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.

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.

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 the 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");
  }
  ```

- 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 `#docs` channels on Discord. Knowing what people use this book for the most helps direct our attention to making those sections the best that they can be.

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

Rust input is interpreted as a sequence of Unicode code points encoded in UTF-8.

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`.

- `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

  ```
  // 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_UNION : `union`
KW_STATICLIFETIME : `'static`

Lexer 2015
KW_DYN : `dyn`

Identifiers

Lexer:
IDENTIFIER_OR_KEYWORD :
   `[a-z A-Z] [a-z A-Z 0-9 _]`*
 | `_ [a-z A-Z 0-9 _]`+

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

An identifier is any nonempty ASCII string of the following form:

Either

- The first character is a letter.
- The remaining characters are alphanumeric or `_`.

Or

- The first character is `_`.
- The identifier is more than one character. `_` alone is not an identifier.
- The remaining characters are alphanumeric or `_`.

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`*
 | `//`

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 :
  A `\r` *not followed by a* `\n`

Non-doc comments

Comments in Rust code 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 inner 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.

Isolated CRs (`\r`), i.e. not followed by LF (`\n`), are not allowed in doc comments.

Examples

```
//! 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

A literal 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.

Examples

Characters and strings

	Example	`#` sets	Characters	Escapes
Character	`'H'`	0	All Unicode	Quote & ASCII & Unicode
String	`"hello"`	0	All Unicode	Quote & ASCII & Unicode
Raw string	`r#"hello"#`	0 or more*	All Unicode	`N/A`
Byte	`b'H'`	0	All ASCII	Quote & Byte

	Example	`#` sets	Characters	Escapes
Byte string	`b"hello"`	0	All ASCII	Quote & Byte
Raw byte string	`br#"hello"#`	0 or more*	All ASCII	`N/A`

* The number of `#` s on each side of the same literal must be equivalent

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 *	Example	Exponentiation	Suffixes
Decimal integer	`98_222`	`N/A`	Integer suffixes
Hex integer	`0xff`	`N/A`	Integer suffixes
Octal integer	`0o77`	`N/A`	Integer suffixes
Binary integer	`0b1111_0000`	`N/A`	Integer suffixes
Floating-point	`123.0E+77`	`Optional`	Floating-point suffixes

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

Suffixes

A suffix is a non-raw identifier immediately (without whitespace) following the primary part of a literal.

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

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

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

However, suffixes on literal tokens parsed as Rust code 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

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) `'`

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
)* "

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 are allowed in string literals. A line-break is either a newline (`U+000A`) or a pair of carriage return and newline (`U+000D` , `U+000A`). Both byte sequences are normally translated to `U+000A` , but as a special exception, when an unescaped `U+005C` character (\ ) occurs immediately before the line-break, then the `U+005C` character, the line-break, and all whitespace at the beginning of the next line are ignored. Thus `a` and `b` are equal:

```rust
let a = "foobar";
let b = "foo\
         bar";

assert_eq!(a,b);
```

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

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 zero or more of the character U+0023 (#) and a U+0022 (double-quote) character. The *raw string body* can contain any sequence of Unicode characters and 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) `'`
>
> 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)* `"`
>
> 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. The type of a byte string literal of length `n` is `&'static [u8; n]` .

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

RAW_BYTE_STRING_CONTENT :
    `"` ASCII$^{*\ \text{(non-greedy)}}$ `"`
  | `#` RAW_BYTE_STRING_CONTENT `#`

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

Raw byte string literals do not process any escapes. They start with the character `U+0062` (`b`), followed by `U+0072` (`r`), followed by zero or more of the character `U+0023` (`#`), and a `U+0022` (double-quote) character. The *raw string body* can contain any sequence of ASCII characters and 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
```

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)
INTEGER_SUFFIX?

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`]

INTEGER_SUFFIX :
```
   u8 | u16 | u32 | u64 | u128 | usize
 | i8 | i16 | i32 | i64 | i128 | isize
```

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 an *integer suffix*, which forcibly sets the type of the literal. The integer suffix must be the name of one of the integral types: `u8` , `i8` , `u16` , `i16` , `u32` , `i32` , `u64` , `i64` , `u128` , `i128` , `usize` , or `isize` .

The type of an *unsuffixed* integer literal is determined by type inference:

- If an integer type can be *uniquely* determined from the surrounding program context, the unsuffixed integer literal 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 literals of various forms:

```
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
0b________1;                      // type i32

0usize;                           // type usize
```

Examples of invalid integer literals:

```
// invalid suffixes

0invalidSuffix;

// uses numbers of the wrong base

123AFB43;
0b0102;
0o0581;

// integers too big for their type (they overflow)

128_i8;
256_u8;

// bin, hex, and octal literals must have at least one digit

0b_;
0b____;
```

Note that the Rust syntax considers `-1i8` as an application of the unary minus operator to an integer literal `1i8`, rather than a single integer literal.

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**: The tuple index may include an `INTEGER_SUFFIX`, 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 identifier)*
 | DEC_LITERAL FLOAT_EXPONENT
 | DEC_LITERAL `.` DEC_LITERAL FLOAT_EXPONENT?
 | DEC_LITERAL (`.` DEC_LITERAL)? FLOAT_EXPONENT? FLOAT_SUFFIX

FLOAT_EXPONENT :
 (`e` | `E`) (`+` | `-`)? (DEC_DIGIT | `_`)* DEC_DIGIT (DEC_DIGIT | `_`)*

FLOAT_SUFFIX :
 `f32` | `f64`

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 forcibly sets the type of the literal. There are two valid *floating-point suffixes*, `f32` and `f64` (the 32-bit and 64-bit floating point types), which explicitly determine the type of the literal.

The type of an *unsuffixed* floating-point literal is determined by type inference:

- If a floating-point type can be *uniquely* determined from the surrounding program context, the unsuffixed floating-point literal 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 literals of various forms:

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

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`.

The representation semantics of floating-point numbers are described in "Machine Types".

Boolean literals

Lexer

BOOLEAN_LITERAL :
    `true`
  | `false`

The two values of the boolean type are written `true` and `false`.

Lifetimes and loop labels

Lexer

LIFETIME_TOKEN :
    `'` IDENTIFIER_OR_KEYWORD
  | `'_`

LIFETIME_OR_LABEL :
    `'` NON_KEYWORD_IDENTIFIER

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
`/`	Slash	Division
`%`	Percent	Remainder
`^`	Caret	Bitwise and Logical XOR
`!`	Not	Bitwise and Logical NOT, Macro Calls, Inner Attributes, Never Type
`&`	And	Bitwise and Logical AND, Borrow, References, Reference patterns
`|`	Or	Bitwise and Logical OR, Closures, Match
`&&`	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
`==`	EqEq	Equal

Symbol	Name	Usage
`!=`	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
`.`	Dot	Field access, Tuple index
`..`	DotDot	Range, Struct expressions, 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
`=>`	FatArrow	Match arms, Macros
`#`	Pound	Attributes
`$`	Dollar	Macros
`?`	Question	Question mark operator, Questionably sized, Macro Kleene Matcher

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

Bracket	Type
`[ ]`	Square brackets
`( )`	Parentheses

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 :
    `<` `>`
  | `<` *GenericArgsLifetimes* `,`? `>`
  | `<` *GenericArgsTypes* `,`? `>`
  | `<` *GenericArgsBindings* `,`? `>`
  | `<` *GenericArgsTypes* `,` *GenericArgsBindings* `,`? `>`
  | `<` *GenericArgsLifetimes* `,` *GenericArgsTypes* `,`? `>`
  | `<` *GenericArgsLifetimes* `,` *GenericArgsBindings* `,`? `>`
  | `<` *GenericArgsLifetimes* `,` *GenericArgsTypes* `,` *GenericArgsBindings* `,`? `>`

GenericArgsLifetimes :
  Lifetime (`,` *Lifetime*)*

GenericArgsTypes :
  Type (`,` *Type*)*

GenericArgsBindings :
  GenericArgsBinding (`,` *GenericArgsBinding*)*

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);
```

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*? `)` (`->` *Type*)?

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 the crate root. Each identifier in the path must resolve to an item.

> **Edition Differences**: In the 2015 Edition, the crate root contains a variety of different items, including external crates, default crates such as `std` and `core`, and items in the top level of the crate (including `use` imports).
>
> Beginning with the 2018 Edition, paths starting with `::` can only reference crates.

