Minnehåndtering i Rust

Stack, heap, borrowing, mutering og flytting 😱

Hvor lagres data?

- Heap vs Stack
  - Stack
    - Billig data access
    - Data må ha kjent størrelse
  - Heap
    - Data kan ha ukjent størrelse
    - Dyr data access

Hvor lagres data?

- Heap vs Stack
  - Stack
    - Billig data access
    - Data må ha kjent størrelse
  - Heap
    - Data kan ha ukjent størrelse
    - Dyr data access

```
let answer: i32 = 42;
let tuple_of_data: (i32, f64) = (123, 4.56);
let answer_to_another_question: String = String::from("Tacos");
let question: &str = "Life, the universe and everything";
```

Hvor lagres data?

- Heap vs Stack
  - Stack
    - Billig data access
    - Data må ha kjent størrelse
  - Heap
    - Data kan ha ukjent størrelse
    - Dyr data access

```
let answer: i32 = 42;
let tuple_of_data: (i32, f64) = (123, 4.56);
let answer_to_another_question: String = String::from("Tacos");
let question: &str = "Life, the universe and everything";
```

Hvor lagres data?

- Heap vs Stack
  - Stack
    - Billig data access
    - Data må ha kjent størrelse
  - Heap
    - Data kan ha ukjent størrelse
    - Dyr data access

```
let answer: i32 = 42;
let tuple_of_data: (i32, f64) = (123, 4.56);
let answer_to_another_question: String = String::from("Tacos");
let question: &str = "Life, the universe and everything";
```

Hvor lagres data?

- Heap vs Stack
  - Stack
    - Billig data access
    - Data må ha kjent størrelse
  - Heap
    - Data kan ha ukjent størrelse
    - Dyr data access

```
let answer: i32 = 42;
let tuple_of_data: (i32, f64) = (123, 4.56);
let answer_to_another_question: String = String::from("Tacos");
let question: &str = "Life, the universe and everything";
```

Hvor lagres data?

- Heap vs Stack
  - Stack
    - Billig data access
    - Data må ha kjent størrelse
  - Heap
    - Data kan ha ukjent størrelse
    - Dyr data access

```
let answer: i32 = 42;
let tuple_of_data: (i32, f64) = (123, 4.56);
let answer_to_another_question: String = String::from("Tacos");
let question: &str = "Life, the universe and everything";
```

Hvor lagres data?

- Heap vs Stack
  - Stack
    - Billig data access
    - Data må ha kjent størrelse
  - Heap
    - Data kan ha ukjent størrelse
    - Dyr data access

```
let answer: i32 = 42;
let tuple_of_data: (i32, f64) = (123, 4.56);
let answer_to_another_question: String = String::from("Tacos");
let question: &str = "Life, the universe and everything";
```

- Data må deallokeres
  - GC
  - Manuell minnehåndtering
  - **Eierskap**

Hvor lagres data?

- Heap vs Stack
  - Stack
    - Billig data access
    - Data må ha kjent størrelse
  - Heap
    - Data kan ha ukjent størrelse
    - Dyr data access

```
let answer: i32 = 42;
let tuple_of_data: (i32, f64) = (123, 4.56);
let answer_to_another_question: String = String::from("Tacos");
let question: &str = "Life, the universe and everything";
```

- Data må deallokeres
  - GC
  - Manuell minnehåndtering
  - **Eierskap**

Eierskap

Eierskap

En verdi har (maximum) én eier

```
let x = String::from("Hei");
```

Eierskap

En verdi har (maximum) én eier

```
let x = String::from("Hei");
```

Eierskap kan flyttes

```
let y = x
```

Eierskap

En verdi har (maximum) én eier

```
let x = String::from("Hei");
```

Eierskap kan flyttes

```
let y = x
```

Men da kan ikke den originale verdien brukes lenger

```
println!("{} = {}", x, y)
// Feiler med "borrow of moved value: `x`"
```

Eierskap

En verdi har (maximum) én eier

```
let x = String::from("Hei");
```

Eierskap kan flyttes

```
let y = x
```

Men da kan ikke den originale verdien brukes lenger

```
println!("{} = {}", x, y)
// Feiler med "borrow of moved value: `x`"
```

Flytting skjer også inn (og ut) av funksjoner:

```
fn print_message(msg: String) {
    println!("{}", msg);
}

fn main() {
    let x = String::from("Hei");
    print_message(x);
    let y = x; // Feiler her
    print_message(y);
}
```

