Scope or Lifetime?

same, same but different

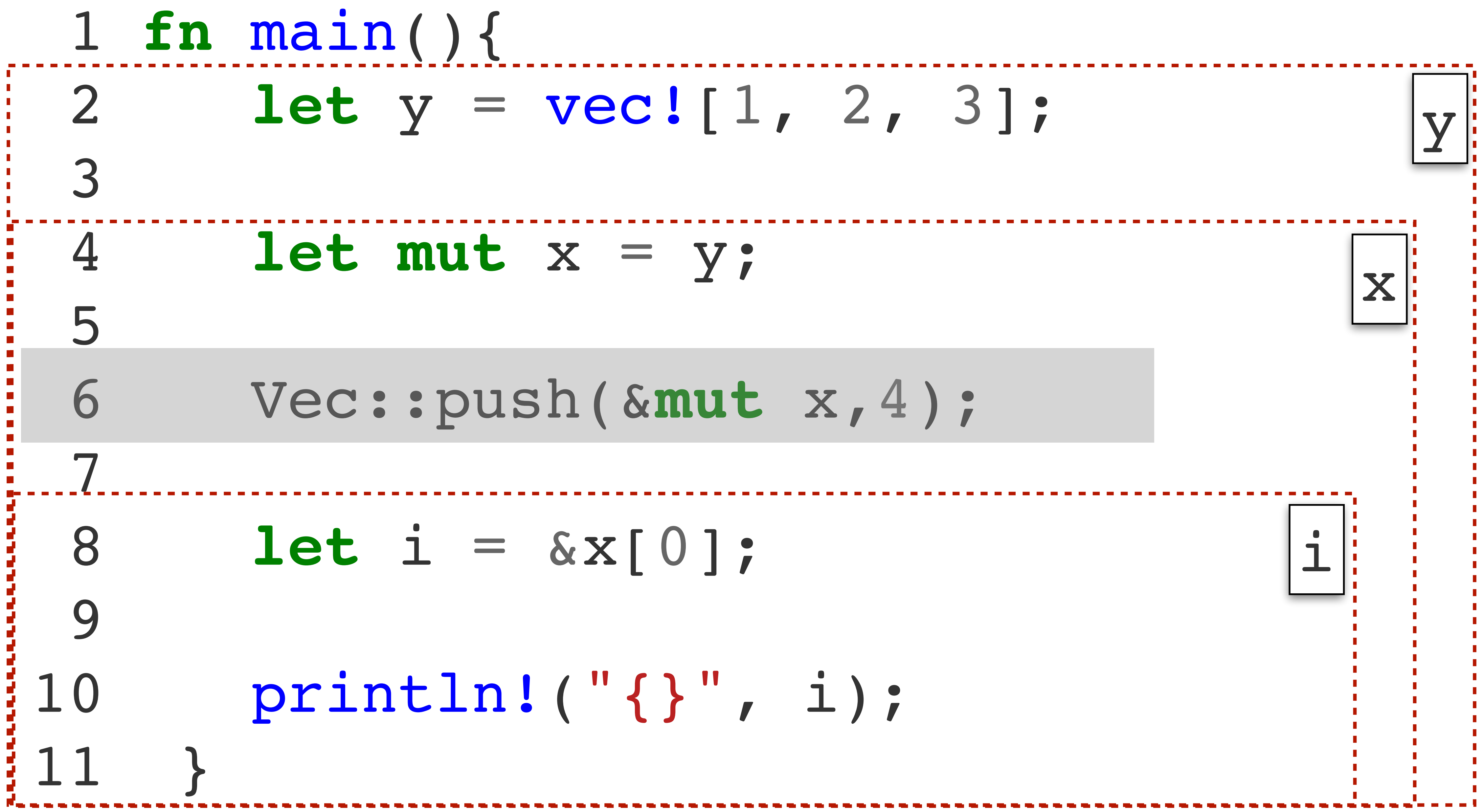
Scope

```
fn main(){
    let y = vec![1, 2, 3];

    let mut x = y;

    x.push(4);

    let i = &x[0];

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

A **scope** indicates the code block where a **variable is valid**:

- starts where the variable is first introduced,
- ends at the corresponding closing “ } ”.

Scope

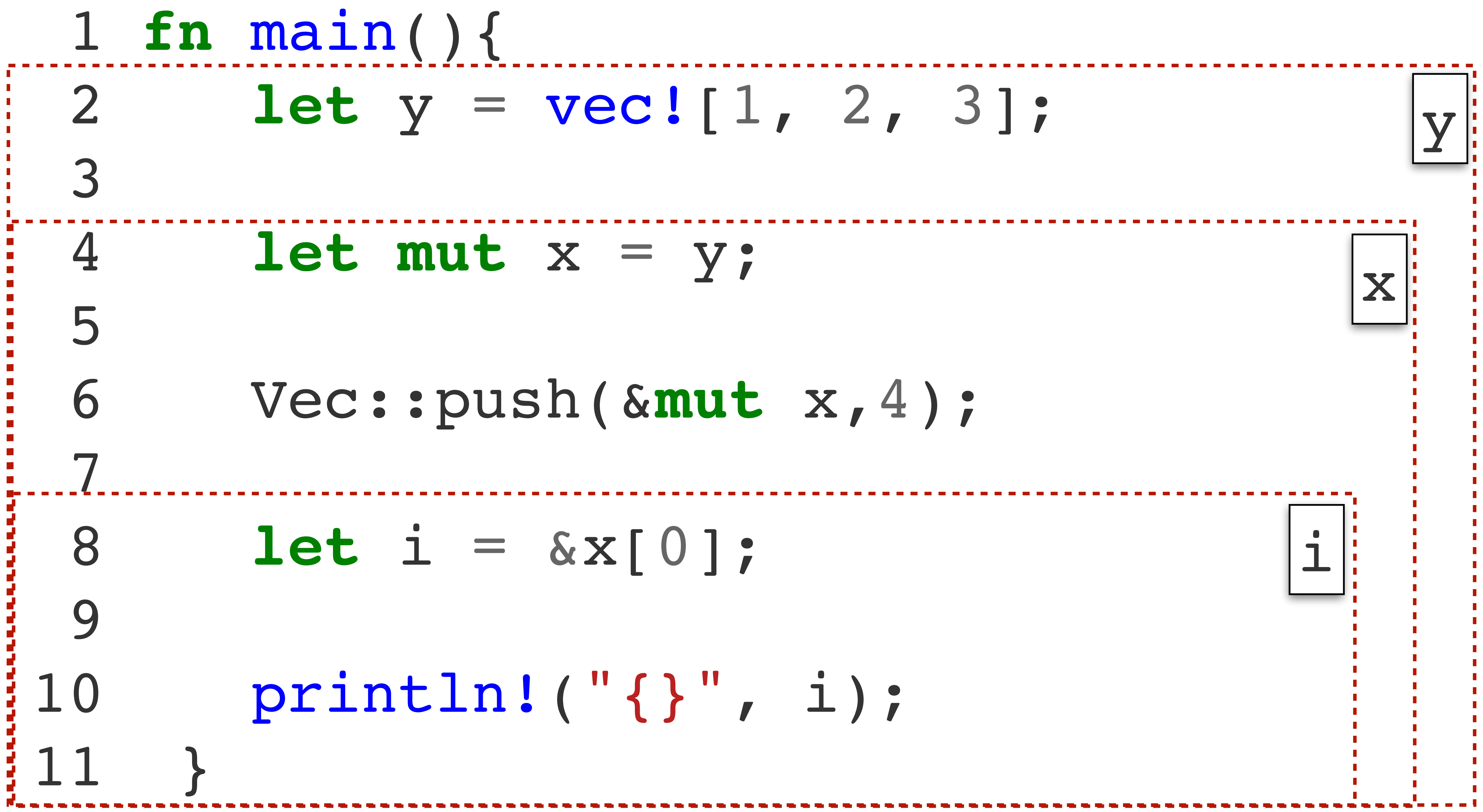


```
fn main(){
    let y = vec![1, 2, 3];

    let mut x = y;

    Vec::push(&mut x,4);

    let i = &x[0];

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

A **scope** indicates the code block where a **variable is valid**:

- starts where the variable is first introduced,
- ends at the corresponding closing “ } ”.

