Practical Programming

Rust : Common Programming Concepts

David Bouchet

david.bouchet.epita@gmail.com

Scalar Types – Integers

Signed Integers:

```
i8        // 8 bits
i16       // 16 bits
i32       // 32 bits
i64       // 64 bits
i128      // 128 bits
isize     // Architecture-dependent size
```

Unsigned Integers:

```
u8        // 8 bits
u16       // 16 bits
u32       // 32 bits
u64       // 64 bits
u128      // 128 bits
usize     // Architecture-dependent size
```

Source: https://doc.rust-lang.org/reference/types/numeric.html#integer-types

Other Scalar Types

The Floating-Point Types:

```
f32    // IEEE754 - Single precision
f64    // IEEE754 - Double precision
```

The Boolean Type:

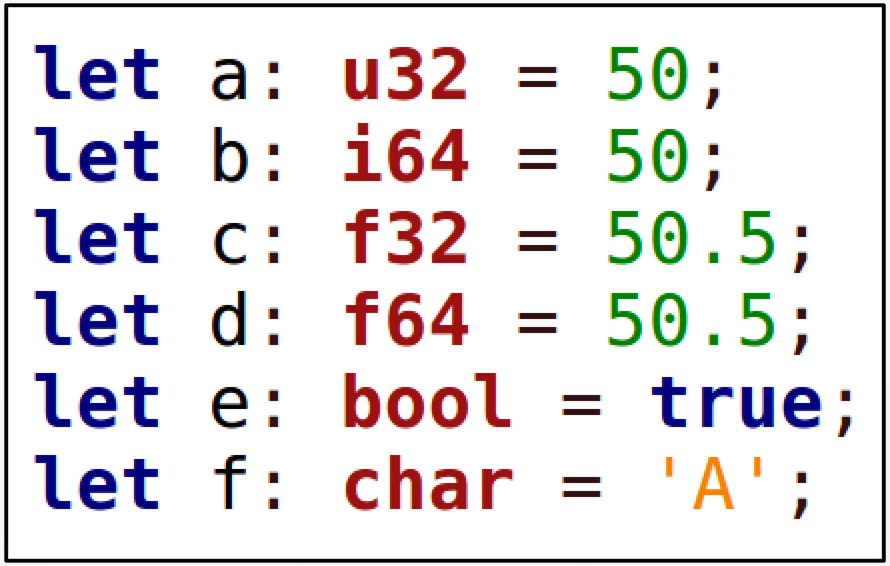
```
bool    // Two values only: true or false
```

The Character Type:

```
char    // Represents a single Unicode character
```

Declaring and Initializing Variables

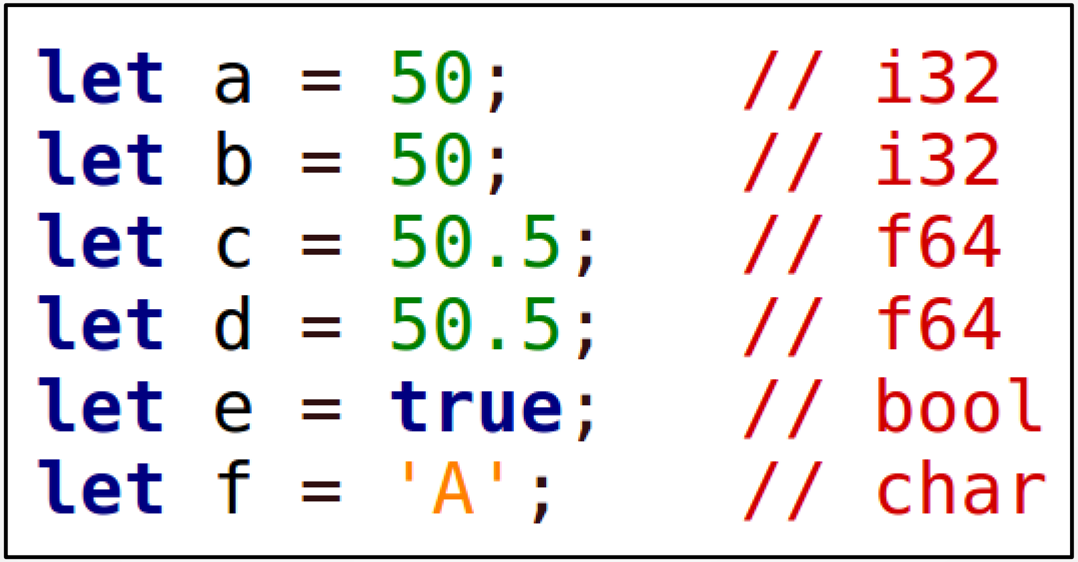
Use the let keyword:



```
let a: u32 = 50;
let b: i64 = 50;
let c: f32 = 50.5;
let d: f64 = 50.5;
let e: bool = true;
let f: char = 'A';
```

Declaring and Initializing Variables

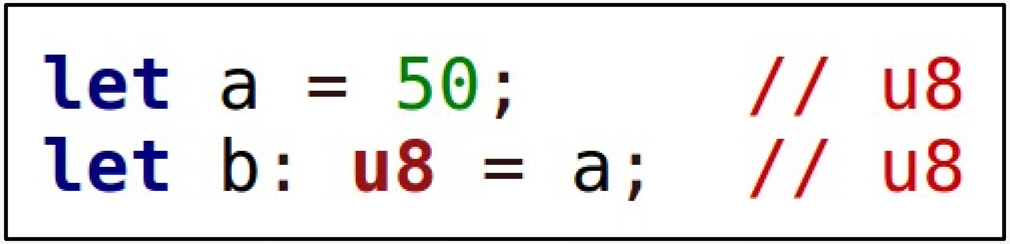
Default Type Inference



```
let a = 50;      // i32
let b = 50;      // i32
let c = 50.5;    // f64
let d = 50.5;    // f64
let e = true;    // bool
let f = 'A';     // char
```

Declaring and Initializing Variables

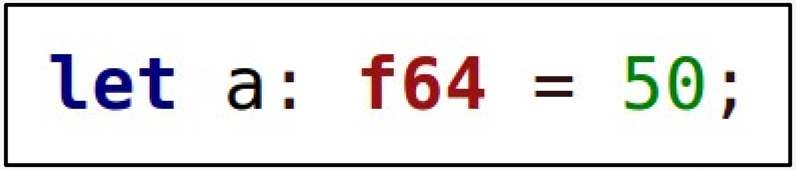
Type Inference



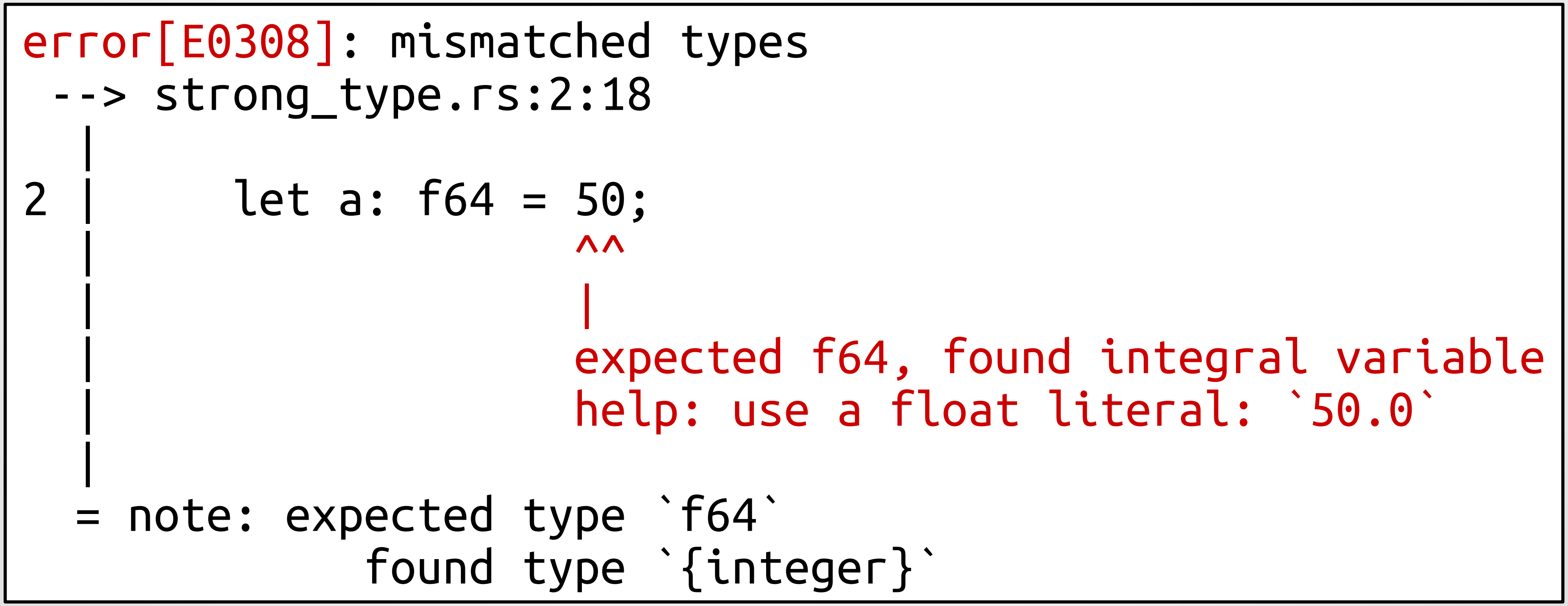
```
let a = 50;      // u8
let b: u8 = a;  // u8
```

The type of *a* is deduced from that of *b*, which is explicitly annotated.

Strong Type System



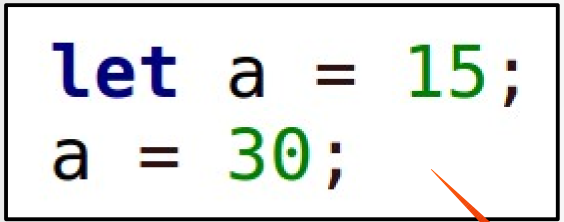
```
let a: f64 = 50;
```



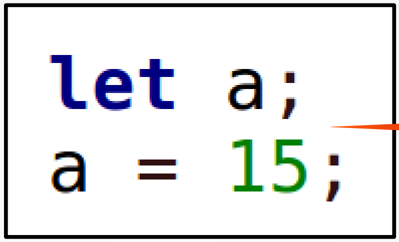
```
error[E0308]: mismatched types
 --> strong_type.rs:2:18
  |
2 |     let a: f64 = 50;
  |                  ^^
  |                  |
  |                  expected f64, found integral variable
  |                  help: use a float literal: `50.0`
  |
  = note: expected type `f64`
             found type `{integer}`
```

