PharmaSUG China 2025 - Paper CC-164

Enhancing Error Handling in R: Leveraging cli and rlang Packages

Ben Gao, MSD, Shanghai, China

ABSTRACT

Effective error handling is essential for developing reliable and user-friendly R programs, particularly when creating Analysis Data Model (ADaM) datasets that involve diverse data sources and intricate derivation logic. This paper introduces techniques to enhance error handling by leveraging functions from the cli and rlang packages. We begin by demonstrating the cli package's ability to generate informative, styled, and structured command line interfaces, featuring inline styling, pluralization, and glue substitutions. Next, we delve into the rlang package's advanced error handling capabilities. The abort function integrates the bulleted messaging from cli, provides detailed error context using the parent and call arguments, and supports creating custom conditions with additional metadata, such as error classes, for more precise error handling. We further examine try_fetch for its superior error context capabilities and local_error_call, which automatically identify environments in error messages. Additionally, we introduce the withRestarts function to establish multiple error recovery strategies, allowing the separation of recovery action definitions from their actual handling steps with try_fetch.

Through practical examples of error handling in data analysis, we demonstrate how programmers can seamlessly integrate these techniques into their workflows, enhancing error message clarity, improving the overall user experience, and facilitating more robust and reliable pharmaceutical data analysis.

INTRODUCTION

R has been gaining popularity for analysis and reporting in the pharmaceutical industry due to its powerful capabilities and extensive package ecosystem. However, the complexity of R programming necessitates robust error handling to ensure reliability and user-friendliness, a challenge that many SAS programmers may find unfamiliar when transitioning to R. Base R provides fundamental error handling capabilities through functions like **try**, **tryCatch**, and **stop**, which allow programmers to catch and handle errors, as well as generate custom error messages. While these tools are useful for basic error management, they may not offer the level of detail and customization needed for more complex error handling scenarios. This can make it challenging to manage intricate error contexts and provide highly informative error messages.

This paper takes a glimpse on addressing these challenges by introducing error handling techniques using the **cli** and **rlang** packages. The cli package offers tools for generating informative and styled command line interfaces, which can enhance the clarity and structure of error messages. Meanwhile, the rlang package provides error handling features, including custom error conditions and detailed error contexts, which are useful for precise error management. This paper aims to equip statistical programmers with practical tools to improve error detection and reporting in R. Through detailed examples, this paper will demonstrate how these techniques can be integrated into existing workflows, e.g., ADaM dataset creation, leading to more reliable and user-friendly R programming processes.

CLI PACKAGE

The cli package in R provides tools for generating informative and styled command line interfaces. It allows users to build a CLI using semantic elements such as headings, lists, alerts, and paragraphs, and supports theming via a CSS-like language. For example, the family of CLI alerts functions, such as *cli_alert()*, *cli_alert_danger()* and *cli_alert_success()*, are typically used to create short status messages. All cli text can contain interpreted string literals through the **glue** package, enabling dynamic and context-aware messages. Additionally, the package supports rich text formatting for error messages, as well as pluralized messages, making error handling more user-friendly and visually appealing.

LIST OF ITEMS

In R programming, printing out a list of items to the console, such as informative notes indicating where errors occur and how to possibly resolve them, can be effectively achieved using the *cli_bullets()* function. *cli_bullets()* creates a styled bullet list from a character vector of items, each of which represents a bullet item and can be formatted differently with a prefixed unicode symbol. Prefix symbols such as *, -, !, and x are useful in generating error messages to visually distinguish different types of information, such as errors, warnings, and notes, enhancing clarity and readability in console output.

```
cli_bullets(c(
        "noindent",
  " " = "indent",
  "*" = "bullet",
  ">" = "arrow",
  "v" = "success",
  "x" = "danger",
  "!" = "warning",
  "i" = "info"
))
```

```
#> noindent
#>   indent
#> • bullet
#> → arrow
#> ✓ success
#> ✖ danger
#> ! warning
#> i info
```

