Rust Style Guide

Motivation - why use a formatting tool?

Formatting code is a mostly mechanical task which takes both time and mental effort. By using an automatic formatting tool, a programmer is relieved of this task and can concentrate on more important things.

Furthermore, by sticking to an established style guide (such as this one), programmers don't need to formulate ad hoc style rules, nor do they need to debate with other programmers what style rules should be used, saving time, communication overhead, and mental energy.

Humans comprehend information through pattern matching. By ensuring that all Rust code has similar formatting, less mental effort is required to comprehend a new project, lowering the barrier to entry for new developers.

Thus, there are productivity benefits to using a formatting tool (such as `rustfmt`), and even larger benefits by using a community-consistent formatting, typically by using a formatting tool's default settings.

The default Rust style

The Rust Style Guide defines the default Rust style, and *recommends* that developers and tools follow the default Rust style. Tools such as `rustfmt` use the style guide as a reference for the default style. Everything in this style guide, whether or not it uses language such as "must" or the imperative mood such as "insert a space ..." or "break the line after ...", refers to the default style.

This should not be interpreted as forbidding developers from following a non-default style, or forbidding tools from adding any particular configuration options.

Bugs

If the style guide differs from rustfmt, that may represent a bug in rustfmt, or a bug in the style guide; either way, please report it to the style team or the rustfmt team or both, for investigation and fix.

If implementing a new formatting tool based on the style guide and default Rust style, please test it on the corpus of existing Rust code, and avoid causing widespread breakage. The implementation and testing of such a tool may surface bugs in either the style guide or rustfmt, as well as bugs in the tool itself.

We typically resolve bugs in a fashion that avoids widespread breakage.

Formatting conventions

Indentation and line width

- Use spaces, not tabs.
- Each level of indentation must be 4 spaces (that is, all indentation outside of string literals and comments must be a multiple of 4).
- The maximum width for a line is 100 characters.

Block indent

Prefer block indent over visual indent:

```
// Block indent
a_function_call(
    foo,
    bar,
);

// Visual indent
a_function_call(foo,
                bar);
```

This makes for smaller diffs (e.g., if `a_function_call` is renamed in the above example) and less rightward drift.

Trailing commas

In comma-separated lists of any kind, use a trailing comma when followed by a newline:

```
function_call(
    argument,
    another_argument,
);

let array = [
    element,
    another_element,
    yet_another_element,
];
```

This makes moving code (e.g., by copy and paste) easier, and makes diffs smaller, as appending or removing items does not require modifying another line to add or remove a comma.

Blank lines

Separate items and statements by either zero or one blank lines (i.e., one or two newlines). E.g,

```
fn foo() {
    let x = ...;

    let y = ...;
    let z = ...;
}

fn bar() {}
fn baz() {}
```

Trailing whitespace

Do not include trailing whitespace on the end of any line. This includes blank lines, comment lines, code lines, and string literals.

Note that avoiding trailing whitespace in string literals requires care to preserve the value of the literal.

Sorting

In various cases, the default Rust style specifies to sort things. If not otherwise specified, such sorting should be "version sorting", which ensures that (for instance) `x8` comes before `x16` even though the character `1` comes before the character `8`.

For the purposes of the Rust style, to compare two strings for version-sorting:

- Process both strings from beginning to end as two sequences of maximal-length chunks, where each chunk consists either of a sequence of characters other than ASCII digits, or a sequence of ASCII digits (a numeric chunk), and compare corresponding chunks from the strings.
- To compare two numeric chunks, compare them by numeric value, ignoring leading zeroes. If the two chunks have equal numeric value, but different numbers of leading digits, and this is the first time this has happened for these strings, treat the chunks as equal (moving on to the next chunk) but remember which string had more leading zeroes.
- To compare two chunks if both are not numeric, compare them by Unicode character lexicographically, with two exceptions:
  - `_` (underscore) sorts immediately after ` ` (space) but before any other character. (This treats underscore as a word separator, as commonly used in identifiers.)
  - Unless otherwise specified, version-sorting should sort non-lowercase characters (characters that can start an `UpperCamelCase` identifier) before lowercase characters.
- If the comparison reaches the end of the string and considers each pair of chunks equal:
  - If one of the numeric comparisons noted the earliest point at which one string had more leading zeroes than the other, sort the string with more leading zeroes first.
  - Otherwise, the strings are equal.

Note that there exist various algorithms called "version sorting", which generally try to solve the same problem, but which differ in various ways (such as in their handling of numbers with leading zeroes). This algorithm does not purport to precisely match the behavior of any particular other algorithm, only to produce a simple and satisfying result for Rust formatting. In particular, this algorithm aims to produce a satisfying result for a set of symbols that have the same number of leading zeroes, and an acceptable and easily understandable result for a set of symbols that has varying numbers of leading zeroes.

As an example, version-sorting will sort the following strings in the order given:

- `_ZYXW`
- `_abcd`
- `A2`
- `ABCD`
- `Z_YXW`
- `ZY_XW`
- `ZY_XW`
- `ZYXW`
- `ZYXW_`
- `a1`
- `abcd`
- `u_zzz`
- `u8`
- `u16`
- `u32`
- `u64`
- `u128`
- `u256`
- `ua`
- `usize`
- `uz`
- `v000`
- `v00`
- `v0`
- `v0s`
- `v00t`
- `v0u`
- `v001`
- `v01`
- `v1`
- `v009`
- `v09`
- `v9`
- `v010`
- `v10`
- `w005s09t`
- `w5s009t`
- `x64`
- `x86`
- `x86_32`
- `x86_64`
- `x86_128`
- `x87`
- `zyxw`

Module-level items

Statements

Expressions

Types

Comments

The following guidelines for comments are recommendations only, a mechanical formatter might skip formatting of comments.

Prefer line comments (`//`) to block comments (`/* ... */`).

When using line comments, put a single space after the opening sigil.

When using single-line block comments, put a single space after the opening sigil and before the closing sigil. For multi-line block comments, put a newline after the opening sigil, and a newline before the closing sigil.

Prefer to put a comment on its own line. Where a comment follows code, put a single space before it. Where a block comment appears inline, use surrounding whitespace as if it were an identifier or keyword.

Examples:

```
// A comment on an item.
struct Foo { ... }

fn foo() {} // A comment after an item.

pub fn foo(/* a comment before an argument */ x: T) {...}
```

Comments should usually be complete sentences. Start with a capital letter, end with a period (`.`). An inline block comment may be treated as a note without punctuation.

Source lines which are entirely a comment should be limited to 80 characters in length (including comment sigils, but excluding indentation) or the maximum width of the line (including comment sigils and indentation), whichever is smaller:

```
// This comment goes up to the ............................... 80 char margin.

{
    // This comment is ........................................... 80 chars wide.
}

{
    {
        {
            {
                {
                    {
                        // This comment is limited by the ........................
100 char margin.
                    }
                }
            }
        }
    }
}
```

Doc comments

Prefer line comments (`///`) to block comments (`/** ... */`).

