| {
            vec.push(n);
        })
    }
}
```

If, instead, the closure were to use `self.vec` directly, then it would attempt to capture `self` by mutable reference. But since `self.set` is already borrowed to iterate over, the code would not compile.

Unique immutable borrows in captures

Captures can occur by a special kind of borrow called a *unique immutable borrow*, which cannot be used anywhere else in the language and cannot be written out explicitly. It occurs when modifying the referent of a mutable reference, as in the following example:

```
let mut b = false;
let x = &mut b;
{
    let mut c = || { *x = true; };
    // The following line is an error:
    // let y = &x;
    c();
}
let z = &x;
```

In this case, borrowing `x` mutably is not possible, because `x` is not `mut`. But at the same time, borrowing `x` immutably would make the assignment illegal, because a `& &mut` reference might not be unique, so it cannot safely be used to modify a value. So a unique immutable borrow is used: it borrows `x` immutably, but like a mutable borrow, it must be unique. In the above example, uncommenting the declaration of `y` will produce an error because it would violate the uniqueness of the closure's borrow of `x`; the declaration of z is valid because the closure's lifetime has expired at the end of the block, releasing the borrow.

Call traits and coercions

Closure types all implement `FnOnce`, indicating that they can be called once by consuming ownership of the closure. Additionally, some closures implement more specific call traits:

- A closure which does not move out of any captured variables implements `FnMut`, indicating that it can be called by mutable reference.
- A closure which does not mutate or move out of any captured variables implements `Fn`, indicating that it can be called by shared reference.

> Note: `move` closures may still implement `Fn` or `FnMut`, even though they capture variables by move. This is because the traits implemented by a closure type are determined by what the closure does with captured values, not how it captures them.

Non-capturing closures are closures that don't capture anything from their environment. They can be coerced to function pointers (e.g., `fn()`) with the matching signature.

```
let add = |x, y| x + y;

let mut x = add(5,7);

type Binop = fn(i32, i32) -> i32;
let bo: Binop = add;
x = bo(5,7);
```

Other traits

All closure types implement `Sized`. Additionally, closure types implement the following traits if allowed to do so by the types of the captures it stores:

- `Clone`
- `Copy`
- `Sync`
- `Send`

The rules for `Send` and `Sync` match those for normal struct types, while `Clone` and `Copy` behave as if derived. For `Clone`, the order of cloning of the captured variables is left unspecified.

Because captures are often by reference, the following general rules arise:

- A closure is `Sync` if all captured variables are `Sync`.
- A closure is `Send` if all variables captured by non-unique immutable reference are `Sync`, and all values captured by unique immutable or mutable reference, copy, or move are `Send`.
- A closure is `Clone` or `Copy` if it does not capture any values by unique immutable or mutable reference, and if all values it captures by copy or move are `Clone` or `Copy`, respectively.

Pointer types

All pointers are explicit first-class values. They can be moved or copied, stored into data structs, and returned from functions.

References (`&` and `&mut`)

Syntax

ReferenceType :
  `&` *Lifetime*? `mut`? *TypeNoBounds*

Shared references (`&`)

Shared references point to memory which is owned by some other value. When a shared reference to a value is created, it prevents direct mutation of the value. Interior mutability provides an exception for this in certain circumstances. As the name suggests, any number of shared references to a value may exist. A shared reference type is written `&type`, or `&'a type` when you need to specify an explicit lifetime. Copying a reference is a "shallow" operation: it involves only copying the pointer itself, that is, pointers are `Copy`. Releasing a reference has no effect on the value it points to, but referencing of a temporary value will keep it alive during the scope of the reference itself.

Mutable references (`&mut`)

Mutable references point to memory which is owned by some other value. A mutable reference type is written `&mut type` or `&'a mut type`. A mutable reference (that hasn't been borrowed) is the only way to access the value it points to, so is not `Copy`.

Raw pointers (`*const` and `*mut`)

Syntax

RawPointerType :
`*` (`mut` | `const`) *TypeNoBounds*

Raw pointers are pointers without safety or liveness guarantees. Raw pointers are written as `*const T` or `*mut T`. For example `*const i32` means a raw pointer to a 32-bit integer. Copying or dropping a raw pointer has no effect on the lifecycle of any other value. Dereferencing a raw pointer is an `unsafe` operation. This can also be used to convert a raw pointer to a reference by reborrowing it (`&*` or `&mut *`). Raw pointers are generally discouraged; they exist to support interoperability with foreign code, and writing performance-critical or low-level functions.

When comparing raw pointers they are compared by their address, rather than by what they point to. When comparing raw pointers to dynamically sized types they also have their additional data compared.

Raw pointers can be created directly using `core::ptr::addr_of!` for `*const` pointers and `core::ptr::addr_of_mut!` for `*mut` pointers.

Smart Pointers

The standard library contains additional 'smart pointer' types beyond references and raw pointers.

Bit validity

Despite pointers and references being similar to `usize` s in the machine code emitted on most platforms, the semantics of transmuting a reference or pointer type to a non-pointer type is currently undecided. Thus, it may not be valid to transmute a pointer or reference type, `P`, to a `[u8; size_of::<P>()]`.

For thin raw pointers (i.e., for `P = *const T` or `P = *mut T` for `T: Sized`), the inverse direction (transmuting from an integer or array of integers to `P`) is always valid. However, the pointer produced via such a transmutation may not be dereferenced (not even if `T` has size zero).

Function pointer types

Syntax

BareFunctionType :
  ForLifetimes? *FunctionTypeQualifiers* `fn`
    `(` *FunctionParametersMaybeNamedVariadic*? `)` *BareFunctionReturnType*?

FunctionTypeQualifiers:
  `unsafe`? (`extern` *Abi*?)?

BareFunctionReturnType:
  `->` *TypeNoBounds*

FunctionParametersMaybeNamedVariadic :
  MaybeNamedFunctionParameters | *MaybeNamedFunctionParametersVariadic*

MaybeNamedFunctionParameters :
  MaybeNamedParam (`,` *MaybeNamedParam*)* `,`?

MaybeNamedParam :
  *OuterAttribute** ((IDENTIFIER | `_`) `:`)? *Type*

MaybeNamedFunctionParametersVariadic :
  (*MaybeNamedParam* `,`)* *MaybeNamedParam* `,` *OuterAttribute** `...`

Function pointer types, written using the `fn` keyword, refer to a function whose identity is not necessarily known at compile-time. They can be created via a coercion from both function items and non-capturing closures.

The `unsafe` qualifier indicates that the type's value is an unsafe function, and the `extern` qualifier indicates it is an extern function.

Variadic parameters can only be specified with `extern` function types with the `"C"` or `"cdecl"` calling convention.

An example where `Binop` is defined as a function pointer type:

```rust
fn add(x: i32, y: i32) -> i32 {
    x + y
}

let mut x = add(5,7);

type Binop = fn(i32, i32) -> i32;
let bo: Binop = add;
x = bo(5,7);
```

Attributes on function pointer parameters

Attributes on function pointer parameters follow the same rules and restrictions as regular function parameters.

Trait objects

> **Syntax**
> *TraitObjectType* :
>   `dyn` ? *TypeParamBounds*
>
> *TraitObjectTypeOneBound* :
>   `dyn` ? *TraitBound*

A *trait object* is an opaque value of another type that implements a set of traits. The set of traits is made up of an object safe *base trait* plus any number of auto traits.

Trait objects implement the base trait, its auto traits, and any supertraits of the base trait.

Trait objects are written as the keyword `dyn` followed by a set of trait bounds, but with the following restrictions on the trait bounds. All traits except the first trait must be auto traits, there may not be more than one lifetime, and opt-out bounds (e.g. `?Sized`) are not allowed. Furthermore, paths to traits may be parenthesized.

For example, given a trait `Trait`, the following are all trait objects:

- `dyn Trait`
- `dyn Trait + Send`
- `dyn Trait + Send + Sync`
- `dyn Trait + 'static`
- `dyn Trait + Send + 'static`
- `dyn Trait +`
- `dyn 'static + Trait`.
- `dyn (Trait)`

> **Edition Differences**: Before the 2021 edition, the `dyn` keyword may be omitted.
>
> Note: For clarity, it is recommended to always use the `dyn` keyword on your trait objects unless your codebase supports compiling with Rust 1.26 or lower.

> **Edition Differences**: In the 2015 edition, if the first bound of the trait object is a path that starts with `::`, then the `dyn` will be treated as a part of the path. The first path

> can be put in parenthesis to get around this. As such, if you want a trait object with the trait `::your_module::Trait`, you should write it as `dyn (::your_module::Trait)`.
>
> Beginning in the 2018 edition, `dyn` is a true keyword and is not allowed in paths, so the parentheses are not necessary.

Two trait object types alias each other if the base traits alias each other and if the sets of auto traits are the same and the lifetime bounds are the same. For example, `dyn Trait + Send + UnwindSafe` is the same as `dyn Trait + UnwindSafe + Send`.

Due to the opaqueness of which concrete type the value is of, trait objects are dynamically sized types. Like all DSTs, trait objects are used behind some type of pointer; for example `&dyn SomeTrait` or `Box<dyn SomeTrait>`. Each instance of a pointer to a trait object includes:

- a pointer to an instance of a type `T` that implements `SomeTrait`
- a *virtual method table*, often just called a *vtable*, which contains, for each method of `SomeTrait` and its supertraits that `T` implements, a pointer to `T`'s implementation (i.e. a function pointer).

The purpose of trait objects is to permit "late binding" of methods. Calling a method on a trait object results in virtual dispatch at runtime: that is, a function pointer is loaded from the trait object vtable and invoked indirectly. The actual implementation for each vtable entry can vary on an object-by-object basis.

An example of a trait object:

```
trait Printable {
    fn stringify(&self) -> String;
}

impl Printable for i32 {
    fn stringify(&self) -> String { self.to_string() }
}

fn print(a: Box<dyn Printable>) {
    println!("{}", a.stringify());
}

fn main() {
    print(Box::new(10) as Box<dyn Printable>);
}
```

In this example, the trait `Printable` occurs as a trait object in both the type signature of `print`, and the cast expression in `main`.

Trait Object Lifetime Bounds

Since a trait object can contain references, the lifetimes of those references need to be expressed as part of the trait object. This lifetime is written as `Trait + 'a`. There are defaults that allow this lifetime to usually be inferred with a sensible choice.

Impl trait

> **Syntax**
>
> *ImplTraitType* : `impl` *TypeParamBounds*
>
> *ImplTraitTypeOneBound* : `impl` *TraitBound*

`impl Trait` provides ways to specify unnamed but concrete types that implement a specific trait. It can appear in two sorts of places: argument position (where it can act as an anonymous type parameter to functions), and return position (where it can act as an abstract return type).

```
trait Trait {}

// argument position: anonymous type parameter
fn foo(arg: impl Trait) {
}

// return position: abstract return type
fn bar() -> impl Trait {
}
```

Anonymous type parameters

> Note: This is often called "impl Trait in argument position". (The term "parameter" is more correct here, but "impl Trait in argument position" is the phrasing used during the development of this feature, and it remains in parts of the implementation.)

Functions can use `impl` followed by a set of trait bounds to declare a parameter as having an anonymous type. The caller must provide a type that satisfies the bounds declared by the anonymous type parameter, and the function can only use the methods available through the trait bounds of the anonymous type parameter.

For example, these two forms are almost equivalent:

```
trait Trait {}

// generic type parameter
fn with_generic_type<T: Trait>(arg: T) {
}

// impl Trait in argument position
fn with_impl_trait(arg: impl Trait) {
}
```

That is, `impl Trait` in argument position is syntactic sugar for a generic type parameter like `<T: Trait>`, except that the type is anonymous and doesn't appear in the *GenericParams* list.

> **Note:** For function parameters, generic type parameters and `impl Trait` are not exactly equivalent. With a generic parameter such as `<T: Trait>`, the caller has the option to explicitly specify the generic argument for `T` at the call site using *GenericArgs*, for example, `foo::<usize>(1)`. If `impl Trait` is the type of *any* function parameter, then the caller can't ever provide any generic arguments when calling that function. This includes generic arguments for the return type or any const generics.
>
> Therefore, changing the function signature from either one to the other can constitute a breaking change for the callers of a function.

Abstract return types

> Note: This is often called "impl Trait in return position".

Functions can use `impl Trait` to return an abstract return type. These types stand in for another concrete type where the caller may only use the methods declared by the specified `Trait`. Each possible return value from the function must resolve to the same concrete type.

`impl Trait` in return position allows a function to return an unboxed abstract type. This is particularly useful with closures and iterators. For example, closures have a unique, un-writable type. Previously, the only way to return a closure from a function was to use a trait object:

```
fn returns_closure() -> Box<dyn Fn(i32) -> i32> {
    Box::new(|x| x + 1)
}
```

This could incur performance penalties from heap allocation and dynamic dispatch. It wasn't possible to fully specify the type of the closure, only to use the `Fn` trait. That means that the trait object is necessary. However, with `impl Trait`, it is possible to write this more simply:

```
fn returns_closure() -> impl Fn(i32) -> i32 {
    |x| x + 1
}
```

which also avoids the drawbacks of using a boxed trait object.

Similarly, the concrete types of iterators could become very complex, incorporating the types of all previous iterators in a chain. Returning `impl Iterator` means that a function only exposes the `Iterator` trait as a bound on its return type, instead of explicitly specifying all of the other iterator types involved.

Return-position `impl Trait` in traits and trait implementations

Functions in traits may also use `impl Trait` as a syntax for an anonymous associated type.

Every `impl Trait` in the return type of an associated function in a trait is desugared to an anonymous associated type. The return type that appears in the implementation's function signature is used to determine the value of the associated type.

Differences between generics and `impl Trait` in return position

In argument position, `impl Trait` is very similar in semantics to a generic type parameter. However, there are significant differences between the two in return position. With `impl Trait`, unlike with a generic type parameter, the function chooses the return type, and the caller cannot choose the return type.

The function:

```
fn foo<T: Trait>() -> T {
    // ...
}
```

allows the caller to determine the return type, `T`, and the function returns that type.

The function:

```
fn foo() -> impl Trait {
    // ...
}
```

doesn't allow the caller to determine the return type. Instead, the function chooses the return type, but only promises that it will implement `Trait`.

Limitations

`impl Trait` can only appear as a parameter or return type of a non-`extern` function. It cannot be the type of a `let` binding, field type, or appear inside a type alias.

Type parameters

Within the body of an item that has type parameter declarations, the names of its type parameters are types:

```
fn to_vec<A: Clone>(xs: &[A]) -> Vec<A> {
    if xs.is_empty() {
        return vec![];
    }
    let first: A = xs[0].clone();
    let mut rest: Vec<A> = to_vec(&xs[1..]);
    rest.insert(0, first);
    rest
}
```

Here, `first` has type `A`, referring to `to_vec`'s `A` type parameter; and `rest` has type `Vec<A>`, a vector with element type `A`.

Inferred type

Syntax
InferredType : `_`

The inferred type asks the compiler to infer the type if possible based on the surrounding information available. It cannot be used in item signatures. It is often used in generic arguments:

```
let x: Vec<_> = (0..10).collect();
```

Dynamically Sized Types

Most types have a fixed size that is known at compile time and implement the trait `Sized`. A type with a size that is known only at run-time is called a *dynamically sized type* (*DST*) or, informally, an unsized type. Slices and trait objects are two examples of DSTs. Such types can only be used in certain cases:

- Pointer types to DSTs are sized but have twice the size of pointers to sized types
  - Pointers to slices also store the number of elements of the slice.
  - Pointers to trait objects also store a pointer to a vtable.
- DSTs can be provided as type arguments to generic type parameters having the special `?Sized` bound. They can also be used for associated type definitions when the corresponding associated type declaration has a `?Sized` bound. By default, any type parameter or associated type has a `Sized` bound, unless it is relaxed using `?Sized`.
- Traits may be implemented for DSTs. Unlike with generic type parameters, `Self: ?Sized` is the default in trait definitions.
- Structs may contain a DST as the last field; this makes the struct itself a DST.

Note: variables, function parameters, const items, and static items must be `Sized`.

Type Layout

The layout of a type is its size, alignment, and the relative offsets of its fields. For enums, how the discriminant is laid out and interpreted is also part of type layout.

Type layout can be changed with each compilation. Instead of trying to document exactly what is done, we only document what is guaranteed today.

Size and Alignment

All values have an alignment and size.

The *alignment* of a value specifies what addresses are valid to store the value at. A value of alignment `n` must only be stored at an address that is a multiple of n. For example, a value with an alignment of 2 must be stored at an even address, while a value with an alignment of 1 can be stored at any address. Alignment is measured in bytes, and must be at least 1, and always a power of 2. The alignment of a value can be checked with the `align_of_val` function.

The *size* of a value is the offset in bytes between successive elements in an array with that item type including alignment padding. The size of a value is always a multiple of its alignment. Note that some types are zero-sized; 0 is considered a multiple of any alignment (for example, on some platforms, the type `[u16; 0]` has size 0 and alignment 2). The size of a value can be checked with the `size_of_val` function.

Types where all values have the same size and alignment, and both are known at compile time, implement the `Sized` trait and can be checked with the `size_of` and `align_of` functions. Types that are not `Sized` are known as dynamically sized types. Since all values of a `Sized` type share the same size and alignment, we refer to those shared values as the size of the type and the alignment of the type respectively.

Primitive Data Layout

The size of most primitives is given in this table.

Type	`size_of::<Type>()`
`bool`	1

Type	`size_of::<Type>()`
`u8` / `i8`	1
`u16` / `i16`	2
`u32` / `i32`	4
`u64` / `i64`	8
`u128` / `i128`	16
`f32`	4
`f64`	8
`char`	4

`usize` and `isize` have a size big enough to contain every address on the target platform. For example, on a 32 bit target, this is 4 bytes and on a 64 bit target, this is 8 bytes.

Most primitives are generally aligned to their size, although this is platform-specific behavior. In particular, on x86 u64 and f64 are only aligned to 32 bits.

Pointers and References Layout

Pointers and references have the same layout. Mutability of the pointer or reference does not change the layout.

Pointers to sized types have the same size and alignment as `usize`.

Pointers to unsized types are sized. The size and alignment is guaranteed to be at least equal to the size and alignment of a pointer.

> Note: Though you should not rely on this, all pointers to DSTs are currently twice the size of the size of `usize` and have the same alignment.

Array Layout

An array of `[T; N]` has a size of `size_of::<T>() * N` and the same alignment of `T`. Arrays are laid out so that the zero-based `nth` element of the array is offset from the start of the array by `n * size_of::<T>()` bytes.

Slice Layout

Slices have the same layout as the section of the array they slice.

> Note: This is about the raw `[T]` type, not pointers (`&[T]` , `Box<[T]>` , etc.) to slices.

`str` Layout

String slices are a UTF-8 representation of characters that have the same layout as slices of type `[u8]` .

Tuple Layout

Tuples are laid out according to the `Rust` representation.

The exception to this is the unit tuple (`()`), which is guaranteed as a zero-sized type to have a size of 0 and an alignment of 1.

Trait Object Layout

Trait objects have the same layout as the value the trait object is of.

> Note: This is about the raw trait object types, not pointers (`&dyn Trait` , `Box<dyn Trait>` , etc.) to trait objects.

Closure Layout

Closures have no layout guarantees.

Representations

All user-defined composite types (`struct` s, `enum` s, and `union` s) have a *representation* that specifies what the layout is for the type. The possible representations for a type are:

- `Rust` (default)
- `C`
- The primitive representations
- `transparent`

The representation of a type can be changed by applying the `repr` attribute to it. The following example shows a struct with a `C` representation.

```
#[repr(C)]
struct ThreeInts {
    first: i16,
    second: i8,
    third: i32
}
```

The alignment may be raised or lowered with the `align` and `packed` modifiers respectively. They alter the representation specified in the attribute. If no representation is specified, the default one is altered.

```
// Default representation, alignment lowered to 2.
#[repr(packed(2))]
struct PackedStruct {
    first: i16,
    second: i8,
    third: i32
}

// C representation, alignment raised to 8
#[repr(C, align(8))]
struct AlignedStruct {
    first: i16,
    second: i8,
    third: i32
}
```

> Note: As a consequence of the representation being an attribute on the item, the representation does not depend on generic parameters. Any two types with the same name have the same representation. For example, `Foo<Bar>` and `Foo<Baz>` both have the same representation.

The representation of a type can change the padding between fields, but does not change the layout of the fields themselves. For example, a struct with a `C` representation that contains a struct `Inner` with the default representation will not change the layout of `Inner`.

The `Rust` Representation

The `Rust` representation is the default representation for nominal types without a `repr` attribute. Using this representation explicitly through a `repr` attribute is guaranteed to be the same as omitting the attribute entirely.

The only data layout guarantees made by this representation are those required for soundness. They are:

1. The fields are properly aligned.
2. The fields do not overlap.
3. The alignment of the type is at least the maximum alignment of its fields.

Formally, the first guarantee means that the offset of any field is divisible by that field's alignment. The second guarantee means that the fields can be ordered such that the offset plus the size of any field is less than or equal to the offset of the next field in the ordering. The ordering does not have to be the same as the order in which the fields are specified in the declaration of the type.

Be aware that the second guarantee does not imply that the fields have distinct addresses: zero-sized types may have the same address as other fields in the same struct.

There are no other guarantees of data layout made by this representation.

The `C` Representation

The `C` representation is designed for dual purposes. One purpose is for creating types that are interoperable with the C Language. The second purpose is to create types that you can soundly perform operations on that rely on data layout such as reinterpreting values as a different type.

Because of this dual purpose, it is possible to create types that are not useful for interfacing with the C programming language.

This representation can be applied to structs, unions, and enums. The exception is zero-variant enums for which the `C` representation is an error.

`#[repr(C)]` Structs

The alignment of the struct is the alignment of the most-aligned field in it.

The size and offset of fields is determined by the following algorithm.

Start with a current offset of 0 bytes.

For each field in declaration order in the struct, first determine the size and alignment of the field. If the current offset is not a multiple of the field's alignment, then add padding bytes to the current offset until it is a multiple of the field's alignment. The offset for the field is what the current offset is now. Then increase the current offset by the size of the field.

Finally, the size of the struct is the current offset rounded up to the nearest multiple of the struct's alignment.

Here is this algorithm described in pseudocode.

```
/// Returns the amount of padding needed after `offset` to ensure that the
/// following address will be aligned to `alignment`.
fn padding_needed_for(offset: usize, alignment: usize) -> usize {
    let misalignment = offset % alignment;
    if misalignment > 0 {
        // round up to next multiple of `alignment`
        alignment - misalignment
    } else {
        // already a multiple of `alignment`
        0
    }
}

struct.alignment = struct.fields().map(|field| field.alignment).max();

let current_offset = 0;

for field in struct.fields_in_declaration_order() {
    // Increase the current offset so that it's a multiple of the alignment
    // of this field. For the first field, this will always be zero.
    // The skipped bytes are called padding bytes.
    current_offset += padding_needed_for(current_offset, field.alignment);

    struct[field].offset = current_offset;

    current_offset += field.size;
}

struct.size = current_offset + padding_needed_for(current_offset,
struct.alignment);
```

> ⚠️ Warning: This pseudocode uses a naive algorithm that ignores overflow issues for the sake of clarity. To perform memory layout computations in actual code, use `Layout`.

Note: This algorithm can produce zero-sized structs. In C, an empty struct declaration like `struct Foo { }` is illegal. However, both gcc and clang support options to enable such structs, and assign them size zero. C++, in contrast, gives empty structs a size of 1, unless they are inherited from or they are fields that have the `[[no_unique_address]]` attribute, in which case they do not increase the overall size of the struct.

`#[repr(C)]` Unions

A union declared with `#[repr(C)]` will have the same size and alignment as an equivalent C union declaration in the C language for the target platform. The union will have a size of the maximum size of all of its fields rounded to its alignment, and an alignment of the maximum alignment of all of its fields. These maximums may come from different fields.