Caveat: `Copy`-trait

```
let x = 23
let y = x
println!("{} = {}", x, y)
```

Caveat: `Copy`-trait

```
let x = 23
let y = x
println!("{} = {}", x, y)
```

- Dette fungerer fordi dataen bare er på stack

Caveat: `Copy`-trait

```
let x = 23
let y = x
println!("{} = {}", x, y)
```

- Dette fungerer fordi dataen bare er på stack
- `Copy` er en trait som tilsier at kopiering er billig, og gjør at data kopieres i stedet for å flytte eierskap

Løsningsforslag: `Clone`

- Lar oss kopiere data på heap

```
let x = String::from("Hei");
let y = x.clone();
println!("{} = {}", x, y)
```

- Løser åpenbart problemet
- Mye kopiering blir fort dyrt

Løsningsforslag: `Clone`

- Lar oss kopiere data på heap

```
let x = String::from("Hei");
let y = x.clone();
println!("{} = {}", x, y)
```

- Løser åpenbart problemet
- Mye kopiering blir fort dyrt

Løsningsforslag: `Clone`

- Lar oss kopiere data på heap

```
let x = String::from("Hei");
let y = x.clone();
println!("{} = {}", x, y)
```

- Løser åpenbart problemet
- Mye kopiering blir fort dyrt

Bedre løsning: Borrowing

Bedre løsning: Borrowing

- Rust lar oss låne verdier

Bedre løsning: Borrowing

- Rust lar oss låne verdier

```
let x = String::from("Hei");
let y = &x;
println!("{} = {}", x, y)
```

Bedre løsning: Borrowing

- Rust lar oss låne verdier

```
let x = String::from("Hei");
let y = &x;
println!("{} = {}", x, y)
```

Bedre løsning: Borrowing

- Rust lar oss låne verdier

```
let x = String::from("Hei");
let y = &x;
println!("{} = {}", x, y)
```

Bedre løsning: Borrowing

- Rust lar oss låne verdier

```
let x = String::from("Hei");
let y = &x;
println!("{} = {}", x, y)
```

- `y` er en referanse til `x` som har en referanse til data på heapen
  - Ingen kopiering av data på heap!
- Funksjoner tar typisk inn referanser, med mindre man faktisk skal konsumere dataen:

Bedre løsning: Borrowing

- Rust lar oss låne verdier

```
let x = String::from("Hei");
let y = &x;
println!("{} = {}", x, y)
```

- `y` er en referanse til `x` som har en referanse til data på heapen
  - Ingen kopiering av data på heap!
- Funksjoner tar typisk inn referanser, med mindre man faktisk skal konsumere dataen:

```
fn print_apple_type(apple: &Apple) {
    println!("{}", apple.kind);
}
fn eat_apple(apple: Apple) {
    println!("Eating apple of kind {}", apple.kind);
}
fn main() {
    let apple = Apple { kind: "Pink lady" };
    print_apple_type(&apple);
    let y = &apple;
    print_apple_type(y);
    eat_apple(apple);
}
```

Bedre løsning: Borrowing

- Rust lar oss låne verdier

```
let x = String::from("Hei");
let y = &x;
println!("{} = {}", x, y)
```

- `y` er en referanse til `x` som har en referanse til data på heapen
  - Ingen kopiering av data på heap!
- Funksjoner tar typisk inn referanser, med mindre man faktisk skal konsumere dataen:

```
fn print_apple_type(apple: &Apple) {
    println!("{}", apple.kind);
}
fn eat_apple(apple: Apple) {
    println!("Eating apple of kind {}", apple.kind);
}
fn main() {
    let apple = Apple { kind: "Pink lady" };
    print_apple_type(&apple);
    let y = &apple;
    print_apple_type(y);
    eat_apple(apple);
}
```

Bedre løsning: Borrowing

- Rust lar oss låne verdier

```
let x = String::from("Hei");
let y = &x;
println!("{} = {}", x, y)
```

- `y` er en referanse til `x` som har en referanse til data på heapen
  - Ingen kopiering av data på heap!
- Funksjoner tar typisk inn referanser, med mindre man faktisk skal konsumere dataen:

```
fn print_apple_type(apple: &Apple) {
    println!("{}", apple.kind);
}
fn eat_apple(apple: Apple) {
    println!("Eating apple of kind {}", apple.kind);
}
fn main() {
    let apple = Apple { kind: "Pink lady" };
    print_apple_type(&apple);
    let y = &apple;
    print_apple_type(y);
    eat_apple(apple);
}
```