Prefer outer doc comments (`///` or `/** ... */`), only use inner doc comments (`//!` and `/*! ... */`) to write module-level or crate-level documentation.

Put doc comments before attributes.

Attributes

Put each attribute on its own line, indented to the level of the item. In the case of inner attributes (`#!`), indent it to the level of the inside of the item. Prefer outer attributes, where possible.

For attributes with argument lists, format like functions.

```
#[repr(C)]
#[foo(foo, bar)]
#[long_multi_line_attribute(
    split,
    across,
    lines,
)]
struct CRepr {
    #![repr(C)]
    x: f32,
    y: f32,
}
```

For attributes with an equal sign, put a single space before and after the `=` , e.g., `#[foo = 42]` .

There must only be a single `derive` attribute. Note for tool authors: if combining multiple `derive` attributes into a single attribute, the ordering of the derived names must generally be preserved for correctness: `#[derive(Foo)] #[derive(Bar)] struct Baz;` must be formatted to `#[derive(Foo, Bar)] struct Baz;` .

small items

In many places in this guide we specify formatting that depends on a code construct being *small*. For example, single-line vs multi-line struct literals:

```
// Normal formatting
Foo {
    f1: an_expression,
    f2: another_expression(),
}

// "small" formatting
Foo { f1, f2 }
```

We leave it to individual tools to decide on exactly what *small* means. In particular, tools are free to use different definitions in different circumstances.

Some suitable heuristics are the size of the item (in characters) or the complexity of an item (for example, that all components must be simple names, not more complex sub-expressions). For more discussion on suitable heuristics, see this issue.

Non-formatting conventions

Cargo.toml conventions

Principles used for deciding these guidelines

Items

Items consist of the set of things permitted at the top level of a module. However, Rust also allows some items to appear within some other types of items, such as within a function. The same formatting conventions apply whether an item appears at module level or within another item.

`extern crate` statements must be first in a file. They must be ordered alphabetically.

`use` statements, and module *declarations* (`mod foo;`, not `mod { ... }`) must come before other items. Put imports before module declarations. Version-sort each, except that `self` and `super` must come before any other names.

Don't automatically move module declarations annotated with `#[macro_use]`, since that might change semantics.

Function definitions

In Rust, people often find functions by searching for `fn [function-name]`, so the formatting of function definitions must enable this.

The proper ordering and spacing is:

```
[pub] [unsafe] [extern ["ABI"]] fn foo(arg1: i32, arg2: i32) -> i32 {
    ...
}
```

Avoid comments within the signature itself.

If the function signature does not fit on one line, then break after the opening parenthesis and before the closing parenthesis and put each argument on its own block-indented line. For example,

```
fn foo(
    arg1: i32,
    arg2: i32,
) -> i32 {
    ...
}
```

Note the trailing comma on the last argument.

Tuples and tuple structs

Write the type list as you would a parameter list to a function.

Build a tuple or tuple struct as you would call a function.

Single-line

```
struct Bar(Type1, Type2);

let x = Bar(11, 22);
let y = (11, 22, 33);
```

Enums

In the declaration, put each variant on its own line, block indented.

Format each variant accordingly as either a struct (but without the `struct` keyword), a tuple struct, or an identifier (which doesn't require special formatting):

```
enum FooBar {
    First(u32),
    Second,
    Error {
        err: Box<Error>,
        line: u32,
    },
}
```

If a struct variant is *small*, format it on one line. In this case, do not use a trailing comma for the field list, but do put spaces around each brace:

```
enum FooBar {
    Error { err: Box<Error>, line: u32 },
}
```

In an enum with multiple struct variants, if any struct variant is written on multiple lines, use the multi-line formatting for all struct variants. However, such a situation might be an indication that you should factor out the fields of the variant into their own struct.

Structs and Unions

Struct names follow on the same line as the `struct` keyword, with the opening brace on the same line when it fits within the right margin. All struct fields are indented once and end with a trailing comma. The closing brace is not indented and appears on its own line.

```
struct Foo {
    a: A,
    b: B,
}
```

If and only if the type of a field does not fit within the right margin, it is pulled down to its own line and indented again.

```
struct Foo {
    a: A,
    long_name:
        LongType,
}
```

Prefer using a unit struct (e.g., `struct Foo;`) to an empty struct (e.g., `struct Foo();` or `struct Foo {}`, these only exist to simplify code generation), but if you must use an empty struct, keep it on one line with no space between the braces: `struct Foo();` or `struct Foo {}`.

The same guidelines are used for untagged union declarations.

```
union Foo {
    a: A,
    b: B,
    long_name:
        LongType,
}
```

Tuple structs

Put the whole struct on one line if possible. Separate types within the parentheses using a comma and space. Don't use a trailing comma for a single-line tuple struct. Don't put spaces around the parentheses or semicolon:

```
pub struct Foo(String, u8);
```

Prefer unit structs to empty tuple structs (these only exist to simplify code generation), e.g., `struct Foo;` rather than `struct Foo();`.

For more than a few fields (in particular if the tuple struct does not fit on one line), prefer a proper struct with named fields.

For a multi-line tuple struct, block-format the fields with a field on each line and a trailing comma:

```
pub struct Foo(
    String,
    u8,
);
```

Traits

Use block-indent for trait items. If there are no items, format the trait (including its `{}`) on a single line. Otherwise, break after the opening brace and before the closing brace:

```
trait Foo {}

pub trait Bar {
    ...
}
```

If the trait has bounds, put a space after the colon but not before, and put spaces around each `+`, e.g.,

```
trait Foo: Debug + Bar {}
```

Prefer not to line-break in the bounds if possible (consider using a `where` clause). Prefer to break between bounds than to break any individual bound. If you must break the bounds, put each bound (including the first) on its own block-indented line, break before the `+` and put the opening brace on its own line:

```
pub trait IndexRanges:
    Index<Range<usize>, Output=Self>
    + Index<RangeTo<usize>, Output=Self>
    + Index<RangeFrom<usize>, Output=Self>
    + Index<RangeFull, Output=Self>
{
    ...
}
```

Impls

Use block-indent for impl items. If there are no items, format the impl (including its `{}`) on a single line. Otherwise, break after the opening brace and before the closing brace:

```
impl Foo {}

impl Bar for Foo {
    ...
}
```

Avoid line-breaking in the signature if possible. If a line break is required in a non-inherent impl, break immediately before `for`, block indent the concrete type and put the opening brace on its own line:

```
impl Bar
    for Foo
{
    ...
}
```

Extern crate

```
extern crate foo;
```

Use spaces around keywords, no spaces around the semicolon.