```
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 `::`.

```
fn foo() {}
fn bar() {
    self::foo();
}
```

`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 `::`.

```
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.

```
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 { // ::a
    pub struct Struct; // ::a::Struct

    pub trait Trait { // ::a::Trait
        fn f(&self); // ::a::Trait::f
    }

    impl Trait for Struct {
        fn f(&self) {} // <::a::Struct as ::a::Trait>::f
    }

    impl Struct {
        fn g(&self) {} // <::a::Struct>::g
    }
}

mod without { // ::without
    fn canonicals() { // ::without::canonicals
        struct OtherStruct; // None

        trait OtherTrait { // None
            fn g(&self); // None
        }

        impl OtherTrait for OtherStruct {
            fn g(&self) {} // None
        }

        impl OtherTrait for ::a::Struct {
            fn g(&self) {} // None
        }

        impl ::a::Trait for OtherStruct {
            fn f(&self) {} // None
        }
    }
}
```

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 can be used to implement custom derive.

Macro Invocation

> **Syntax**
>
> *MacroInvocation* :
>   *SimplePath* `!` *DelimTokenTree*
>
> *DelimTokenTree* :
>   `(` *TokenTree** `)`
> | `[` *TokenTree** `]`
> | `{` *TokenTree** `}`
>
> *TokenTree* :
>   *Token*$_{\text{except delimiters}}$ | *DelimTokenTree*
>
> *MacroInvocationSemi* :
>   *SimplePath* `!` `(` *TokenTree** `)` `;`
> | *SimplePath* `!` `[` *TokenTree** `]` `;`
> | *SimplePath* `!` `{` *TokenTree** `}`

A macro invocation executes 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$_{\text{except \$ and delimiters}}$
 | *MacroMatcher*
 | `$` IDENTIFIER `:` *MacroFragSpec*
 | `$` `(` *MacroMatch*$^+$ `)` *MacroRepSep*$^?$ *MacroRepOp*

MacroFragSpec :
   `block` | `expr` | `ident` | `item` | `lifetime` | `literal`
 | `meta` | `pat` | `path` | `stmt` | `tt` | `ty` | `vis`

MacroRepSep :
 Token$_{\text{except delimiters and repetition operators}}$

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.

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` : a *Pattern*
- `expr` : an *Expression*
- `ty` : a *Type*
- `ident` : an IDENTIFIER_OR_KEYWORD
- `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.

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 occurrences.

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. Second, 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:

```rust
#[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 prelude of the crate, 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.

```rust
#[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:

```rust
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.

```rust
//// 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:

```rust
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.

```rust
#[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` 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.

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. You can modify the

`Span` of any 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.

```rust
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 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 expand to a type or any number of items, including `macro_rules` definitions. They may be invoked in a type expression, item position (except as a statement), including items in `extern` blocks, inherent and trait implementations, and trait definitions. They cannot be used in a statement, expression, or pattern.

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`.

```
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:

```
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.

```
#[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:

```
#[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
}
```

```rust
// 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() {}"
```

Crates and source files

Syntax

Crate :
  UTF8BOM$^?$
  SHEBANG$^?$
  InnerAttribute*
  Item*

Lexer

UTF8BOM : `\uFEFF`

SHEBANG : `#!` ~[`\n`]* `\n`

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.

```
// 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)]
```

The optional *UTF8 byte order mark* (UTF8BOM production) indicates that the file is encoded in UTF8. It can only occur at the beginning of the file and is ignored by the compiler.

A source file can have a *shebang* (SHEBANG production), which indicates to the operating system what program to use to execute this file. It serves essentially to treat the source file as an executable script. The shebang can only occur at the beginning of the file (but after the optional *UTF8BOM*). It is ignored by the compiler. For example:

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

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

Preludes and `no_std`

All crates have a *prelude* that automatically inserts names from a specific module, the *prelude module*, into scope of each module and an `extern crate` into the crate root module. By default, the *standard prelude* is used. The linked crate is `std` and the prelude module is `std::prelude::v1` .

The prelude can be changed to the *core prelude* by using the `no_std` attribute on the root crate module. The linked crate is `core` and the prelude module is `core::prelude::v1` . 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.

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 be one of the following:

- `()`
- `Result<(), E> where E: Error`

Note: The implementation of which return types are allowed is determined by the unstable `Termination` trait.

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+002D) 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.

⚠ 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"`

`target_family`

Key-value option set at most once with the target's operating system value.

Example values:

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

`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_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. For example, for targets with 32-bit pointers, this is set to `"32"`. Likewise, it is set to `"64"` for targets with 64-bit pointers.

`target_vendor`

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

Example values:

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

`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.

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.

Some examples on functions:

```
// 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() {
  // ...
}
```

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.

```rust
#[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:

```rust
#[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 :
   `mod` IDENTIFIER `;`
 | `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.

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**: Previous 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

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;
}
```

Prelude Items

Modules implicitly have some names in scope. These name are to built-in types, macros imported with `#[macro_use]` on an extern crate, and by the crate's prelude. These names are all made of a single identifier. These names are not part of the module, so for example, any name `name`, `self::name` is not a valid path. The names added by the prelude can be removed by placing the `no_implicit_prelude` attribute onto the module or one of its ancestor modules.

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. 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 `crateid` provided against the `crateid` attributes that were declared on the external crate when it was compiled. If no `crateid` 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:

```rust
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

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". Crates in the extern prelude are in scope in the entire crate, including inner modules. If imported with `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.

The `no_implicit_prelude` attribute can be used on a module to disable prelude lookups within that module.

> **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 `proc_macro`, `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 proc_macro;
> use proc_macro::TokenStream;
> ```

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 import 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::option::Option::{Some, None};
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() {
    // 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:

```rust
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:

```
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 *Generics*?
    (*FunctionParameters*?)
    FunctionReturnType? *WhereClause*?
    BlockExpression

FunctionQualifiers :
  AsyncConstQualifiers? `unsafe`? (`extern` *Abi*?)?

AsyncConstQualifiers :
  `async` | `const`

Abi :
  STRING_LITERAL | RAW_STRING_LITERAL

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

FunctionParam :
  *OuterAttribute** *Pattern* `:` *Type*

FunctionReturnType :
  `->` *Type*

A *function* consists of a block, along with a name and a set of parameters. 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.

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;
}
```

As with `let` bindings, function arguments are irrefutable patterns, so any pattern that is valid in a let binding is also valid as an argument:

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

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
};
```

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 in Rust 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. 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.

If a const function is called outside a const context, it is indistinguishable from any other function. You can freely do anything with a const function that you can do with a regular function.

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.

Exhaustive list of permitted structures in const functions:

> **Note**: this list is more restrictive than what you can write in regular constants

- Type parameters where the parameters only have any trait bounds of the following kind:
    - lifetimes
    - `Sized` or `?Sized`

  This means that `<T: 'a + ?Sized>`, `<T: 'b + Sized>`, and `<T>` are all permitted.

  This rule also applies to type parameters of impl blocks that contain const methods.

  This does not apply to tuple struct and tuple variant constructors.

- Arithmetic and comparison operators on integers
- All boolean operators except for `&&` and `||` which are banned since they are short-circuiting.
- Any kind of aggregate constructor (array, `struct`, `enum`, tuple, ...)
- Calls to other *safe* const functions (whether by function call or method call)
- Index expressions on arrays and slices
- Field accesses on structs and tuples
- Reading from constants (but not statics, not even taking a reference to a static)
- `&` and `*` (only dereferencing of references, not raw pointers)
- Casts except for raw pointer to integer casts
- `unsafe` blocks and `const unsafe fn` are allowed, but the body/block may only do the following unsafe operations:
    - calls to const unsafe functions

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:

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

is roughly equivalent to:

```
// 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 callers 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 will only be available while running tests.

```
fn test_only() {
    #![test]
}
```

> 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 *Generics*? *WhereClause*? `=` *Type* `;`

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 an enum type cannot be used to qualify the constructors:

```
enum E { A }
type F = E;
let _: F = E::A;  // OK
// let _: F = F::A;  // Doesn't work
```

Structs

Syntax

Struct :
  StructStruct
  | *TupleStruct*

StructStruct :
  `struct` IDENTIFIER *Generics*? *WhereClause*? (`{` *StructFields*? `}` | `;`)

TupleStruct :
  `struct` IDENTIFIER *Generics*? `(` *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:

```rust
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:

```rust
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:

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

is equivalent to

```rust
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 *Generics*? *WhereClause*? `{` *EnumItems*? `}`

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

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

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

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

EnumItemDiscriminant :
  `=` *Expression*

An *enumeration*, also referred to as *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:

```rust
enum Animal {
    Dog,
    Cat,
}

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

Enum constructors can have either named or unnamed fields:

```rust
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. Each enum instance has a *discriminant* which is an integer associated to it that is used to determine which variant it holds. An opaque reference to this discriminant can be obtained with the `mem::discriminant` function.

Custom Discriminant Values for Fieldless Enumerations

If there is no data attached to *any* of the variants of an enumeration, then the discriminant can be directly chosen and accessed.

These enumerations can be cast to integer types with the `as` operator by a numeric cast. The enumeration can optionally specify which integer each discriminant gets by following the variant name with `=` followed by a constant expression. If the first variant in the declaration is unspecified, then it is set to zero. For every other unspecified discriminant, it is set to one higher than the previous variant in the declaration.

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

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

Under the default representation, the specified discriminant is interpreted as an `isize` value although the compiler is allowed to use a smaller type in the actual memory layout. The size and thus acceptable values can be changed by using a primitive representation or the `C` representation.

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.
}
```

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 *Generics*? *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.

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:

```rust
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:

```rust
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 at 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 `#[repr(C)]` 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` at type `bool` is undefined behavior. Effectively, writing to and then reading from a `#[repr(C)]` union 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:

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

Writes to `Copy` union fields do not require reads for running destructors, so these writes don't have to be placed in `unsafe` blocks

```rust
u.f1 = 2;
```

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

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.

```rust
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. 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 it contains 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,
};
```

Constants with Destructors

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

```rust
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 _: () = ();
```

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. Non-`mut` static items that contain a type that is not interior mutable may be placed in read-only memory. Static items do not call `drop` at the end of the program.

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.
- Statics allow using paths to statics in the constant expression used to initialize them, but statics may not refer to other statics by value, only through a reference.
- Constants cannot refer to statics.

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.

```rust
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 *Generics*? (`:` *TypeParamBounds*?)?
WhereClause? `{`
   *TraitItem**
  `}`

TraitItem :
  *OuterAttribute** *Visibility*? (
      TraitFunc
    | *TraitMethod*
    | *TraitConst*
    | *TraitType*
    | *MacroInvocationSemi*
)

TraitFunc :
   TraitFunctionDecl (`;` | *BlockExpression*)

TraitMethod :
   TraitMethodDecl (`;` | *BlockExpression*)

TraitFunctionDecl :
  FunctionQualifiers `fn` IDENTIFIER *Generics*?
    `(` *TraitFunctionParameters*? `)`
   FunctionReturnType? *WhereClause*?

TraitMethodDecl :
  FunctionQualifiers `fn` IDENTIFIER *Generics*?
    `(` *SelfParam* (`,` *TraitFunctionParam*)* `,`? `)`
    FunctionReturnType? *WhereClause*?

TraitFunctionParameters :
  TraitFunctionParam (`,` *TraitFunctionParam*)* `,`?

TraitFunctionParam† :
  *OuterAttribute** (*Pattern* `:`)? *Type*

TraitConst :
  `const` IDENTIFIER `:` *Type* (`=` *Expression*)? `;`

TraitType :
  `type` IDENTIFIER (`:` *TypeParamBounds*?)? `;`

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.

Items associated with a trait do not need to be defined in the trait, but they may be. If the trait provides a definition, then this definition acts as a default for any implementation which does not override it. If it does not, then any implementation must provide a definition.

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):

- It must not require `Self: Sized`
- All associated functions must either have a `where Self: Sized` bound, or
  - Not have any type parameters (although lifetime parameters are allowed), and
  - Be a method that does not use `Self` except in the type of the receiver.
- It must not have any associated constants.
- All supertraits must also be object safe.

When there isn't a `Self: Sized` bound on a method, the type of a method receiver must be one of the following types:

- `&Self`
- `&mut Self`
- `Box<Self>`
- `Rc<Self>`
- `Arc<Self>`
- `Pin<P>` where `P` is one of the types above

```rust
// 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>>) {}
}
```

```rust
// 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:

```
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` *Generics*? *Type* *WhereClause*? `{`
    *InnerAttribute**

> *InherentImplItem**
> `}`
>
> *InherentImplItem* :
> *OuterAttribute** (
> *MacroInvocationSemi*
> | (*Visibility*? (*ConstantItem* | *Function* | *Method*))
>)
>
> *TraitImpl* :
> `unsafe`? `impl` *Generics*? `!`? *TypePath* `for` *Type*
> *WhereClause*?
> `{`
> *InnerAttribute**
> *TraitImplItem**
> `}`
>
> *TraitImplItem* :
> *OuterAttribute** (
> *MacroInvocationSemi*
> | (*Visibility*? (*TypeAlias* | *ConstantItem* | *Function* | *Method*))
>)

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 is 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 type and lifetime parameters, which can be used in the rest of the implementation. Type parameters declared for an implementation must be used at least once in either the trait or the implementing type of an 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 */
}
```

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. That attributes that have meaning here are `cfg`, `deprecated`, `doc`, and the lint check attributes.

External blocks

Syntax

ExternBlock :
  `extern` *Abi*? `{`
   *InnerAttribute**
   *ExternalItem**
  `}`

ExternalItem :
  *OuterAttribute** (
    MacroInvocationSemi
   | (*Visibility*? (*ExternalStaticItem* | *ExternalFunctionItem*))
)

ExternalStaticItem :
  `static` `mut`? IDENTIFIER `:` *Type* `;`

ExternalFunctionItem :
  `fn` IDENTIFIER *Generics*?
  `(` (*NamedFunctionParameters* | *NamedFunctionParametersWithVariadics*)? `)`
  FunctionReturnType? *WhereClause*? `;`

NamedFunctionParameters :
  NamedFunctionParam (`,` *NamedFunctionParam*)* `,`?