```
#[repr(C)]
union Union {
    f1: u16,
    f2: [u8; 4],
}

assert_eq!(std::mem::size_of::<Union>(), 4);  // From f2
assert_eq!(std::mem::align_of::<Union>(), 2); // From f1

#[repr(C)]
union SizeRoundedUp {
   a: u32,
   b: [u16; 3],
}

assert_eq!(std::mem::size_of::<SizeRoundedUp>(), 8);  // Size of 6 from b,
                                                      // rounded up to 8 from
                                                      // alignment of a.
assert_eq!(std::mem::align_of::<SizeRoundedUp>(), 4); // From a
```

`#[repr(C)]` Field-less Enums

For field-less enums, the `C` representation has the size and alignment of the default `enum` size and alignment for the target platform's C ABI.

> Note: The enum representation in C is implementation defined, so this is really a "best guess". In particular, this may be incorrect when the C code of interest is compiled with certain flags.

> ⚠️ Warning: There are crucial differences between an `enum` in the C language and Rust's field-less enums with this representation. An `enum` in C is mostly a `typedef` plus some named constants; in other words, an object of an `enum` type can hold any integer value. For example, this is often used for bitflags in `C`. In contrast, Rust's field-less enums can only legally hold the discriminant values, everything else is undefined behavior. Therefore, using a field-less enum in FFI to model a C `enum` is often wrong.

`#[repr(C)]` Enums With Fields

The representation of a `repr(C)` enum with fields is a `repr(C)` struct with two fields, also called a "tagged union" in C:

- a `repr(C)` version of the enum with all fields removed ("the tag")
- a `repr(C)` union of `repr(C)` structs for the fields of each variant that had them ("the payload")

> Note: Due to the representation of `repr(C)` structs and unions, if a variant has a single field there is no difference between putting that field directly in the union or wrapping it in a struct; any system which wishes to manipulate such an `enum`'s representation may therefore use whichever form is more convenient or consistent for them.

```rust
// This Enum has the same representation as ...
#[repr(C)]
enum MyEnum {
    A(u32),
    B(f32, u64),
    C { x: u32, y: u8 },
    D,
 }

// ... this struct.
#[repr(C)]
struct MyEnumRepr {
    tag: MyEnumDiscriminant,
    payload: MyEnumFields,
}

// This is the discriminant enum.
#[repr(C)]
enum MyEnumDiscriminant { A, B, C, D }

// This is the variant union.
#[repr(C)]
union MyEnumFields {
    A: MyAFields,
    B: MyBFields,
    C: MyCFields,
    D: MyDFields,
}

#[repr(C)]
#[derive(Copy, Clone)]
struct MyAFields(u32);

#[repr(C)]
#[derive(Copy, Clone)]
struct MyBFields(f32, u64);

#[repr(C)]
#[derive(Copy, Clone)]
struct MyCFields { x: u32, y: u8 }

// This struct could be omitted (it is a zero-sized type), and it must be in
// C/C++ headers.
#[repr(C)]
#[derive(Copy, Clone)]
struct MyDFields;
```

> Note: `union` s with non- `Copy` fields are unstable, see 55149.

Primitive representations

The *primitive representations* are the representations with the same names as the primitive integer types. That is: `u8`, `u16`, `u32`, `u64`, `u128`, `usize`, `i8`, `i16`, `i32`, `i64`, `i128`, and `isize`.

Primitive representations can only be applied to enumerations and have different behavior whether the enum has fields or no fields. It is an error for zero-variant enums to have a primitive representation. Combining two primitive representations together is an error.

Primitive Representation of Field-less Enums

For field-less enums, primitive representations set the size and alignment to be the same as the primitive type of the same name. For example, a field-less enum with a `u8` representation can only have discriminants between 0 and 255 inclusive.

Primitive Representation of Enums With Fields

The representation of a primitive representation enum is a `repr(C)` union of `repr(C)` structs for each variant with a field. The first field of each struct in the union is the primitive representation version of the enum with all fields removed ("the tag") and the remaining fields are the fields of that variant.

> Note: This representation is unchanged if the tag is given its own member in the union, should that make manipulation more clear for you (although to follow the C++ standard the tag member should be wrapped in a `struct`).

```
// This enum has the same representation as ...
#[repr(u8)]
enum MyEnum {
    A(u32),
    B(f32, u64),
    C { x: u32, y: u8 },
    D,
 }

// ... this union.
#[repr(C)]
union MyEnumRepr {
    A: MyVariantA,
    B: MyVariantB,
    C: MyVariantC,
    D: MyVariantD,
}

// This is the discriminant enum.
#[repr(u8)]
#[derive(Copy, Clone)]
enum MyEnumDiscriminant { A, B, C, D }

#[repr(C)]
#[derive(Clone, Copy)]
struct MyVariantA(MyEnumDiscriminant, u32);

#[repr(C)]
#[derive(Clone, Copy)]
struct MyVariantB(MyEnumDiscriminant, f32, u64);

#[repr(C)]
#[derive(Clone, Copy)]
struct MyVariantC { tag: MyEnumDiscriminant, x: u32, y: u8 }

#[repr(C)]
#[derive(Clone, Copy)]
struct MyVariantD(MyEnumDiscriminant);
```

> Note: `union` s with non- `Copy` fields are unstable, see 55149.

Combining primitive representations of enums with fields and `#[repr(C)]`

For enums with fields, it is also possible to combine `repr(C)` and a primitive representation (e.g., `repr(C, u8)`). This modifies the `repr(C)` by changing the representation of the discriminant enum to the chosen primitive instead. So, if you chose the `u8` representation, then the discriminant enum would have a size and alignment of 1 byte.

The discriminant enum from the example earlier then becomes:

```
#[repr(C, u8)] // `u8` was added
enum MyEnum {
    A(u32),
    B(f32, u64),
    C { x: u32, y: u8 },
    D,
 }

// ...

#[repr(u8)] // So `u8` is used here instead of `C`
enum MyEnumDiscriminant { A, B, C, D }

// ...
```

For example, with a `repr(C, u8)` enum it is not possible to have 257 unique discriminants ("tags") whereas the same enum with only a `repr(C)` attribute will compile without any problems.

Using a primitive representation in addition to `repr(C)` can change the size of an enum from the `repr(C)` form:

```
#[repr(C)]
enum EnumC {
    Variant0(u8),
    Variant1,
}

#[repr(C, u8)]
enum Enum8 {
    Variant0(u8),
    Variant1,
}

#[repr(C, u16)]
enum Enum16 {
    Variant0(u8),
    Variant1,
}

// The size of the C representation is platform dependant
assert_eq!(std::mem::size_of::<EnumC>(), 8);
// One byte for the discriminant and one byte for the value in Enum8::Variant0
assert_eq!(std::mem::size_of::<Enum8>(), 2);
// Two bytes for the discriminant and one byte for the value in Enum16:
:Variant0
// plus one byte of padding.
assert_eq!(std::mem::size_of::<Enum16>(), 4);
```

The alignment modifiers

The `align` and `packed` modifiers can be used to respectively raise or lower the alignment of `struct` s and `union` s. `packed` may also alter the padding between fields (although it will not alter the padding inside of any field). On their own, `align` and `packed` do not provide guarantees about the order of fields in the layout of a struct or the layout of an enum variant, although they may be combined with representations (such as `C`) which do provide such guarantees.

The alignment is specified as an integer parameter in the form of `#[repr(align(x))]` or `#[repr(packed(x))]` . The alignment value must be a power of two from 1 up to 2^{29}. For `packed` , if no value is given, as in `#[repr(packed)]` , then the value is 1.

For `align` , if the specified alignment is less than the alignment of the type without the `align` modifier, then the alignment is unaffected.

For `packed` , if the specified alignment is greater than the type's alignment without the `packed` modifier, then the alignment and layout is unaffected. The alignments of each field, for the purpose of positioning fields, is the smaller of the specified alignment and the alignment of the field's type. Inter-field padding is guaranteed to be the minimum required in order to satisfy each field's (possibly altered) alignment (although note that, on its own, `packed` does not provide any guarantee about field ordering). An important consequence of these rules is that a type with `#[repr(packed(1))]` (or `#[repr(packed)]`) will have no inter-field padding.

The `align` and `packed` modifiers cannot be applied on the same type and a `packed` type cannot transitively contain another `align` ed type. `align` and `packed` may only be applied to the `Rust` and `C` representations.

The `align` modifier can also be applied on an `enum` . When it is, the effect on the `enum` 's alignment is the same as if the `enum` was wrapped in a newtype `struct` with the same `align` modifier.

> Note: References to unaligned fields are not allowed because it is undefined behavior. When fields are unaligned due to an alignment modifier, consider the following options for using references and dereferences:

```
#[repr(packed)]
struct Packed {
    f1: u8,
    f2: u16,
}
let mut e = Packed { f1: 1, f2: 2 };
// Instead of creating a reference to a field, copy the value to a local
variable.
let x = e.f2;
// Or in situations like `println!` which creates a reference, use braces
// to change it to a copy of the value.
println!("{}", {e.f2});
// Or if you need a pointer, use the unaligned methods for reading and
writing
// instead of dereferencing the pointer directly.
let ptr: *const u16 = std::ptr::addr_of!(e.f2);
let value = unsafe { ptr.read_unaligned() };
let mut_ptr: *mut u16 = std::ptr::addr_of_mut!(e.f2);
unsafe { mut_ptr.write_unaligned(3) }
```

The `transparent` Representation

The `transparent` representation can only be used on a `struct` or an `enum` with a single variant that has:

- a single field with non-zero size, and
- any number of fields with size 0 and alignment 1 (e.g. `PhantomData<T>`).

Structs and enums with this representation have the same layout and ABI as the single non-zero sized field.

This is different than the `C` representation because a struct with the `C` representation will always have the ABI of a `C` `struct` while, for example, a struct with the `transparent` representation with a primitive field will have the ABI of the primitive field.

Because this representation delegates type layout to another type, it cannot be used with any other representation.

Interior Mutability

Sometimes a type needs to be mutated while having multiple aliases. In Rust this is achieved using a pattern called *interior mutability*. A type has interior mutability if its internal state can be changed through a shared reference to it. This goes against the usual requirement that the value pointed to by a shared reference is not mutated.

`std::cell::UnsafeCell<T>` type is the only allowed way to disable this requirement. When `UnsafeCell<T>` is immutably aliased, it is still safe to mutate, or obtain a mutable reference to, the `T` it contains. As with all other types, it is undefined behavior to have multiple `&mut UnsafeCell<T>` aliases.

Other types with interior mutability can be created by using `UnsafeCell<T>` as a field. The standard library provides a variety of types that provide safe interior mutability APIs. For example, `std::cell::RefCell<T>` uses run-time borrow checks to ensure the usual rules around multiple references. The `std::sync::atomic` module contains types that wrap a value that is only accessed with atomic operations, allowing the value to be shared and mutated across threads.

Subtyping and Variance

Subtyping is implicit and can occur at any stage in type checking or inference. Subtyping is restricted to two cases: variance with respect to lifetimes and between types with higher ranked lifetimes. If we were to erase lifetimes from types, then the only subtyping would be due to type equality.

Consider the following example: string literals always have `'static` lifetime. Nevertheless, we can assign `s` to `t`:

```
fn bar<'a>() {
    let s: &'static str = "hi";
    let t: &'a str = s;
}
```

Since `'static` outlives the lifetime parameter `'a`, `&'static str` is a subtype of `&'a str`.

Higher-ranked function pointers and trait objects have another subtype relation. They are subtypes of types that are given by substitutions of the higher-ranked lifetimes. Some examples:

```
// Here 'a is substituted for 'static
let subtype: &(for<'a> fn(&'a i32) -> &'a i32) = &((|x| x) as fn(&_) -> &_);
let supertype: &(fn(&'static i32) -> &'static i32) = subtype;

// This works similarly for trait objects
let subtype: &(dyn for<'a> Fn(&'a i32) -> &'a i32) = &|x| x;
let supertype: &(dyn Fn(&'static i32) -> &'static i32) = subtype;

// We can also substitute one higher-ranked lifetime for another
let subtype: &(for<'a, 'b> fn(&'a i32, &'b i32))= &((|x, y| {}) as fn(&_, &_));
let supertype: &for<'c> fn(&'c i32, &'c i32) = subtype;
```

Variance

Variance is a property that generic types have with respect to their arguments. A generic type's *variance* in a parameter is how the subtyping of the parameter affects the subtyping of the type.

- `F<T>` is *covariant* over `T` if `T` being a subtype of `U` implies that `F<T>` is a subtype of `F<U>` (subtyping "passes through")

- `F<T>` is *contravariant* over `T` if `T` being a subtype of `U` implies that `F<U>` is a subtype of `F<T>`
- `F<T>` is *invariant* over `T` otherwise (no subtyping relation can be derived)

Variance of types is automatically determined as follows

Type	Variance in `'a`	Variance in `T`
`&'a T`	covariant	covariant
`&'a mut T`	covariant	invariant
`*const T`		covariant
`*mut T`		invariant
`[T]` and `[T; n]`		covariant
`fn() -> T`		covariant
`fn(T) -> ()`		contravariant
`std::cell::UnsafeCell<T>`		invariant
`std::marker::PhantomData<T>`		covariant
`dyn Trait<T> + 'a`	covariant	invariant

The variance of other `struct`, `enum`, and `union` types is decided by looking at the variance of the types of their fields. If the parameter is used in positions with different variances then the parameter is invariant. For example the following struct is covariant in `'a` and `T` and invariant in `'b`, `'c`, and `U`.

```
use std::cell::UnsafeCell;
struct Variance<'a, 'b, 'c, T, U: 'a> {
    x: &'a U,               // This makes `Variance` covariant in 'a, and would
                            // make it covariant in U, but U is used later
    y: *const T,            // Covariant in T
    z: UnsafeCell<&'b f64>, // Invariant in 'b
    w: *mut U,              // Invariant in U, makes the whole struct invariant

    f: fn(&'c ()) -> &'c () // Both co- and contravariant, makes 'c invariant
                            // in the struct.
}
```

When used outside of an `struct`, `enum`, or `union`, the variance for parameters is checked at each location separately.

```
fn generic_tuple<'short, 'long: 'short>(
    // 'long is used inside of a tuple in both a co- and invariant position.
    x: (&'long u32, UnsafeCell<&'long u32>),
) {
    // As the variance at these positions is computed separately,
    // we can freely shrink 'long in the covariant position.
    let _: (&'short u32, UnsafeCell<&'long u32>) = x;
}

fn takes_fn_ptr<'short, 'middle: 'short>(
    // 'middle is used in both a co- and contravariant position.
    f: fn(&'middle ()) -> &'middle (),
) {
    // As the variance at these positions is computed separately,
    // we can freely shrink 'middle in the covariant position
    // and extend it in the contravariant position.
    let _: fn(&'static ()) -> &'short () = f;
}
```

Trait and lifetime bounds

Syntax

TypeParamBounds :
  TypeParamBound (`+` *TypeParamBound*)* `+`?

TypeParamBound :
  Lifetime | *TraitBound*

TraitBound :
  `?`? *ForLifetimes*? *TypePath*
  | `(` `?`? *ForLifetimes*? *TypePath* `)`

LifetimeBounds :
  (*Lifetime* `+`)* *Lifetime*?

Lifetime :
  LIFETIME_OR_LABEL
  | `'static`
  | `'_`

Trait and lifetime bounds provide a way for generic items to restrict which types and lifetimes are used as their parameters. Bounds can be provided on any type in a where clause. There are also shorter forms for certain common cases:

- Bounds written after declaring a generic parameter: `fn f<A: Copy>() {}` is the same as `fn f<A>() where A: Copy {}`.
- In trait declarations as supertraits: `trait Circle : Shape {}` is equivalent to `trait Circle where Self : Shape {}`.
- In trait declarations as bounds on associated types: `trait A { type B: Copy; }` is equivalent to `trait A where Self::B: Copy { type B; }`.

Bounds on an item must be satisfied when using the item. When type checking and borrow checking a generic item, the bounds can be used to determine that a trait is implemented for a type. For example, given `Ty: Trait`

- In the body of a generic function, methods from `Trait` can be called on `Ty` values. Likewise associated constants on the `Trait` can be used.
- Associated types from `Trait` can be used.

- Generic functions and types with a `T: Trait` bounds can be used with `Ty` being used for `T`.

```
trait Shape {
    fn draw(&self, surface: Surface);
    fn name() -> &'static str;
}

fn draw_twice<T: Shape>(surface: Surface, sh: T) {
    sh.draw(surface);           // Can call method because T: Shape
    sh.draw(surface);
}

fn copy_and_draw_twice<T: Copy>(surface: Surface, sh: T) where T: Shape {
    let shape_copy = sh;        // doesn't move sh because T: Copy
    draw_twice(surface, sh);    // Can use generic function because T: Shape
}

struct Figure<S: Shape>(S, S);

fn name_figure<U: Shape>(
    figure: Figure<U>,          // Type Figure<U> is well-formed because U:
Shape
) {
    println!(
        "Figure of two {}",
        U::name(),              // Can use associated function
    );
}
```

Bounds that don't use the item's parameters or higher-ranked lifetimes are checked when the item is defined. It is an error for such a bound to be false.

`Copy`, `Clone`, and `Sized` bounds are also checked for certain generic types when using the item, even if the use does not provide a concrete type. It is an error to have `Copy` or `Clone` as a bound on a mutable reference, trait object, or slice. It is an error to have `Sized` as a bound on a trait object or slice.

```
struct A<'a, T>
where
    i32: Default,           // Allowed, but not useful
    i32: Iterator,          // Error: `i32` is not an iterator
    &'a mut T: Copy,        // (at use) Error: the trait bound is not satisfied
    [T]: Sized,             // (at use) Error: size cannot be known at
compilation
{
    f: &'a T,
}
struct UsesA<'a, T>(A<'a, T>);
```

Trait and lifetime bounds are also used to name trait objects.

`?Sized`

`?` is only used to relax the implicit `Sized` trait bound for type parameters or associated types. `?Sized` may not be used as a bound for other types.

Lifetime bounds

Lifetime bounds can be applied to types or to other lifetimes. The bound `'a: 'b` is usually read as `'a` *outlives* `'b`. `'a: 'b` means that `'a` lasts at least as long as `'b`, so a reference `&'a ()` is valid whenever `&'b ()` is valid.

```
fn f<'a, 'b>(x: &'a i32, mut y: &'b i32) where 'a: 'b {
    y = x;                      // &'a i32 is a subtype of &'b i32 because 'a:
'b
    let r: &'b &'a i32 = &&0;   // &'b &'a i32 is well formed because 'a: 'b
}
```

`T: 'a` means that all lifetime parameters of `T` outlive `'a`. For example, if `'a` is an unconstrained lifetime parameter, then `i32: 'static` and `&'static str: 'a` are satisfied, but `Vec<&'a ()>: 'static` is not.

Higher-ranked trait bounds

ForLifetimes :
`for` [*GenericParams*]

Trait bounds may be *higher ranked* over lifetimes. These bounds specify a bound that is true *for all* lifetimes. For example, a bound such as `for<'a> &'a T: PartialEq<i32>` would require an implementation like

```
impl<'a> PartialEq<i32> for &'a T {
    // ...
}
```

and could then be used to compare a `&'a T` with any lifetime to an `i32`.

Only a higher-ranked bound can be used here, because the lifetime of the reference is shorter than any possible lifetime parameter on the function:

```
fn call_on_ref_zero<F>(f: F) where for<'a> F: Fn(&'a i32) {
    let zero = 0;
    f(&zero);
}
```

Higher-ranked lifetimes may also be specified just before the trait: the only difference is the scope of the lifetime parameter, which extends only to the end of the following trait instead of the whole bound. This function is equivalent to the last one.

```
fn call_on_ref_zero<F>(f: F) where F: for<'a> Fn(&'a i32) {
    let zero = 0;
    f(&zero);
}
```

Implied bounds

Lifetime bounds required for types to be well-formed are sometimes inferred.

```
fn requires_t_outlives_a<'a, T>(x: &'a T) {}
```

The type parameter `T` is required to outlive `'a` for the type `&'a T` to be well-formed. This is inferred because the function signature contains the type `&'a T` which is only valid if `T: 'a` holds.

Implied bounds are added for all parameters and outputs of functions. Inside of `requires_t_outlives_a` you can assume `T: 'a` to hold even if you don't explicitly specify this:

```
fn requires_t_outlives_a_not_implied<'a, T: 'a>() {}

fn requires_t_outlives_a<'a, T>(x: &'a T) {
    // This compiles, because `T: 'a` is implied by
    // the reference type `&'a T`.
    requires_t_outlives_a_not_implied::<'a, T>();
}


fn not_implied<'a, T>() {
    // This errors, because `T: 'a` is not implied by
    // the function signature.
    requires_t_outlives_a_not_implied::<'a, T>();
}
```

Only lifetime bounds are implied, trait bounds still have to be explicitly added. The following example therefore causes an error:

```
use std::fmt::Debug;
struct IsDebug<T: Debug>(T);
// error[E0277]: `T` doesn't implement `Debug`
fn doesnt_specify_t_debug<T>(x: IsDebug<T>) {}
```

Lifetime bounds are also inferred for type definitions and impl blocks for any type:

```rust
struct Struct<'a, T> {
    // This requires `T: 'a` to be well-formed
    // which is inferred by the compiler.
    field: &'a T,
}

enum Enum<'a, T> {
    // This requires `T: 'a` to be well-formed,
    // which is inferred by the compiler.
    //
    // Note that `T: 'a` is required even when only
    // using `Enum::OtherVariant`.
    SomeVariant(&'a T),
    OtherVariant,
}

trait Trait<'a, T: 'a> {}

// This would error because `T: 'a` is not implied by any type
// in the impl header.
//     impl<'a, T> Trait<'a, T> for () {}

// This compiles as `T: 'a` is implied by the self type `&'a T`.
impl<'a, T> Trait<'a, T> for &'a T {}
```

Type coercions

Type coercions are implicit operations that change the type of a value. They happen automatically at specific locations and are highly restricted in what types actually coerce.

Any conversions allowed by coercion can also be explicitly performed by the type cast operator, `as`.

Coercions are originally defined in RFC 401 and expanded upon in RFC 1558.

Coercion sites

A coercion can only occur at certain coercion sites in a program; these are typically places where the desired type is explicit or can be derived by propagation from explicit types (without type inference). Possible coercion sites are:

- `let` statements where an explicit type is given.

  For example, `&mut 42` is coerced to have type `&i8` in the following:

  ```
  let _: &i8 = &mut 42;
  ```

- `static` and `const` item declarations (similar to `let` statements).

- Arguments for function calls

  The value being coerced is the actual parameter, and it is coerced to the type of the formal parameter.

  For example, `&mut 42` is coerced to have type `&i8` in the following:

  ```
  fn bar(_: &i8) { }

  fn main() {
      bar(&mut 42);
  }
  ```

  For method calls, the receiver (`self` parameter) type is coerced differently, see the documentation on method-call expressions for details.

- Instantiations of struct, union, or enum variant fields

  For example, `&mut 42` is coerced to have type `&i8` in the following:

  ```
  struct Foo<'a> { x: &'a i8 }

  fn main() {
      Foo { x: &mut 42 };
  }
  ```

- Function results—either the final line of a block if it is not semicolon-terminated or any expression in a `return` statement

  For example, `x` is coerced to have type `&dyn Display` in the following:

  ```
  use std::fmt::Display;
  fn foo(x: &u32) -> &dyn Display {
      x
  }
  ```

If the expression in one of these coercion sites is a coercion-propagating expression, then the relevant sub-expressions in that expression are also coercion sites. Propagation recurses from these new coercion sites. Propagating expressions and their relevant sub-expressions are:

- Array literals, where the array has type `[U; n]`. Each sub-expression in the array literal is a coercion site for coercion to type `U`.

- Array literals with repeating syntax, where the array has type `[U; n]`. The repeated sub-expression is a coercion site for coercion to type `U`.

- Tuples, where a tuple is a coercion site to type `(U_0, U_1, ..., U_n)`. Each sub-expression is a coercion site to the respective type, e.g. the zeroth sub-expression is a coercion site to type `U_0`.

- Parenthesized sub-expressions (`(e)`): if the expression has type `U`, then the sub-expression is a coercion site to `U`.

- Blocks: if a block has type `U`, then the last expression in the block (if it is not semicolon-terminated) is a coercion site to `U`. This includes blocks which are part of control flow statements, such as `if` / `else`, if the block has a known type.

Coercion types

Coercion is allowed between the following types:

- `T` to `U` if `T` is a subtype of `U` (*reflexive case*)
- `T_1` to `T_3` where `T_1` coerces to `T_2` and `T_2` coerces to `T_3` (*transitive case*)

  Note that this is not fully supported yet.
- `&mut T` to `&T`
- `*mut T` to `*const T`
- `&T` to `*const T`
- `&mut T` to `*mut T`
- `&T` or `&mut T` to `&U` if `T` implements `Deref<Target = U>` . For example:

  ```
  use std::ops::Deref;

  struct CharContainer {
      value: char,
  }

  impl Deref for CharContainer {
      type Target = char;

      fn deref<'a>(&'a self) -> &'a char {
          &self.value
      }
  }

  fn foo(arg: &char) {}

  fn main() {
      let x = &mut CharContainer { value: 'y' };
      foo(x); //&mut CharContainer is coerced to &char.
  }
  ```
- `&mut T` to `&mut U` if `T` implements `DerefMut<Target = U>` .

- TyCtor(`T`) to TyCtor(`U`), where TyCtor(`T`) is one of
    - `&T`
    - `&mut T`
    - `*const T`
    - `*mut T`
    - `Box<T>`

  and where `U` can be obtained from `T` by unsized coercion.

- Function item types to `fn` pointers
- Non capturing closures to `fn` pointers
- `!` to any `T`

Unsized Coercions

The following coercions are called `unsized coercions`, since they relate to converting sized types to unsized types, and are permitted in a few cases where other coercions are not, as described above. They can still happen anywhere else a coercion can occur.

Two traits, `Unsize` and `CoerceUnsized`, are used to assist in this process and expose it for library use. The following coercions are built-ins and, if `T` can be coerced to `U` with one of them, then an implementation of `Unsize<U>` for `T` will be provided:

- `[T; n]` to `[T]`.
- `T` to `dyn U`, when `T` implements `U + Sized`, and `U` is object safe.
- `Foo<..., T, ...>` to `Foo<..., U, ...>`, when:
    - `Foo` is a struct.
    - `T` implements `Unsize<U>`.
    - The last field of `Foo` has a type involving `T`.
    - If that field has type `Bar<T>`, then `Bar<T>` implements `Unsized<Bar<U>>`.
    - T is not part of the type of any other fields.

Additionally, a type `Foo<T>` can implement `CoerceUnsized<Foo<U>>` when `T` implements `Unsize<U>` or `CoerceUnsized<Foo<U>>`. This allows it to provide a unsized coercion to `Foo<U>`.

> Note: While the definition of the unsized coercions and their implementation has been stabilized, the traits themselves are not yet stable and therefore can't be used directly in stable Rust.

Least upper bound coercions

In some contexts, the compiler must coerce together multiple types to try and find the most general type. This is called a "Least Upper Bound" coercion. LUB coercion is used and only used in the following situations:

- To find the common type for a series of if branches.
- To find the common type for a series of match arms.
- To find the common type for array elements.
- To find the type for the return type of a closure with multiple return statements.
- To check the type for the return type of a function with multiple return statements.

In each such case, there are a set of types `T0..Tn` to be mutually coerced to some target type `T_t`, which is unknown to start. Computing the LUB coercion is done iteratively. The target type `T_t` begins as the type `T0`. For each new type `Ti`, we consider whether

- If `Ti` can be coerced to the current target type `T_t`, then no change is made.
- Otherwise, check whether `T_t` can be coerced to `Ti`; if so, the `T_t` is changed to `Ti`. (This check is also conditioned on whether all of the source expressions considered thus far have implicit coercions.)
- If not, try to compute a mutual supertype of `T_t` and `Ti`, which will become the new target type.

Examples:

```
// For if branches
let bar = if true {
    a
} else if false {
    b
} else {
    c
};

// For match arms
let baw = match 42 {
    0 => a,
    1 => b,
    _ => c,
};

// For array elements
let bax = [a, b, c];

// For closure with multiple return statements
let clo = || {
    if true {
        a
    } else if false {
        b
    } else {
        c
    }
};
let baz = clo();

// For type checking of function with multiple return statements
fn foo() -> i32 {
    let (a, b, c) = (0, 1, 2);
    match 42 {
        0 => a,
        1 => b,
        _ => c,
    }
}
```

In these examples, types of the `ba*` are found by LUB coercion. And the compiler checks whether LUB coercion result of `a`, `b`, `c` is `i32` in the processing of the function `foo`.

Caveat

This description is obviously informal. Making it more precise is expected to proceed as part of a general effort to specify the Rust type checker more precisely.

Destructors

When an initialized variable or temporary goes out of scope, its *destructor* is run, or it is *dropped*. Assignment also runs the destructor of its left-hand operand, if it's initialized. If a variable has been partially initialized, only its initialized fields are dropped.

The destructor of a type `T` consists of:

1. If `T: Drop`, calling `<T as std::ops::Drop>::drop`
2. Recursively running the destructor of all of its fields.
   - The fields of a struct are dropped in declaration order.
   - The fields of the active enum variant are dropped in declaration order.
   - The fields of a tuple are dropped in order.
   - The elements of an array or owned slice are dropped from the first element to the last.
   - The variables that a closure captures by move are dropped in an unspecified order.
   - Trait objects run the destructor of the underlying type.
   - Other types don't result in any further drops.

If a destructor must be run manually, such as when implementing your own smart pointer, `std::ptr::drop_in_place` can be used.

Some examples:

```rust
struct PrintOnDrop(&'static str);

impl Drop for PrintOnDrop {
    fn drop(&mut self) {
        println!("{}", self.0);
    }
}

let mut overwritten = PrintOnDrop("drops when overwritten");
overwritten = PrintOnDrop("drops when scope ends");

let tuple = (PrintOnDrop("Tuple first"), PrintOnDrop("Tuple second"));

let moved;
// No destructor run on assignment.
moved = PrintOnDrop("Drops when moved");
// Drops now, but is then uninitialized.
moved;

// Uninitialized does not drop.
let uninitialized: PrintOnDrop;

// After a partial move, only the remaining fields are dropped.
let mut partial_move = (PrintOnDrop("first"), PrintOnDrop("forgotten"));
// Perform a partial move, leaving only `partial_move.0` initialized.
core::mem::forget(partial_move.1);
// When partial_move's scope ends, only the first field is dropped.
```

Drop scopes

Each variable or temporary is associated to a *drop scope*. When control flow leaves a drop scope all variables associated to that scope are dropped in reverse order of declaration (for variables) or creation (for temporaries).

Drop scopes are determined after replacing `for`, `if let`, and `while let` expressions with the equivalent expressions using `match`. Overloaded operators are not distinguished from built-in operators and binding modes are not considered.

Given a function, or closure, there are drop scopes for:

- The entire function
- Each statement
- Each expression
- Each block, including the function body
  - In the case of a block expression, the scope for the block and the expression are

the same scope.

- Each arm of a `match` expression

Drop scopes are nested within one another as follows. When multiple scopes are left at once, such as when returning from a function, variables are dropped from the inside outwards.

- The entire function scope is the outer most scope.
- The function body block is contained within the scope of the entire function.
- The parent of the expression in an expression statement is the scope of the statement.
- The parent of the initializer of a `let` statement is the `let` statement's scope.
- The parent of a statement scope is the scope of the block that contains the statement.
- The parent of the expression for a `match` guard is the scope of the arm that the guard is for.
- The parent of the expression after the `=>` in a `match` expression is the scope of the arm that it's in.
- The parent of the arm scope is the scope of the `match` expression that it belongs to.
- The parent of all other scopes is the scope of the immediately enclosing expression.

Scopes of function parameters

All function parameters are in the scope of the entire function body, so are dropped last when evaluating the function. Each actual function parameter is dropped after any bindings introduced in that parameter's pattern.

```
// Drops `y`, then the second parameter, then `x`, then the first parameter
fn patterns_in_parameters(
    (x, _): (PrintOnDrop, PrintOnDrop),
    (_, y): (PrintOnDrop, PrintOnDrop),
) {}

// drop order is 3 2 0 1
patterns_in_parameters(
    (PrintOnDrop("0"), PrintOnDrop("1")),
    (PrintOnDrop("2"), PrintOnDrop("3")),
);
```

Scopes of local variables

Local variables declared in a `let` statement are associated to the scope of the block that contains the `let` statement. Local variables declared in a `match` expression are associated to the arm scope of the `match` arm that they are declared in.

```
let declared_first = PrintOnDrop("Dropped last in outer scope");
{
    let declared_in_block = PrintOnDrop("Dropped in inner scope");
}
let declared_last = PrintOnDrop("Dropped first in outer scope");
```

If multiple patterns are used in the same arm for a `match` expression, then an unspecified pattern will be used to determine the drop order.

Temporary scopes

The *temporary scope* of an expression is the scope that is used for the temporary variable that holds the result of that expression when used in a place context, unless it is promoted.

Apart from lifetime extension, the temporary scope of an expression is the smallest scope that contains the expression and is one of the following:

- The entire function.
- A statement.
- The body of an `if`, `while` or `loop` expression.
- The `else` block of an `if` expression.
- The condition expression of an `if` or `while` expression, or a `match` guard.
- The body expression for a match arm.
- The second operand of a lazy boolean expression.

Notes:

Temporaries that are created in the final expression of a function body are dropped *after* any named variables bound in the function body. Their drop scope is the entire function, as there is no smaller enclosing temporary scope.

The scrutinee of a `match` expression is not a temporary scope, so temporaries in the scrutinee can be dropped after the `match` expression. For example, the temporary for `1` in `match 1 { ref mut z => z };` lives until the end of the statement.

Some examples:

```
let local_var = PrintOnDrop("local var");

// Dropped once the condition has been evaluated
if PrintOnDrop("If condition").0 == "If condition" {
    // Dropped at the end of the block
    PrintOnDrop("If body").0
} else {
    unreachable!()
};

// Dropped at the end of the statement
(PrintOnDrop("first operand").0 == ""
// Dropped at the )
|| PrintOnDrop("second operand").0 == "")
// Dropped at the end of the expression
|| PrintOnDrop("third operand").0 == "";

// Dropped at the end of the function, after local variables.
// Changing this to a statement containing a return expression would make the
// temporary be dropped before the local variables. Binding to a variable
// which is then returned would also make the temporary be dropped first.
match PrintOnDrop("Matched value in final expression") {
    // Dropped once the condition has been evaluated
    _ if PrintOnDrop("guard condition").0 == "" => (),
    _ => (),
}
```

Operands

Temporaries are also created to hold the result of operands to an expression while the other operands are evaluated. The temporaries are associated to the scope of the expression with that operand. Since the temporaries are moved from once the expression is evaluated, dropping them has no effect unless one of the operands to an expression breaks out of the expression, returns, or panics.

```
loop {
    // Tuple expression doesn't finish evaluating so operands drop in reverse
order
    (
        PrintOnDrop("Outer tuple first"),
        PrintOnDrop("Outer tuple second"),
        (
            PrintOnDrop("Inner tuple first"),
            PrintOnDrop("Inner tuple second"),
            break,
        ),
        PrintOnDrop("Never created"),
    );
}
```

Constant promotion

Promotion of a value expression to a `'static` slot occurs when the expression could be written in a constant and borrowed, and that borrow could be dereferenced where the expression was originally written, without changing the runtime behavior. That is, the promoted expression can be evaluated at compile-time and the resulting value does not contain interior mutability or destructors (these properties are determined based on the value where possible, e.g. `&None` always has the type `&'static Option<_>`, as it contains nothing disallowed).

Temporary lifetime extension

> **Note**: The exact rules for temporary lifetime extension are subject to change. This is describing the current behavior only.

The temporary scopes for expressions in `let` statements are sometimes *extended* to the scope of the block containing the `let` statement. This is done when the usual temporary scope would be too small, based on certain syntactic rules. For example:

```
let x = &mut 0;
// Usually a temporary would be dropped by now, but the temporary for `0` lives
// to the end of the block.
println!("{}", x);
```

If a borrow, dereference, field, or tuple indexing expression has an extended temporary scope then so does its operand. If an indexing expression has an extended temporary scope then the indexed expression also has an extended temporary scope.

Extending based on patterns

An *extending pattern* is either

- An identifier pattern that binds by reference or mutable reference.
- A struct, tuple, tuple struct, or slice pattern where at least one of the direct subpatterns is an extending pattern.

So `ref x`, `V(ref x)` and `[ref x, y]` are all extending patterns, but `x`, `&ref x` and `&(ref x,)` are not.

If the pattern in a `let` statement is an extending pattern then the temporary scope of the initializer expression is extended.

Extending based on expressions

For a let statement with an initializer, an *extending expression* is an expression which is one of the following:

- The initializer expression.
- The operand of an extending borrow expression.
- The operand(s) of an extending array, cast, braced struct, or tuple expression.
- The final expression of any extending block expression.

So the borrow expressions in `&mut 0`, `(&1, &mut 2)`, and `Some { 0: &mut 3 }` are all extending expressions. The borrows in `&0 + &1` and `Some(&mut 0)` are not: the latter is syntactically a function call expression.

The operand of any extending borrow expression has its temporary scope extended.

Examples

Here are some examples where expressions have extended temporary scopes:

```
// The temporary that stores the result of `temp()` lives in the same scope
// as x in these cases.
let x = &temp();
let x = &temp() as &dyn Send;
let x = (&*&temp(),);
let x = { [Some { 0: &temp(), }] };
let ref x = temp();
let ref x = *&temp();
```

Here are some examples where expressions don't have extended temporary scopes:

```
// The temporary that stores the result of `temp()` only lives until the
// end of the let statement in these cases.

let x = Some(&temp());         // ERROR
let x = (&temp()).use_temp();  // ERROR
```

Not running destructors

`std::mem::forget` can be used to prevent the destructor of a variable from being run, and `std::mem::ManuallyDrop` provides a wrapper to prevent a variable or field from being dropped automatically.

> Note: Preventing a destructor from being run via `std::mem::forget` or other means is safe even if it has a type that isn't `'static`. Besides the places where destructors are guaranteed to run as defined by this document, types may *not* safely rely on a destructor being run for soundness.

Lifetime elision

Rust has rules that allow lifetimes to be elided in various places where the compiler can infer a sensible default choice.

Lifetime elision in functions

In order to make common patterns more ergonomic, lifetime arguments can be *elided* in function item, function pointer, and closure trait signatures. The following rules are used to infer lifetime parameters for elided lifetimes. It is an error to elide lifetime parameters that cannot be inferred. The placeholder lifetime, `'_`, can also be used to have a lifetime inferred in the same way. For lifetimes in paths, using `'_` is preferred. Trait object lifetimes follow different rules discussed below.

- Each elided lifetime in the parameters becomes a distinct lifetime parameter.
- If there is exactly one lifetime used in the parameters (elided or not), that lifetime is assigned to *all* elided output lifetimes.

In method signatures there is another rule

- If the receiver has type `&Self` or `&mut Self`, then the lifetime of that reference to `Self` is assigned to all elided output lifetime parameters.

Examples:

```
fn print1(s: &str);                                   // elided
fn print2(s: &'_ str);                                // also elided
fn print3<'a>(s: &'a str);                            // expanded

fn debug1(lvl: usize, s: &str);                       // elided
fn debug2<'a>(lvl: usize, s: &'a str);                // expanded

fn substr1(s: &str, until: usize) -> &str;            // elided
fn substr2<'a>(s: &'a str, until: usize) -> &'a str;  // expanded

fn get_mut1(&mut self) -> &mut dyn T;                 // elided
fn get_mut2<'a>(&'a mut self) -> &'a mut dyn T;       // expanded

fn args1<T: ToCStr>(&mut self, args: &[T]) -> &mut Command;                  //
elided
fn args2<'a, 'b, T: ToCStr>(&'a mut self, args: &'b [T]) -> &'a mut Command; //
expanded

fn new1(buf: &mut [u8]) -> Thing<'_>;                 // elided - preferred
fn new2(buf: &mut [u8]) -> Thing;                     // elided
fn new3<'a>(buf: &'a mut [u8]) -> Thing<'a>;          // expanded

type FunPtr1 = fn(&str) -> &str;                      // elided
type FunPtr2 = for<'a> fn(&'a str) -> &'a str;        // expanded

type FunTrait1 = dyn Fn(&str) -> &str;                // elided
type FunTrait2 = dyn for<'a> Fn(&'a str) -> &'a str;  // expanded


// The following examples show situations where it is not allowed to elide the
// lifetime parameter.

// Cannot infer, because there are no parameters to infer from.
fn get_str() -> &str;                                 // ILLEGAL

// Cannot infer, ambiguous if it is borrowed from the first or second
parameter.
fn frob(s: &str, t: &str) -> &str;                    // ILLEGAL
```

Default trait object lifetimes

The assumed lifetime of references held by a trait object is called its *default object lifetime bound*. These were defined in RFC 599 and amended in RFC 1156.

These default object lifetime bounds are used instead of the lifetime parameter elision rules defined above when the lifetime bound is omitted entirely. If `'_` is used as the lifetime bound then the bound follows the usual elision rules.

If the trait object is used as a type argument of a generic type then the containing type is first used to try to infer a bound.

- If there is a unique bound from the containing type then that is the default
- If there is more than one bound from the containing type then an explicit bound must be specified

If neither of those rules apply, then the bounds on the trait are used:

- If the trait is defined with a single lifetime *bound* then that bound is used.
- If `'static` is used for any lifetime bound then `'static` is used.
- If the trait has no lifetime bounds, then the lifetime is inferred in expressions and is `'static` outside of expressions.

```
// For the following trait...
trait Foo { }

// These two are the same because Box<T> has no lifetime bound on T
type T1 = Box<dyn Foo>;
type T2 = Box<dyn Foo + 'static>;

// ...and so are these:
impl dyn Foo {}
impl dyn Foo + 'static {}

// ...so are these, because &'a T requires T: 'a
type T3<'a> = &'a dyn Foo;
type T4<'a> = &'a (dyn Foo + 'a);

// std::cell::Ref<'a, T> also requires T: 'a, so these are the same
type T5<'a> = std::cell::Ref<'a, dyn Foo>;
type T6<'a> = std::cell::Ref<'a, dyn Foo + 'a>;


// This is an example of an error.
struct TwoBounds<'a, 'b, T: ?Sized + 'a + 'b> {
    f1: &'a i32,
    f2: &'b i32,
    f3: T,
}
type T7<'a, 'b> = TwoBounds<'a, 'b, dyn Foo>;
//                                  ^^^^^^^
// Error: the lifetime bound for this object type cannot be deduced from
context
```

Note that the innermost object sets the bound, so `&'a Box<dyn Foo>` is still `&'a Box<dyn Foo + 'static>`.

```rust
// For the following trait...
trait Bar<'a>: 'a { }

// ...these two are the same:
type T1<'a> = Box<dyn Bar<'a>>;
type T2<'a> = Box<dyn Bar<'a> + 'a>;

// ...and so are these:
impl<'a> dyn Bar<'a> {}
impl<'a> dyn Bar<'a> + 'a {}
```

`'static` lifetime elision

Both constant and static declarations of reference types have *implicit* `'static` lifetimes unless an explicit lifetime is specified. As such, the constant declarations involving `'static` above may be written without the lifetimes.

```rust
// STRING: &'static str
const STRING: &str = "bitstring";

struct BitsNStrings<'a> {
    mybits: [u32; 2],
    mystring: &'a str,
}

// BITS_N_STRINGS: BitsNStrings<'static>
const BITS_N_STRINGS: BitsNStrings<'_> = BitsNStrings {
    mybits: [1, 2],
    mystring: STRING,
};
```

Note that if the `static` or `const` items include function or closure references, which themselves include references, the compiler will first try the standard elision rules. If it is unable to resolve the lifetimes by its usual rules, then it will error. By way of example:

```rust
// Resolved as `for<'a> fn(&'a str) -> &'a str`.
const RESOLVED_SINGLE: fn(&str) -> &str = |x| x;

// Resolved as `for<'a, 'b, 'c> Fn(&'a Foo, &'b Bar, &'c Baz) -> usize`.
const RESOLVED_MULTIPLE: &dyn Fn(&Foo, &Bar, &Baz) -> usize = &somefunc;
```