Scope

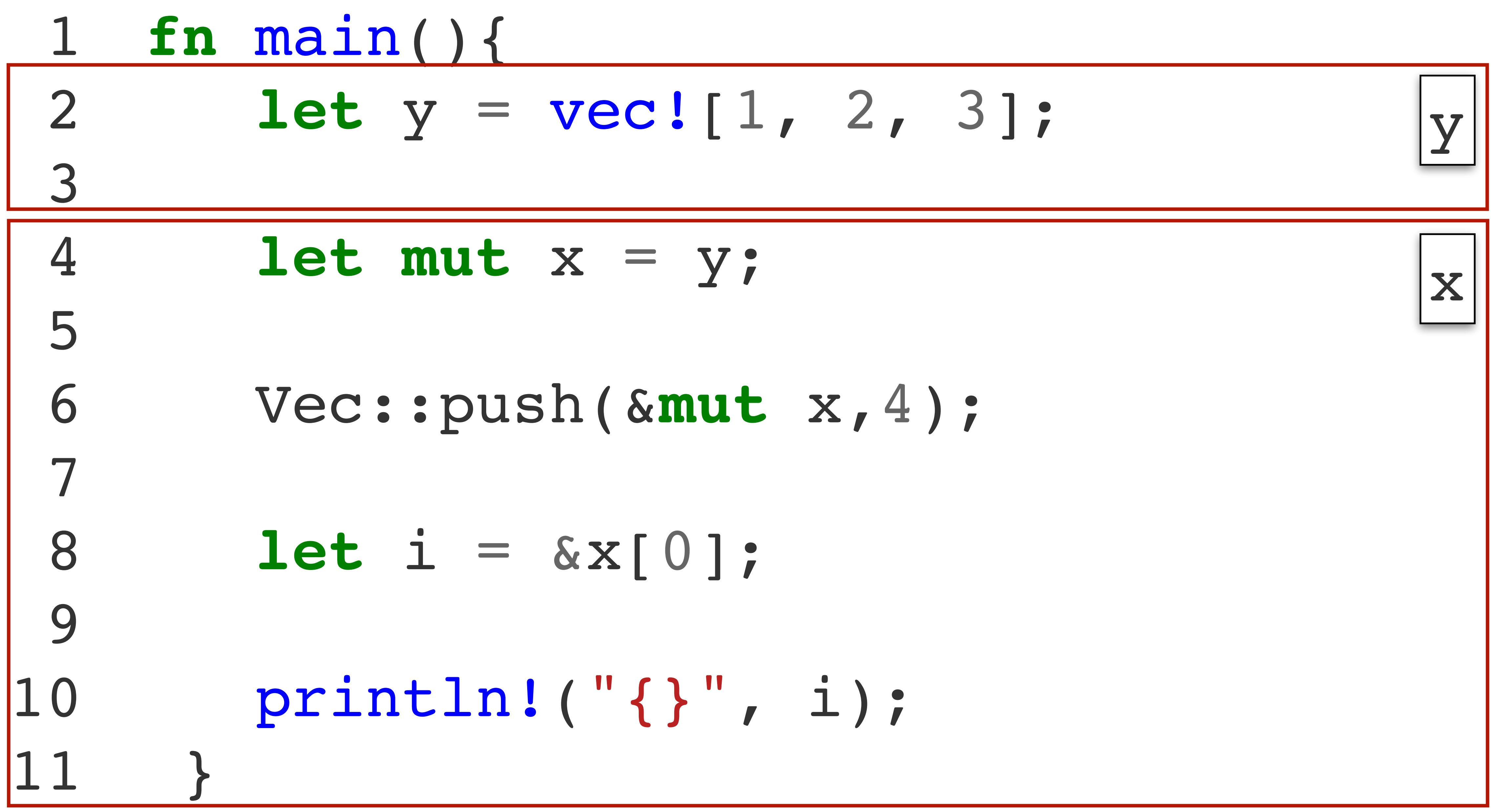


```
fn main(){
    let y = vec![1, 2, 3];

    let mut x = y;

    Vec::push(&mut x,4);

    let i = &x[0];

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

A **scope** indicates the code block where a **variable is valid**:

- starts where the variable is first introduced,
- ends at the corresponding closing “ } ”.

Liveness



```
fn main(){
    let y = vec![1, 2, 3];

    let mut x = y;

    Vec::push(&mut x,4);

    let i = &x[0];

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

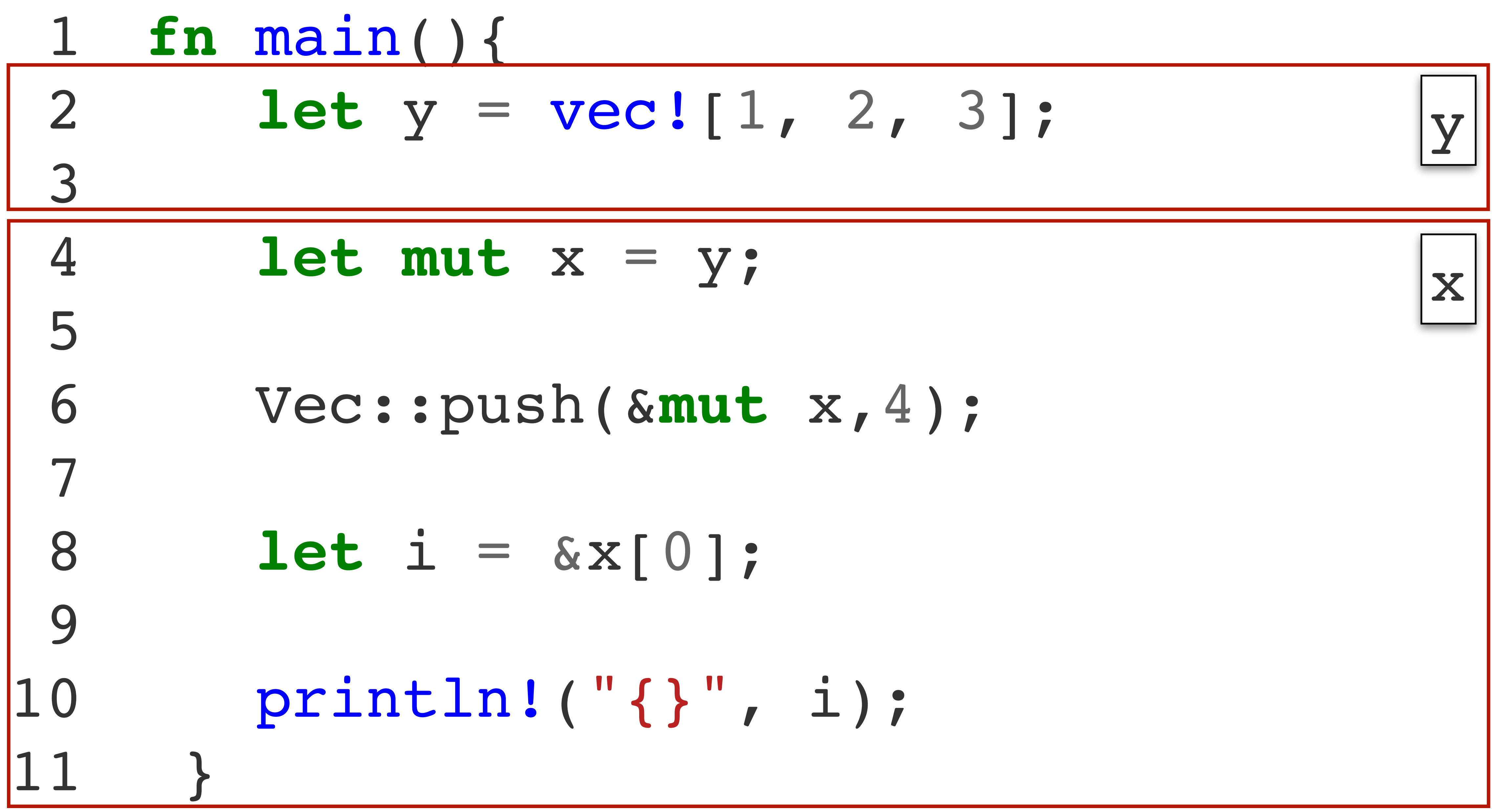
A **variable is live (owner is valid)** from initialisation until:

- its value is moved, or
- it goes out of scope (and is dropped).

Liveness

Safety Principle 1

Every value has a variable that's called its owner. There can only be one owner at a time.