Modules

```
mod foo {
}

mod foo;
```

Use spaces around keywords and before the opening brace, no spaces around the semicolon.

`macro_rules!`

Use `{}` for the full definition of the macro.

```
macro_rules! foo {
}
```

Generics

Prefer to put a generics clause on one line. Break other parts of an item declaration rather than line-breaking a generics clause. If a generics clause is large enough to require line-breaking, prefer a `where` clause instead.

Do not put spaces before or after `<` nor before `>`. Only put a space after `>` if it is followed by a word or opening brace, not an opening parenthesis. Put a space after each comma. Do not use a trailing comma for a single-line generics clause.

```
fn foo<T: Display, U: Debug>(x: Vec<T>, y: Vec<U>) ...

impl<T: Display, U: Debug> SomeType<T, U> { ...
```

If the generics clause must be formatted across multiple lines, put each parameter on its own block-indented line, break after the opening `<` and before the closing `>`, and use a trailing comma.

```
fn foo<
    T: Display,
    U: Debug,
>(x: Vec<T>, y: Vec<U>) ...
```

If an associated type is bound in a generic type, put spaces around the `=`:

```
<T: Example<Item = u32>>
```

Prefer to use single-letter names for generic parameters.

`where` clauses

These rules apply for `where` clauses on any item.

If immediately following a closing bracket of any kind, write the keyword `where` on the same line, with a space before it.

Otherwise, put `where` on a new line at the same indentation level. Put each component of a `where` clause on its own line, block-indented. Use a trailing comma, unless the clause is terminated with a semicolon. If the `where` clause is followed by a block (or assignment), start that block on a new line.

Examples:

```
fn function<T, U>(args)
where
    T: Bound,
    U: AnotherBound,
{
    body
}

fn foo<T>(
    args
) -> ReturnType
where
    T: Bound,
{
    body
}

fn foo<T, U>(
    args,
) where
    T: Bound,
    U: AnotherBound,
{
    body
}

fn foo<T, U>(
    args
) -> ReturnType
where
    T: Bound,
    U: AnotherBound;  // Note, no trailing comma.

// Note that where clauses on `type` aliases are not enforced and should not
// be used.
type Foo<T>
where
    T: Bound
= Bar<T>;
```

If a `where` clause is very short, prefer using an inline bound on the type parameter.

If a component of a `where` clause does not fit and contains `+`, break it before each `+` and block-indent the continuation lines. Put each bound on its own line. E.g.,

```
impl<T: ?Sized, Idx> IndexRanges<Idx> for T
where
    T: Index<Range<Idx>, Output = Self::Output>
        + Index<RangeTo<Idx>, Output = Self::Output>
        + Index<RangeFrom<Idx>, Output = Self::Output>
        + Index<RangeInclusive<Idx>, Output = Self::Output>
        + Index<RangeToInclusive<Idx>, Output = Self::Output>
        + Index<RangeFull>,
```

Type aliases

Keep type aliases on one line when they fit. If necessary to break the line, do so before the `=`, and

block-indent the right-hand side:

```
pub type Foo = Bar<T>;

// If multi-line is required
type VeryLongType<T, U: SomeBound>
    = AnEvenLongerType<T, U, Foo<T>>;
```

When there is a trailing `where` clause after the type, and no `where` clause present before the type, break before the `=` and indent. Then break before the `where` keyword and format the clauses normally, e.g.,

```
// With only a trailing where clause
type VeryLongType<T, U>
    = AnEvenLongerType<T, U, Foo<T>>
where
    T: U::AnAssociatedType,
    U: SomeBound;
```

When there is a `where` clause before the type, format it normally, and break after the last clause. Do not indent before the `=` to leave it visually distinct from the indented clauses that precede it. If there is additionally a `where` clause after the type, break before the `where` keyword and format the clauses normally.

```
// With only a preceding where clause.
type WithPrecedingWC<T, U>
where
    T: U::AnAssociatedType,
    U: SomeBound,
= AnEvenLongerType<T, U, Foo<T>>;

// Or with both a preceding and trailing where clause.
type WithPrecedingWC<T, U>
where
    T: U::AnAssociatedType,
    U: SomeBound,
= AnEvenLongerType<T, U, Foo<T>>
where
    T: U::AnAssociatedType2,
    U: SomeBound2;
```

Associated types

Format associated types like type aliases. Where an associated type has a bound, put a space after the colon but not before:

```
pub type Foo: Bar;
```

extern items

When writing extern items (such as `extern "C" fn`), always specify the ABI. For example, write `extern "C" fn foo ...` or `unsafe extern "C" { ...}` and avoid `extern fn foo ...` and `unsafe`

```
extern { ... }.
```

Imports (`use` statements)

Format imports on one line where possible. Don't put spaces around braces.

```
use a::b::c;
use a::b::d::*;
use a::b::{foo, bar, baz};
```

Large list imports

Prefer to use multiple imports rather than a multi-line import. However, tools should not split imports by default.

If an import does require multiple lines (either because a list of single names does not fit within the max width, or because of the rules for nested imports below), then break after the opening brace and before the closing brace, use a trailing comma, and block indent the names.

```
// Prefer
foo::{long, list, of, imports};
foo::{more, imports};

// If necessary
foo::{
    long, list, of, imports, more,
    imports,  // Note trailing comma
};
```

Ordering of imports

A *group* of imports is a set of imports on the same or sequential lines. One or more blank lines or other items (e.g., a function) separate groups of imports.

Within a group of imports, imports must be version-sorted. Groups of imports must not be merged or re-ordered.

E.g., input:

```
use d;
use c;

use b;
use a;
```

output:

```
use c;
use d;

use a;
use b;
```

Because of `macro_use`, attributes must also start a new group and prevent re-ordering.

Ordering list import

Names in a list import must be version-sorted, except that:

- `self` and `super` always come first if present, and
- groups and glob imports always come last if present.

This applies recursively. For example, `a::*` comes before `b::a` but `a::b` comes before `a::*`. E.g., `use foo::bar::{a, b::c, b::d, b::d::{x, y, z}, b::{self, r, s}};`.

Normalisation

Tools must make the following normalisations, recursively:

- `use a::self;` -> `use a;`
- `use a::{};` -> (nothing)
- `use a::{b};` -> `use a::b;`

Tools must not otherwise merge or un-merge import lists or adjust glob imports (without an explicit option).

Nested imports

If there are any nested imports in a list import, then use the multi-line form, even if the import fits on one line. Each nested import must be on its own line, but non-nested imports must be grouped on as few lines as possible.

For example,

```
use a::b::{
    x, y, z,
    u::{...},
    w::{...},
};
```

Merging/un-merging imports

An example:

```
// Un-merged
use a::b;
use a::c::d;

// Merged
use a::{b, c::d};
```

Tools must not merge or un-merge imports by default. They may offer merging or un-merging as an option.