```
// There is insufficient information to bound the return reference lifetime
// relative to the argument lifetimes, so this is an error.
const RESOLVED_STATIC: &dyn Fn(&Foo, &Bar) -> &Baz = &somefunc;
//                                            ^
// this function's return type contains a borrowed value, but the signature
// does not say whether it is borrowed from argument 1 or argument 2
```

Special types and traits

Certain types and traits that exist in the standard library are known to the Rust compiler. This chapter documents the special features of these types and traits.

`Box<T>`

`Box<T>` has a few special features that Rust doesn't currently allow for user defined types.

- The dereference operator for `Box<T>` produces a place which can be moved from. This means that the `*` operator and the destructor of `Box<T>` are built-in to the language.
- Methods can take `Box<Self>` as a receiver.
- A trait may be implemented for `Box<T>` in the same crate as `T`, which the orphan rules prevent for other generic types.

`Rc<T>`

Methods can take `Rc<Self>` as a receiver.

`Arc<T>`

Methods can take `Arc<Self>` as a receiver.

`Pin<P>`

Methods can take `Pin<P>` as a receiver.

`UnsafeCell<T>`

`std::cell::UnsafeCell<T>` is used for interior mutability. It ensures that the compiler doesn't perform optimisations that are incorrect for such types. It also ensures that `static` items which have a type with interior mutability aren't placed in memory marked as read only.

`PhantomData<T>`

`std::marker::PhantomData<T>` is a zero-sized, minimum alignment, type that is considered to own a `T` for the purposes of variance, drop check, and auto traits.

Operator Traits

The traits in `std::ops` and `std::cmp` are used to overload operators, indexing expressions, and call expressions.

`Deref` and `DerefMut`

As well as overloading the unary `*` operator, `Deref` and `DerefMut` are also used in method resolution and deref coercions.

`Drop`

The `Drop` trait provides a destructor, to be run whenever a value of this type is to be destroyed.

`Copy`

The `Copy` trait changes the semantics of a type implementing it. Values whose type implements `Copy` are copied rather than moved upon assignment.

`Copy` can only be implemented for types which do not implement `Drop`, and whose fields are all `Copy`. For enums, this means all fields of all variants have to be `Copy`. For unions, this means all variants have to be `Copy`.

`Copy` is implemented by the compiler for

- Tuples of `Copy` types
- Function pointers
- Function items
- Closures that capture no values or that only capture values of `Copy` types

`Clone`

The `Clone` trait is a supertrait of `Copy`, so it also needs compiler generated implementations. It is implemented by the compiler for the following types:

- Types with a built-in `Copy` implementation (see above)
- Tuples of `Clone` types

`Send`

The `Send` trait indicates that a value of this type is safe to send from one thread to another.

`Sync`

The `Sync` trait indicates that a value of this type is safe to share between multiple threads. This trait must be implemented for all types used in immutable `static` items.

`Termination`

The `Termination` trait indicates the acceptable return types for the main function and test functions.

Auto traits

The `Send`, `Sync`, `Unpin`, `UnwindSafe`, and `RefUnwindSafe` traits are *auto traits*. Auto traits have special properties.

If no explicit implementation or negative implementation is written out for an auto trait for a given type, then the compiler implements it automatically according to the following rules:

- `&T`, `&mut T`, `*const T`, `*mut T`, `[T; n]`, and `[T]` implement the trait if `T` does.
- Function item types and function pointers automatically implement the trait.
- Structs, enums, unions, and tuples implement the trait if all of their fields do.
- Closures implement the trait if the types of all of their captures do. A closure that captures a `T` by shared reference and a `U` by value implements any auto traits that both `&T` and `U` do.

For generic types (counting the built-in types above as generic over `T`), if a generic implementation is available, then the compiler does not automatically implement it for types that could use the implementation except that they do not meet the requisite trait bounds. For instance, the standard library implements `Send` for all `&T` where `T` is `Sync`; this means that the compiler will not implement `Send` for `&T` if `T` is `Send` but not `Sync`.

Auto traits can also have negative implementations, shown as `impl !AutoTrait for T` in the standard library documentation, that override the automatic implementations. For example `*mut T` has a negative implementation of `Send`, and so `*mut T` is not `Send`, even if `T` is. There is currently no stable way to specify additional negative implementations; they exist only in the standard library.

Auto traits may be added as an additional bound to any trait object, even though normally only one trait is allowed. For instance, `Box<dyn Debug + Send + UnwindSafe>` is a valid type.

Sized

The `Sized` trait indicates that the size of this type is known at compile-time; that is, it's not a dynamically sized type. Type parameters (except `Self` in traits) are `Sized` by default, as are associated types. `Sized` is always implemented automatically by the compiler, not by implementation items. These implicit `Sized` bounds may be relaxed by using the special `?Sized` bound.

Names

An *entity* is a language construct that can be referred to in some way within the source program, usually via a path. Entities include types, items, generic parameters, variable bindings, loop labels, lifetimes, fields, attributes, and lints.

A *declaration* is a syntactical construct that can introduce a *name* to refer to an entity. Entity names are valid within a *scope* — a region of source text where that name may be referenced.

Some entities are explicitly declared in the source code, and some are implicitly declared as part of the language or compiler extensions.

Paths are used to refer to an entity, possibly in another scope. Lifetimes and loop labels use a dedicated syntax using a leading quote.

Names are segregated into different *namespaces*, allowing entities in different namespaces to share the same name without conflict.

Name resolution is the compile-time process of tying paths, identifiers, and labels to entity declarations.

Access to certain names may be restricted based on their *visibility*.

Explicitly declared entities

Entities that explicitly introduce a name in the source code are:

- Items:
  - Module declarations
  - External crate declarations
  - Use declarations
  - Function declarations and function parameters
  - Type aliases
  - struct, union, enum, enum variant declarations, and their named fields
  - Constant item declarations
  - Static item declarations
  - Trait item declarations and their associated items
  - External block items
  - `macro_rules` declarations and matcher metavariables

    - Implementation associated items
- Expressions:
    - Closure parameters
    - `while let` pattern bindings
    - `for` pattern bindings
    - `if let` pattern bindings
    - `match` pattern bindings
    - Loop labels
- Generic parameters
- Higher ranked trait bounds
- `let` statement pattern bindings
- The `macro_use` attribute can introduce macro names from another crate
- The `macro_export` attribute can introduce an alias for the macro into the crate root

Additionally, macro invocations and attributes can introduce names by expanding to one of the above items.

Implicitly declared entities

The following entities are implicitly defined by the language, or are introduced by compiler options and extensions:

- Language prelude:
    - Boolean type — `bool`
    - Textual types — `char` and `str`
    - Integer types — `i8`, `i16`, `i32`, `i64`, `i128`, `u8`, `u16`, `u32`, `u64`, `u128`
    - Machine-dependent integer types — `usize` and `isize`
    - floating-point types — `f32` and `f64`
- Built-in attributes
- Standard library prelude items, attributes, and macros
- Standard library crates in the root module
- External crates linked by the compiler
- Tool attributes
- Lints and tool lint attributes
- Derive helper attributes are valid within an item without being explicitly imported
- The `'static` lifetime

Additionally, the crate root module does not have a name, but can be referred to with certain path qualifiers or aliases.

Namespaces

A *namespace* is a logical grouping of declared names. Names are segregated into separate namespaces based on the kind of entity the name refers to. Namespaces allow the occurrence of a name in one namespace to not conflict with the same name in another namespace.

Within a namespace, names are organized in a hierarchy, where each level of the hierarchy has its own collection of named entities.

There are several different namespaces that each contain different kinds of entities. The usage of a name will look for the declaration of that name in different namespaces, based on the context, as described in the name resolution chapter.

The following is a list of namespaces, with their corresponding entities:

- Type Namespace
  - Module declarations
  - External crate declarations
  - External crate prelude items
  - Struct, union, enum, enum variant declarations
  - Trait item declarations
  - Type aliases
  - Associated type declarations
  - Built-in types: boolean, numeric, and textual
  - Generic type parameters
  - `Self` type
  - Tool attribute modules
- Value Namespace
  - Function declarations
  - Constant item declarations
  - Static item declarations
  - Struct constructors
  - Enum variant constructors
  - `Self` constructors
  - Generic const parameters
  - Associated const declarations
  - Associated function declarations
  - Local bindings — `let`, `if let`, `while let`, `for`, `match` arms, function parameters, closure parameters
  - Captured closure variables

- Macro Namespace
    - `macro_rules` declarations
    - Built-in attributes
    - Tool attributes
    - Function-like procedural macros
    - Derive macros
    - Derive macro helpers
    - Attribute macros
- Lifetime Namespace
    - Generic lifetime parameters
- Label Namespace
    - Loop labels
    - Block labels

An example of how overlapping names in different namespaces can be used unambiguously:

```
// Foo introduces a type in the type namespace and a constructor in the value
// namespace.
struct Foo(u32);

// The `Foo` macro is declared in the macro namespace.
macro_rules! Foo {
    () => {};
}

// `Foo` in the `f` parameter type refers to `Foo` in the type namespace.
// `'Foo` introduces a new lifetime in the lifetime namespace.
fn example<'Foo>(f: Foo) {
    // `Foo` refers to the `Foo` constructor in the value namespace.
    let ctor = Foo;
    // `Foo` refers to the `Foo` macro in the macro namespace.
    Foo!{}
    // `'Foo` introduces a label in the label namespace.
    'Foo: loop {
        // `'Foo` refers to the `'Foo` lifetime parameter, and `Foo`
        // refers to the type namespace.
        let x: &'Foo Foo;
        // `'Foo` refers to the label.
        break 'Foo;
    }
}
```

Named entities without a namespace

The following entities have explicit names, but the names are not a part of any specific namespace.

Fields

Even though struct, enum, and union fields are named, the named fields do not live in an explicit namespace. They can only be accessed via a field expression, which only inspects the field names of the specific type being accessed.

Use declarations

A use declaration has named aliases that it imports into scope, but the `use` item itself does not belong to a specific namespace. Instead, it can introduce aliases into multiple namespaces, depending on the item kind being imported.

Sub-namespaces

The macro namespace is split into two sub-namespaces: one for bang-style macros and one for attributes. When an attribute is resolved, any bang-style macros in scope will be ignored. And conversely resolving a bang-style macro will ignore attribute macros in scope. This prevents one style from shadowing another.

For example, the `cfg` attribute and the `cfg` macro are two different entities with the same name in the macro namespace, but they can still be used in their respective context.

It is still an error for a `use` import to shadow another macro, regardless of their sub-namespaces.

Scopes

> **Note**: This is a placeholder for future expansion.

Preludes

A *prelude* is a collection of names that are automatically brought into scope of every module in a crate.

These prelude names are not part of the module itself: they are implicitly queried during name resolution. For example, even though something like `Box` is in scope in every module, you cannot refer to it as `self::Box` because it is not a member of the current module.

There are several different preludes:

- Standard library prelude
- Extern prelude
- Language prelude
- `macro_use` prelude
- Tool prelude

Standard library prelude

Each crate has a standard library prelude, which consists of the names from a single standard library module. The module used depends on the crate's edition, and on whether the `no_std` attribute is applied to the crate:

Edition	`no_std` not applied	`no_std` applied
2015	`std::prelude::rust_2015`	`core::prelude::rust_2015`
2018	`std::prelude::rust_2018`	`core::prelude::rust_2018`
2021	`std::prelude::rust_2021`	`core::prelude::rust_2021`

Note:

`std::prelude::rust_2015` and `std::prelude::rust_2018` have the same contents as `std::prelude::v1`.

`core::prelude::rust_2015` and `core::prelude::rust_2018` have the same contents as `core::prelude::v1`.

Extern prelude

External crates imported with `extern crate` in the root module or provided to the compiler (as with the `--extern` flag with `rustc`) are added to the *extern prelude*. If imported with an alias such as `extern crate orig_name as new_name`, then the symbol `new_name` is instead added to the prelude.

The `core` crate is always added to the extern prelude. The `std` crate is added as long as the `no_std` attribute is not specified in the crate root.

> **Edition Differences**: In the 2015 edition, crates in the extern prelude cannot be referenced via use declarations, so it is generally standard practice to include `extern crate` declarations to bring them into scope.
>
> Beginning in the 2018 edition, use declarations can reference crates in the extern prelude, so it is considered unidiomatic to use `extern crate`.

> **Note**: Additional crates that ship with `rustc`, such as `alloc`, and `test`, are not automatically included with the `--extern` flag when using Cargo. They must be brought into scope with an `extern crate` declaration, even in the 2018 edition.
>
> ```
> extern crate alloc;
> use alloc::rc::Rc;
> ```
>
> Cargo does bring in `proc_macro` to the extern prelude for proc-macro crates only.

The `no_std` attribute

By default, the standard library is automatically included in the crate root module. The `std` crate is added to the root, along with an implicit `macro_use` attribute pulling in all macros exported from `std` into the `macro_use` prelude. Both `core` and `std` are added to the extern prelude.

The *`no_std`* *attribute* may be applied at the crate level to prevent the `std` crate from being automatically added into scope. It does three things:

- Prevents `std` from being added to the extern prelude.
- Affects which module is used to make up the standard library prelude (as described

above).
- Injects the `core` crate into the crate root instead of `std`, and pulls in all macros exported from `core` in the `macro_use` prelude.

Note: Using the core prelude over the standard prelude is useful when either the crate is targeting a platform that does not support the standard library or is purposefully not using the capabilities of the standard library. Those capabilities are mainly dynamic memory allocation (e.g. `Box` and `Vec`) and file and network capabilities (e.g. `std::fs` and `std::io`).

> ⚠️ Warning: Using `no_std` does not prevent the standard library from being linked in. It is still valid to put `extern crate std;` into the crate and dependencies can also link it in.

Language prelude

The language prelude includes names of types and attributes that are built-in to the language. The language prelude is always in scope. It includes the following:

- Type namespace
  - Boolean type — `bool`
  - Textual types — `char` and `str`
  - Integer types — `i8`, `i16`, `i32`, `i64`, `i128`, `u8`, `u16`, `u32`, `u64`, `u128`
  - Machine-dependent integer types — `usize` and `isize`
  - floating-point types — `f32` and `f64`
- Macro namespace
  - Built-in attributes

`macro_use` prelude

The `macro_use` prelude includes macros from external crates that were imported by the `macro_use` attribute applied to an `extern crate`.

Tool prelude

The tool prelude includes tool names for external tools in the type namespace. See the tool attributes section for more details.

The `no_implicit_prelude` attribute

The *`no_implicit_prelude`* *attribute* may be applied at the crate level or on a module to indicate that it should not automatically bring the standard library prelude, extern prelude, or tool prelude into scope for that module or any of its descendants.

This attribute does not affect the language prelude.

> **Edition Differences**: In the 2015 edition, the `no_implicit_prelude` attribute does not affect the `macro_use` prelude, and all macros exported from the standard library are still included in the `macro_use` prelude. Starting in the 2018 edition, it will remove the `macro_use` prelude.

Paths

A *path* is a sequence of one or more path segments *logically* separated by a namespace qualifier (`::`). If a path consists of only one segment, it refers to either an item or a variable in a local control scope. If a path has multiple segments, it always refers to an item.

Two examples of simple paths consisting of only identifier segments:

```
x;
x::y::z;
```

Types of paths

Simple Paths

Syntax

SimplePath :
`::` ? *SimplePathSegment* (`::` *SimplePathSegment*)*

SimplePathSegment :
IDENTIFIER | `super` | `self` | `crate` | `$crate`

Simple paths are used in visibility markers, attributes, macros, and `use` items. Examples:

```
use std::io::{self, Write};
mod m {
    #[clippy::cyclomatic_complexity = "0"]
    pub (in super) fn f1() {}
}
```

Paths in expressions

Syntax

PathInExpression :

`::`? *PathExprSegment* (`::` *PathExprSegment*)*

PathExprSegment :
PathIdentSegment (`::` *GenericArgs*)?

PathIdentSegment :
IDENTIFIER | `super` | `self` | `Self` | `crate` | `$crate`

GenericArgs :
`<` `>`
| `<` (*GenericArg* `,`)* *GenericArg* `,`? `>`

GenericArg :
Lifetime | *Type* | *GenericArgsConst* | *GenericArgsBinding*

GenericArgsConst :
BlockExpression
| *LiteralExpression*
| `-` *LiteralExpression*
| *SimplePathSegment*

GenericArgsBinding :
IDENTIFIER `=` *Type*

Paths in expressions allow for paths with generic arguments to be specified. They are used in various places in expressions and patterns.

The `::` token is required before the opening `<` for generic arguments to avoid ambiguity with the less-than operator. This is colloquially known as "turbofish" syntax.

```
(0..10).collect::<Vec<_>>();
Vec::<u8>::with_capacity(1024);
```

The order of generic arguments is restricted to lifetime arguments, then type arguments, then const arguments, then equality constraints.

Const arguments must be surrounded by braces unless they are a literal or a single segment path.

The synthetic type parameters corresponding to `impl Trait` types are implicit, and these cannot be explicitly specified.

Qualified paths

Syntax

QualifiedPathInExpression :
  QualifiedPathType (`::` *PathExprSegment*)+

QualifiedPathType :
  `<` *Type* (`as` *TypePath*)? `>`

QualifiedPathInType :
  QualifiedPathType (`::` *TypePathSegment*)+

Fully qualified paths allow for disambiguating the path for trait implementations and for specifying canonical paths. When used in a type specification, it supports using the type syntax specified below.

```
struct S;
impl S {
    fn f() { println!("S"); }
}
trait T1 {
    fn f() { println!("T1 f"); }
}
impl T1 for S {}
trait T2 {
    fn f() { println!("T2 f"); }
}
impl T2 for S {}
S::f();  // Calls the inherent impl.
<S as T1>::f();  // Calls the T1 trait function.
<S as T2>::f();  // Calls the T2 trait function.
```

Paths in types

Syntax

TypePath :
  `::` ? *TypePathSegment* (`::` *TypePathSegment*)*

TypePathSegment :
  PathIdentSegment (`::` ? (*GenericArgs* | *TypePathFn*))?

TypePathFn :
`(` *TypePathFnInputs*? `)` (`->` *TypeNoBounds*)?

TypePathFnInputs :
Type (`,` *Type*)* `,` ?

Type paths are used within type definitions, trait bounds, type parameter bounds, and qualified paths.

Although the `::` token is allowed before the generics arguments, it is not required because there is no ambiguity like there is in *PathInExpression*.

```
impl ops::Index<ops::Range<usize>> for S { /*...*/ }
fn i<'a>() -> impl Iterator<Item = ops::Example<'a>> {
    // ...
}
type G = std::boxed::Box<dyn std::ops::FnOnce(isize) -> isize>;
```

Path qualifiers

Paths can be denoted with various leading qualifiers to change the meaning of how it is resolved.

`::`

Paths starting with `::` are considered to be *global paths* where the segments of the path start being resolved from a place which differs based on edition. Each identifier in the path must resolve to an item.

> **Edition Differences**: In the 2015 Edition, identifiers resolve from the "crate root" (`crate::` in the 2018 edition), which contains a variety of different items, including external crates, default crates such as `std` or `core`, and items in the top level of the crate (including `use` imports).
>
> Beginning with the 2018 Edition, paths starting with `::` resolve from crates in the extern prelude. That is, they must be followed by the name of a crate.

```
pub fn foo() {
    // In the 2018 edition, this accesses `std` via the extern prelude.
    // In the 2015 edition, this accesses `std` via the crate root.
    let now = ::std::time::Instant::now();
    println!("{:?}", now);
}
```

```
// 2015 Edition
mod a {
    pub fn foo() {}
}
mod b {
    pub fn foo() {
        ::a::foo(); // call `a`'s foo function
        // In Rust 2018, `::a` would be interpreted as the crate `a`.
    }
}
```

`self`

`self` resolves the path relative to the current module. `self` can only be used as the first segment, without a preceding `::`.

In a method body, a path which consists of a single `self` segment resolves to the method's self parameter.

```
fn foo() {}
fn bar() {
    self::foo();
}
struct S(bool);
impl S {
  fn baz(self) {
        self.0;
    }
}
```

`Self`

`Self`, with a capital "S", is used to refer to the implementing type within traits and implementations.

`Self` can only be used as the first segment, without a preceding `::`.

```rust
trait T {
    type Item;
    const C: i32;
    // `Self` will be whatever type that implements `T`.
    fn new() -> Self;
    // `Self::Item` will be the type alias in the implementation.
    fn f(&self) -> Self::Item;
}
struct S;
impl T for S {
    type Item = i32;
    const C: i32 = 9;
    fn new() -> Self {           // `Self` is the type `S`.
        S
    }
    fn f(&self) -> Self::Item {  // `Self::Item` is the type `i32`.
        Self::C                  // `Self::C` is the constant value `9`.
    }
}
```

super

`super` in a path resolves to the parent module. It may only be used in leading segments of the path, possibly after an initial `self` segment.

```rust
mod a {
    pub fn foo() {}
}
mod b {
    pub fn foo() {
        super::a::foo(); // call a's foo function
    }
}
```

`super` may be repeated several times after the first `super` or `self` to refer to ancestor modules.

```
mod a {
    fn foo() {}

    mod b {
        mod c {
            fn foo() {
                super::super::foo(); // call a's foo function
                self::super::super::foo(); // call a's foo function
            }
        }
    }
}
```

`crate`

`crate` resolves the path relative to the current crate. `crate` can only be used as the first segment, without a preceding `::`.

```
fn foo() {}
mod a {
    fn bar() {
        crate::foo();
    }
}
```

`$crate`

`$crate` is only used within macro transcribers, and can only be used as the first segment, without a preceding `::`. `$crate` will expand to a path to access items from the top level of the crate where the macro is defined, regardless of which crate the macro is invoked.

```
pub fn increment(x: u32) -> u32 {
    x + 1
}

#[macro_export]
macro_rules! inc {
    ($x:expr) => ( $crate::increment($x) )
}
```

Canonical paths

Items defined in a module or implementation have a *canonical path* that corresponds to where within its crate it is defined. All other paths to these items are aliases. The canonical path is defined as a *path prefix* appended by the path segment the item itself defines.

Implementations and use declarations do not have canonical paths, although the items that implementations define do have them. Items defined in block expressions do not have canonical paths. Items defined in a module that does not have a canonical path do not have a canonical path. Associated items defined in an implementation that refers to an item without a canonical path, e.g. as the implementing type, the trait being implemented, a type parameter or bound on a type parameter, do not have canonical paths.

The path prefix for modules is the canonical path to that module. For bare implementations, it is the canonical path of the item being implemented surrounded by angle (`<>`) brackets. For trait implementations, it is the canonical path of the item being implemented followed by `as` followed by the canonical path to the trait all surrounded in angle (`<>`) brackets.

The canonical path is only meaningful within a given crate. There is no global namespace across crates; an item's canonical path merely identifies it within the crate.

```rust
// Comments show the canonical path of the item.

mod a { // crate::a
    pub struct Struct; // crate::a::Struct

    pub trait Trait { // crate::a::Trait
        fn f(&self); // crate::a::Trait::f
    }

    impl Trait for Struct {
        fn f(&self) {} // <crate::a::Struct as crate::a::Trait>::f
    }

    impl Struct {
        fn g(&self) {} // <crate::a::Struct>::g
    }
}

mod without { // crate::without
    fn canonicals() { // crate::without::canonicals
        struct OtherStruct; // None

        trait OtherTrait { // None
            fn g(&self); // None
        }

        impl OtherTrait for OtherStruct {
            fn g(&self) {} // None
        }

        impl OtherTrait for crate::a::Struct {
            fn g(&self) {} // None
        }

        impl crate::a::Trait for OtherStruct {
            fn f(&self) {} // None
        }
    }
}
```

Name resolution

> **Note**: This is a placeholder for future expansion.

Visibility and Privacy

> **Syntax**
>
> *Visibility* :
>
> `pub`
>
> | `pub` `(` `crate` `)`
>
> | `pub` `(` `self` `)`
>
> | `pub` `(` `super` `)`
>
> | `pub` `(` `in` *SimplePath* `)`

These two terms are often used interchangeably, and what they are attempting to convey is the answer to the question "Can this item be used at this location?"

Rust's name resolution operates on a global hierarchy of namespaces. Each level in the hierarchy can be thought of as some item. The items are one of those mentioned above, but also include external crates. Declaring or defining a new module can be thought of as inserting a new tree into the hierarchy at the location of the definition.

To control whether interfaces can be used across modules, Rust checks each use of an item to see whether it should be allowed or not. This is where privacy warnings are generated, or otherwise "you used a private item of another module and weren't allowed to."

By default, everything is *private*, with two exceptions: Associated items in a `pub` Trait are public by default; Enum variants in a `pub` enum are also public by default. When an item is declared as `pub`, it can be thought of as being accessible to the outside world. For example:

```
// Declare a private struct
struct Foo;

// Declare a public struct with a private field
pub struct Bar {
    field: i32,
}

// Declare a public enum with two public variants
pub enum State {
    PubliclyAccessibleState,
    PubliclyAccessibleState2,
}
```

With the notion of an item being either public or private, Rust allows item accesses in two cases:

1. If an item is public, then it can be accessed externally from some module `m` if you can access all the item's ancestor modules from `m`. You can also potentially be able to name the item through re-exports. See below.
2. If an item is private, it may be accessed by the current module and its descendants.

These two cases are surprisingly powerful for creating module hierarchies exposing public APIs while hiding internal implementation details. To help explain, here's a few use cases and what they would entail:

- A library developer needs to expose functionality to crates which link against their library. As a consequence of the first case, this means that anything which is usable externally must be `pub` from the root down to the destination item. Any private item in the chain will disallow external accesses.

- A crate needs a global available "helper module" to itself, but it doesn't want to expose the helper module as a public API. To accomplish this, the root of the crate's hierarchy would have a private module which then internally has a "public API". Because the entire crate is a descendant of the root, then the entire local crate can access this private module through the second case.

- When writing unit tests for a module, it's often a common idiom to have an immediate child of the module to-be-tested named `mod test`. This module could access any items of the parent module through the second case, meaning that internal implementation details could also be seamlessly tested from the child module.

In the second case, it mentions that a private item "can be accessed" by the current module and its descendants, but the exact meaning of accessing an item depends on what the item is. Accessing a module, for example, would mean looking inside of it (to import more items). On the other hand, accessing a function would mean that it is invoked. Additionally, path expressions and import statements are considered to access an item in the sense that the import/expression is only valid if the destination is in the current visibility scope.

Here's an example of a program which exemplifies the three cases outlined above:

```
// This module is private, meaning that no external crate can access this
// module. Because it is private at the root of this current crate, however,
any
// module in the crate may access any publicly visible item in this module.
mod crate_helper_module {

    // This function can be used by anything in the current crate
    pub fn crate_helper() {}

    // This function *cannot* be used by anything else in the crate. It is not
    // publicly visible outside of the `crate_helper_module`, so only this
    // current module and its descendants may access it.
    fn implementation_detail() {}
}