Figure 1 cli_bullets() Prefix Symbols

GLUE SUBSTITUTION

The **glue** package offers interpreted string literals by embedding R expressions in curly braces, which are then evaluated and inserted into the string. cli package integrates the glue package's functionalities so that all cli text is treated as a glue template to allow interpreted string literals. This integration makes it easy to predict what the final string will look like and simplifies the process of constructing complex strings in error messages.

```
name <- "glue"
glue("We are learning how to use the {name} R package.")
#> We are learning how to use the glue R package.
```

Figure 2 Glue String Substitution Example

CONTAINERS

Container elements are used to group elements and can be created using commands such as *cli_div()*, *cli_ul()*, *cli_ol()*, and *cli_li()*. These containers can apply themes that affect the styling of their contents, and the themes are removed when the container exits. For example, *cli_div()* can add a new theme to change the color and font weight of text, which is reverted back to the original style when the container is closed using *cli_end()*. Containers can be auto-closed by default when the function that created them terminates. This feature allows for temporary styling changes within a specific scope, enhancing the flexibility and readability of console output.

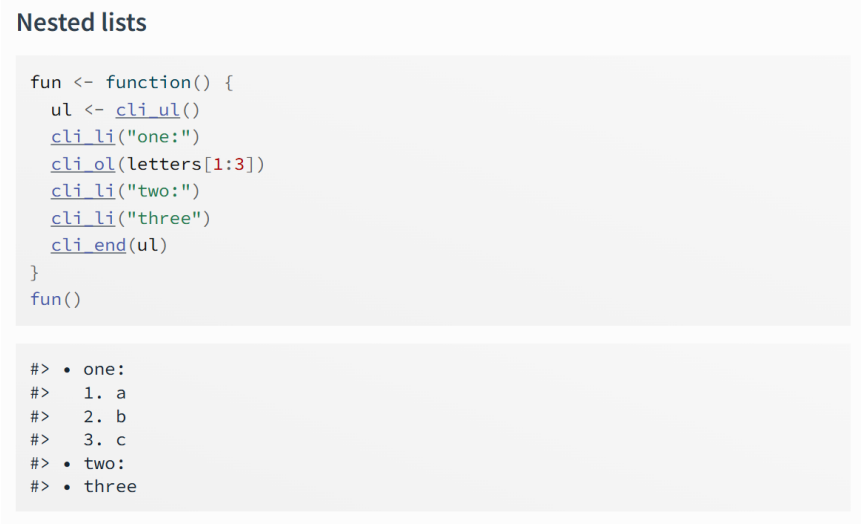

Figure 3 Container Example

INLINE STYLING

cli enables users to apply inline styling, such as bold, italic, color and more, to text by adding classes. The default classes are defined like below. The *cli_text()* may serve a starting point to explore the inline styling features for its simple interface. These classes will help pop out the decorated text to users in the messages. For example, using *.arg* class will surround the arguments with backticks.

cli Classes	Semantic Meaning
emph	Emphasized text
strong	Strong importance
arg	A function argument
code	A piece of code
file	A file name
var	A variable name
val	A generic value
fn	A function name
obj_type_friendly	Formats the type of an R object in a readable way

Table 1 Inline Styling Classes (Non-exhaustive)

```
fun <- function() {
  cli_ul()
  cli_li("{.emph Emphasized} text")
  cli_li("{.strong Strong} importance")
  cli_li("A piece of code: {.code sum(a) / length(a)}")
  cli_li("A package name: {.pkg cli}")
  cli_li("A function name: {.fn cli_text}")
  cli_li("A keyboard key: press {.kbd ENTER}")
  cli_li("A file name: {.file /usr/bin/env}")
  cli_li("An email address: {.email bugs.bunny@acme.com}")
  cli_li("A URL: {.url https://acme.com}")
  cli_li("An environment variable: {.envvar R_LIBS}")
  cli_li("Some {.field field}")
}
fun()
```

```
#> • Emphasized text
#> • Strong importance
#> • A piece of code: `sum(a) / length(a)`
#> • A package name: cli
#> • A function name: `cli_text()`
#> • A keyboard key: press [ENTER]
#> • A file name: /usr/bin/env
#> • An email address: bugs.bunny@acme.com
#> • A URL: <https://acme.com>
#> • An environment variable: `R_LIBS`
#> • Some field
```

