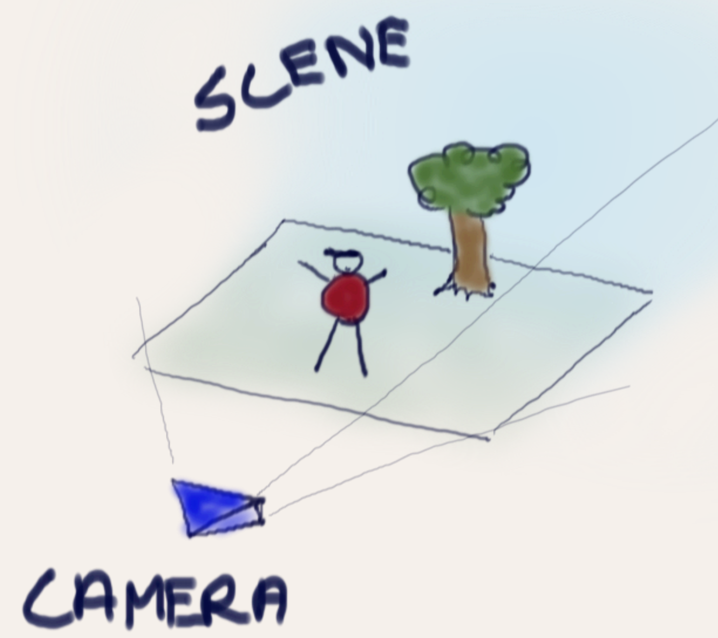
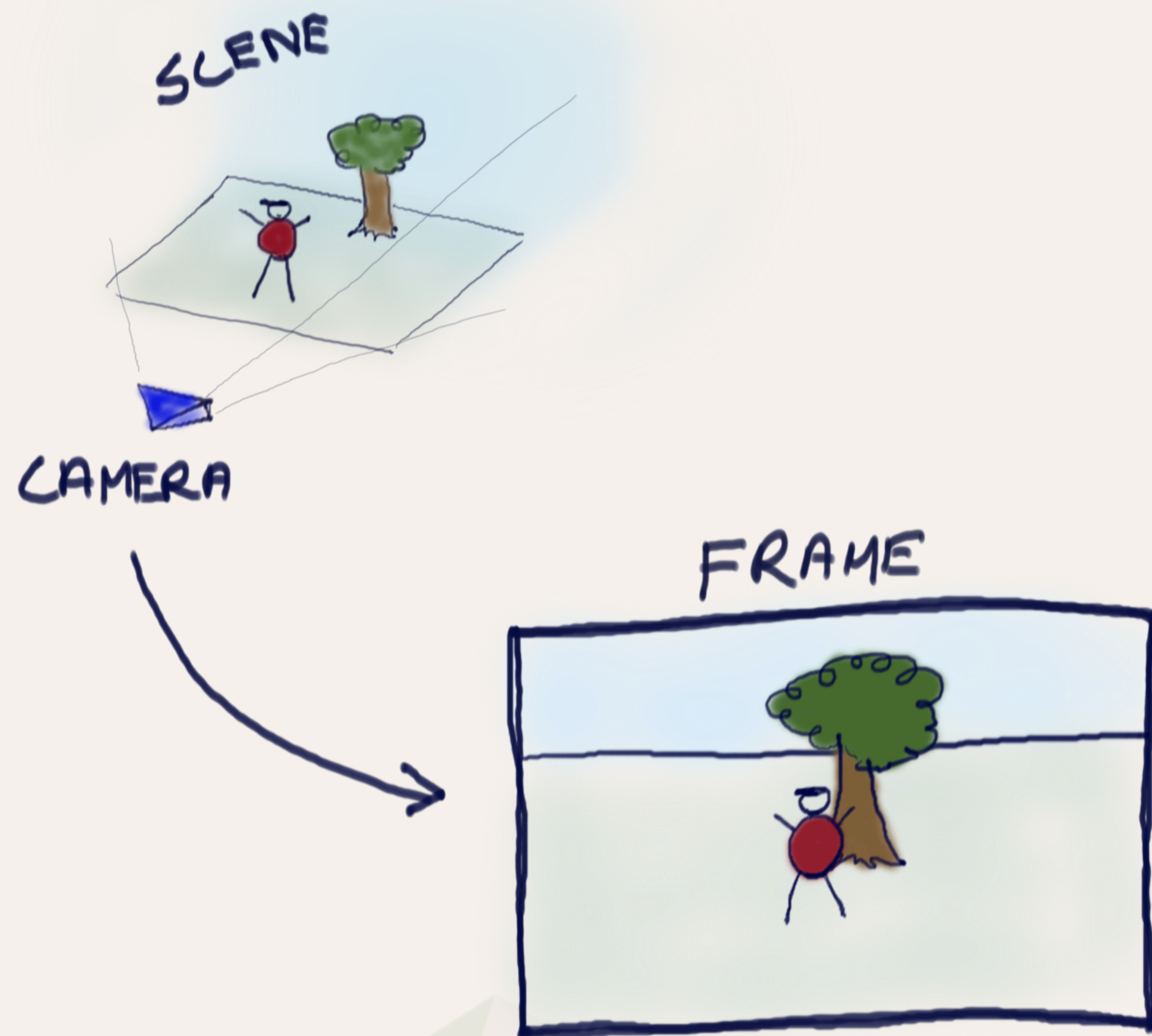
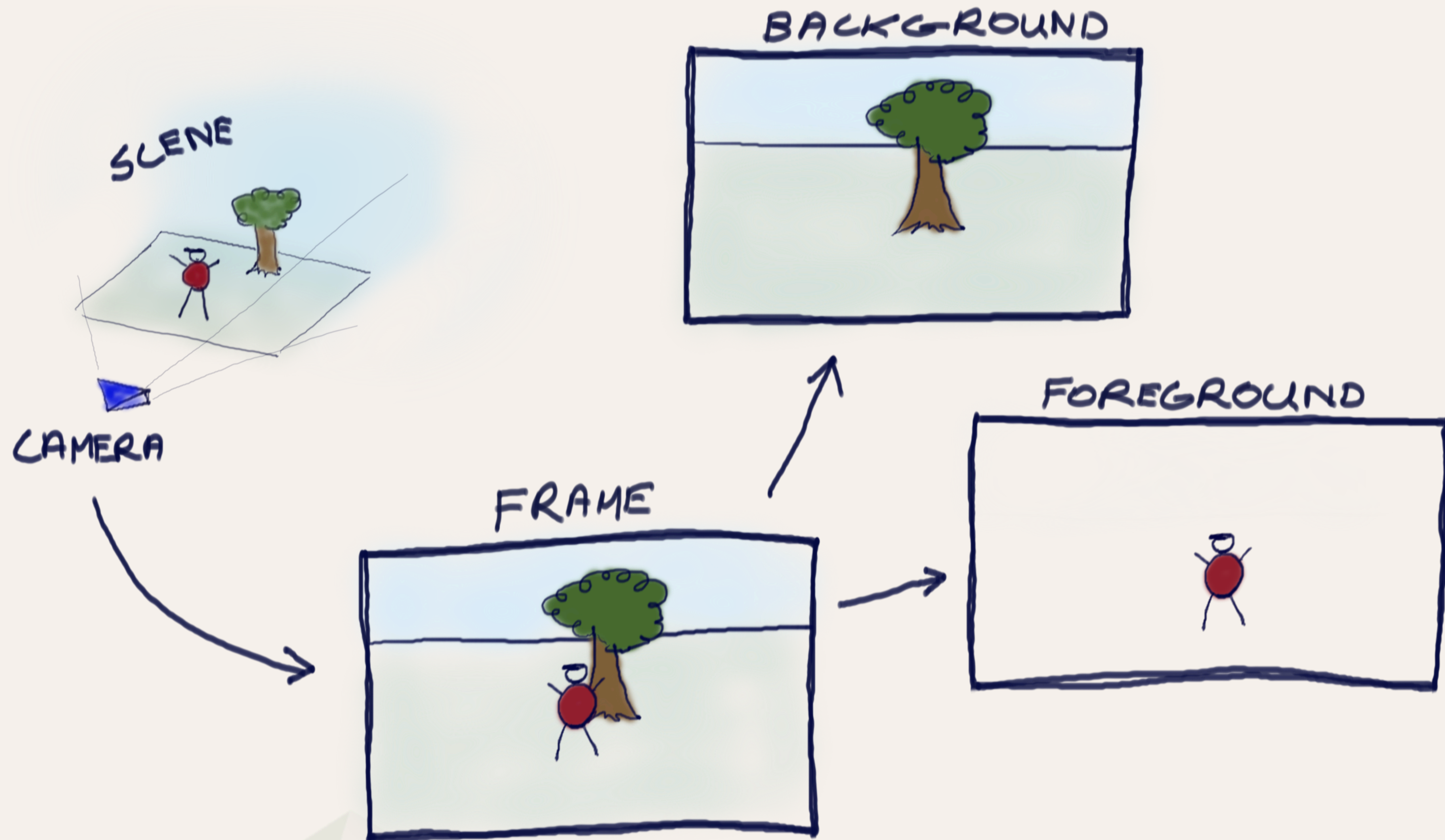
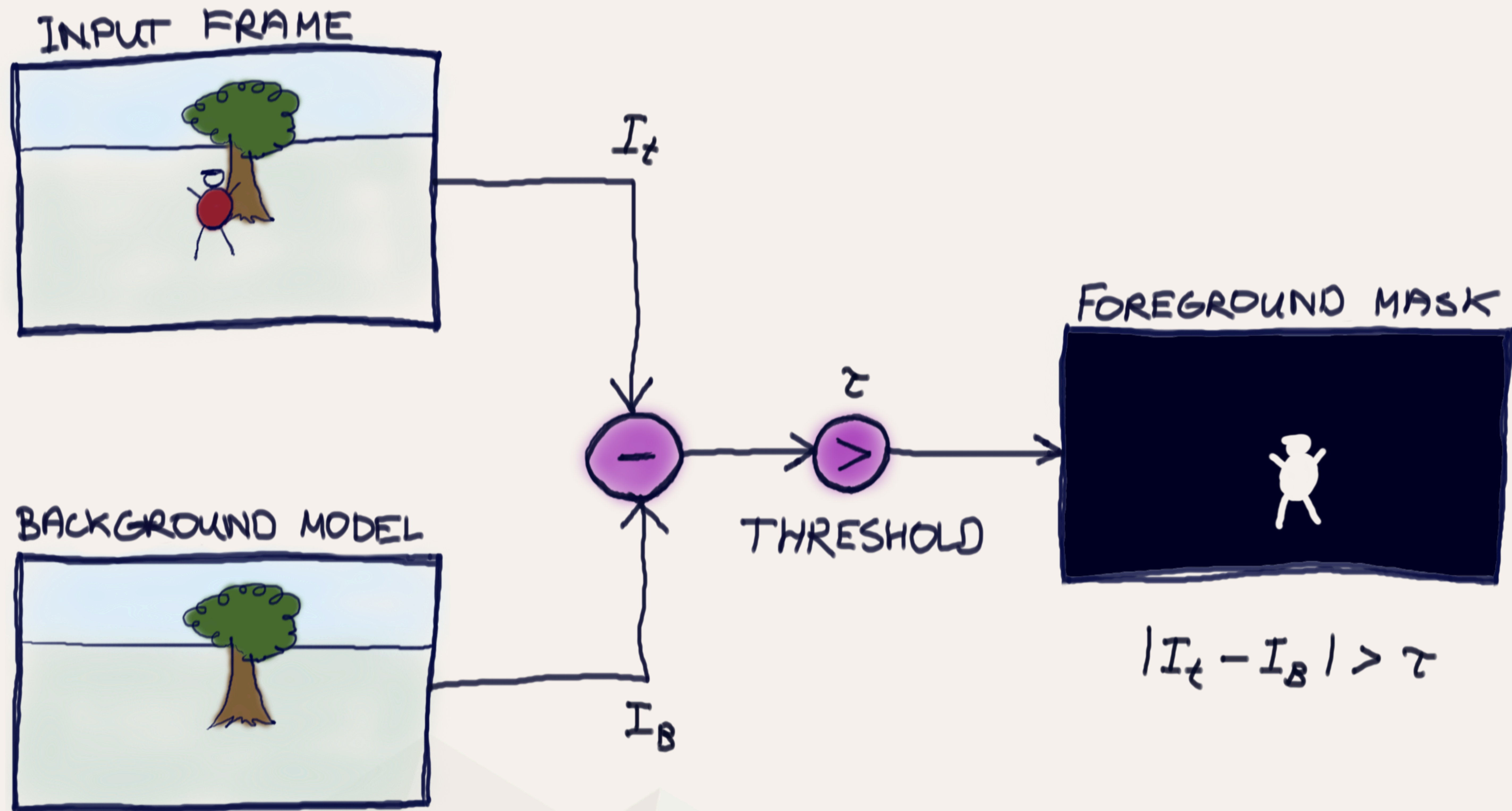
The Rust Programming Language

