Intro til Rust

Variabler

```
let x: i32 = 5;
```

Variabler

```rust
let x: i32 = 5;

let mut y: i32 = 5;
y += 5;
```

Variabler

```
let x: i32 = 5;

let mut y: i32 = 5;
y += 5;

let xref: &i32 = &x;

let yref: &mut i32 = &mut y;
*yref += 2;
```

Primitive typer - integer

```
let x: u8  = 0; //           0 - 127
let x: i8  = 0; //        -128 - 255
let x: u16 = 0; //           0 - 65535
let x: i16 = 0; //      -32768 - 32767
let x: u32 = 0; //           0 - 4294967295
let x: i32 = 0; // -2147483648 - 2147483647
```

Primitive typer - flyttall

```
let x: f32 = 17484.1819;
let x: f64 = 1847547191.18487491719;
```

Primitive typer - strenger

```rust
let static_text: &str = "Hello World";
```

Primitive typer - strenger

```rust
let static_text: &str = "Hello World";

let dynamic_text: String = "Hello World".to_string();

let text_ref: &String = &dynamic_text;
```

Primitive typer - strenger

```
let static_text: &str = "Hello World";

let dynamic_text: String = "Hello World".to_string();

let text_ref: &String = &dynamic_text;

let slice: &str = &dynamic_text[3..8];

assert_eq!(slice, "lo Wo");
```

Printing

```rust
println!("Print til skjerm");

let x = 2;
let y = 7;
let result = x + y;
println!("{} + {} = {}", x, y, result);

println!("{x} + {y} = {result}");
```

Enum

```
enum Action {
    Move { x: u32, y: u32 },
    Wait,
}

let action = Action::Move { x: 5, y: 1 };
```

Match statement

```
let action: Action = player.get_user_action();

match action {
    Move(x, y) => {
        board[x, y] = player;
        advance_turn();
    }
    Wait => advance_turn(),
}
```

Option

```
let x: Option<i32> = parse_int("5");

match x {
    Some(value) => println!("{value}"),
    None => println!("Not a number"),
}
```

Result

```
let x: Result<String, i32> = fetch_url("google.com");

match x {
    Ok(response) => println!("{response}"),
    Err(error_code) => {
        println!("Failed with error {error_code}")
    }
}
```

Lister

```
let strings: Vec<&str> = vec!["one", "two", "three"];
println!("{}", strings[1]);

let mut ints: Vec<i32> = Vec::new();
ints.push(4);
ints.push(9);
ints.push(2);
```

Funksjoner

```
fn add_two(x: i32) -> i32 {
    x + 2
}

fn is_even(x: &i32) -> bool {
    if x % 2 == 0 {
        true
    } else {
        false
    }
}

add_two(5);
is_even(&9);
```

Structs

```rust
struct Player {
    pub health: u8,
    pub position: (u8, u8),
    inventory: Vec<String>,
}

let health = 100;
let start_position = (0, 0);

let player = Player {
    health,
    position: start_position,
    inventory: vec![]
};
```

Methods

```rust
impl Player {
    fn new() -> Player {
        Player {
            health: 100,
            position: (0, 0),
            inventory: vec![],
        }
    }

    fn get_health(&self) -> u8 {
        self.health
    }

    fn take_damage(&mut self, damage: u8) {
        self.health -= damage
    }
}
```

Traits

```
trait Movable {
    fn move_to(&mut self, x: u8, y: u8);
}

impl Movable for Player {
    fn move_to(&mut self, x: u8, y: u8) {
        self.position = (x, y)
    }
}
```

Ownership

```
let player = Player::new();
let player2 = player;

println!("{}", player.health);
```

Ownership

```
fn mystery(player: Player) {
    ...
}

let player = Player::new();
mystery(player);

println!("{}", player.health);
```

Ownership

```rust
#[derive(Clone)]
struct Player {
    pub health: u8,
    pub position: (u8, u8),
    inventory: Vec<String>,
}
let player = Player::new();
let player2 = player.clone();

println!("{}", player.health);
```

Borrowing

```
let mut x = 5;
let xref = &x;
let xmutref = &mut x;

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

Borrowing

```
let mut x = 5;

if x == 7 {
    let xmutref = &mut x;
}

let xref = &x;
```

Generics

```
fn last<T>(list: &mut Vec<T>) -> Option<T> {
    list.pop()
}
```

Generics

```
use std::ops::Add;

fn plus<T: Add>(a: T, b: T) -> T::Output {
    a + b
}
```

Generics

```
use std::ops::Add;
use std::ops::Mul;

fn plus_multiply<A, B, C>(a: A, b: B, c: C) -> A::Output
    where A: Add<B::Output>,
          B: Mul<C>,
{
    a + (b * c)
}
```

Lifetimes

```
fn max(first: &i32, second: &i32) -> &i32 {
    if first >= second {
        first
    } else {
        second
    }
}
```

Lifetimes

```
let mut largest: &i32 = &0;

if something {
    let x = fetch();
    let y = fetch();
    largest = max(&x, &y);
}

do_something(largest);
```

Lifetimes

```rust
fn max<'a>(first: &'a i32, second: &'a i32) -> &'a i32 {
    if first >= second {
        first
    } else {
        second
    }
}
```

Iterators

```rust
let list = vec![1, 2, 3];

for x in &list {
    println!("{}", x);
}

list.iter().for_each(|x| {
    println!("{}", x);
});

list.into_iter().for_each(|x| {
    println!("{}", x);
});
```

Iterators - chaining

```
use std::collections::HashMap;

let map: HashMap<i32, i32> = vec![1, 2, 3, 4].into_iter()
    .filter(|x| {
        x % 2 == 0
    }).map(|x| {
        (x, x*x)
    }).collect();
```

Installasjon - Rustup

```
# https://rutup.rs

rustup toolchain install stable
```

Cargo

```
cargo new repo
cd repo

cargo add serde
cargo run
cargo test
```

Oppgaver

`https://github.com/kalkins/rust-intro.git`