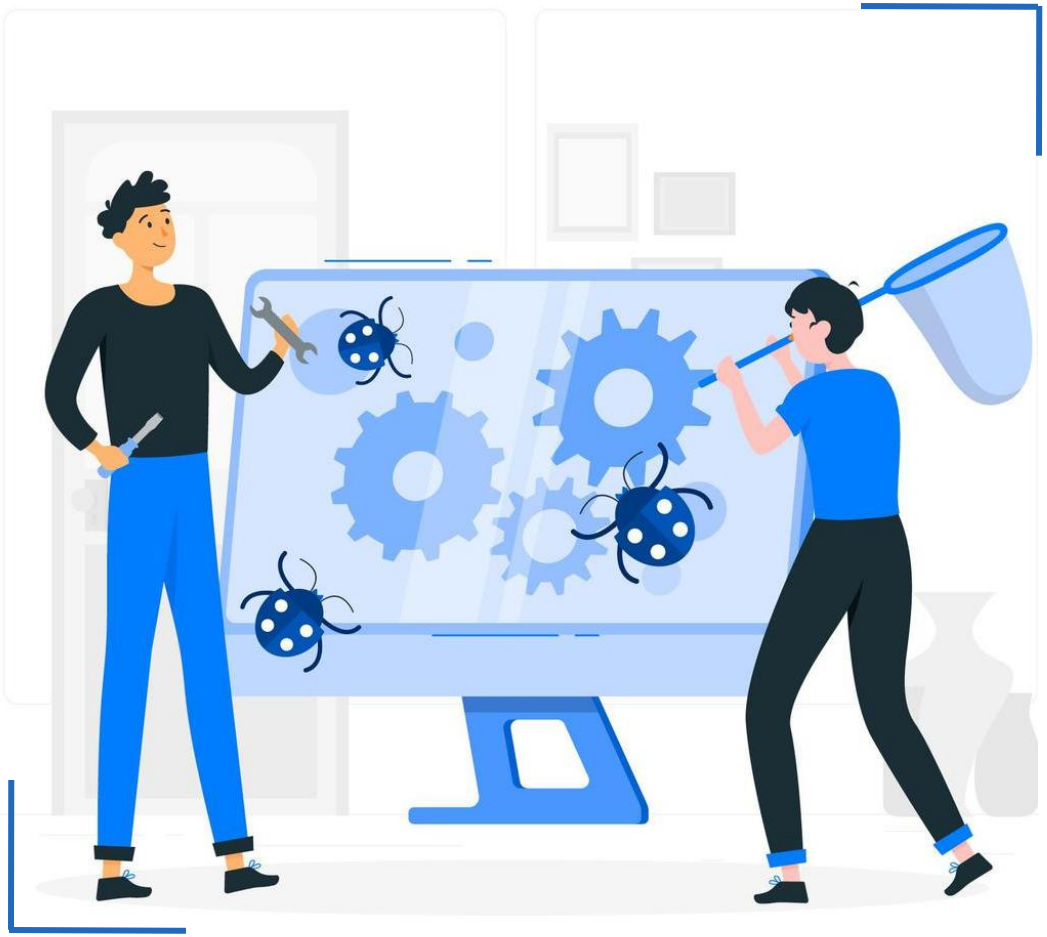
Debug no R

Maria E. G. Alves
Augusto F. M. Barros

Prof. José Cláudio Faria

UESC 2023.2

Introdução



Linguagem R

- É uma linguagem estatística e gráfica, multi-paradigma, dinâmica, voltada à manipulação, análise e visualização de dados.

- A linguagem R é considerada como uma das melhores ferramentas para esses fins, além de ser fácil de aprender até mesmo por quem não tem familiaridade com programação.

O que é Debug?

BUGS sempre vão existir!

Six Stages of Debugging

1. That can't happen.
2. That doesn't happen on my machine.
3. That shouldn't happen.
4. Why does that happen?
5. Oh, I see.
6. How did that ever work?