NamedFunctionParam :
  *OuterAttribute** (IDENTIFIER | `_`) `:` *Type*

NamedFunctionParametersWithVariadics :

(*NamedFunctionParam* `,`)* *NamedFunctionParam* `,` *OuterAttribute** `...`

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 kind 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.

Functions

Functions within external blocks are declared in the same way as other Rust functions, with the exception that they may not have a body and are instead terminated by a semicolon. Patterns are not allowed in parameters, only IDENTIFIER or `_` may be used.

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))`

Variadic functions

Functions within external blocks may be variadic by specifying `...` after one or more named arguments in the argument list:

```
extern {
    fn foo(x: i32, ...);
}
```

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.

The `name` key must be included if `kind` is specified.

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.

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();
}
```

Attributes on function parameters

Attributes on extern function parameters follow the same rules and restrictions as regular function parameters.

Type and Lifetime Parameters

Syntax

Generics :

`<` *GenericParams* `>`

GenericParams :
  LifetimeParams
  | (*LifetimeParam* `,`)* *TypeParams*

LifetimeParams :
  (*LifetimeParam* `,`)* *LifetimeParam*?

LifetimeParam :
  OuterAttribute? LIFETIME_OR_LABEL (`:` *LifetimeBounds*)?

TypeParams:
  (*TypeParam* `,`)* *TypeParam*?

TypeParam :
  OuterAttribute? IDENTIFIER (`:` *TypeParamBounds*?)? (`=` *Type*)?

Functions, type aliases, structs, enumerations, unions, traits, and implementations may be *parameterized* by types 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` . Lifetime parameters must be declared before type parameters. Some examples of items with type and lifetime parameters:

```
fn foo<'a, T>() {}
trait A<U> {}
struct Ref<'a, T> where T: 'a { r: &'a T }
```

References, raw pointers, arrays, slices, tuples, and function pointers have lifetime or type parameters as well, but are not referred to with path syntax.

Where clauses

Syntax

WhereClause :
  `where` (*WhereClauseItem* `,`)* *WhereClauseItem* ?

WhereClauseItem :
    LifetimeWhereClauseItem
  | *TypeBoundWhereClauseItem*

LifetimeWhereClauseItem :
  Lifetime `:` *LifetimeBounds*

TypeBoundWhereClauseItem :
  ForLifetimes? *Type* `:` *TypeParamBounds*?

ForLifetimes :
  `for` `<` *LifetimeParams* `>`

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.

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 defining the item. It is an error to have `Copy` or `Clone` as a bound on a mutable reference, trait object or slice or `Sized` as a bound on a trait object or slice.

```
struct A<T>
where
    T: Iterator,            // Could use A<T: Iterator> instead
    T::Item: Copy,
    String: PartialEq<T>,
    i32: Default,           // Allowed, but not useful
    i32: Iterator,          // Error: the trait bound is not satisfied
    [T]: Copy,              // Error: the trait bound is not satisfied
{
    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

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

Method :
  FunctionQualifiers `fn` IDENTIFIER *Generics*?
    `(` *SelfParam* (`,` *FunctionParam*)* `,`? `)`
    FunctionReturnType? *WhereClause*?
    BlockExpression

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

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

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

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 as `type`, then an identifier, and finally an optional list of trait bounds.

The identifier is the name of the declared type alias. The optional trait bounds must be fulfilled by the implementations of the type alias.

An *associated type definition* defines a type alias on another type. It is written as `type`, then an identifier, then an `=`, and finally a type.

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.

```
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();
}
```

Associated Types Container Example

Consider the following example of a `Container` trait. Notice that the type is available for use in the method signatures:

```rust
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`:

```rust
impl<T> Container for Vec<T> {
    type E = T;
    fn empty() -> Vec<T> { Vec::new() }
    fn insert(&mut self, x: T) { self.push(x); }
}
```

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 Constants Examples

A basic example:

```rust
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:

```rust
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);
}
```

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 in Rust 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:

```rust
// 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_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:

```
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.

Attributes

> **Syntax**
>
> *InnerAttribute* :
>   `#` `!` `[` *Attr* `]`
>
> *OuterAttribute* :
>   `#` `[` *Attr* `]`
>
> *Attr* :

SimplePath AttrInput$^?$

AttrInput :
   DelimTokenTree
   | `=` *LiteralExpression*$_{\text{without suffix}}$

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 a literal 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:

```
// 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* `=` *LiteralExpression*$_{\textit{without suffix}}$

| *SimplePath* `(` *MetaSeq*? `)`

MetaSeq :
MetaItemInner (`,` MetaItemInner)* `,`?

MetaItemInner :
MetaItem
| *LiteralExpression*$_{without\ suffix}$

Literal expressions in meta items must not include integer or float type suffixes.

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)`

Style	Example
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. 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.

```rust
// 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.
  - `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.
- 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.

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 be one of the following:

- `()`
- `Result<(), E> where E: Error`

> Note: The implementation of which return types are allowed is determined by the unstable `Termination` trait.

> Note: The test mode is enabled by passing the `--test` argument to `rustc` or using `cargo test`.

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 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
    }

    fn ne(&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.

```rust
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 }
}
```

This example shows how one can use `allow` and `warn` to toggle a particular check on and off:

```rust
#[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:

```rust
#[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 }
}
```

Tool lint attributes

Tool lints allows using scoped lints, to `allow`, `warn`, `deny` or `forbid` lints of certain tools.

Currently `clippy` is the only available lint tool.

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() {
    // ...
}
```

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", 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 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 an unsafe 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.

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`

Feature	Implicitly Enables	Description
`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
`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

Feature	Implicitly Enables	Description
`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`	`sse3`	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
`xsaveopt`		`xsaveopt` — Save processor extended states optimized
`xsaves`		`xsaves` — Save processor extended states supervisor

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` macro in the standard library for runtime feature detection on the x86 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.

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-derefence.
(|_: &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 = "8"]

fn f<T>(x: T) {}

// This fails to compile because monomorphizing to
// `f::<(i32, i32, i32, i32, i32, i32, i32, i32, i32)>>` requires more
// than 8 type elements.
f((1, 2, 3, 4, 5, 6, 7, 8, 9));
```

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 in an *EnumerationVariantExpression*.

