Configuring Rustfmt

Rustfmt is designed to be very configurable. You can create a TOML file called `rustfmt.toml` or `.rustfmt.toml`, place it in the project or any other parent directory and it will apply the options in that file. If none of these directories contain such a file, both your home directory and a directory called `rustfmt` in your global config directory (e.g. `.config/rustfmt/`) are checked as well.

A possible content of `rustfmt.toml` or `.rustfmt.toml` might look like this:

```toml
indent_style = "Block"
reorder_imports = false
```

Each configuration option is either stable or unstable. Stable options can be used directly, while unstable options are opt-in. To enable unstable options, set `unstable_features = true` in `rustfmt.toml` or pass `--unstable-features` to rustfmt.

Configuration Options

Below you find a detailed visual guide on all the supported configuration options of rustfmt:

`array_width`

Maximum width of an array literal before falling back to vertical formatting.

- **Default value**: `60`
- **Possible values**: any positive integer that is less than or equal to the value specified for `max_width`
- **Stable**: Yes

By default this option is set as a percentage of `max_width` provided by `use_small_heuristics`, but a value set directly for `array_width` will take precedence.

See also `max_width` and `use_small_heuristics`

`attr_fn_like_width`

Maximum width of the args of a function-like attributes before falling back to vertical formatting.

- **Default value**: `70`
- **Possible values**: any positive integer that is less than or equal to the value specified for `max_width`
- **Stable**: Yes

By default this option is set as a percentage of `max_width` provided by `use_small_heuristics`, but a value set directly for `attr_fn_like_width` will take precedence.

See also `max_width` and `use_small_heuristics`

`binop_separator`

Where to put a binary operator when a binary expression goes multiline.

- **Default value**: `"Front"`
- **Possible values**: `"Front"`, `"Back"`
- **Stable**: No (tracking issue: #3368)

`"Front"` (default):

```
fn main() {
    let or = foofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoo
        || barbarbarbarbarbarbarbarbarbarbarbarbarbarbarbar;

    let sum = 123456789012345678901234567890
        + 123456789012345678901234567890
        + 123456789012345678901234567890;

    let range = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
        ..bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb;
}
```

`"Back"`:

```
fn main() {
    let or = foofoofoofoofoofoofoofoofoofoofoofoofoofoofoofoo ||
        barbarbarbarbarbarbarbarbarbarbarbarbarbarbarbar;

    let sum = 123456789012345678901234567890 +
        123456789012345678901234567890 +
        123456789012345678901234567890;

    let range = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa..
        bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb;
}
```

`blank_lines_lower_bound`

Minimum number of blank lines which must be put between items. If two items have fewer blank lines between them, additional blank lines are inserted.

- **Default value**: `0`
- **Possible values**: *unsigned integer*
- **Stable**: No (tracking issue: #3382)

Example

Original Code (rustfmt will not change it with the default value of `0`):

```
#![rustfmt::skip]

fn foo() {
    println!("a");
}
```

```rust
fn bar() {
    println!("b");
    println!("c");
}
```

`1`

```rust
fn foo() {

    println!("a");
}

fn bar() {

    println!("b");

    println!("c");
}
```

`blank_lines_upper_bound`

Maximum number of blank lines which can be put between items. If more than this number of consecutive empty lines are found, they are trimmed down to match this integer.

- **Default value**: `1`
- **Possible values**: any non-negative integer
- **Stable**: No (tracking issue: #3381)

Example

Original Code:

```rust
#![rustfmt::skip]

fn foo() {
    println!("a");
}



fn bar() {
    println!("b");


    println!("c");
}
```

`1` (default):

```rust
fn foo() {
    println!("a");
}

fn bar() {
    println!("b");

    println!("c");
}
```

`2`:

```rust
fn foo() {
    println!("a");
}


fn bar() {
    println!("b");


    println!("c");
}
```

See also: `blank_lines_lower_bound`

`brace_style`

Brace style for items

- **Default value**: `"SameLineWhere"`
- **Possible values**: `"AlwaysNextLine"`, `"PreferSameLine"`, `"SameLineWhere"`
- **Stable**: No (tracking issue: #3376)

Functions

`"SameLineWhere"` (default):

```rust
fn lorem() {
    // body
}

fn lorem(ipsum: usize) {
    // body
}

fn lorem<T>(ipsum: T)
where
    T: Add + Sub + Mul + Div,
{
    // body
}
```

`"AlwaysNextLine"`:

```
fn lorem()
{
    // body
}

fn lorem(ipsum: usize)
{
    // body
}

fn lorem<T>(ipsum: T)
where
    T: Add + Sub + Mul + Div,
{
    // body
}
```

`"PreferSameLine"`:

```
fn lorem() {
    // body
}

fn lorem(ipsum: usize) {
    // body
}

fn lorem<T>(ipsum: T)
where
    T: Add + Sub + Mul + Div, {
    // body
}
```

Structs and enums

`"SameLineWhere"` (default):

```
struct Lorem {
    ipsum: bool,
}

struct Dolor<T>
where
    T: Eq,
{
    sit: T,
}
```

`"AlwaysNextLine"`:

```rust
struct Lorem
{
    ipsum: bool,
}

struct Dolor<T>
where
    T: Eq,
{
    sit: T,
}
```

`"PreferSameLine"`:

```rust
struct Lorem {
    ipsum: bool,
}

struct Dolor<T>
where
    T: Eq, {
    sit: T,
}
```

`chain_width`

Maximum width of a chain to fit on one line.

- **Default value**: `60`
- **Possible values**: any positive integer that is less than or equal to the value specified for `max_width`
- **Stable**: Yes

By default this option is set as a percentage of `max_width` provided by `use_small_heuristics`, but a value set directly for `chain_width` will take precedence.

See also `max_width` and `use_small_heuristics`

`color`

Whether to use colored output or not.

- **Default value**: `"Auto"`
- **Possible values**: "Auto", "Always", "Never"

- **Stable**: No (tracking issue: #3385)

`combine_control_expr`

Combine control expressions with function calls.

- **Default value**: `true`
- **Possible values**: `true`, `false`
- **Stable**: No (tracking issue: #3369)

`true` (default):

```rust
fn example() {
    // If
    foo!(if x {
        foo();
    } else {
        bar();
    });

    // IfLet
    foo!(if let Some(..) = x {
        foo();
    } else {
        bar();
    });

    // While
    foo!(while x {
        foo();
        bar();
    });

    // WhileLet
    foo!(while let Some(..) = x {
        foo();
        bar();
    });

    // ForLoop
    foo!(for x in y {
        foo();
        bar();
    });

    // Loop
    foo!(loop {
        foo();
        bar();
    });
}
```

`false`:

```
fn example() {
    // If
    foo!(
        if x {
            foo();
        } else {
            bar();
        }
    );

    // IfLet
    foo!(
        if let Some(..) = x {
            foo();
        } else {
            bar();
        }
    );

    // While
    foo!(
        while x {
            foo();
            bar();
        }
    );

    // WhileLet
    foo!(
        while let Some(..) = x {
            foo();
            bar();
        }
    );

    // ForLoop
    foo!(
        for x in y {
            foo();
            bar();
        }
    );

    // Loop
    foo!(
        loop {
            foo();
            bar();
        }
    );
}
```

`comment_width`

Maximum length of comments. No effect unless `wrap_comments = true`.