Immutability

A variable is immutable by default.



```
let a = 15;
a = 30;
```



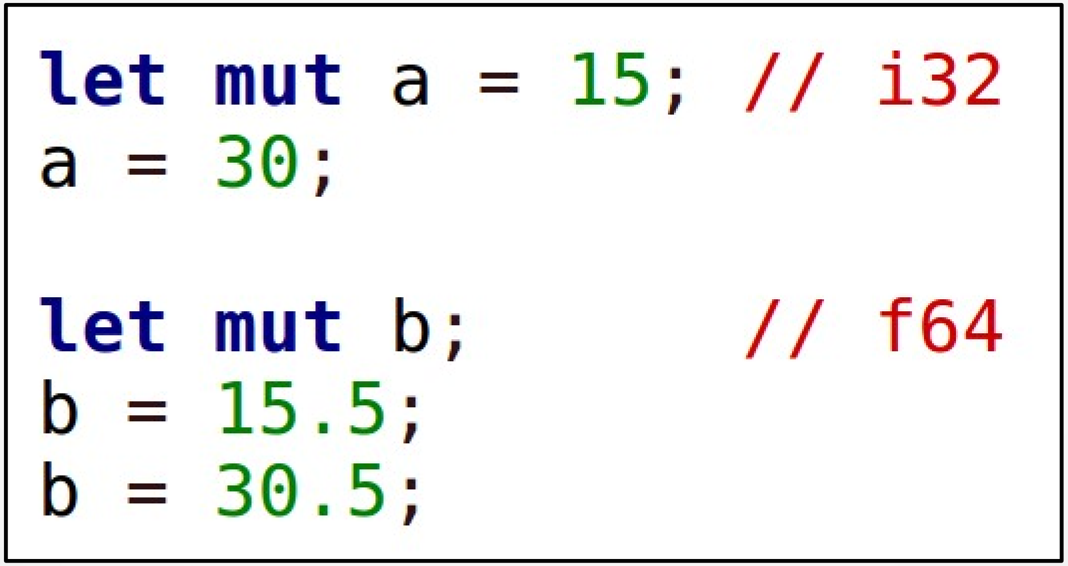
```
let a;
a = 15;
```

OK (initialization)

```
error[E0384]: cannot assign twice to immutable variable `a`
 --> immutability.rs:4:5
  |
3 |     let a = 15;
  |         - first assignment to `a`
4 |     a = 30              // Error
  |     ^^^^^^ cannot assign twice to immutable variable
```

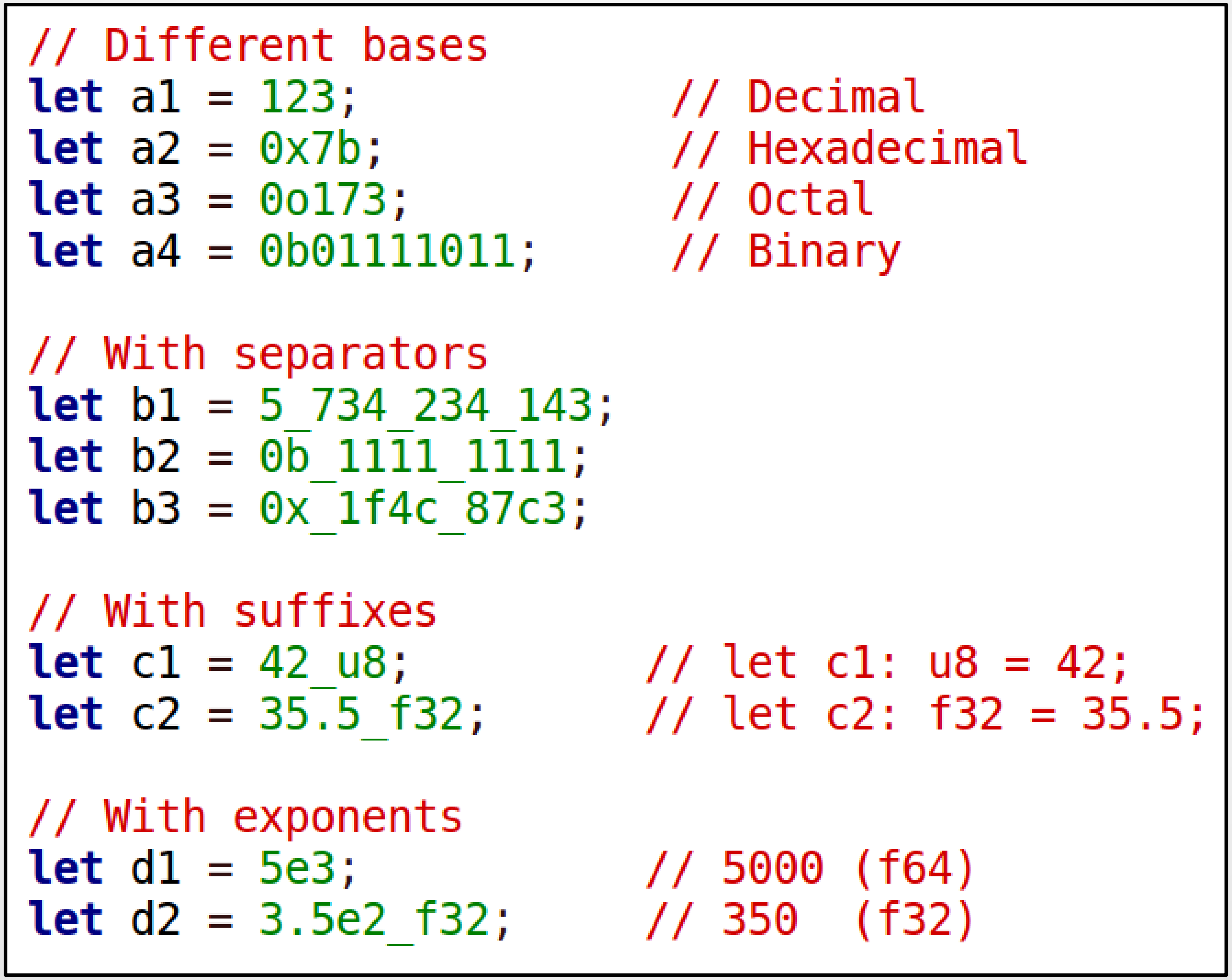
Mutability

Use the mut keyword:



```
let mut a = 15; // i32
a = 30;

let mut b;      // f64
b = 15.5;
b = 30.5;
```

Number Literals



```
// Different bases
let a1 = 123;            // Decimal
let a2 = 0x7b;           // Hexadecimal
let a3 = 0o173;          // Octal
let a4 = 0b01111011;     // Binary

// With separators
let b1 = 5_734_234_143;
let b2 = 0b_1111_1111;
let b3 = 0x_1f4c_87c3;

// With suffixes
let c1 = 42_u8;         // let c1: u8 = 42;
let c2 = 35.5_f32;      // let c2: f32 = 35.5;

// With exponents
let d1 = 5e3;           // 5000 (f64)
let d2 = 3.5e2_f32;     // 350  (f32)
```

Operators

All operators are given on this page:

https://doc.rust-lang.org/book/appendix-02-operators.html#operators

Note: the **++** and **--** operators are not available.

```
let r = c++;
```

Macros

Rust uses functions and macros.

Macros are really powerful, much more powerful than those of the C language, but also much more complicated to define.
We will use them only.

Macros can be called in the same way as functions.

Macros end with the “!” symbol.

Printing on the Terminal

Use println!() to print on the standard output.

```
println!("Hello World");

let a = 2;
let b = 3;
println!("{} + {} = {}", a, b, a + b);
println!("{2} + {1} = {0}", a + b, b, a);

let c = 'A';
let d = 8.0;
println!("c = {} and d = {}", c, d);
```

```
Hello World
2 + 3 = 5
2 + 3 = 5
c = A and d = 8
```

See also https://doc.rust-lang.org/std/fmt/index.html

Printing on the Terminal

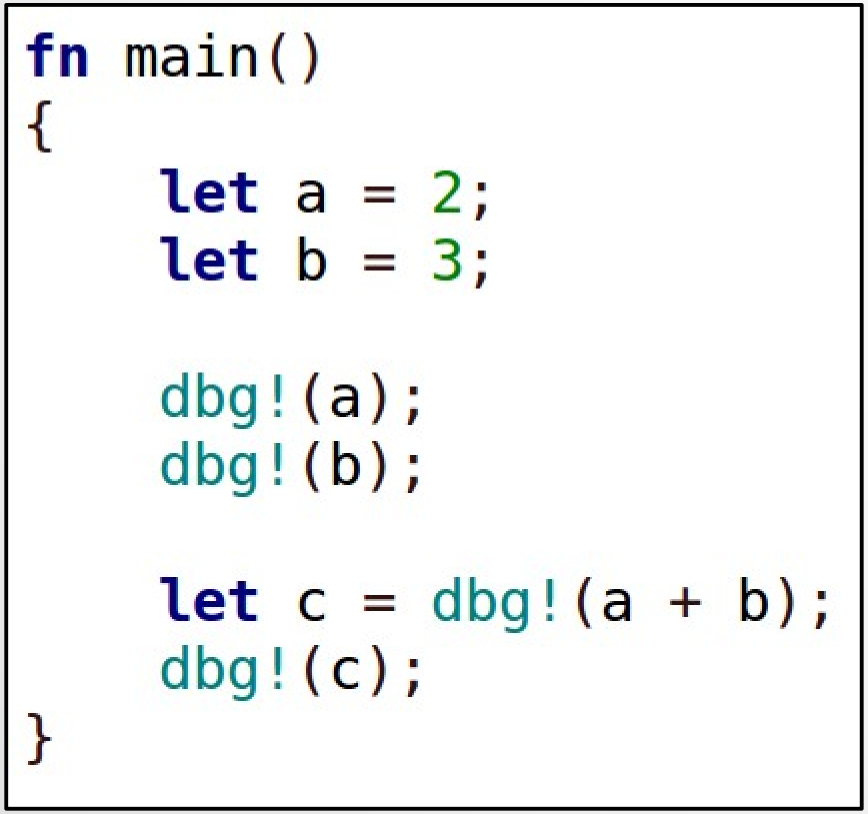
print!() is equivalent to println()! except that a newline is not printed at the end of the message.

eprinln!() and eprint!() are equivalent to println!() and print!() respectively except that the message is printed on the standard error.

Printing for Debugging

A macro for quick and dirty debugging: dbg!()

dbg.rs

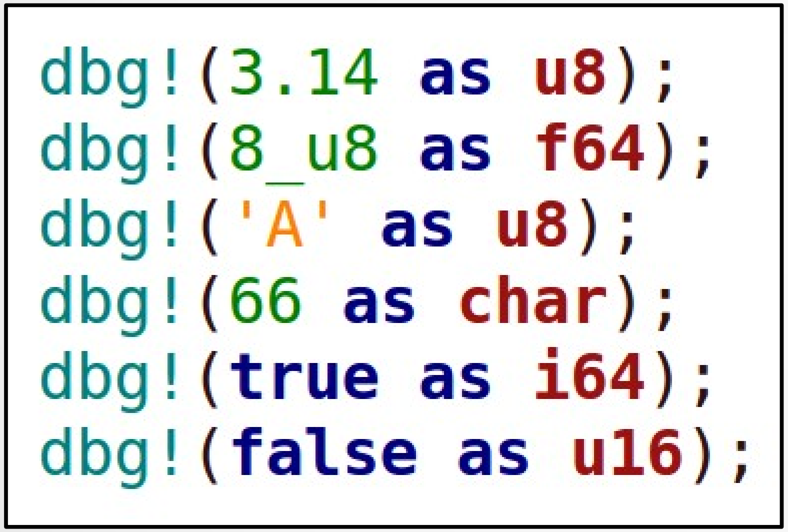


```
fn main()
{
    let a = 2;
    let b = 3;

    dbg!(a);
    dbg!(b);

    let c = dbg!(a + b);
    dbg!(c);
}
```