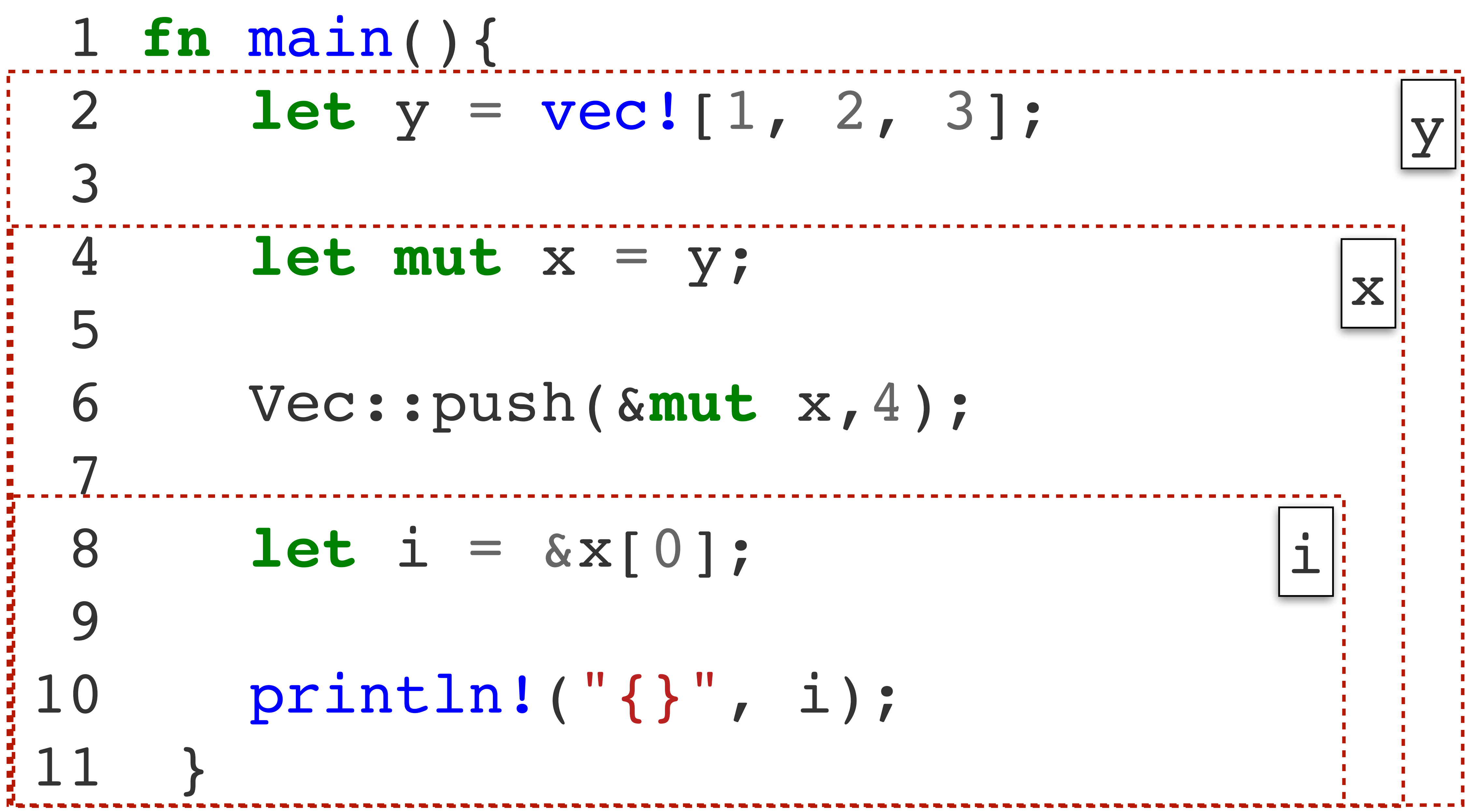


```
fn main(){
    let y = vec![1, 2, 3];

    let mut x = y;

    Vec::push(&mut x,4);

    let i = &x[0];

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

A **variable is live (owner is valid)** from initialisation until:

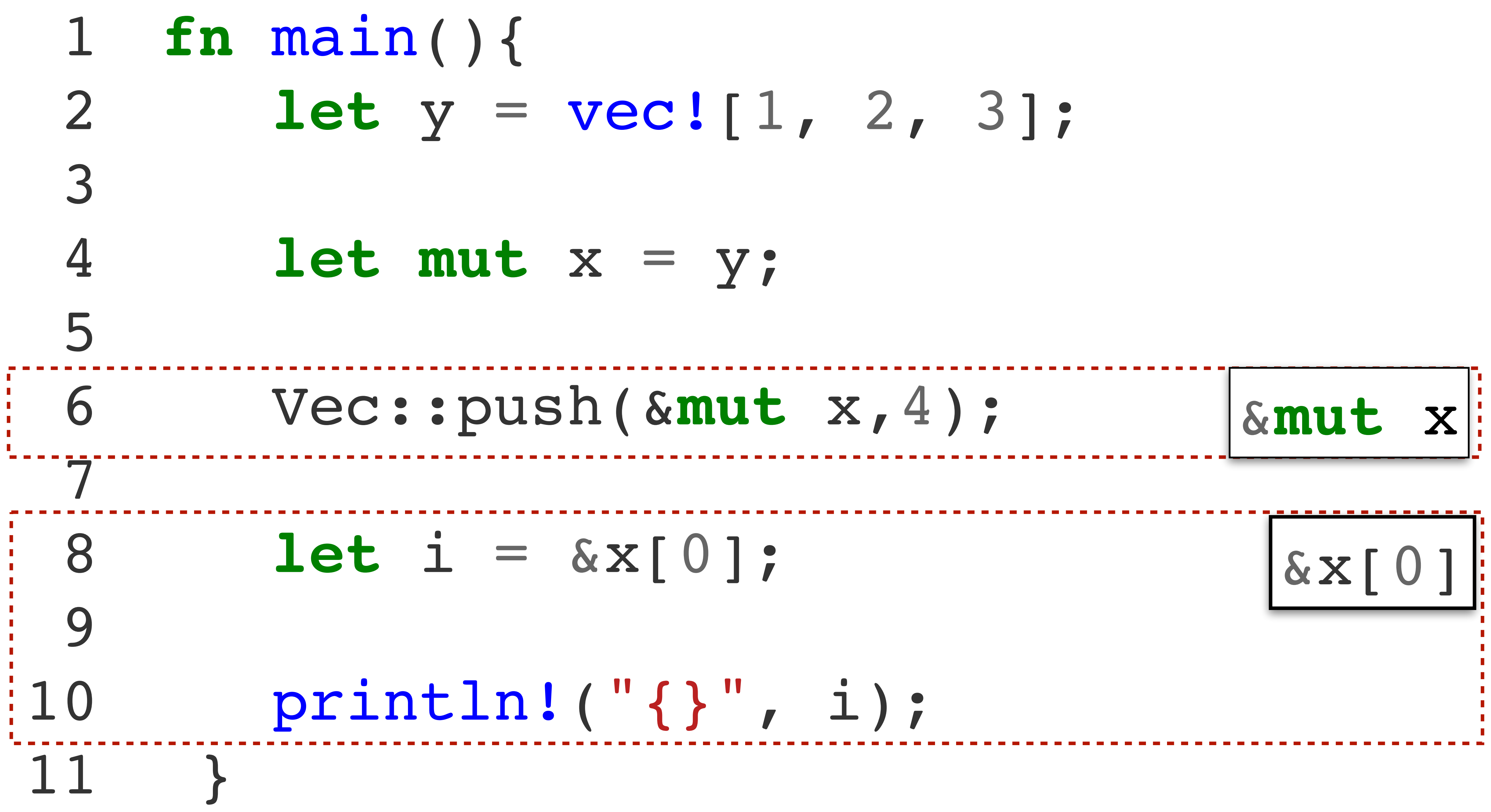
- its value is moved, or
- it goes out of scope (and is dropped).

Scope



```
fn main(){
    let y = vec![1, 2, 3];

    let mut x = y;

    Vec::push(&mut x,4);

    let i = &x[0];