- **Default value**: `80`
- **Possible values**: any positive integer
- **Stable**: No (tracking issue: #3349)

Note: A value of `0` results in `wrap_comments` being applied regardless of a line's width.

`80` (default; comments shorter than `comment_width`):

```rust
// Lorem ipsum dolor sit amet, consectetur adipiscing elit.
```

`60` (comments longer than `comment_width`):

```rust
// Lorem ipsum dolor sit amet,
// consectetur adipiscing elit.
```

See also `wrap_comments`.

`condense_wildcard_suffixes`

Replace strings of _ wildcards by a single .. in tuple patterns

- **Default value**: `false`
- **Possible values**: `true`, `false`
- **Stable**: No (tracking issue: #3384)

`false` (default):

```rust
fn main() {
    let (lorem, ipsum, _, _) = (1, 2, 3, 4);
    let (lorem, ipsum, ..) = (1, 2, 3, 4);
}
```

`true`:

```rust
fn main() {
    let (lorem, ipsum, ..) = (1, 2, 3, 4);
}
```

`control_brace_style`

Brace style for control flow constructs

- **Default value**: `"AlwaysSameLine"`
- **Possible values**: `"AlwaysNextLine"`, `"AlwaysSameLine"`, `"ClosingNextLine"`
- **Stable**: No (tracking issue: #3377)

`"AlwaysSameLine"` (default):

```rust
fn main() {
    if lorem {
        println!("ipsum!");
    } else {
        println!("dolor!");
    }
}
```

`"AlwaysNextLine"`:

```rust
fn main() {
    if lorem
    {
        println!("ipsum!");
    }
    else
    {
        println!("dolor!");
    }
}
```

`"ClosingNextLine"`:

```rust
fn main() {
    if lorem {
        println!("ipsum!");
    }
    else {
        println!("dolor!");
    }
}
```

`disable_all_formatting`

Don't reformat anything.

Note that this option may be soft-deprecated in the future once the ignore option is stabilized. Nightly toolchain users are encouraged to use ignore instead when possible.

- **Default value**: `false`
- **Possible values**: `true`, `false`

- **Stable**: Yes

`edition`

Specifies which edition is used by the parser.

- **Default value**: `"2015"`
- **Possible values**: `"2015"`, `"2018"`, `"2021"`
- **Stable**: Yes

Rustfmt is able to pick up the edition used by reading the `Cargo.toml` file if executed through the Cargo's formatting tool `cargo fmt`. Otherwise, the edition needs to be specified in your config file:

```toml
edition = "2018"
```

`empty_item_single_line`

Put empty-body functions and impls on a single line

- **Default value**: `true`
- **Possible values**: `true`, `false`
- **Stable**: No (tracking issue: #3356)

`true` (default):

```rust
fn lorem() {}

impl Lorem {}
```

`false`:

```rust
fn lorem() {
}

impl Lorem {
}
```

See also `brace_style`, `control_brace_style`.

`enum_discrim_align_threshold`

The maximum length of enum variant having discriminant, that gets vertically aligned with others. Variants without discriminants would be ignored for the purpose of alignment.

Note that this is not how much whitespace is inserted, but instead the longest variant name that doesn't get ignored when aligning.

- **Default value** : 0
- **Possible values**: any positive integer
- **Stable**: No (tracking issue: #3372)

`0` **(default):**

```rust
enum Bar {
    A = 0,
    Bb = 1,
    RandomLongVariantGoesHere = 10,
    Ccc = 71,
}

enum Bar {
    VeryLongVariantNameHereA = 0,
    VeryLongVariantNameHereBb = 1,
    VeryLongVariantNameHereCcc = 2,
}
```

`20` **:**

```rust
enum Foo {
    A   = 0,
    Bb  = 1,
    RandomLongVariantGoesHere = 10,
    Ccc = 2,
}

enum Bar {
    VeryLongVariantNameHereA = 0,
    VeryLongVariantNameHereBb = 1,
    VeryLongVariantNameHereCcc = 2,
}
```

`error_on_line_overflow`

Error if Rustfmt is unable to get all lines within `max_width`, except for comments and string literals. If this happens, then it is a bug in Rustfmt. You might be able to work around the bug by refactoring your code to avoid long/complex expressions, usually by extracting a local variable or using a shorter name.

- **Default value**: `false`
- **Possible values**: `true`, `false`
- **Stable**: No (tracking issue: #3391)

See also `max_width`.

`error_on_unformatted`

Error if unable to get comments or string literals within `max_width`, or they are left with trailing whitespaces.

- **Default value**: `false`
- **Possible values**: `true`, `false`
- **Stable**: No (tracking issue: #3392)

`fn_args_layout`

Control the layout of arguments in a function

- **Default value**: `"Tall"`
- **Possible values**: `"Compressed"`, `"Tall"`, `"Vertical"`
- **Stable**: Yes

`"Tall"` **(default):**

```
trait Lorem {
    fn lorem(ipsum: Ipsum, dolor: Dolor, sit: Sit, amet: Amet);

    fn lorem(ipsum: Ipsum, dolor: Dolor, sit: Sit, amet: Amet) {
        // body
    }

    fn lorem(
        ipsum: Ipsum,
        dolor: Dolor,
        sit: Sit,
        amet: Amet,
        consectetur: Consectetur,
        adipiscing: Adipiscing,
        elit: Elit,
    );

    fn lorem(
        ipsum: Ipsum,
        dolor: Dolor,
        sit: Sit,
        amet: Amet,
        consectetur: Consectetur,
        adipiscing: Adipiscing,
        elit: Elit,
    ) {
        // body
    }
}
```

`"Compressed"`**:**

```
trait Lorem {
    fn lorem(ipsum: Ipsum, dolor: Dolor, sit: Sit, amet: Amet);

    fn lorem(ipsum: Ipsum, dolor: Dolor, sit: Sit, amet: Amet) {
        // body
    }

    fn lorem(
        ipsum: Ipsum, dolor: Dolor, sit: Sit, amet: Amet, consectetur: Consectetur,
        adipiscing: Adipiscing, elit: Elit,
    );

    fn lorem(
        ipsum: Ipsum, dolor: Dolor, sit: Sit, amet: Amet, consectetur: Consectetur,
        adipiscing: Adipiscing, elit: Elit,
    ) {
        // body
    }
}
```

`"Vertical"`:

```
trait Lorem {
    fn lorem(
        ipsum: Ipsum,
        dolor: Dolor,
        sit: Sit,
        amet: Amet,
    );

    fn lorem(
        ipsum: Ipsum,
        dolor: Dolor,
        sit: Sit,
        amet: Amet,
    ) {
        // body
    }

    fn lorem(
        ipsum: Ipsum,
        dolor: Dolor,
        sit: Sit,
        amet: Amet,
        consectetur: Consectetur,
        adipiscing: Adipiscing,
        elit: Elit,
    );

    fn lorem(
        ipsum: Ipsum,
        dolor: Dolor,
        sit: Sit,
        amet: Amet,
        consectetur: Consectetur,
        adipiscing: Adipiscing,
        elit: Elit,
    ) {
        // body
    }
}
```

`fn_call_width`

Maximum width of the args of a function call before falling back to vertical formatting.