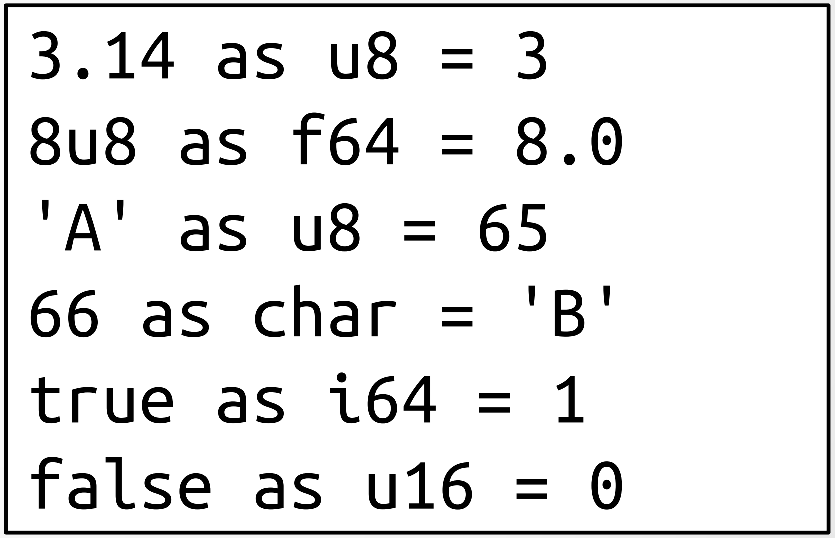
```
[dbg.rs:6] a = 2
[dbg.rs:7] b = 3
[dbg.rs:9] a + b = 5
[dbg.rs:10] c = 5
```

Type Conversions

Use the as keyword to convert one type into another.



```rust
dbg!(3.14 as u8);
dbg!(8_u8 as f64);
dbg!('A' as u8);
dbg!(66 as char);
dbg!(true as i64);
dbg!(false as u16);
```

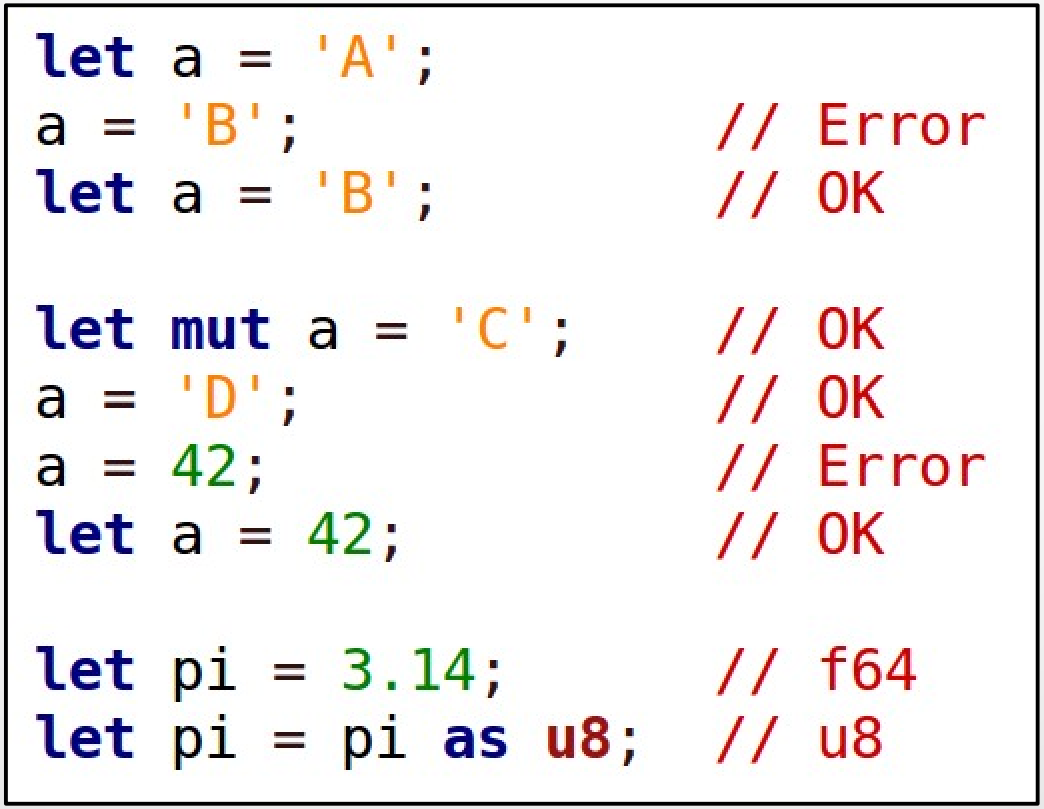


```
3.14 as u8 = 3
8u8 as f64 = 8.0
'A' as u8 = 65
66 as char = 'B'
true as i64 = 1
false as u16 = 0
```

Shadowing

A variable can be shadowed in its scope:

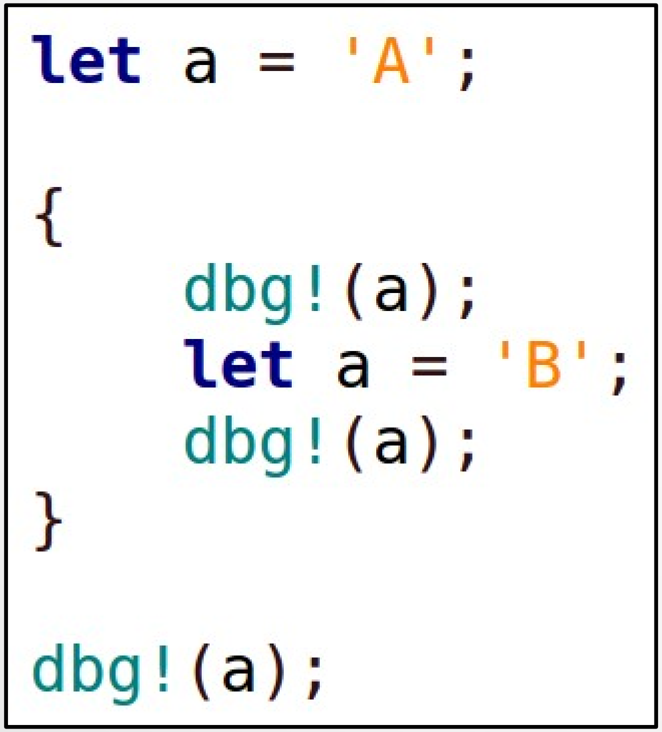
- A new variable with the same name is created.
- The previous variable can no longer be accessed.



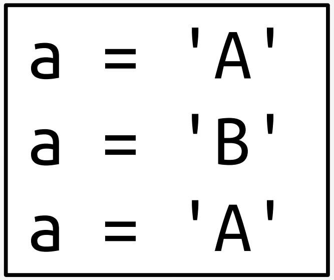
```
let a = 'A';
a = 'B';            // Error
let a = 'B';        // OK

let mut a = 'C';    // OK
a = 'D';            // OK
a = 42;             // Error
let a = 42;         // OK

let pi = 3.14;      // f64
let pi = pi as u8;  // u8
```

Shadowing (Inner Block)

A variable can be shadowed in an inner block:

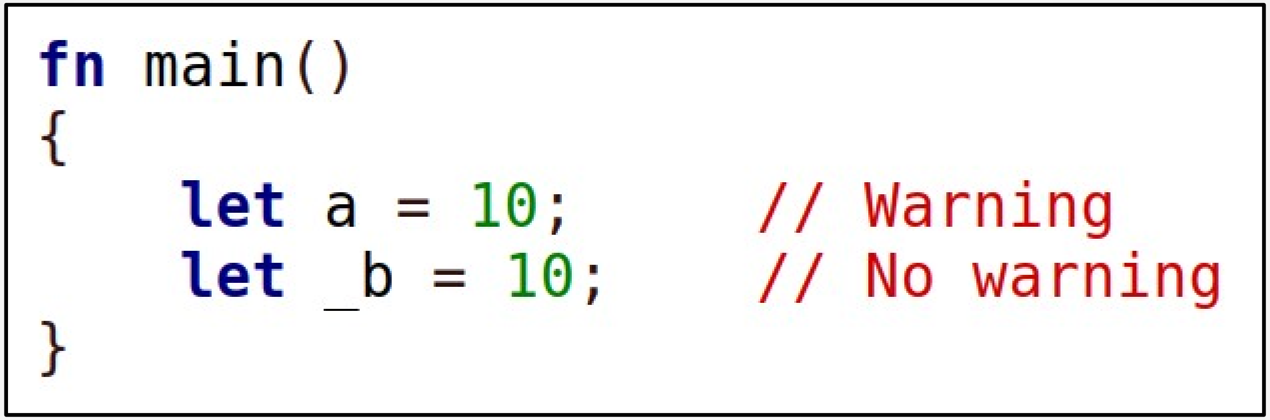


```
let a = 'A';

{
    dbg!(a);
    let a = 'B';
    dbg!(a);
}

dbg!(a);
```

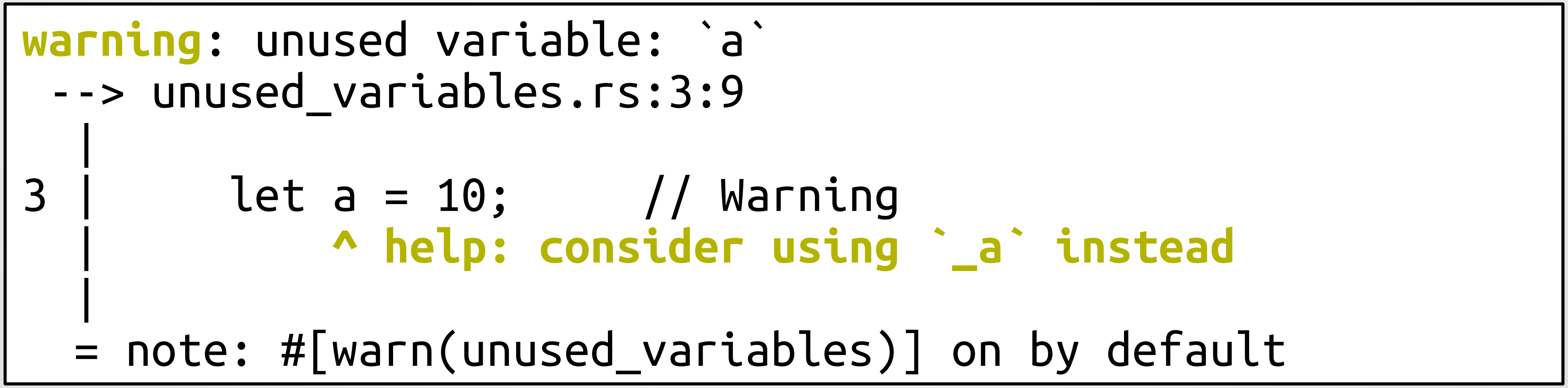


```
a = 'A'
a = 'B'
a = 'A'
```

Unused Variables



```rust
fn main()
{
    let a = 10;     // Warning
    let _b = 10;    // No warning
}
```



```
warning: unused variable: `a`
 --> unused_variables.rs:3:9
  |
3 |     let a = 10;     // Warning
  |         ^ help: consider using `_a` instead
  |
  = note: #[warn(unused_variables)] on by default
```

Constants

Use *const* instead of *let*.