// This function is "public to the root" meaning that it's available to
external
// crates linking against this one.
pub fn public_api() {}

// Similarly to 'public_api', this module is public so external crates may look
// inside of it.
pub mod submodule {
    use crate::crate_helper_module;

    pub fn my_method() {
        // Any item in the local crate may invoke the helper module's public
        // interface through a combination of the two rules above.
        crate_helper_module::crate_helper();
    }

    // This function is hidden to any module which is not a descendant of
    // `submodule`
    fn my_implementation() {}

    #[cfg(test)]
    mod test {

        #[test]
        fn test_my_implementation() {
            // Because this module is a descendant of `submodule`, it's allowed
            // to access private items inside of `submodule` without a privacy
            // violation.
            super::my_implementation();
        }
    }
}
```

For a Rust program to pass the privacy checking pass, all paths must be valid accesses given the two rules above. This includes all use statements, expressions, types, etc.

`pub(in path)`, `pub(crate)`, `pub(super)`, and `pub(self)`

In addition to public and private, Rust allows users to declare an item as visible only within a given scope. The rules for `pub` restrictions are as follows:

- `pub(in path)` makes an item visible within the provided `path`. `path` must be an ancestor module of the item whose visibility is being declared.
- `pub(crate)` makes an item visible within the current crate.
- `pub(super)` makes an item visible to the parent module. This is equivalent to `pub(in super)`.
- `pub(self)` makes an item visible to the current module. This is equivalent to `pub(in self)` or not using `pub` at all.

> **Edition Differences**: Starting with the 2018 edition, paths for `pub(in path)` must start with `crate`, `self`, or `super`. The 2015 edition may also use paths starting with `::` or modules from the crate root.

Here's an example:

```rust
pub mod outer_mod {
    pub mod inner_mod {
        // This function is visible within `outer_mod`
        pub(in crate::outer_mod) fn outer_mod_visible_fn() {}
        // Same as above, this is only valid in the 2015 edition.
        pub(in outer_mod) fn outer_mod_visible_fn_2015() {}

        // This function is visible to the entire crate
        pub(crate) fn crate_visible_fn() {}

        // This function is visible within `outer_mod`
        pub(super) fn super_mod_visible_fn() {
            // This function is visible since we're in the same `mod`
            inner_mod_visible_fn();
        }

        // This function is visible only within `inner_mod`,
        // which is the same as leaving it private.
        pub(self) fn inner_mod_visible_fn() {}
    }
    pub fn foo() {
        inner_mod::outer_mod_visible_fn();
        inner_mod::crate_visible_fn();
        inner_mod::super_mod_visible_fn();

        // This function is no longer visible since we're outside of
`inner_mod`
        // Error! `inner_mod_visible_fn` is private
        //inner_mod::inner_mod_visible_fn();
    }
}

fn bar() {
    // This function is still visible since we're in the same crate
    outer_mod::inner_mod::crate_visible_fn();

    // This function is no longer visible since we're outside of `outer_mod`
    // Error! `super_mod_visible_fn` is private
    //outer_mod::inner_mod::super_mod_visible_fn();

    // This function is no longer visible since we're outside of `outer_mod`
    // Error! `outer_mod_visible_fn` is private
    //outer_mod::inner_mod::outer_mod_visible_fn();

    outer_mod::foo();
}

fn main() { bar() }
```

Note: This syntax only adds another restriction to the visibility of an item. It does not guarantee that the item is visible within all parts of the specified scope. To access an item, all of its parent items up to the current scope must still be visible as well.

Re-exporting and Visibility

Rust allows publicly re-exporting items through a `pub use` directive. Because this is a public directive, this allows the item to be used in the current module through the rules above. It essentially allows public access into the re-exported item. For example, this program is valid:

```
pub use self::implementation::api;

mod implementation {
    pub mod api {
        pub fn f() {}
    }
}
```

This means that any external crate referencing `implementation::api::f` would receive a privacy violation, while the path `api::f` would be allowed.

When re-exporting a private item, it can be thought of as allowing the "privacy chain" being short-circuited through the reexport instead of passing through the namespace hierarchy as it normally would.

Memory model

Rust does not yet have a defined memory model. Various academics and industry professionals are working on various proposals, but for now, this is an under-defined place in the language.

Memory allocation and lifetime

The *items* of a program are those functions, modules, and types that have their value calculated at compile-time and stored uniquely in the memory image of the rust process. Items are neither dynamically allocated nor freed.

The *heap* is a general term that describes boxes. The lifetime of an allocation in the heap depends on the lifetime of the box values pointing to it. Since box values may themselves be passed in and out of frames, or stored in the heap, heap allocations may outlive the frame they are allocated within. An allocation in the heap is guaranteed to reside at a single location in the heap for the whole lifetime of the allocation - it will never be relocated as a result of moving a box value.

Variables

A *variable* is a component of a stack frame, either a named function parameter, an anonymous temporary, or a named local variable.

A *local variable* (or *stack-local* allocation) holds a value directly, allocated within the stack's memory. The value is a part of the stack frame.

Local variables are immutable unless declared otherwise. For example: `let mut x = ...`.

Function parameters are immutable unless declared with `mut`. The `mut` keyword applies only to the following parameter. For example: `|mut x, y|` and `fn f(mut x: Box<i32>, y: Box<i32>)` declare one mutable variable `x` and one immutable variable `y`.

Local variables are not initialized when allocated. Instead, the entire frame worth of local variables are allocated, on frame-entry, in an uninitialized state. Subsequent statements within a function may or may not initialize the local variables. Local variables can be used only after they have been initialized through all reachable control flow paths.

In this next example, `init_after_if` is initialized after the `if` expression while `uninit_after_if` is not because it is not initialized in the `else` case.

```
fn initialization_example() {
    let init_after_if: ();
    let uninit_after_if: ();

    if random_bool() {
        init_after_if = ();
        uninit_after_if = ();
    } else {
        init_after_if = ();
    }

    init_after_if; // ok
    // uninit_after_if; // err: use of possibly uninitialized `uninit_after_if`
}
```

Linkage

> Note: This section is described more in terms of the compiler than of the language.

The compiler supports various methods to link crates together both statically and dynamically. This section will explore the various methods to link crates together, and more information about native libraries can be found in the FFI section of the book.

In one session of compilation, the compiler can generate multiple artifacts through the usage of either command line flags or the `crate_type` attribute. If one or more command line flags are specified, all `crate_type` attributes will be ignored in favor of only building the artifacts specified by command line.

- `--crate-type=bin`, `#![crate_type = "bin"]` - A runnable executable will be produced. This requires that there is a `main` function in the crate which will be run when the program begins executing. This will link in all Rust and native dependencies, producing a single distributable binary. This is the default crate type.

- `--crate-type=lib`, `#![crate_type = "lib"]` - A Rust library will be produced. This is an ambiguous concept as to what exactly is produced because a library can manifest itself in several forms. The purpose of this generic `lib` option is to generate the "compiler recommended" style of library. The output library will always be usable by rustc, but the actual type of library may change from time-to-time. The remaining output types are all different flavors of libraries, and the `lib` type can be seen as an alias for one of them (but the actual one is compiler-defined).

- `--crate-type=dylib`, `#![crate_type = "dylib"]` - A dynamic Rust library will be produced. This is different from the `lib` output type in that this forces dynamic library generation. The resulting dynamic library can be used as a dependency for other libraries and/or executables. This output type will create `*.so` files on Linux, `*.dylib` files on macOS, and `*.dll` files on Windows.

- `--crate-type=staticlib`, `#![crate_type = "staticlib"]` - A static system library will be produced. This is different from other library outputs in that the compiler will never attempt to link to `staticlib` outputs. The purpose of this output type is to create a static library containing all of the local crate's code along with all upstream dependencies. This output type will create `*.a` files on Linux, macOS and Windows (MinGW), and `*.lib` files on Windows (MSVC). This format is recommended for use in situations such as linking Rust code into an existing non-Rust application because it will not have dynamic dependencies on other Rust code.

- `--crate-type=cdylib`, `#![crate_type = "cdylib"]` - A dynamic system library will be produced. This is used when compiling a dynamic library to be loaded from another language. This output type will create `*.so` files on Linux, `*.dylib` files on macOS, and `*.dll` files on Windows.

- `--crate-type=rlib`, `#![crate_type = "rlib"]` - A "Rust library" file will be produced. This is used as an intermediate artifact and can be thought of as a "static Rust library". These `rlib` files, unlike `staticlib` files, are interpreted by the compiler in future linkage. This essentially means that `rustc` will look for metadata in `rlib` files like it looks for metadata in dynamic libraries. This form of output is used to produce statically linked executables as well as `staticlib` outputs.

- `--crate-type=proc-macro`, `#![crate_type = "proc-macro"]` - The output produced is not specified, but if a `-L` path is provided to it then the compiler will recognize the output artifacts as a macro and it can be loaded for a program. Crates compiled with this crate type must only export procedural macros. The compiler will automatically set the `proc_macro` configuration option. The crates are always compiled with the same target that the compiler itself was built with. For example, if you are executing the compiler from Linux with an `x86_64` CPU, the target will be `x86_64-unknown-linux-gnu` even if the crate is a dependency of another crate being built for a different target.

Note that these outputs are stackable in the sense that if multiple are specified, then the compiler will produce each form of output without having to recompile. However, this only applies for outputs specified by the same method. If only `crate_type` attributes are specified, then they will all be built, but if one or more `--crate-type` command line flags are specified, then only those outputs will be built.

With all these different kinds of outputs, if crate A depends on crate B, then the compiler could find B in various different forms throughout the system. The only forms looked for by the compiler, however, are the `rlib` format and the dynamic library format. With these two options for a dependent library, the compiler must at some point make a choice between these two formats. With this in mind, the compiler follows these rules when determining what format of dependencies will be used:

1. If a static library is being produced, all upstream dependencies are required to be available in `rlib` formats. This requirement stems from the reason that a dynamic library cannot be converted into a static format.

   Note that it is impossible to link in native dynamic dependencies to a static library, and in this case warnings will be printed about all unlinked native dynamic dependencies.

2. If an `rlib` file is being produced, then there are no restrictions on what format the

upstream dependencies are available in. It is simply required that all upstream dependencies be available for reading metadata from.

The reason for this is that `rlib` files do not contain any of their upstream dependencies. It wouldn't be very efficient for all `rlib` files to contain a copy of `libstd.rlib`!

3. If an executable is being produced and the `-C prefer-dynamic` flag is not specified, then dependencies are first attempted to be found in the `rlib` format. If some dependencies are not available in an rlib format, then dynamic linking is attempted (see below).

4. If a dynamic library or an executable that is being dynamically linked is being produced, then the compiler will attempt to reconcile the available dependencies in either the rlib or dylib format to create a final product.

   A major goal of the compiler is to ensure that a library never appears more than once in any artifact. For example, if dynamic libraries B and C were each statically linked to library A, then a crate could not link to B and C together because there would be two copies of A. The compiler allows mixing the rlib and dylib formats, but this restriction must be satisfied.

   The compiler currently implements no method of hinting what format a library should be linked with. When dynamically linking, the compiler will attempt to maximize dynamic dependencies while still allowing some dependencies to be linked in via an rlib.

   For most situations, having all libraries available as a dylib is recommended if dynamically linking. For other situations, the compiler will emit a warning if it is unable to determine which formats to link each library with.

In general, `--crate-type=bin` or `--crate-type=lib` should be sufficient for all compilation needs, and the other options are just available if more fine-grained control is desired over the output format of a crate.

Static and dynamic C runtimes

The standard library in general strives to support both statically linked and dynamically linked C runtimes for targets as appropriate. For example the `x86_64-pc-windows-msvc` and `x86_64-unknown-linux-musl` targets typically come with both runtimes and the user selects which one they'd like. All targets in the compiler have a default mode of linking to the C runtime. Typically targets are linked dynamically by default, but there are exceptions which are static by default such as:

- `arm-unknown-linux-musleabi`
- `arm-unknown-linux-musleabihf`
- `armv7-unknown-linux-musleabihf`
- `i686-unknown-linux-musl`
- `x86_64-unknown-linux-musl`

The linkage of the C runtime is configured to respect the `crt-static` target feature. These target features are typically configured from the command line via flags to the compiler itself. For example to enable a static runtime you would execute:

```
rustc -C target-feature=+crt-static foo.rs
```

whereas to link dynamically to the C runtime you would execute:

```
rustc -C target-feature=-crt-static foo.rs
```

Targets which do not support switching between linkage of the C runtime will ignore this flag. It's recommended to inspect the resulting binary to ensure that it's linked as you would expect after the compiler succeeds.

Crates may also learn about how the C runtime is being linked. Code on MSVC, for example, needs to be compiled differently (e.g. with `/MT` or `/MD`) depending on the runtime being linked. This is exported currently through the `cfg` attribute `target_feature` option:

```
#[cfg(target_feature = "crt-static")]
fn foo() {
    println!("the C runtime should be statically linked");
}

#[cfg(not(target_feature = "crt-static"))]
fn foo() {
    println!("the C runtime should be dynamically linked");
}
```

Also note that Cargo build scripts can learn about this feature through environment variables. In a build script you can detect the linkage via:

```
use std::env;

fn main() {
    let linkage =
env::var("CARGO_CFG_TARGET_FEATURE").unwrap_or(String::new());

    if linkage.contains("crt-static") {
        println!("the C runtime will be statically linked");
    } else {
        println!("the C runtime will be dynamically linked");
    }
}
```

To use this feature locally, you typically will use the `RUSTFLAGS` environment variable to specify flags to the compiler through Cargo. For example to compile a statically linked binary on MSVC you would execute:

```
RUSTFLAGS='-C target-feature=+crt-static' cargo build --target x86_64-pc-
windows-msvc
```

Inline assembly

Support for inline assembly is provided via the `asm!` and `global_asm!` macros. It can be used to embed handwritten assembly in the assembly output generated by the compiler.

Support for inline assembly is stable on the following architectures:

- x86 and x86-64
- ARM
- AArch64
- RISC-V
- LoongArch

The compiler will emit an error if `asm!` is used on an unsupported target.

Example

```
use std::arch::asm;

// Multiply x by 6 using shifts and adds
let mut x: u64 = 4;
unsafe {
    asm!(
        "mov {tmp}, {x}",
        "shl {tmp}, 1",
        "shl {x}, 2",
        "add {x}, {tmp}",
        x = inout(reg) x,
        tmp = out(reg) _,
    );
}
assert_eq!(x, 4 * 6);
```

Syntax

The following ABNF specifies the general syntax:

```
format_string := STRING_LITERAL / RAW_STRING_LITERAL
dir_spec := "in" / "out" / "lateout" / "inout" / "inlateout"
reg_spec := <register class> / "\"" <explicit register> "\""
operand_expr := expr / "_" / expr "=>" expr / expr "=>" "_"
reg_operand := [ident "="] dir_spec "(" reg_spec ")" operand_expr
clobber_abi := "clobber_abi(" <abi> *("," <abi>) [","] ")"
option := "pure" / "nomem" / "readonly" / "preserves_flags" / "noreturn" /
"nostack" / "att_syntax" / "raw"
options := "options(" option *("," option) [","] ")"
operand := reg_operand / clobber_abi / options
asm := "asm!(" format_string *("," format_string) *("," operand) [","] ")"
global_asm := "global_asm!(" format_string *("," format_string) *("," operand)
[","] ")"
```

Scope

Inline assembly can be used in one of two ways.

With the `asm!` macro, the assembly code is emitted in a function scope and integrated into the compiler-generated assembly code of a function. This assembly code must obey strict rules to avoid undefined behavior. Note that in some cases the compiler may choose to emit the assembly code as a separate function and generate a call to it.

With the `global_asm!` macro, the assembly code is emitted in a global scope, outside a function. This can be used to hand-write entire functions using assembly code, and generally provides much more freedom to use arbitrary registers and assembler directives.

Template string arguments

The assembler template uses the same syntax as format strings (i.e. placeholders are specified by curly braces). The corresponding arguments are accessed in order, by index, or by name. However, implicit named arguments (introduced by RFC #2795) are not supported.

An `asm!` invocation may have one or more template string arguments; an `asm!` with multiple template string arguments is treated as if all the strings were concatenated with a `\n` between them. The expected usage is for each template string argument to correspond to a line of assembly code. All template string arguments must appear before any other arguments.

As with format strings, positional arguments must appear before named arguments and explicit register operands.

Explicit register operands cannot be used by placeholders in the template string. All other named and positional operands must appear at least once in the template string, otherwise a compiler error is generated.

The exact assembly code syntax is target-specific and opaque to the compiler except for the way operands are substituted into the template string to form the code passed to the assembler.

Currently, all supported targets follow the assembly code syntax used by LLVM's internal assembler which usually corresponds to that of the GNU assembler (GAS). On x86, the `.intel_syntax noprefix` mode of GAS is used by default. On ARM, the `.syntax unified` mode is used. These targets impose an additional restriction on the assembly code: any assembler state (e.g. the current section which can be changed with `.section`) must be restored to its original value at the end of the asm string. Assembly code that does not conform to the GAS syntax will result in assembler-specific behavior. Further constraints on the directives used by inline assembly are indicated by Directives Support.

Operand type

Several types of operands are supported:

- `in(<reg>) <expr>`
    - `<reg>` can refer to a register class or an explicit register. The allocated register name is substituted into the asm template string.
    - The allocated register will contain the value of `<expr>` at the start of the asm code.
    - The allocated register must contain the same value at the end of the asm code (except if a `lateout` is allocated to the same register).
- `out(<reg>) <expr>`
    - `<reg>` can refer to a register class or an explicit register. The allocated register name is substituted into the asm template string.
    - The allocated register will contain an undefined value at the start of the asm code.
    - `<expr>` must be a (possibly uninitialized) place expression, to which the contents of the allocated register are written at the end of the asm code.
    - An underscore (`_`) may be specified instead of an expression, which will cause the contents of the register to be discarded at the end of the asm code (effectively acting as a clobber).
- `lateout(<reg>) <expr>`

    - Identical to `out` except that the register allocator can reuse a register allocated to an `in`.
    - You should only write to the register after all inputs are read, otherwise you may clobber an input.
- `inout(<reg>) <expr>`
    - `<reg>` can refer to a register class or an explicit register. The allocated register name is substituted into the asm template string.
    - The allocated register will contain the value of `<expr>` at the start of the asm code.
    - `<expr>` must be a mutable initialized place expression, to which the contents of the allocated register are written at the end of the asm code.
- `inout(<reg>) <in expr> => <out expr>`
    - Same as `inout` except that the initial value of the register is taken from the value of `<in expr>`.
    - `<out expr>` must be a (possibly uninitialized) place expression, to which the contents of the allocated register are written at the end of the asm code.
    - An underscore (`_`) may be specified instead of an expression for `<out expr>`, which will cause the contents of the register to be discarded at the end of the asm code (effectively acting as a clobber).
    - `<in expr>` and `<out expr>` may have different types.
- `inlateout(<reg>) <expr>` / `inlateout(<reg>) <in expr> => <out expr>`
    - Identical to `inout` except that the register allocator can reuse a register allocated to an `in` (this can happen if the compiler knows the `in` has the same initial value as the `inlateout`).
    - You should only write to the register after all inputs are read, otherwise you may clobber an input.
- `sym <path>`
    - `<path>` must refer to a `fn` or `static`.
    - A mangled symbol name referring to the item is substituted into the asm template string.
    - The substituted string does not include any modifiers (e.g. GOT, PLT, relocations, etc).
    - `<path>` is allowed to point to a `#[thread_local]` static, in which case the asm code can combine the symbol with relocations (e.g. `@plt`, `@TPOFF`) to read from thread-local data.

Operand expressions are evaluated from left to right, just like function call arguments. After the `asm!` has executed, outputs are written to in left to right order. This is significant if two outputs point to the same place: that place will contain the value of the rightmost output.

Since `global_asm!` exists outside a function, it can only use `sym` operands.

Register operands

Input and output operands can be specified either as an explicit register or as a register class from which the register allocator can select a register. Explicit registers are specified as string literals (e.g. `"eax"`) while register classes are specified as identifiers (e.g. `reg`).

Note that explicit registers treat register aliases (e.g. `r14` vs `lr` on ARM) and smaller views of a register (e.g. `eax` vs `rax`) as equivalent to the base register. It is a compile-time error to use the same explicit register for two input operands or two output operands. Additionally, it is also a compile-time error to use overlapping registers (e.g. ARM VFP) in input operands or in output operands.

Only the following types are allowed as operands for inline assembly:

- Integers (signed and unsigned)
- Floating-point numbers
- Pointers (thin only)
- Function pointers
- SIMD vectors (structs defined with `#[repr(simd)]` and which implement `Copy`). This includes architecture-specific vector types defined in `std::arch` such as `__m128` (x86) or `int8x16_t` (ARM).

Here is the list of currently supported register classes:

Architecture	Register class	Registers	LLVM constraint code
x86	`reg`	`ax` , `bx` , `cx` , `dx` , `si` , `di` , `bp` , `r[8-15]` (x86-64 only)	`r`
x86	`reg_abcd`	`ax` , `bx` , `cx` , `dx`	`Q`
x86-32	`reg_byte`	`al` , `bl` , `cl` , `dl` , `ah` , `bh` , `ch` , `dh`	`q`
x86-64	`reg_byte` *	`al` , `bl` , `cl` , `dl` , `sil` , `dil` , `bpl` , `r[8-15]b`	`q`
x86	`xmm_reg`	`xmm[0-7]` (x86) `xmm[0-15]` (x86-64)	`x`

Architecture	Register class	Registers	LLVM constraint code
x86	`ymm_reg`	`ymm[0-7]` (x86) `ymm[0-15]` (x86-64)	`x`
x86	`zmm_reg`	`zmm[0-7]` (x86) `zmm[0-31]` (x86-64)	`v`
x86	`kreg`	`k[1-7]`	`Yk`
x86	`kreg0`	`k0`	Only clobbers
x86	`x87_reg`	`st([0-7])`	Only clobbers
x86	`mmx_reg`	`mm[0-7]`	Only clobbers
x86-64	`tmm_reg`	`tmm[0-7]`	Only clobbers
AArch64	`reg`	`x[0-30]`	`r`
AArch64	`vreg`	`v[0-31]`	`w`
AArch64	`vreg_low16`	`v[0-15]`	`x`
AArch64	`preg`	`p[0-15]`, `ffr`	Only clobbers
ARM (ARM/Thumb2)	`reg`	`r[0-12]`, `r14`	`r`
ARM (Thumb1)	`reg`	`r[0-7]`	`r`
ARM	`sreg`	`s[0-31]`	`t`
ARM	`sreg_low16`	`s[0-15]`	`x`
ARM	`dreg`	`d[0-31]`	`w`
ARM	`dreg_low16`	`d[0-15]`	`t`
ARM	`dreg_low8`	`d[0-8]`	`x`
ARM	`qreg`	`q[0-15]`	`w`
ARM	`qreg_low8`	`q[0-7]`	`t`
ARM	`qreg_low4`	`q[0-3]`	`x`
RISC-V	`reg`	`x1`, `x[5-7]`, `x[9-15]`, `x[16-31]` (non-RV32E)	`r`
RISC-V	`freg`	`f[0-31]`	`f`
RISC-V	`vreg`	`v[0-31]`	Only clobbers
LoongArch	`reg`	`$r1`, `$r[4-20]`, `$r[23,30]`	`r`
LoongArch	`freg`	`$f[0-31]`	`f`

Notes:

- On x86 we treat `reg_byte` differently from `reg` because the compiler can allocate `al` and `ah` separately whereas `reg` reserves the whole register.
- On x86-64 the high byte registers (e.g. `ah`) are not available in the `reg_byte` register class.
- Some register classes are marked as "Only clobbers" which means that registers in these classes cannot be used for inputs or outputs, only clobbers of the form `out(<explicit register>) _` or `lateout(<explicit register>) _`.

Each register class has constraints on which value types they can be used with. This is necessary because the way a value is loaded into a register depends on its type. For example, on big-endian systems, loading a `i32x4` and a `i8x16` into a SIMD register may result in different register contents even if the byte-wise memory representation of both values is identical. The availability of supported types for a particular register class may depend on what target features are currently enabled.

Architecture	Register class	Target feature	Allowed types
x86-32	`reg`	None	`i16`, `i32`, `f32`
x86-64	`reg`	None	`i16`, `i32`, `f32`, `i64`, `f64`
x86	`reg_byte`	None	`i8`
x86	`xmm_reg`	`sse`	`i32`, `f32`, `i64`, `f64`, `i8x16`, `i16x8`, `i32x4`, `i64x2`, `f32x4`, `f64x2`
x86	`ymm_reg`	`avx`	`i32`, `f32`, `i64`, `f64`, `i8x16`, `i16x8`, `i32x4`, `i64x2`, `f32x4`, `f64x2` `i8x32`, `i16x16`, `i32x8`, `i64x4`, `f32x8`, `f64x4`
x86	`zmm_reg`	`avx512f`	`i32`, `f32`, `i64`, `f64`, `i8x16`, `i16x8`, `i32x4`, `i64x2`, `f32x4`, `f64x2` `i8x32`, `i16x16`, `i32x8`, `i64x4`, `f32x8`, `f64x4` `i8x64`, `i16x32`, `i32x16`, `i64x8`, `f32x16`, `f64x8`

Architecture	Register class	Target feature	Allowed types
x86	`kreg`	`avx512f`	`i8`, `i16`
x86	`kreg`	`avx512bw`	`i32`, `i64`
x86	`mmx_reg`	N/A	Only clobbers
x86	`x87_reg`	N/A	Only clobbers
x86	`tmm_reg`	N/A	Only clobbers
AArch64	`reg`	None	`i8`, `i16`, `i32`, `f32`, `i64`, `f64`
AArch64	`vreg`	`neon`	`i8`, `i16`, `i32`, `f32`, `i64`, `f64`, `i8x8`, `i16x4`, `i32x2`, `i64x1`, `f32x2`, `f64x1`, `i8x16`, `i16x8`, `i32x4`, `i64x2`, `f32x4`, `f64x2`
AArch64	`preg`	N/A	Only clobbers
ARM	`reg`	None	`i8`, `i16`, `i32`, `f32`
ARM	`sreg`	`vfp2`	`i32`, `f32`
ARM	`dreg`	`vfp2`	`i64`, `f64`, `i8x8`, `i16x4`, `i32x2`, `i64x1`, `f32x2`
ARM	`qreg`	`neon`	`i8x16`, `i16x8`, `i32x4`, `i64x2`, `f32x4`
RISC-V32	`reg`	None	`i8`, `i16`, `i32`, `f32`
RISC-V64	`reg`	None	`i8`, `i16`, `i32`, `f32`, `i64`, `f64`
RISC-V	`freg`	`f`	`f32`
RISC-V	`freg`	`d`	`f64`
RISC-V	`vreg`	N/A	Only clobbers
LoongArch64	`reg`	None	`i8`, `i16`, `i32`, `i64`, `f32`, `f64`
LoongArch64	`freg`	None	`f32`, `f64`

> **Note**: For the purposes of the above table pointers, function pointers and `isize` / `usize` are treated as the equivalent integer type (`i16` / `i32` / `i64` depending on the target).

If a value is of a smaller size than the register it is allocated in then the upper bits of that register will have an undefined value for inputs and will be ignored for outputs. The only exception is the `freg` register class on RISC-V where `f32` values are NaN-boxed in a `f64` as required by the RISC-V architecture.

When separate input and output expressions are specified for an `inout` operand, both expressions must have the same type. The only exception is if both operands are pointers or integers, in which case they are only required to have the same size. This restriction exists because the register allocators in LLVM and GCC sometimes cannot handle tied operands with different types.

Register names

Some registers have multiple names. These are all treated by the compiler as identical to the base register name. Here is the list of all supported register aliases:

Architecture	Base register	Aliases
x86	`ax`	`eax`, `rax`
x86	`bx`	`ebx`, `rbx`
x86	`cx`	`ecx`, `rcx`
x86	`dx`	`edx`, `rdx`
x86	`si`	`esi`, `rsi`
x86	`di`	`edi`, `rdi`
x86	`bp`	`bpl`, `ebp`, `rbp`
x86	`sp`	`spl`, `esp`, `rsp`
x86	`ip`	`eip`, `rip`
x86	`st(0)`	`st`
x86	`r[8-15]`	`r[8-15]b`, `r[8-15]w`, `r[8-15]d`
x86	`xmm[0-31]`	`ymm[0-31]`, `zmm[0-31]`
AArch64	`x[0-30]`	`w[0-30]`
AArch64	`x29`	`fp`
AArch64	`x30`	`lr`
AArch64	`sp`	`wsp`
AArch64	`xzr`	`wzr`
AArch64	`v[0-31]`	`b[0-31]`, `h[0-31]`, `s[0-31]`, `d[0-31]`, `q[0-31]`

Architecture	Base register	Aliases
ARM	`r[0-3]`	`a[1-4]`
ARM	`r[4-9]`	`v[1-6]`
ARM	`r9`	`rfp`
ARM	`r10`	`sl`
ARM	`r11`	`fp`
ARM	`r12`	`ip`
ARM	`r13`	`sp`
ARM	`r14`	`lr`
ARM	`r15`	`pc`
RISC-V	`x0`	`zero`
RISC-V	`x1`	`ra`
RISC-V	`x2`	`sp`
RISC-V	`x3`	`gp`
RISC-V	`x4`	`tp`
RISC-V	`x[5-7]`	`t[0-2]`
RISC-V	`x8`	`fp`, `s0`
RISC-V	`x9`	`s1`
RISC-V	`x[10-17]`	`a[0-7]`
RISC-V	`x[18-27]`	`s[2-11]`
RISC-V	`x[28-31]`	`t[3-6]`
RISC-V	`f[0-7]`	`ft[0-7]`
RISC-V	`f[8-9]`	`fs[0-1]`
RISC-V	`f[10-17]`	`fa[0-7]`
RISC-V	`f[18-27]`	`fs[2-11]`
RISC-V	`f[28-31]`	`ft[8-11]`
LoongArch	`$r0`	`$zero`
LoongArch	`$r1`	`$ra`
LoongArch	`$r2`	`$tp`
LoongArch	`$r3`	`$sp`
LoongArch	`$r[4-11]`	`$a[0-7]`

Architecture	Base register	Aliases
LoongArch	`$r[12-20]`	`$t[0-8]`
LoongArch	`$r21`	
LoongArch	`$r22`	`$fp` , `$s9`
LoongArch	`$r[23-31]`	`$s[0-8]`
LoongArch	`$f[0-7]`	`$fa[0-7]`
LoongArch	`$f[8-23]`	`$ft[0-15]`
LoongArch	`$f[24-31]`	`$fs[0-7]`

Some registers cannot be used for input or output operands:

Architecture	Unsupported register	Reason
All	`sp`	The stack pointer must be restored to its original value at the end of an asm code block.
All	`bp` (x86), `x29` (AArch64), `x8` (RISC-V), `$fp` (LoongArch)	The frame pointer cannot be used as an input or output.
ARM	`r7` or `r11`	On ARM the frame pointer can be either `r7` or `r11` depending on the target. The frame pointer cannot be used as an input or output.
All	`si` (x86-32), `bx` (x86-64), `r6` (ARM), `x19` (AArch64), `x9` (RISC-V), `$s8` (LoongArch)	This is used internally by LLVM as a "base pointer" for functions with complex stack frames.
x86	`ip`	This is the program counter, not a real register.
AArch64	`xzr`	This is a constant zero register which can't be modified.
AArch64	`x18`	This is an OS-reserved register on some AArch64 targets.
ARM	`pc`	This is the program counter, not a real register.

Architecture	Unsupported register	Reason
ARM	`r9`	This is an OS-reserved register on some ARM targets.
RISC-V	`x0`	This is a constant zero register which can't be modified.
RISC-V	`gp`, `tp`	These registers are reserved and cannot be used as inputs or outputs.
LoongArch	`$r0` or `$zero`	This is a constant zero register which can't be modified.
LoongArch	`$r2` or `$tp`	This is reserved for TLS.
LoongArch	`$r21`	This is reserved by the ABI.

The frame pointer and base pointer registers are reserved for internal use by LLVM. While `asm!` statements cannot explicitly specify the use of reserved registers, in some cases LLVM will allocate one of these reserved registers for `reg` operands. Assembly code making use of reserved registers should be careful since `reg` operands may use the same registers.

Template modifiers

The placeholders can be augmented by modifiers which are specified after the `:` in the curly braces. These modifiers do not affect register allocation, but change the way operands are formatted when inserted into the template string. Only one modifier is allowed per template placeholder.

The supported modifiers are a subset of LLVM's (and GCC's) asm template argument modifiers, but do not use the same letter codes.

Architecture	Register class	Modifier	Example output	LLVM modifier
x86-32	`reg`	None	`eax`	`k`
x86-64	`reg`	None	`rax`	`q`
x86-32	`reg_abcd`	`l`	`al`	`b`
x86-64	`reg`	`l`	`al`	`b`
x86	`reg_abcd`	`h`	`ah`	`h`
x86	`reg`	`x`	`ax`	`w`

Architecture	Register class	Modifier	Example output	LLVM modifier
x86	`reg`	`e`	`eax`	`k`
x86-64	`reg`	`r`	`rax`	`q`
x86	`reg_byte`	None	`al` / `ah`	None
x86	`xmm_reg`	None	`xmm0`	`x`
x86	`ymm_reg`	None	`ymm0`	`t`
x86	`zmm_reg`	None	`zmm0`	`g`
x86	`*mm_reg`	`x`	`xmm0`	`x`
x86	`*mm_reg`	`y`	`ymm0`	`t`
x86	`*mm_reg`	`z`	`zmm0`	`g`
x86	`kreg`	None	`k1`	None
AArch64	`reg`	None	`x0`	`x`
AArch64	`reg`	`w`	`w0`	`w`
AArch64	`reg`	`x`	`x0`	`x`
AArch64	`vreg`	None	`v0`	None
AArch64	`vreg`	`v`	`v0`	None
AArch64	`vreg`	`b`	`b0`	`b`
AArch64	`vreg`	`h`	`h0`	`h`
AArch64	`vreg`	`s`	`s0`	`s`
AArch64	`vreg`	`d`	`d0`	`d`
AArch64	`vreg`	`q`	`q0`	`q`
ARM	`reg`	None	`r0`	None
ARM	`sreg`	None	`s0`	None
ARM	`dreg`	None	`d0`	`P`
ARM	`qreg`	None	`q0`	`q`
ARM	`qreg`	`e` / `f`	`d0` / `d1`	`e` / `f`
RISC-V	`reg`	None	`x1`	None
RISC-V	`freg`	None	`f0`	None
LoongArch	`reg`	None	`$r1`	None
LoongArch	`freg`	None	`$f0`	None

Notes:

- on ARM `e` / `f`: this prints the low or high doubleword register name of a NEON quad (128-bit) register.
- on x86: our behavior for `reg` with no modifiers differs from what GCC does. GCC will infer the modifier based on the operand value type, while we default to the full register size.
- on x86 `xmm_reg`: the `x`, `t` and `g` LLVM modifiers are not yet implemented in LLVM (they are supported by GCC only), but this should be a simple change.

As stated in the previous section, passing an input value smaller than the register width will result in the upper bits of the register containing undefined values. This is not a problem if the inline asm only accesses the lower bits of the register, which can be done by using a template modifier to use a subregister name in the asm code (e.g. `ax` instead of `rax`). Since this an easy pitfall, the compiler will suggest a template modifier to use where appropriate given the input type. If all references to an operand already have modifiers then the warning is suppressed for that operand.

ABI clobbers

The `clobber_abi` keyword can be used to apply a default set of clobbers to an `asm!` block. This will automatically insert the necessary clobber constraints as needed for calling a function with a particular calling convention: if the calling convention does not fully preserve the value of a register across a call then `lateout("...") _` is implicitly added to the operands list (where the `...` is replaced by the register's name).

`clobber_abi` may be specified any number of times. It will insert a clobber for all unique registers in the union of all specified calling conventions.

Generic register class outputs are disallowed by the compiler when `clobber_abi` is used: all outputs must specify an explicit register. Explicit register outputs have precedence over the implicit clobbers inserted by `clobber_abi`: a clobber will only be inserted for a register if that register is not used as an output. The following ABIs can be used with `clobber_abi`:

Architecture	ABI name	Clobbered registers
x86-32	`"C"`, `"system"`, `"efiapi"`, `"cdecl"`, `"stdcall"`, `"fastcall"`	`ax`, `cx`, `dx`, `xmm[0-7]`, `mm[0-7]`, `k[0-7]`, `st([0-7])`

Architecture	ABI name	Clobbered registers
x86-64	`"C"`, `"system"` (on Windows), `"efiapi"`, `"win64"`	`ax`, `cx`, `dx`, `r[8-11]`, `xmm[0-31]`, `mm[0-7]`, `k[0-7]`, `st([0-7])`, `tmm[0-7]`
x86-64	`"C"`, `"system"` (on non-Windows), `"sysv64"`	`ax`, `cx`, `dx`, `si`, `di`, `r[8-11]`, `xmm[0-31]`, `mm[0-7]`, `k[0-7]`, `st([0-7])`, `tmm[0-7]`
AArch64	`"C"`, `"system"`, `"efiapi"`	`x[0-17]`, `x18` *, `x30`, `v[0-31]`, `p[0-15]`, `ffr`
ARM	`"C"`, `"system"`, `"efiapi"`, `"aapcs"`	`r[0-3]`, `r12`, `r14`, `s[0-15]`, `d[0-7]`, `d[16-31]`
RISC-V	`"C"`, `"system"`, `"efiapi"`	`x1`, `x[5-7]`, `x[10-17]`, `x[28-31]`, `f[0-7]`, `f[10-17]`, `f[28-31]`, `v[0-31]`
LoongArch	`"C"`, `"system"`, `"efiapi"`	`$r1`, `$r[4-20]`, `$f[0-23]`

Notes:

- On AArch64 `x18` only included in the clobber list if it is not considered as a reserved register on the target.

The list of clobbered registers for each ABI is updated in rustc as architectures gain new registers: this ensures that `asm!` clobbers will continue to be correct when LLVM starts using these new registers in its generated code.

Options

Flags are used to further influence the behavior of the inline assembly block. Currently the following options are defined:

- `pure`: The `asm!` block has no side effects, must eventually return, and its outputs depend only on its direct inputs (i.e. the values themselves, not what they point to) or values read from memory (unless the `nomem` options is also set). This allows the compiler to execute the `asm!` block fewer times than specified in the program (e.g. by hoisting it out of a loop) or even eliminate it entirely if the outputs are not used. The `pure` option must be combined with either the `nomem` or `readonly` options, otherwise a compile-time error is emitted.

- `nomem` : The `asm!` blocks does not read or write to any memory. This allows the compiler to cache the values of modified global variables in registers across the `asm!` block since it knows that they are not read or written to by the `asm!`. The compiler also assumes that this `asm!` block does not perform any kind of synchronization with other threads, e.g. via fences.
- `readonly` : The `asm!` block does not write to any memory. This allows the compiler to cache the values of unmodified global variables in registers across the `asm!` block since it knows that they are not written to by the `asm!`. The compiler also assumes that this `asm!` block does not perform any kind of synchronization with other threads, e.g. via fences.
- `preserves_flags` : The `asm!` block does not modify the flags register (defined in the rules below). This allows the compiler to avoid recomputing the condition flags after the `asm!` block.
- `noreturn` : The `asm!` block never returns, and its return type is defined as `!` (never). Behavior is undefined if execution falls through past the end of the asm code. A `noreturn` asm block behaves just like a function which doesn't return; notably, local variables in scope are not dropped before it is invoked.
- `nostack` : The `asm!` block does not push data to the stack, or write to the stack red-zone (if supported by the target). If this option is *not* used then the stack pointer is guaranteed to be suitably aligned (according to the target ABI) for a function call.
- `att_syntax` : This option is only valid on x86, and causes the assembler to use the `.att_syntax prefix` mode of the GNU assembler. Register operands are substituted in with a leading `%`.
- `raw` : This causes the template string to be parsed as a raw assembly string, with no special handling for `{` and `}`. This is primarily useful when including raw assembly code from an external file using `include_str!`.

The compiler performs some additional checks on options:

- The `nomem` and `readonly` options are mutually exclusive: it is a compile-time error to specify both.
- It is a compile-time error to specify `pure` on an asm block with no outputs or only discarded outputs (`_`).
- It is a compile-time error to specify `noreturn` on an asm block with outputs.

`global_asm!` only supports the `att_syntax` and `raw` options. The remaining options are not meaningful for global-scope inline assembly

Rules for inline assembly

To avoid undefined behavior, these rules must be followed when using function-scope inline assembly (`asm!`):

- Any registers not specified as inputs will contain an undefined value on entry to the asm block.
  - An "undefined value" in the context of inline assembly means that the register can (non-deterministically) have any one of the possible values allowed by the architecture. Notably it is not the same as an LLVM `undef` which can have a different value every time you read it (since such a concept does not exist in assembly code).
- Any registers not specified as outputs must have the same value upon exiting the asm block as they had on entry, otherwise behavior is undefined.
  - This only applies to registers which can be specified as an input or output. Other registers follow target-specific rules.
  - Note that a `lateout` may be allocated to the same register as an `in` , in which case this rule does not apply. Code should not rely on this however since it depends on the results of register allocation.
- Behavior is undefined if execution unwinds out of an asm block.
  - This also applies if the assembly code calls a function which then unwinds.
- The set of memory locations that assembly code is allowed to read and write are the same as those allowed for an FFI function.
  - Refer to the unsafe code guidelines for the exact rules.
  - If the `readonly` option is set, then only memory reads are allowed.
  - If the `nomem` option is set then no reads or writes to memory are allowed.
  - These rules do not apply to memory which is private to the asm code, such as stack space allocated within the asm block.
- The compiler cannot assume that the instructions in the asm are the ones that will actually end up executed.
  - This effectively means that the compiler must treat the `asm!` as a black box and only take the interface specification into account, not the instructions themselves.
  - Runtime code patching is allowed, via target-specific mechanisms.
- Unless the `nostack` option is set, asm code is allowed to use stack space below the stack pointer.
  - On entry to the asm block the stack pointer is guaranteed to be suitably aligned (according to the target ABI) for a function call.
  - You are responsible for making sure you don't overflow the stack (e.g. use stack probing to ensure you hit a guard page).
  - You should adjust the stack pointer when allocating stack memory as required by

  the target ABI.
  - The stack pointer must be restored to its original value before leaving the asm block.
- If the `noreturn` option is set then behavior is undefined if execution falls through to the end of the asm block.
- If the `pure` option is set then behavior is undefined if the `asm!` has side-effects other than its direct outputs. Behavior is also undefined if two executions of the `asm!` code with the same inputs result in different outputs.
  - When used with the `nomem` option, "inputs" are just the direct inputs of the `asm!`.
  - When used with the `readonly` option, "inputs" comprise the direct inputs of the `asm!` and any memory that the `asm!` block is allowed to read.
- These flags registers must be restored upon exiting the asm block if the `preserves_flags` option is set:
  - x86
    - Status flags in `EFLAGS` (CF, PF, AF, ZF, SF, OF).
    - Floating-point status word (all).
    - Floating-point exception flags in `MXCSR` (PE, UE, OE, ZE, DE, IE).
  - ARM
    - Condition flags in `CPSR` (N, Z, C, V)
    - Saturation flag in `CPSR` (Q)
    - Greater than or equal flags in `CPSR` (GE).
    - Condition flags in `FPSCR` (N, Z, C, V)
    - Saturation flag in `FPSCR` (QC)
    - Floating-point exception flags in `FPSCR` (IDC, IXC, UFC, OFC, DZC, IOC).
  - AArch64
    - Condition flags (`NZCV` register).
    - Floating-point status (`FPSR` register).
  - RISC-V
    - Floating-point exception flags in `fcsr` (`fflags`).
    - Vector extension state (`vtype` , `vl` , `vcsr`).
  - LoongArch
    - Floating-point condition flags in `$fcc[0-7]` .
- On x86, the direction flag (DF in `EFLAGS`) is clear on entry to an asm block and must be clear on exit.
  - Behavior is undefined if the direction flag is set on exiting an asm block.
- On x86, the x87 floating-point register stack must remain unchanged unless all of the `st([0-7])` registers have been marked as clobbered with `out("st(0)") _, out("st(1)") _, ...` .
  - If all x87 registers are clobbered then the x87 register stack is guaranteed to be

empty upon entering an `asm` block. Assembly code must ensure that the x87 register stack is also empty when exiting the asm block.

- The requirement of restoring the stack pointer and non-output registers to their original value only applies when exiting an `asm!` block.
  - This means that `asm!` blocks that never return (even if not marked `noreturn`) don't need to preserve these registers.
  - When returning to a different `asm!` block than you entered (e.g. for context switching), these registers must contain the value they had upon entering the `asm!` block that you are *exiting*.
    - You cannot exit an `asm!` block that has not been entered. Neither can you exit an `asm!` block that has already been exited (without first entering it again).
    - You are responsible for switching any target-specific state (e.g. thread-local storage, stack bounds).
    - You cannot jump from an address in one `asm!` block to an address in another, even within the same function or block, without treating their contexts as potentially different and requiring context switching. You cannot assume that any particular value in those contexts (e.g. current stack pointer or temporary values below the stack pointer) will remain unchanged between the two `asm!` blocks.
    - The set of memory locations that you may access is the intersection of those allowed by the `asm!` blocks you entered and exited.
- You cannot assume that two `asm!` blocks adjacent in source code, even without any other code between them, will end up in successive addresses in the binary without any other instructions between them.
- You cannot assume that an `asm!` block will appear exactly once in the output binary. The compiler is allowed to instantiate multiple copies of the `asm!` block, for example when the function containing it is inlined in multiple places.
- On x86, inline assembly must not end with an instruction prefix (such as `LOCK`) that would apply to instructions generated by the compiler.
  - The compiler is currently unable to detect this due to the way inline assembly is compiled, but may catch and reject this in the future.

Note: As a general rule, the flags covered by `preserves_flags` are those which are *not* preserved when performing a function call.

Correctness and Validity

In addition to all of the previous rules, the string argument to `asm!` must ultimately become — after all other arguments are evaluated, formatting is performed, and operands are translated— assembly that is both syntactically correct and semantically valid for the target architecture. The formatting rules allow the compiler to generate assembly with correct syntax. Rules concerning operands permit valid translation of Rust operands into and out of `asm!`. Adherence to these rules is necessary, but not sufficient, for the final expanded assembly to be both correct and valid. For instance:

- arguments may be placed in positions which are syntactically incorrect after formatting
- an instruction may be correctly written, but given architecturally invalid operands
- an architecturally unspecified instruction may be assembled into unspecified code
- a set of instructions, each correct and valid, may cause undefined behavior if placed in immediate succession

As a result, these rules are *non-exhaustive*. The compiler is not required to check the correctness and validity of the initial string nor the final assembly that is generated. The assembler may check for correctness and validity but is not required to do so. When using `asm!`, a typographical error may be sufficient to make a program unsound, and the rules for assembly may include thousands of pages of architectural reference manuals. Programmers should exercise appropriate care, as invoking this `unsafe` capability comes with assuming the responsibility of not violating rules of both the compiler or the architecture.

Directives Support

Inline assembly supports a subset of the directives supported by both GNU AS and LLVM's internal assembler, given as follows. The result of using other directives is assembler-specific (and may cause an error, or may be accepted as-is).

If inline assembly includes any "stateful" directive that modifies how subsequent assembly is processed, the block must undo the effects of any such directives before the inline assembly ends.

The following directives are guaranteed to be supported by the assembler:

- `.2byte`
- `.4byte`
- `.8byte`
- `.align`
- `.alt_entry`
- `.ascii`
- `.asciz`

- `.balign`
- `.balignl`
- `.balignw`
- `.bss`
- `.byte`
- `.comm`
- `.data`
- `.def`
- `.double`
- `.endef`
- `.equ`
- `.equiv`
- `.eqv`
- `.fill`
- `.float`
- `.global`
- `.globl`
- `.inst`
- `.lcomm`
- `.long`
- `.octa`
- `.option`
- `.p2align`
- `.popsection`
- `.private_extern`
- `.pushsection`
- `.quad`
- `.scl`
- `.section`
- `.set`
- `.short`
- `.size`
- `.skip`
- `.sleb128`
- `.space`
- `.string`
- `.text`
- `.type`
- `.uleb128`

- `.word`

Target Specific Directive Support

Dwarf Unwinding

The following directives are supported on ELF targets that support DWARF unwind info:

- `.cfi_adjust_cfa_offset`
- `.cfi_def_cfa`
- `.cfi_def_cfa_offset`
- `.cfi_def_cfa_register`
- `.cfi_endproc`
- `.cfi_escape`
- `.cfi_lsda`
- `.cfi_offset`
- `.cfi_personality`
- `.cfi_register`
- `.cfi_rel_offset`
- `.cfi_remember_state`
- `.cfi_restore`
- `.cfi_restore_state`
- `.cfi_return_column`
- `.cfi_same_value`
- `.cfi_sections`
- `.cfi_signal_frame`
- `.cfi_startproc`
- `.cfi_undefined`
- `.cfi_window_save`

Structured Exception Handling

On targets with structured exception Handling, the following additional directives are guaranteed to be supported:

- `.seh_endproc`
- `.seh_endprologue`
- `.seh_proc`
- `.seh_pushreg`
- `.seh_savereg`

- `.seh_setframe`
- `.seh_stackalloc`

x86 (32-bit and 64-bit)

On x86 targets, both 32-bit and 64-bit, the following additional directives are guaranteed to be supported:

- `.nops`
- `.code16`
- `.code32`
- `.code64`

Use of `.code16`, `.code32`, and `.code64` directives are only supported if the state is reset to the default before exiting the assembly block. 32-bit x86 uses `.code32` by default, and x86_64 uses `.code64` by default.

ARM (32-bit)

On ARM, the following additional directives are guaranteed to be supported:

- `.even`
- `.fnstart`
- `.fnend`
- `.save`
- `.movsp`
- `.code`
- `.thumb`
- `.thumb_func`

Unsafety

Unsafe operations are those that can potentially violate the memory-safety guarantees of Rust's static semantics.

The following language level features cannot be used in the safe subset of Rust:

- Dereferencing a raw pointer.
- Reading or writing a mutable or external static variable.
- Accessing a field of a `union`, other than to assign to it.
- Calling an unsafe function (including an intrinsic or foreign function).
- Implementing an unsafe trait.

The `unsafe` keyword

The `unsafe` keyword can occur in several different contexts: unsafe functions (`unsafe fn`), unsafe blocks (`unsafe {}`), unsafe traits (`unsafe trait`), and unsafe trait implementations (`unsafe impl`). It plays several different roles, depending on where it is used and whether the `unsafe_op_in_unsafe_fn` lint is enabled:

- it is used to mark code that *defines* extra safety conditions (`unsafe fn` , `unsafe trait`)
- it is used to mark code that needs to *satisfy* extra safety conditions (`unsafe {}` , `unsafe impl` , `unsafe fn` without `unsafe_op_in_unsafe_fn`)

The following discusses each of these cases. See the keyword documentation for some illustrative examples.

Unsafe functions (`unsafe fn`)

Unsafe functions are functions that are not safe in all contexts and/or for all possible inputs. We say they have *extra safety conditions*, which are requirements that must be upheld by all callers and that the compiler does not check. For example, `get_unchecked` has the extra safety condition that the index must be in-bounds. The unsafe function should come with documentation explaining what those extra safety conditions are.

Such a function must be prefixed with the keyword `unsafe` and can only be called from inside an `unsafe` block, or inside `unsafe fn` without the `unsafe_op_in_unsafe_fn` lint.

Unsafe blocks (`unsafe {}`)

A block of code can be prefixed with the `unsafe` keyword, to permit calling `unsafe` functions or dereferencing raw pointers. By default, the body of an unsafe function is also considered to be an unsafe block; this can be changed by enabling the `unsafe_op_in_unsafe_fn` lint.

By putting operations into an unsafe block, the programmer states that they have taken care of satisfying the extra safety conditions of all operations inside that block.

Unsafe blocks are the logical dual to unsafe functions: where unsafe functions define a proof obligation that callers must uphold, unsafe blocks state that all relevant proof obligations of functions or operations called inside the block have been discharged. There are many ways to discharge proof obligations; for example, there could be run-time checks or data structure invariants that guarantee that certain properties are definitely true, or the unsafe block could be inside an `unsafe fn`, in which case the block can use the proof obligations of that function to discharge the proof obligations arising inside the block.

Unsafe blocks are used to wrap foreign libraries, make direct use of hardware or implement features not directly present in the language. For example, Rust provides the language features necessary to implement memory-safe concurrency in the language but the implementation of threads and message passing in the standard library uses unsafe blocks.

Rust's type system is a conservative approximation of the dynamic safety requirements, so in some cases there is a performance cost to using safe code. For example, a doubly-linked list is not a tree structure and can only be represented with reference-counted pointers in safe code. By using `unsafe` blocks to represent the reverse links as raw pointers, it can be implemented without reference counting. (See "Learn Rust With Entirely Too Many Linked Lists" for a more in-depth exploration of this particular example.)

Unsafe traits (`unsafe trait`)

An unsafe trait is a trait that comes with extra safety conditions that must be upheld by *implementations* of the trait. The unsafe trait should come with documentation explaining what those extra safety conditions are.

Such a trait must be prefixed with the keyword `unsafe` and can only be implemented by `unsafe impl` blocks.

Unsafe trait implementations (`unsafe impl`)

When implementing an unsafe trait, the implementation needs to be prefixed with the `unsafe` keyword. By writing `unsafe impl`, the programmer states that they have taken care of satisfying the extra safety conditions required by the trait.

Unsafe trait implementations are the logical dual to unsafe traits: where unsafe traits define a proof obligation that implementations must uphold, unsafe implementations state that all relevant proof obligations have been discharged.

Behavior considered undefined

Rust code is incorrect if it exhibits any of the behaviors in the following list. This includes code within `unsafe` blocks and `unsafe` functions. `unsafe` only means that avoiding undefined behavior is on the programmer; it does not change anything about the fact that Rust programs must never cause undefined behavior.

It is the programmer's responsibility when writing `unsafe` code to ensure that any safe code interacting with the `unsafe` code cannot trigger these behaviors. `unsafe` code that satisfies this property for any safe client is called *sound*; if `unsafe` code can be misused by safe code to exhibit undefined behavior, it is *unsound*.

> ⚠️ ***Warning:*** The following list is not exhaustive; it may grow or shrink. There is no formal model of Rust's semantics for what is and is not allowed in unsafe code, so there may be more behavior considered unsafe. We also reserve the right to make some of the behavior in that list defined in the future. In other words, this list does not say that anything will *definitely* always be undefined in all future Rust version (but we might make such commitments for some list items in the future).
>
> ⚠️ Please read the Rustonomicon before writing unsafe code.

- Data races.
- Accessing (loading from or storing to) a place that is dangling or based on a misaligned pointer.
- Performing a place projection that violates the requirements of in-bounds pointer arithmetic. A place projection is a field expression, a tuple index expression, or an array/slice index expression.
- Breaking the pointer aliasing rules. `Box<T>`, `&mut T` and `&T` follow LLVM's scoped noalias model, except if the `&T` contains an `UnsafeCell<U>`. References and boxes must not be dangling while they are live. The exact liveness duration is not specified, but some bounds exist:
  - For references, the liveness duration is upper-bounded by the syntactic lifetime assigned by the borrow checker; it cannot be live any *longer* than that lifetime.
  - Each time a reference or box is passed to or returned from a function, it is considered live.

    - When a reference (but not a `Box`!) is passed to a function, it is live at least as long as that function call, again except if the `&T` contains an `UnsafeCell<U>`.

  All this also applies when values of these types are passed in a (nested) field of a compound type, but not behind pointer indirections.

- Mutating immutable bytes. All bytes inside a `const` item are immutable. The bytes owned by an immutable binding are immutable, unless those bytes are part of an `UnsafeCell<U>`.

  Moreover, the bytes pointed to by a shared reference, including transitively through other references (both shared and mutable) and `Box`es, are immutable; transitivity includes those references stored in fields of compound types.

  A mutation is any write of more than 0 bytes which overlaps with any of the relevant bytes (even if that write does not change the memory contents).

- Invoking undefined behavior via compiler intrinsics.

- Executing code compiled with platform features that the current platform does not support (see `target_feature`), *except* if the platform explicitly documents this to be safe.

- Calling a function with the wrong call ABI or unwinding from a function with the wrong unwind ABI.

- Producing an invalid value, even in private fields and locals. "Producing" a value happens any time a value is assigned to or read from a place, passed to a function/primitive operation or returned from a function/primitive operation. The following values are invalid (at their respective type):

    - A value other than `false` (`0`) or `true` (`1`) in a `bool`.

    - A discriminant in an `enum` not included in the type definition.

    - A null `fn` pointer.

    - A value in a `char` which is a surrogate or above `char::MAX`.

    - A `!` (all values are invalid for this type).

    - An integer (`i*` / `u*`), floating point value (`f*`), or raw pointer obtained from uninitialized memory, or uninitialized memory in a `str`.

    - A reference or `Box<T>` that is dangling, misaligned, or points to an invalid value.

  - Invalid metadata in a wide reference, `Box<T>`, or raw pointer:
    - `dyn Trait` metadata is invalid if it is not a pointer to a vtable for `Trait` that matches the actual dynamic trait the pointer or reference points to.
    - Slice metadata is invalid if the length is not a valid `usize` (i.e., it must not be read from uninitialized memory).
  - Invalid values for a type with a custom definition of invalid values. In the standard library, this affects `NonNull<T>` and `NonZero*`.

    > **Note**: `rustc` achieves this with the unstable `rustc_layout_scalar_valid_range_*` attributes.

- Incorrect use of inline assembly. For more details, refer to the rules to follow when writing code that uses inline assembly.
- **In const context**: transmuting or otherwise reinterpreting a pointer (reference, raw pointer, or function pointer) into some allocated object as a non-pointer type (such as integers). 'Reinterpreting' refers to loading the pointer value at integer type without a cast, e.g. by doing raw pointer casts or using a union.

Note: Uninitialized memory is also implicitly invalid for any type that has a restricted set of valid values. In other words, the only cases in which reading uninitialized memory is permitted are inside `union`s and in "padding" (the gaps between the fields/elements of a type).

> **Note**: Undefined behavior affects the entire program. For example, calling a function in C that exhibits undefined behavior of C means your entire program contains undefined behaviour that can also affect the Rust code. And vice versa, undefined behavior in Rust can cause adverse affects on code executed by any FFI calls to other languages.

Pointed-to bytes

The span of bytes a pointer or reference "points to" is determined by the pointer value and the size of the pointee type (using `size_of_val`).

Places based on misaligned pointers

A place is said to be "based on a misaligned pointer" if the last `*` projection during place computation was performed on a pointer that was not aligned for its type. (If there is no `*` projection in the place expression, then this is accessing the field of a local and rustc will guarantee proper alignment. If there are multiple `*` projection, then each of them incurs a load of the pointer-to-be-dereferenced itself from memory, and each of these loads is subject to the alignment constraint. Note that some `*` projections can be omitted in surface Rust syntax due to automatic dereferencing; we are considering the fully expanded place expression here.)

For instance, if `ptr` has type `*const S` where `S` has an alignment of 8, then `ptr` must be 8-aligned or else `(*ptr).f` is "based on an misaligned pointer". This is true even if the type of the field `f` is `u8` (i.e., a type with alignment 1). In other words, the alignment requirement derives from the type of the pointer that was dereferenced, *not* the type of the field that is being accessed.

Note that a place based on a misaligned pointer only leads to Undefined Behavior when it is loaded from or stored to. `addr_of!` / `addr_of_mut!` on such a place is allowed. `&` / `&mut` on a place requires the alignment of the field type (or else the program would be "producing an invalid value"), which generally is a less restrictive requirement than being based on an aligned pointer. Taking a reference will lead to a compiler error in cases where the field type might be more aligned than the type that contains it, i.e., `repr(packed)`. This means that being based on an aligned pointer is always sufficient to ensure that the new reference is aligned, but it is not always necessary.

Dangling pointers

A reference/pointer is "dangling" if it is null or not all of the bytes it points to are part of the same live allocation (so in particular they all have to be part of *some* allocation).

If the size is 0, then the pointer must either point inside of a live allocation (including pointing just after the last byte of the allocation), or it must be directly constructed from a non-zero integer literal.

Note that dynamically sized types (such as slices and strings) point to their entire range, so it is important that the length metadata is never too large. In particular, the dynamic size of a Rust value (as determined by `size_of_val`) must never exceed `isize::MAX`.

Behavior not considered `unsafe`

The Rust compiler does not consider the following behaviors *unsafe*, though a programmer may (should) find them undesirable, unexpected, or erroneous.

Deadlocks

Leaks of memory and other resources

Exiting without calling destructors

Exposing randomized base addresses through pointer leaks

Integer overflow

If a program contains arithmetic overflow, the programmer has made an error. In the following discussion, we maintain a distinction between arithmetic overflow and wrapping arithmetic. The first is erroneous, while the second is intentional.

When the programmer has enabled `debug_assert!` assertions (for example, by enabling a non-optimized build), implementations must insert dynamic checks that `panic` on overflow. Other kinds of builds may result in `panics` or silently wrapped values on overflow, at the implementation's discretion.

In the case of implicitly-wrapped overflow, implementations must provide well-defined (even if still considered erroneous) results by using two's complement overflow conventions.

The integral types provide inherent methods to allow programmers explicitly to perform wrapping arithmetic. For example, `i32::wrapping_add` provides two's complement, wrapping addition.

The standard library also provides a `Wrapping<T>` newtype which ensures all standard arithmetic operations for `T` have wrapping semantics.

See RFC 560 for error conditions, rationale, and more details about integer overflow.

Logic errors

Safe code may impose extra logical constraints that can be checked at neither compile-time nor runtime. If a program breaks such a constraint, the behavior may be unspecified but will not result in undefined behavior. This could include panics, incorrect results, aborts, and non-termination. The behavior may also differ between runs, builds, or kinds of build.

For example, implementing both `Hash` and `Eq` requires that values considered equal have equal hashes. Another example are data structures like `BinaryHeap`, `BTreeMap`, `BTreeSet`, `HashMap` and `HashSet` which describe constraints on the modification of their keys while they are in the data structure. Violating such constraints is not considered unsafe, yet the program is considered erroneous and its behavior unpredictable.

Constant evaluation

Constant evaluation is the process of computing the result of expressions during compilation. Only a subset of all expressions can be evaluated at compile-time.

Constant expressions

Certain forms of expressions, called constant expressions, can be evaluated at compile time. In const contexts, these are the only allowed expressions, and are always evaluated at compile time. In other places, such as let statements, constant expressions *may* be, but are not guaranteed to be, evaluated at compile time. Behaviors such as out of bounds array indexing or overflow are compiler errors if the value must be evaluated at compile time (i.e. in const contexts). Otherwise, these behaviors are warnings, but will likely panic at run-time.

The following expressions are constant expressions, so long as any operands are also constant expressions and do not cause any `Drop::drop` calls to be run.

- Literals.
- Const parameters.
- Paths to functions and constants. Recursively defining constants is not allowed.
- Paths to statics. These are only allowed within the initializer of a static.
- Tuple expressions.
- Array expressions.
- Struct expressions.
- Block expressions, including `unsafe` blocks.
  - let statements and thus irrefutable patterns, including mutable bindings
  - assignment expressions
  - compound assignment expressions
  - expression statements
- Field expressions.
- Index expressions, array indexing or slice with a `usize` .
- Range expressions.
- Closure expressions which don't capture variables from the environment.
- Built-in negation, arithmetic, logical, comparison or lazy boolean operators used on integer and floating point types, `bool` , and `char` .
- Shared borrows, except if applied to a type with interior mutability.
- The dereference operator except for raw pointers.
- Grouped expressions.

- Cast expressions, except
  - pointer to address casts and
  - function pointer to address casts.
- Calls of const functions and const methods.
- loop, while and `while let` expressions.
- if, `if let` and match expressions.

Const context

A *const context* is one of the following:

- Array type length expressions
- Array repeat length expressions
- The initializer of
  - constants
  - statics
  - enum discriminants
- A const generic argument

Const Functions

A *const fn* is a function that one is permitted to call from a const context. Declaring a function `const` has no effect on any existing uses, it only restricts the types that arguments and the return type may use, as well as prevent various expressions from being used within it. You can freely do anything with a const function that you can do with a regular function.

When called from a const context, the function is interpreted by the compiler at compile time. The interpretation happens in the environment of the compilation target and not the host. So `usize` is `32` bits if you are compiling against a `32` bit system, irrelevant of whether you are building on a `64` bit or a `32` bit system.

Const functions have various restrictions to make sure that they can be evaluated at compile-time. It is, for example, not possible to write a random number generator as a const function. Calling a const function at compile-time will always yield the same result as calling it at runtime, even when called multiple times. There's one exception to this rule: if you are doing complex floating point operations in extreme situations, then you might get (very slightly) different results. It is advisable to not make array lengths and enum discriminants depend on floating point computations.

Notable features that are allowed in const contexts but not in const functions include:

- floating point operations
  - floating point values are treated just like generic parameters without trait bounds beyond `Copy`. So you cannot do anything with them but copy/move them around.

Conversely, the following are possible in a const function, but not in a const context:

- Use of generic type and lifetime parameters.
  - Const contexts do allow limited use of const generic parameters.

Application Binary Interface (ABI)

This section documents features that affect the ABI of the compiled output of a crate.

See *extern functions* for information on specifying the ABI for exporting functions. See *external blocks* for information on specifying the ABI for linking external libraries.

The `used` attribute

The *`used` attribute* can only be applied to `static` items. This attribute forces the compiler to keep the variable in the output object file (.o, .rlib, etc. excluding final binaries) even if the variable is not used, or referenced, by any other item in the crate. However, the linker is still free to remove such an item.

Below is an example that shows under what conditions the compiler keeps a `static` item in the output object file.