Bedre løsning: Borrowing

- Rust lar oss låne verdier

```
let x = String::from("Hei");
let y = &x;
println!("{} = {}", x, y)
```

- `y` er en referanse til `x` som har en referanse til data på heapen
  - Ingen kopiering av data på heap!
- Funksjoner tar typisk inn referanser, med mindre man faktisk skal konsumere dataen:

```
fn print_apple_type(apple: &Apple) {
    println!("{}", apple.kind);
}
fn eat_apple(apple: Apple) {
    println!("Eating apple of kind {}", apple.kind);
}
fn main() {
    let apple = Apple { kind: "Pink lady" };
    print_apple_type(&apple);
    let y = &apple;
    print_apple_type(y);
    eat_apple(apple);
}
```

Bedre løsning: Borrowing

- Rust lar oss låne verdier

```
let x = String::from("Hei");
let y = &x;
println!("{} = {}", x, y)
```

- `y` er en referanse til `x` som har en referanse til data på heapen
  - Ingen kopiering av data på heap!
- Funksjoner tar typisk inn referanser, med mindre man faktisk skal konsumere dataen:

```
fn print_apple_type(apple: &Apple) {
    println!("{}", apple.kind);
}
fn eat_apple(apple: Apple) {
    println!("Eating apple of kind {}", apple.kind);
}
fn main() {
    let apple = Apple { kind: "Pink lady" };
    print_apple_type(&apple);
    let y = &apple;
    print_apple_type(y);
    eat_apple(apple);
}
```

Rettigheter på data - hvordan fungerer borrow checking

- Alle variabler har et sett med rettigheter på dataene sine - **R**ead, **W**rite og **O**wn
- Referanser endrer rettighetene til det de referer til
- Rust sjekker at man ikke bruker en variabel til noe den ikke har rettigheter til

```
let x = String::from("Jeg er på heapen");
let y = &x;
println!("{}", y)
let z = x;
```

Variabel	O	R	W
x			
*y			
z			

Rettigheter på data - hvordan fungerer borrow checking

- Alle variabler har et sett med rettigheter på dataene sine - **R**ead, **W**rite og **O**wn
- Referanser endrer rettighetene til det de referer til
- Rust sjekker at man ikke bruker en variabel til noe den ikke har rettigheter til

```
let x = String::from("Jeg er på heapen");
let y = &x;
println!("{}", y)
let z = x;
```

Variabel	O	R	W
x	✅	✅	
*y			
z			

Rettigheter på data - hvordan fungerer borrow checking

- Alle variabler har et sett med rettigheter på dataene sine - **R**ead, **W**rite og **O**wn
- Referanser endrer rettighetene til det de referer til
- Rust sjekker at man ikke bruker en variabel til noe den ikke har rettigheter til

```
let x = String::from("Jeg er på heapen");
let y = &x;
println!("{}", y)
let z = x;
```

Variabel	O	R	W
x	❌	✅	
*y		✅	
z			

Rettigheter på data - hvordan fungerer borrow checking

- Alle variabler har et sett med rettigheter på dataene sine - **R**ead, **W**rite og **O**wn
- Referanser endrer rettighetene til det de referer til
- Rust sjekker at man ikke bruker en variabel til noe den ikke har rettigheter til

```
let x = String::from("Jeg er på heapen");
let y = &x;
println!("{}", y)
let z = x;
```

Variabel	O	R	W
x	✅	✅	
*y		❌	
z			

Rettigheter på data - hvordan fungerer borrow checking

- Alle variabler har et sett med rettigheter på dataene sine - **R**ead, **W**rite og **O**wn
- Referanser endrer rettighetene til det de referer til
- Rust sjekker at man ikke bruker en variabel til noe den ikke har rettigheter til

```
let x = String::from("Jeg er på heapen");
let y = &x;
println!("{}", y)
let z = x;
```

Variabel	O	R	W
x	❌	❌	
*y			
z	✅	✅	

Rettigheter på data - hvordan fungerer borrow checking

- Alle variabler har et sett med rettigheter på dataene sine - **R**ead, **W**rite og **O**wn
- Referanser endrer rettighetene til det de referer til
- Rust sjekker at man ikke bruker en variabel til noe den ikke har rettigheter til

```
let x = String::from("Jeg er på heapen");
let y = &x;
println!("{}", y)
let z = x;
```