```
// `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 => { },
}
```

Non-exhaustive types are always considered inhabited in downstream crates.

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 in Rust 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` *Pattern* (`:` *Type*)? (`=` *Expression*)? `;`

A `let` *statement* introduces a new set of variables, given by an irrefutable pattern. The pattern is followed optionally by a type annotation and then optionally by an initializer expression. 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.

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*
    | *EnumerationVariantExpression*
    | *CallExpression*
    | *MethodCallExpression*
    | *FieldExpression*
    | *ClosureExpression*
    | *ContinueExpression*
    | *BreakExpression*
    | *RangeExpression*
    | *ReturnExpression*
    | *MacroInvocation*
)

ExpressionWithBlock :
  *OuterAttribute**†
  (
      BlockExpression
    | *AsyncBlockExpression*
    | *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 (operands). The meaning of each kind of expression dictates several things:

- Whether or not to evaluate the sub-expressions when evaluating the expression
- The order in which to evaluate the sub-expressions
- How to combine the sub-expressions' 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.

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
`+` `-`	left to right
`<<` `>>`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`==` `!=` `<` `>` `<=` `>=`	Require parentheses

Operator/Expression	Associativity
`&&`	left to right
`||`	left to right
`..` `..=`	Require parentheses
`=` `+=` `-=` `*=` `/=` `%=` `&=` `|=` `^=` `<<=` `>>=`	right to left
`return` `break` closures	

Place Expressions and Value Expressions

Expressions are divided into two main categories: place expressions and value expressions. Likewise within each expression, sub-expressions may 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 an assignment or compound assignment expression.
- The operand of a unary borrow 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*.

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 doesn't implement `Drop`.
- The result of dereferencing an expression with type `Box<T>` and that can also be moved out of.

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 is a mutable place expression context.
- Array indexing of a type that implements `DerefMut`, 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 initialized to that value and 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 it's operands:

```
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 and enum variant 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
 | INTEGER_LITERAL
 | FLOAT_LITERAL
 | BOOLEAN_LITERAL

A *literal expression* consists of one of the literal forms described earlier. It directly describes a number, character, string, or boolean value.

```
"hello";   // string type
'5';       // character type
5;         // integer type
```

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.

Blocks are written as `{`, then any inner attributes, then statements, then an optional expression, and finally a `}`. Statements are usually required to be followed by a semicolon, with two exceptions. Item declaration statements do not need to be followed by a semicolon. 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 expression is executed, if given.

The type of a block is the type of the final expression, or `()` if the final expression is omitted.

```rust
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 expression in value expression context. 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 a closure 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:

```rust
loop {
    async move {
        break; // This would break out of the loop.
    }
}
```

`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() };
```

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 and enum variant 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 `-` create a value greater than the maximum value, or less than the minimum value that can be stored. This includes unary `-` on the smallest value of any signed integer type.
- Using `/` or `%`, where the left-hand argument is the smallest integer of a signed integer type and the right-hand argument is `-1`.
- 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.

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;
```

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` and `*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

```rust
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 Tr**
`+`	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::BitA`
`|`	Bitwise OR	Logical OR		`std::ops::BitO`
`^`	Bitwise XOR	Logical XOR		`std::ops::BitX`
`<<`	Left Shift			`std::ops::Shl`
`>>`	Right Shift**			`std::ops::Shr`

* Integer division rounds towards zero.

** Arithmetic right shift on signed integer types, logical right shift on unsigned integer types.

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 type 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 the operators the traits for 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. Here `*T` means either `*const T` or `*mut T`.

Type of `e`	`U`	Cast performed by `e as U`
Integer or Float type	Integer or Float type	Numeric cast
C-like enum	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`	Numeric type	Pointer to address cast
Integer type	`*V` where `V: Sized`	Address to pointer cast
`&[T; n]`	`*const T`	Array to pointer 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 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
  - 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 will saturate to the maximum value of the integer type.
    - Values smaller than the minimum integer value 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
- Enum cast
    - Casts an enum to its discriminant, then uses a numeric cast if needed.
- 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.

* 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.

Assignment expressions

Syntax

AssignmentExpression :
  Expression `=` *Expression*

An *assignment expression* consists of a place expression followed by an equals sign (`=`) and a value expression. Such an expression always has the `unit` type.

Evaluating an assignment expression drops the left-hand operand, unless it's an uninitialized local variable or field of a local variable, and either copies or moves its right-hand operand to its left-hand operand. The left-hand operand must be a place expression: using a value expression results in a compiler error, rather than promoting it to a temporary.

```
x = y;
```

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*

The `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, and `>>` operators may be composed with the `=` operator. The expression `place_exp OP= value` is equivalent to `place_expr = place_expr OP val`. For example, `x = x + 1` may be written as `x += 1`. Any such expression always has the `unit` type. These operators can all be overloaded using the trait with the same name as for the normal operation followed by 'Assign', for example, `std::ops::AddAssign` is used to overload `+=`. As with `=`, `place_expr` must be a place expression.

```
let mut x = 10;
x += 4;
assert_eq!(x, 14);
```

Grouped expressions

Syntax

GroupedExpression :
    `(` *InnerAttribute** *Expression* `)`

An expression enclosed in parentheses evaluates to the result of the enclosed expression. Parentheses can be used to explicitly specify evaluation order within an expression.

An example of a parenthesized expression:

```
let x: i32 = 2 + 3 * 4;
let y: i32 = (2 + 3) * 4;
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");
```

Group expression attributes

Inner attributes are allowed directly after the opening parenthesis of a group expression in the same expression contexts as attributes on block expressions.

Array and array index expressions

Array expressions

Syntax

ArrayExpression :

`[` *InnerAttribute** *ArrayElements*? `]`

ArrayElements :
  Expression (`,` *Expression*)* `,`?
| *Expression* `;` *Expression*

An *array expression* can be written by enclosing zero or more comma-separated expressions of uniform type in square brackets. This produces an array containing each of these values in the order they are written.

Alternatively there can be exactly two expressions inside the brackets, separated by a semi-colon. The expression after the `;` must be a have type `usize` and be a constant expression, such as a literal or a constant item. `[a; b]` creates an array containing `b` copies of the value of `a`. If the expression after the semi-colon has a value greater than 1 then this requires that the type of `a` is `Copy`.

```
[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
```

Array expression attributes

Inner attributes are allowed directly after the opening bracket of an array expression in the same expression contexts as attributes on block expressions.

Array and slice indexing expressions

Syntax

IndexExpression :
  Expression `[` *Expression* `]`

Array and slice-typed expressions 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 :
   `(` *InnerAttribute** *TupleElements*? `)`

TupleElements :
  (*Expression* `,`)$^{+}$ *Expression*$^{?}$

Tuples are written by enclosing zero or more comma-separated expressions in parentheses. They are used to create tuple-typed values.

```rust
(0.0, 4.5);
("a", 4usize, true);
();
```

You can disambiguate a single-element tuple from a value in parentheses with a comma:

```rust
(0,); // single-element tuple
(0); // zero in parentheses
```

Tuple expression attributes

Inner attributes are allowed directly after the opening parenthesis of a tuple expression in the same expression contexts as attributes on block expressions.

Tuple indexing expressions

Syntax

TupleIndexingExpression :
  Expression `.` TUPLE_INDEX

Tuples and struct tuples can be indexed using the number corresponding to the position of the field. The index must be written as a decimal literal with no underscores or suffix. Tuple indexing expressions also differ from field expressions in that they can unambiguously be called as a function. In all other aspects they have the same behavior.

```
let pair = (1, 2);
assert_eq!(pair.1, 2);
let unit_x = Point(1.0, 0.0);
assert_eq!(unit_x.0, 1.0);
```

Struct expressions

Syntax

StructExpression :
    StructExprStruct
  | *StructExprTuple*
  | *StructExprUnit*

StructExprStruct :
  PathInExpression `{` *InnerAttribute** (*StructExprFields* | *StructBase*)? `}`

StructExprFields :
  StructExprField (`,` *StructExprField*)* (`,` *StructBase* | `,` ?)

StructExprField :
    IDENTIFIER
  | (IDENTIFIER | TUPLE_INDEX) `:` *Expression*

StructBase :
  `..` *Expression*

StructExprTuple :
  PathInExpression `(`
    *InnerAttribute**
    (*Expression* (`,` *Expression*)* `,` ?)?
  `)`

StructExprUnit : *PathInExpression*

A *struct expression* creates a struct or union value. It consists of a path to a struct 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 also be created using this syntax, except that it must specify exactly one field.