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

Non-Lexical Lifetime



```
fn main(){
    let y = vec![1, 2, 3];

    let mut x = y;

    Vec::push(&mut x,4);

    let i = &x[0];

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

A **scope** indicates the code block where a **variable is valid**:

- starts where the variable is first introduced,
- ends at the corresponding closing “ } ”.

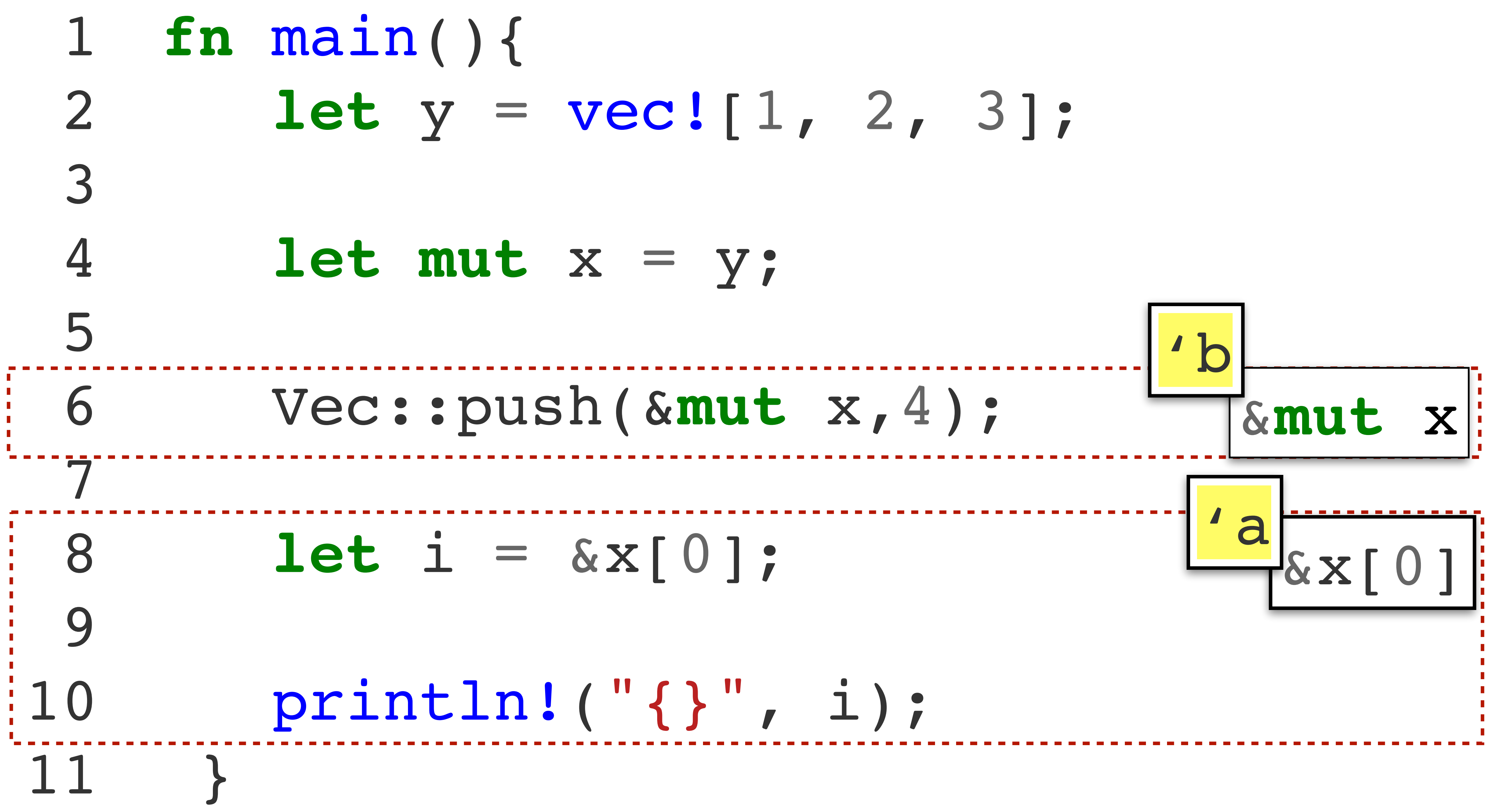
A **lifetime** indicates the code block where a **borrow is valid**:

- starts where the reference is created,
- ends where the reference is last used/needed.

Non-Lexical Lifetime

Safety Principle 3

A mutable borrow can only be created if its lender has no other borrows living at that time.



```
fn main(){
    let y = vec![1, 2, 3];

    let mut x = y;

    Vec::push(&mut x,4);

    let i = &x[0];

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

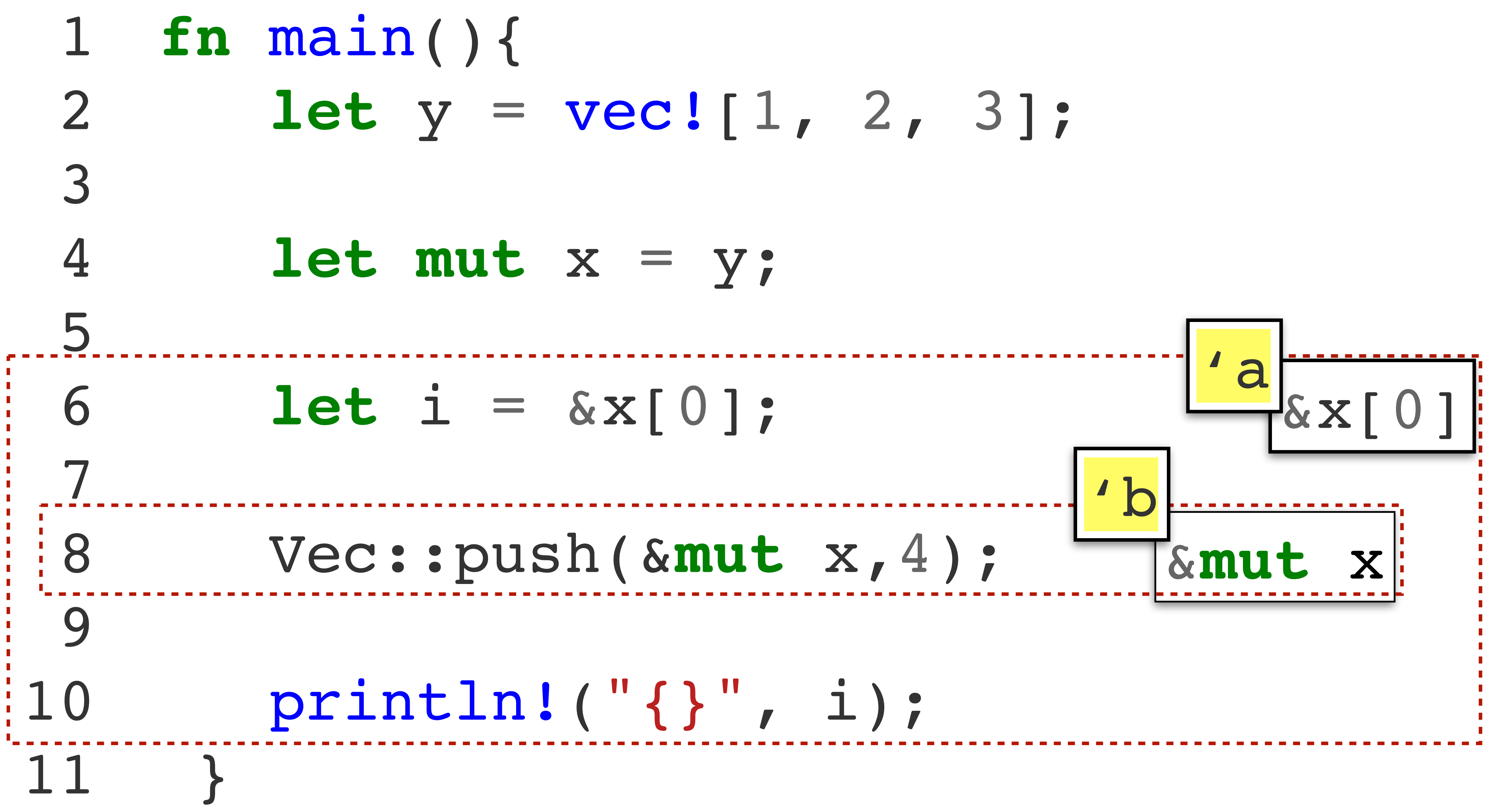
A **lifetime** indicates the code block where a **borrow is valid**:

- starts where the reference is created,
- ends where the reference is last used/needed.