```
// foo.rs

// This is kept because of `#[used]`:
#[used]
static FOO: u32 = 0;

// This is removable because it is unused:
#[allow(dead_code)]
static BAR: u32 = 0;

// This is kept because it is publicly reachable:
pub static BAZ: u32 = 0;

// This is kept because it is referenced by a public, reachable function:
static QUUX: u32 = 0;

pub fn quux() -> &'static u32 {
    &QUUX
}

// This is removable because it is referenced by a private, unused (dead)
function:
static CORGE: u32 = 0;

#[allow(dead_code)]
fn corge() -> &'static u32 {
    &CORGE
}
```

```
$ rustc -O --emit=obj --crate-type=rlib foo.rs

$ nm -C foo.o
0000000000000000 R foo::BAZ
0000000000000000 r foo::FOO
0000000000000000 R foo::QUUX
0000000000000000 T foo::quux
```

The `no_mangle` attribute

The *`no_mangle` attribute* may be used on any item to disable standard symbol name mangling. The symbol for the item will be the identifier of the item's name.

Additionally, the item will be publicly exported from the produced library or object file, similar to the `used` attribute.

The `link_section` attribute

The *`link_section` attribute* specifies the section of the object file that a function or static's content will be placed into. It uses the *MetaNameValueStr* syntax to specify the section name.

```
#[no_mangle]
#[link_section = ".example_section"]
pub static VAR1: u32 = 1;
```

The `export_name` attribute

The *`export_name` attribute* specifies the name of the symbol that will be exported on a function or static. It uses the *MetaNameValueStr* syntax to specify the symbol name.

```
#[export_name = "exported_symbol_name"]
pub fn name_in_rust() { }
```

The Rust runtime

This section documents features that define some aspects of the Rust runtime.

The `panic_handler` attribute

The *`panic_handler` attribute* can only be applied to a function with signature `fn(&PanicInfo) -> !`. The function marked with this attribute defines the behavior of panics. The `PanicInfo` struct contains information about the location of the panic. There must be a single `panic_handler` function in the dependency graph of a binary, dylib or cdylib crate.

Below is shown a `panic_handler` function that logs the panic message and then halts the thread.

```
#![no_std]

use core::fmt::{self, Write};
use core::panic::PanicInfo;

struct Sink {
    // ..
}

#[panic_handler]
fn panic(info: &PanicInfo) -> ! {
    let mut sink = Sink::new();

    // logs "panicked at '$reason', src/main.rs:27:4" to some `sink`
    let _ = writeln!(sink, "{}", info);

    loop {}
}
```

Standard behavior

The standard library provides an implementation of `panic_handler` that defaults to unwinding the stack but that can be changed to abort the process. The standard library's panic behavior can be modified at runtime with the set_hook function.

The `global_allocator` attribute

The *`global_allocator` attribute* is used on a static item implementing the `GlobalAlloc` trait to set the global allocator.

The `windows_subsystem` attribute

The *`windows_subsystem` attribute* may be applied at the crate level to set the subsystem when linking on a Windows target. It uses the *MetaNameValueStr* syntax to specify the subsystem with a value of either `console` or `windows`. This attribute is ignored on non-Windows targets, and for non-`bin` crate types.

The "console" subsystem is the default. If a console process is run from an existing console then it will be attached to that console, otherwise a new console window will be created.

The "windows" subsystem is commonly used by GUI applications that do not want to display a console window on startup. It will run detached from any existing console.