Differences between constants and immutable variables:

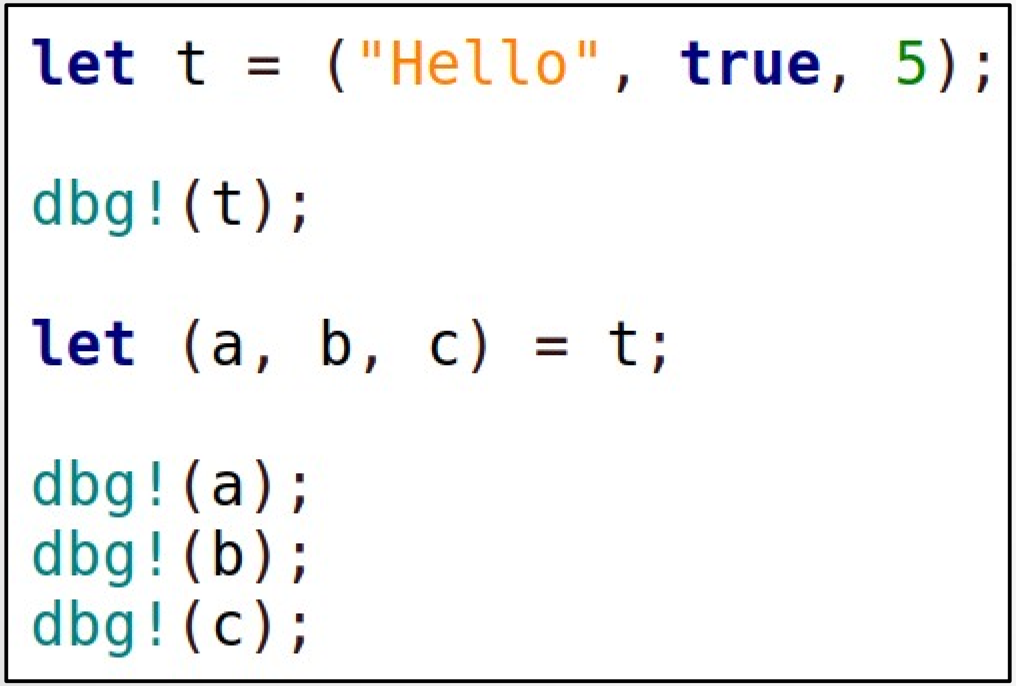
- **No type inference** for constants.
- Constants can be declared in the **global scope**.
- Constants must be initialized to a **constant expression**.
- Constants should have **upper case names**.

Constants

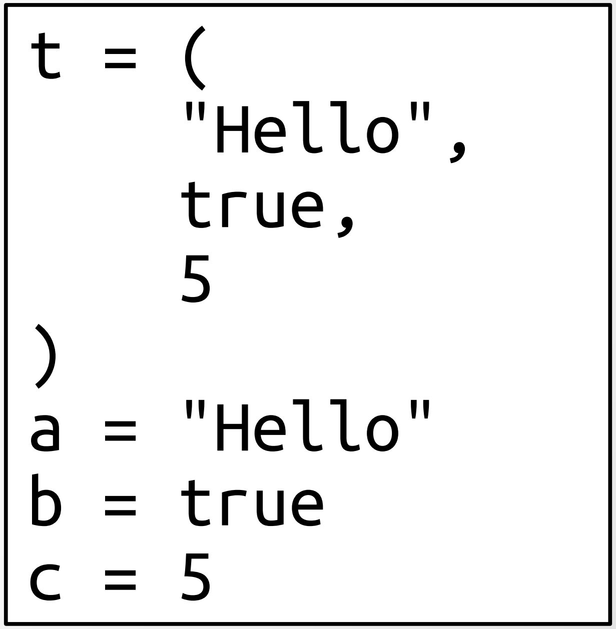
```
const OK_1: u8 = 24;
let error_1: u8 = 24;          // Must be constant

fn main() {
    let var: u8 = 1;

    const OK_3: bool = true;
    const OK_4: bool = OK_3;

    const ERR_2 = true         // The type is missing
    const ERR_3: u8 = var + 1; // Not a constant expression
    const warning: char = 'A'; // Should have an upper case
                               // name
}
```

Tuples

Tuples are fixed-length collections of values of different types.



```rust
let t = ("Hello", true, 5);

dbg!(t);

let (a, b, c) = t;

dbg!(a);
dbg!(b);
dbg!(c);
```



```
t = (
    "Hello",
    true,
    5
)
a = "Hello"
b = true
c = 5
```

Statements and Expressions

Statements:

- Do not return values.
- Cannot be assigned to variables.

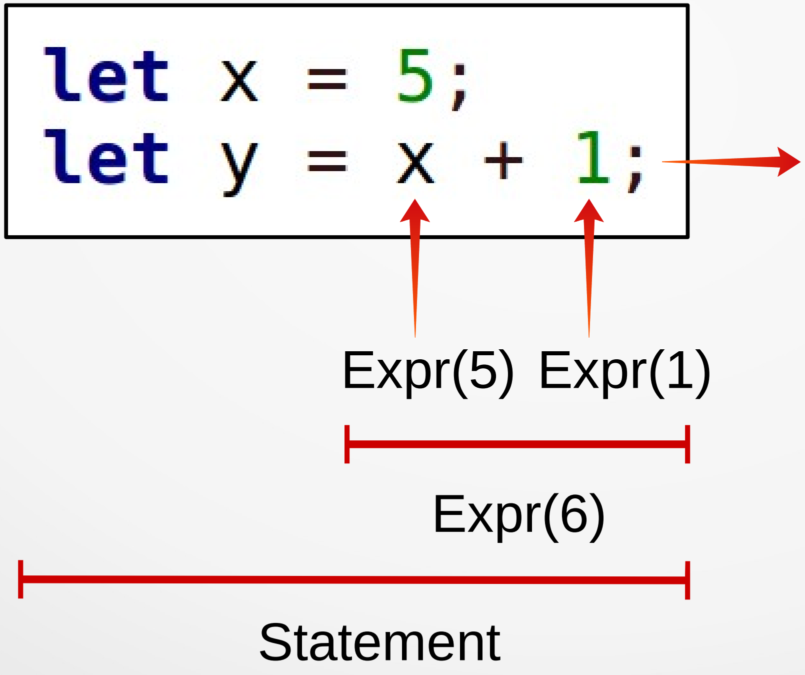
Expressions:

- Return values.
- Can be assigned to variables.

Rust is an expression-based language.
An expression evaluates something and returns the result.

Statements and Expressions

The let keyword is a statement.
A statement can contain expressions.



```
let x = 5;
let y = x + 1;
```

A statement ends with a semicolon.

Expr(5) Expr(1)

Expr(6)

Statement

Statements and Expressions

If you place a semicolon at the end of an expression, this expression becomes a statement.

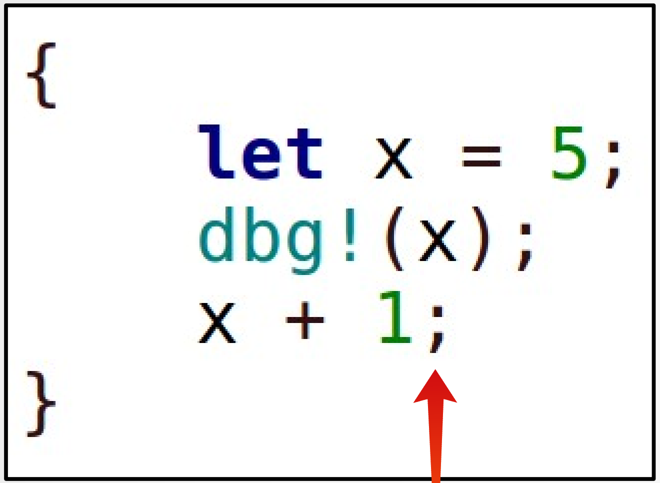
A block returns the value of its last instruction:

- If the last instruction is a statement, the block returns an ***empty tuple***, which means no value.
- Otherwise, it returns the value of the expression.

The symbol of an ***empty tuple*** is **()**.

Statements and Expressions

Block ending with a statement



```
{
    let x = 5;
    dbg!(x);
    x + 1;
}
```

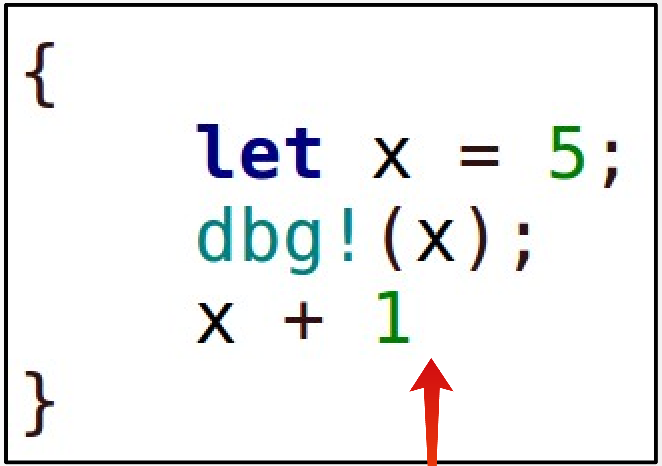
A Semicolon

"`x + 1;`" is a statement.

The block returns `()`.

The expression is lost.

Block ending with an expression



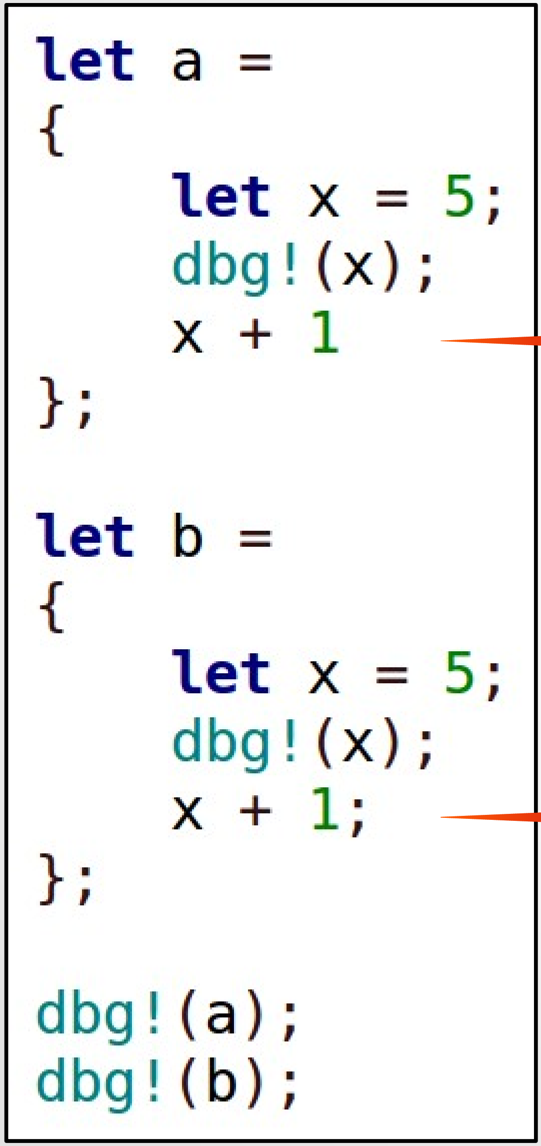
```
{
    let x = 5;
    dbg!(x);
    x + 1
}
```

No Semicolon

"`x + 1`" is an expression.

The block returns 6.

Statements and Expressions



```
let a =
{
    let x = 5;
    dbg!(x);
    x + 1
};

let b =
{
    let x = 5;
    dbg!(x);
    x + 1;
};

dbg!(a);
dbg!(b);
```

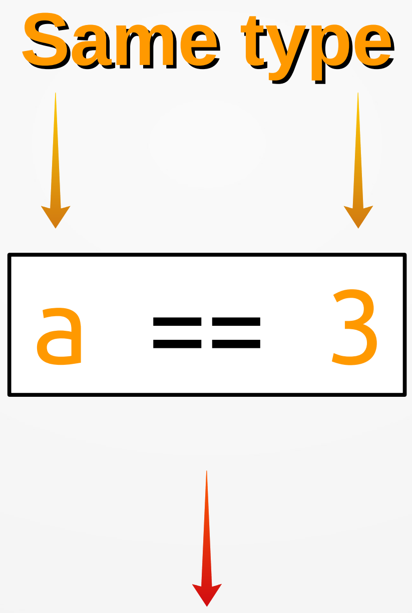
“x + 1” is evaluated and 6 is returned.

“x + 1” is lost and () is returned.

```
x = 5
x = 5
a = 6
b = ()
```

Conditions

A ***condition*** is always a boolean type.