Non-Lexical Lifetime

Safety Principle 3

A mutable borrow can only be created if its lender has no other borrows living at that time.



```
fn main(){
    let y = vec![1, 2, 3];

    let mut x = y;

    let i = &x[0];

    Vec::push(&mut x,4);

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

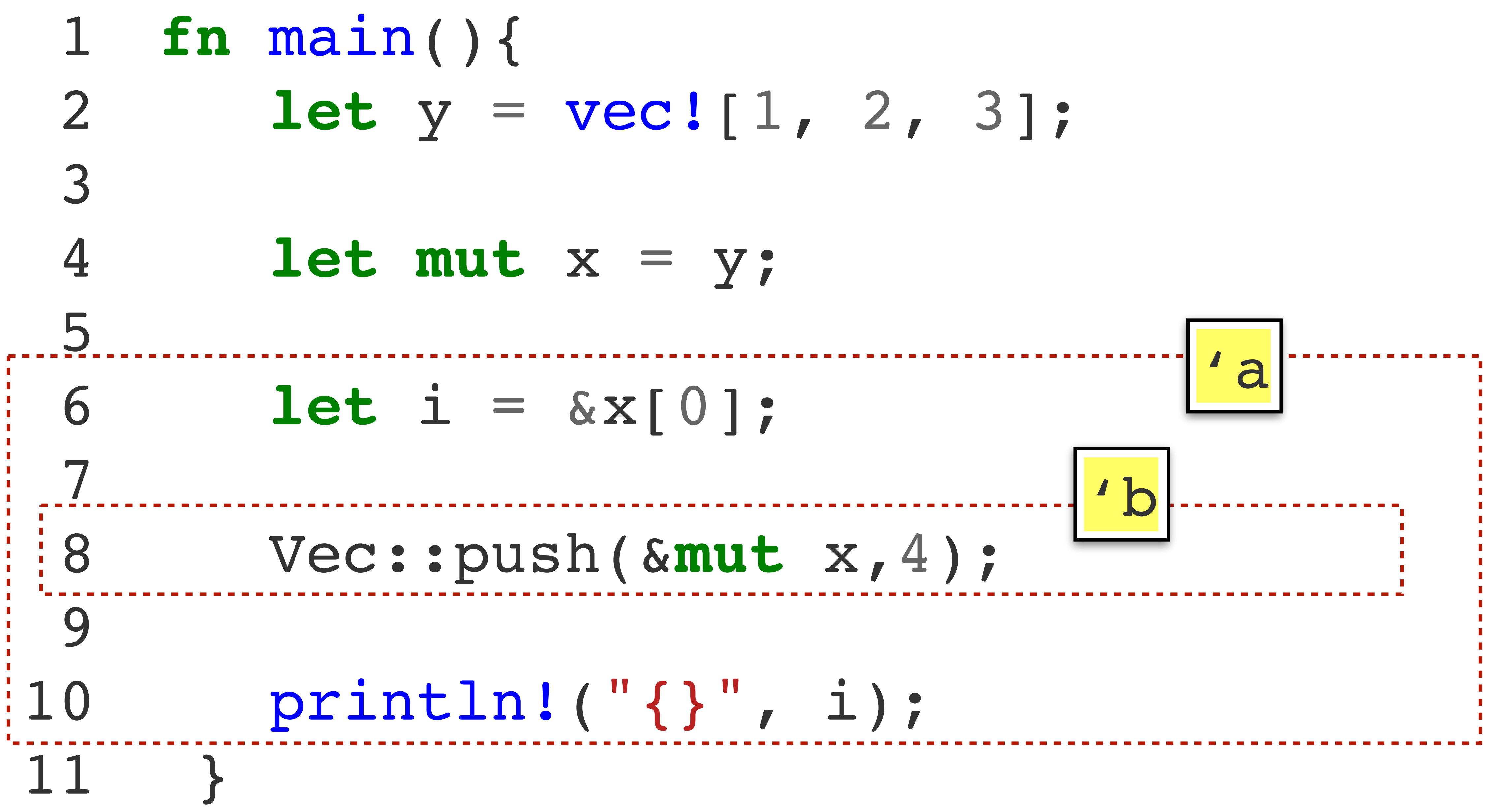
A **lifetime** indicates the code block where a **borrow is valid**:

- starts where the reference is created,
- ends where the reference is last used/needed.

Non-Lexical Lifetime

Safety Principle 3

A mutable borrow can only be created if its lender has no other borrows living at that time.



```
fn main(){
    let y = vec![1, 2, 3];

    let mut x = y;

    let i = &x[0];

    Vec::push(&mut x,4);

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

A **lifetime** indicates the code block where a **borrow is valid**:

- starts where the reference is created,
- ends where the reference is last used/needed.

Safety Principle 4

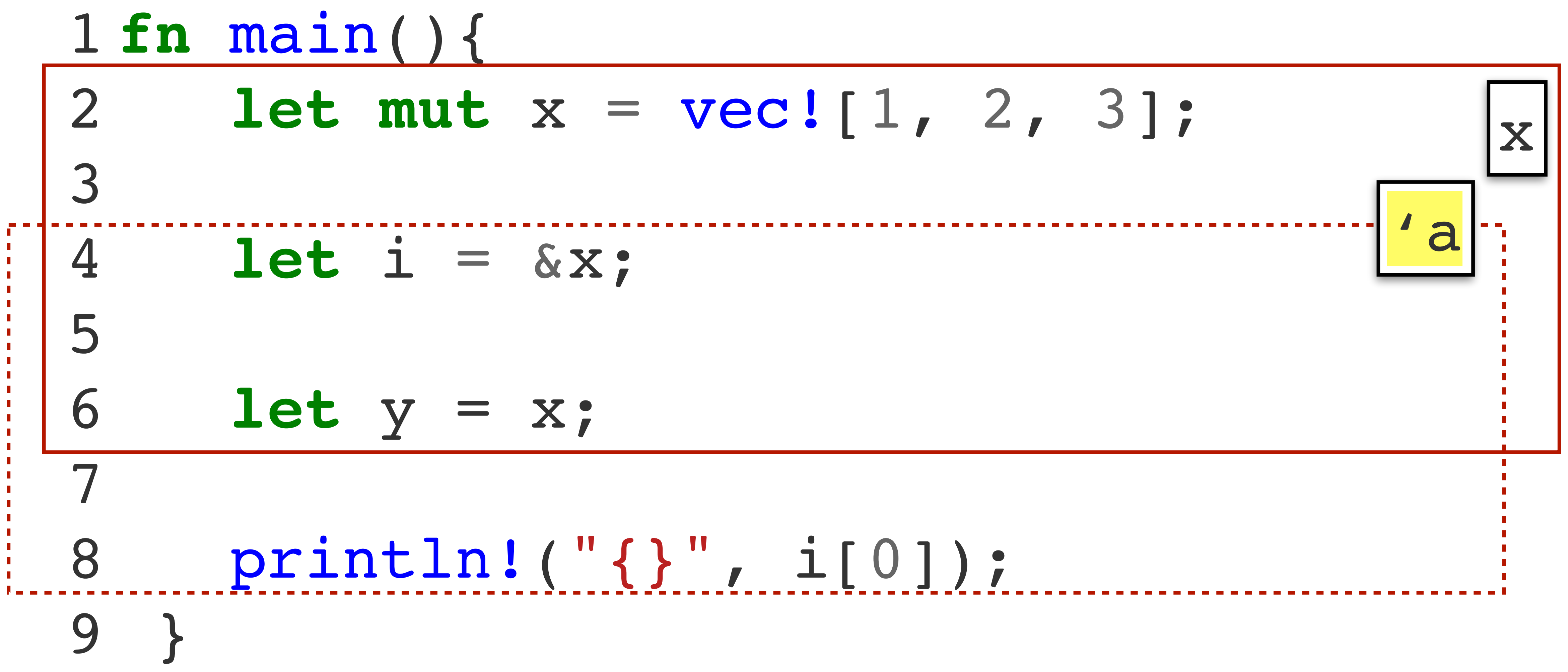
The lender cannot be modified as long as one of its (shared) borrowers still lives.

```rust
fn main(){
    let mut x = vec![1, 2, 3];

    let i = &x;

    x = vec![3,4];

    println!("{}", i[0]);
 }
```

'a

Safety Principle 2

The lender needs to outlive all of its (alive) references. 🚫



```
fn main(){
    let mut x = vec![1, 2, 3];

    let i = &x;

    let y = x;

    println!("{}", i[0]);
 }
```

Reborrowing

Safety Principle 1

Every value has a variable that’s called its owner.
There can only be one owner at a time.