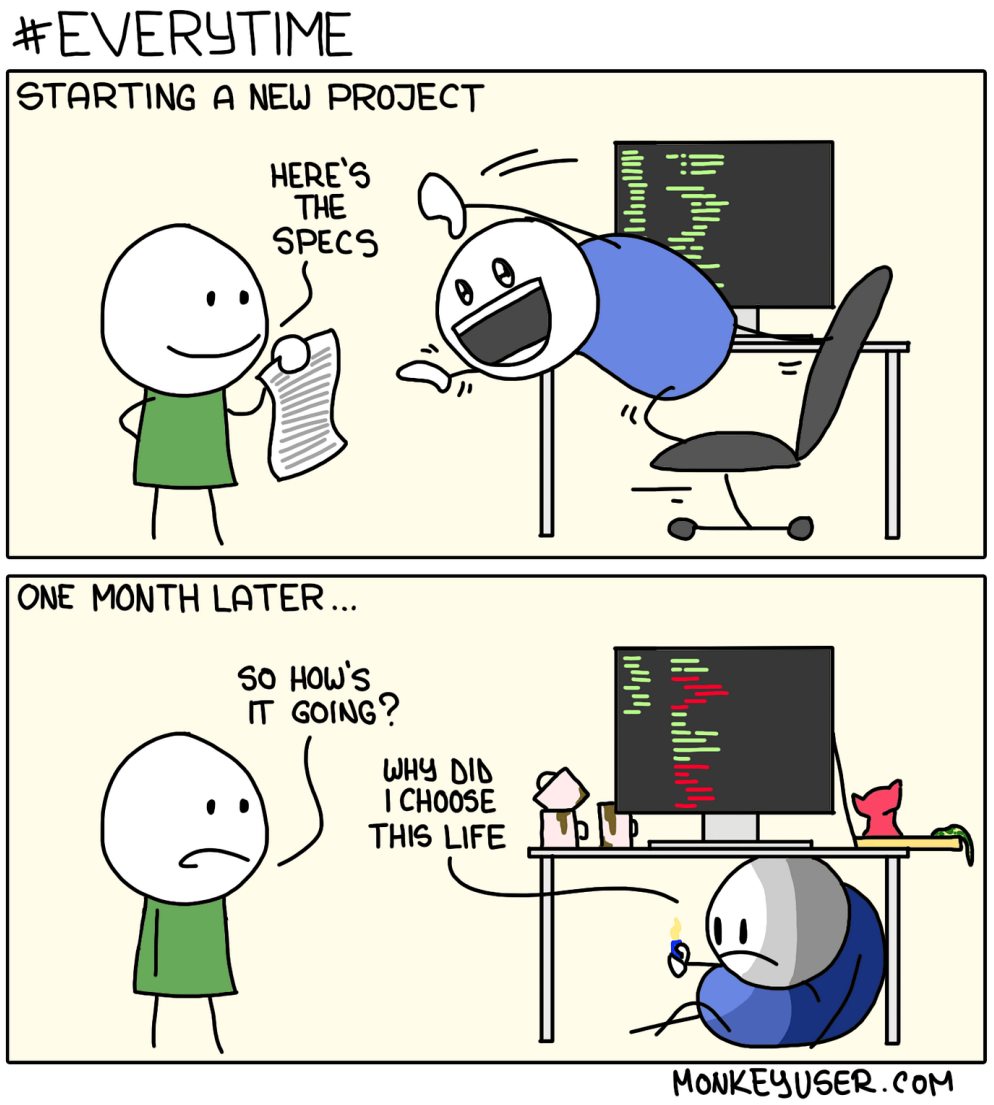
Depurar é a arte e a ciência de corrigir problemas inesperados em seu código.