Kris van Rens


SCENE
CAMERA


SCENE
CAMERA
FRAME

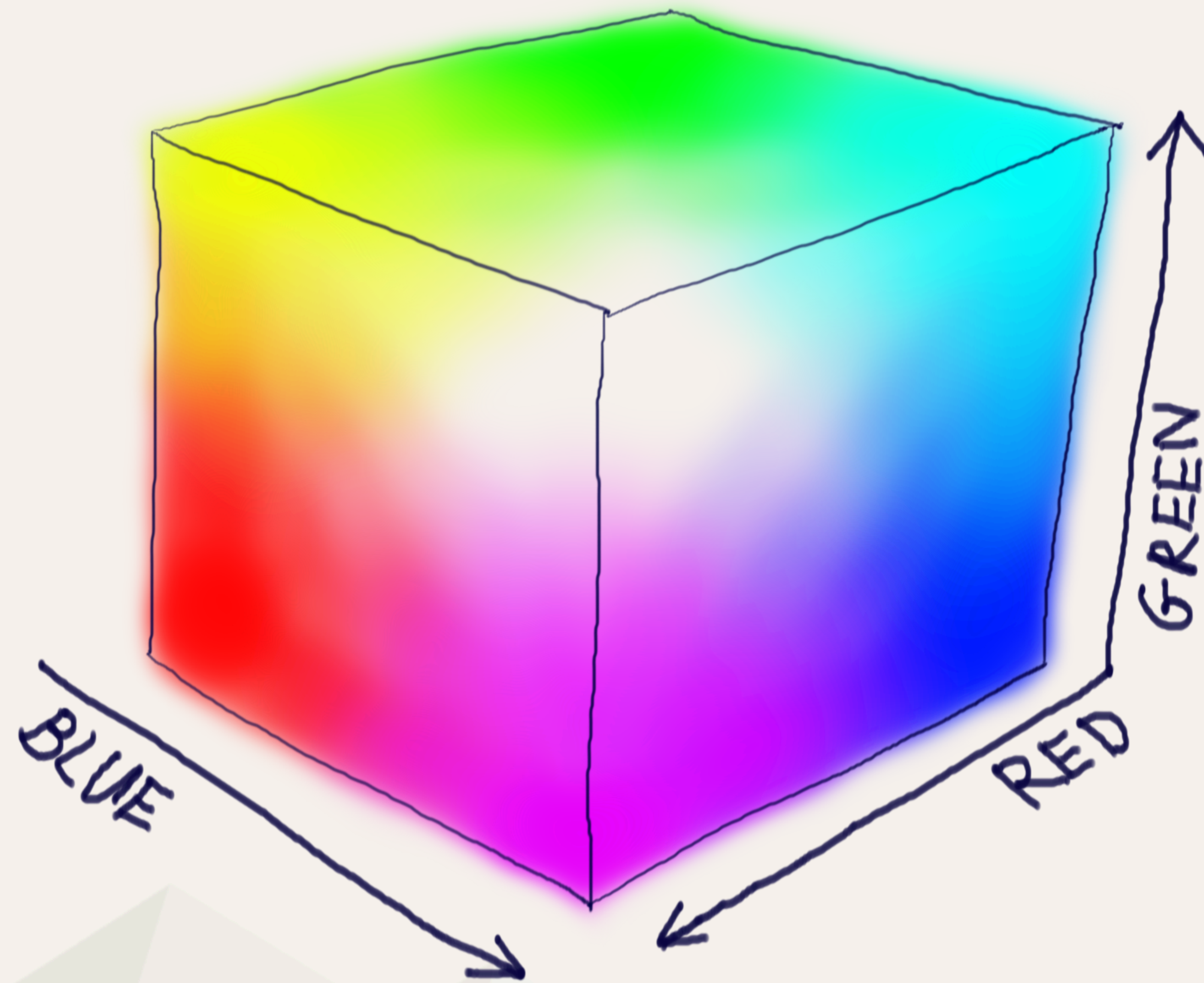

SCENE
CAMERA
BACKGROUND
FRAME
FOREGROUND


INPUT FRAME
I_t
BACKGROUND MODEL
I_B
−
τ
>
THRESHOLD
FOREGROUND MASK
$|I_t - I_B| > \tau$

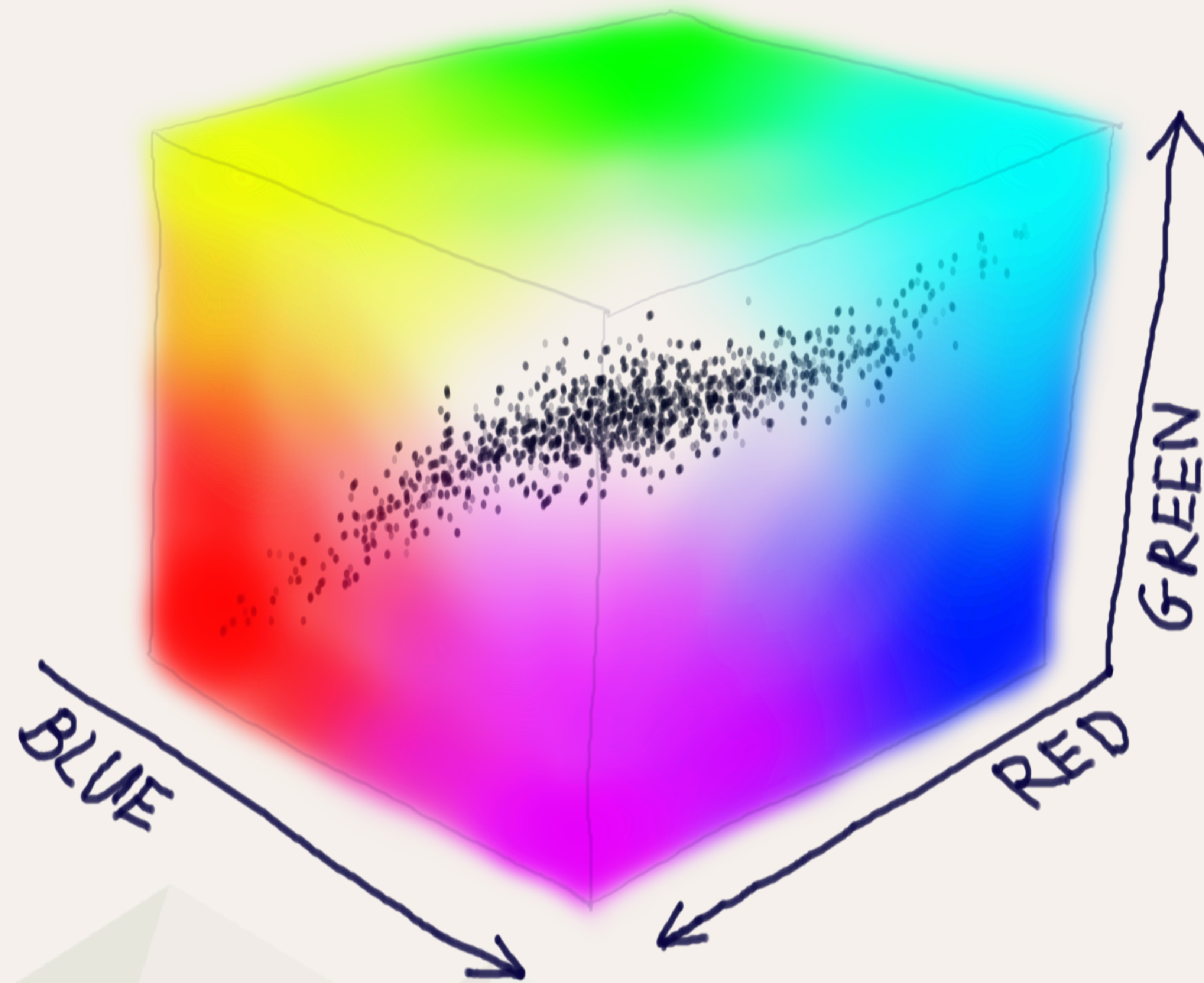
Which color channel and difference metric must we choose?

Let’s take a look at distribution: $d = |I_t - I_B|$

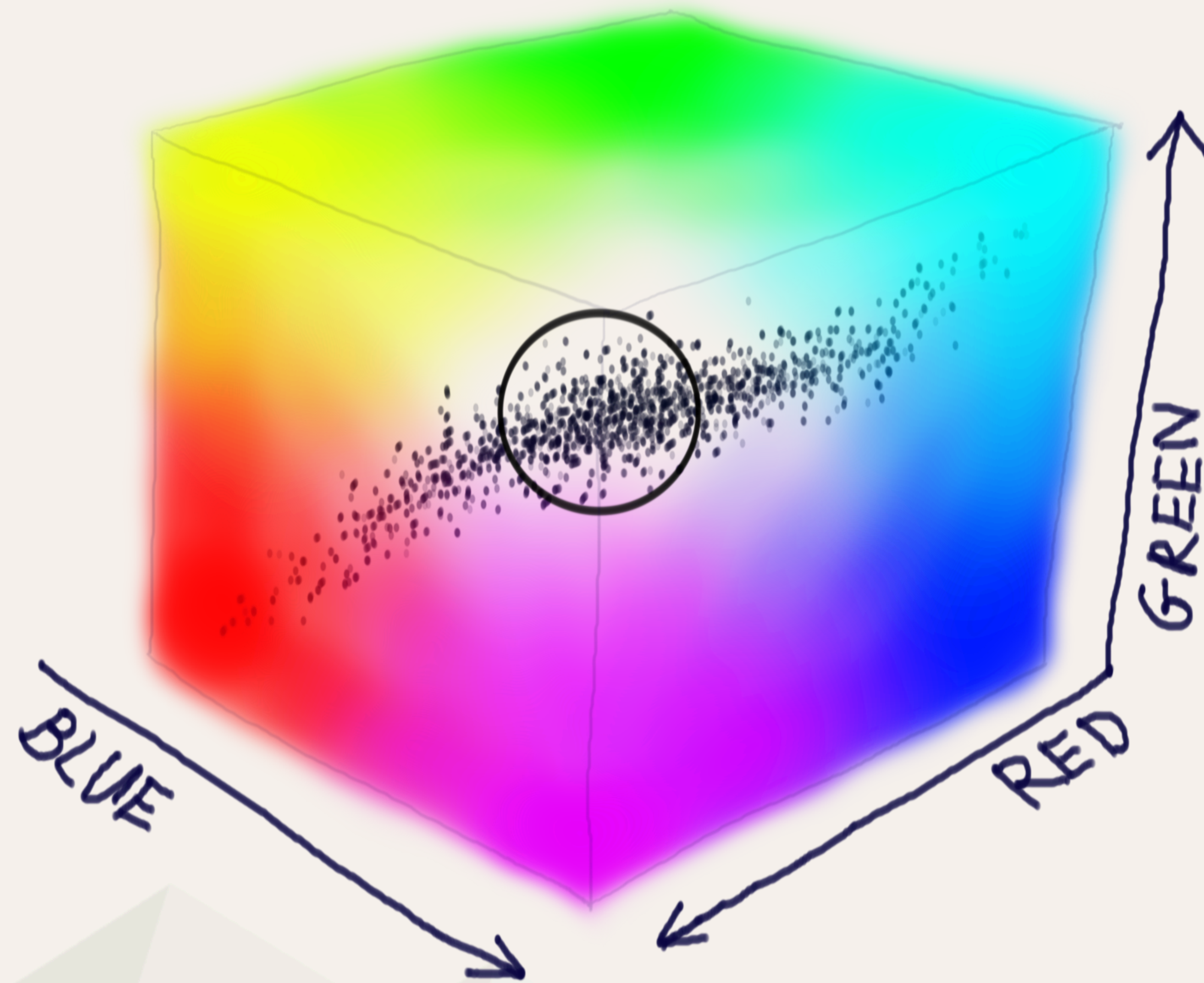
Pixel difference distribution: $d = |I_t - I_B|$ in RGB color space:



Pixel difference distribution: $d = |I_t - I_B|$ in RGB color space:



Pixel difference distribution: $d = |I_t - I_B|$ in RGB color space:

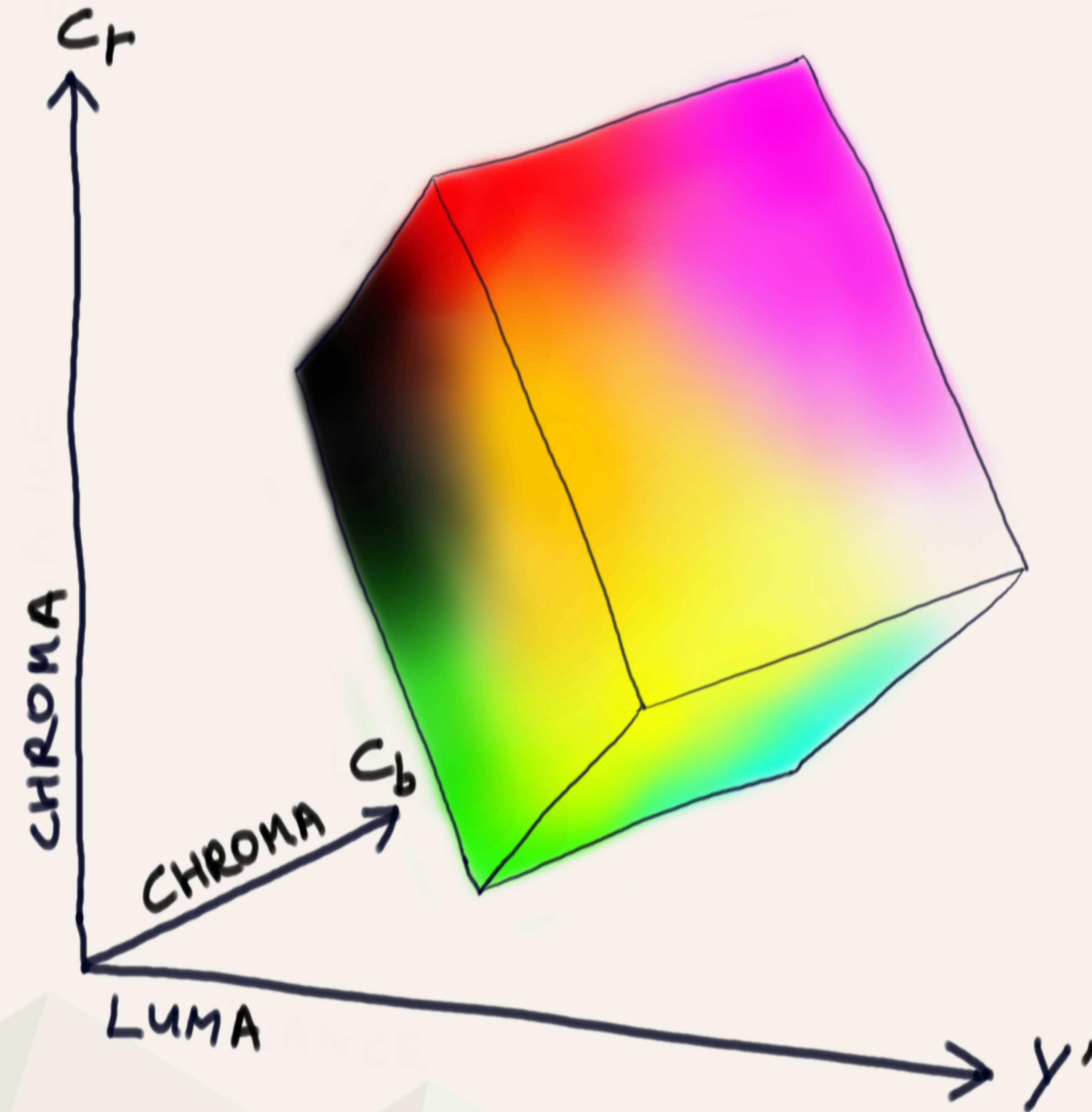


Conversion of RGB ⟷ Y'CbCr using BT.709 definitions:

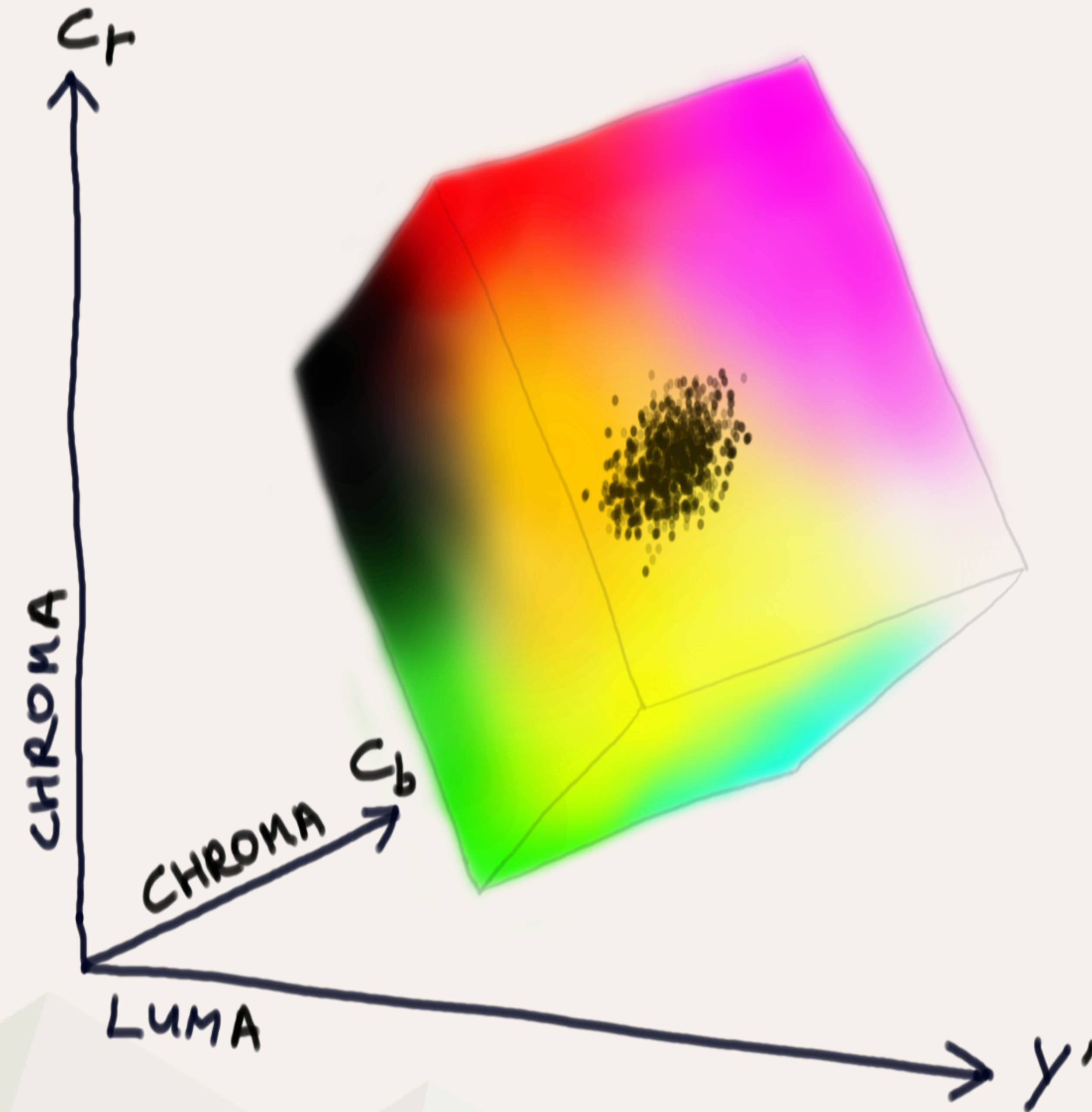
$$\begin{bmatrix} Y' \\ C_b \\ C_r \end{bmatrix} = \begin{bmatrix} 0.2126 & 0.7152 & 0.0722 \\ -0.09991 & -0.33609 & 0.436 \\ 0.615 & -0.55861 & -0.05639 \end{bmatrix} \begin{bmatrix} R' \\ G' \\ B' \end{bmatrix}$$

$$\begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.28033 \\ 1 & -0.21482 & -0.38059 \\ 1 & 2.12798 & 0 \end{bmatrix} \begin{bmatrix} Y' \\ C_b \\ C_r \end{bmatrix}$$

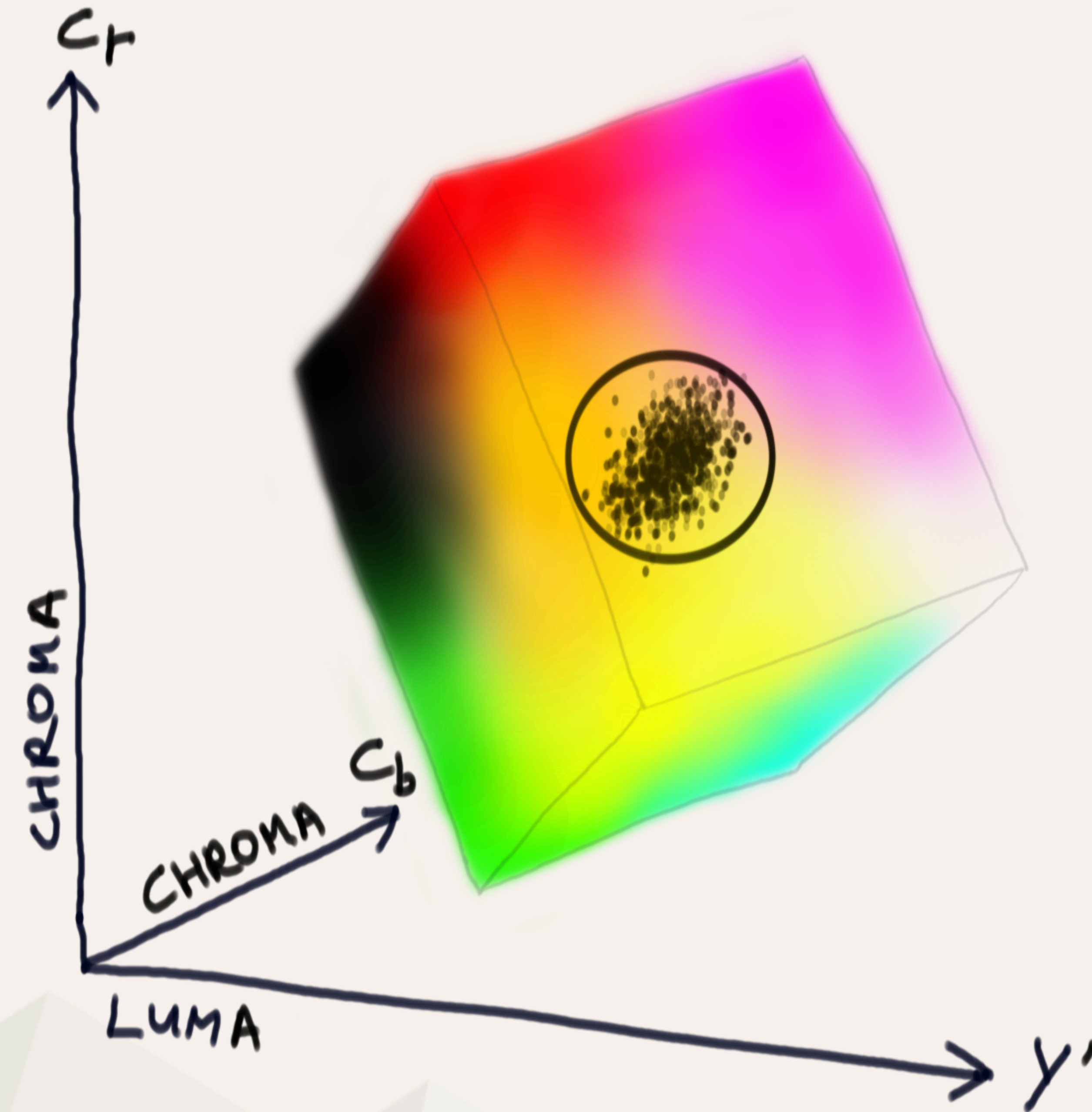
Pixel difference distribution: $d = |I_t - I_B|$ in Y'CbCr color space:



Pixel difference distribution: $d = |I_t - I_B|$ in Y'CbCr color space:



Pixel difference distribution: $d = |I_t - I_B|$ in Y'CbCr color space:



My point here..

Observing a problem from another angle can be enlightening!

> *Learning another programming language will improve your perspective on the language(s) you already know.*

It will improve your **problem-solving ability**.

What’s ahead?

- A Rust primer
- An in-depth look
- Moving on from here

A little bit about me

kris@vanrens.org

A Rust primer

We have a problem..

Root cause analysis of some very large C/C++ projects show that a consistent **70%** of software bugs are **memory errors**.

The promise of Rust

> *When a program written in safe Rust compiles, it is guaranteed to be free of memory errors and data races.*

The promise of Rust

> *When a program written in safe Rust **compiles**, it is **guaranteed** to be free of memory errors and data races.*

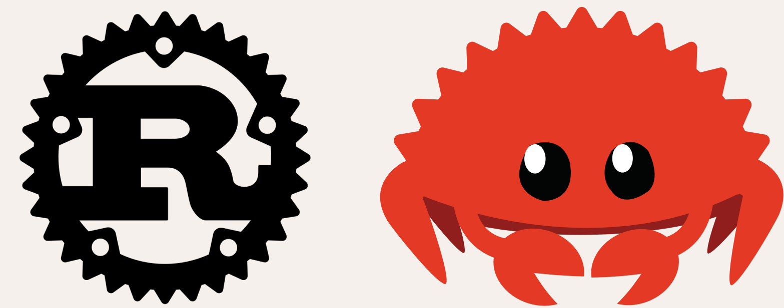
The promise of Rust

> *When a program written in safe Rust compiles, it is guaranteed to be free of* ***memory errors and data races****.*

The promise of Rust

> *When a program written in **safe Rust** compiles, it is guaranteed to be free of memory errors and data races.*

The (short) history of Rust



- Started as a personal project of Graydon Hoare in 2006,
- Mozilla started funding the project in 2009,
- The compiler, `rustc` became self-hosted in 2011,
- Reached v1.0 on the 15th of May, 2015,
- Covered by the Rust Foundation since February 2021,
- Accepted for Linux kernel driver development in 2021 (added in v6.1).

The main feature

Rust is designed to be **memory-safe**. It enforces this at **compile-time**.

Rust guarantees the compilation result to be **free of *undefined behavior***.

How Rust fulfills the promises

Meet the **borrow checker**

```rust
fn main() {
  let x = "Hi there!".to_string();

  let _y = x; // Move operation.

  println!("Message: {x}");
}
```

```
error[E0382]: borrow of moved value: `x`
 --> <source>:6:22
  |
2 |   let x = "Hi there!".to_string();
  |       - move occurs because `x` has type `String`, which does not
  |         implement the `Copy` trait
3 |
4 |   let _y = x; // Move operation.
  |            - value moved here
5 |
6 |   println!("Message: {x}");
  |                      ^^^ value borrowed here after move

error: aborting due to previous error

For more information about this error, try `rustc --explain E0382`.
```

How Rust fulfills the promises

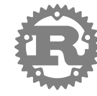
Meet the **borrow checker**

```rust
fn main() {
  let x;

  {
    let y = 42;
    x = &y;
  }

  assert_eq!(*x, 42);
}
```

```
error[E0597]: `y` does not live long enough
 --> <source>:6:13
  |
6 |     x = &y;
  |         ^^ borrowed value does not live long enough
7 |   }
  |   - `y` dropped here while still borrowed
8 |
9 |   assert_eq!(*x, 42);
  |   ------------------ borrow later used here

error: aborting due to previous error

For more information about this error, try `rustc --explain E0597`.
```

How Rust fulfills the promises

Lifetime parameters

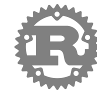


```rust
fn main() {
  struct X {
    v: &u8  // Takes a reference to a u8.
  }

  let answer = 42;
  let x = X{ v: &answer };
}
```

```
error[E0106]: missing lifetime specifier
 --> <source>:3:12
  |
3 |       v: & u8
  |          ^ expected named lifetime parameter
  |
help: consider introducing a named lifetime parameter
  |
2 ~   struct X<'a> {
3 ~       v: &u8
  |

error: aborting due to previous error

For more information about this error, try `rustc --explain E0106`.
```

How Rust fulfills the promises

Lifetime parameters

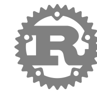


```rust
fn main() {
  struct X<'a> {
    v: &'a u8
  }

  let answer = 42;
  let x = X{ v: &answer };
}
```

```
OK!
```