Statements

Let statements

Put a space after the `:` and on both sides of the `=` (if they are present). Don't put a space before the semicolon.

```
// A comment.
let pattern: Type = expr;

let pattern;
let pattern: Type;
let pattern = expr;
```

If possible, format the declaration on a single line. If not possible, then try splitting after the `=`, if the declaration fits on two lines. Block-indent the expression.

```
let pattern: Type =
    expr;
```

If the first line still does not fit on a single line, split after the `:`, and use block indentation. If the type requires multiple lines, even after line-breaking after the `:`, then place the first line on the same line as the `:`, subject to the combining rules.

```
let pattern:
    Type =
    expr;
```

e.g,

```
let Foo {
    f: abcd,
    g: qwer,
}: Foo<Bar> =
    Foo { f, g };

let (abcd,
    defg):
    Baz =
{ ... }
```

If the expression covers multiple lines, if the first line of the expression fits in the remaining space, it stays on the same line as the `=`, and the rest of the expression is not further indented. If the first line does not fit, then put the expression on subsequent lines, block indented. If the expression is a block and the type or pattern cover multiple lines, put the opening brace on a new line and not indented (this provides separation for the interior of the block from the type); otherwise, the opening brace follows the `=`.

Examples:

```
let foo = Foo {
    f: abcd,
    g: qwer,
};

let foo =
    ALongName {
        f: abcd,
        g: qwer,
    };

let foo: Type = {
    an_expression();
    ...
};

let foo:
    ALongType =
{
    an_expression();
    ...
};

let Foo {
    f: abcd,
    g: qwer,
}: Foo<Bar> = Foo {
    f: blimblimblim,
    g: blamblamblam,
};

let Foo {
    f: abcd,
    g: qwer,
}: Foo<Bar> = foo(
    blimblimblim,
    blamblamblam,
);
```

else blocks (let-else statements)

A let statement can contain an `else` component, making it a let-else statement. In this case, always apply the same formatting rules to the components preceding the `else` block (i.e. the `let pattern: Type = initializer_expr` portion) as described for other let statements.

Format the entire let-else statement on a single line if all the following are true:

- the entire statement is *short*
- the `else` block contains only a single-line expression and no statements
- the `else` block contains no comments
- the let statement components preceding the `else` block can be formatted on a single line

```
let Some(1) = opt else { return };
```

Otherwise, the let-else statement requires some line breaks.

If breaking a let-else statement across multiple lines, never break between the `else` and the `{`, and always break before the `}`.

If the let statement components preceding the `else` can be formatted on a single line, but the let-else does not qualify to be placed entirely on a single line, put the `else {` on the same line as the initializer expression, with a space between them, then break the line after the `{`. Indent the closing `}` to match the `let`, and indent the contained block one step further.

```
let Some(1) = opt else {
    return;
};

let Some(1) = opt else {
    // nope
    return
};
```

If the let statement components preceding the `else` can be formatted on a single line, but the `else {` does not fit on the same line, break the line before the `else`.

```
    let Some(x) =
some_really_really_really_really_really_really_really_really_long_name
    else {
        return;
    };
```

If the initializer expression is multi-line, put the `else` keyword and opening brace of the block (i.e. `else {`) on the same line as the end of the initializer expression, with a space between them, if and only if all the following are true:

- The initializer expression ends with one or more closing parentheses, square brackets, and/or braces
- There is nothing else on that line
- That line has the same indentation level as the initial `let` keyword.

For example:

```
let Some(x) = y.foo(
    "abc",
    fairly_long_identifier,
    "def",
    "123456",
    "string",
    "cheese",
) else {
    bar()
}
```

Otherwise, put the `else` keyword and opening brace on the next line after the end of the initializer expression, with the `else` keyword at the same indentation level as the `let` keyword.

For example:

```
fn main() {
    let Some(x) = abcdef()
        .foo(
            "abc",
            some_really_really_really_long_ident,
            "ident",
            "123456",
        )
        .bar()
        .baz()
        .qux("ffffffffffffffff")
    else {
        return
    };

    let Some(aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa) =
        bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
    else {
        return;
    };

    let LongStructName(AnotherStruct {
        multi,
        line,
        pattern,
    }) = slice.as_ref()
    else {
        return;
    };

    let LongStructName(AnotherStruct {
        multi,
        line,
        pattern,
    }) = multi_line_function_call(
        arg1,
        arg2,
        arg3,
        arg4,
    ) else {
        return;
    };
}
```

Macros in statement position

For a macro use in statement position, use parentheses or square brackets as delimiters, and terminate it with a semicolon. Do not put spaces around the name, `!`, the delimiters, or the `;`.

```
// A comment.
a_macro!(...);
```

Expressions in statement position

Do not put space between the expression and the semicolon.

```
<expr>;
```

Terminate all expressions in statement position with a semicolon, unless they end with a block or are used as the value for a block.

E.g.,

```
{
    an_expression();
    expr_as_value()
}

return foo();

loop {
    break;
}
```

Use a semicolon where an expression has void type, even if it could be propagated. E.g.,

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

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

Expressions

Blocks

A block expression must have a newline after the initial `{` and before the terminal `}`, unless it qualifies to be written as a single line based on another style rule.

A keyword before the block (such as `unsafe` or `async`) must be on the same line as the opening brace, with a single space between the keyword and the opening brace. Indent the contents of the block.

```
fn block_as_stmt() {
    a_call();

    {
        a_call_inside_a_block();

        // a comment in a block
        the_value
    }
}

fn block_as_expr() {
    let foo = {
        a_call_inside_a_block();

        // a comment in a block
        the_value
    };
}

fn unsafe_block_as_stmt() {
    a_call();

    unsafe {
        a_call_inside_a_block();

        // a comment in a block
        the_value
    }
}
```

If a block has an attribute, put it on its own line before the block:

```
fn block_as_stmt() {
    #[an_attribute]
    {
        #![an_inner_attribute]

        // a comment in a block
        the_value
    }
}
```

Avoid writing comments on the same lines as either of the braces.

Write an empty block as `{}`.

Write a block on a single line if:

- it is either used in expression position (not statement position) or is an unsafe block in statement position,
- it contains a single-line expression and no statements, and
- it contains no comments

For a single-line block, put spaces after the opening brace and before the closing brace.

Examples:

```
fn main() {
    // Single line
    let _ = { a_call() };
    let _ = unsafe { a_call() };

    // Not allowed on one line
    // Statement position.
    {
        a_call()
    }

    // Contains a statement
    let _ = {
        a_call();
    };
    unsafe {
        a_call();
    }

    // Contains a comment
    let _ = {
        // A comment
    };
    let _ = {
        // A comment
        a_call()
    };

    // Multiple lines
    let _ = {
        a_call();
        another_call()
    };
    let _ = {
        a_call(
            an_argument,
            another_arg,
        )
    };
}
```

Closures

Don't put any extra spaces before the first `|` (unless the closure is prefixed by a keyword such as `move`); put a space between the second `|` and the expression of the closure. Between the `|`s, use function definition syntax, but elide types where possible.