- **Default value**: `60`
- **Possible values**: any positive integer that is less than or equal to the value specified for `max_width`
- **Stable**: Yes

By default this option is set as a percentage of `max_width` provided by `use_small_heuristics`, but a value set directly for `fn_call_width` will take precedence.

See also `max_width` and `use_small_heuristics`

`fn_single_line`

Put single-expression functions on a single line

- **Default value**: `false`
- **Possible values**: `true`, `false`
- **Stable**: No (tracking issue: #3358)

`false` (default):

```
fn lorem() -> usize {
    42
}

fn lorem() -> usize {
    let ipsum = 42;
    ipsum
}
```

`true`:

```
fn lorem() -> usize { 42 }

fn lorem() -> usize {
    let ipsum = 42;
    ipsum
}
```

See also `control_brace_style`.

`force_explicit_abi`

Always print the abi for extern items

- **Default value**: `true`
- **Possible values**: `true`, `false`
- **Stable**: Yes

Note: Non-"C" ABIs are always printed. If `false` then "C" is removed.

`true` (default):

```rust
extern "C" {
    pub static lorem: c_int;
}
```

`false`:

```rust
extern {
    pub static lorem: c_int;
}
```

`force_multiline_blocks`

Force multiline closure and match arm bodies to be wrapped in a block

- **Default value**: `false`
- **Possible values**: `false`, `true`
- **Stable**: No (tracking issue: [#3374](#3374))

`false` (default):

```rust
fn main() {
    result.and_then(|maybe_value| match maybe_value {
        None => foo(),
        Some(value) => bar(),
    });

    match lorem {
        None => |ipsum| {
            println!("Hello World");
        },
        Some(dolor) => foo(),
    }
}
```

`true`:

```rust
fn main() {
    result.and_then(|maybe_value| {
        match maybe_value {
            None => foo(),
            Some(value) => bar(),
        }
    });

    match lorem {
        None => {
            |ipsum| {
                println!("Hello World");
            }
        }
        Some(dolor) => foo(),
    }
}
```

`format_code_in_doc_comments`

Format code snippet included in doc comments.

- **Default value**: `false`
- **Possible values**: `true`, `false`
- **Stable**: No (tracking issue: #3348)

`false` (default):

```rust
/// Adds one to the number given.
///
/// # Examples
///
/// ```rust
/// let five=5;
///
/// assert_eq!(
///     6,
///     add_one(5)
/// );
/// # fn add_one(x: i32) -> i32 {
/// #       x + 1
/// # }
/// ```
fn add_one(x: i32) -> i32 {
    x + 1
}
```

`true`

```rust
/// Adds one to the number given.
///
/// # Examples
///
/// ```rust
/// let five = 5;
```

```
///
/// assert_eq!(6, add_one(5));
/// # fn add_one(x: i32) -> i32 {
/// #     x + 1
/// # }
/// ```
fn add_one(x: i32) -> i32 {
    x + 1
}
```

`format_generated_files`

Format generated files. A file is considered generated if any of the first five lines contain a `@generated` comment marker. By default, generated files are reformatted, i. e. `@generated` marker is ignored. This option is currently ignored for stdin (`@generated` in stdin is ignored.)

- **Default value**: `true`
- **Possible values**: `true` , `false`
- **Stable**: No (tracking issue: #5080)

`format_macro_matchers`

Format the metavariable matching patterns in macros.

- **Default value**: `false`
- **Possible values**: `true` , `false`
- **Stable**: No (tracking issue: #3354)

`false` (default):

```
macro_rules! foo {
    ($a: ident : $b: ty) => {
        $a(42): $b;
    };
    ($a: ident $b: ident $c: ident) => {
        $a = $b + $c;
    };
}
```

`true` :

```
macro_rules! foo {
    ($a:ident : $b:ty) => {
        $a(42): $b;
    };
    ($a:ident $b:ident $c:ident) => {
        $a = $b + $c;
    };
}
```

See also `format_macro_bodies`.

`format_macro_bodies`

Format the bodies of macros.

- **Default value**: `true`
- **Possible values**: `true`, `false`
- **Stable**: No (tracking issue: #3355)

`true` (default):

```
macro_rules! foo {
    ($a: ident : $b: ty) => {
        $a(42): $b;
    };
    ($a: ident $b: ident $c: ident) => {
        $a = $b + $c;
    };
}
```

`false`:

```
macro_rules! foo {
    ($a: ident : $b: ty) => { $a(42): $b; };
    ($a: ident $b: ident $c: ident) => { $a=$b+$c; };
}
```

See also `format_macro_matchers`.

`format_strings`

Format string literals where necessary

- **Default value**: `false`
- **Possible values**: `true`, `false`
- **Stable**: No (tracking issue: #3353)

`false` (default):

```
fn main() {
    let lorem = "ipsum dolor sit amet consectetur adipiscing elit lorem ipsum dolor
sit amet consectetur adipiscing";
}
```

`true`:

```
fn main() {
    let lorem = "ipsum dolor sit amet consectetur adipiscing elit lorem ipsum dolor
sit amet \
                 consectetur adipiscing";
}
```

See also `max_width`.

`hard_tabs`

Use tab characters for indentation, spaces for alignment

- **Default value**: `false`
- **Possible values**: `true`, `false`
- **Stable**: Yes

`false` (default):

```rust
fn lorem() -> usize {
    42 // spaces before 42
}
```

`true`:

```rust
fn lorem() -> usize {
	42 // tabs before 42
}
```

See also: `tab_spaces`.

`hex_literal_case`

Control the case of the letters in hexadecimal literal values

- **Default value**: `Preserve`
- **Possible values**: `Upper`, `Lower`
- **Stable**: No (tracking issue: #5081)

`hide_parse_errors`

Do not show parse errors if the parser failed to parse files.

- **Default value**: `false`
- **Possible values**: `true`, `false`
- **Stable**: No (tracking issue: #3390)

`ignore`

Skip formatting files and directories that match the specified pattern. The pattern format is the same as .gitignore. Be sure to use Unix/forwardslash `/` style paths. This path style will work on all platforms. Windows style paths with backslashes `\` are not supported.

- **Default value**: format every file
- **Possible values**: See an example below
- **Stable**: No (tracking issue: #3395)

Example

If you want to ignore specific files, put the following to your config file:

```toml
ignore = [
    "src/types.rs",
    "src/foo/bar.rs",
]
```

If you want to ignore every file under `examples/`, put the following to your config file:

```toml
ignore = [
    "examples",
]
```

If you want to ignore every file under the directory where you put your rustfmt.toml:

```toml
ignore = ["/"]
```

`imports_indent`

Indent style of imports

- **Default Value**: `"Block"`
- **Possible values**: `"Block"`, `"Visual"`
- **Stable**: No (tracking issue: #3360)

`"Block"` (default):

```rust
use foo::{
    xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy,
    zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz,
};
```

`"Visual"`:

```rust
use foo::{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx, yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy,
          zzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzzz};
```

See also: `imports_layout`.