Figure 4 Inline Class Examples

When cli performs inline styling, it can automatically collapse glue substitutions, after formatting. This is handy to combines multiple elements into a single string. For example, it can be used to create a message showing a list of files or packages.

```
pkgs <- c("pkg1", "pkg2", "pkg3")
cli_text("Packages: {pkgs}.")
```

```
#> Packages: pkg1, pkg2, and pkg3.
```

Figure 5 Automatic Glue Collapsing

PLURALIZATION

cli has tools to create messages that are printed correctly in singular and plural forms. This usually requires minimal extra work and increases the quality of the messages greatly.

Pluralization Markup

In the simplest case, the message contains a single *{ }* glue substitution, which specifies the quantity that is used to select between the singular and plural forms. The pluralization markup uses the ***{?*** And ***}*** delimiters. In the example hereafter, *nfile* is used to determine whether the singular or plural form of *file* is used.

```
nfile <- 1; cli_text("Found {nfile} file{?s}.")

#> Found 1 file.

nfile <- 2; cli_text("Found {nfile} file{?s}.")

#> Found 2 files.
```

Figure 6 Pluralization Markup Example

Irregular Plurals

If the plural form is more difficult than adding a simple *s* suffix, then the singular and plural forms can be given, separated with a forward slash, e.g., *director{?y/ies}*.

```
ndir <- 1; cli_text("Found {ndir} director{?y/ies}.")

#> Found 1 directory.

ndir <- 5; cli_text("Found {ndir} director{?y/ies}.")

#> Found 5 directories.
```

Figure 7 Irregular Plurals Example

Use “no” Instead of Zero

For readability, it is better to use the *no()* helper function to include a zero count in a message. *no()* prints the word "no" if the count is zero, and prints the numeric count otherwise.

```
nfile <- 0; cli_text("Found {no(nfile)} file{?s}.")

#> Found no files.
```

Figure 8 Use "no" Example

Use the Length of Character Vectors

With the auto-collapsing feature of cli mentioned above, it is easy to include a list of objects in a message. When cli interprets a character vector as a pluralization quantity, it takes the length of the vector as the quantity. Note that the length is only used for non-numeric vectors (when *is.numeric(x)* return *FALSE*). If the users want to use the length of a numeric vector, convert it to character via *as.character()*.

```
pkgs <- character()
cli_text("Will remove {?no/the/the} {.pkg {pkgs}} package{?s}.")
```

```
#> Will remove no packages.
```

```
pkgs <- c("pkg1", "pkg2", "pkg3")
cli_text("Will remove {?no/the/the} {.pkg {pkgs}} package{?s}.")
```

```
#> Will remove the pkg1, pkg2, and pkg3 packages.
```

Figure 9 Use Length of Character Vector Example

The exact rules of cli pluralization can be looked up in the cli package documentation. In the example above, the pluralization markup contains three alternatives (**no/the/the**), the first one is used for zero, the second for one, and the third one for larger quantities.

CUSTOM THEMES

The cli package comes with a simple build-in theme that provides a consistent style for console outputs, including digits, colors, font style, symbols and more. Beyond the built-in theme, cli allows users to customize and create their own themes to match specific preferences or requirements. An example will be provided below to demonstrate the basic usage of custom themes, but its advanced features are beyond the scope of this paper. Users are encouraged to explore further to have fine-tuned control over the appearance of messages

```
fun <- function() {
  cli_div(theme = list (.alert = list(color = "red")))
  cli_alert("This will be red")
  cli_end()
  cli_alert("Back to normal color")
}
fun()
```

```
#> → This will be red
#> → Back to normal color
```

Figure 10 Custom Theme Example

CASE STUDY

In data analysis, importing SAS datasets from a specified path is a frequent task. We can leverage the cli package to create a *check_data_obs* function that prints data summary information to the console, such as identifying empty datasets. The function loops through each SAS dataset found under a folder path and attempts to import it using *haven::read_sas()*. During the iteration, it prints to the console if a dataset is empty and the corresponding number of records, and finally, it provides a summary of the entire importing process, including the names of datasets.