How Rust fulfills the promises

Lifetime parameters in action!



```rust
1  fn main() {
2    struct X<'a> {
3      v: &'a u8
4    }
5
6    let x;
7    {
8      let answer = 42;
9      x = X{ v: &answer };
10   }
11
12   assert_eq!(*x.v, 42);
13 }
```

```
error[E0597]: `answer` does not live long enough
  --> <source>:9:18
   |
9  |     x = X{ v: &answer };
   |               ^^^^^^^ borrowed value does not live long enough
10 |   }
   |   - `answer` dropped here while still borrowed
11 |
12 |   assert_eq!(*x.v, 42);
   |   -------------------- borrow later used here

error: aborting due to previous error

For more information about this error, try `rustc --explain E0597`.
```

The compiler is a pair-programmer

```
$ rustc --explain E0597
```

```
This error occurs because a value was dropped while it was still borrowed.

Erroneous code example:
```

```rust
struct Foo<'a> {
    x: Option<&'a u32>,
}

let mut x = Foo { x: None };
{
    let y = 0;
    x.x = Some(&y); // error: `y` does not live long enough
}
println!("{:?}", x.x);
```

```
Here, `y` is dropped at the end of the inner scope, but it is borrowed by
`x` until the `println`. To fix the previous example, just remove the scope
so that `y` isn't dropped until after the println
```

```
...including a working example etc. etc....
```

The Rust ecosystem

We *can* have Nice Things™

Compiler: `rustc`

- **LLVM-based self-hosting compiler**, featuring the *borrow checker*
- Outputs **high-performance machine code** for many platforms
- Provides short, readable, **helpful error messages**
- **Conventional debuggers can be used**, like `gdb` and `lldb`
- Can be slow for larger projects, similar to template-heavy C++ compilation

Build system and package manager: Cargo

- **Downloads, compiles, distributes** and uploads packages
- Rust **packages are called *crates***, the official registry is crates.io

Build system and package manager: Cargo

Cargo expects a predefined directory structure:

```
my-project/
│
├── Cargo.lock
├── Cargo.toml
├── src/
│   ├── lib.rs
│   ├── main.rs
│   ├── my-module/
│   │   ├── mod.rs
│   │   └── some_module.rs
│   └── bin/
│       ├── some-executable.rs
│       └── multi-file-executable/
│           ├── main.rs
│           └── some_module.rs
│
└── tests/
    └── some-tests.rs
```

Cargo.toml

```toml
[package]
name = "my-project"
version = "0.1.0"
edition = "2021"

[dependencies]
time = "1.1.*"
```

Documentation generator `rustdoc`

```rust
//! Here's some info about my module; blah blah.
//! There are no memory errors in the code, yay!
//!
//! See this [GitHub] page for more info.
//!
//! [GitHub]: https://blah.github.io

/// This is a random example function.
///
/// It can be used as follows:
///
/// ```rust
/// let x = some_random_example();
/// ```
/// Please note that even the above code example
///  will be tested at document-generation time!
///
fn some_random_example() -> u32 {
  // ...
}
```

All crates | Click or press 'S' to search, '?' for more options...

Crate lib
See all lib's items
Functions
Crates
lib

Crate lib [-][src]

Here's some info about my module; blah blah. There are no memory errors in the code, yay!

See this GitHub page for more info.

Functions

some_random_example — This is a random example function.

All crates | Click or press 'S' to search, '?' for more options...

Other items in lib
Functions
some_random_example

Function lib::some_random_example [-][src]

```rust
pub fn some_random_example() -> u32
```

This is a random example function.

It can be used as follows:

```rust
let x = some_random_example();
```

Please note that even the above code example will be tested at document-generation time!

Various other tools and resources

- Formatter `rustfmt` (invoked as `cargo fmt`)
- LINTer `clippy` (invoked as `cargo clippy`)
- Language server for IDEs (e.g. use `rust-analyzer` in VSCode)
- Excellent online documentation and books

An in-depth look

The ownership model

Three simple rules, guarded by the **borrow checker**:

Rule 1*: Each value has an owner.*

Rule 2*: There can only be one owner at a time.*

Rule 3*: When the owner goes out of scope, the value is dropped.*

Language semantics

Variables are constant by default

```rust
// Constants of type 'i32'.
let value1a = 42;
let value1b : i32 = 42;
let value1c = 42i32;

// Boolean constants.
let value2a = true;
let value2b : bool = true;

// Double-precision IEEE floats.
let value3a : f64 = 3.14159265;
let value3b = 3.14159265_f64;
let value3c : f64 = 6.022e23;

// UTF-8 string.
let value4 = "I am a string".to_string();

// Non-owning reference to UTF-8 text.
let value5 : &str = "I am a string too";

// Constant tuple value.
let value6 : (u32, String) = (12345, "hello".to_string());
```

```rust
// Destructuring a tuple into two constant values.
let (value7, value8) = value6;

struct Something {
  id: char,
  length: u8
}

// Constant data structure.
let value9 = Something{ id: 'x', length: 16 };

enum Price {
  Free,
  Nonfree(u32)
}

// Constant enumerator values.
let value10a = Price::Free;
let value10b = Price::Nonfree(3000);

// Owning pointer to value on the heap.
let value11 = Box::new(Price::Nonfree(17));
```

Language semantics

Variables are constant by default

```rust
// Constants of type 'i32'.
let mut value1a = 42;
let mut value1b : i32 = 42;
let mut value1c = 42i32;

// Boolean constants.
let mut value2a = true;
let mut value2b : bool = true;

// Double-precision IEEE floats.
let mut value3a : f64 = 3.14159265;
let mut value3b = 3.14159265_f64;
let mut value3c : f64 = 6.022e23;

// UTF-8 string.
let mut value4 = "I am a string".to_string();

// Non-owning reference to UTF-8 text.
let mut value5 : &str = "I am a string too";

// Constant tuple value.
let mut value6 : (u32, String) = (12345, "hello".to_string());
```

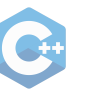
```rust
// Destructuring a tuple into two constant values.
let (mut value7, mut value8) = value6;

struct Something {
  id: char,
  length: u8
}

// Constant data structure.
let mut value9 = Something{ id: 'x', length: 16 };

enum Price {
  Free,
  Nonfree(u32)
}

// Constant enumerator values.
let mut value10a = Price::Free;
let mut value10b = Price::Nonfree(3000);

// Owning pointer to value on the heap.
let mut value11 = Box::new(Price::Nonfree(17));
```

Language semantics

Rust uses move semantics as a default, some types are `Copy`

```rust
let value1 : u8 = 42;
let value2 = value1;

println!("{value1}, {value2}");
```

```
42, 42
```



```cpp
uint8_t value1 = 42;
auto value2 = value1;

std::print("{}, {}\n", value1, value2);
```

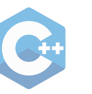
```
42, 42
```

Language semantics

Rust uses move semantics as a default, some types are `Copy`

```rust
let value1 : u8 = 42;
let value2 = value1;

println!("{value1}, {value2}");

let value3 = "Hello!".to_string();
let value4 = value3;

println!("{value3}, {value4}");
```

```
Compiler error!
```



```cpp
uint8_t value1 = 42;
auto value2 = value1;

std::print("{}, {}\n", value1, value2);

std::string value3 = "Hello!";
auto value4 = std::move(value3);

std::print("{}, {}\n", value3, value4);
```

```
42, 42
, Hello!
```

Language semantics

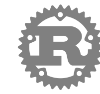
Rust uses move semantics as a default

```rust
let value1 : u8 = 42;
let value2 = value1;

// OK; u8 is a Copy type.
println!("{value1}, {value2}");

let value3 = "Hello!".to_string();
let value4 = value3;

// Error; String is not a Copy type.
println!("{value3}, {value4}");
```

```
error[E0382]: borrow of moved value: `value3`
  --> <source>:12:13
   |
8  |   let value3 = "Hello!".to_string();
   |       ------ move occurs because `value3` has type `String`, which
   |              does not implement the `Copy` trait
9  |   let value4 = value3;
   |                ------ value moved here
...
12 |   println!("{value3}, {value4}");
   |             ^^^^^^^^ value borrowed here after move

error: aborting due to previous error

For more information about this error, try `rustc --explain E0382`.
```

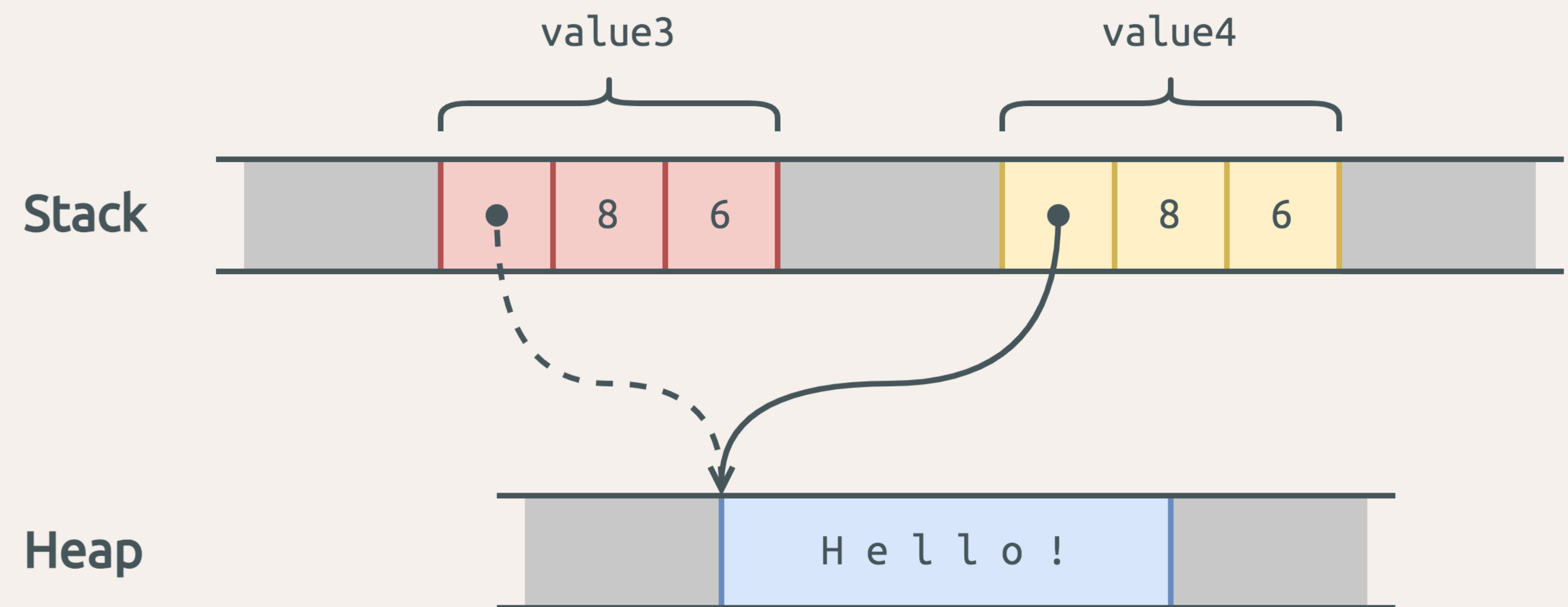
Language semantics

Rust uses move semantics as a default



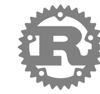
```rust
let value1 : u8 = 42;
let value2 = value1;

// OK; u8 is a Copy type.
println!("{value1}, {value2}");

let value3 = "Hello!".to_string();
let value4 = value3;

// Error; String is not a Copy type.
println!("{value3}, {value4}");
```

```
error: borrow of moved value: `value3`
...
```



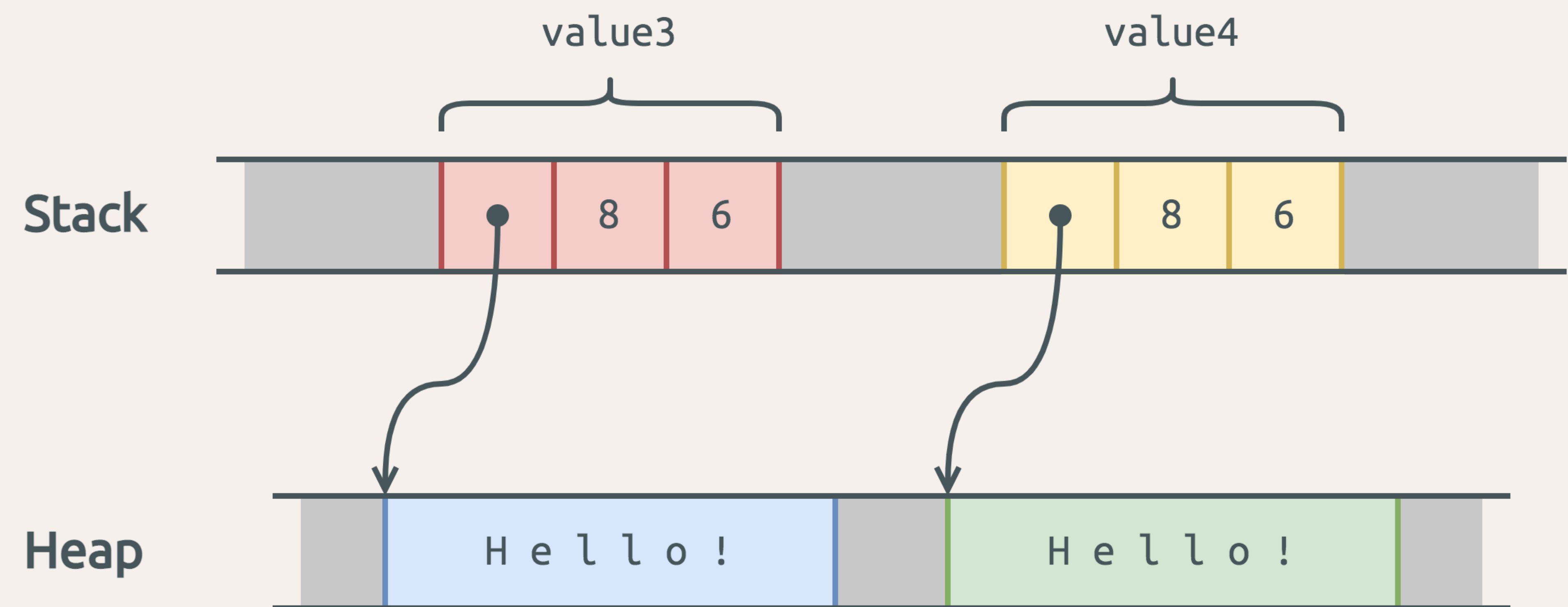
Language semantics

Rust uses move semantics as a default

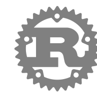


```
1  let value1 : u8 = 42;
2  let value2 = value1;
3
4  // OK; u8 is a Copy type.
5  println!("{value1}, {value2}");
6
7  let value3 = "Hello!".to_string();
8  let value4 = value3.clone();
9
10 // OK; String was explicitly copied.
11 println!("{value3}, {value4}");
```

```
42, 42
Hello!, Hello!
```