`imports_layout`

Item layout inside a imports block

- **Default value**: "Mixed"
- **Possible values**: "Horizontal", "HorizontalVertical", "Mixed", "Vertical"
- **Stable**: No (tracking issue: #3361)

`"Mixed"` (default):

```
use foo::{xxxxxxxxxxxxxxxxxx, yyyyyyyyyyyyyyyyyy, zzzzzzzzzzzzzzzzzz};

use foo::{
    aaaaaaaaaaaaaaaaaa, bbbbbbbbbbbbbbbbbb, cccccccccccccccccc, dddddddddddddddddd,
    eeeeeeeeeeeeeeeeee, ffffffffffffffffff,
};
```

`"Horizontal"`:

Note: This option forces all imports onto one line and may exceed `max_width`.

```
use foo::{xxx, yyy, zzz};

use foo::{aaa, bbb, ccc, ddd, eee, fff};
```

`"HorizontalVertical"`:

```
use foo::{xxxxxxxxxxxxxxxxxx, yyyyyyyyyyyyyyyyyy, zzzzzzzzzzzzzzzzzz};

use foo::{
    aaaaaaaaaaaaaaaaaa,
    bbbbbbbbbbbbbbbbbb,
    cccccccccccccccccc,
    dddddddddddddddddd,
    eeeeeeeeeeeeeeeeee,
    ffffffffffffffffff,
};
```

`"Vertical"`:

```
use foo::{
    xxx,
    yyy,
    zzz,
};

use foo::{
    aaa,
    bbb,
    ccc,
    ddd,
    eee,
    fff,
};
```

`indent_style`

Indent on expressions or items.

- **Default value**: `"Block"`
- **Possible values**: `"Block"`, `"Visual"`
- **Stable**: No (tracking issue: #3346)

Array

`"Block"` **(default):**

```
fn main() {
    let lorem = vec![
        "ipsum",
        "dolor",
        "sit",
        "amet",
        "consectetur",
        "adipiscing",
        "elit",
    ];
}
```

`"Visual"`**:**

```
fn main() {
    let lorem = vec!["ipsum",
                     "dolor",
                     "sit",
                     "amet",
                     "consectetur",
                     "adipiscing",
                     "elit"];
}
```

Control flow

`"Block"` **(default):**

```
fn main() {
    if lorem_ipsum
        && dolor_sit
        && amet_consectetur
        && lorem_sit
        && dolor_consectetur
```

```
          && amet_ipsum
          && lorem_consectetur
    {
        // ...
    }
}
```

`"Visual"`:

```
fn main() {
    if lorem_ipsum
       && dolor_sit
       && amet_consectetur
       && lorem_sit
       && dolor_consectetur
       && amet_ipsum
       && lorem_consectetur
    {
        // ...
    }
}
```

See also: `control_brace_style`.

Function arguments

`"Block"` (default):

```
fn lorem() {}

fn lorem(ipsum: usize) {}

fn lorem(
    ipsum: usize,
    dolor: usize,
    sit: usize,
    amet: usize,
    consectetur: usize,
    adipiscing: usize,
    elit: usize,
) {
    // body
}
```

`"Visual"`:

```
fn lorem() {}

fn lorem(ipsum: usize) {}
fn lorem(ipsum: usize,
         dolor: usize,
         sit: usize,
         amet: usize,
         consectetur: usize,
         adipiscing: usize,
         elit: usize) {
    // body
}
```

Function calls

`"Block"` (default):

```
fn main() {
    lorem(
        "lorem",
        "ipsum",
        "dolor",
        "sit",
        "amet",
        "consectetur",
        "adipiscing",
        "elit",
    );
}
```

`"Visual"`:

```
fn main() {
    lorem("lorem",
          "ipsum",
          "dolor",
          "sit",
          "amet",
          "consectetur",
          "adipiscing",
          "elit");
}
```

Generics

`"Block"` (default):

```
fn lorem<
    Ipsum: Eq = usize,
    Dolor: Eq = usize,
    Sit: Eq = usize,
    Amet: Eq = usize,
    Adipiscing: Eq = usize,
    Consectetur: Eq = usize,
    Elit: Eq = usize,
>(
    ipsum: Ipsum,
    dolor: Dolor,
    sit: Sit,
    amet: Amet,
    adipiscing: Adipiscing,
    consectetur: Consectetur,
    elit: Elit,
) -> T {
    // body
}
```

`"Visual"`:

```
fn lorem<Ipsum: Eq = usize,
         Dolor: Eq = usize,
         Sit: Eq = usize,
         Amet: Eq = usize,
         Adipiscing: Eq = usize,
         Consectetur: Eq = usize,
         Elit: Eq = usize>(
    ipsum: Ipsum,
    dolor: Dolor,
    sit: Sit,
    amet: Amet,
    adipiscing: Adipiscing,
    consectetur: Consectetur,
    elit: Elit)
    -> T {
    // body
}
```

Struct

`"Block"` **(default):**

```
fn main() {
    let lorem = Lorem {
        ipsum: dolor,
        sit: amet,
    };
}
```

`"Visual"`:

```rust
fn main() {
    let lorem = Lorem { ipsum: dolor,
                        sit: amet };
}
```

See also: `struct_lit_single_line` , `indent_style` .

Where predicates

`"Block"` **(default):**

```rust
fn lorem<Ipsum, Dolor, Sit, Amet>() -> T
where
    Ipsum: Eq,
    Dolor: Eq,
    Sit: Eq,
    Amet: Eq,
{
    // body
}
```

`"Visual"` **:**

```rust
fn lorem<Ipsum, Dolor, Sit, Amet>() -> T
    where Ipsum: Eq,
          Dolor: Eq,
          Sit: Eq,
          Amet: Eq
{
    // body
}
```

`inline_attribute_width`

Write an item and its attribute on the same line if their combined width is below a threshold

- **Default value**: 0
- **Possible values**: any positive integer
- **Stable**: No (tracking issue: #3343)

Example

`0` **(default):**

```rust
#[cfg(feature = "alloc")]
use core::slice;
```

`50` **:**

```rust
#[cfg(feature = "alloc")] use core::slice;
```

`license_template_path`

Check whether beginnings of files match a license template.

- **Default value**: `""`
- **Possible values**: path to a license template file
- **Stable**: No (tracking issue: #3352)

A license template is a plain text file which is matched literally against the beginning of each source file, except for `{}`-delimited blocks, which are matched as regular expressions. The following license template therefore matches strings like `// Copyright 2017 The Rust Project Developers.`, `// Copyright 2018 The Rust Project Developers.`, etc.:

```
// Copyright {\d+} The Rust Project Developers.
```

`\{`, `\}` and `\\` match literal braces / backslashes.

`match_arm_blocks`

Controls whether arm bodies are wrapped in cases where the first line of the body cannot fit on the same line as the `=>` operator.

The Style Guide requires that bodies are block wrapped by default if a line break is required after the `=>`, but this option can be used to disable that behavior to prevent wrapping arm bodies in that event, so long as the body does not contain multiple statements nor line comments.