- *cli_alert_success()* and *cli_alert_danger()* are used to give instant status update while iterating over each SAS data set.
- The *.val* inline class is used to display the data set name in a different style; glue substitution *{.val {nrow(tb)}}* uses the *nrow()* function to retrieve the number of rows within the *tb* data frame and inline this number into the status message.

- *cli_h2()* creates a heading for the overall summary information.
- *cli_ul()* creates an unordered list container that includes 2 messages regarding non-empty data sets:
  1. *cli::cli_li("Non-empty Datasets {.strong {length(vec_n_obs)}} dataset{?s}.")* creates a list item and uses the *.strong* inline class to bolden the text representing the number of non-empty data sets. The pluralization markup *{?s}* is used at the end for more than one non-empty datasets.
  2. *cli::cli_text("The dataset{?s} imported are: {vec_n_obs}.")* lists the non-empty data set names separated by commas. It used the pluralization markup *dataset{?s}* and inline collapsing features on the vector *vec_n_obs* to combine the individual names.
- *Cli_div()* creates a new container to define customized properties within that container. For example, texts in scope are displayed in red color within an orange background. The separator is defined as *","* when collapsing vectors.
- The 2nd *cli_ul()* is placed within the div container created above to have text about empty data sets displayed in the new style. *cli::cli_li("Empty datasets: {.strong {no(length(vec_zero_obs))}}* dataset{?s}.") uses the *no()* function to display "no datasets" in case all the data sets are non-empty.

Program for Case Study:

```
check_data_obs <- function(path, ext=".sas7bdat") {
  # get the number of files in .sas7bdat extension under the path
  files <- list.files(path, pattern = stringr::str_c("*", ext),
full.names = FALSE)[1:5]
  vec_zero_obs <- list()
  vec_n_obs <- list()

  purr::walk(files, \(x) {
    tb <- haven::read_sas(file.path(path, x))

    if (nrow(tb) == 0) {
      cli::cli_alert_danger("The dataset {.val {x}} is empty. Please
check the file and try again if necessary.")
      vec_zero_obs <<- append(vec_zero_obs, x)
    } else {
      Cli::cli_alert_success("The dataset {.val {x}} is imported with
{.val {nrow(tb)}} records.")
      vec_n_obs <<- append(vec_n_obs, x)
    }
  })

  Cli::cli_h2("Summary of Dataset Importing under path {.file
{path}})")

  ul_id1 <- cli_ul()
  cli::cli_li("Non-empty Datasets {.strong {length(vec_n_obs)}}
dataset{?s}.")
  cli::cli_text("The dataset{?s} imported are: {vec_n_obs}.")
  cli::cli_end(ul_id1)

  div_id1 <- cli::cli_div(
    id="check_data_obs_summary",
    theme = list(
      span = list("background-color" = "orange",
                  color = "red",
                  "vec-sep" = ", ",
                  "vec-sep2" = " and ",
                  "vec-last" = " and ")
    )
  )

  ul_id2 <- cli::cli_ul()
  cli::cli_li("Empty datasets: {.strong {no(length(vec_zero_obs))}}
dataset{?s}.")
  cli::cli_text("The empty dataset{?s} are: {vec_zero_obs}.")
  cli::cli_end(ul_id2)

  cli::cli_end(div_id1)
}

check_data_obs('Path/to/datasets')
```

```
> check_data_obs('            /path/to/the/datasets                    ')
✓ The dataset "ae.sas7bdat" can be imported with 4924 records.
✕ The dataset "apco.sas7bdat" is empty. Please check the file and try again if necessary.
✓ The dataset "apdm.sas7bdat" can be imported with 2 records.
✓ The dataset "aprp.sas7bdat" can be imported with 27 records.
✓ The dataset "apsc.sas7bdat" can be imported with 1 records.

── Summary of Dataset Importing under path      /path/to/the/datasets                    ) ──

• Non-empty Datasets: 4 datasets.
The datasets imported are: ae.sas7bdat, apdm.sas7bdat, aprp.sas7bdat, and apsc.sas7bdat.
• Empty datasets: 1 dataset.
The empty dataset are: apco.sas7bdat.
```