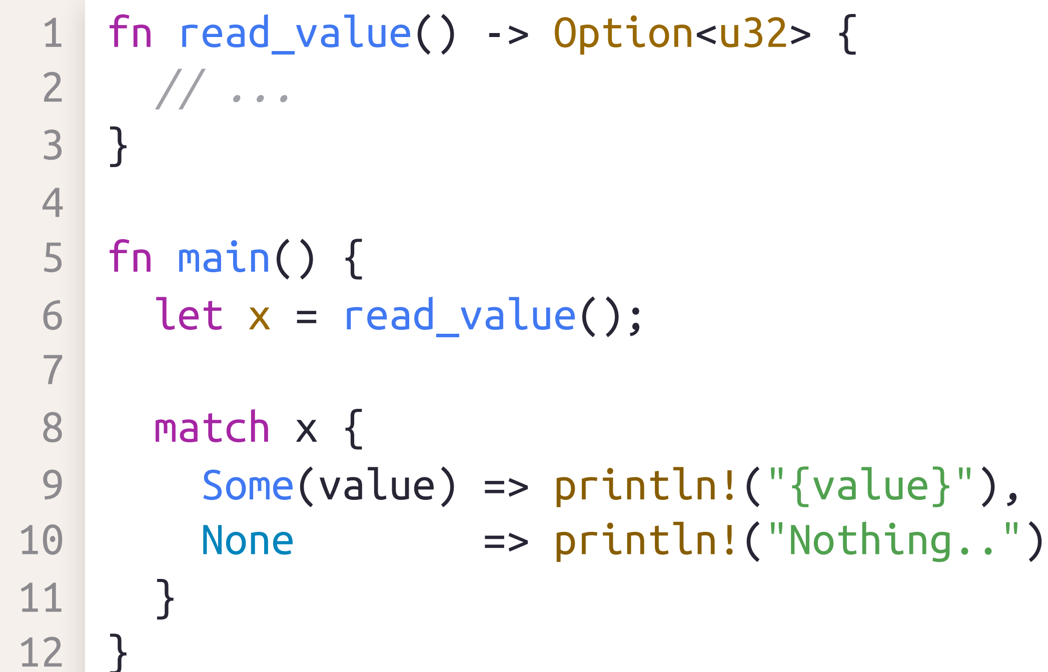
Many things are expressions



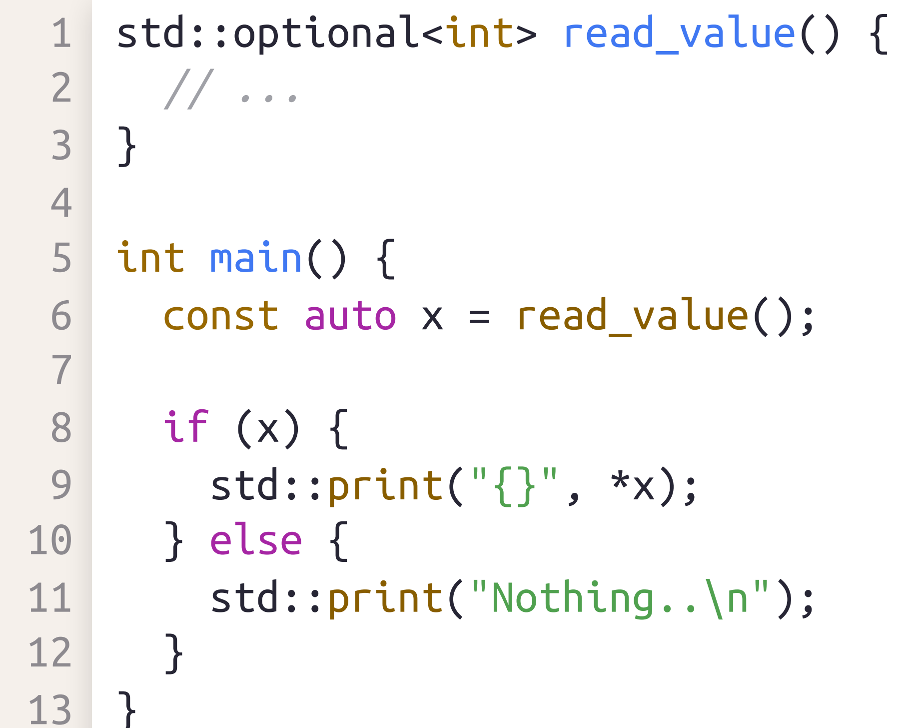
```rust
let a = 23;  // let declaration.
a + 17  // Expression evaluating to value 40.

let mut b = 0;  // let declaration.
b = 23  // Assignment is NOT an expression in Rust!

// Block expression.
let c = {
  let something = do_something();

  if something.value >= b {
    do_something_else(b);
  }

  something.value
};

// Conditional block expression.
let d = if must_be_pi() {
  3.14159265
} else {
  (0..100).sum()
};
```

```rust
// Match expression.
let e = match fruit {
  Mango   => 123,
  Avocado => 456,
  Grape   => 789,
  _       => println!("Unrecognized fruit!")
};

// if let expression.
if let Some(fruit) = get_optional_fruit() {
  eat(fruit);
} else {
  be_hungry();
}

// Loop expression.
let f = loop {
  if let Some(needle) = search_haystack() {
    break needle.to_string()
  } else {
    break "Found nothing".to_string()
  }
};
```

Result types: Option<T>



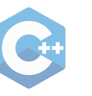
```rust
fn read_value() -> Option<u32> {
  // ...
}

fn main() {
  let x = read_value();

  match x {
    Some(value) => println!("{value}"),
    None        => println!("Nothing..")
  }
}
```



```cpp
std::optional<int> read_value() {
  // ...
}

int main() {
  const auto x = read_value();

  if (x) {
    std::print("{}", *x);
  } else {
    std::print("Nothing..\n");
  }
}
```

Result types: Option<T>

```rust
fn read_value() -> Option<u32> {
  return Some(42);
}

fn main() {
  let x = read_value();

  match x {
    Some(v) if v > 9999 => println!("So high!"),
    Some(1337)          => println!("1337 h4x0r!!1"),
    Some(value)         => println!("{value}"),
    None                => println!("Nothing..")
  }
}
```



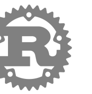
```cpp
std::optional<int> read_value() {
  return 42;
}

int main() {
  const auto x = read_value();

  if (x) {
    if (x > 9999) {
      std::print("So high!\n");
    } else if (x == 1337) {
      std::print("1ee7 h4x0r!!1\n");
    } else {
      std::print("{}\n", *x);
    }
  } else {
    std::print("Nothing..\n");
  }
}
```

Result types: `Option<T>`

`if let`

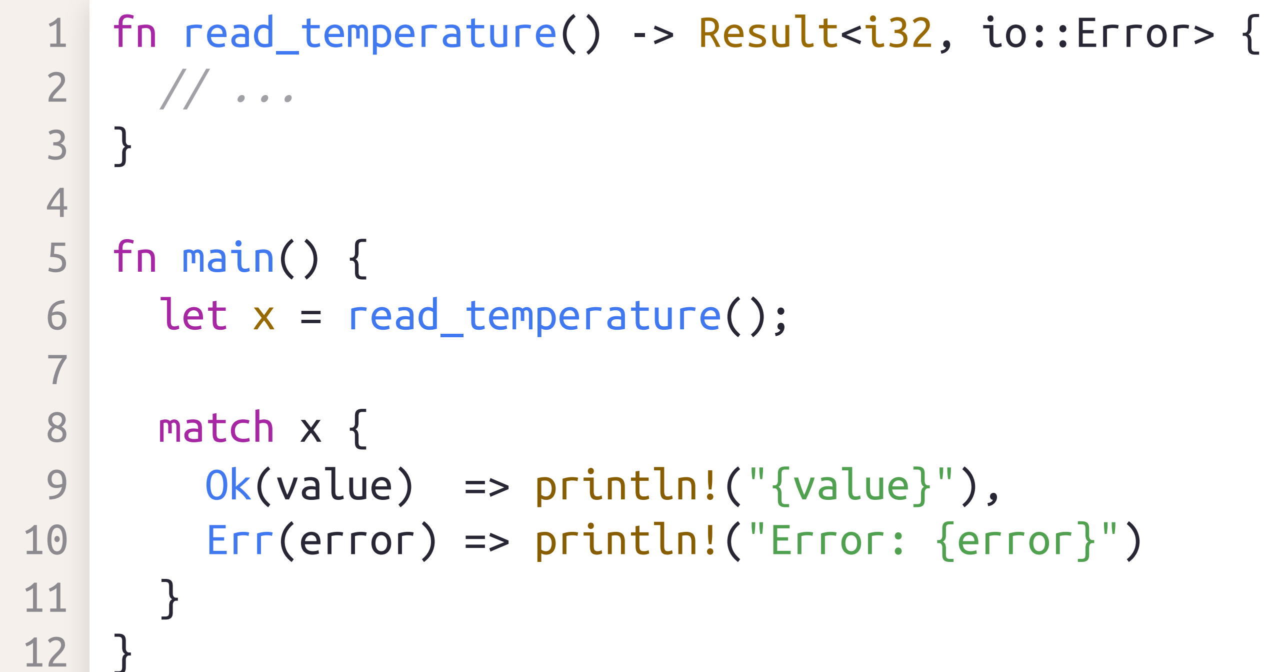
```rust
fn read_value() -> Option<u32> {
  // ...
}

fn do_something(value: u32) {
  // ...
}

fn main() {
  if let Some(value) = read_value() {
    do_something(value);
  }
}
```

`let else`

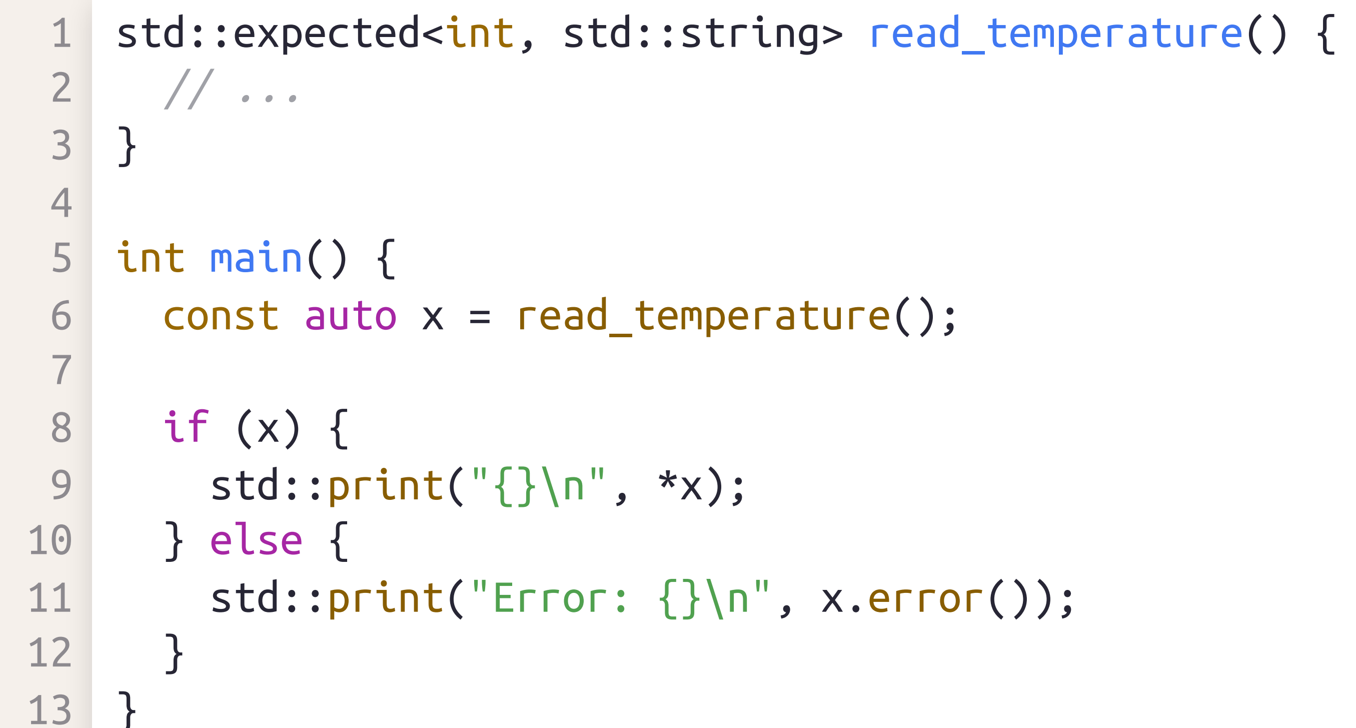


```rust
fn read_value() -> Option<u32> {
  // ...
}

fn do_something(value: u32) {
  // ...
}

pub fn main() {
  let Some(value) = read_value() else {
    panic!("Failed to read value")
  };

  do_something(value);
}
```

Result types: `Result<T, E>`



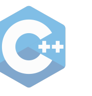
```rust
fn read_temperature() -> Result<i32, io::Error> {
  // ...
}

fn main() {
  let x = read_temperature();

  match x {
    Ok(value)  => println!("{value}"),
    Err(error) => println!("Error: {error}")
  }
}
```



```cpp
std::expected<int, std::string> read_temperature() {
  // ...
}

int main() {
  const auto x = read_temperature();

  if (x) {
    std::print("{}\n", *x);
  } else {
    std::print("Error: {}\n", x.error());
  }
}
```

Result types: Result<T, E>

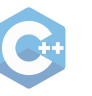
```rust
fn read_temperature() -> Result<i32, io::Error> {
  // ...
}

fn display_temperature() -> Result<(), io::Error> {
  let t = read_temperature()?;

  // ..do other stuff..

  println!("Temperature: {t}°C");
  Ok(())
}

fn main() -> Result<(), io::Error> {
  display_temperature()?;
  Ok(())
}
```



```cpp
std::expected<int, std::string> read_temperature() {
  // ...
}

std::expected<void, std::string> display_temperature() {
  const auto t = read_temperature();
  if (!t) {
    return std::unexpected{t.error()};
  }

  // ..do other stuff..

  std::print("Temperature: {}°C\n", *t);
  return {};
}

int main() {
  if (const auto result = display_temperature(); !result) {
    std::print(stderr, "Error: {}\n", result.error());
  }
}
```

Result types: `Result<T, E>`

```rust
fn read_temperature() -> Result<i32, io::Error> {
  // ...
}

fn display_temperature() -> Result<(), io::Error> {
  let t = read_temperature()?;

  // ..do other stuff..

  println!("Temperature: {t}°C");
  Ok(())
}

fn main() -> Result<(), io::Error> {
  display_temperature()?;
  Ok(())
}
```



```cpp
int read_temperature() {
  // ..may throw an exception..
}

void display_temperature() {
  const auto t = read_temperature();

  // ..do other stuff..

  std::print("Temperature: {}°C\n", t);
}

int main() {
  try {
    display_temperature();
  } catch (const char *error) {
    std::print(stderr, "Error: {}\n", error);
  }
}
```

Error handling

There are no exceptions in Rust. All errors are either:

- Ordinary errors: use `Result<T, E>`,
- Errors that *should never happen*: panic.