- **Default value**: `true`
- **Possible values**: `true`, `false`
- **Stable**: No (tracking issue: #3373)

`true` (default):

```rust
fn main() {
    match lorem {
        ipsum => {
foooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo(x)
        }
        dolor => println!("{}", sit),
        sit => foo(
            "foooooooooooooooooooooooo",
            "baaaaaaaaaaaaaaaaaaaaaaaarr",
            "baaaaaaaaaaaaaaaaaaaazzzzzzzzzzzz",
            "qqqqqqqqquuuuuuuuuuuuuuuuuuuuuuuuuuxxx",
        ),
    }
}
```

`false`:

```rust
fn main() {
    match lorem {
        lorem =>

foooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo(x),

        ipsum => println!("{}", sit),
        sit => foo(
            "foooooooooooooooooooooooo",
            "baaaaaaaaaaaaaaaaaaaaaaaarr",
            "baaaaaaaaaaaaaaaaaaaazzzzzzzzzzzzz",
            "qqqqqqqqquuuuuuuuuuuuuuuuuuuuuuuuuuxxx",
        ),
    }
}
```

See also: [`match_block_trailing_comma`](#match_block_trailing_comma).

`match_arm_leading_pipes`

Controls whether to include a leading pipe on match arms

- **Default value**: `Never`
- **Possible values**: `Always`, `Never`, `Preserve`
- **Stable**: Yes

`Never` (default):

```rust
// Leading pipes are removed from this:
// fn foo() {
//     match foo {
//         | "foo" | "bar" => {}
//         | "baz"
//         | "something relatively long"
//         | "something really really really realllllllllllllly long" => println!
("x"),
//         | "qux" => println!("y"),
//         _ => {}
//     }
// }

// Becomes
fn foo() {
    match foo {
        "foo" | "bar" => {}
        "baz"
        | "something relatively long"
        | "something really really really realllllllllllllly long" => println!("x"),
        "qux" => println!("y"),
        _ => {}
    }
}
```

`Always`:

```
// Leading pipes are emitted on all arms of this:
// fn foo() {
//     match foo {
//         "foo" | "bar" => {}
//         "baz"
//         | "something relatively long"
//         | "something really really really realllllllllllllly long" => println!
("x"),
//         "qux" => println!("y"),
//         _ => {}
//     }
// }

// Becomes:
fn foo() {
    match foo {
        | "foo" | "bar" => {}
        | "baz"
        | "something relatively long"
        | "something really really really realllllllllllllly long" => println!("x"),
        | "qux" => println!("y"),
        | _ => {}
    }
}
```

`Preserve`:

```
fn foo() {
    match foo {
        | "foo" | "bar" => {}
        | "baz"
        | "something relatively long"
        | "something really really really realllllllllllllly long" => println!("x"),
        | "qux" => println!("y"),
        _ => {}
    }

    match baz {
        "qux" => {}
        "foo" | "bar" => {}
        _ => {}
    }
}
```

`match_block_trailing_comma`

Put a trailing comma after a block based match arm (non-block arms are not affected)

- **Default value**: `false`
- **Possible values**: `true`, `false`
- **Stable**: Yes

`false` (default):

```rust
fn main() {
    match lorem {
        Lorem::Ipsum => {
            println!("ipsum");
        }
        Lorem::Dolor => println!("dolor"),
    }
}
```

`true`:

```rust
fn main() {
    match lorem {
        Lorem::Ipsum => {
            println!("ipsum");
        },
        Lorem::Dolor => println!("dolor"),
    }
}
```

See also: `trailing_comma`, `match_arm_blocks`.

`max_width`

Maximum width of each line

- **Default value**: `100`
- **Possible values**: any positive integer
- **Stable**: Yes

See also `error_on_line_overflow`.

`merge_derives`

Merge multiple derives into a single one.

- **Default value**: `true`
- **Possible values**: `true`, `false`
- **Stable**: Yes

`true` (default):

```
#[derive(Eq, PartialEq, Debug, Copy, Clone)]
pub enum Foo {}
```

`false`:

```
#[derive(Eq, PartialEq, Debug, Copy, Clone)]
pub enum Bar {}

#[derive(Eq, PartialEq)]
#[derive(Debug)]
#[derive(Copy, Clone)]
pub enum Foo {}
```

`imports_granularity`

How imports should be grouped into `use` statements. Imports will be merged or split to the configured level of granularity.

- **Default value**: `Preserve`
- **Possible values**: `Preserve`, `Crate`, `Module`, `Item`, `One`
- **Stable**: No (tracking issue: #4991)

`Preserve` (default):

Do not change the granularity of any imports and preserve the original structure written by the developer.

```
use foo::b;
use foo::b::{f, g};
use foo::{a, c, d::e};
use qux::{h, i};
```

`Crate`:

Merge imports from the same crate into a single `use` statement. Conversely, imports from different crates are split into separate statements.

```
use foo::{
    a, b,
    b::{f, g},
    c,
    d::e,
};
use qux::{h, i};
```

`Module`:

Merge imports from the same module into a single `use` statement. Conversely, imports from different modules are split into separate statements.

```
use foo::b::{f, g};
use foo::d::e;
use foo::{a, b, c};
use qux::{h, i};
```

`Item`:

Flatten imports so that each has its own `use` statement.

```
use foo::a;
use foo::b;
use foo::b::f;
use foo::b::g;
use foo::c;
use foo::d::e;
use qux::h;
use qux::i;
```

`One`:

Merge all imports into a single `use` statement as long as they have the same visibility.

```
pub use foo::{x, y};
use {
    bar::{
        a,
        b::{self, f, g},
        c,
        d::e,
    },
    qux::{h, i},
};
```

`merge_imports`

This option is deprecated. Use `imports_granularity = "Crate"` instead.

- **Default value**: `false`
- **Possible values**: `true`, `false`

`false` (default):

```
use foo::{a, c, d};
use foo::{b, g};
use foo::{e, f};
```

`true`:

```
use foo::{a, b, c, d, e, f, g};
```

`newline_style`

Unix or Windows line endings

- **Default value**: `"Auto"`
- **Possible values**: `"Auto"`, `"Native"`, `"Unix"`, `"Windows"`
- **Stable**: Yes

`Auto` (default):

The newline style is detected automatically on a per-file basis. Files with mixed line endings will be converted to the first detected line ending style.

`Native`

Line endings will be converted to `\r\n` on Windows and `\n` on all other platforms.

`Unix`

Line endings will be converted to `\n`.

`Windows`

Line endings will be converted to `\r\n`.

`normalize_comments`

Convert /* */ comments to // comments where possible

- **Default value**: `false`
- **Possible values**: `true`, `false`
- **Stable**: No (tracking issue: #3350)

`false` (default):

```
// Lorem ipsum:
fn dolor() -> usize {}

/* sit amet: */
fn adipiscing() -> usize {}
```

`true`:

```
// Lorem ipsum:
fn dolor() -> usize {}
```

```rust
// sit amet:
fn adipiscing() -> usize {}
```

`normalize_doc_attributes`

Convert `#![doc]` and `#[doc]` attributes to `//!` and `///` doc comments.

