If you have easy access to the source code of a function (and can modify the code), then it's usually easiest to insert browser() calls directly into the code as you track down various bugs. However, if you do not have easy access to a function's code, or perhaps a function is inside a package that would require rebuilding after each edit, it is sometimes easier to make use of the trace() function to make temporary code modifications.

The simplest use of trace() is to just call trace() on a function without any other arguments.

```
trace("check_n_value")
```

Now, whenever check_n_value() is called by any other functions, you will see a message printed to the console indicating that the function was called.

```
error_if_n_is_greater_than_zero(5)
trace: check_n_value(n)
Error in check_n_value(n): n should be <= 0
```

Here we can see that check_n_value() was called once before the error occurred. But we can do more with trace(), such as inserting a call to browser() in a specific place, such as right before the call tostop().

We can obtain the expression numbers of each part of a function by calling as.list() on the body()of a function.

```
as.list(body(check_n_value))
[[1]]
`{`

[[2]]
if (n > 0) {
    stop("n should be <= 0")
}
```

Here, the if statement is the second expression in the function (the first "expression" being the very beginning of the function). We can further break down the second expression as follows.

```
as.list(body(check_n_value)[[2]])
[[1]]
`if`

[[2]]
n > 0

[[3]]
{
    stop("n should be <= 0")
}
```

Now we can see the call to stop() is the third sub-expression within the second expression of the overall function. We can specify this to trace() by passing an integer vector wrapped in a list to the at argument.

```
trace("check_n_value", browser, at = list(c(2, 3)))
[1] "check_n_value"
```

The trace() function has a side effect of modifying the function and converting into a new object of class "functionWithTrace".