Functional update syntax

A struct expression 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 in 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`.
```

Struct expression attributes

Inner attributes are allowed directly after the opening brace or parenthesis of a struct expression in the same expression contexts as attributes on block expressions.

Enumeration Variant expressions

Syntax

EnumerationVariantExpression :
    EnumExprStruct
  | *EnumExprTuple*
  | *EnumExprFieldless*

EnumExprStruct :
  PathInExpression `{` *EnumExprFields*? `}`

EnumExprFields :
    EnumExprField (`,` *EnumExprField*)* `,` ?

EnumExprField :
    IDENTIFIER
  | (IDENTIFIER | TUPLE_INDEX) `:` *Expression*

EnumExprTuple :
  PathInExpression `(`
    (*Expression* (`,` *Expression*)* `,` ?)?
  `)`

EnumExprFieldless : *PathInExpression*

Enumeration variants can be constructed similarly to structs, using a path to an enum variant instead of to a struct:

```
let q = Message::Quit;
let w = Message::WriteString("Some string".to_string());
let m = Message::Move { x: 50, y: 200 };
```

Enum variant expressions have the same syntax, behavior, and restrictions as struct expressions, except they do not support base update with the `..` syntax.

Call expressions

Syntax

CallExpression :
  Expression `(` *CallParams*? `)`

CallParams :
  Expression (`,` *Expression*)* `,`?

A *call expression* consists of an expression followed by a parenthesized expression-list. It invokes a function, providing zero or more input variables. 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. Rust will also automatically dereference `f` as required. Some examples of call expressions:

```
let three: i32 = add(1i32, 2i32);
let name: &'static str = (|| "Rust")();
```

Disambiguating Function Calls

Rust treats all function calls as sugar for a more explicit, fully-qualified syntax. Upon compilation, Rust will desugar all function calls into the explicit form. Rust may sometimes require you to qualify function calls with trait, depending on the ambiguity of a call in light of in-scope items.

> **Note**: In the past, the Rust community used the terms "Unambiguous Function Call Syntax", "Universal Function Call Syntax", or "UFCS", in documentation, issues, RFCs, and other community writings. However, the term lacks descriptive power and potentially confuses the issue at hand. We mention it 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.

```rust
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.

⚠ ***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 a trait method and you'll be fine.

Field access expressions

Syntax

FieldExpression :
  Expression `.` IDENTIFIER

A *field expression* consists of an expression followed by a single dot and an identifier, when not immediately followed by a parenthesized expression-list (the latter is always a method call expression). A field expression denotes a field of a struct or union. To call a function stored in a struct, parentheses are needed around the field expression.

```
mystruct.myfield;
foo().x;
(Struct {a: 10, b: 20}).a;
mystruct.method();          // Method expression
(mystruct.function_field()) // Call expression containing a field
expression
```

A field access is a place expression referring to the location of that field. When the subexpression is mutable, the field expression is also mutable.

Also, if the type of the expression to the left of the dot is a pointer, it is automatically dereferenced as many times as necessary to make the field access possible. In cases of ambiguity, we prefer fewer autoderefs to more.

Finally, 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.

```
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** *Pattern* (`:` *Type*)?

A *closure expression*, also know as a lambda expression or a lambda, defines a closure and denotes it as a value, in a single expression. A closure expression is a pipe-symbol-delimited (`|`) list of irrefutable patterns followed by an expression. Type annotations may optionally be added for the type of the parameters or for the return type. If there is a return type, the expression used for the body of the closure must be a normal block. A closure expression also may begin with the `move` keyword before the initial `|` .

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 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.

Closure expressions are most useful when passing functions as arguments to other functions, as an abbreviation for defining and capturing a separate function.

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 type is `'static` .

The compiler will determine which of the closure traits the closure's type will implement by how it acts on its captured variables. The closure will also implement

`Send` and/or `Sync` if all of its captured types do. These traits allow functions to accept closures using generics, even though the exact types can't be named.

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

Syntax

LoopExpression :
  LoopLabel? (
      InfiniteLoopExpression
    | *PredicateLoopExpression*
    | *PredicatePatternLoopExpression*
    | *IteratorLoopExpression*
)

Rust supports four 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.

All four types of loop support `break` expressions, `continue` expressions, and labels. Only `loop` supports 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*$_{\text{except struct expression}}$ *BlockExpression*

A `while` loop begins by evaluating the boolean loop conditional expression. If the loop conditional expression evaluates to `true`, the loop body block executes, then control returns to the loop conditional expression. If the loop conditional expression evaluates to `false`, the `while` expression completes.

An example:

```rust
let mut i = 0;

while i < 10 {
    println!("hello");
    i = i + 1;
}
```

Predicate pattern loops

Syntax
PredicatePatternLoopExpression :
  `while` `let` *MatchArmPatterns* `=` *Expression*except struct or 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.

```rust
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*_{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:

```rust
let v = &["apples", "cake", "coffee"];

for text in v {
    println!("I like {}.", text);
}
```

An example of a for loop over a series of integers:

```rust
let mut sum = 0;
for n in 1..11 {
    sum += n;
}
assert_eq!(sum, 55);
```

A for loop is equivalent to the following block expression.

```rust
'label: for PATTERN in iter_expr {
    /* loop body */
}
```

is equivalent to

```rust
{
    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 PAT = 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.

`break` expressions

Syntax
BreakExpression :
  `break` LIFETIME_OR_LABEL? *Expression*?

When `break` is encountered, execution of the associated loop body is immediately terminated, for example:

```rust
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`.

`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:

```rust
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	Ran
RangeExpr	start `..` end	std::ops::Range	star ≤ x end
RangeFromExpr	start `..`	std::ops::RangeFrom	star ≤ x
RangeToExpr	`..` end	std::ops::RangeTo	x < end
RangeFullExpr	`..`	std::ops::RangeFull	-
RangeInclusiveExpr	start `..=` end	std::ops::RangeInclusive	star ≤ x end
RangeToInclusiveExpr	`..=` end	std::ops::RangeToInclusive	x ≤ end

Examples:

```rust
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.

```rust
let x = std::ops::Range {start: 0, end: 10};
let y = 0..10;

assert_eq!(x, y);
```

Ranges can be used in `for` loops:

```rust
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 form of an `if` expression is a condition expression, followed by a consequent block, any number of `else if` conditions and blocks, and an optional trailing `else` block. The condition expressions must have type `bool`. If a condition expression evaluates to `true`, the consequent block is executed and any subsequent `else if` or `else` block is skipped. If a condition expression 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` *MatchArmPatterns* `=` *Expression*_{except struct or 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 expression it expects the keyword `let` followed by a pattern, an `=` and a scrutinee expression. 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.

```rust
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:

```rust
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:

```rust
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` *Expression*$_{\textit{except struct expression}}$ `{`
    *InnerAttribute**

    MatchArms?
   `}`

MatchArms :
  (*MatchArm* `=>` (*ExpressionWithoutBlock* `,` | *ExpressionWithBlock* `,` ?))*
  MatchArm `=>` *Expression* `,` ?

MatchArm :
  *OuterAttribute** *MatchArmPatterns* *MatchArmGuard*?

MatchArmPatterns :
  `|` ? *Pattern* (`|` *Pattern*)*

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:

```rust
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.

```rust
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`, `cold`, 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`

Await expressions are legal only within an async context, like an `async fn` or an `async` block. They operate on a future. Their effect is to suspend the current computation until the given future is ready to produce a value.