```
#![windows_subsystem = "windows"]
```

Appendices

Appendix: Macro Follow-Set Ambiguity Formal Specification

This page documents the formal specification of the follow rules for Macros By Example. They were originally specified in RFC 550, from which the bulk of this text is copied, and expanded upon in subsequent RFCs.

Definitions & Conventions

- `macro` : anything invokable as `foo!(...)` in source code.
- `MBE` : macro-by-example, a macro defined by `macro_rules` .
- `matcher` : the left-hand-side of a rule in a `macro_rules` invocation, or a subportion thereof.
- `macro parser` : the bit of code in the Rust parser that will parse the input using a grammar derived from all of the matchers.
- `fragment` : The class of Rust syntax that a given matcher will accept (or "match").
- `repetition` : a fragment that follows a regular repeating pattern
- `NT` : non-terminal, the various "meta-variables" or repetition matchers that can appear in a matcher, specified in MBE syntax with a leading `$` character.
- `simple NT` : a "meta-variable" non-terminal (further discussion below).
- `complex NT` : a repetition matching non-terminal, specified via repetition operators (`*` , `+` , `?`).
- `token` : an atomic element of a matcher; i.e. identifiers, operators, open/close delimiters, *and* simple NT's.
- `token tree` : a tree structure formed from tokens (the leaves), complex NT's, and finite sequences of token trees.
- `delimiter token` : a token that is meant to divide the end of one fragment and the start of the next fragment.
- `separator token` : an optional delimiter token in an complex NT that separates each pair of elements in the matched repetition.
- `separated complex NT` : a complex NT that has its own separator token.
- `delimited sequence` : a sequence of token trees with appropriate open- and close-delimiters at the start and end of the sequence.
- `empty fragment` : The class of invisible Rust syntax that separates tokens, i.e. whitespace, or (in some lexical contexts), the empty token sequence.
- `fragment specifier` : The identifier in a simple NT that specifies which fragment the

NT accepts.
- `language` : a context-free language.

Example:

```
macro_rules! i_am_an_mbe {
    (start $foo:expr $($i:ident),* end) => ($foo)
}
```

`(start $foo:expr $($i:ident),* end)` is a matcher. The whole matcher is a delimited sequence (with open- and close-delimiters `(` and `)`), and `$foo` and `$i` are simple NT's with `expr` and `ident` as their respective fragment specifiers.

`$(i:ident),*` is *also* an NT; it is a complex NT that matches a comma-separated repetition of identifiers. The `,` is the separator token for the complex NT; it occurs in between each pair of elements (if any) of the matched fragment.

Another example of a complex NT is `$(hi $e:expr ;)+`, which matches any fragment of the form `hi <expr>; hi <expr>; ...` where `hi <expr>;` occurs at least once. Note that this complex NT does not have a dedicated separator token.

(Note that Rust's parser ensures that delimited sequences always occur with proper nesting of token tree structure and correct matching of open- and close-delimiters.)

We will tend to use the variable "M" to stand for a matcher, variables "t" and "u" for arbitrary individual tokens, and the variables "tt" and "uu" for arbitrary token trees. (The use of "tt" does present potential ambiguity with its additional role as a fragment specifier; but it will be clear from context which interpretation is meant.)

"SEP" will range over separator tokens, "OP" over the repetition operators `*`, `+`, and `?`, "OPEN"/"CLOSE" over matching token pairs surrounding a delimited sequence (e.g. `[` and `]`).

Greek letters "α" "β" "γ" "δ" stand for potentially empty token-tree sequences. (However, the Greek letter "ε" (epsilon) has a special role in the presentation and does not stand for a token-tree sequence.)

- This Greek letter convention is usually just employed when the presence of a sequence is a technical detail; in particular, when we wish to *emphasize* that we are operating on a sequence of token-trees, we will use the notation "tt ..." for the sequence, not a Greek letter.

Note that a matcher is merely a token tree. A "simple NT", as mentioned above, is an meta-variable NT; thus it is a non-repetition. For example, `$foo:ty` is a simple NT but `$($foo:ty)`

`+` is a complex NT.

Note also that in the context of this formalism, the term "token" generally *includes* simple NTs.

Finally, it is useful for the reader to keep in mind that according to the definitions of this formalism, no simple NT matches the empty fragment, and likewise no token matches the empty fragment of Rust syntax. (Thus, the *only* NT that can match the empty fragment is a complex NT.) This is not actually true, because the `vis` matcher can match an empty fragment. Thus, for the purposes of the formalism, we will treat `$v:vis` as actually being `$($v:vis)?`, with a requirement that the matcher match an empty fragment.

The Matcher Invariants

To be valid, a matcher must meet the following three invariants. The definitions of FIRST and FOLLOW are described later.

1. For any two successive token tree sequences in a matcher `M` (i.e. `M = ... tt uu ...`) with `uu ...` nonempty, we must have FOLLOW(`... tt`) ∪ {ε} ⊇ FIRST(`uu ...`).
2. For any separated complex NT in a matcher, `M = ... $(tt ...) SEP OP ...`, we must have `SEP` ∈ FOLLOW(`tt ...`).
3. For an unseparated complex NT in a matcher, `M = ... $(tt ...) OP ...`, if OP = `*` or `+`, we must have FOLLOW(`tt ...`) ⊇ FIRST(`tt ...`).

The first invariant says that whatever actual token that comes after a matcher, if any, must be somewhere in the predetermined follow set. This ensures that a legal macro definition will continue to assign the same determination as to where `... tt` ends and `uu ...` begins, even as new syntactic forms are added to the language.

The second invariant says that a separated complex NT must use a separator token that is part of the predetermined follow set for the internal contents of the NT. This ensures that a legal macro definition will continue to parse an input fragment into the same delimited sequence of `tt ...`'s, even as new syntactic forms are added to the language.

The third invariant says that when we have a complex NT that can match two or more copies of the same thing with no separation in between, it must be permissible for them to be placed next to each other as per the first invariant. This invariant also requires they be nonempty, which eliminates a possible ambiguity.

NOTE: The third invariant is currently unenforced due to historical oversight and significant reliance on the behaviour. It is currently undecided what to do about this going forward. Macros that do not respect the behaviour may become invalid in a

future edition of Rust. See the tracking issue.

FIRST and FOLLOW, informally

A given matcher M maps to three sets: FIRST(M), LAST(M) and FOLLOW(M).

Each of the three sets is made up of tokens. FIRST(M) and LAST(M) may also contain a distinguished non-token element ε ("epsilon"), which indicates that M can match the empty fragment. (But FOLLOW(M) is always just a set of tokens.)

Informally:

- FIRST(M): collects the tokens potentially used first when matching a fragment to M.
- LAST(M): collects the tokens potentially used last when matching a fragment to M.
- FOLLOW(M): the set of tokens allowed to follow immediately after some fragment matched by M.

  In other words: t ∈ FOLLOW(M) if and only if there exists (potentially empty) token sequences α, β, γ, δ where:

  - M matches β,
  - t matches γ, and
  - The concatenation α β γ δ is a parseable Rust program.

We use the shorthand ANYTOKEN to denote the set of all tokens (including simple NTs). For example, if any token is legal after a matcher M, then FOLLOW(M) = ANYTOKEN.

(To review one's understanding of the above informal descriptions, the reader at this point may want to jump ahead to the examples of FIRST/LAST before reading their formal definitions.)

FIRST, LAST

Below are formal inductive definitions for FIRST and LAST.

"A ∪ B" denotes set union, "A ∩ B" denotes set intersection, and "A \ B" denotes set difference (i.e. all elements of A that are not present in B).

FIRST

FIRST(M) is defined by case analysis on the sequence M and the structure of its first token-tree (if any):

- if M is the empty sequence, then FIRST(M) = { ε },
- if M starts with a token t, then FIRST(M) = { t },

  (Note: this covers the case where M starts with a delimited token-tree sequence, `M = OPEN tt ... CLOSE ...`, in which case `t = OPEN` and thus FIRST(M) = { `OPEN` }.)

  (Note: this critically relies on the property that no simple NT matches the empty fragment.)

- Otherwise, M is a token-tree sequence starting with a complex NT: `M = $( tt ... ) OP α`, or `M = $( tt ... ) SEP OP α`, (where `α` is the (potentially empty) sequence of token trees for the rest of the matcher).
  - Let SEP_SET(M) = { SEP } if SEP is present and ε ∈ FIRST(`tt ...`); otherwise SEP_SET(M) = {}.
- Let ALPHA_SET(M) = FIRST(`α`) if OP = `*` or `?` and ALPHA_SET(M) = {} if OP = `+`.
- FIRST(M) = (FIRST(`tt ...`) \ {ε}) ∪ SEP_SET(M) ∪ ALPHA_SET(M).

The definition for complex NTs deserves some justification. SEP_SET(M) defines the possibility that the separator could be a valid first token for M, which happens when there is a separator defined and the repeated fragment could be empty. ALPHA_SET(M) defines the possibility that the complex NT could be empty, meaning that M's valid first tokens are those of the following token-tree sequences `α`. This occurs when either `*` or `?` is used, in which case there could be zero repetitions. In theory, this could also occur if `+` was used with a potentially-empty repeating fragment, but this is forbidden by the third invariant.

From there, clearly FIRST(M) can include any token from SEP_SET(M) or ALPHA_SET(M), and if the complex NT match is nonempty, then any token starting FIRST(`tt ...`) could work too. The last piece to consider is ε. SEP_SET(M) and FIRST(`tt ...`) \ {ε} cannot contain ε, but ALPHA_SET(M) could. Hence, this definition allows M to accept ε if and only if ε ∈ ALPHA_SET(M) does. This is correct because for M to accept ε in the complex NT case, both the complex NT and α must accept it. If OP = `+`, meaning that the complex NT cannot be empty, then by definition ε ∉ ALPHA_SET(M). Otherwise, the complex NT can accept zero repetitions, and then ALPHA_SET(M) = FOLLOW(`α`). So this definition is correct with respect to \varepsilon as well.

LAST

LAST(M), defined by case analysis on M itself (a sequence of token-trees):

- if M is the empty sequence, then LAST(M) = { ε }
- if M is a singleton token t, then LAST(M) = { t }
- if M is the singleton complex NT repeating zero or more times, `M = $( tt ... ) *`, or `M = $( tt ... ) SEP *`
    - Let sep_set = { SEP } if SEP present; otherwise sep_set = {}.
    - if ε ∈ LAST(`tt ...`) then LAST(M) = LAST(`tt ...`) ∪ sep_set
    - otherwise, the sequence `tt ...` must be non-empty; LAST(M) = LAST(`tt ...`) ∪ {ε}.
- if M is the singleton complex NT repeating one or more times, `M = $( tt ... ) +`, or `M = $( tt ... ) SEP +`
    - Let sep_set = { SEP } if SEP present; otherwise sep_set = {}.
    - if ε ∈ LAST(`tt ...`) then LAST(M) = LAST(`tt ...`) ∪ sep_set
    - otherwise, the sequence `tt ...` must be non-empty; LAST(M) = LAST(`tt ...`)
- if M is the singleton complex NT repeating zero or one time, `M = $( tt ...) ?`, then LAST(M) = LAST(`tt ...`) ∪ {ε}.
- if M is a delimited token-tree sequence `OPEN tt ... CLOSE`, then LAST(M) = { `CLOSE` }.
- if M is a non-empty sequence of token-trees `tt uu ...`,
    - If ε ∈ LAST(`uu ...`), then LAST(M) = LAST(`tt`) ∪ (LAST(`uu ...`) \ { ε }).
    - Otherwise, the sequence `uu ...` must be non-empty; then LAST(M) = LAST(`uu ...`).

Examples of FIRST and LAST

Below are some examples of FIRST and LAST. (Note in particular how the special ε element is introduced and eliminated based on the interaction between the pieces of the input.)

Our first example is presented in a tree structure to elaborate on how the analysis of the matcher composes. (Some of the simpler subtrees have been elided.)

```
INPUT:  $(  $d:ident   $e:expr   );*    $( $( h )* );*    $( f ; )+   g
            ~~~~~~~~   ~~~~~~~                ~
                |         |                   |
FIRST:   { $d:ident }  { $e:expr }          { h }


INPUT:  $(  $d:ident   $e:expr   );*    $( $( h )* );*    $( f ; )+
            ~~~~~~~~~~~~~~~~~~             ~~~~~~~           ~~~
                        |                      |                |
FIRST:          { $d:ident }               { h, ε }         { f }

INPUT:  $(  $d:ident   $e:expr   );*    $( $( h )* );*    $( f ; )+   g
        ~~~~~~~~~~~~~~~~~~~~~~~~~~~~    ~~~~~~~~~~~~~~    ~~~~~~~~~   ~
                        |                       |               |       |
FIRST:        { $d:ident, ε }            {  h, ε, ;  }      { f }   { g }


INPUT:  $(  $d:ident   $e:expr   );*    $( $( h )* );*    $( f ; )+   g
        ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                                        |
FIRST:                       { $d:ident, h, ;,  f }
```

Thus:

- FIRST(`$($d:ident $e:expr );* $( $(h)* );* $( f ;)+ g`) = { `$d:ident`, `h`, `;`, `f` }

Note however that:

- FIRST(`$($d:ident $e:expr );* $( $(h)* );* $($( f ;)+ g)*`) = { `$d:ident`, `h`, `;`, `f`, ε }

Here are similar examples but now for LAST.

- LAST(`$d:ident $e:expr`) = { `$e:expr` }
- LAST(`$( $d:ident $e:expr );*`) = { `$e:expr`, ε }
- LAST(`$( $d:ident $e:expr );* $(h)*`) = { `$e:expr`, ε, `h` }
- LAST(`$( $d:ident $e:expr );* $(h)* $( f ;)+`) = { `;` }
- LAST(`$( $d:ident $e:expr );* $(h)* $( f ;)+ g`) = { `g` }

FOLLOW(M)

Finally, the definition for FOLLOW(M) is built up as follows. pat, expr, etc. represent simple nonterminals with the given fragment specifier.

- FOLLOW(pat) = { `=>` , `,` , `=` , `|` , `if` , `in` }`.
- FOLLOW(expr) = FOLLOW(stmt) = { `=>` , `,` , `;` }`.
- FOLLOW(ty) = FOLLOW(path) = { `{` , `[` , `,` , `=>` , `:` , `=` , `>` , `>>` , `;` , `|` , `as` , `where` , block nonterminals}.
- FOLLOW(vis) = { `,` l any keyword or identifier except a non-raw `priv` ; any token that can begin a type; ident, ty, and path nonterminals}.
- FOLLOW(t) = ANYTOKEN for any other simple token, including block, ident, tt, item, lifetime, literal and meta simple nonterminals, and all terminals.
- FOLLOW(M), for any other M, is defined as the intersection, as t ranges over (LAST(M) \ {ε}), of FOLLOW(t).

The tokens that can begin a type are, as of this writing, { `(` , `[` , `!` , `*` , `&` , `&&` , `?` , lifetimes, `>` , `>>` , `::` , any non-keyword identifier, `super` , `self` , `Self` , `extern` , `crate` , `$crate` , `_` , `for` , `impl` , `fn` , `unsafe` , `typeof` , `dyn` }, although this list may not be complete because people won't always remember to update the appendix when new ones are added.

Examples of FOLLOW for complex M:

- FOLLOW(`$( $d:ident $e:expr )*`) = FOLLOW(`$e:expr`)
- FOLLOW(`$( $d:ident $e:expr )* $(;)*`) = FOLLOW(`$e:expr`) ∩ ANYTOKEN = FOLLOW(`$e:expr`)
- FOLLOW(`$( $d:ident $e:expr )* $(;)* $( f |)+`) = ANYTOKEN

Examples of valid and invalid matchers

With the above specification in hand, we can present arguments for why particular matchers are legal and others are not.

- `($ty:ty < foo ,)` : illegal, because FIRST(`< foo ,`) = { `<` } ⊈ FOLLOW(`ty`)
- `($ty:ty , foo <)` : legal, because FIRST(`, foo <`) = { `,` } is ⊆ FOLLOW(`ty`).
- `($pa:pat $pb:pat $ty:ty ,)` : illegal, because FIRST(`$pb:pat $ty:ty ,`) = { `$pb:pat` } ⊈ FOLLOW(`pat`), and also FIRST(`$ty:ty ,`) = { `$ty:ty` } ⊈ FOLLOW(`pat`).
- `( $($a:tt $b:tt)* ; )` : legal, because FIRST(`$b:tt`) = { `$b:tt` } is ⊆ FOLLOW(`tt`) =

ANYTOKEN, as is FIRST(`;`) = { `;` }.

- `( $($t:tt),* , $(t:tt),* )` : legal, (though any attempt to actually use this macro will signal a local ambiguity error during expansion).
- `($ty:ty $(; not sep)* -)` : illegal, because FIRST(`$(; not sep)* -`) = { `;`, `-` } is not in FOLLOW(`ty`).
- `($($ty:ty)-+)` : illegal, because separator `-` is not in FOLLOW(`ty`).
- `($($e:expr)*)` : illegal, because expr NTs are not in FOLLOW(expr NT).

Influences

Rust is not a particularly original language, with design elements coming from a wide range of sources. Some of these are listed below (including elements that have since been removed):

- SML, OCaml: algebraic data types, pattern matching, type inference, semicolon statement separation
- C++: references, RAII, smart pointers, move semantics, monomorphization, memory model
- ML Kit, Cyclone: region based memory management
- Haskell (GHC): typeclasses, type families
- Newsqueak, Alef, Limbo: channels, concurrency
- Erlang: message passing, thread failure, ~~linked thread failure~~, ~~lightweight concurrency~~
- Swift: optional bindings
- Scheme: hygienic macros
- C#: attributes
- Ruby: closure syntax, ~~block syntax~~
- NIL, Hermes: ~~typestate~~
- Unicode Annex #31: identifier and pattern syntax

Glossary

Abstract syntax tree

An ‘abstract syntax tree’, or ‘AST’, is an intermediate representation of the structure of the program when the compiler is compiling it.

Alignment

The alignment of a value specifies what addresses values are preferred to start at. Always a power of two. References to a value must be aligned. More.

Arity

Arity refers to the number of arguments a function or operator takes. For some examples, `f(2, 3)` and `g(4, 6)` have arity 2, while `h(8, 2, 6)` has arity 3. The `!` operator has arity 1.

Array

An array, sometimes also called a fixed-size array or an inline array, is a value describing a collection of elements, each selected by an index that can be computed at run time by the program. It occupies a contiguous region of memory.

Associated item

An associated item is an item that is associated with another item. Associated items are defined in implementations and declared in traits. Only functions, constants, and type aliases can be associated. Contrast to a free item.

Blanket implementation

Any implementation where a type appears uncovered. `impl<T> Foo for T`, `impl<T> Bar<T> for T`, `impl<T> Bar<Vec<T>> for T`, and `impl<T> Bar<T> for Vec<T>` are considered blanket impls. However, `impl<T> Bar<Vec<T>> for Vec<T>` is not a blanket impl, as all instances of `T` which appear in this `impl` are covered by `Vec`.

Bound

Bounds are constraints on a type or trait. For example, if a bound is placed on the argument a function takes, types passed to that function must abide by that constraint.

Combinator

Combinators are higher-order functions that apply only functions and earlier defined combinators to provide a result from its arguments. They can be used to manage control flow in a modular fashion.

Crate

A crate is the unit of compilation and linking. There are different types of crates, such as libraries or executables. Crates may link and refer to other library crates, called external crates. A crate has a self-contained tree of modules, starting from an unnamed root module called the crate root. Items may be made visible to other crates by marking them as public in the crate root, including through paths of public modules. More.

Dispatch

Dispatch is the mechanism to determine which specific version of code is actually run when it involves polymorphism. Two major forms of dispatch are static dispatch and dynamic dispatch. While Rust favors static dispatch, it also supports dynamic dispatch through a mechanism called 'trait objects'.

Dynamically sized type

A dynamically sized type (DST) is a type without a statically known size or alignment.

Entity

An *entity* is a language construct that can be referred to in some way within the source program, usually via a path. Entities include types, items, generic parameters, variable bindings, loop labels, lifetimes, fields, attributes, and lints.

Expression

An expression is a combination of values, constants, variables, operators and functions that evaluate to a single value, with or without side-effects.

For example, `2 + (3 * 4)` is an expression that returns the value 14.

Free item

An item that is not a member of an implementation, such as a *free function* or a *free const*. Contrast to an associated item.

Fundamental traits

A fundamental trait is one where adding an impl of it for an existing type is a breaking change. The `Fn` traits and `Sized` are fundamental.

Fundamental type constructors

A fundamental type constructor is a type where implementing a blanket implementation over it is a breaking change. `&`, `&mut`, `Box`, and `Pin` are fundamental.

Any time a type `T` is considered local, `&T`, `&mut T`, `Box<T>`, and `Pin<T>` are also considered local. Fundamental type constructors cannot cover other types. Any time the term "covered type" is used, the `T` in `&T`, `&mut T`, `Box<T>`, and `Pin<T>` is not considered covered.

Inhabited

A type is inhabited if it has constructors and therefore can be instantiated. An inhabited type is not "empty" in the sense that there can be values of the type. Opposite of Uninhabited.

Inherent implementation

An implementation that applies to a nominal type, not to a trait-type pair. More.

Inherent method

A method defined in an inherent implementation, not in a trait implementation.

Initialized

A variable is initialized if it has been assigned a value and hasn't since been moved from. All other memory locations are assumed to be uninitialized. Only unsafe Rust can create a memory location without initializing it.

Local trait

A `trait` which was defined in the current crate. A trait definition is local or not independent of applied type arguments. Given `trait Foo<T, U>`, `Foo` is always local, regardless of the types substituted for `T` and `U`.

Local type

A `struct`, `enum`, or `union` which was defined in the current crate. This is not affected by applied type arguments. `struct Foo` is considered local, but `Vec<Foo>` is not. `LocalType<ForeignType>` is local. Type aliases do not affect locality.

Module

A module is a container for zero or more items. Modules are organized in a tree, starting from an unnamed module at the root called the crate root or the root module. Paths may be used to refer to items from other modules, which may be restricted by visibility rules. More

Name

A *name* is an identifier or lifetime or loop label that refers to an entity. A *name binding* is when an entity declaration introduces an identifier or label associated with that entity. Paths, identifiers, and labels are used to refer to an entity.

Name resolution

Name resolution is the compile-time process of tying paths, identifiers, and labels to entity declarations.

Namespace

A *namespace* is a logical grouping of declared names based on the kind of entity the name refers to. Namespaces allow the occurrence of a name in one namespace to not conflict with the same name in another namespace.

Within a namespace, names are organized in a hierarchy, where each level of the hierarchy has its own collection of named entities.

Nominal types

Types that can be referred to by a path directly. Specifically enums, structs, unions, and trait objects.

Object safe traits

Traits that can be used as trait objects. Only traits that follow specific rules are object safe.

Path

A *path* is a sequence of one or more path segments used to refer to an entity in the current scope or other levels of a namespace hierarchy.

Prelude

Prelude, or The Rust Prelude, is a small collection of items - mostly traits - that are imported into every module of every crate. The traits in the prelude are pervasive.

Scope

A *scope* is the region of source text where a named entity may be referenced with that name.

Scrutinee

A scrutinee is the expression that is matched on in `match` expressions and similar pattern matching constructs. For example, in `match x { A => 1, B => 2 }`, the expression `x` is the scrutinee.

Size

The size of a value has two definitions.

The first is that it is how much memory must be allocated to store that value.

The second is that it is the offset in bytes between successive elements in an array with that item type.

It is a multiple of the alignment, including zero. The size can change depending on compiler version (as new optimizations are made) and target platform (similar to how `usize` varies per-platform).

More.

Slice

A slice is dynamically-sized view into a contiguous sequence, written as `[T]`.

It is often seen in its borrowed forms, either mutable or shared. The shared slice type is `&[T]`, while the mutable slice type is `&mut [T]`, where `T` represents the element type.

Statement

A statement is the smallest standalone element of a programming language that commands a computer to perform an action.

String literal

A string literal is a string stored directly in the final binary, and so will be valid for the `'static` duration.

Its type is `'static` duration borrowed string slice, `&'static str`.

String slice

A string slice is the most primitive string type in Rust, written as `str`. It is often seen in its borrowed forms, either mutable or shared. The shared string slice type is `&str`, while the mutable string slice type is `&mut str`.

Strings slices are always valid UTF-8.

Trait

A trait is a language item that is used for describing the functionalities a type must provide. It allows a type to make certain promises about its behavior.

Generic functions and generic structs can use traits to constrain, or bound, the types they accept.

Turbofish

Paths with generic parameters in expressions must prefix the opening brackets with a `::`. Combined with the angular brackets for generics, this looks like a fish `::<>`. As such, this syntax is colloquially referred to as turbofish syntax.

Examples:

```
let ok_num = Ok::<_, ()>(5);
let vec = [1, 2, 3].iter().map(|n| n * 2).collect::<Vec<_>>();
```

This `::` prefix is required to disambiguate generic paths with multiple comparisons in a comma-separate list. See the bastion of the turbofish for an example where not having the prefix would be ambiguous.

Uncovered type

A type which does not appear as an argument to another type. For example, `T` is uncovered, but the `T` in `Vec<T>` is covered. This is only relevant for type arguments.

Undefined behavior

Compile-time or run-time behavior that is not specified. This may result in, but is not limited to: process termination or corruption; improper, incorrect, or unintended computation; or platform-specific results. More.

Uninhabited

A type is uninhabited if it has no constructors and therefore can never be instantiated. An uninhabited type is "empty" in the sense that there are no values of the type. The canonical example of an uninhabited type is the never type `!`, or an enum with no variants `enum Never { }`. Opposite of Inhabited.