B

Operators and Symbols

This appendix contains a glossary of Rust’s syntax, including operators and other symbols that appear by themselves or in the context of paths, generics, trait bounds, macros, attributes, comments, tuples, and brackets.

Operators

Table B-1 contains the operators in Rust, an example of how the operator would appear in context, a short explanation, and whether that operator is overloadable. If an operator is overloadable, the relevant trait to use to overload that operator is listed.

Prod: The tables in this appendix should be labeled Table B-1, Table B-2, etc.

Operators

| Operator | Example | Explanation | Overloadable? |
| --- | --- | --- | --- |
| ! | ident!(...), ident!{...}, ident![...] | Macro expansion |  |
| ! | !expr | Bitwise or logical complement | Not |
| != | expr != expr | Nonequality comparison | PartialEq |
| % | expr % expr | Arithmetic remainder | Rem |
| %= | var %= expr | Arithmetic remainder and assignment | RemAssign |
| & | &expr, &mut expr | Borrow |  |
| & | &type, &mut type, &'a type, &'a mut type | Borrowed pointer type |  |
| & | expr & expr | Bitwise AND | BitAnd |
| &= | var &= expr | Bitwise AND and assignment | BitAndAssign |
| && | expr && expr | Short-circuiting logical AND |  |
| \* | expr \* expr | Arithmetic multiplication | Mul |
| \*= | var \*= expr | Arithmetic multiplication and assignment | MulAssign |
| \* | \*expr | Dereference | Deref |
| \* | \*const type, \*mut type | Raw pointer |  |
| + | trait + trait, 'a + trait | Compound type constraint |  |
| + | expr + expr | Arithmetic addition | Add |
| += | var += expr | Arithmetic addition and assignment | AddAssign |
| , | expr, expr | Argument and element separator |  |
| - | - expr | Arithmetic negation | Neg |
| - | expr - expr | Arithmetic subtraction | Sub |
| -= | var -= expr | Arithmetic subtraction and assignment | SubAssign |
| -> | fn(...) -> type, |…| -> type | Function and closure return type |  |
| . | expr.ident | Member access |  |
| .. | .., expr.., ..expr, expr..expr | Right-exclusive range literal | PartialOrd |
| ..= | ..=expr, expr..=expr | Right-inclusive range literal | PartialOrd |
| .. | ..expr | Struct literal update syntax |  |
| .. | variant(x, ..), struct\_type { x, .. } | “And the rest” pattern binding |  |
| ... | expr...expr | (Deprecated, use ..= instead) In a pattern: inclusive range pattern |  |
| / | expr / expr | Arithmetic division | Div |
| /= | var /= expr | Arithmetic division and assignment | DivAssign |
| : | pat: type, ident: type | Constraints |  |
| : | ident: expr | Struct field initializer |  |
| : | 'a: loop {...} | Loop label |  |
| ; | expr; | Statement and item terminator |  |
| ; | [...; len] | Part of fixed-size array syntax |  |
| << | expr << expr | Left-shift | Shl |
| <<= | var <<= expr | Left-shift and assignment | ShlAssign |
| < | expr < expr | Less than comparison | PartialOrd |
| <= | expr <= expr | Less than or equal to comparison | PartialOrd |
| = | var = expr, ident = type | Assignment/equivalence |  |
| == | expr == expr | Equality comparison | PartialEq |
| => | pat => expr | Part of match arm syntax |  |
| > | expr > expr | Greater than comparison | PartialOrd |
| >= | expr >= expr | Greater than or equal to comparison | PartialOrd |
| >> | expr >> expr | Right-shift | Shr |
| >>= | var >>= expr | Right-shift and assignment | ShrAssign |
| @ | ident @ pat | Pattern binding |  |
| ^ | expr ^ expr | Bitwise exclusive OR | BitXor |
| ^= | var ^= expr | Bitwise exclusive OR and assignment | BitXorAssign |
| | | pat | pat | Pattern alternatives |  |
| | | expr | expr | Bitwise OR | BitOr |
| |= | var |= expr | Bitwise OR and assignment | BitOrAssign |
| || | expr || expr | Short-circuiting logical OR |  |
| ? | expr? | Error propagation |  |

Non-operator Symbols

The following tables contain all symbols that don’t function as operators; that is, they don’t behave like a function or method call.

Table B-2 shows symbols that appear on their own and are valid in a variety of locations.

