**Semana 4**

**22. Herramientas para hacer debugging**

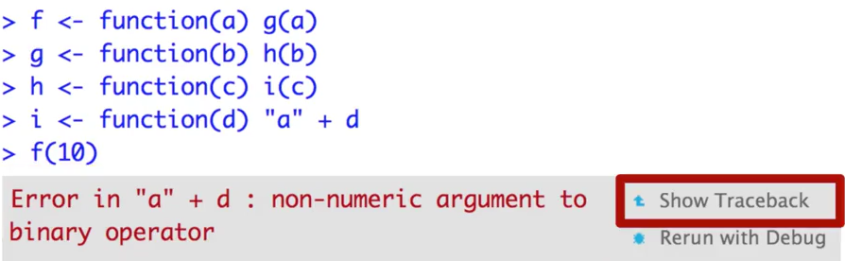
Es muy común que aparezcan errores en nuestro código, y se puede intentar resolver con la función *print()* para ver dónde está exactamente el error. En R hay un conjunto de herramientas que te ayudan a localizar más rápidamente el error, y además que ayudan a depurar.

Para llevar a cabo una depuración, hay que llevar a cabo los siguientes pasos:

1. Primero hay que darnos cuenta de que hay un error.
2. Reproducir el error. Esto es complicado, pues la función que estamos usando puede ser aleatoria. Una vez encontrado y reproducido.
3. Debemos encontrar en qué parte del código se lleva a cabo dicho error.
4. Arreglar el problema y probar.

Una consideración importante al escribir el código en R es que las funciones no sean demasiado largas, porque si hay algún problema es difícil encontrarlo. Es bueno que las funciones no pasen de una página, y si lo hacen se pueden dividir en otras más pequeñas. En RStudio hay 3 herramientas que te ayudan a hacer la depuración

1. Traceback (*traceback()*) nos ayuda a reproducir los pasos que se han llevado a cabo hasta que se encuentra donde se está llevando a cabo el problema.
2. Rerun y Debug vuelven a correr el código y cada uno va llevando a cabo paso por paso las líneas.
3. Breakpoints (*breakpoints()*)



En el caso de que el error fuera cargado con la función *source()*, esta opción nos permite redirigirnos al archivo fuente. Si no es posible arreglar el error, se necesita de un modo más fino, conocido como debugger o depurador, que permite ir a los pasos que crearon el error, pausando entre cada uno de ellos para ver exactamente el origen del problema.

Si queremos entrar en ese modo, hay que darle a la opción de *Rerun with Debug*, dentro del error. Dentro de este modo, puedes ejecutar los siguientes comandos:

* N: Ejecuta el siguiente comando
* S: Pasa a la siguiente función
* F: Termina el bucle actual o la función que se está ejecutando.
* C: Continúa ejecutándose normalmente
* Q: Para la función y regresa a la consola.

El uso de esta consola y sus comandos se pueden modificar en la práctica.

La programación defensiva es una aproximación a la escritura del código, de manera que se detectan errores pronto. Así genera un mensaje que indica de manera clara qué está pasando. Esto se puede hacer de 3 maneras diferentes:

1. Verificar que los parámetros de entrada sean correctos. Es común que con funciones no vectorizadas se usen dentro funciones que sí lo son, por lo que hay que comprobar los valores usando *ifs* o la función *stop* para probar esto.
2. Evitar la evaluación no estándar. Esto es común con funciones como *subset, transform* o *with*.
3. Evitar funciones que nos regresen diferentes tipos de salida. Para evitar esto, con el operador corchete se puede usar el parámetro *drop=FALSE* para evitar cambios de clase. La función *vapply()* es más restrictiva.

Una última recomendación es que cuando se ven código de modo interactivo queremos resultados rápidos, pero debemos evitar hacer suposiciones.