Use closures without the enclosing `{}`, if possible. Add the `{}` when you have a return type, when

there are statements, when there are comments inside the closure, or when the body expression is a control-flow expression that spans multiple lines. If using braces, follow the rules above for blocks. Examples:

```
|arg1, arg2| expr

move |arg1: i32, arg2: i32| -> i32 {
    expr1;
    expr2
}

|| Foo {
    field1,
    field2: 0,
}

|| {
    if true {
        blah
    } else {
        boo
    }
}

|x| unsafe {
    expr
}
```

Struct literals

If a struct literal is *small*, format it on a single line, and do not use a trailing comma. If not, split it across multiple lines, with each field on its own block-indented line, and use a trailing comma.

For each `field: value` entry, put a space after the colon only.

Put a space before the opening brace. In the single-line form, put spaces after the opening brace and before the closing brace.

```
Foo { field1, field2: 0 }
let f = Foo {
    field1,
    field2: an_expr,
};
```

Functional record update syntax is treated like a field, but it must never have a trailing comma. Do not put a space after `...`.

```
let f = Foo {
    field1,
    ..an_expr
};
```

Unit literals

Never break between the opening and closing parentheses of the `()` unit literal.

Tuple literals

Use a single-line form where possible. Do not put spaces between the opening parenthesis and the first element, or between the last element and the closing parenthesis. Separate elements with a comma followed by a space.

Where a single-line form is not possible, write the tuple across multiple lines, with each element of the tuple on its own block-indented line, and use a trailing comma.

```
(a, b, c)

let x = (
    a_long_expr,
    another_very_long_expr,
);
```

Tuple struct literals

Do not put space between the identifier and the opening parenthesis. Otherwise, follow the rules for tuple literals:

```
Foo(a, b, c)

let x = Foo(
    a_long_expr,
    another_very_long_expr,
);
```

Enum literals

Follow the formatting rules for the various struct literals. Prefer using the name of the enum as a qualifying name, unless the enum is in the prelude:

```
Foo::Bar(a, b)
Foo::Baz {
    field1,
    field2: 1001,
}
Ok(an_expr)
```

Array literals

Write small array literals on a single line. Do not put spaces between the opening square bracket and the first element, or between the last element and the closing square bracket. Separate

elements with a comma followed by a space.

If using the repeating initializer, put a space after the semicolon only.

Apply the same rules if using `vec!` or similar array-like macros; always use square brackets with such macros. Examples:

```
fn main() {
    let x = [1, 2, 3];
    let y = vec![a, b, c, d];
    let a = [42; 10];
}
```

For arrays that have to be broken across lines, if using the repeating initializer, break after the `;`, not before. Otherwise, follow the rules below for function calls. In any case, block-indent the contents of the initializer, and put line breaks after the opening square bracket and before the closing square bracket:

```
fn main() {
    [
        a_long_expression();
        1234567890
    ]
    let x = [
        an_expression,
        another_expression,
        a_third_expression,
    ];
}
```

Array accesses, indexing, and slicing

Don't put spaces around the square brackets. Avoid breaking lines if possible. Never break a line between the target expression and the opening square bracket. If the indexing expression must be broken onto a subsequent line, or spans multiple lines itself, then block-indent the indexing expression, and put newlines after the opening square bracket and before the closing square bracket:

Examples:

```
fn main() {
    foo[42];
    &foo[..10];
    bar[0..100];
    foo[4 + 5 / bar];
    a_long_target[
        a_long_indexing_expression
    ];
}
```

Unary operations

Do not include a space between a unary op and its operand (i.e., `!x`, not `! x`). However, there must

be a space after `&mut`. Avoid line-breaking between a unary operator and its operand.

Binary operations

Do include spaces around binary ops (i.e., `x + 1`, not `x+1`) (including `=` and other assignment operators such as `+=` or `*=`).

For comparison operators, because for `T op U`, `&T op &U` is also implemented: if you have `t: &T`, and `u: U`, prefer `*t op u` to `t op &u`. In general, within expressions, prefer dereferencing to taking references, unless necessary (e.g. to avoid an unnecessarily expensive operation).

Use parentheses liberally; do not necessarily elide them due to precedence. Tools should not automatically insert or remove parentheses. Do not use spaces to indicate precedence.

If line-breaking, block-indent each subsequent line. For assignment operators, break after the operator; for all other operators, put the operator on the subsequent line. Put each sub-expression on its own line:

```
foo_bar
    + bar
    + baz
    + qux
    + whatever
```

Prefer line-breaking at an assignment operator (either `=` or `+=`, etc.) rather than at other binary operators.

Casts (`as`)

Format `as` casts like a binary operator. In particular, always include spaces around `as`, and if line-breaking, break before the `as` (never after) and block-indent the subsequent line. Format the type on the right-hand side using the rules for types.

However, unlike with other binary operators, if chaining a series of `as` casts that require line-breaking, and line-breaking before the first `as` suffices to make the remainder fit on the next line, don't break before any subsequent `as`; instead, leave the series of types all on the same line:

```
let cstr = very_long_expression()
    as *const str as *const [u8] as *const std::os::raw::c_char;
```

If the subsequent line still requires line-breaking, break and block-indent before each `as` as with other binary operators.

Control flow

Do not include extraneous parentheses for `if` and `while` expressions.

```
if true {
}
```

is better than

```
if (true) {
}
```

Do include extraneous parentheses if it makes an arithmetic or logic expression easier to understand (`(x * 15) + (y * 20)` is fine)

Function calls

Do not put a space between the function name, and the opening parenthesis.

Do not put a space between an argument, and the comma which follows.

Do put a space between an argument, and the comma which precedes it.

Prefer not to break a line in the callee expression.

For a function call with no arguments (a nullary function call like `func()`), never break within the parentheses, and never put a space between the parentheses. Always write a nullary function call as a single-line call, never a multi-line call.

Single-line calls

Do not put a space between the function name and open paren, between the open paren and the first argument, or between the last argument and the close paren.

Do not put a comma after the last argument.

```
foo(x, y, z)
```

Multi-line calls

If the function call is not *small*, it would otherwise over-run the max width, or any argument or the callee is multi-line, then format the call across multiple lines. In this case, put each argument on its own block-indented line, break after the opening parenthesis and before the closing parenthesis, and use a trailing comma:

```
a_function_call(
    arg1,
    a_nested_call(a, b),
)
```

Method calls

Follow the function rules for calling.