Error handling: Panic

An error is displayed, the stack unwinds and the panicking thread exits:

```rust
fn do_something_fallable() -> Result<(), io::Error> {
  // ...
}

fn main() {
  let _ = 10 / (env::args().len() - 1); // Panics due to divide-by-zero.

  do_something_fallable().expect("Error"); // Panics due to failed expected result.

  assert!(false); // Panics due to failed assertion.
}
```

```
thread 'main' panicked at 'attempt to divide by zero', /app/example.rs:9:11
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Error handling: Result<>

```rust
fn do_something_fallable() -> Result<(), io::Error> {
  // ...
}

fn main() {
  // 1) Using an if let declaration:
  if let Err(error) = do_something_fallable() {
    // ..error handling..
  }

  // 2) Using a match expression:
  match do_something_fallable() {
    Err(error) => {
      // ..error handling..
    },
    _ => {}
  }
}
```

```rust
fn do_something_fallable_deepest() -> Result<(), io::Error> {
  // ...
}

fn do_something_fallable_deeper() -> Result<(), io::Error> {
  do_something_fallable_deepest()?;

  // ...happy flow; do other stuff..
}

fn do_something_fallable() -> Result<(), io::Error> {
  do_something_fallable_deeper()?;

  // ...happy flow; do other stuff..
}

fn main() {
  if let Err(error) = do_something_fallable() {
    // ..error handling..
  }
}
```

Traits

Traits define shared behavior between types: *ad-hoc polymorphism*

```rust
struct Beast {
  id: u8,
  name: String
}

fn main() {
  let b = Beast{ id: 42, name: "Blaktor".to_string() };

  // Print out a message introducing 'b'..
}
```

```
..message..
```

Traits

Traits define shared behavior between types: *ad-hoc polymorphism*

```rust
trait Introducible {
  fn introduce(&self) -> String;
}

impl Introducible for Beast {
  fn introduce(&self) -> String {
    format!("Hi, I'm {} (#{}), nice to eat you! Grrrr!",
            self.name, self.id.to_string())
  }
}
```

```rust
struct Beast {
  id: u8,
  name: String
}

fn main() {
  let b = Beast{ id: 42, name: "Blaktor".to_string() };

  println!("{}", b.introduce());
}
```

```
Hi, I'm Blaktor (#42), nice to eat you! Grrrr!
```

Traits

Traits can be defined on any existing type

```rust
trait Introducible {
  fn introduce(&self) -> String;
}

impl Introducible for i32 {
  fn introduce(&self) -> String {
    format!("Howdy! I'm int (i32, value {}), can I help?",
            self.to_string())
  }
}
```

```rust
fn main() {
  let i : i32 = 17;

  println!("{}", i.introduce());
}
```

```
Howdy! I'm int (i32, value 17), can I help?
```

Traits using #[derive(...)]

Using the #[derive(...)] attribute

```rust
#[derive(Debug)]
struct Beast {
  id: u8,
  name: String
}
```

Debug is the trait name

```rust
fn main() {
  let b = Beast{ id: 42, name: "Blaktor".to_string() };
  let i : i32 = 17;

  println!("{b:?}");
  println!("{i:?}");
}
```

```
Beast { id: 42, name: "Blaktor"}
17
```

Traits: inherent `impl`

```rust
trait Introducible {
  fn introduce(&self) -> String;
}

impl Introducible for Beast {
  fn introduce(&self) -> String {
    format!("Hi, I'm {}, nice to eat you! Grrrr!", self.identifier())
  }
}

impl Introducible for i32 {
  fn introduce(&self) -> String {
    format!("Howdy! I'm int (i32, value {}), can I help?", self.to_string())
  }
}

// Inherent impl for Beast (adds a method to Beast only).
impl Beast {
  fn identifier(&self) -> String {
    format!("{} (#{})", self.name, self.id)
  }
}
```

```rust
#[derive(Debug)]
struct Beast {
  id: u8,
  name: String
}

 fn main() {
  let b = Beast{ id: 42,
                 name: "Blaktor".to_string() };

  println!("{b:?}");
  println!("{}", b.introduce());

  let i : i32 = 17;

  println!("{i:?}");
  println!("{}", i.introduce());
}
```

Generic programming

Rust supports type parameters and (some) *const generic* types

```rust
fn display_value<T: Display>(arg: &T) {
  println!("Value = '{arg}'");
}

fn compare_values<T: Eq>(arg1: T, arg2: T) -> bool {
  arg1 == arg2
}

fn display_const<const C: u32>() {
  println!("Const = '{C}'");
}
```

```rust
fn main() {
  let x = 42;
  display_value(&x);
  display_value(&"Headphone");

  display_value(&compare_values(true, false));
  display_value(&compare_values(17, 12 + 5));

  display_const::<42>();
}
```

```
Value = '42'
Value = 'Headphone'
Value = 'false'
Value = 'true'
Const = '42'
```

Generic programming

```rust
trait Dramatizible {
  fn dramatize(&self) -> String;
}

impl Dramatizible for DartsScore::<u8> {
  fn dramatize(&self) -> String {
    match &self.score {
      180    => format!("Onehundred aand eiiiiiighty!!"),
      points => format!("You've thrown {points} points!")
    }
  }
}
```

```rust
struct DartsScore<T> {
  score: T
}

fn main() {
  let s1 = DartsScore{ score: 180 };
  let s2 = DartsScore{ score: 179 };

  println!("{}", s1.dramatize());
  println!("{}", s2.dramatize());
}
```

```
Onehundred aand eiiiiiighty!!
You've thrown 179 points!
```

Traits and `const` generics

```rust
trait Dramatizible {
  fn dramatize(&self) -> String;
}

impl <const S: Sport> Dramatizible for Score::<u32, S> {
  fn dramatize(&self) -> String {
    match S {
      Sport::Curling => {
        format!("You've wiped up {} points!", &self.score)
      },
      Sport::Darts => {
        match &self.score {
          180    => format!("Onehundred aand eiiiiiighty!!"),
          points => format!("You've thrown {points} points!")
        }
      }
    }
  }
}
```

(requires incomplete `#[feature(adt_const_params)]`)

```rust
#[derive(PartialEq, Eq)]
enum Sport {
  Curling,
  Darts
}

struct Score<T, const S: Sport> {
  score: T
}

fn main() {
  let s1 = Score::<u32, {Sport::Darts}>{ score: 180 };
  let s2 = Score::<u32, {Sport::Darts}>{ score: 179 };
  let s3 = Score::<u32, {Sport::Curling}>{ score: 42 };

  println!("{}", s1.dramatize());
  println!("{}", s2.dramatize());
  println!("{}", s3.dramatize());
}
```

```
Onehundred aand eiiiiiighty!!
You've thrown 179 points!
You've wiped up 42 points!
```

Concurrent programming

```rust
use std::thread;

fn main() {
  let handle = thread::spawn(|| {
    println!("Called from another thread!");
  });

  handle.join().unwrap();
}
```

```
Called from another thread!
```

Explicitly propagate thread panic using `unwrap()`

```rust
fn count_vowels(s: &str) -> usize {
  Regex::new("[^aeiouAEIOU]").unwrap().replace_all(&s, "").len()
}

fn main() {
  let words: Vec<&str> = "One thread per word for this sentence!"
                             .split(' ').collect();

  let mut threads = vec![];
  for word in words {
    threads.push(thread::spawn(move || count_vowels(word)));
  }

  println!("There are {} vowels in the sentence", {
    let mut sum = 0;
    for handle in threads {
      sum += handle.join().unwrap(); // Propagate child thread panic.
    }
    sum
  });
}
```

```
There are 11 vowels in the sentence
```

Concurrent programming

- Sharing immutable data is guaranteed to be safe: **no data races**
- Marker traits `Send` and `Sync` are used to define/check shared access rules
- Facilities like *channels* are available to facilitate inter-thread data passing
- Higher-level constructs such as futures are available as well
- For shared mutable access there are mutexes/RW locks/atomics etc.

```rust
struct Data {
  value: Mutex<String>
}
```

```rust
let s = Data{ value: Mutex::new("Shared..".to_string()) };

{
  let mut guard = s.value.lock().unwrap();

  // 's.value' is locked within this scope..
}
```

Asynchronous programming

```rust
async fn request_id(ip: Ipv4Addr, port: u16)
                    -> std::io::Result<u32> {
  // ..request ID value from server..
}

async fn fetch_ids(servers: Vec<(Ipv4Addr, u16)>)
                   -> Vec<std::io::Result<u32>> {
  use async_std::task;

  let mut handles = vec![];
  for (ip, port) in servers {
    handles.push(task::spawn_local(request_id(ip, port)));
  }

  let mut results = vec![];
  for handle in handles {
    results.push(handle.await);
  }

  results
}
```

```rust
fn main() {
  let servers = vec![
    (Ipv4Addr::new(192, 168,  1,  2), 123),
    (Ipv4Addr::new( 10,   0,  1, 33), 456),
    (Ipv4Addr::new(172,  16, 16, 12), 789)
  ];

  let results = task::block_on(fetch_ids(servers));
  for result in results {
    match result {
      Ok(id)     => println!("ID: {id}"),
      Err(error) => println!("Error: {error}")
    }
  }
}
```

All requests are carried out in parallel, on the same thread.

Macros using `macro_rules!`

Rust supports (partially) hygienic macros for meta-progamming

```rust
macro_rules! html_page {
  ({ title : $t:tt,
     body  : $b:tt }) => {{
    {
      let mut result = "<html>\n".to_string();
      result += &format!("  <title>{}</title>\n", $t.to_string());
      result += &format!("  <body>{}</body>\n", $b.to_string());
      result += "</html>";
      result
    }
  }};
}
```

```rust
fn main() {
  let page = html_page!({
    title: "Adonis",
    body: "Look at my splendid abs!"
  });

  println!("{page}");
}
```

```
<html>
  <title>Adonis</title>
  <body>Look at my splendid abs!</body>
</html>
```

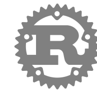
Procedural macros

Procedural macros allow creating **syntax extensions**

They can be seen as “compiler plugins” to modify the compilation AST 😎

Procedural macros

Example using clap, a command-line argument parser crate



```rust
/// Calculation trainer for additions/subtractions.
#[derive(Parser)]
#[clap(author, version, about, long_about = None)]
struct Args {
  /// Maximum calculation result value.
  #[clap(short, long, default_value_t = 25)]
  maximum: u32,

  /// Calculation mode.
  #[clap(arg_enum, default_value_t = Mode::Mix)]
  mode: Mode
}

#[derive(ArgEnum, Clone)]
enum Mode {
  Add,
  Sub,
  Mix
}
```

```rust
let args = Args::parse();
```

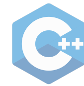
```
$ ./reken-trainer

reken-trainer 0.1.0
Calculation trainer for additions/subtractions

USAGE:
    reken-trainer [OPTIONS] [MODE]

ARGS:
    <MODE>    Calculation mode [default: mix]
                [possible values: add, sub, mix]

OPTIONS:
    -h, --help                 Print help information
    -m, --maximum <MAXIMUM>    Maximum calculation result value
                                 [default: 25]
    -V, --version              Print version information
```

clap parses help info from documentation using macros.

Unsafe Rust

Huh? I thought Rust was safe!?

Fast C++: a fast exponential



```cpp
static constexpr double fast_exp(double x) {
  union {
    double d;
    struct {
      int j, i;
    } n;
  } eco;

  eco.n.i = ((1048576 / M_LN2) * x) + 1072632447;

  return eco.d;
}

int main() {
  for (double x = -1.0; x <= 1.02; x += 0.2) {
    std::print("e^{:+.2f} = {:.5f}\n", x, fast_exp(x));
  }
}
```

```
e^-1.00 = 0.37483
e^-0.80 = 0.44696
e^-0.60 = 0.53820
e^-0.40 = 0.68247
e^-0.20 = 0.82674
e^-0.00 = 0.97101
e^+0.20 = 1.23055
e^+0.40 = 1.51909
e^+0.60 = 1.80763
e^+0.80 = 2.19234
e^+1.00 = 2.76942
```

Rust will forbid this implementation

Nicol N. Schraudolph: A fast, Compact Approximation of the Exponential Function, Neural Computation 11, 853–862 (1999)

Unsafe Rust: a fast exponential