Stand-Alone Syntax

| Symbol | Explanation |
| --- | --- |
| 'ident | Named lifetime or loop label |
| ...u8, ...i32, ...f64, ...usize, and so on | Numeric literal of specific type |
| "..." | String literal |
| r"...", r#"..."#, r##"..."##, and so on | Raw string literal; escape characters not processed |
| b"..." | Byte string literal; constructs an array of bytes instead of a string |
| br"...", br#"..."#, br##"..."##, and so on | Raw byte string literal; combination of raw and byte string literal |
| '...' | Character literal |
| b'...' | ASCII byte literal |
| |…| expr | Closure |
| ! | Always empty bottom type for diverging functions |
| \_ | “Ignored” pattern binding; also used to make integer literals readable |

Table B-3 shows symbols that appear in the context of a path through the module hierarchy to an item.

Path-Related Syntax

| Symbol | Explanation |
| --- | --- |
| ident::ident | Namespace path |
| ::path | Path relative to the crate root (that is, an explicitly absolute path) |
| self::path | Path relative to the current module (that is, an explicitly relative path) |
| super::path | Path relative to the parent of the current module |
| type::ident, <type as trait>::ident | Associated constants, functions, and types |
| <type>::... | Associated item for a type that cannot be directly named (for example, <&T>::..., <[T]>::..., and so on) |
| trait::method(...) | Disambiguating a method call by naming the trait that defines it |
| type::method(...) | Disambiguating a method call by naming the type for which it’s defined |
| <type as trait>::method(...) | Disambiguating a method call by naming the trait and type |

Table B-4 shows symbols that appear in the context of using generic type parameters.

Generics

| Symbol | Explanation |
| --- | --- |
| path<...> | Specifies parameters to a generic type in a type (for example, Vec<u8>) |
| path::<...>, method::<...> | Specifies parameters to a generic type, function, or method in an expression; often referred to as turbofish (for example, "42".parse::<i32>()) |
| fn ident<...> ... | Define generic function |
| struct ident<...> ... | Define generic structure |
| enum ident<...> ... | Define generic enumeration |
| impl<...> ... | Define generic implementation |
| for<...> type | Higher-ranked lifetime bounds |
| type<ident=type> | A generic type where one or more associated types have specific assignments (for example, Iterator<Item=T>) |

Table B-5 shows symbols that appear in the context of constraining generic type parameters with trait bounds.

Trait Bound Constraints

| Symbol | Explanation |
| --- | --- |
| T: U | Generic parameter T constrained to types that implement U |
| T: 'a | Generic type T must outlive lifetime 'a (meaning the type cannot transitively contain any references with lifetimes shorter than 'a) |
| T: 'static | Generic type T contains no borrowed references other than 'static ones |
| 'b: 'a | Generic lifetime 'b must outlive lifetime 'a |
| T: ?Sized | Allow generic type parameter to be a dynamically sized type |
| 'a + trait, trait + trait | Compound type constraint |

Table B-6 shows symbols that appear in the context of calling or defining macros and specifying attributes on an item.

Macros and Attributes

| Symbol | Explanation |
| --- | --- |
| #[meta] | Outer attribute |
| #![meta] | Inner attribute |
| $ident | Macro substitution |
| $ident:kind | Macro capture |
| $(…)… | Macro repetition |
| ident!(...), ident!{...}, ident![...] | Macro invocation |

Table B-7 shows symbols that create comments.

Comments

| Symbol | Explanation |
| --- | --- |
| // | Line comment |
| //! | Inner line doc comment |
| /// | Outer line doc comment |
| /\*...\*/ | Block comment |
| /\*!...\*/ | Inner block doc comment |
| /\*\*...\*/ | Outer block doc comment |

Table B-8 shows symbols that appear in the context of using tuples.

Tuples

| Symbol | Explanation |
| --- | --- |
| () | Empty tuple (aka unit), both literal and type |
| (expr) | Parenthesized expression |
| (expr,) | Single-element tuple expression |
| (type,) | Single-element tuple type |
| (expr, ...) | Tuple expression |
| (type, ...) | Tuple type |
| expr(expr, ...) | Function call expression; also used to initialize tuple structs and tuple enum variants |
| expr.0, expr.1, and so on | Tuple indexing |

Table B-9 shows the contexts in which curly brackets are used.

Curly Brackets

| Context | Explanation |
| --- | --- |
| {...} | Block expression |
| Type {...} | struct literal |

Table B-10 shows the contexts in which square brackets are used.

Square Brackets

| Context | Explanation |
| --- | --- |
| [...] | Array literal |
| [expr; len] | Array literal containing len copies of expr |
| [type; len] | Array type containing len instances of type |
| expr[expr] | Collection indexing; overloadable (Index, IndexMut) |
| expr[..], expr[a..], expr[..b], expr[a..b] | Collection indexing pretending to be collection slicing, using Range, RangeFrom, RangeTo, or RangeFull as the “index” |