Same type

```
a == 3
```

Returns either ***true*** or ***false***.

The *if* Expression

```
if condition1
{
    // ...
}

else if condition2
{
    // ...
}

else
{
    // ...
}
```

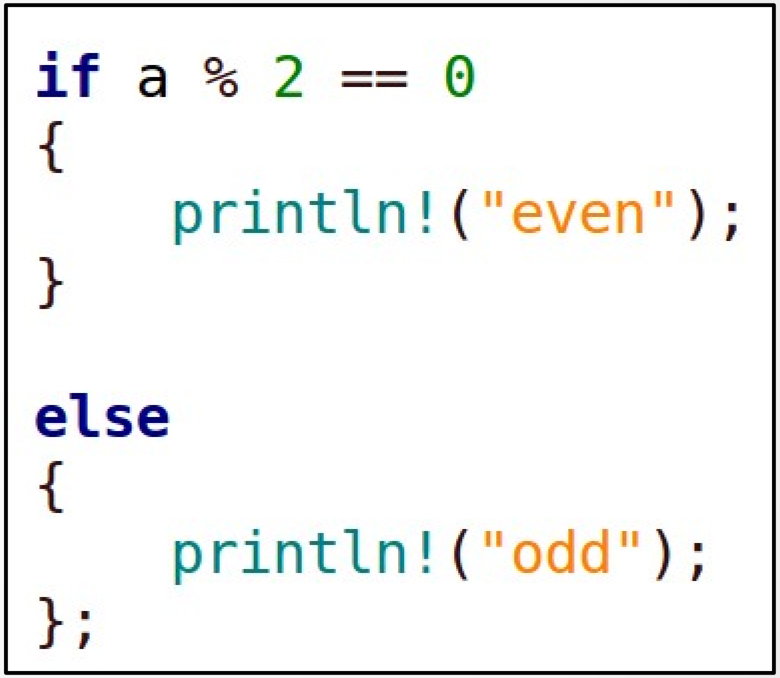
General Form

The *else* and *else if* blocks are optional.

Multiple *else if* blocks are possible.

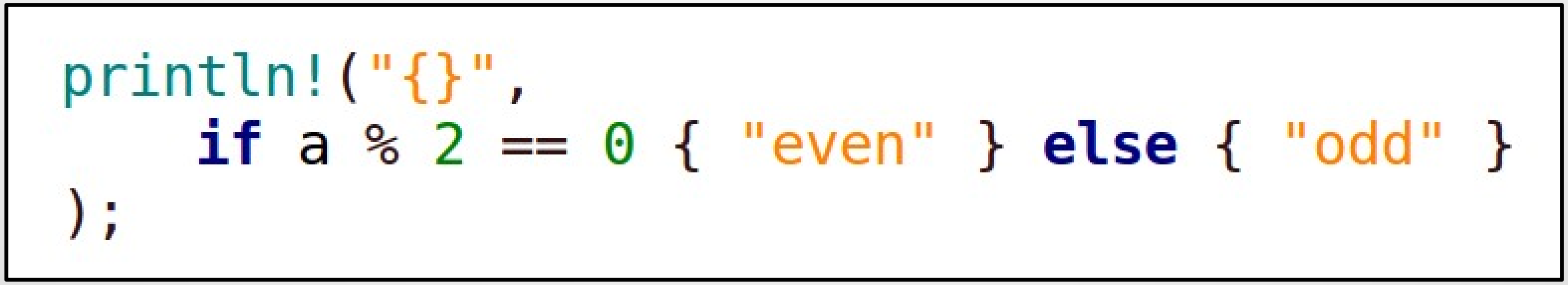
The *if* Expression

Conditional Statement



```
if a % 2 == 0
{
    println!("even");
}

else
{
    println!("odd");
};
```

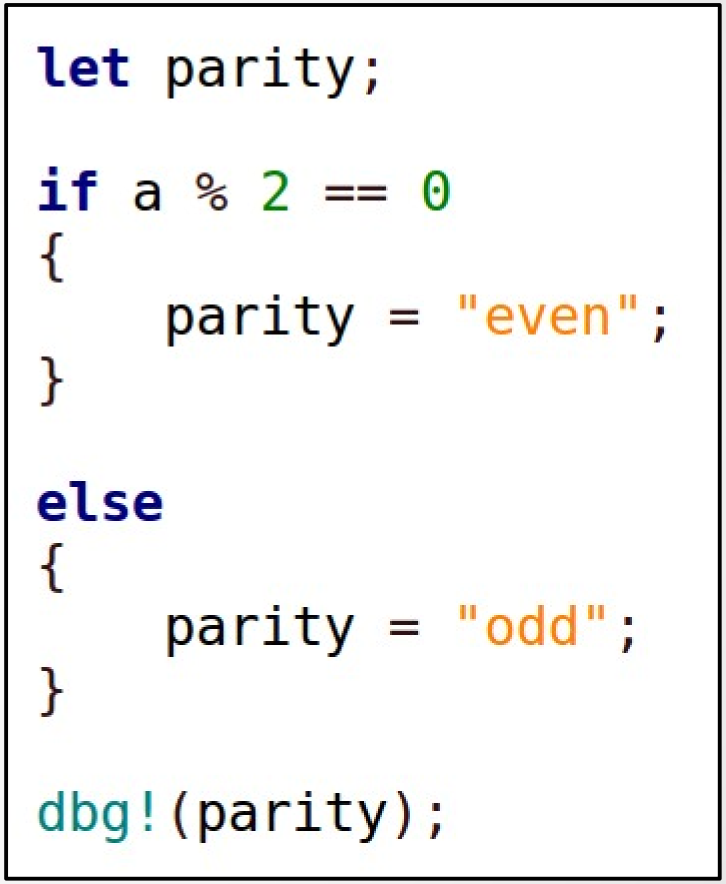
Conditional Expression



```
println!("{}",
    if a % 2 == 0 { "even" } else { "odd" }
);
```

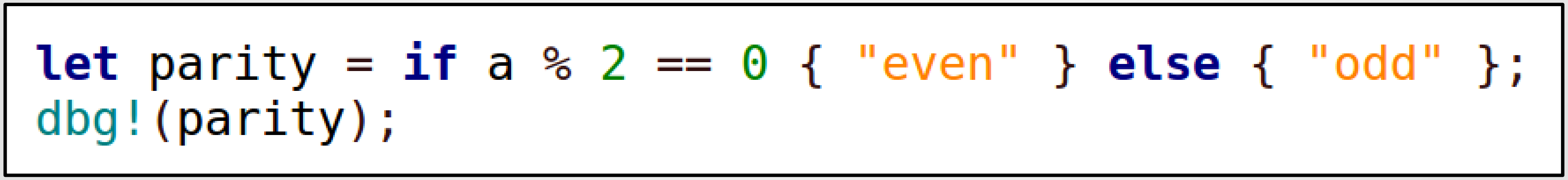
The *if* Expression

Conditional Statement



```
let parity;

if a % 2 == 0
{
    parity = "even";
}

else
{
    parity = "odd";
}

dbg!(parity);
```

Conditional Expression



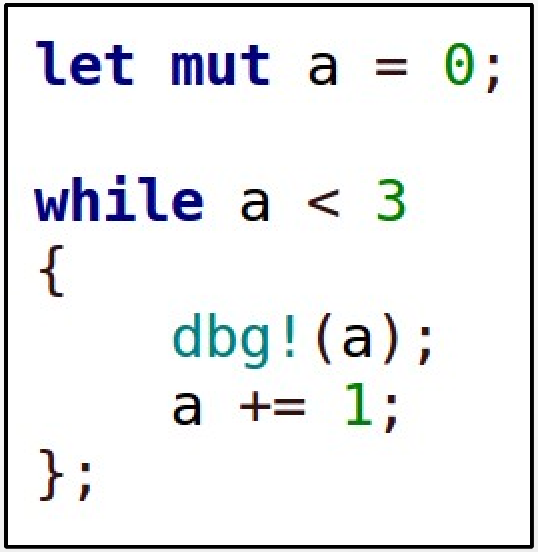
```
let parity = if a % 2 == 0 { "even" } else { "odd" };
dbg!(parity);
```

Conditional Loops (*while*)

```
while condition
{
    // ...
}
```

General Form

Example



```
let mut a = 0;

while a < 3
{
    dbg!(a);
    a += 1;
};
```

```
a = 0
a = 1
a = 2
```

Conditional Loops (*for*)

```
for var in iterator
{
    // ...
}
```

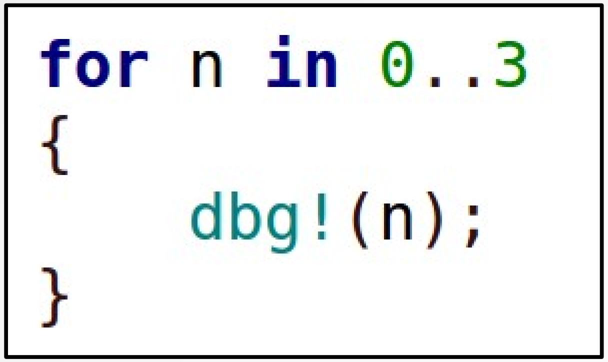
General Form

An iterator is a type specification.
We will study iterators in a further lesson.
For now, we will use simple kinds of iterators : Ranges

For this lesson

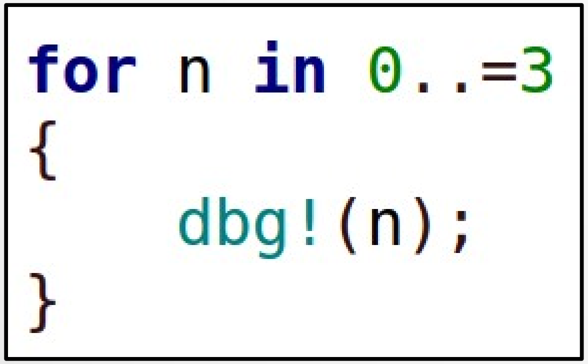
```
for var in range
{
    // ...
}
```

Conditional Loops (*for*)



```
for n in 0..3
{
    dbg!(n);
}
```

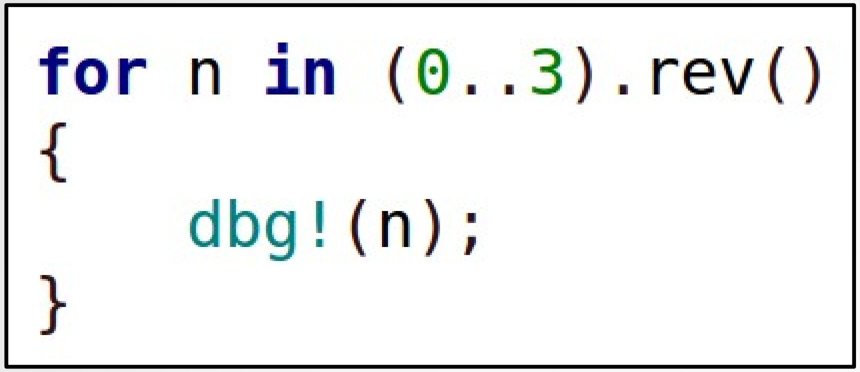
```
n = 0
n = 1
n = 2
```



```
for n in 0..=3
{
    dbg!(n);
}
```