```rust
#[derive(Clone, Copy)]
struct N {
  _j : i32,
  _i : i32
}

union Eco {
  d : f64,
  n : N
}

fn fast_exp(x: f64) -> f64 {
  let i = ((1048576_f64 / std::f64::consts::LN_2) * x)
        + 1072632447_f64;
  let eco = Eco{ n : N { _j : 0, _i : i as i32 } };
  eco.d
}
```

```rust
fn main() {
  for i in -5..=5 {
    let x = i as f64 * 0.2_f64;
    println!("e^{:+.2} = {:.5}", x, fast_exp(x));
  }
}
```

```
error[E0133]: access to union field is unsafe and requires
              unsafe function or block
  --> <source>:14:3
   |
14 |   eco.d
   |   ^^^^^ access to union field
   |
   = note: the field may not be properly initialized: using
           uninitialized data will cause undefined behavior

error: aborting due to previous error

For more information about this error, try
`rustc --explain E0133`.
```

Unsafe Rust: a fast exponential

```rust
#[derive(Clone, Copy)]
struct N {
  _j : i32,
  _i : i32
}

union Eco {
  d : f64,
  n : N
}

unsafe fn fast_exp(x: f64) -> f64 {
  let i = ((1048576_f64 / std::f64::consts::LN_2) * x)
        + 1072632447_f64;
  let eco = Eco{ n : N { _j : 0, _i : i as i32 } };
  eco.d
}
```

Sprinkling **unsafe** all over the place is unnecessary

```rust
fn main() {
  for i in -5..=5 {
    let x = i as f64 * 0.2_f64;

    unsafe {
      println!("e^{:+.2} = {:.5}", x, fast_exp(x));
    }
  }
}
```

```
e^-1.00 = 0.37483
e^-0.80 = 0.44696
e^-0.60 = 0.53820
e^-0.40 = 0.68247
e^-0.20 = 0.82674
e^+0.00 = 0.97101
e^+0.20 = 1.23055
e^+0.40 = 1.51909
e^+0.60 = 1.80763
e^+0.80 = 2.19234
e^+1.00 = 2.76942
```

Unsafe Rust: a fast exponential

```rust
#[derive(Clone, Copy)]
struct N {
  _j : i32,
  _i : i32
}

union Eco {
  d : f64,
  n : N
}

fn fast_exp(x: f64) -> f64 {
  debug_assert!((-1.0..=1.0).contains(self));
  unsafe {
    let i = ((1048576_f64 / std::f64::consts::LN_2) * x)
            + 1072632447_f64;
    let eco = Eco{ n : N { _j : 0, _i : i as i32 } };
    eco.d
  }
}
```

```rust
fn main() {
  for i in -5..=5 {
    let x = i as f64 * 0.2_f64;
    println!("e^{:+.2} = {:.5}", x, fast_exp(x));
  }
}
```

```
e^-1.00 = 0.37483
e^-0.80 = 0.44696
e^-0.60 = 0.53820
e^-0.40 = 0.68247
e^-0.20 = 0.82674
e^+0.00 = 0.97101
e^+0.20 = 1.23055
e^+0.40 = 1.51909
e^+0.60 = 1.80763
e^+0.80 = 2.19234
e^+1.00 = 2.76942
```

No more **unsafe** in the user code

Unsafe Rust: a fast exponential

```rust
#[derive(Clone, Copy)]
struct N { _j : i32, _i : i32 }

union Eco { d : f64, n : N }

trait FastExp {
  fn fast_exp(&self) -> f64;
}

impl FastExp for f64 {
  fn fast_exp(&self) -> f64 {
    debug_assert!((-1.0..=1.0).contains(self));
    unsafe {
      let i = ((1048576_f64 / std::f64::consts::LN_2) * self)
            + 1072632447_f64;
      let eco = Eco{ n : N { _j : 0, _i : i as i32 } };
      eco.d
    }
  }
}
```

```rust
fn main() {
  for i in -5..=5 {
    let x = i as f64 * 0.2_f64;
    println!("e^{:+.2} = {:.5}", x, x.fast_exp());
  }
}
```

```
e^-1.00 = 0.37483
e^-0.80 = 0.44696
e^-0.60 = 0.53820
e^-0.40 = 0.68247
e^-0.20 = 0.82674
e^+0.00 = 0.97101
e^+0.20 = 1.23055
e^+0.40 = 1.51909
e^+0.60 = 1.80763
e^+0.80 = 2.19234
e^+1.00 = 2.76942
```

Using a trait to directly extend on the f64 type

Unsafe Rust: a fast exponential

Test code:

```rust
/// Tests RMS error over the range -1.000..=1.000 in steps of 1/1000.
#[test]
fn test_rms_error() {
  let rmse = {
    let mut sum = 0.0;
    for x in -1000..=1000 {
      let xx = (x as f64) * 0.001_f64;
      sum += (xx.exp() - xx.fast_exp()).powf(2.0);
    }
    sum / 2001.0
  };

  println!("RMSE = {rmse}");

  assert!(rmse < 0.001);
}
```

```rust
#[test]
#[should_panic]
fn test_range_panic_low() {
  let _ = -1.1.fast_exp();
}

#[test]
#[should_panic]
fn test_range_panic_high() {
  let _ = 1.1.fast_exp();
}
```

```
$ cargo test
...
```

Unsafe Rust: a fast exponential

Benchmark code:

```rust
#[bench]
fn bench_std_exp(b: &mut Bencher) {
  b.iter(|| (-1000..=1000).fold(0_f64, |x, _| ((x as f64) * 0.001).exp()));
}

#[bench]
fn bench_fast_exp(b: &mut Bencher) {
  b.iter(|| (-1000..=1000).fold(0_f64, |x, _| ((x as f64) * 0.001).fast_exp()));
}
```

```
$ cargo bench

test lib::tests::bench_fast_exp ... bench:      17,824 ns/iter (+/- 395)
test lib::tests::bench_std_exp  ... bench:      33,411 ns/iter (+/- 264)
```

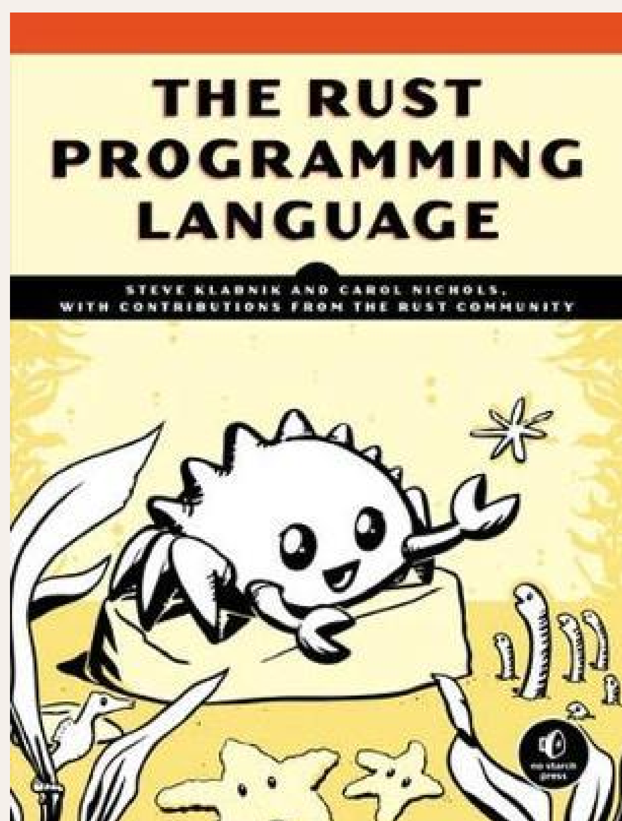
Foreign Function Interfaces (FFIs)

...

Moving on from here

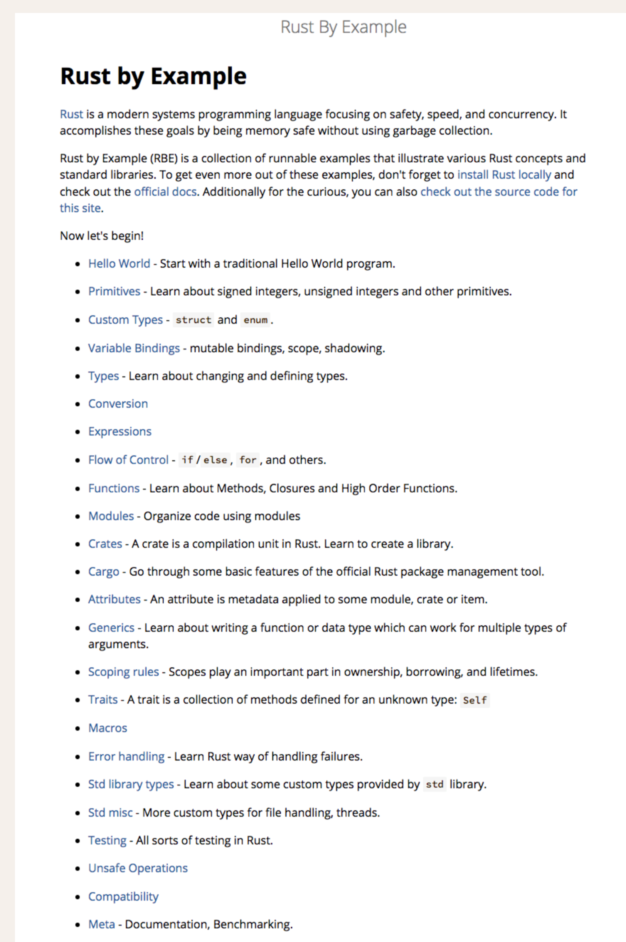
Learning more about Rust

The official book:



Freely available online

Rust by example:



Rust By Example

Rust by Example

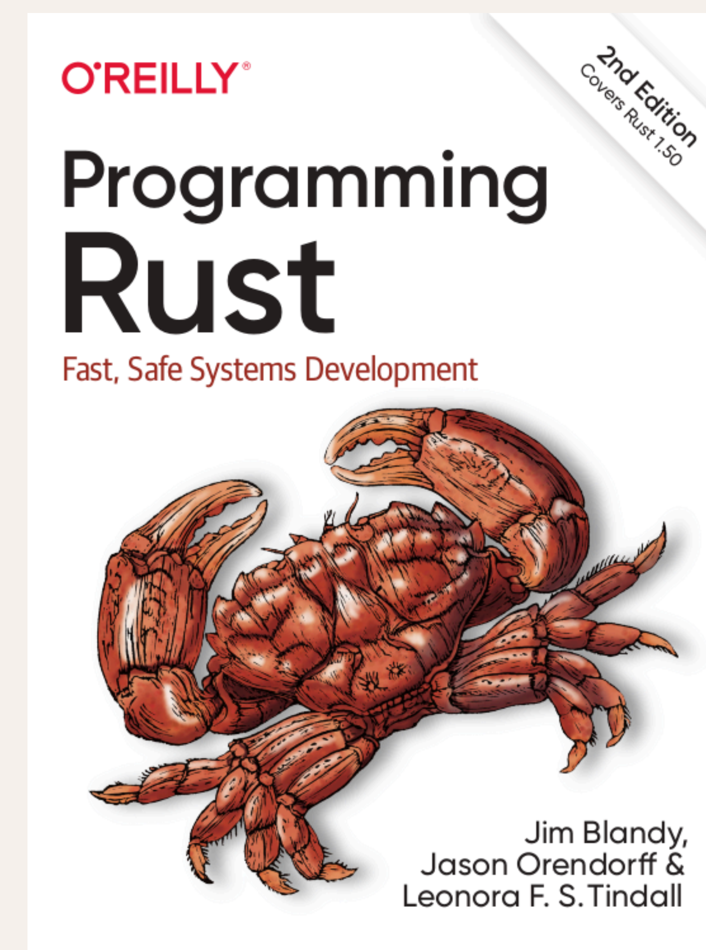
Rust is a modern systems programming language focusing on safety, speed, and concurrency. It accomplishes these goals by being memory safe without using garbage collection.

Rust by Example (RBE) is a collection of runnable examples that illustrate various Rust concepts and standard libraries. To get even more out of these examples, don't forget to install Rust locally and check out the official docs. Additionally for the curious, you can also check out the source code for this site.

Now let's begin!

- Hello World - Start with a traditional Hello World program.
- Primitives - Learn about signed integers, unsigned integers and other primitives.
- Custom Types - `struct` and `enum`.
- Variable Bindings - mutable bindings, scope, shadowing.
- Types - Learn about changing and defining types.
- Conversion
- Expressions
- Flow of Control - `if`/`else`, `for`, and others.
- Functions - Learn about Methods, Closures and High Order Functions.
- Modules - Organize code using modules
- Crates - A crate is a compilation unit in Rust. Learn to create a library.
- Cargo - Go through some basic features of the official Rust package management tool.
- Attributes - An attribute is metadata applied to some module, crate or item.
- Generics - Learn about writing a function or data type which can work for multiple types of arguments.
- Scoping rules - Scopes play an important part in ownership, borrowing, and lifetimes.
- Traits - A trait is a collection of methods defined for an unknown type: `Self`
- Macros
- Error handling - Learn Rust way of handling failures.
- Std library types - Learn about some custom types provided by `std` library.
- Std misc - More custom types for file handling, threads.
- Testing - All sorts of testing in Rust.
- Unsafe Operations
- Compatibility
- Meta - Documentation, Benchmarking.

Also a good book:

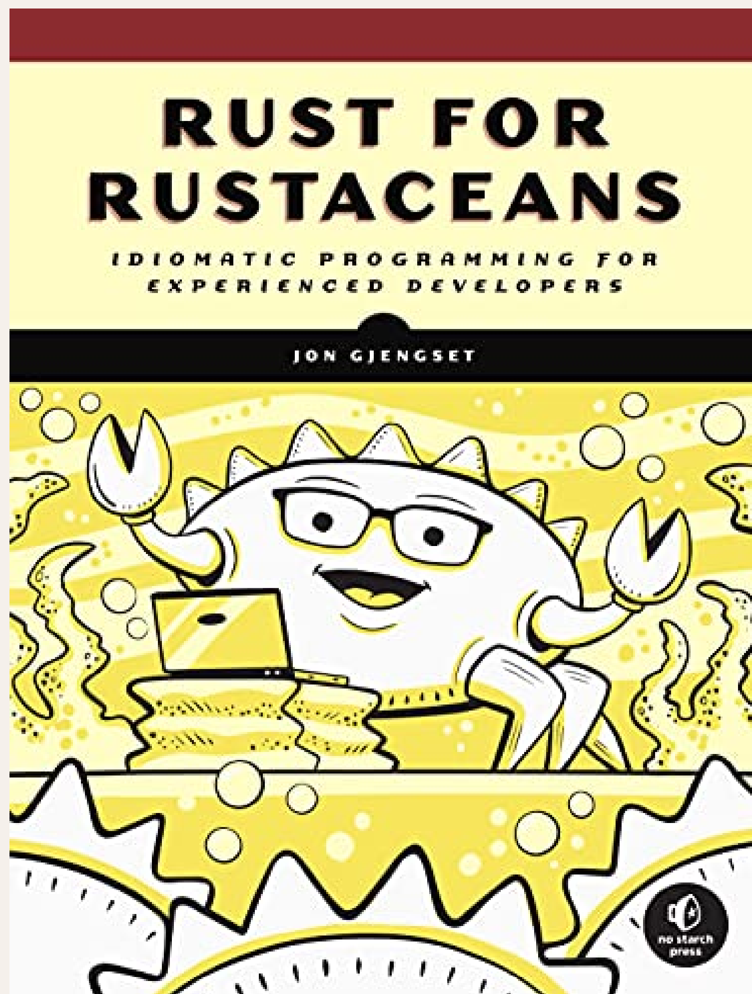


Also a good book:

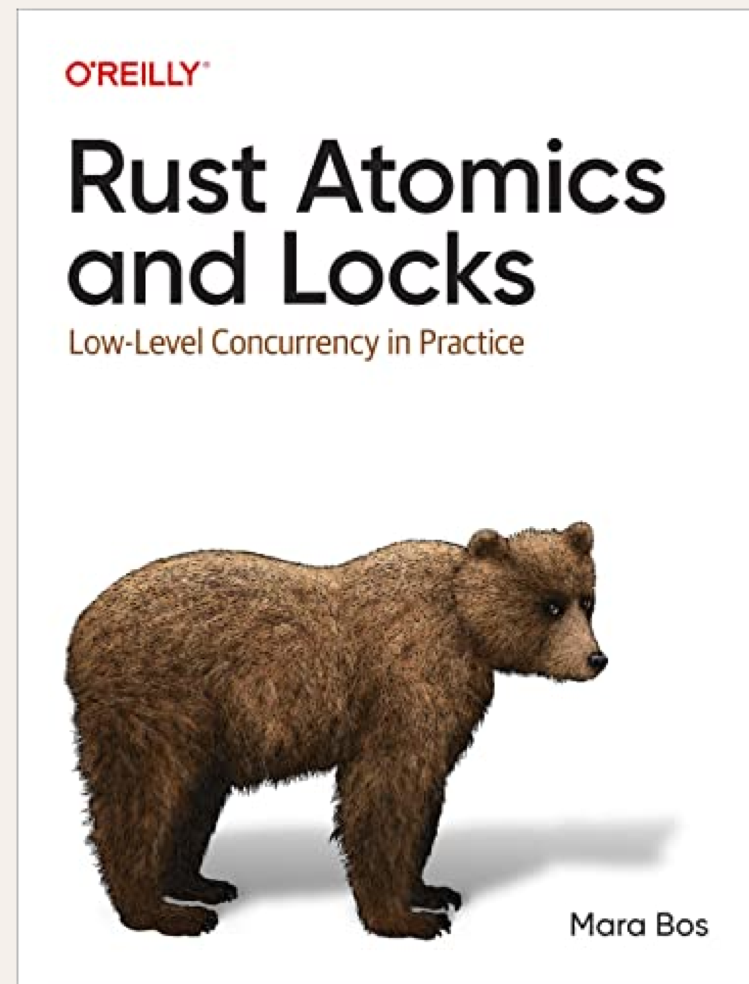
(source: Manning)

Learning more about Rust

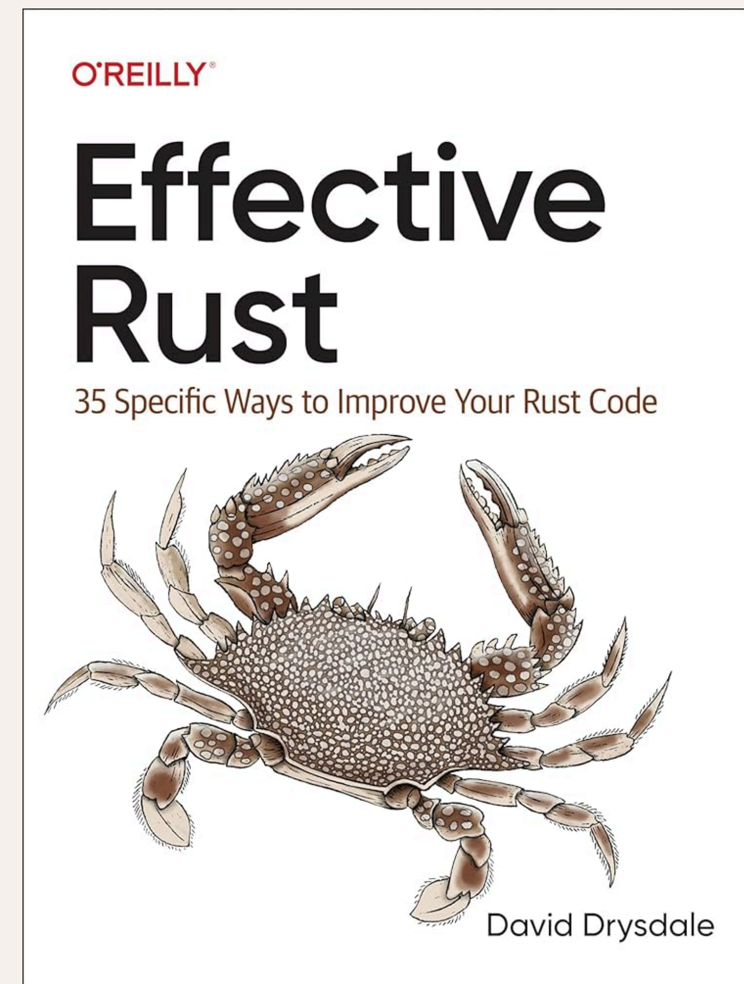
Also a good book:



Also a good book:



Also a good book:



Also a good book:

End

Thank you 😃

All emoji in this presentation are part of the Twemoji set, licensed under CC-BY 4.0.
All other images are mine, unless specified otherwise.

Extra slides

Extra slides

Miscellaneous

Food for thought (1)

What does it mean if a C/C++ program compiles?

Food for thought (2)

Is C/C++ undefined behavior worth the inconvenience?

Extra slides

Example code

Rust lifetime validation (1/3)