- **Default value**: `false`
- **Possible values**: `true`, `false`
- **Stable**: No (tracking issue: #3351)

`false` (default):

```rust
#![doc = "Example documentation"]

#[doc = "Example item documentation"]
pub enum Bar {}

/// Example item documentation
pub enum Foo {}
```

`true`:

```rust
//! Example documentation

/// Example item documentation
pub enum Foo {}
```

`overflow_delimited_expr`

When structs, slices, arrays, and block/array-like macros are used as the last argument in an expression list, allow them to overflow (like blocks/closures) instead of being indented on a new line.

- **Default value**: `false`
- **Possible values**: `true`, `false`
- **Stable**: No (tracking issue: #3370)

`false` (default):

```rust
fn example() {
    foo(ctx, |param| {
        action();
        foo(param)
    });

    foo(
        ctx,
        Bar {
            x: value,
            y: value2,
        },
    );

    foo(
        ctx,
        &[
            MAROON_TOMATOES,
            PURPLE_POTATOES,
            ORGANE_ORANGES,
            GREEN_PEARS,
            RED_APPLES,
        ],
    );

    foo(
        ctx,
        vec![
            MAROON_TOMATOES,
            PURPLE_POTATOES,
            ORGANE_ORANGES,
            GREEN_PEARS,
            RED_APPLES,
        ],
    );
}
```

`true`:

```
fn example() {
    foo(ctx, |param| {
        action();
        foo(param)
    });

    foo(ctx, Bar {
        x: value,
        y: value2,
    });

    foo(ctx, &[
        MAROON_TOMATOES,
        PURPLE_POTATOES,
        ORGANE_ORANGES,
        GREEN_PEARS,
        RED_APPLES,
    ]);

    foo(ctx, vec![
        MAROON_TOMATOES,
        PURPLE_POTATOES,
        ORGANE_ORANGES,
        GREEN_PEARS,
        RED_APPLES,
    ]);
}
```

`remove_nested_parens`

Remove nested parens.

- **Default value**: `true`,
- **Possible values**: `true`, `false`
- **Stable**: Yes

`true` (default):

```rust
fn main() {
    (foo());
}
```

`false`:

```rust
fn main() {
    (foo());

    ((((foo()))));
}
```

`reorder_impl_items`

Reorder impl items. `type` and `const` are put first, then macros and methods.

- **Default value**: `false`
- **Possible values**: `true`, `false`
- **Stable**: No (tracking issue: #3363)

`false` (default)

```rust
struct Dummy;

impl Iterator for Dummy {
    fn next(&mut self) -> Option<Self::Item> {
        None
    }

    type Item = i32;
}

impl Iterator for Dummy {
    type Item = i32;

    fn next(&mut self) -> Option<Self::Item> {
        None
    }
}
```

`true`

```rust
struct Dummy;

impl Iterator for Dummy {
    type Item = i32;

    fn next(&mut self) -> Option<Self::Item> {
        None
    }
}
```

`reorder_imports`

Reorder import and extern crate statements alphabetically in groups (a group is separated by a newline).

- **Default value**: `true`
- **Possible values**: `true`, `false`
- **Stable**: Yes

`true` **(default):**

```rust
use dolor;
use ipsum;
use lorem;
use sit;
```

`false`**:**

```rust
use lorem;
use ipsum;
use dolor;
use sit;
```

`group_imports`

Controls the strategy for how imports are grouped together.

- **Default value**: `Preserve`
- **Possible values**: `Preserve`, `StdExternalCrate`, `One`
- **Stable**: No (tracking issue: #5083)

`Preserve` **(default):**

Preserve the source file's import groups.

```
use super::update::convert_publish_payload;
use chrono::Utc;

use alloc::alloc::Layout;
use juniper::{FieldError, FieldResult};
use uuid::Uuid;

use std::sync::Arc;

use broker::database::PooledConnection;

use super::schema::{Context, Payload};
use crate::models::Event;
use core::f32;
```

`StdExternalCrate` :

Discard existing import groups, and create three groups for:

1. `std` , `core` and `alloc` ,
2. external crates,
3. `self` , `super` and `crate` imports.

```
use alloc::alloc::Layout;
use core::f32;
use std::sync::Arc;

use broker::database::PooledConnection;
use chrono::Utc;
use juniper::{FieldError, FieldResult};
use uuid::Uuid;

use super::schema::{Context, Payload};
use super::update::convert_publish_payload;
use crate::models::Event;
```

`One` :

Discard existing import groups, and create a single group for everything

```
use super::schema::{Context, Payload};
use super::update::convert_publish_payload;
use crate::models::Event;
use alloc::alloc::Layout;
use broker::database::PooledConnection;
use chrono::Utc;
use core::f32;
use juniper::{FieldError, FieldResult};
use std::sync::Arc;
use uuid::Uuid;
```

`reorder_modules`

Reorder `mod` declarations alphabetically in group.

- **Default value**: `true`
- **Possible values**: `true` , `false`
- **Stable**: Yes

`true` (default)

```
mod a;
mod b;

mod dolor;
mod ipsum;
mod lorem;
mod sit;
```

`false`

```
mod b;
mod a;

mod lorem;
mod ipsum;
mod dolor;
mod sit;
```

Note `mod` with `#[macro_export]` will not be reordered since that could change the semantics of the original source code.

`report_fixme`

Report `FIXME` items in comments.

- **Default value**: `"Never"`
- **Possible values**: `"Always"` , `"Unnumbered"` , `"Never"`
- **Stable**: No (tracking issue: #3394)

Warns about any comments containing `FIXME` in them when set to `"Always"` . If it contains a `#X` (with `X` being a number) in parentheses following the `FIXME` , `"Unnumbered"` will ignore it.

See also `report_todo` .

`report_todo`

Report `TODO` items in comments.

- **Default value**: `"Never"`
- **Possible values**: `"Always"`, `"Unnumbered"`, `"Never"`
- **Stable**: No (tracking issue: #3393)

Warns about any comments containing `TODO` in them when set to `"Always"`. If it contains a `#X` (with `X` being a number) in parentheses following the `TODO`, `"Unnumbered"` will ignore it.

See also `report_fixme`.

`required_version`

Require a specific version of rustfmt. If you want to make sure that the specific version of rustfmt is used in your CI, use this option.

- **Default value**: `CARGO_PKG_VERSION`
- **Possible values**: any published version (e.g. `"0.3.8"`)
- **Stable**: No (tracking issue: #3386)

`short_array_element_width_threshold`

The width threshold for an array element to be considered "short".

The layout of an array is dependent on the length of each of its elements. If the length of every element in an array is below this threshold (all elements are "short") then the array can be formatted in the mixed/compressed style, but if any one element has a length that exceeds this threshold then the array elements will have to be formatted vertically.