Figure 11 Console Output for Case Study

RLANG PACKAGE

Functions from cli can be used to generate informative and styled text messages to the console output. Next, we will focus specifically on rlang's error handling functions to enhance the reliability and maintainability of R code. The **rlang** package implements a structured approach to error handling through several key functions. *abort()* serves as the foundation, enabling developers to generate rich, class-based error conditions containing diagnostic metadata and supporting parent error chaining for causal tracing. For control flow, the *try_fetch()* function provides structured condition handling by executing user-defined handlers within the active call stack when errors occur, enabling recovery with values, contextual rethrowing via error chaining, or diagnostic inspection while preserving full debugging information. Together, these functions emphasize contextual metadata storage within error objects and class-based condition management, significantly enhancing error diagnostics, recovery strategies, and debugging workflows beyond base R's capabilities. We will introduce rlang's error handling functions by walking through example functions designed to check duplicate records and deriving baseline flags.

ABORT() AND CLI_ABORT()

The *abort()* function is equivalent to its base counterpart, i.e., *base::stop()* because they both signal an error. The cli package offers a wrapper function *cli_abort()* that enables cli formatting with sophisticated paragraph wrapping and bullet indenting that make long lines easier to read. This wrapper transforms a character vector of lines to a width-wrapped list of error bullets (via a call to *cli_bullets()* introduced in section - **List of items**). This makes it easy to write messages in a list format where each bullet conveys a single important point. These bullets can be combined with the interpolation, semantic formatting of text elements, and pluralization from cli.

```
cli_abort(
  message,
  ...,
  call = .envir,
  .envir = parent.frame(),
  .frame = .envir
)
```

Figure 12 cli_abort() Function Signature

For example, an R function called *chk_dup records()* is created to signal errors if duplicate records exist within a data set. The basic form takes 3 parameters:

- *df*: the data frame to check upon.
- *by_vars*: the grouping variables used to identify duplicate records.
- *msg*: the error message to print to the console.

The sample program for this function is as below. The implementation details irrelevant to error handling will not be discussed here.

```
library(rlang)
library(cli)
library(dplyr)

dup_environment <- rlang::new_environment()

get_duplicates_dataset <- function() {
  dup_environment$duplicates
}


chk_dup_records <- function(df,
                            by_vars,
                            msg = stringr::str_c(
                              "Dataset contains duplicate records",
                              "with respect to",
                              "{.var {by_vars}}",
                              sep = " ")) {
  if (is.null(by_vars)) {
    by_vars <- exprs(!!!parse_exprs(names(df)))
  }

  df_by <- df %>%
    ungroup() %>%
    mutate(!!!by_vars, .keep = "none")

  is_duplicate <- duplicated(df_by) | duplicated(df_by, fromLast = TRUE)

  df_dups <- df %>%
    ungroup() %>%
    mutate(!!!by_vars) %>%
    select(!!!by_vars, everything()) %>%
    filter(is_duplicate) %>%
    arrange(!!!by_vars)


  if (nrow(df_dups) >= 1L) {
    dup_environment$duplicates <- structure(
      df_dups,
      class = union("duplicate_records", class(df_dups)),
      by_vars = by_vars
    )

    full_msg <- c(
      msg,
      "i" = "Run {.run get_duplicates_dataset()} to access the duplicate
records.")


    cli_abort(full_msg)
  }
}


chk_dup_records(lb, exprs(USUBJID, LBTESTCD, LBDTC))
```