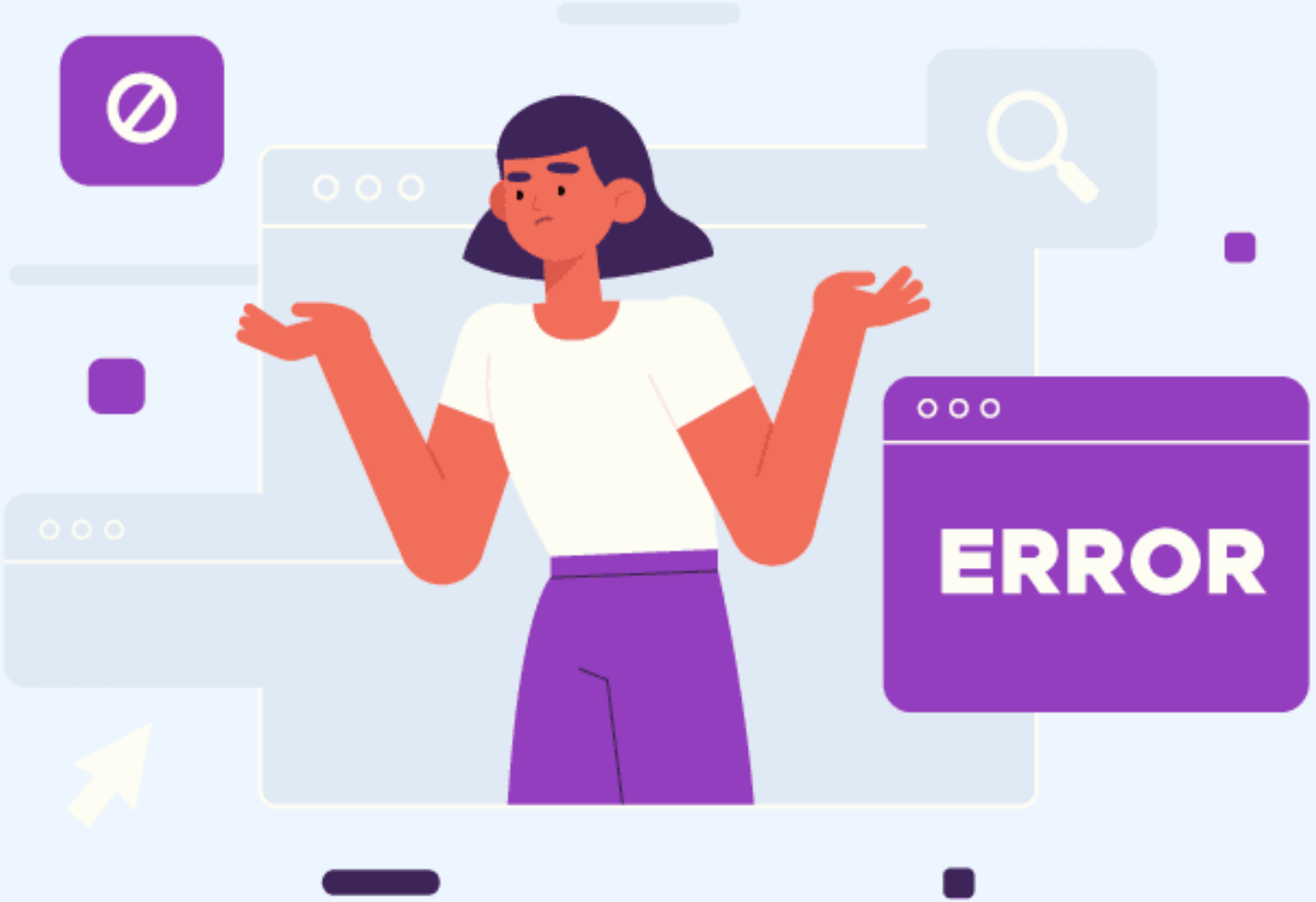
- Um programa gramaticalmente correto pode nos dar resultados incorretos devido a erros lógicos.
- Caso tais bugs ocorram, é preciso descobrir por que e onde eles ocorrem para que possam ser corrigidos.
- O procedimento para identificar e corrigir bugs é chamado de **debug**.

Por que Debugar no R?

- Um analista de dados que saiba debugar um código faz seu trabalho em menos tempo, gerando menos despesa e um produto mais eficiente.

- É essencial para verificar a integridade dos dados coletados e também checar se as funções que geram resultados que serão levados em conta para a análise estão programadas com a lógica correta.





O que acontece quando algo dá errado com seu código R?

O que você faz?

Que ferramentas você tem para resolver o problema?

Princípios Fundamentais de Debug no R

A Essência da Depuração

- **O princípio da confirmação**: Consertar um programa com bugs é um processo de confirmar, uma por uma, que muitas coisas que você acredita serem verdadeiras sobre o código são realmente verdadeiras.
- Quando descobrimos que uma de nossas suposições não é verdadeira, encontramos uma pista para a localização de um bug.

Comece pequeno

- Atenha-se a pequenos casos de teste simples, pelo menos no início do processo de depuração do R. Trabalhar com grandes objetos de dados pode dificultar a reflexão sobre o problema.
- É claro que deveríamos eventualmente testar nosso código em casos grandes e complicados, mas começar aos poucos.

Princípios Fundamentais de Debug no R

Debugging em Modular

- A maioria dos desenvolvedores de software profissionais concorda que o código deve ser escrito de maneira modular.
- As funções não devem ser muito longas e devem chamar outra função, se necessário. Isso torna o código mais fácil na fase de escrita e também para que outras pessoas entendam quando chegar a hora de estender o código.