More specifically, an `<expr>.await` expression has the following effect.

1. Evaluate `<expr>` to a future `tmp`;
2. Pin `tmp` using `Pin::new_unchecked`;
3. This pinned future is then polled by calling the `Future::poll` method and passing it the current task context;
4. 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 2;
5. 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 `<expr>.await` expression is roughly equivalent to the following (this desugaring is not normative):

```
match /* <expr> */ {
    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.

Patterns

Syntax

Pattern :
      LiteralPattern
   | *IdentifierPattern*
   | *WildcardPattern*
   | *RangePattern*
   | *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 :
   BOOLEAN_LITERAL
 | CHAR_LITERAL
 | BYTE_LITERAL
 | STRING_LITERAL
 | RAW_STRING_LITERAL
 | BYTE_STRING_LITERAL
 | RAW_BYTE_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).

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 (`@` *Pattern*)?

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.

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:

```rust
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.

Wildcard pattern

Syntax

WildcardPattern :
   `_`

The *wildcard pattern* 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:

```rust
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.

Range patterns

Syntax

RangePattern :
  RangePatternBound `..=` *RangePatternBound*
  | *RangePatternBound* `...` *RangePatternBound*

RangePatternBound :
  CHAR_LITERAL
  | BYTE_LITERAL
  | `-`? INTEGER_LITERAL
  | `-`? FLOAT_LITERAL
  | *PathInExpression*
  | *QualifiedPathInExpression*

Range patterns match values that are within the closed range defined by its lower and upper bounds. For example, a pattern `'m'..='p'` will match only the values `'m'`, `'n'`, `'o'`, and `'p'`. The bounds can be literals or paths that point to constant values.

A pattern a `..=` b must always have a ≤ b. It is an error to have a range pattern `10..=0`, for example.

The `...` syntax is kept for backwards compatibility.

Range patterns only work on scalar types. The accepted types are:

- Integer types (u8, i8, u16, i16, usize, isize, etc.).
- Character types (char).
- Floating point types (f32 and f64). This is being deprecated and will not be available in a future version of Rust (see issue #41620).

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!(),
});

// 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 (non-`usize` and -`isize`) 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}'`.

Reference patterns

Syntax
ReferencePattern :
  (`&` | `&&`) `mut`? *Pattern*

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 values that match all criteria defined by its subpatterns. They are also used to destructure a struct.

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 {..} => (),
}
```

If `..` is not used, it is required to match 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: _} => (),
}
```

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 when one of its subpatterns is refutable.

Tuple struct patterns

Syntax

TupleStructPattern :
  PathInExpression `(` *TupleStructItems*? `)`

TupleStructItems :

  Pattern (`,` *Pattern*)* `,`?
| (*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 when one of its subpatterns is refutable.

Tuple patterns

Syntax
TuplePattern :
  `(` *TuplePatternItems*? `)`

TuplePatternItems :
  Pattern `,`
| *Pattern* (`,` *Pattern*)+ `,`?
| (*Pattern* `,`)* `..` (`,` *Pattern*)* `,`?

Tuple patterns match tuple values that match all criteria defined by its subpatterns. They are also used to destructure a tuple.

This pattern is refutable when one of its subpatterns is refutable.

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 :
`[` *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 */ }
};
```

Path patterns

Syntax

PathPattern :
    PathInExpression
  | *QualifiedPathInExpression*

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.

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 — `true` or `false`
  - 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*.

```rust
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.
- Recursive type definitions can cross module boundaries, but not module *visibility* boundaries, or crate boundaries (in order to simplify the module system and type checker).

An example of a *recursive* type and its use:

```rust
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

The `bool` type is a datatype which can be either `true` or `false`. The boolean type uses one byte of memory. It is used in comparisons and bitwise operations like `&`, `|`, and `!`.

```
fn main() {
    let x = true;
    let y: bool = false; // with the boolean type annotation

    // Use of booleans in conditional expressions
    if x {
        println!("x is true");
    }
}
```

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.

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]`, it is 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`.

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.

```
let x: ! = panic!();
// Can be coerced into any type.
let y: u32 = x;
```

NB. The never type was expected to be stabilized in 1.41, but due to some last minute regressions detected the stabilization was temporarily reverted. The `!` type can only appear in function return types presently. See the tracking issue for more details.

Tuple types

Syntax
TupleType :
  `(` `)`
 | `(` (*Type* `,`)$^+$ *Type*$^?$ `)`

A tuple *type* is a heterogeneous product of other types, called the *elements* of the tuple. It has no nominal name and is instead structurally typed.

Tuple types and values are denoted by listing the types or values of their elements, respectively, in a parenthesized, comma-separated list.

Because tuple elements don't have a name, they can only be accessed by pattern-matching or by using `N` directly as a field to access the `N` th element.

An example of a tuple type and its use:

```
type Pair<'a> = (i32, &'a str);
let p: Pair<'static> = (10, "ten");
let (a, b) = p;

assert_eq!(a, 10);
assert_eq!(b, "ten");
assert_eq!(p.0, 10);
assert_eq!(p.1, "ten");
```

For historical reasons and convenience, the tuple type with no elements (`()`) is often called 'unit' or 'the unit type'.

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 an 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]`.

To use a slice type it generally has to be used behind a pointer for example as:

- `&[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', 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 in an enumeration variant 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 or to write to a field that doesn't implement `Copy` . See the item documentation for further details.

The memory layout of a `union` is undefined by default, 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 allows the closure to compile. The choice is made only with regards to the contents of the closure expression; the compiler does not take into account surrounding code, such as the lifetimes of involved variables.

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:

```rust
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 may 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 (`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 in Rust 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 (`&`)

These point to memory *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`)

These also point to memory 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 in Rust code; 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.

Smart Pointers

The standard library contains additional 'smart pointer' types beyond references and raw pointers.

Function pointer types

Syntax

BareFunctionType :
  ForLifetimes? *FunctionQualifiers* `fn`
    `(` *FunctionParametersMaybeNamedVariadic*? `)` *BareFunctionReturnType*?

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:

```
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 optional 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:

- `Trait`
- `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: 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.

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.

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:

```rust
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*

Anonymous type parameters

> Note: This section is a placeholder for more comprehensive reference material.

> Note: This is often called "impl Trait in argument position".

Functions can declare an argument to be an anonymous type parameter where the callee must provide a type that has the bounds declared by the anonymous type parameter and the function can only use the methods available by the trait bounds of the anonymous type parameter.

They are written as `impl` followed by a set of trait bounds.

Abstract return types

> Note: This section is a placeholder for more comprehensive reference material.

> Note: This is often called "impl Trait in return position".

Functions, except for associated trait functions, can return an abstract return type. These types stand in for another concrete type where the use-site may only use the trait methods declared by the trait bounds of the type.

They are written as `impl` followed by a set of trait bounds.

Type parameters

Within the body of an item that has type parameter declarations, the names of its type parameters are types:

```rust
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:

```rust
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 when a bound of `?Sized`. By default any type parameter has a `Sized` bound.
- Traits may be implemented for DSTs. Unlike type parameters `Self: ?Sized` by 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. The size of a value can be checked with the `size_of_val` function.

Types where all values have the same size and alignment 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
`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

Arrays are laid out so that the `nth` element of the array is offset from the start of the array by `n * the size of the type` bytes. An array of `[T; n]` has a size of `size_of::<T>() * n` and the same alignment of `T`.

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 do not have any guarantees about their layout.

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 (`&Trait` , `Box<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:

- 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 Default Representation

Nominal types without a `repr` attribute have the default representation.

Informally, this representation is also called the `rust` representation.

There are no 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.

#[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)] Enums

For C-like enumerations, 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 C-like enumerations 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 C-like enumerations can only legally hold the discriminant values, everything else is undefined behaviour. Therefore, using a C-like enumeration in FFI to model a C `enum` is often wrong.

It is an error for zero-variant enumerations to have the `C` representation.

For all other enumerations, the layout is unspecified.

Likewise, combining the `C` representation with a primitive representation, the layout is unspecified.

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.

For C-like enumerations, they set the size and alignment to be the same as the primitive type of the same name. For example, a C-like enumeration with a `u8` representation can only have discriminants between 0 and 255 inclusive.

It is an error for zero-variant enumerations to have a primitive representation.

For all other enumerations, the layout is unspecified.

Likewise, combining two primitive representations together is unspecified.

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.

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.

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 default 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.

⚠ ***Warning:*** Dereferencing an unaligned pointer is undefined behavior and it is possible to safely create unaligned pointers to `packed` fields. Like all ways to create undefined behavior in safe Rust, this is a bug.

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 in Rust 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 in Rust is very restricted and occurs only due to 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: &(for<'a> Fn(&'a i32) -> &'a i32) = &|x| x;
let supertype: &(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

Type	Variance in `'a`	Variance in `T`
`fn(T) -> ()`		contravariant
`std::cell::UnsafeCell<T>`		invariant
`std::marker::PhantomData<T>`		covariant
`Trait<T> + 'a`	covariant	invariant

The variance of other `struct`, `enum`, `union`, and tuple 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` and `U`.

```
use std::cell::UnsafeCell;
struct Variance<'a, 'b, 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
}
```

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);
    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
    );
}
```

Trait and lifetime bounds are also used to name trait objects.

`?Sized`

`?` is only used to declare that the `Sized` trait may not be implemented for a type parameter or associated type. `?Sized` may not be used as a bound for other types.

Lifetime bounds

Lifetime bounds can be applied to types or other lifetimes. The bound `'a: 'b` is usually read as `'a` *outlives* `'b`. `'a: 'b` means that `'a` lasts longer than `'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

Type bounds may be *higher ranked* over lifetimes. These bounds specify a bound 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 as the lifetime of the reference is shorter than a 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.

```rust
fn call_on_ref_zero<F>(f: F) where F: for<'a> Fn(&'a i32) {
    let zero = 0;
    f(&zero);
}
```

Type coercions

Coercions are defined in RFC 401. RFC 1558 then expanded on that. A coercion is implicit and has no syntax.

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, `42` is coerced to have type `i8` in the following:

  ```rust
  let _: i8 = 42;
  ```

- `static` and `const` statements (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, `42` is coerced to have type `i8` in the following:

  ```rust
  fn bar(_: i8) { }

  fn main() {
      bar(42);
  }
  ```

For method calls, the receiver (`self` parameter) can only take advantage of unsized coercions.

- Instantiations of struct or variant fields

  For example, `42` is coerced to have type `i8` in the following:

  ```rust
  struct Foo { x: i8 }

  fn main() {
      Foo { x: 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, `42` is coerced to have type `i8` in the following:

  ```rust
  fn foo() -> i8 {
      42
  }
  ```

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:

```rust
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.
- 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 `U`, when `U` is a trait object type and either `T` implements `U` or `T` is a trait object for a subtrait of `U`.
- `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.

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 the second parameter, then `y`, then the first parameter, then
`x`
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 for one of the following:

- The entire function body.
- A statement.
- The body of a `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 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, 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, borrowed, and dereferencing that borrow 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 a 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

Not running destructors in Rust is safe even if it has a type that isn't `'static`. `std::mem::ManuallyDrop` provides a wrapper to prevent a variable or field from being dropped automatically.

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 as 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>`.

```
// 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.

```
// 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:

```
// Resolved as `fn<'a>(&'a str) -> &'a str`.
const RESOLVED_SINGLE: fn(&str) -> &str = |x| x;

// Resolved as `Fn<'a, 'b, 'c>(&'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

- Numeric types
- `char`, `bool`, and `!`
- Tuples of `Copy` types
- Arrays of `Copy` types
- Shared references
- Raw pointers
- Function pointers and function item 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
- Arrays 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.

Auto traits

The `Send`, `Sync`, `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 are `Sized` by default. `Sized` is always implemented automatically by the compiler, not by implementation items.

Memory model

Rust does not yet have a defined memory model. Various academics and industry 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.

Memory ownership

When a stack frame is exited, its local allocations are all released, and its references to boxes are dropped.

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 distributable binary.

- `--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 osx, 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. The static library is actually a `*.a` archive on linux and osx and a `*.lib` file on windows. 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 at once 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:

```rust
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:

```sh
RUSTFLAGS='-C target-feature=+crt-static' cargo build --target x86_64-
pc-windows-msvc
```

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.

Unsafe functions

Unsafe functions are functions that are not safe in all contexts and/or for all possible inputs. Such a function must be prefixed with the keyword `unsafe` and can only be called from an `unsafe` block or another `unsafe` function.

Unsafe blocks

A block of code can be prefixed with the `unsafe` keyword, to permit calling `unsafe` functions or dereferencing raw pointers within a safe function.

When a programmer has sufficient conviction that a sequence of potentially unsafe operations is actually safe, they can encapsulate that sequence (taken as a whole) within an `unsafe` block. The compiler will consider uses of such code safe, in the surrounding context.

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 is in the standard library.

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 with only boxes.

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. 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. The following list is just what we know for sure is undefined behavior. Please read the Rustonomicon before writing unsafe code.

- Data races.
- Dereferencing (using the `*` operator on) a dangling or unaligned raw pointer.
- Breaking the pointer aliasing rules. `&mut T` and `&T` follow LLVM's scoped noalias model, except if the `&T` contains an `UnsafeCell<U>`.
- Mutating immutable data. All data inside a `const` item is immutable. Moreover, all data reached through a shared reference or data owned by an immutable binding is immutable, unless that data is contained within an `UnsafeCell<U>`.
- Invoking undefined behavior via compiler intrinsics.
- Executing code compiled with platform features that the current platform does not support (see `target_feature`).
- 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, unaligned, 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.

A reference/pointer is "dangling" if it is null or not all of the bytes it points to are part of the same allocation (so in particular they all have to be part of *some* allocation). The span of bytes it points to is determined by the pointer value and the size of the pointee type (using `size_of_val`). As a consequence, if the span is empty, "dangling" is the same as "non-null". Note that slices and strings point to their entire range, so it is important that the length metadata is never too large. In particular, allocations and therefore slices and strings cannot be bigger than `isize::MAX` bytes.

> **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.

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.

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.
- Paths to functions and constants. Recursively defining constants is not allowed.
- Tuple expressions.
- Array expressions.
- Struct expressions.
- Enum variant expressions.
- Block expressions, including `unsafe` blocks.
  - let statements and thus irrefutable patterns, with the caveat that until `if` and `match` are implemented, one cannot use both short circuiting operators (`&&` and `||`) and let statements within the same constant.
  - 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.
- Grouped expressions.
- Cast expressions, except pointer to address and function pointer to address casts.
- Calls of const functions and const methods.

Const context

A *const context* is one of the following:

- Array type length expressions
- Repeat expression length expressions
- The initializer of
  - constants
  - statics
  - enum discriminants

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.

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.

```
#![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.

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.

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 such a memory 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.

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.

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.

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.

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.