A variable was used after its contents have been moved elsewhere.

Erroneous code example:

```
struct MyStruct { s: u32 }

fn main() {
    let mut x = MyStruct{ s: 5u32 };
    let y = x;
    x.s = 6;
    println!("{}", x.s);
}
```

Since `MyStruct` is a type that is not marked `Copy`, the data gets moved out of `x` when we set `y`. This is fundamental to Rust's ownership system: outside of workarounds like `Rc`, a value cannot be owned by more than one variable.

Sometimes we don't need to move the value. Using a reference, we can let another function borrow the value without changing its ownership. In the example below, we don't actually have to move our string to `calculate_length`, we can give it a reference to it with `&` instead.

```
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

A mutable reference can be created with `&mut`.

Sometimes we don't want a reference, but a duplicate. All types marked `Clone` can be duplicated by calling `.clone()`. Subsequent changes to a clone do not affect the original variable.

Most types in the standard library are marked `Clone`. The example below demonstrates using `clone()` on a string. `s1` is first set to "many", and then copied to `s2`. Then the first character of `s1` is removed, without affecting `s2`. "any many" is printed to the console.

```
fn main() {
    let mut s1 = String::from("many");
    let s2 = s1.clone();
    s1.remove(0);
    println!("{} {}", s1, s2);
}
```

If we control the definition of a type, we can implement `Clone` on it ourselves with `#[derive(Clone)]`.

Some types have no ownership semantics at all and are trivial to duplicate. An example is `i32` and the other number types. We don't have to call `.clone()` to clone them, because they are marked `Copy` in addition to `Clone`. Implicit cloning is more convenient in this case. We can mark our own types `Copy` if all their members also are marked `Copy`.

In the example below, we implement a `Point` type. Because it only stores two integers, we opt-out of ownership semantics with `Copy`. Then we can `let p2 = p1` without `p1` being moved.

```
#[derive(Copy, Clone)]
struct Point { x: i32, y: i32 }

fn main() {
    let mut p1 = Point{ x: -1, y: 2 };
    let p2 = p1;
    p1.x = 1;
    println!("p1: {}, {}", p1.x, p1.y);
    println!("p2: {}, {}", p2.x, p2.y);
}
```

Alternatively, if we don't control the struct's definition, or mutable shared ownership is truly required, we can use `Rc` and `RefCell`:

```
use std::cell::RefCell;
use std::rc::Rc;

struct MyStruct { s: u32 }

fn main() {
    let mut x = Rc::new(RefCell::new(MyStruct{ s: 5u32 }));
    let y = x.clone();
    x.borrow_mut().s = 6;
    println!("{}", x.borrow().s);
}
```

With this approach, x and y share ownership of the data via the `Rc` (reference count type). `RefCell` essentially performs runtime borrow checking: ensuring that at most one writer or multiple readers can access the data at any one time.

If you wish to learn more about ownership in Rust, start with the Understanding Ownership chapter in the Book.