- **Default value**: `10`
- **Possible values**: any positive integer that is less than or equal to the value specified for `max_width`
- **Stable**: Yes

`10` (default):

```rust
fn main() {
    pub const FORMAT_TEST: [u64; 5] = [
        0x0000000000000000,
        0xaaaaaaaaaaaaaaaa,
        0xbbbbbbbbbbbbbbbb,
        0xcccccccccccccccc,
        0xdddddddddddddddd,
    ];
}
```

`20`:

```rust
fn main() {
    pub const FORMAT_TEST: [u64; 5] = [
        0x0000000000000000, 0xaaaaaaaaaaaaaaaa, 0xbbbbbbbbbbbbbbbb,
0xcccccccccccccccc,
        0xdddddddddddddddd,
    ];
}
```

See also `max_width`.

`skip_children`

Don't reformat out of line modules

- **Default value**: `false`
- **Possible values**: `true`, `false`
- **Stable**: No (tracking issue: #3389)

`single_line_if_else_max_width`

Maximum line length for single line if-else expressions. A value of `0` (zero) results in if-else expressions always being broken into multiple lines. Note this occurs when `use_small_heuristics` is set to `Off`.

- **Default value**: `50`
- **Possible values**: any positive integer that is less than or equal to the value specified for `max_width`
- **Stable**: Yes

By default this option is set as a percentage of `max_width` provided by `use_small_heuristics`, but a value set directly for `single_line_if_else_max_width` will take precedence.

See also `max_width` and `use_small_heuristics`

`space_after_colon`

Leave a space after the colon.

- **Default value**: `true`
- **Possible values**: `true`, `false`
- **Stable**: No (tracking issue: #3366)

`true` (default):

```
fn lorem<T: Eq>(t: T) {
    let lorem: Dolor = Lorem {
        ipsum: dolor,
        sit: amet,
    };
}
```

`false`:

```
fn lorem<T:Eq>(t:T) {
    let lorem:Dolor = Lorem {
        ipsum:dolor,
        sit:amet,
    };
}
```

See also: `space_before_colon` .

`space_before_colon`

Leave a space before the colon.

- **Default value**: `false`
- **Possible values**: `true` , `false`
- **Stable**: No (tracking issue: #3365)

`false` (default):

```
fn lorem<T: Eq>(t: T) {
    let lorem: Dolor = Lorem {
        ipsum: dolor,
        sit: amet,
    };
}
```

`true` :

```
fn lorem<T : Eq>(t : T) {
    let lorem : Dolor = Lorem {
        ipsum : dolor,
        sit : amet,
    };
}
```

See also: `space_after_colon` .

`spaces_around_ranges`

Put spaces around the .., ..=, and ... range operators

- **Default value**: `false`
- **Possible values**: `true` , `false`
- **Stable**: No (tracking issue: #3367)

`false` (default):

```
fn main() {
    let lorem = 0..10;
    let ipsum = 0..=10;

    match lorem {
        1..5 => foo(),
        _ => bar,
    }

    match lorem {
        1..=5 => foo(),
        _ => bar,
    }

    match lorem {
        1...5 => foo(),
        _ => bar,
    }
}
```

`true`:

```
fn main() {
    let lorem = 0 .. 10;
    let ipsum = 0 ..= 10;

    match lorem {
        1 .. 5 => foo(),
        _ => bar,
    }

    match lorem {
        1 ..= 5 => foo(),
        _ => bar,
    }

    match lorem {
        1 ... 5 => foo(),
        _ => bar,
    }
}
```

`struct_field_align_threshold`

The maximum diff of width between struct fields to be aligned with each other.

- **Default value** : 0
- **Possible values**: any non-negative integer
- **Stable**: No (tracking issue: #3371)

`0` (default):

```
struct Foo {
    x: u32,
    yy: u32,
    zzz: u32,
}
```

`20`:

```
struct Foo {
    x:   u32,
    yy:  u32,
    zzz: u32,
}
```

`struct_lit_single_line`

Put small struct literals on a single line

- **Default value**: `true`
- **Possible values**: `true`, `false`
- **Stable**: No (tracking issue: #3357)

`true` (default):

```
fn main() {
    let lorem = Lorem { foo: bar, baz: ofo };
}
```

`false`:

```
fn main() {
    let lorem = Lorem {
        foo: bar,
        baz: ofo,
    };
}
```

See also: `indent_style`.

`struct_lit_width`

Maximum width in the body of a struct literal before falling back to vertical formatting. A value of `0` (zero) results in struct literals always being broken into multiple lines. Note this occurs when `use_small_heuristics` is set to `Off`.

- **Default value**: `18`
- **Possible values**: any positive integer that is less than or equal to the value specified for `max_width`
- **Stable**: Yes

By default this option is set as a percentage of `max_width` provided by `use_small_heuristics`, but a value set directly for `struct_lit_width` will take precedence.

See also `max_width`, `use_small_heuristics`, and `struct_lit_single_line`

`struct_variant_width`

Maximum width in the body of a struct variant before falling back to vertical formatting. A value of `0` (zero) results in struct literals always being broken into multiple lines. Note this occurs when `use_small_heuristics` is set to `Off`.

- **Default value**: `35`
- **Possible values**: any positive integer that is less than or equal to the value specified for [`max_width`]
- **Stable**: Yes

By default this option is set as a percentage of [`max_width`] provided by [`use_small_heuristics`], but a value set directly for `struct_variant_width` will take precedence.

See also [`max_width`] and [`use_small_heuristics`]

`tab_spaces`

Number of spaces per tab

- **Default value**: `4`
- **Possible values**: any positive integer
- **Stable**: Yes

`4` (default):

```
fn lorem() {
    let ipsum = dolor();
    let sit = vec![
        "amet consectetur adipiscing elit amet",
        "consectetur adipiscing elit amet consectetur.",
    ];
}
```

`2`:

```
fn lorem() {
  let ipsum = dolor();
  let sit = vec![
    "amet consectetur adipiscing elit amet",
    "consectetur adipiscing elit amet consectetur.",
  ];
}
```

See also: [`hard_tabs`].

`trailing_comma`

How to handle trailing commas for lists

- **Default value**: `"Vertical"`

- **Possible values**: `"Always"`, `"Never"`, `"Vertical"`
- **Stable**: No (tracking issue: #3379)

`"Vertical"` (default):

```
fn main() {
    let Lorem { ipsum, dolor, sit } = amet;
    let Lorem {
        ipsum,
        dolor,
        sit,
        amet,
        consectetur,
        adipiscing,
    } = elit;
}
```

`"Always"`:

```
fn main() {
    let Lorem { ipsum, dolor, sit, } = amet;
    let Lorem {
        ipsum,
        dolor,
        sit,
        amet,
        consectetur,
        adipiscing,
    } = elit;
}
```

`"Never"`:

```
fn main() {
    let Lorem { ipsum, dolor, sit } = amet;
    let Lorem {
        ipsum,
        dolor,
        sit,
        amet,
        consectetur,
        adipiscing
    } = elit;
}
```

See also: `match_block_trailing_comma`.