Safety Principle 2

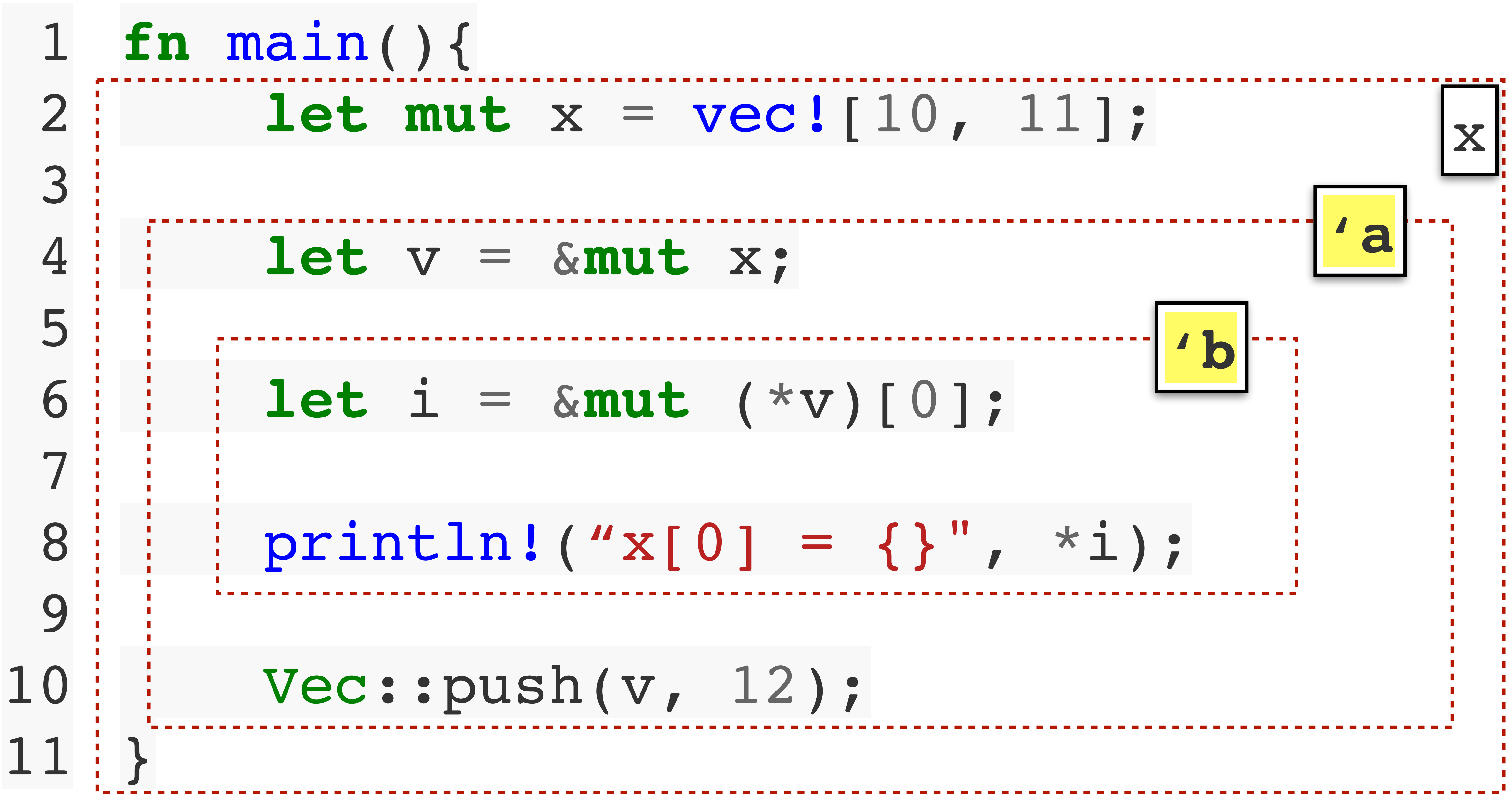
The lender needs to outlive
all of its (alive) references.

Safety Principle 3

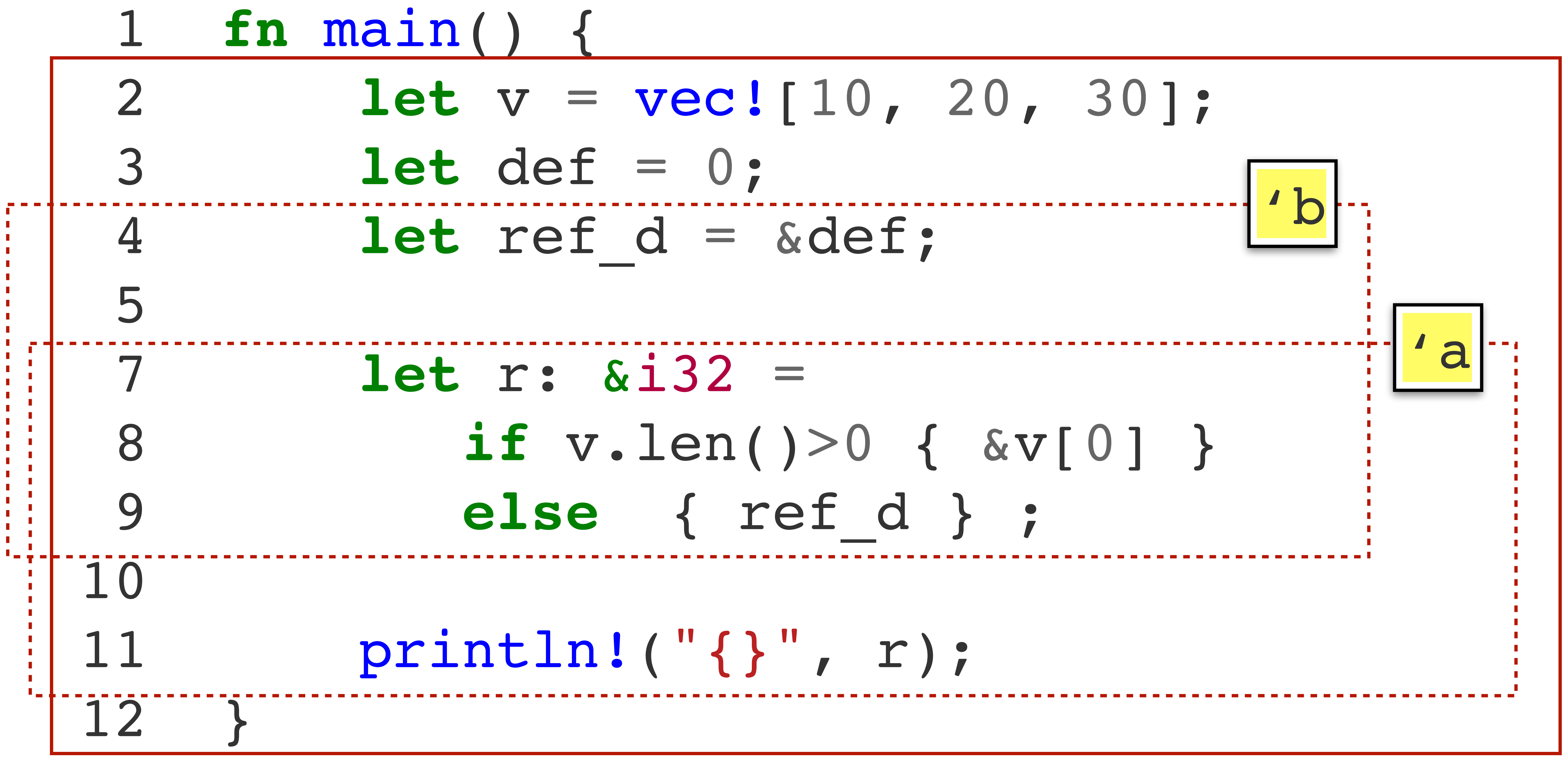
A mutable borrow can only be created if its
lender has no other borrows living at that time.