Do not put any spaces around the `.`.

```
x.foo().bar().baz(x, y, z);
```

Macro uses

If a macro can be parsed like other constructs, format it like those constructs. For example, a macro use `foo!(a, b, c)` can be parsed like a function call (ignoring the `!`), so format it using the rules for function calls.

The style guide defines specific formatting for particular macros in the language or standard library. The style guide does not define formatting for any third-party macros, even if similar to those in the language or standard library.

Format string macros

For macros which take a format string, if all other arguments are *small*, format the arguments before the format string on a single line if they fit, and format the arguments after the format string on a single line if they fit, with the format string on its own line. If the arguments are not small or do not fit, put each on its own line as with a function. For example:

```
println!(
    "Hello {} and {}",
    name1, name2,
);

assert_eq!(
    x, y,
    "x and y were not equal, see {}",
    reason,
);
```

Chains of fields and method calls

A chain is a sequence of field accesses, method calls, and/or uses of the try operator `?`. E.g., `a.b.c().d` or `foo?.bar().baz?`.

Format the chain on one line if it is "small" and otherwise possible to do so. If formatting on multiple lines, put each field access or method call in the chain on its own line, with the line-break before the `.` and after any `?`. Block-indent each subsequent line:

```
let foo = bar
    .baz?
    .qux();
```

If the length of the last line of the first element plus its indentation is less than or equal to the indentation of the second line, then combine the first and second lines if they fit. Apply this rule

recursively.

```
x.baz?
    .qux()

x.y.z
    .qux()

let foo = x
    .baz?
    .qux();

foo(
    expr1,
    expr2,
).baz?
    .qux();
```

Multi-line elements

If any element in a chain is formatted across multiple lines, put that element and any later elements on their own lines.

```
a.b.c()?
    .foo(
        an_expr,
        another_expr,
    )
    .bar
    .baz
```

Note there is block indent due to the chain and the function call in the above example.

Prefer formatting the whole chain in multi-line style and each element on one line, rather than putting some elements on multiple lines and some on a single line, e.g.,

```
// Better
self.pre_comment
    .as_ref()
    .map_or(false, |comment| comment.starts_with("//"))

// Worse
self.pre_comment.as_ref().map_or(
    false,
    |comment| comment.starts_with("//"),
)
```

Control flow expressions

This section covers `for` and `loop` expressions, as well as `if` and `while` expressions with their sub-expression variants. This includes those with a single `let` sub-expression (i.e. `if let` and `while let`) as well as "let-chains": those with one or more `let` sub-expressions and one or more bool-type conditions (i.e. `if a && let Some(b) = c`).

Put the keyword, any initial clauses, and the opening brace of the block all on a single line, if they fit.

Apply the usual rules for block formatting to the block.

If there is an `else` component, then put the closing brace, `else`, any following clause, and the opening brace all on the same line, with a single space before and after the `else` keyword:

```
if ... {
    ...
} else {
    ...
}

if let ... {
    ...
} else if ... {
    ...
} else {
    ...
}
```

If the control line needs to be broken, then prefer breaking after the `=` for any `let` sub-expression in an `if` or `while` expression that does not fit, and before `in` in a `for` expression; the following line should be block indented. If the control line is broken for any reason, then the opening brace should be on its own line and not indented. Examples:

```
while let Some(foo)
    = a_long_expression
{
    ...
}

for foo
    in a_long_expression
{
    ...
}

if a_long_expression
    && another_long_expression
    || a_third_long_expression
{
    ...
}

if let Some(a) = b
    && another_long_expression
    && a_third_long_expression
{
    // ...
}

if let Some(relatively_long_thing)
    = a_long_expression
    && another_long_expression
    && a_third_long_expression
{
    // ...
}

if some_expr
    && another_long_expression
    && let Some(relatively_long_thing) =
        a_long_long_long_long_long_long_really_reallllllllllyyyyyyy_long_expression
    && a_third_long_expression
{
    // ...
}
```

A let-chain control line is allowed to be formatted on a single line provided it only consists of two clauses, with the first, left-hand side operand being a literal or an `ident` (which can optionally be preceded by any number of unary prefix operators), and the second, right-hand side operand being a single-line `let` clause. Otherwise, the control line must be broken and formatted according to the above rules. For example:

```
if a && let Some(b) = foo() {
    // ...
}

if true && let Some(b) = foo() {
    // ...
}

let operator = if !from_hir_call && let Some(p) = parent {
    // ...
};

if let Some(b) = foo()
    && a
{
    // ..
}

if foo()
    && let Some(b) = bar
{
    // ...
}

if gen_pos != GenericArgPosition::Type
    && let Some(b) = gen_args.bindings.first()
{
    // ..
}
```

Where the initial clause spans multiple lines and ends with one or more closing parentheses, square brackets, or braces, and there is nothing else on that line, and that line is not indented beyond the indent on the first line of the control flow expression, then put the opening brace of the block on the same line with a preceding space. For example:

```
if !self.config.file_lines().intersects(
    &self.codemap.lookup_line_range(
        stmt.span,
    ),
) {  // Opening brace on same line as initial clause.
    ...
}
```

Single line `if else`

Put an `if else` or `if let else` on a single line if it occurs in expression context (i.e., is not a standalone statement), it contains a single `else` clause, and is *small*:

```
let y = if x { 0 } else { 1 };

// Examples that must be multi-line.
let y = if something_very_long {
    not_small
} else {
    also_not_small
};

if x {
    0
} else {
    1
}
```

Match

Prefer not to line-break inside the discriminant expression. Always break after the opening brace and before the closing brace. Block-indent the match arms once:

```
match foo {
    // arms
}

let x = match foo.bar.baz() {
    // arms
};
```

Use a trailing comma for a match arm if and only if not using a block.

Never start a match arm pattern with `|`:

```
match foo {
    // Don't do this.
    | foo => bar,
    // Or this.
    | a_very_long_pattern
    | another_pattern
    | yet_another_pattern
    | a_fourth_pattern => {
        ...
    }
}
```

Prefer:

```
match foo {
    foo => bar,
    a_very_long_pattern
    | another_pattern
    | yet_another_pattern
    | a_fourth_pattern => {
        ...
    }
}
```

Avoid splitting the left-hand side (before the `=>`) of a match arm where possible. If the right-hand

side of the match arm is kept on the same line, never use a block (unless the block is empty).

If the right-hand side consists of multiple statements, or has line comments, or the start of the line does not fit on the same line as the left-hand side, use a block. Do not flatten a right-hand side block containing a single macro call because its expanded form could contain a trailing semicolon.

Block-indent the body of a block arm.