Variabel	O	R	W
x			
*y			
z	✅	✅	

Rettigheter på data - hvordan fungerer borrow checking

- Alle variabler har et sett med rettigheter på dataene sine - **R**ead, **W**rite og **O**wn
- Referanser endrer rettighetene til det de referer til
- Rust sjekker at man ikke bruker en variabel til noe den ikke har rettigheter til

```
let x = String::from("Jeg er på heapen");
let y = &x;
println!("{}", y)
let z = x;
```

Variabel	O	R	W
x			
*y			
z	✅	✅	

Rettigheter på data - hvordan fungerer borrow checking

- Alle variabler har et sett med rettigheter på dataene sine - **R**ead, **W**rite og **O**wn
- Referanser endrer rettighetene til det de referer til
- Rust sjekker at man ikke bruker en variabel til noe den ikke har rettigheter til

```
let x = String::from("Jeg er på heapen");
let y = &x;
println!("{}", y)
let z = x;
```

Variabel	O	R	W
x			
*y			
z	✅	✅	

Rettigheter på data - hvordan fungerer borrow checking

- Alle variabler har et sett med rettigheter på dataene sine - **R**ead, **W**rite og **O**wn
- Referanser endrer rettighetene til det de referer til
- Rust sjekker at man ikke bruker en variabel til noe den ikke har rettigheter til

```
let x = String::from("Jeg er på heapen");
let y = &x;
println!("{}", y)
let z = x;
```

Variabel	O	R	W
x			
*y			
z	✅	✅	

Rettigheter på data - hvordan fungerer borrow checking

- Alle variabler har et sett med rettigheter på dataene sine - **R**ead, **W**rite og **O**wn
- Referanser endrer rettighetene til det de referer til
- Rust sjekker at man ikke bruker en variabel til noe den ikke har rettigheter til

```
let x = String::from("Jeg er på heapen");
let y = &x;
println!("{}", y)
let z = x;
```

Variabel	O	R	W
x			
*y			
z	✅	✅	

Rettigheter på data - hvordan fungerer borrow checking

- Alle variabler har et sett med rettigheter på dataene sine - **R**ead, **W**rite og **O**wn
- Referanser endrer rettighetene til det de referer til
- Rust sjekker at man ikke bruker en variabel til noe den ikke har rettigheter til

```
let x = String::from("Jeg er på heapen");
let y = &x;
println!("{}", y)
let z = x;
```

Variabel	O	R	W
x			
*y			
z	✅	✅	

Rettigheter på data - hvordan fungerer borrow checking

- Alle variabler har et sett med rettigheter på dataene sine - **R**ead, **W**rite og **O**wn
- Referanser endrer rettighetene til det de referer til
- Rust sjekker at man ikke bruker en variabel til noe den ikke har rettigheter til

```
let x = String::from("Jeg er på heapen");
let y = &x;
println!("{}", y)
let z = x;
```

Variabel	O	R	W
x			
*y			
z	✅	✅	

Rettigheter på data - hvordan fungerer borrow checking

- Alle variabler har et sett med rettigheter på dataene sine - **R**ead, **W**rite og **O**wn
- Referanser endrer rettighetene til det de referer til
- Rust sjekker at man ikke bruker en variabel til noe den ikke har rettigheter til

```
let x = String::from("Jeg er på heapen");
let y = &x;
println!("{}", y)
let z = x;
```

Variabel	O	R	W
x			
*y			
z	✅	✅	

Rettigheter på data - hvordan fungerer borrow checking

- Alle variabler har et sett med rettigheter på dataene sine - **R**ead, **W**rite og **O**wn
- Referanser endrer rettighetene til det de referer til
- Rust sjekker at man ikke bruker en variabel til noe den ikke har rettigheter til

```
let x = String::from("Jeg er på heapen");
let y = &x;
println!("{}", y)
let z = x;
```

Variabel	O	R	W
x			
*y			
z	✅	✅	

Rettigheter på data - hvordan fungerer borrow checking

- Alle variabler har et sett med rettigheter på dataene sine - **R**ead, **W**rite og **O**wn
- Referanser endrer rettighetene til det de referer til
- Rust sjekker at man ikke bruker en variabel til noe den ikke har rettigheter til

```
let x = String::from("Jeg er på heapen");
let y = &x;
println!("{}", y)
let z = x;
```

Variabel	O	R	W
x			
*y			
z	✅	✅	

Rettigheter på data - hvordan fungerer borrow checking

- Alle variabler har et sett med rettigheter på dataene sine - **R**ead, **W**rite og **O**wn
- Referanser endrer rettighetene til det de referer til
- Rust sjekker at man ikke bruker en variabel til noe den ikke har rettigheter til

```
let x = String::from("Jeg er på heapen");
let y = &x;
println!("{}", y)
let z = x;
```

Variabel	O	R	W
x			
*y			
z	✅	✅	

Hvorfor mister man eierskap når noen leser?