Safety Principle 4

The lender cannot be modified as long as
one of its (shared) borrowers still lives.



```
fn main(){
    let mut x = vec![10, 11];

    let v = &mut x;

    let i = &mut (*v)[0];

    println!(“x[0] = {}", *i);

    Vec::push(v, 12);
}
```



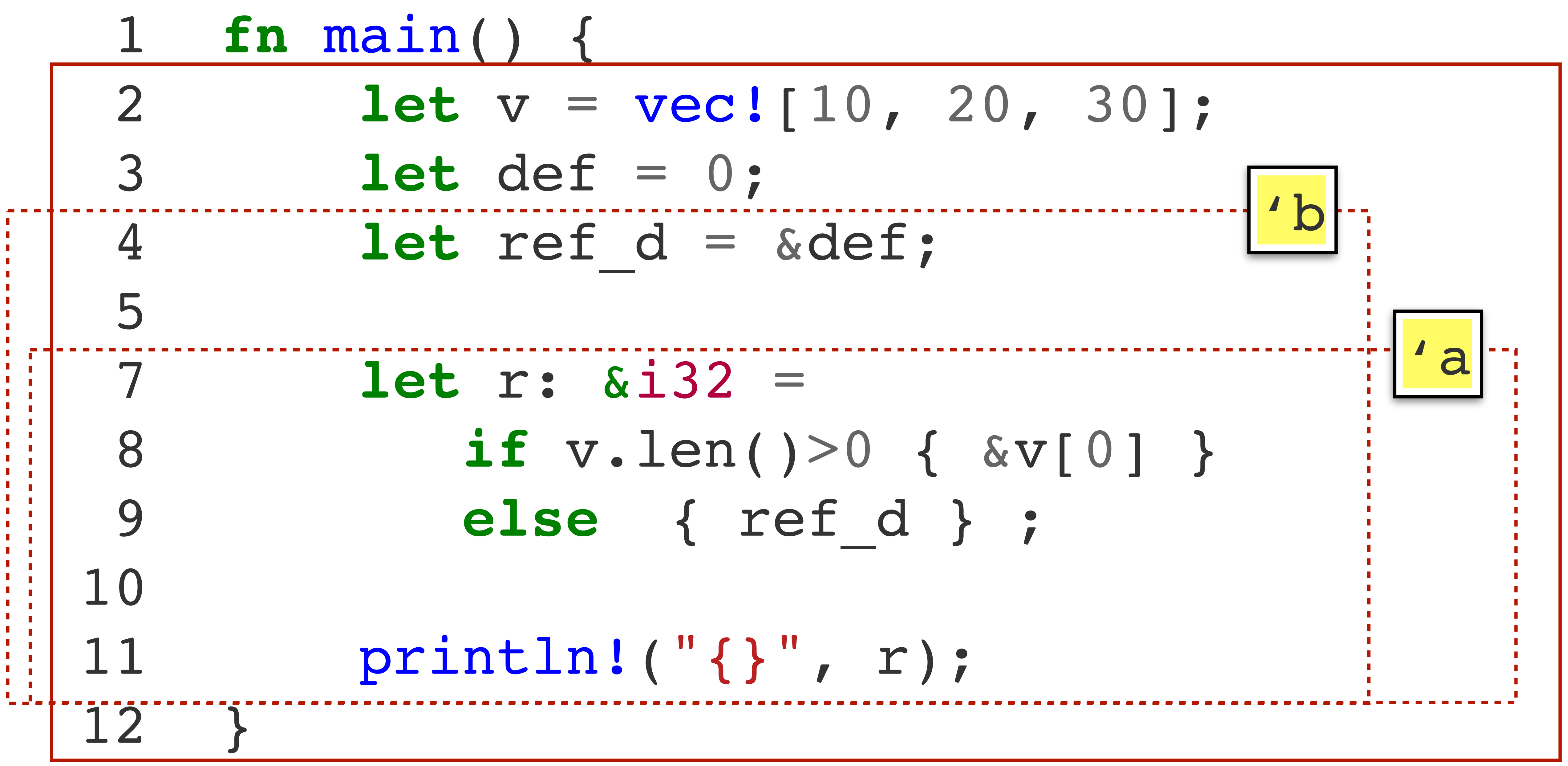
```rust
fn main() {
    let v = vec![10, 20, 30];
    let def = 0;
    let ref_d = &def;

    let r: &i32 =
       if v.len()>0 { &v[0] }
       else  { ref_d } ;

    println!("{}", r);
}
```

Safety Principle 2

The lender needs to outlive all of its (alive) references.

A **lifetime** indicates the code block where a **borrow is valid**:

- starts where the reference is created,
- ends where the reference is last used/needed.



```rust
fn main() {
    let v = vec![10, 20, 30];
    let def = 0;
    let ref_d = &def;

    let r: &i32 =
        if v.len()>0 { &v[0] }
        else  { ref_d } ;

    println!("{}", r);
}
```

Safety Principle 2

The lender needs to outlive all of its (alive) references.

A **lifetime** indicates the code block where a **borrow is valid**:

- starts where the reference is created,
- ends where the reference is last used/needed.

```
fn main() {
    let v = vec![10, 20, 30];
    let def = 0;
    let ref_d = &def;

    let r: &i32 =
       if v.len()>0 { &v[0] }
       else  { ref_d } ;

    println!("{}", r);
}
```

Safety Principle 2

The lender needs to outlive all of its (alive) references.

```
if v.len()>0 { &v[0] }
else  { ref_d } ;
```

```
fn main() {
    let v = vec![10, 20, 30];
    let def = 0;
    let ref_d = &def;

    let r: &i32 =



    println!("{}", r);
}
```

Safety Principle 2

The lender needs to outlive all of its (alive) references.

```
fn get_first(v: &Vec<i32>, ref_d: &i32) -> &i32 {
    if v.len()>0 { &v[0] }
    else  { ref_d } ;
}
```

```
fn main() {
    let v = vec![10, 20, 30];
    let def = 0;
    let ref_d = &def;

    let r: &i32 =



    println!("{}", r);
}
```

Safety Principle 2

The lender needs to outlive all of its (alive) references.

```
fn get_first(v: &Vec<i32>, ref_d: &i32) -> &i32 {
    if v.len()>0 { &v[0] }
    else  { ref_d } ;
}
```

```
fn main() {
     let v = vec![10, 20, 30];
     let def = 0;
     let ref_d = &def;

     let r: &i32 = get_first(&v, ref_d);



     println!("{}", r);
}
```

Safety Principle 2

The lender needs to outlive all of its (alive) references.

Non-Lexical Lifetime

(parameterised)

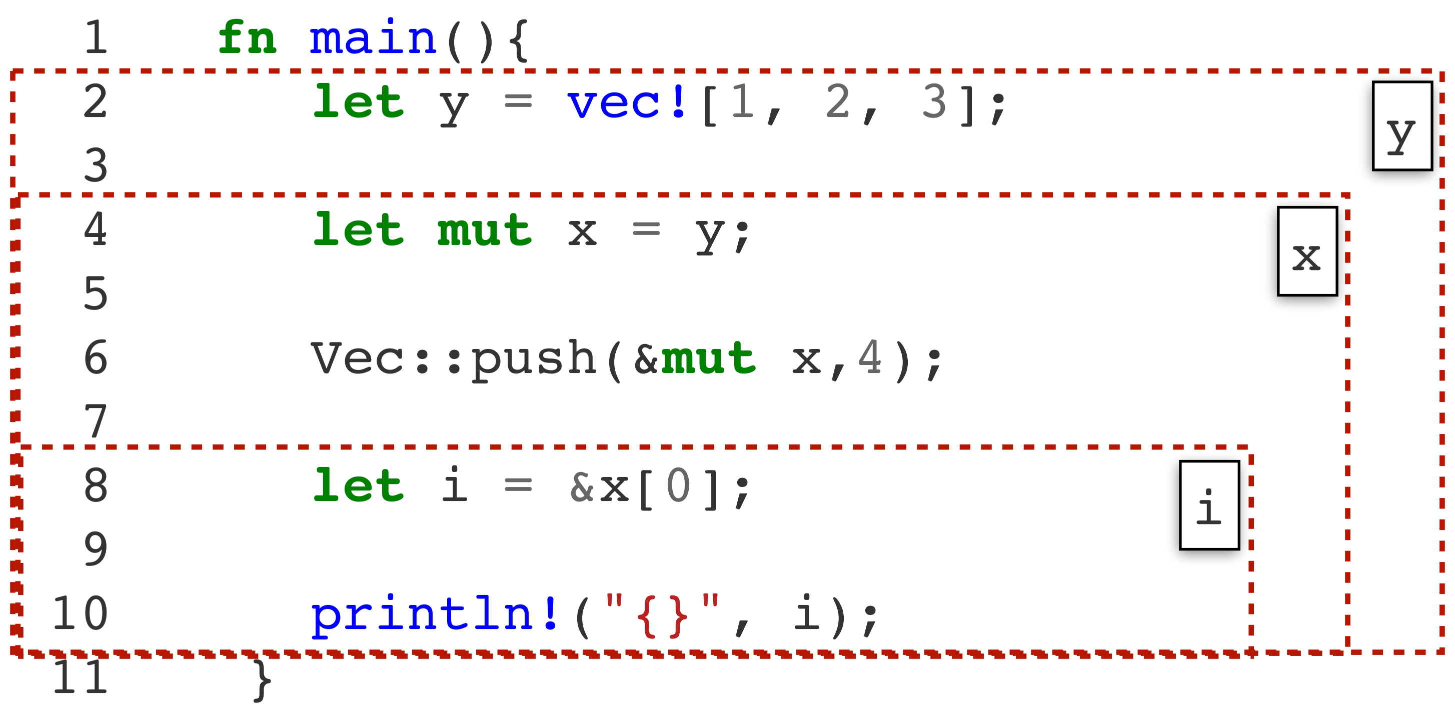
```
fn get_first<‘a>(v: &‘a Vec<i32>, ref_d:&‘a i32) -> &‘a i32 {
    if v.len()>0 { &v[0] }
    else  { ref_d } ;
}
```

```
fn main() {
     let v = vec![10, 20, 30];
     let def = 0;
     let ref_d = &def;

     let r: &i32 = get_first(&v, ref_d);



     println!("{}", r);
}
```

Safety Principle 2

The lender needs to outlive all of its (alive) references.