Examples:

```
match foo {
    foo => bar,
    a_very_long_pattern | another_pattern if an_expression() => {
        no_room_for_this_expression()
    }
    foo => {
        // A comment.
        an_expression()
    }
    foo => {
        let a = statement();
        an_expression()
    }
    bar => {}
    // Trailing comma on last item.
    foo => bar,
    baz => qux!(),
    lorem => {
        ipsum!()
    }
}
```

If the body is a single expression with no line comments and not a control flow expression, start it on the same line as the left-hand side. If not, then it must be in a block. Example:

```
match foo {
    // A combinable expression.
    foo => a_function_call(another_call(
        argument1,
        argument2,
    )),
    // A non-combinable expression
    bar => {
        a_function_call(
            another_call(
                argument1,
                argument2,
            ),
            another_argument,
        )
    }
}
```

Line-breaking

If using a block form on the right-hand side of a match arm makes it possible to avoid breaking on the left-hand side, do that:

```
// Assuming the following line does not fit in the max width
a_very_long_pattern | another_pattern => ALongStructName {
    ...
},
// Prefer this
a_very_long_pattern | another_pattern => {
    ALongStructName {
        ...
    }
}
// To splitting the pattern.
```

Never break after `=>` without using the block form of the body.

If the left-hand side must be split and there is an `if` clause, break before the `if` and block indent. In this case, always use a block body and start the body on a new line:

```
a_very_long_pattern | another_pattern
    if expr =>
{
    ...
}
```

If required to break the pattern, put each clause of the pattern on its own line with no additional indent, breaking before the `|`. If there is an `if` clause, use the above form:

```
a_very_long_pattern
| another_pattern
| yet_another_pattern
| a_forth_pattern => {
    ...
}
a_very_long_pattern
| another_pattern
| yet_another_pattern
| a_forth_pattern
    if expr =>
{
    ...
}
```

If the pattern is multi-line, and the last line is less wide than the indent, do not put the `if` clause on a new line. E.g.,

```
Token::Dimension {
     value,
     ref unit,
     ..
} if num_context.is_ok(context.parsing_mode, value) => {
    ...
}
```

If every clause in a pattern is *small*, but the whole pattern does not fit on one line, then format the pattern across multiple lines with as many clauses per line as possible. Again, break before a `|`:

```
foo | bar | baz
| qux => {
    ...
}
```

We define a pattern clause to be *small* if it fits on a single line and matches "small" in the following grammar:

```
small:
    - small_no_tuple
    - unary tuple constructor: `(` small_no_tuple `,` `)`
    - `&` small

small_no_tuple:
    - single token
    - `&` small_no_tuple
```

E.g., `&&Some(foo)` matches, `Foo(4, Bar)` does not.

Combinable expressions

Where a function call has a single argument, and that argument is formatted across multiple-lines, format the outer call as if it were a single-line call, if the result fits. Apply the same combining behaviour to any similar expressions which have multi-line, block-indented lists of sub-expressions delimited by parentheses (e.g., macros or tuple struct literals). E.g.,

```
foo(bar(
    an_expr,
    another_expr,
))

let x = foo(Bar {
    field: whatever,
});

foo(|param| {
    action();
    foo(param)
})

let x = combinable([
    an_expr,
    another_expr,
]);

let arr = [combinable(
    an_expr,
    another_expr,
)];
```

Apply this behavior recursively.

For a function with multiple arguments, if the last argument is a multi-line closure with an explicit block, there are no other closure arguments, and all the arguments and the first line of the closure fit on the first line, use the same combining behavior:

```
foo(first_arg, x, |param| {
    action();
    foo(param)
})
```

Ranges

Do not put spaces in ranges, e.g., `0..10`, `x..=y`, `..x.len()`, `foo...`.

When writing a range with both upper and lower bounds, if the line must be broken within the range, break before the range operator and block indent the second line:

```
a_long_expression
    ..another_long_expression
```

For the sake of indicating precedence, if either bound is a compound expression, use parentheses around it, e.g., `..(x + 1)`, `(x.f)..(x.f.len())`, or `0..(x - 10)`.

Hexadecimal literals

Hexadecimal literals may use upper- or lower-case letters, but they must not be mixed within the same literal. Projects should use the same case for all literals, but we do not make a recommendation for either lower- or upper-case.

Patterns

Format patterns like their corresponding expressions. See the section on `match` for additional formatting for patterns in match arms.

Types and Bounds

Single line formatting

- `[T]` no spaces
- `[T; expr]`, e.g., `[u32; 42]`, `[Vec<Foo>; 10 * 2 + foo()]` (space after colon, no spaces around square brackets)
- `*const T`, `*mut T` (no space after `*`, space before type)
- `&'a T`, `&T`, `&'a mut T`, `&mut T` (no space after `&`, single spaces separating other words)
- `unsafe extern "C" fn<'a, 'b, 'c>(T, U, V) -> W` or `fn()` (single spaces around keywords and sigils, and after commas, no trailing commas, no spaces around brackets)
- `!` gets treated like any other type name, `Name`
- `(A, B, C, D)` (spaces after commas, no spaces around parens, no trailing comma unless it is a one-tuple)
- `<Baz<T> as SomeTrait>::Foo::Bar` or `Foo::Bar` or `::Foo::Bar` (no spaces around `::` or angle brackets, single spaces around `as`)
- `Foo::Bar<T, U, V>` (spaces after commas, no trailing comma, no spaces around angle brackets)
- `T + T + T` (single spaces between types, and `+`).
- `impl T + T + T` (single spaces between keyword, types, and `+`).

Do not put space around parentheses used in types, e.g., `(Foo)`

Line breaks

Avoid breaking lines in types where possible. Prefer breaking at outermost scope, e.g., prefer

```
Foo<
    Bar,
    Baz<Type1, Type2>,
>
```

to

```
Foo<Bar, Baz<
    Type1,
    Type2,
>>
```

If a type requires line-breaks in order to fit, this section outlines where to break such types if necessary.

Break `[T; expr]` after the `;` if necessary.

Break function types following the rules for function declarations.

Break generic types following the rules for generics.

Break types with `+` by breaking before the `+` and block-indenting the subsequent lines. When

breaking such a type, break before *every* `+`:

```
impl Clone
    + Copy
    + Debug

Box<
    Clone
    + Copy
    + Debug
>
```

Precise capturing bounds

A `use<'a, T>` precise capturing bound is formatted as if it were a single path segment with non-turbofished angle-bracketed args, like a trait bound whose identifier is `use`.

```
fn foo() -> impl Sized + use<'a> {}

// is formatted analogously to:

fn foo() -> impl Sized + Use<'a> {}
```

Other style advice

Expressions

Prefer to use Rust's expression oriented nature where possible;

```
// use
let x = if y { 1 } else { 0 };
// not
let x;
if y {
    x = 1;
} else {
    x = 0;
}
```

Names

- Types shall be `UpperCamelCase`,
- Enum variants shall be `UpperCamelCase`,
- Struct fields shall be `snake_case`,
- Function and method names shall be `snake_case`,
- Local variables shall be `snake_case`,
- Macro names shall be `snake_case`,
- Constants (`const`s and immutable `static`s) shall be `SCREAMING_SNAKE_CASE`.
- When a name is forbidden because it is a reserved word (such as `crate`), either use a raw identifier (`r#crate`) or use a trailing underscore (`crate_`). Don't misspell the word (`krate`).