Hvorfor mister man eierskap når noen leser?

```
let x = String::from("Hei");
let y = &x;
drop(x);
println!("{}", y);
```

Hvorfor mister man eierskap når noen leser?

```
let x = String::from("Hei");
let y = &x;
drop(x);
println!("{}", y);
```

- Data må leve minst like lenge som alle referanser til seg
- Dette spiller også inn på lifetimes:

Hvorfor mister man eierskap når noen leser?

```
let x = String::from("Hei");
let y = &x;
drop(x);
println!("{}", y);
```

- Data må leve minst like lenge som alle referanser til seg
- Dette spiller også inn på lifetimes:

```
fn max_of_boxed<'a>(x: &'a Box<i32>, y: &'a Box<i32>) -> &'a Box<i32> {
    max(x, y)
}
```

Muterbarhet og borrow checking

- Mange lesere, én skriver

```
let mut v = vec![1, 2, 3];
let num = &mut v[2];

*num += 1;

println!("Tredje element er {}", *num);
println!("Vektoren er nå {:?}", v);
```

Variabel	O	R	W
v			
num			
*num			

Muterbarhet og borrow checking

- Mange lesere, én skriver

```
let mut v = vec![1, 2, 3];
let num = &mut v[2];

*num += 1;

println!("Tredje element er {}", *num);
println!("Vektoren er nå {:?}", v);
```

Variabel	O	R	W
v	✅	✅	✅
num			
*num			

Muterbarhet og borrow checking

- Mange lesere, én skriver

```
let mut v = vec![1, 2, 3];
let num = &mut v[2];

*num += 1;

println!("Tredje element er {}", *num);
println!("Vektoren er nå {:?}", v);
```

Variabel	O	R	W
v	❌	❌	❌
num	✅	✅	
*num		✅	✅

Muterbarhet og borrow checking

- Mange lesere, én skriver

```
let mut v = vec![1, 2, 3];
let num = &mut v[2];

*num += 1;

println!("Tredje element er {}", *num);
println!("Vektoren er nå {:?}", v);
```

Variabel	O	R	W
v			
num	✅	✅	
*num		✅	✅

Muterbarhet og borrow checking

- Mange lesere, én skriver

```
let mut v = vec![1, 2, 3];
let num = &mut v[2];

*num += 1;

println!("Tredje element er {}", *num);
println!("Vektoren er nå {:?}", v);
```

Variabel	O	R	W
v	✅	✅	✅
num	❌	❌	
*num		❌	❌

Muterbarhet og borrow checking

- Mange lesere, én skriver

```
let mut v = vec![1, 2, 3];
let num = &mut v[2];

*num += 1;

println!("Tredje element er {}", *num);
println!("Vektoren er nå {:?}", v);
```

Variabel	O	R	W
v	✅	✅	✅
num			
*num			

Muterbarhet og borrow checking

- Mange lesere, én skriver

```
let mut v = vec![1, 2, 3];
let num = &mut v[2];

*num += 1;

println!("Tredje element er {}", *num);
println!("Vektoren er nå {:?}", v);
```

Variabel	O	R	W
v	✅	✅	✅
num			
*num			

Muterbarhet og borrow checking

- Mange lesere, én skriver

```
let mut v = vec![1, 2, 3];
let num = &mut v[2];

*num += 1;

println!("Tredje element er {}", *num);
println!("Vektoren er nå {:?}", v);
```

Variabel	O	R	W
v	✅	✅	✅
num			
*num			

Muterbarhet og borrow checking

- Mange lesere, én skriver

```
let mut v = vec![1, 2, 3];
let num = &mut v[2];

*num += 1;

println!("Tredje element er {}", *num);
println!("Vektoren er nå {:?}", v);
```

Variabel	O	R	W
v	✅	✅	✅
num			
*num			

Muterbarhet og borrow checking

- Mange lesere, én skriver

```
let mut v = vec![1, 2, 3];
let num = &mut v[2];

*num += 1;

println!("Tredje element er {}", *num);
println!("Vektoren er nå {:?}", v);
```

Variabel	O	R	W
v	✅	✅	✅
num			
*num			

Muterbarhet og borrow checking

- Mange lesere, én skriver

```
let mut v = vec![1, 2, 3];
let num = &mut v[2];

*num += 1;

println!("Tredje element er {}", *num);
println!("Vektoren er nå {:?}", v);
```