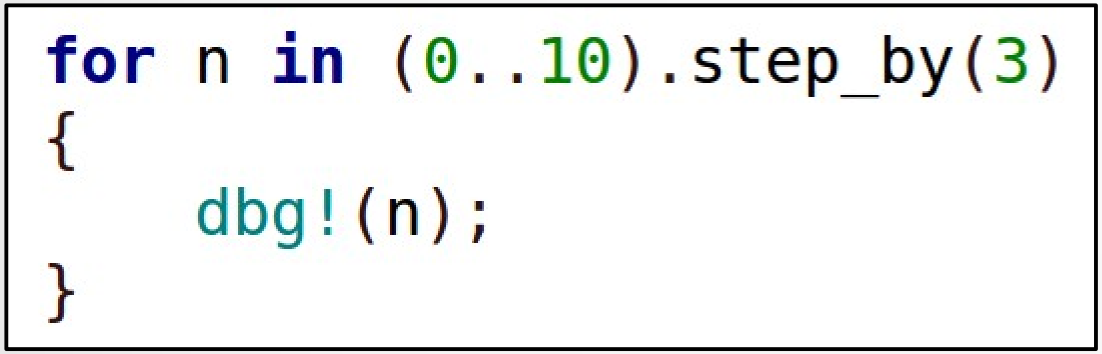
```
n = 0
n = 1
n = 2
n = 3
```

Conditional Loops (*for*)



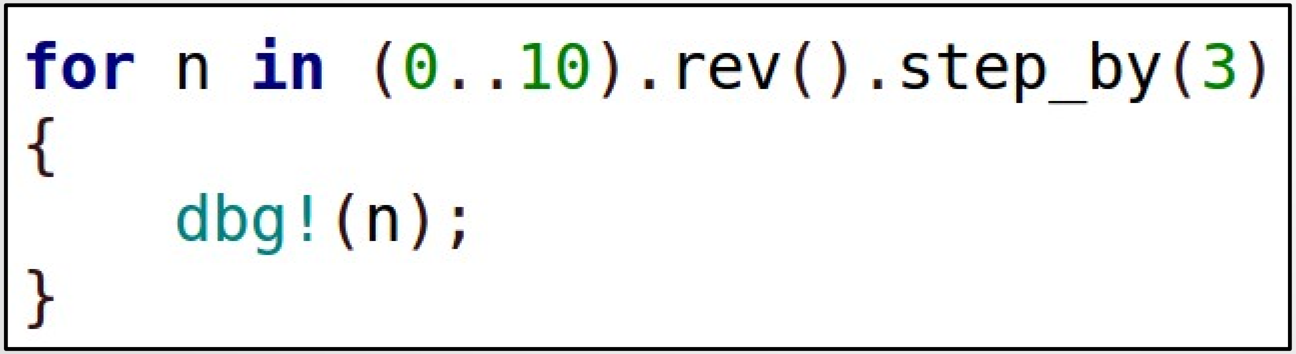
```
for n in (0..3).rev()
{
    dbg!(n);
}
```

```
n = 2
n = 1
n = 0
```



```
for n in (0..10).step_by(3)
{
    dbg!(n);
}
```

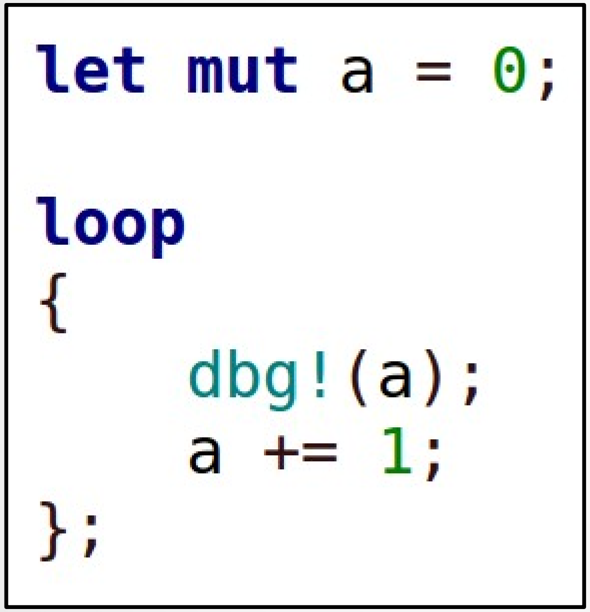
```
n = 0
n = 3
n = 6
n = 9
```



```
for n in (0..10).rev().step_by(3)
{
    dbg!(n);
}
```

```
n = 9
n = 6
n = 3
n = 0
```

Infinite Loops (*loop*)



```
let mut a = 0;

loop
{
    dbg!(a);
    a += 1;
};
```

```
a = 0
a = 1
a = 2
a = 3
# ... snip ...
a = 41302
a = 41303
^C
```

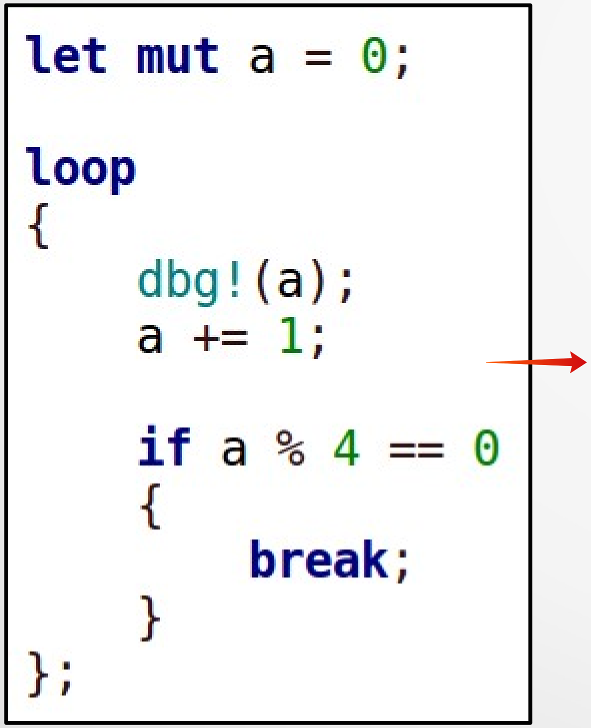
break and *continue*

The *break* and *continue* instructions can be used in loop bodies (*for*, *while*, *loop*)

- *break*: Terminates the loop.
- *continue*: Goes to the next iteration.

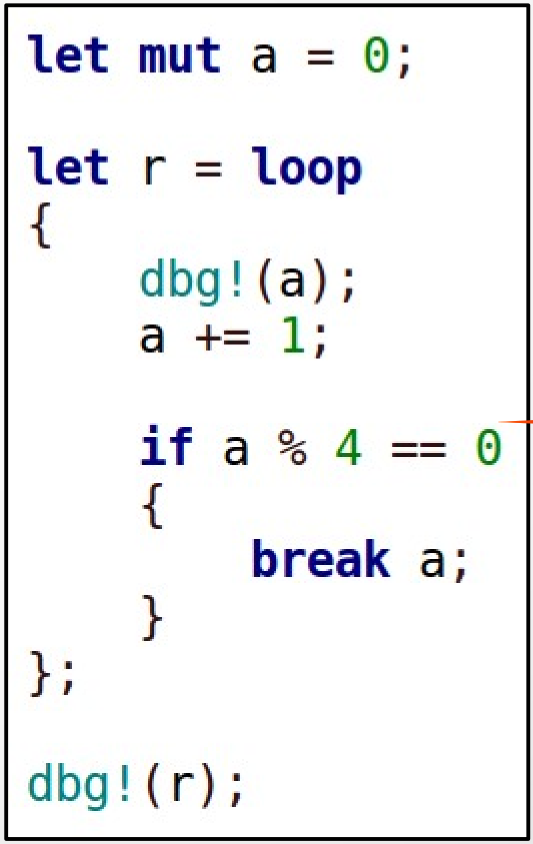
loop and *break*

When used with *loop*,
break can return a value.



```rust
let mut a = 0;

loop
{
    dbg!(a);
    a += 1;

    if a % 4 == 0
    {
        break;
    }
};
```

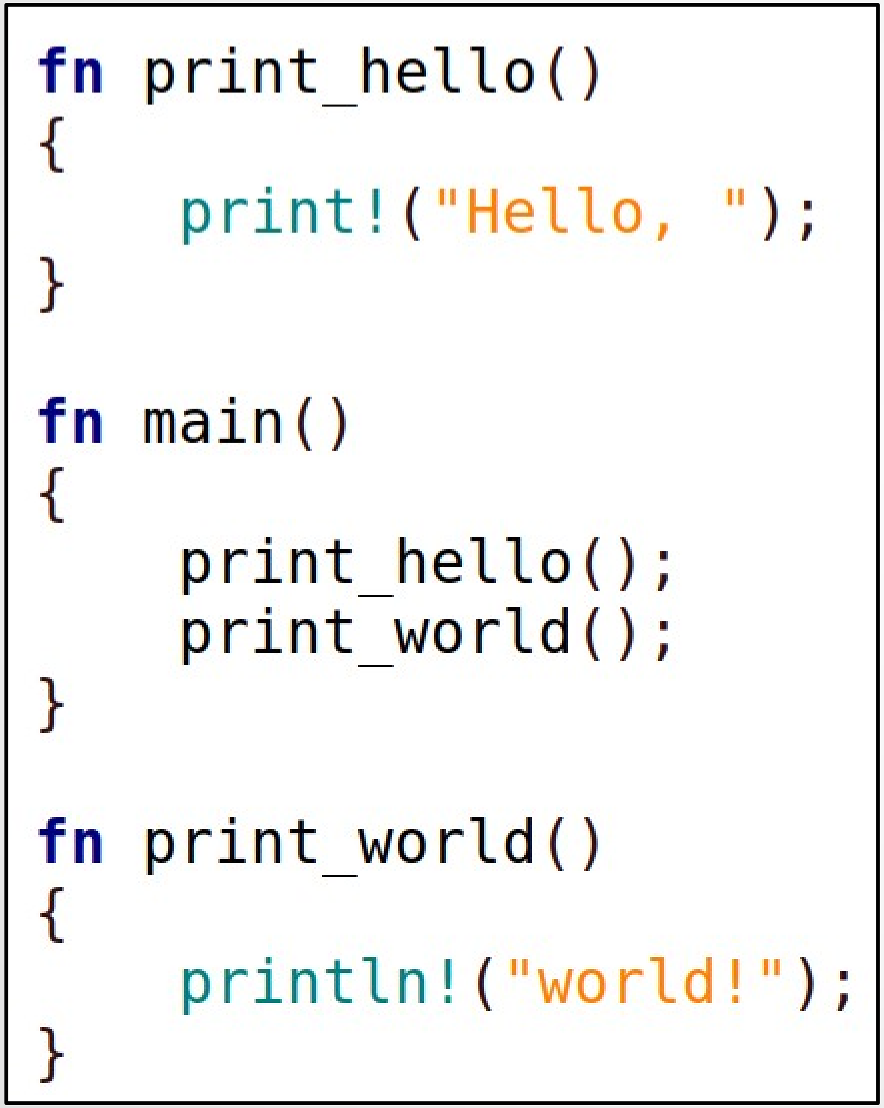
```
a = 0
a = 1
a = 2
a = 3
```



```rust
let mut a = 0;

let r = loop
{
    dbg!(a);
    a += 1;

    if a % 4 == 0
    {
        break a;
    }
};

dbg!(r);
```

```
a = 0
a = 1
a = 2
a = 3
r = 4
```

