A temporary value is being dropped while a borrow is still in active use.

Erroneous code example:

```
fn foo() -> i32 { 22 }
fn bar(x: &i32) -> &i32 { x }
let p = bar(&foo());
         // ------ creates a temporary
let q = *p;
```

Here, the expression `&foo()` is borrowing the expression `foo()`. As `foo()` is a call to a function, and not the name of a variable, this creates a **temporary** – that temporary stores the return value from `foo()` so that it can be borrowed. You could imagine that `let p = bar(&foo());` is equivalent to the following, which uses an explicit temporary variable.

Erroneous code example:

```
# fn foo() -> i32 { 22 }
# fn bar(x: &i32) -> &i32 { x }
let p = {
  let tmp = foo(); // the temporary
  bar(&tmp) // error: `tmp` does not live long enough
}; // <-- tmp is freed as we exit this block
let q = p;
```

Whenever a temporary is created, it is automatically dropped (freed) according to fixed rules. Ordinarily, the temporary is dropped at the end of the enclosing statement – in this case, after the `let`. This is illustrated in the example above by showing that `tmp` would be freed as we exit the block.

To fix this problem, you need to create a local variable to store the value in rather than relying on a temporary. For example, you might change the original program to the following:

```
fn foo() -> i32 { 22 }
fn bar(x: &i32) -> &i32 { x }
let value = foo(); // dropped at the end of the enclosing block
let p = bar(&value);
let q = *p;
```

By introducing the explicit `let value`, we allocate storage that will last until the end of the enclosing block (when `value` goes out of scope). When we borrow `&value`, we are borrowing a local variable that already exists, and hence no temporary is created.

Temporaries are not always dropped at the end of the enclosing statement. In simple cases where the `&` expression is immediately stored into a variable, the compiler will automatically extend the lifetime of the temporary until the end of the enclosing block. Therefore, an alternative way to fix the original program is to write `let tmp = &foo()` and not `let tmp = foo()`:

```
fn foo() -> i32 { 22 }
fn bar(x: &i32) -> &i32 { x }
let value = &foo();
let p = bar(value);
let q = *p;
```

Here, we are still borrowing `foo()`, but as the borrow is assigned directly into a variable, the temporary will not be dropped until the end of the enclosing block. Similar rules apply when temporaries are stored into aggregate structures like a tuple or struct:

```
// Here, two temporaries are created, but
// as they are stored directly into `value`,
// they are not dropped until the end of the
// enclosing block.
fn foo() -> i32 { 22 }
let value = (&foo(), &foo());
```