Variabel	O	R	W
v	✅	✅	✅
num			
*num			

Muterbarhet og borrow checking

- Mange lesere, én skriver

```
let mut v = vec![1, 2, 3];
let num = &mut v[2];

*num += 1;

println!("Tredje element er {}", *num);
println!("Vektoren er nå {:?}", v);
```

Variabel	O	R	W
v	✅	✅	✅
num			
*num			

Muterbarhet og borrow checking

- Mange lesere, én skriver

```
let mut v = vec![1, 2, 3];
let num = &mut v[2];

*num += 1;

println!("Tredje element er {}", *num);
println!("Vektoren er nå {:?}", v);
```

Variabel	O	R	W
v	✅	✅	✅
num			
*num			

Muterbarhet og borrow checking

- Mange lesere, én skriver

```
let mut v = vec![1, 2, 3];
let num = &mut v[2];

*num += 1;

println!("Tredje element er {}", *num);
println!("Vektoren er nå {:?}", v);
```

Variabel	O	R	W
v	✅	✅	✅
num			
*num			

Muterbarhet og borrow checking

- Mange lesere, én skriver

```
let mut v = vec![1, 2, 3];
let num = &mut v[2];

*num += 1;

println!("Tredje element er {}", *num);
println!("Vektoren er nå {:?}", v);
```

Variabel	O	R	W
v	✅	✅	✅
num			
*num			

Muterbarhet og borrow checking

- Mange lesere, én skriver

```
let mut v = vec![1, 2, 3];
let num = &mut v[2];

*num += 1;

println!("Tredje element er {}", *num);
println!("Vektoren er nå {:?}", v);
```

Variabel	O	R	W
v	✅	✅	✅
num			
*num			

Muterbarhet og borrow checking

- Mange lesere, én skriver

```
let mut v = vec![1, 2, 3];
let num = &mut v[2];

*num += 1;

println!("Tredje element er {}", *num);
println!("Vektoren er nå {:?}", v);
```

Variabel	O	R	W
v	✅	✅	✅
num			
*num			

Muterbarhet og borrow checking

- Mange lesere, én skriver

```
let mut v = vec![1, 2, 3];
let num = &mut v[2];

*num += 1;

println!("Tredje element er {}", *num);
println!("Vektoren er nå {:?}", v);
```

Variabel	O	R	W
v	✅	✅	✅
num			
*num			

Muterbarhet og borrow checking

- Mange lesere, én skriver

```
let mut v = vec![1, 2, 3];
let num = &mut v[2];

*num += 1;

println!("Tredje element er {}", *num);
println!("Vektoren er nå {:?}", v);
```

Variabel	O	R	W
v	✅	✅	✅
num			
*num			

Muterbarhet og borrow checking

- Mange lesere, én skriver

```
let mut v = vec![1, 2, 3];
let num = &mut v[2];

*num += 1;

println!("Tredje element er {}", *num);
println!("Vektoren er nå {:?}", v);
```

Variabel	O	R	W
v	✅	✅	✅
num			
*num			

Muterbarhet og borrow checking

- Mange lesere, én skriver

```
let mut v = vec![1, 2, 3];
let num = &mut v[2];

*num += 1;

println!("Tredje element er {}", *num);
println!("Vektoren er nå {:?}", v);
```

Variabel	O	R	W
v	✅	✅	✅
num			
*num			

Muterbarhet og borrow checking

- Mange lesere, én skriver

```
let mut v = vec![1, 2, 3];
let num = &mut v[2];

*num += 1;

println!("Tredje element er {}", *num);
println!("Vektoren er nå {:?}", v);
```

Variabel	O	R	W
v	✅	✅	✅
num			
*num			

Muterbarhet og borrow checking

- Mange lesere, én skriver

```
let mut v = vec![1, 2, 3];
let num = &mut v[2];

*num += 1;

println!("Tredje element er {}", *num);
println!("Vektoren er nå {:?}", v);
```

Variabel	O	R	W
v	✅	✅	✅
num			
*num			

Muterbarhet og borrow checking

- Mange lesere, én skriver

```
let mut v = vec![1, 2, 3];
let num = &mut v[2];

*num += 1;

println!("Tredje element er {}", *num);
println!("Vektoren er nå {:?}", v);
```

Variabel	O	R	W
v	✅	✅	✅
num			
*num			

Demo