In above function, *{.var {by_vars}}* uses the *.var* inline class to format the grouping variables (with backticks surrounding the variables). The final error message (*msg)* adds a user-friendly hyperlink, by using the *.run* class, that runs R code in the current session upon clicking. Running this function on a LB data set using 3 grouping variables (USUBJID, LBTESTCD, LBDTC) generates the following output at console:

```
> chk_dup_records(lb, exprs(USUBJID, LBTESTCD, LBDTC))
Error in `chk_dup_records()` at ~/.active-rstudio-document:61:1:
! Dataset contains duplicate records with respect to `USUBJID`, `LBTESTCD`, and `LBDTC`
i Run get_duplicates_dataset() to access the duplicate records.
Run `rlang::last_trace()` to see where the error occurred.
```

Figure 13 Error Message for Basic Duplicate Checking Function

CONDITION OBJECTS

In R, condition objects are created behind the scenes when conditions are signaled using functions such as *stop()*, *warning()*, *message()* or *abort()*. Error, in particular, is a type of condition object that can be captured and inspected using functions like *rlang::catch_cnd()*. A condition object typically includes:

- A **message element**, which contains the text to display to the user.
- A **call element**, which indicates the call that triggered the condition.
- A **class attribute**, making them S3 objects, which determines the handlers for the condition.

We can examine the error objects from prior example. The *call* element contains the function call we made, and the *class* attribute shows a subclass of *rlang_error*.

```
$ message       : Named chr "Dataset contains duplicate records with respect to `USUBJID`, `LBTESTCD`, and `LBDTC`"

$ call          : language chk_dup_records(lb, exprs(USUBJID, LBTESTCD, LBDTC))

- attr(*, "class")= chr [1:3] "rlang_error" "error" "condition"
```

Figure 14 Error Object Structure

Additional metadata can be store in the condition object by using the **…** arguments, allowing for more detailed and specific error handling.

ADDITIONAL METADATA

cli_abort() or abort() makes it easy to supply condition metadata as below:

- Supply **class** to create a classed condition that can be caught or handled selectively, allowing for finer-grained error handling.
- Supply additional **metadata** with named ... arguments. This data is stored in the condition object and can be examined by handlers.
- Supply **call** to inform users about which function the error occurred in.
- Supply another condition as **parent** to create a chained condition.

Adding Class and Other Custom Metadata

To enrich the error object for downstream inspection, we add 2 levels of metadata to it, i.e., **dup_key_vars** (grouping variables) and **duplicate_records_error** (a new subclass).

```
cli_abort(
  full_msg,
  class = "duplicate_records_error",
  dup_key_vars = stringr::str_c(map_chr(by_vars, as_label), sep=", "))
```

The captured condition object contains two new information: they can help identify the type of error based on its new class and generate more informative error messages with *dup_key_vars*.

```
$ dup_key_vars  : chr [1:3] "USUBJID" "LBTESTCD" "LBDTC"

- attr(*, "class")= chr [1:4] "duplicate_records_error" "rlang_error" "error" "condition"
```

Figure 15 Error Object Structure With New Metadata

Adding Function Calls

By default, *cli_abort()* includes the erroring function in the message. For example, *chk_dup_records(lb, exprs(USUBJID, LBTESTCD, LBDTC))* is included in the *call* element of the error object. This works well when *cli_abort()* is called directly within the failing function. However, when the *cli_abort()* call is exported to another function (which we call an "error helper"), we need to be explicit about which function *cli_abort()* is throwing an error for. Imagine a scenario where this *chk_dup_records()* function is wrapped in a baseline flag derivation function *derive_ablfl(). chk_dup_records()* stops the execution of *derive_ablfl()* if any duplicates are found. Since *derive_ablfl()* is the user-facing function, we would like it to appear in the error message instead of *chk_dup_records()*.

To fix this, let *cli_abort()* know about the function that it is throwing the error by passing the corresponding function environment as the *call* argument. A *derive_ablfl()* template is created below to demonstrate this idea.

