Recently We have been working on a very large legacy project which utilises the excellent [data.table](https://github.com/Rdatatable/data.table) package throughout. What this has resulted in is an R CMD check containing literally thousands of NOTEs similar to the following:

❯ checking R code for possible problems ... NOTE

my\_fn: no visible binding for global variable ‘mpg’

There are several reasons why you might see these NOTEs and, for our code base, some of the NOTEs were potentially more damaging than others. This was a problem as these NOTEs were hidden firstly by a suppression of them due to a manipulation of the \_R\_CHECK\_CODETOOLS\_PROFILE\_ option of the .Renviron file. Once this was removed we discovered the more damaging NOTEs were hidden within the sheer amount of NOTEs we had in the R CMD check.

**Non-standard Evaluation**

If we have a function where we are using data.table’s modification by reference features, i.e. we are using a variable in an unquoted fashion (also known as non-standard evaluation (NSE)) then this issue will occur. Take the following function as an example.

my\_fn <- function() {

mtcars <- data.table::data.table(mtcars)

mtcars[, mpg\_div\_hp := mpg / hp]

mtcars[]

}

Here, we would find the following NOTEs:

❯ checking R code for possible problems ... NOTE

my\_fn: no visible binding for global variable ‘mpg\_div\_hp’

my\_fn: no visible binding for global variable ‘mpg’

my\_fn: no visible binding for global variable ‘hp’

Undefined global functions or variables:

hp mpg mpg\_div\_hp

Sometimes you may also see these NOTEs for syntactic sugar such as !! or := if you haven’t correctly imported the package they come from.

**Option One**

Include all variable names within a globalVariables() call in the package documentation file.

globalVariables(c("mpg", "hp", "mpg\_div\_hp"))

For our package, as there are literally thousands of variables to list in this file, it makes it very difficult to maintain this list and makes the file very long. If, however, the variables belong to data which are stored within your package then this can be greatly simplified to

globalVariables(names(my\_data))

You may wish to import any syntactic sugar functionality here as well. For example

globalVariables(c(":=", "!!"))

**Option Two**

The second option involves binding the variable locally to the function. At the top of your function you can define the variable as a NULL value.

my\_fn <- function() {

mpg <- hp <- mpg\_div\_hp <- NULL

mtcars <- data.table::data.table(mtcars)

mtcars[, mpg\_div\_hp := mpg / hp]

mtcars[]

}

Therefore your variable(s) are now bound to object(s) and so the R CMD check has nothing to complain about. This is the method that the data.table team, feels like a much neater and more importantly maintainable solution than the first option.

**A Note on the Tidyverse**

You may also come across this problem whilst programming using the tidyverse for which there is a very neat solution.

#' @importFrom rlang .data

my\_fn <- function() {

mtcars %>%

mutate(mpg\_div\_hp = .data$mpg / .data$hp)

}

Note the import!

**Selecting Variables with the data.table .. Prefix**

NOTEs can occur when we are using the .. syntax of data.table, for example

double\_dot <- function() {

mtcars <- data.table::data.table(mtcars)

select\_cols <- c("cyl", "wt")

mtcars[, ..select\_cols]

}

This will yield

❯ checking R code for possible problems ... NOTE

Undefined global functions or variables:

..select\_cols

In this instance, this can be solved by avoiding the .. syntax and using the alternative with = FALSE notation.

double\_dot <- function() {

mtcars <- data.table::data.table(mtcars)

select\_cols <- c("cyl", "wt")

mtcars[, select\_cols, with = FALSE]

}

Even though the .. prefix is syntactic sugar, we cannot use globalVariables(c("..")) since the actual variable in this case is ..select\_cols; we would therefore need to use globalVariables(c("..select\_cols")) if we wanted to use the globalVariables() approach.

**Missing Imports**

In our code base, I also found NOTEs for functions or datasets which were not correctly imported. For example, consider the following simple function.

Rversion <- function() {

info <- sessionInfo()

info$R.version

}

This gives the following NOTEs:

❯ checking R code for possible problems ... NOTE

Rversion: no visible global function definition for ‘sessionInfo’

Consider adding

importFrom("utils", "sessionInfo")

to your NAMESPACE file.

Here the R CMD check is rather helpful and tells us the solution; we need to ensure that we explicitly import the function from the utils package in the documentation. This can easily be done with the [roxygen2](https://github.com/r-lib/roxygen2) package by including an @importFrom utils sessionInfo tag.

**Trying to Call Removed Functionality**

If you have a function which has been removed from your package but attempt to call it from another function, R will only give you a NOTE about this.

use\_non\_existent\_function <- function() {

this\_function\_doesnt\_exist()

}

This will give the NOTE

❯ checking R code for possible problems ... NOTE

use\_non\_existent\_function: no visible global function definition for

‘this\_function\_doesnt\_exist’

Of course it goes without saying that you should make sure to remove any calls to functions which have been removed from your package. As a side note, when I first started working on the project, I was initially unaware that within our package we had the option \_R\_CHECK\_CODETOOLS\_PROFILE\_="suppressUndefined=TRUE" set within our .Renviron file which will suppresses all unbound global variable NOTEs from appearing in the R CMD check. However given that this can mask these deeper issues within your package, such as not recognising when a function calls functionality which has been removed from the package. This can end up meaning the end user can face nasty and confusing error messages. Therefore I would not recommend using this setting and would suggest tackling each of your packages NOTEs individually to remove them all.

I actually discovered all of our package NOTEs when introducing the [lintr](https://github.com/jimhester/lintr) package to our CI pipeline. lintr will pick up on some – but not all – of these unbound global variable problems ('lintr of course does not take the \_R\_CHECK\_CODETOOLS\_PROFILE\_ into account). Take our original function as an example

my\_fn <- function() {

mtcars <- data.table::data.table(mtcars)

mtcars[, mpg\_div\_hp := mpg / hp]

mtcars[]

}

Here, lintr will highlight the variables mpg and hp as problems but it currently won’t highlight the variables on the LHS of :=, i.e. mpg\_div\_hp.

**Conclusion**

When developing your package, if you are experiencing these unbound global variables NOTEs you should

1. Strive to define any unbound variables locally within a function.
2. Ensure that any functions or data from external packages (including utils, stats, etc.) have the correct @importFrom tag
3. Do not suppress this check in the .Renviron file and the solutions proposed here should remove the current need to do so
4. Any package wide unbound variables, which are typically syntactic sugar (e.g. :=), should be defined within the package description file inside a globalVariables() function, which should be a very short and maintainable list.