Scope

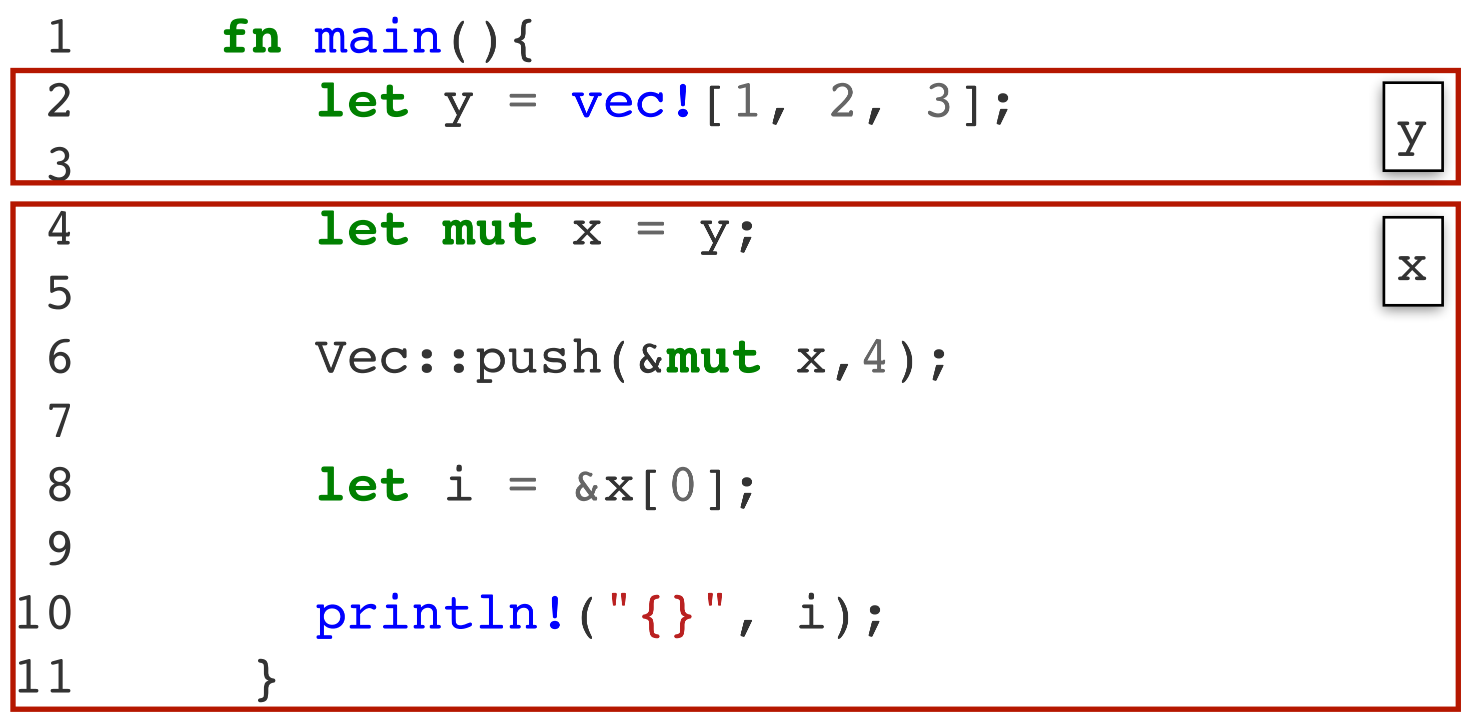


```
fn main(){
    let y = vec![1, 2, 3];

    let mut x = y;

    Vec::push(&mut x,4);

    let i = &x[0];

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

A **scope** indicates the code block where a **variable is valid**:

- starts where the variable is first introduced,
- ends at the corresponding closing “ } ”.

Liveness

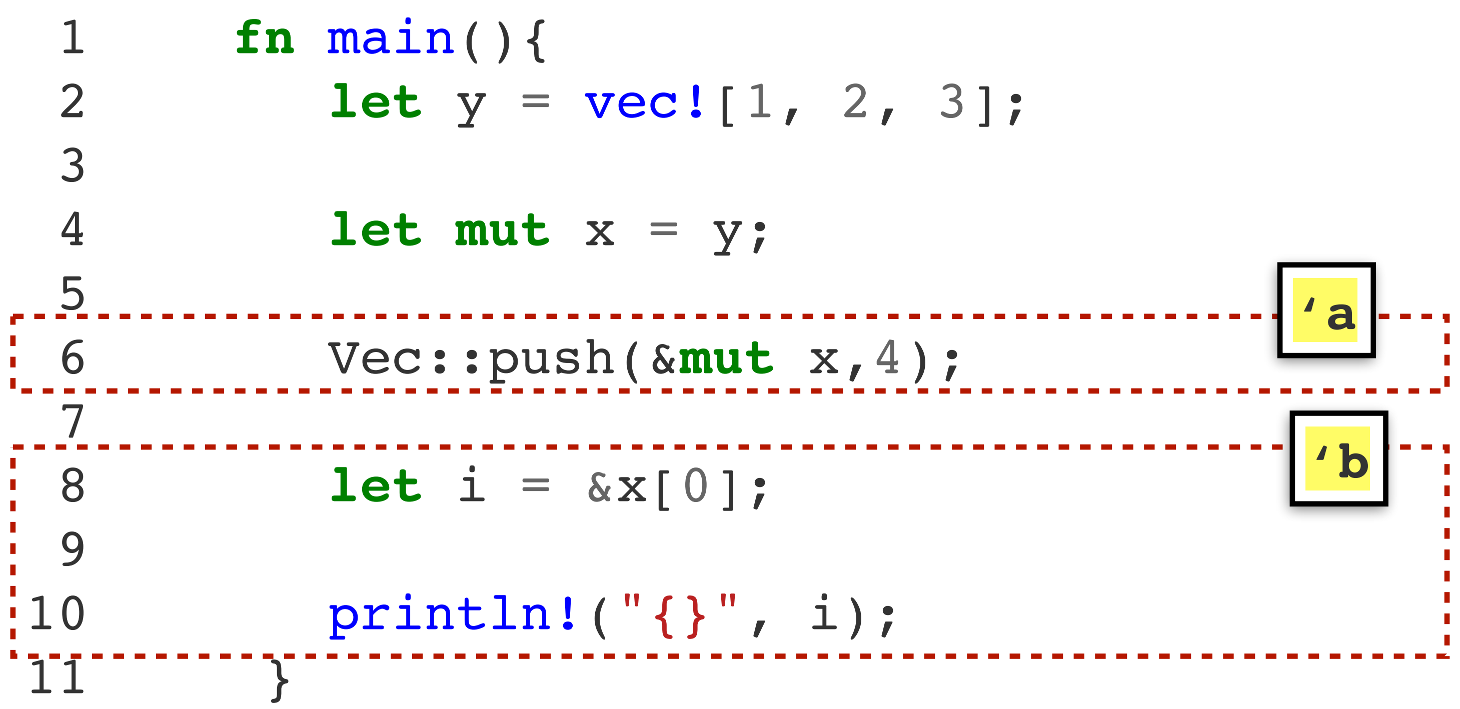


```
fn main(){
    let y = vec![1, 2, 3];

    let mut x = y;

    Vec::push(&mut x,4);

    let i = &x[0];

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

A **variable is live (owner is valid)** from initialisation until:

- its value is moved, or
- it goes out of scope (and is dropped).

Non-Lexical Lifetime



```
fn main(){
    let y = vec![1, 2, 3];

    let mut x = y;

    Vec::push(&mut x,4);

    let i = &x[0];

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

A **lifetime** indicates the code block where a **borrow is valid**:

- starts where the reference is created,
- ends where the reference is last used/needed.

https://cel.cs.brown.edu/aquascope