`trailing_semicolon`

Add trailing semicolon after break, continue and return

- **Default value**: `true`
- **Possible values**: `true` , `false`
- **Stable**: No (tracking issue: #3378)

`true` (default):

```rust
fn foo() -> usize {
    return 0;
}
```

`false` :

```rust
fn foo() -> usize {
    return 0
}
```

`type_punctuation_density`

Determines if `+` or `=` are wrapped in spaces in the punctuation of types

- **Default value**: `"Wide"`
- **Possible values**: `"Compressed"` , `"Wide"`
- **Stable**: No (tracking issue: #3364)

`"Wide"` (default):

```rust
fn lorem<Ipsum: Dolor + Sit = Amet>() {
    // body
}
```

`"Compressed"` :

```rust
fn lorem<Ipsum: Dolor+Sit=Amet>() {
    // body
}
```

`unstable_features`

Enable unstable features on the unstable channel.

- **Default value**: `false`
- **Possible values**: `true` , `false`
- **Stable**: No (tracking issue: #3387)

`use_field_init_shorthand`

Use field initialize shorthand if possible.

- **Default value**: `false`
- **Possible values**: `true` , `false`
- **Stable**: Yes

`false` (default):

```rust
struct Foo {
    x: u32,
    y: u32,
    z: u32,
}

fn main() {
    let x = 1;
    let y = 2;
    let z = 3;
    let a = Foo { x, y, z };
    let b = Foo { x: x, y: y, z: z };
}
```

`true` :

```rust
struct Foo {
    x: u32,
    y: u32,
    z: u32,
}

fn main() {
    let x = 1;
    let y = 2;
    let z = 3;
    let a = Foo { x, y, z };
}
```

`use_small_heuristics`

This option can be used to simplify the management and bulk updates of the granular width configuration settings (`fn_call_width` , `attr_fn_like_width` , `struct_lit_width` , `struct_variant_width` , `array_width` , `chain_width` , `single_line_if_else_max_width`), that respectively control when formatted constructs are multi-lined/vertical based on width.

Note that explicitly provided values for the width configuration settings take precedence and override the calculated values determined by `use_small_heuristics` .

- **Default value**: `"Default"`
- **Possible values**: `"Default"` , `"Off"` , `"Max"`
- **Stable**: Yes

`Default` **(default):**

When `use_small_heuristics` is set to `Default`, the values for the granular width settings are calculated as a ratio of the value for `max_width`.

The ratios are:

- `fn_call_width` - `60%`
- `attr_fn_like_width` - `70%`
- `struct_lit_width` - `18%`
- `struct_variant_width` - `35%`
- `array_width` - `60%`
- `chain_width` - `60%`
- `single_line_if_else_max_width` - `50%`

For example when `max_width` is set to `100`, the width settings are:

- `fn_call_width=60`
- `attr_fn_like_width=70`
- `struct_lit_width=18`
- `struct_variant_width=35`
- `array_width=60`
- `chain_width=60`
- `single_line_if_else_max_width=50`

and when `max_width` is set to `200`:

- `fn_call_width=120`
- `attr_fn_like_width=140`
- `struct_lit_width=36`
- `struct_variant_width=70`
- `array_width=120`
- `chain_width=120`
- `single_line_if_else_max_width=100`

```
enum Lorem {
    Ipsum,
    Dolor(bool),
    Sit { amet: Consectetur, adipiscing: Elit },
}

fn main() {
    lorem(
        "lorem",
        "ipsum",
        "dolor",
        "sit",
        "amet",
        "consectetur",
        "adipiscing",
    );
    let lorem = Lorem {
        ipsum: dolor,
        sit: amet,
    };
    let lorem = Lorem { ipsum: dolor };

    let lorem = if ipsum { dolor } else { sit };
}
```

`Off`:

When `use_small_heuristics` is set to `Off`, the granular width settings are functionally disabled and ignored. See the documentation for the respective width config options for specifics.

```rust
enum Lorem {
    Ipsum,
    Dolor(bool),
    Sit {
        amet: Consectetur,
        adipiscing: Elit,
    },
}

fn main() {
    lorem("lorem", "ipsum", "dolor", "sit", "amet", "consectetur", "adipiscing");

    let lorem = Lorem {
        ipsum: dolor,
        sit: amet,
    };

    let lorem = if ipsum {
        dolor
    } else {
        sit
    };
}
```

`Max`:

When `use_small_heuristics` is set to `Max`, then each granular width setting is set to the same value as `max_width`.

So if `max_width` is set to `200`, then all the width settings are also set to `200`.

- `fn_call_width=200`
- `attr_fn_like_width=200`
- `struct_lit_width=200`
- `struct_variant_width=200`
- `array_width=200`

- `chain_width=200`
- `single_line_if_else_max_width=200`

```rust
enum Lorem {
    Ipsum,
    Dolor(bool),
    Sit { amet: Consectetur, adipiscing: Elit },
}

fn main() {
    lorem("lorem", "ipsum", "dolor", "sit", "amet", "consectetur", "adipiscing");

    let lorem = Lorem { ipsum: dolor, sit: amet };

    let lorem = if ipsum { dolor } else { sit };
}
```

See also:

- `max_width`
- `fn_call_width`
- `attr_fn_like_width`
- `struct_lit_width`
- `struct_variant_width`
- `array_width`
- `chain_width`
- `single_line_if_else_max_width`

`use_try_shorthand`

Replace uses of the try! macro by the ? shorthand

- **Default value**: `false`
- **Possible values**: `true` , `false`
- **Stable**: Yes

`false` (default):

```rust
fn main() {
    let lorem = ipsum.map(|dolor| dolor.sit())?;

    let lorem = try!(ipsum.map(|dolor| dolor.sit()));
}
```

`true` :

```rust
fn main() {
    let lorem = ipsum.map(|dolor| dolor.sit())?;
}
```

`version`

Which version of the formatting rules to use. `Version::One` is backwards-compatible with Rustfmt 1.0. Other versions are only backwards compatible within a major version number.

- **Default value**: `One`
- **Possible values**: `One`, `Two`
- **Stable**: No (tracking issue: #3383)

Example

```
version = "Two"
```

`where_single_line`

Forces the `where` clause to be laid out on a single line.

- **Default value**: `false`
- **Possible values**: `true`, `false`
- **Stable**: No (tracking issue: #3359)

`false` (default):

```
impl<T> Lorem for T
where
    Option<T>: Ipsum,
{
    // body
}
```

`true`:

```
impl<T> Lorem for T
where Option<T>: Ipsum
{
    // body
}
```

See also `brace_style`, `control_brace_style`.

`wrap_comments`

Break comments to fit on the line

- **Default value**: `false`
- **Possible values**: `true`, `false`
- **Stable**: No (tracking issue: #3347)

`false` (default):

```
// Lorem ipsum dolor sit amet, consectetur adipiscing elit,
// sed do eiusmod tempor incididunt ut labore et dolore
// magna aliqua. Ut enim ad minim veniam, quis nostrud
// exercitation ullamco laboris nisi ut aliquip ex ea
// commodo consequat.

// Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
```

`true`:

```
// Lorem ipsum dolor sit amet, consectetur adipiscing elit,
// sed do eiusmod tempor incididunt ut labore et dolore
// magna aliqua. Ut enim ad minim veniam, quis nostrud
// exercitation ullamco laboris nisi ut aliquip ex ea
// commodo consequat.
```

Internal Options

`emit_mode`

Internal option

`make_backup`

Internal option, use `--backup`

`print_misformatted_file_names`

Internal option, use `-l` or `--files-with-diff`