Antibugging

- Se tivermos uma seção de um código em que uma variável x deve ser positiva, podemos inserir esta linha:

Stopifnot(x>0)

- Se houver um bug no código anterior que renderize x igual a -3, por exemplo, a chamada para *stopifnot()* trará coisas ali mesmo, com uma mensagem de erro como esta:

Error: x > 0 is not TRUE

Lendo uma mensagem de erro: alguns erros comuns

Erro simples

```
"1" + "2"
## Error in "1" + "2": argumento não-numérico para operador binário
```

Selecionando dimensões que não existem

```
x <- 1:10
x[,2]
## Error in x[, 2]: número incorreto de dimensões
```

Argumentos não existentes

```
f <- function(x) x^2
f(y = 1:10)
## Error in f(y = 1:10): unused argument (y = 1:10)
```

Note que o R te fala onde aconteceu o erro.

Debugging Básico

- Nem todos os problemas são inesperados, e comunicar esses problemas é o trabalho das **condições**: erros, avisos e mensagens.

Erros fatais são gerados por **stop()** e forçam o encerramento de toda a execução.

Erros são usados quando não há como uma função continuar.

```
#. Exemplo

erro_fatal <- function(x) {
  if (x < 0) {
    stop("Erro Fatal: O argumento não pode ser menor que
          zero.")
  } else {
    return(sqrt(x))
  }
}

# Testando a função
erro_fatal(4)  # Retorna a raiz quadrada de 4
# [1] 2
erro_fatal(-2) # Gera um erro fatal
# Error in erro_fatal(-2) :
#   Erro Fatal: O argumento não pode ser menor que zero.
```

Debugging Básico

Os avisos são gerados por **warning()** e usados para exibir possíveis problemas.

```
#. Exemplo

aviso_exemplo <- function(vec) {
  if (any(vec < 0))
    warning("Alguns elementos do vetor são negativos.
             Isso pode causar problemas.")
  else
      return(sqrt(vec))
}

# Testando a função
aviso_exemplo(c(4, 1, 9))
# [1] 2 1 3
aviso_exemplo(c(4, -1, 9))  # Gera um aviso
# Warning message:
# In aviso_exemplo(c(4, -1, 9)) : Alguns elementos do vetor
 são negativos.
# Isso pode causar problemas.
```

Debugging Básico

As mensagens são geradas por **message()** e usadas para fornecer resultados informativos de uma forma que pode ser facilmente suprimida pelo usuário usando **suppressMessages()**.

Costuma-se usar mensagens para informar ao usuário qual valor a função escolheu para um importante argumento ausente.

```
#.Exemplo

mensagem_exemplo <- function(x=5) {
  if (missing(x))   # Argumento não fornecido
     message("Você não forneceu um valor para 'x'. ",
             "Será usado o valor padrão: ", x)
  return(sqrt(x))
}

# Testando a função
mensagem_exemplo(9)
# [1] 3
mensagem_exemplo()
# [1] 2.236068
# Você não forneceu um valor para 'x'. Será usado o valor
 padrão: 5

# Usando suppressMessages() para evitar a impressão da
 mensagem
(resultado <- suppressMessages(mensagem_exemplo()))
# [1] 2.236068
```

Debugging Básico

- Inserir instruções print() ou cat() no código pode ajudar a visualizar o valor das variáveis em diferentes pontos do programa. Isso é útil para entender como os valores estão mudando durante a execução. Imagens abaixo para ilustrar:

```r
#. Exemplo usando print()

funcao_com_print <- function(x,y) {
  z <- x + y
  print(paste("O valor de x é:", x))
  print(paste("O valor de y é:", y))
  print(paste("A soma de x+y é:", z))
}

# Testando a função
funcao_com_print(3,5)
# [1] "O valor de x é: 3"
# [1] "O valor de y é: 5"
# [1] "A soma de x+y é: 8"
```

```r
#. Exemplo usando cat()

funcao_com_cat <- function(x,y) {
  z <- x + y
  cat("O valor de x é:", x,
      "\nO valor de y é:", y,
      "\nA soma de x+y é:",z,"\n")
}

# Testando a função
funcao_com_cat(3,5)
# O valor de x é: 3
# O valor de y é: 5
# A soma de x+y é: 8
```

Ferramentas de Debug no **R**

- Ajuda
- Traceback
- Browser
- Debug
- Trace
- Recover
- Try
- TryCatch

Ajuda

- A função help(função) é usada para obter informações de ajuda sobre funções.
- Da mesma forma podemos usar ?função

- A função help.search("palavra") em R é usada para procurar por tópicos de ajuda relacionados a uma palavra-chave específica.
- Da mesma forma podemos usar ??"palavra"