Modules

Avoid `#[path]` annotations where possible.

`Cargo.toml` conventions

Formatting conventions

Use the same line width and indentation as Rust code.

Put a blank line between the last key-value pair in a section and the header of the next section. Do not place a blank line between section headers and the key-value pairs in that section, or between key-value pairs in a section.

Version-sort key names within each section, with the exception of the `[package]` section. Put the `[package]` section at the top of the file; put the `name` and `version` keys in that order at the top of that section, followed by the remaining keys other than `description` in order, followed by the `description` at the end of that section.

Don't use quotes around any standard key names; use bare keys. Only use quoted keys for non-standard keys whose names require them, and avoid introducing such key names when possible. See the TOML specification for details.

Put a single space both before and after the `=` between a key and value. Do not indent any key names; start all key names at the start of a line.

Use multi-line strings (rather than newline escape sequences) for any string values that include multiple lines, such as the crate description.

For array values, such as a list of features, put the entire list on the same line as the key, if it fits. Otherwise, use block indentation: put a newline after the opening square bracket, indent each item by one indentation level, put a comma after each item (including the last), and put the closing square bracket at the start of a line by itself after the last item.

```
some_feature = [
    "another_feature",
    "yet_another_feature",
    "some_dependency?/some_feature",
]
```

For table values, such as a crate dependency with a path, write the entire table using curly braces and commas on the same line as the key if it fits. If the entire table does not fit on the same line as the key, separate it out into a separate section with key-value pairs:

```
[dependencies]
crate1 = { path = "crate1", version = "1.2.3" }

[dependencies.extremely_long_crate_name_goes_here]
path = "extremely_long_path_name_goes_right_here"
version = "4.5.6"
```

Metadata conventions

The authors list, if present, should consist of strings that each contain an author name followed by

an email address in angle brackets: `Full Name <email@address>`. It should not contain bare email addresses, or names without email addresses. (The authors list may also include a mailing list address without an associated name.)

The license field must contain a valid SPDX expression, using valid SPDX license names. (As an exception, by widespread convention, the license field may use `/` in place of `OR`; for example, `MIT/Apache-2.0`.)

The homepage field, if present, must consist of a single URL, including the scheme (e.g. `https://example.org/`, not just `example.org`.)

Within the description field, wrap text at 80 columns. Don't start the description field with the name of the crate (e.g. "cratename is a ..."); just describe the crate itself. If providing a multi-sentence description, the first sentence should go on a line by itself and summarize the crate, like the subject of an email or commit message; subsequent sentences can then describe the crate in more detail.

Guiding principles and rationale

When deciding on style guidelines, the style team follows these guiding principles (in rough priority order):

- readability
    - scan-ability
    - avoiding misleading formatting
    - accessibility - readable and editable by users using the widest variety of hardware, including non-visual accessibility interfaces
    - readability of code in contexts without syntax highlighting or IDE assistance, such as rustc error messages, diffs, grep, and other plain-text contexts
- aesthetics
    - sense of 'beauty'
    - consistent with other languages/tools
- specifics
    - compatibility with version control practices - preserving diffs, merge-friendliness, etc.
    - preventing rightward drift
    - minimising vertical space
- application
    - ease of manual application
    - ease of implementation (in `rustfmt`, and in other tools/editors/code generators)
    - internal consistency
    - simplicity of formatting rules

Rust style editions

The default Rust style evolves over time, as Rust does. However, to avoid breaking established code style, and CI jobs checking code style, changes to the default Rust style only appear in *style editions*.

Code written in a given Rust edition uses the corresponding Rust style edition by default. To make it easier to migrate code style separately from the semantic changes between Rust editions, formatting tools such as `rustfmt` allow updating the style edition separately from the Rust edition.

The current version of the style guide describes the latest Rust style edition. Each distinct past style will have a corresponding archived version of the style guide.

Note that archived versions of the style guide do not document formatting for newer Rust constructs that did not exist at the time that version of the style guide was archived. However, each style edition will still format all constructs valid in that Rust edition, with the style of newer constructs coming from the first subsequent style edition providing formatting rules for that construct (without any of the systematic/global changes from that style edition).

Not all Rust editions have corresponding changes to the Rust style. For instance, Rust 2015, Rust 2018, and Rust 2021 all use the same style edition.

Rust next style edition

- Never break within a nullary function call `func()` or a unit literal `()`.

Rust 2024 style edition

This style guide describes the Rust 2024 style edition. The Rust 2024 style edition is currently nightly-only and may change before the release of Rust 2024.

For a full history of changes in the Rust 2024 style edition, see the git history of the style guide. Notable changes in the Rust 2024 style edition include:

- Miscellaneous `rustfmt` bugfixes.
- Use version-sort (sort `x8`, `x16`, `x32`, `x64`, `x128` in that order).
- Change "ASCIIbetical" sort to Unicode-aware "non-lowercase before lowercase".

Rust 2015/2018/2021 style edition

The archived version of the style guide at https://github.com/rust-lang/rust/tree/37343f4a4d4ed7ad0891cb79e8eb25acf43fb821/src/doc/style-guide/src describes the style edition corresponding to Rust 2015, Rust 2018, and Rust 2021.

Nightly

This chapter documents style and formatting for nightly-only syntax. The rest of the style guide documents style for stable Rust syntax; nightly syntax only appears in this chapter. Each section here includes the name of the feature gate, so that searches (e.g. `git grep`) for a nightly feature in the Rust repository also turn up the style guide section.

Style and formatting for nightly-only syntax should be removed from this chapter and integrated into the appropriate sections of the style guide at the time of stabilization.

There is no guarantee of the stability of this chapter in contrast to the rest of the style guide. Refer to the style team policy for nightly formatting procedure regarding breaking changes to this chapter.

Frontmatter

Location: Placed before comments and attributes in the root.

Tracking issue: #136889

Feature gate: `frontmatter`

There should be no blank lines between the frontmatter and either the start of the file or a shebang. There can be zero or one line between the frontmatter and any following content.

The frontmatter fences should use the minimum number of dashes necessary for the contained content (one more than the longest series of initial dashes in the content, with a minimum of 3 to be recognized as frontmatter delimiters). If an infostring is present after the opening fence, there should be one space separating them. The frontmatter fence lines should not have trailing whitespace.

```
#!/usr/bin/env cargo
--- cargo
[dependencies]
regex = "1"
---

fn main() {}
```