- The code below is to be included in the updated *chk_dup_records()*:

```
cli_abort(
  full_msg,
  class = "duplicate_records_error",
  dup_key_vars = stringr::str_c(map_chr(by_vars, as_label), sep=", "),
  call = caller_env())
```

- A new function *derive_ablfl()* definition: it is a placeholder at this stage to simply call *chk_dup_records()*, without full implementations yet.

```
derive_ablfl <- function(){
  chk_dup_records(lb, exprs(USUBJID, LBTESTCD, LBDTC))
  return(NULL)
}

cnd <- catch_cnd(derive_ablfl())
cnd$call
```

Catching the condition object generated from calling *derive_ablfl()* and display its *call* field shows “derive_ablfl()” at the console, which is what we expected.

```
> derive_ablfl <- function(){
+   chk_dup_records(lb, exprs(USUBJID, LBTESTCD, LBDTC))
+   ret .... [TRUNCATED]

> cnd <- catch_cnd(derive_ablfl())

> cnd$call
derive_ablfl()
```

Figure 16 Condition Object With Call Added

local_error_call() from rlang package is an alternative way to explicitly pass a call argument to cli_abort(). It sets the call in a local binding that is automatically picked up by *abort()*. Users are encouraged to

explore its full features. In our example, we can remove the *call* argument from *cli_abort()* within chk_dup_records(), and add *local_error_call(caller_env())* to it.

```
chk_dup_records <- function(…)) {
  …… # code not shown
  local_error_call(caller_env())

  if (nrow(df_dups) >= 1L) {
    cli_abort(
      full_msg,
      class = "duplicate_records_error",
      dup_key_vars = stringr::str_c(map_chr(by_vars, as_label), sep=", ")
    )
  }
}
```

CAPTURE AND HANDLING ERRORS

Instead of letting errors display at the console, we can register handlers to manage the errors. Handlers are functions that take a condition object as argument and are called when the corresponding condition class has been signaled. Base R provides two functions, *tryCatch()* and *withCallingHandlers()*, to register handler functions for the signaled condition. *tryCatch()* registers exiting handlers, and is typically used to handle error conditions. It allows us to override the default error behavior. By contrast, *withCallingHandlers()* sets up calling handlers: code execution continues normally once the handler returns. This tends to make *withCallingHandlers()* a more natural pairing with non-error conditions. We can use condition handlers to recover from conditions with a value, rethrow conditions in an error chain, or inspect conditions to log data about errors and warnings.

The rlang's *try_fetch()* function generalizes *tryCatch()* and *withCallingHandlers()* in a single function. It reproduces the behavior of both calling and exiting handlers depending on the return value of the handler. If the handler returns the *zap()* sentinel, it is taken as a calling handler that declines to recover from a condition. Otherwise, it is taken as an exiting handler which returns a value from the catching site. The important difference between *tryCatch()* and *try_fetch()* is that the program in the *expr* argument is still fully running when an error handler is called. Because the call stack is preserved, this makes it possible to capture a full backtrace from within the handler.

We will use *try_fetch()* to handle the duplication errors in a higher level function that calls *derive_ablfl()*, e.g., an R function *create_adlb()* for creating ADLB dataset. We implement a simple handler that returns the original data. This demonstrates the basic usage of *try_fetch()*, a more useful handler mechanism will be discussed later. Note that *try_fetch()* uses the newly added subclass **duplicate_records_error** to only catch the duplication error, instead of catching all types of errors. Since we are returning the original dataset, *try_fetch()* is functioning as an exiting handler.

```
create_adlb <- function(source_data) {
  lb_chk <- try_fetch({
    derive_ablfl()
    },
  duplicate_records_error = function(e) {
    return(source_data)
  })

  lb_chk
}

create_adlb(lb)
```

RESTARTS

The condition system allows us to split the error handling code into two parts: place code that actually recovers from errors into restarts, and condition handlers can then handle a condition by invoking an appropriate restart. We can place restart code in mid- or low-level functions, such as *derive_ablfl()*, while moving the condition handlers into the upper levels of the program, such as *create_adlb()*.

We can use *withRestarts()* to establish a restart within *derive_ablfl()*. The form of *withRestarts()* is very similar to a *try_fetch()*. In general, a restart name should describe the action the restart takes. We will replace the *derive_ablfl()* template with a more meaningful implementation.