Falha em uma Função: traceback()

- A função **traceback(**função**)** é usada para fornecer todas as informações sobre como sua função chegou a um erro. Ele exibirá todas as funções executadas antes da chegada do erro, chamadas de “ **pilha de chamadas** ”.

```
f <- function(a) g(a)
g <- function(b) h(b)
h <- function(c) i(c)
i <- function(d) "a" + d
f(10)
```

- Leia a pilha de chamadas de baixo para cima: a primeira é a chamada inicial, a última é a que dispara o erro.
- Às vezes, essas são informações suficientes para permitir que você rastreie o erro e o corrija. No entanto, geralmente mostra onde o erro ocorreu, mas não por quê.

Executando passo a passo: browser()

- A função browser() é inserida em funções para abrir o depurador interativo R. Isso interromperá a execução de function() e você poderá examinar a função com o próprio ambiente.

- No modo de depuração, podemos modificar objetos, observar os objetos no ambiente atual e também continuar executando.

```
x <- list(2:6, 'a'); x
fun <- function(x) {
  browser()
  for (i in seq_along(x))
  z[[i]] <-  x[[i]]+1
  return(z)
}
```

- Browse[1]> (comando em consoles) confirma que você está no modo de depuração.

- *Comandos especiais que você pode usar no modo de depuração usando o teclado:*

- Next: **n** : executa a próxima etapa na função. Caso no nome da sua variável seja igual a algum dos comandos, use a função print() se for olhar seu valor.

- Step into: **s** : funciona como next, mas se o próximo passo for uma função, ele entrará nessa função para que você possa trabalhar em cada linha.

- Continue: **c**: deixa a depuração interativa e continua a execução regular da função.

- Stop: **Q** : interrompe a depuração, encerra a função e retorna ao espaço de trabalho global.

- **where**: imprime o rastreamento de pilha de chamadas ativas (o equivalente interativo de traceback).

- Digitando ***ls()***, podemos listar todos os objetos no ambiente local.

Executando passo a passo: debug()

- A função debug() tem a mesma lógica do browser(), mas ao invés de parar em um ponto específico ela para a execução logo no início do código. Ela insere automaticamente a instrução **browser()** no início da função.

- Com o comando debug(mean), por exemplo, toda hora que você executar a função mean() você executará passo a passo a função.

- Para parar de debugar, use **undebug()**. Para debugar apenas uma vez, **debugonce()**.

Pontos de Rastreamento: trace()

- A função trace() em R é usada para adicionar pontos de rastreamento (tracing) em funções existentes. O rastreamento é uma técnica de depuração que permite monitorar a execução de uma função, exibindo informações úteis.

- A sintaxe básica da função trace() é a seguinte:

```
trace("função",
      tracer = expressao,
      at = pontos_de_depuração,
      print = imprimir_mensagem)
```

função: O nome da função à qual você deseja adicionar pontos de rastreamento.

tracer: Uma expressão (normalmente criada com quote()) que será avaliada sempre que a função for chamada. Pode incluir comandos como browser() para iniciar o navegador interativo no ponto de rastreamento.

at: Uma especificação dos pontos nos quais o rastreamento deve ser aplicado. Pode ser um vetor de números inteiros indicando as linhas do corpo da função ou uma fórmula indicando a condição sob a qual o rastreamento deve ser ativado.

print: um valor lógico indicando se deve ou não imprimir mensagens de rastreamento. Se TRUE, as mensagens serão impressas; se FALSE, as mensagens não serão impressas.

Recuperação Após um Erro: recover()

- É utilizada para entrar no modo de recuperação após a ocorrência de um erro. Isso permite que você examine o ambiente no momento do erro e fornece uma interface interativa
- Possui os mesmos comandos da função **browser**.
- É normalmente usada em ativação automática em caso de erro:

```
options(error = recover)
```

- Em **recover()**, R imprime toda a pilha de chamadas e permite que você selecione o navegador de função que deseja inserir. Em seguida, a sessão de depuração começa no local selecionado.

O que as diferencia: browser() **X** recover()

Browser

- **Propósito**: Inicia um ambiente de depuração interativo.
- **Uso**: Colocando browser() em um ponto específico do código, você pode pausar a execução e entrar em um ambiente interativo onde pode inspecionar variáveis, avaliar expressões e interagir com o código.
- **Quando Usar**: Geralmente usado durante a fase de desenvolvimento para entender o estado do código em um ponto específico e identificar problemas.