```rust
use std::thread;

fn contains(data: Vec<u8>, value: &u8) -> thread::JoinHandle<bool> {
  thread::spawn(|| {
    let result = data.contains(value);
    result
  })
}

fn main() {
  let data : Vec<u8> = vec!(1, 2, 3, 4, 5);
  assert_eq!(contains(data, &4).join().unwrap(), true);
}
```

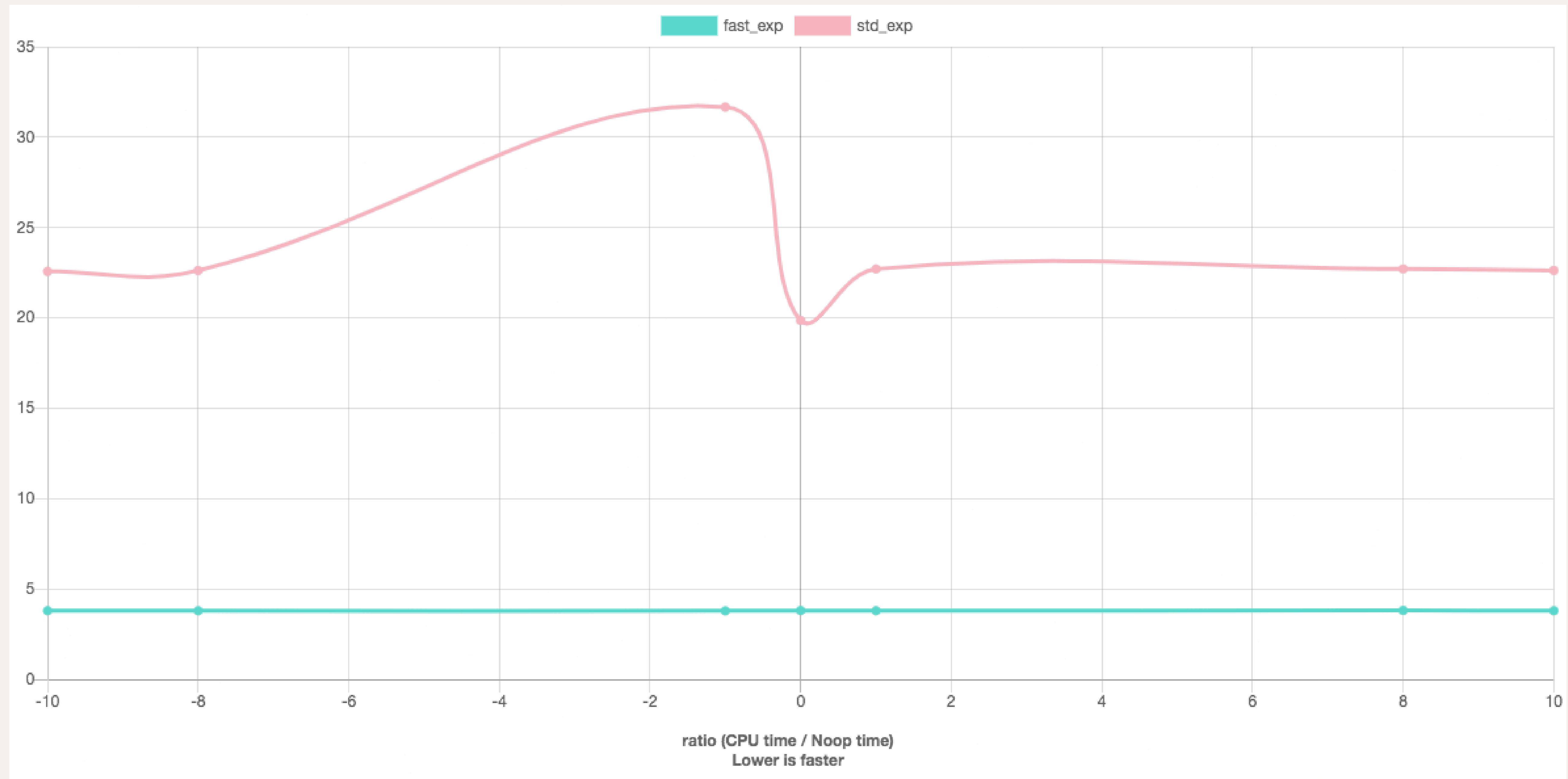
Rust lifetime validation (2/3)

```
error[E0759]: `value` has an anonymous lifetime `'_` but it needs to satisfy a `'static` lifetime requirement
 --> <source>:4:19
  |
3 |   fn contains(data: Vec<u8>, value: &u8) -> thread::JoinHandle<bool> {
  |                                     --- this data with an anonymous lifetime `'_`...
4 |     thread::spawn(|| {
  |  _________________^
5 | |     let result = data.contains(value);
6 | |     result
7 | |   })
  | |___^ ...is captured here...
  |
note: ...and is required to live as long as `'static` here
 --> <source>:4:5
  |
4 |   thread::spawn(|| {
  |   ^^^^^^^^^^^^^

error: aborting due to previous error

For more information about this error, try `rustc --explain E0759`.
```

Rust lifetime validation (3/3)

```rust
use std::thread;

fn contains(data: Vec<u8>, value: &'static u8) -> thread::JoinHandle<bool> {
  thread::spawn(move || {
    let result = data.contains(value);
    result
  })
}

fn main() {
  let data : Vec<u8> = vec!(1, 2, 3, 4, 5);
  assert_eq!(contains(data, &4).join().unwrap(), true);
}
```

Fast C++: a fast exponential



Extra slides

Undefined behavior

<digression>

Undefined behavior

A.K.A. **nasal demons** 🐉

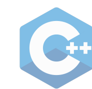
Undefined behavior (UB)

According to the C programming language standard:

> *Behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements.*

Undefined behavior (UB)

Example: signed integer overflow



```
#include <limits>

int main() {
  int x = std::numeric_limits<int>::max();
  return x + 42;
}
```

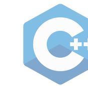
```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 41
```

No warnings/errors..

The UB-sanitizer finds it during runtime (`-fsanitize=undefined`)

Undefined behavior (UB)

Example: reading out-of-bounds



```cpp
int main() {
  int array[3] = {}; // All zeroes.
  int sum = 0;

  for (unsigned int i = 0; i <= (sizeof(array) / sizeof(int)); i++) {
    // Whoops:               ^^ We overrun the array index here!

    sum += array[i];
  }

  return sum;
}
```

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 16
```

No warnings/errors..

The UB-sanitizer finds it during runtime (`-fsanitize=undefined`)

Undefined behavior in five lines

Undefined behavior, or **UB** ..

- .. provides room for compiler optimization and language extension,
- .. doesn't have to be warned about by the compiler,
- .. can *potentially* be found by a UB sanitizer,
- .. is unconsciously depended upon by `{un,}`experienced developers,
- .. a tenacious source of security-related issues.

</digression>

Undefined behavior in C++

- Pointer-related UB
- Buffer overflows
- Integer Overflows
- Types, Cast and Const UB
- Function and template UB
- OOP-related UB
- Source file and preprocessing UB

Source: StackOverflow – CC BY-SA 3.0