Defining Functions



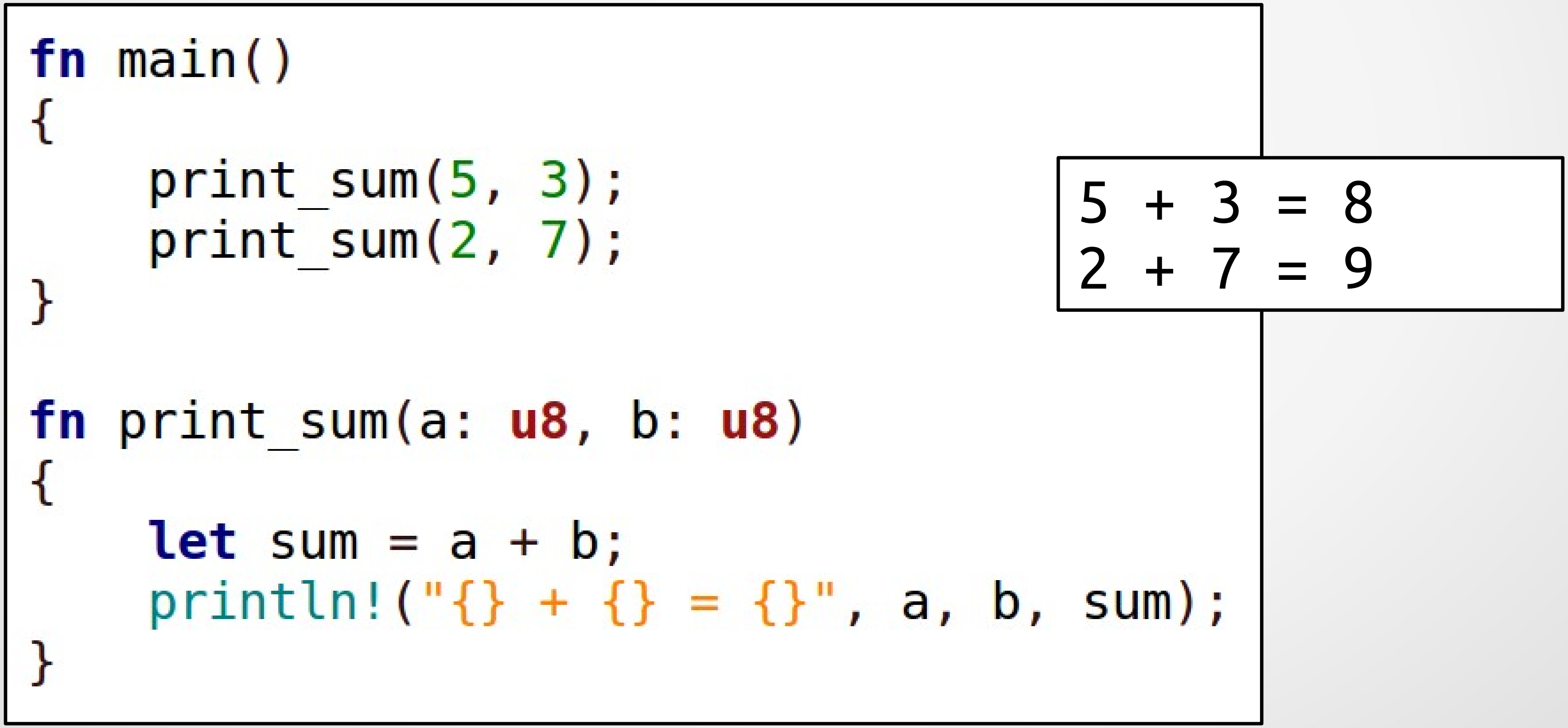
```
fn print_hello()
{
    print!("Hello, ");
}

fn main()
{
    print_hello();
    print_world();
}

fn print_world()
{
    println!("world!");
}
```

```
Hello, world!
```

Functions can be defined anywhere.

Passing Arguments to Functions

Types of parameters must be specified.

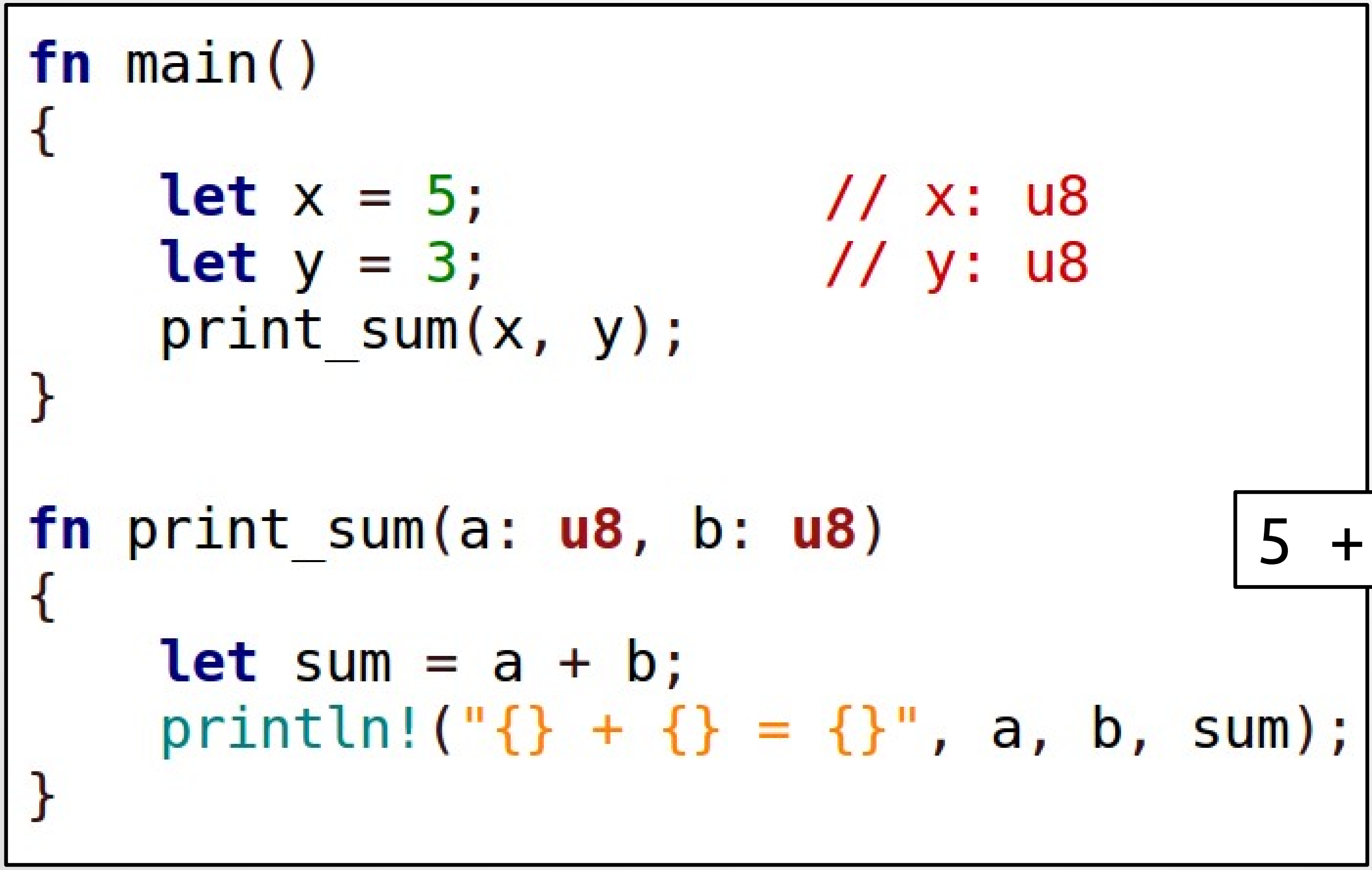


```rust
fn main()
{
    print_sum(5, 3);
    print_sum(2, 7);
}

fn print_sum(a: u8, b: u8)
{
    let sum = a + b;
    println!("{} + {} = {}", a, b, sum);
}
```

```
5 + 3 = 8
2 + 7 = 9
```

Passing Arguments to Functions

Type Inference

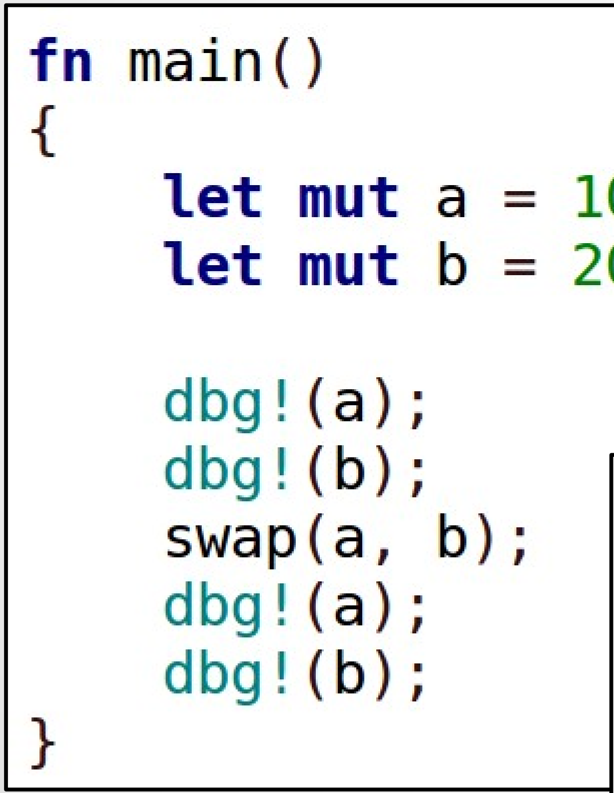


```
fn main()
{
    let x = 5;              // x: u8
    let y = 3;              // y: u8
    print_sum(x, y);
}

fn print_sum(a: u8, b: u8)
{
    let sum = a + b;
    println!("{} + {} = {}", a, b, sum);
}
```

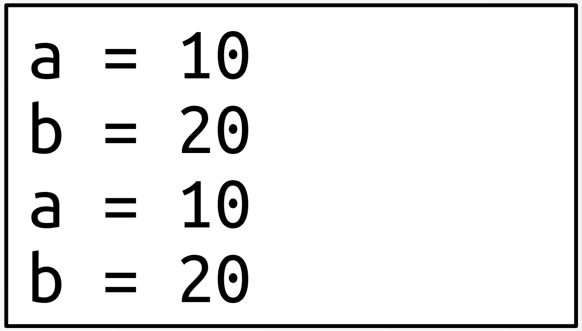
```
5 + 3 = 8
```

Passing Arguments by Value

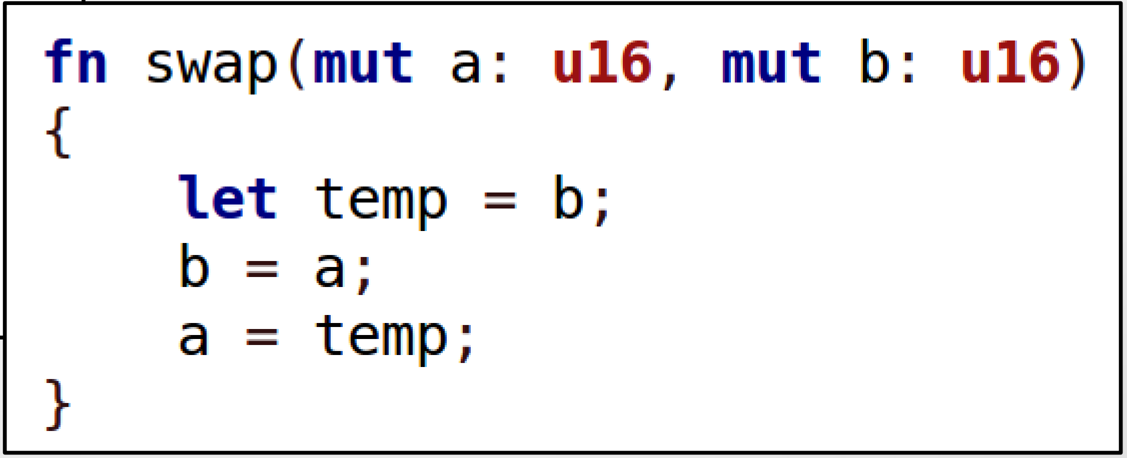
By default, arguments are passed by value.



```
fn main()
{
    let mut a = 10;
    let mut b = 20;

    dbg!(a);
    dbg!(b);
    swap(a, b);
    dbg!(a);
    dbg!(b);
}
```

