A borrowed value was moved out.

Erroneous code example:

```
use std::cell::RefCell;

struct TheDarkKnight;

impl TheDarkKnight {
    fn nothing_is_true(self) {}
}

fn main() {
    let x = RefCell::new(TheDarkKnight);

    x.borrow().nothing_is_true(); // error: cannot move out of borrowed content
}
```

Here, the `nothing_is_true` method takes the ownership of `self`. However, `self` cannot be moved because `.borrow()` only provides an `&TheDarkKnight`, which is a borrow of the content owned by the `RefCell`. To fix this error, you have three choices:

- Try to avoid moving the variable.
- Somehow reclaim the ownership.
- Implement the `Copy` trait on the type.

This can also happen when using a type implementing `Fn` or `FnMut`, as neither allows moving out of them (they usually represent closures which can be called more than once). Much of the text following applies equally well to non-`FnOnce` closure bodies.

Examples:

```
use std::cell::RefCell;

struct TheDarkKnight;

impl TheDarkKnight {
    fn nothing_is_true(&self) {} // First case, we don't take ownership
}

fn main() {
    let x = RefCell::new(TheDarkKnight);

    x.borrow().nothing_is_true(); // ok!
}
```

Or:

```
use std::cell::RefCell;

struct TheDarkKnight;

impl TheDarkKnight {
    fn nothing_is_true(self) {}
}

fn main() {
    let x = RefCell::new(TheDarkKnight);
    let x = x.into_inner(); // we get back ownership

    x.nothing_is_true(); // ok!
}
```

Or:

```
use std::cell::RefCell;

#[derive(Clone, Copy)] // we implement the Copy trait
struct TheDarkKnight;

impl TheDarkKnight {
    fn nothing_is_true(self) {}
}

fn main() {
    let x = RefCell::new(TheDarkKnight);

    x.borrow().nothing_is_true(); // ok!
}
```

Moving a member out of a mutably borrowed struct will also cause E0507 error:

```
struct TheDarkKnight;

impl TheDarkKnight {
    fn nothing_is_true(self) {}
}

struct Batcave {
    knight: TheDarkKnight
}

fn main() {
    let mut cave = Batcave {
        knight: TheDarkKnight
    };
```

```
    let borrowed = &mut cave;

    borrowed.knight.nothing_is_true(); // E0507
}
```

It is fine only if you put something back. `mem::replace` can be used for that:

```
# struct TheDarkKnight;
# impl TheDarkKnight { fn nothing_is_true(self) {} }
# struct Batcave { knight: TheDarkKnight }
use std::mem;

let mut cave = Batcave {
    knight: TheDarkKnight
};
let borrowed = &mut cave;

mem::replace(&mut borrowed.knight, TheDarkKnight).nothing_is_true(); // ok!
```

For more information on Rust's ownership system, take a look at the References & Borrowing section of the Book.