Recover

- **Propósito**: Entra no modo de recuperação após um erro, permitindo a inspeção do ambiente no momento do erro.
- **Uso**: Configurando options(error = recover), você pode fazer com que o R entre automaticamente no modo de recuperação sempre que ocorrer um erro.
- **Quando Usar**: Útil para identificar e corrigir erros em funções complexas, especialmente quando o erro não é imediatamente aparente.

A Pilha de Chamadas

- Infelizmente, as pilhas de chamadas impressas por **traceback(), browser() + where** e **recover()** não são consistentes.
- A tabela a seguir mostra como as pilhas de chamadas de um conjunto simples de chamadas aninhadas são exibidas pelas duas ferramentas.
- Observe que a numeração é diferente entre elas e exibe chamadas na ordem oposta.

`traceback()`	`where`	`recover()`
`4: stop("Error")`	`where 1: stop("Error")`	`1: f()`
`3: h(x)`	`where 2: h(x)`	`2: g(x)`
`2: g(x)`	`where 3: g(x)`	`3: h(x)`
`1: f()`	`where 4: f()`	

Ignore Erros: try()

- **try()** permite que a execução continue mesmo após a ocorrência de um erro.
  - Por exemplo, normalmente se você executar uma função que gera um erro, ela termina imediatamente e não retorna um valor:

```
f1 <- function(x) {
  log(x)
  10
}
f1("x")
# Error in log(x) : non-numeric
 argument to mathematical function
```

  - No entanto, se você agrupar a instrução que cria o erro em try(), a mensagem de erro será impressa, mas a execução continuará:
  - *Você pode suprimir a mensagem com* **try(..., silent = TRUE)**.

```
f2 <- function(x) {
  try(log(x))
  10
}
f2("a")
# [1] 10
# Error in log(x) : non-numeric
 argument to mathematical function
```

Lidando com Condições: tryCatch()

- É uma ferramenta geral para lidar com condições: além de erros, você pode realizar diferentes ações para avisos, mensagens e interrupções.

- Com tryCatch() você mapeia condições para **manipuladores**, funções nomeadas que são chamadas com a condição como entrada . Se uma condição for sinalizada, tryCatch() chamará o primeiro manipulador cujo nome corresponda a uma das classes da condição.

- Uma função manipuladora pode fazer qualquer coisa, mas normalmente retornará um valor ou criará uma mensagem de erro mais informativa.

Argumentos de tryCatch()

expr: A expressão que você deseja avaliar.

error: Uma função a ser chamada se uma condição de erro ocorrer.

warning: Uma função a ser chamada se uma condição de aviso ocorrer.

message: Uma função a ser chamada se uma condição de mensagem ocorrer.

```
tryCatch(expr,
         error = function(e) NULL,
         warning = function(w) NULL,
         message = function(m) NULL,
         finally = NULL,
         silent = FALSE)
```

finally: Uma expressão ou função que será executada independentemente de ocorrer ou não uma condição.

silent: Um argumento lógico. Se TRUE, suprime as mensagens, avisos e erros padrão, permitindo que você controle a saída manualmente.

Resumo

- Existem três indicações principais de um problema/condição: *message*, *warning*, *error*, sendo apenas um *error* fatal
- Ao analisar uma função com um problema, certifique-se de reproduzir o problema, declarar claramente suas expectativas e como o resultado difere de sua expectativa
- Ferramentas de depuração interativas *traceback*, *debug*, *browser*, *trace,* podem ser usadas para encontrar código problemático em funções.

Função	Efeito
debug()	Marca uma função para debugging.
undebug()	Desmarca uma função para debugging.
browser()	Permite percorrer o código de execução de uma função passo a passo.
trace()	Modifica a função para permite a inserção temporária de de código auxiliar.
untrace()	Cancela a função anterior e remove o código temporário.
traceback()	Imprime a sequência de chamadas a funções que produziram o último erro não capturado.

Referências

- Debug - Seminário de Referência
- Linguagem R - Remessa Online
- Qual é a utilidade de saber depurar um código em R? - Stack Overflow em Português
- Exceptions and Debugging - Advanced R
- Breakpoint Troubleshooting in the RStudio IDE - Posit Support
- R Debug - DataFlair
- R course - Cinelli (2016)
- Depuração visual com R - StatET
- Capítulo 17 Debugging - Resolvendo bugs em suas funções
- Debug - Statistical Computing
- tryCatch in R - Statology
- Depuração em Programação R - Acervo Lima

Fim.