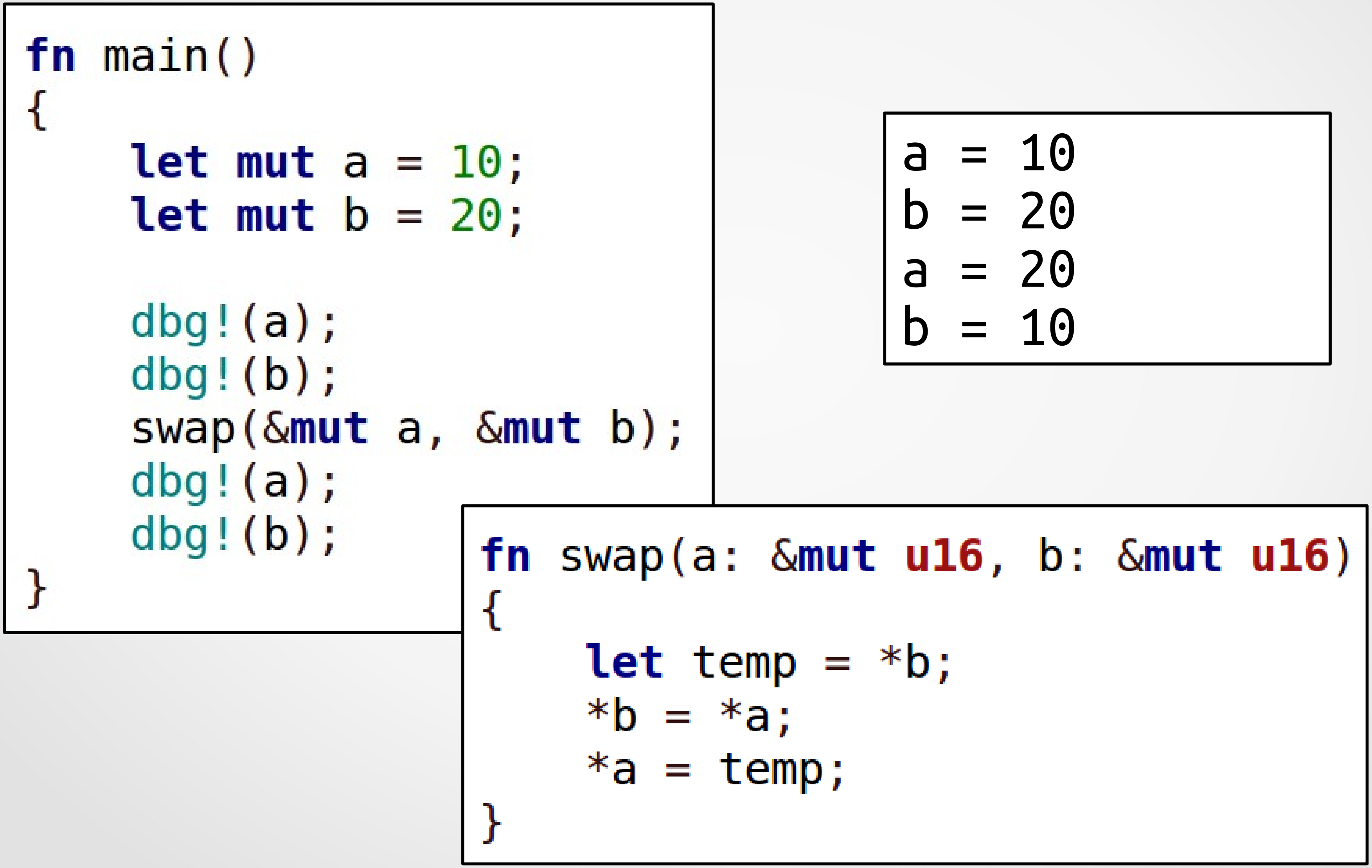


```
a = 10
b = 20
a = 10
b = 20
```



```
fn swap(mut a: u16, mut b: u16)
{
    let temp = b;
    b = a;
    a = temp;
}
```

Passing Arguments by Reference



```rust
fn main()
{
    let mut a = 10;
    let mut b = 20;

    dbg!(a);
    dbg!(b);
    swap(&mut a, &mut b);
    dbg!(a);
    dbg!(b);
}
```

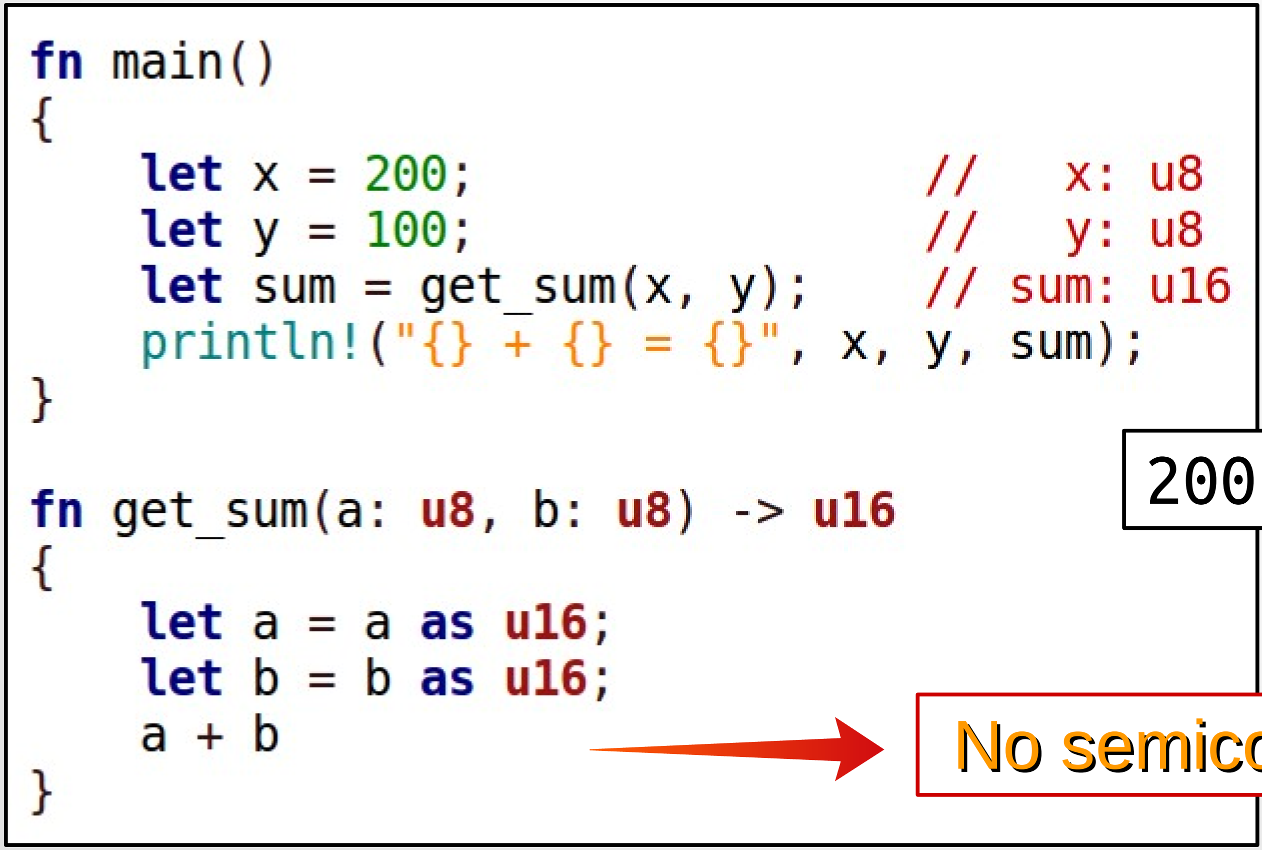
```
a = 10
b = 20
a = 20
b = 10
```

```rust
fn swap(a: &mut u16, b: &mut u16)
{
    let temp = *b;
    *b = *a;
    *a = temp;
}
```

Returning Values from Functions

Types of return values must be specified.

Return values are those of the blocks.



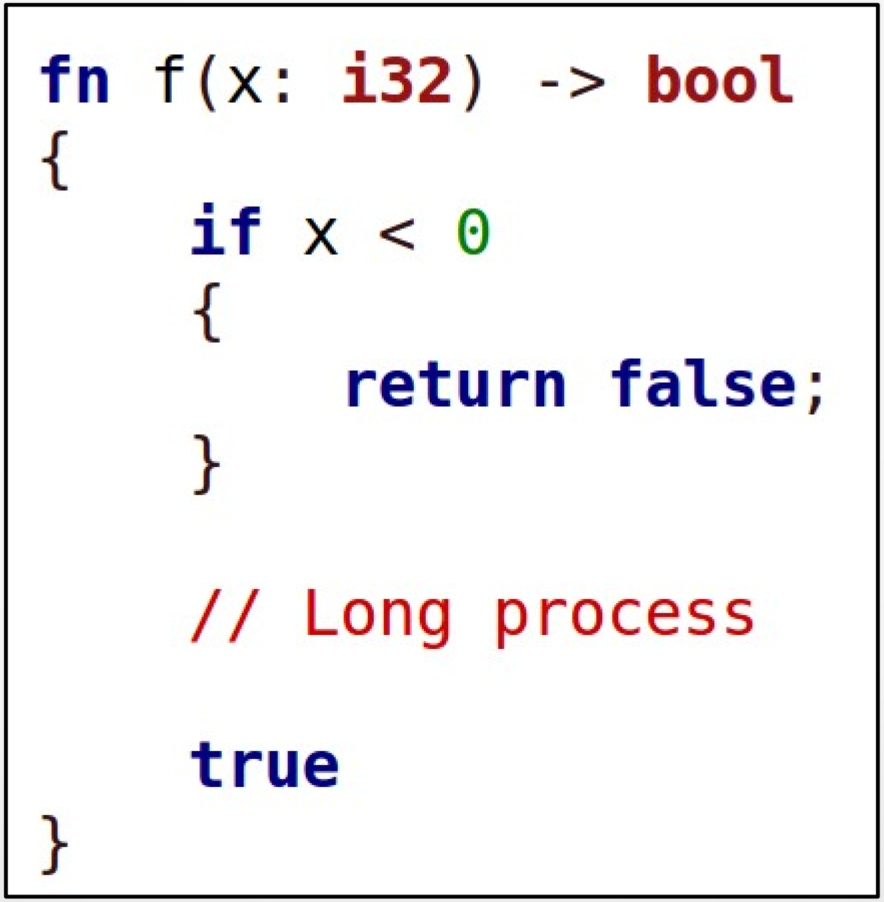
```
fn main()
{
    let x = 200;                  //   x: u8
    let y = 100;                  //   y: u8
    let sum = get_sum(x, y);      // sum: u16
    println!("{} + {} = {}", x, y, sum);
}

fn get_sum(a: u8, b: u8) -> u16
{
    let a = a as u16;
    let b = b as u16;
    a + b
}
```

```
200 + 100 = 300
```

No semicolon!

Early Returns (*return*)

Example



```
fn f(x: i32) -> bool
{
    if x < 0
    {
        return false;
    }

    // Long process

    true
}
```

Function Signatures

A ***function signature*** contains the *fn* keyword, the name of the function, all information about the function itself, its parameters and its return values.

For instance:

```
fn div(a: u64, b:u64) -> u64
```

is the ***function signature*** of *div()*.