The new version of *derive_ablfl()* with restarts added will look like below. A restart strategy called *reorder_records* is defined, which is used to reorder *df_derive* with a new sequence of keys (*new_order*). The higher level function *create_adlb()* that calls *derive_ablfl()* will need to provide this new sequence of keys to finalize the sorting order and help select the last record before initial treatment.

```
derive_ablfl <- function(df,
                         by_vars,
                         order,
                         filter) {

  filter_expr <- enexpr(filter)
  df_derive <- df %>% dplyr::filter(!!filter_expr)
  df_other <- df %>% dplyr::filter(!(!!filter_expr) |
is.na(!!filter_expr))

  withRestarts(
    {
      if (!is.null(by_vars) || !is.null(order)) {
        if (!is.null(by_vars)) {
          df_derive <- df_derive %>%
            group_by(!!!by_vars) %>%
            arrange(!!!order)

          chk_dup_records(df_derive, exprs(!!!by_vars, !!!order))
        } else {
          df_derive <- df_derive %>%
            arrange(!!!order)

          chk_dup_records(df_derive, exprs(!!!order))
        }
      }
    },
    # define restart points
    reorder_records = function(new_order) {
      cli_inform(c("Recovering from duplicate records with new keys",
                   "i" = "The new order is {.val {new_order}}"))

      df_derive <- df_derive %>% arrange(!!!new_order)
      return(df_derive)
    })

  df_derive_obs <- df_derive %>%
    mutate(obs_num = row_number()) %>%
    ungroup() %>%
    group_by(!!!by_vars) %>%
    mutate(ABLFL = if_else(obs_num == n(), "Y", NA_character_)) %>%
    ungroup()

  bind_rows(df_derive_obs, df_other)
}
```

In *create_adlb()*, we will establish a error handler using *try_fetch()*. As explained in the error handling section, the handler function bound by *try_fetch()* will be run without unwinding the stack - the flow of control will still be in the call to *derive_ablfl()* when this handler function is called. A call to *invokeRestart()* will find and invoke the most recently bound restart with the given name of *reorder_records*.

```
create_adlb <- function(source_lb) {
  lb <-
    try_fetch(
      derive_ablfl(
        source_lb,
        by_vars = exprs(USUBJID, LBTESTCD),
        order = exprs(LBDTC),
        filter = LBDTC <= RFSTDTC & !is.na(LBSTRESC)
      ),

      duplicate_records_error = function(e) {
        invokeRestart(
          "reorder_records",
          new_order = exprs(USUBJID, LBTESTCD, LBDTC, LBSEQ))
      }
    )
  lb %>% arrange(USUBJID, LBTESTCD, LBDTC, LBSEQ)
}
create_adlb(lb)
```

The *withRestarts()* function provides a robust mechanism for error recovery by allowing users to define restart points within a block of code. This approach enhances code maintainability by enabling the separation of error detection from error handling, thus promoting cleaner and more modular code. Additionally, withRestarts empowers users to dynamically choose the most appropriate recovery strategy at runtime, improving the flexibility and adaptability of error management processes.

CONCLUSION

This paper has explored various techniques for enhancing error handling in R using the cli and rlang packages. By leveraging inline classes, pluralization, glue substitution and custom themes from the cli package, we can create more informative and user-friendly error messages. Additionally, the rlang package's rich set of error handling functions provide powerful tools for robust error management. The use of withRestarts further exemplifies how we can implement sophisticated error recovery mechanisms. While this paper only scratches the surface of the vast landscape of error handling in R, it aims to provide readers with a starting point for developing more resilient and maintainable R code. Future exploration and practice will undoubtedly uncover even more techniques and best practices in this critical area of R programming.

REFERENCES

Csárdi G (2025). *cli: Helpers for Developing Command Line Interfaces*. R package version 3.6.5, https://cli.r-lib.org.

Henry L, Wickham H (2025). *rlang: Functions for Base Types and Core R and 'Tidyverse' Features*. R package version 1.1.6, https://github.com/r-lib/rlang, https://rlang.r-lib.org.

Wickham, Hadley. *Beyond Exception Handling: Conditions and Restarts.* http://adv-r.had.co.nz/beyond-exception-handling.html.

Straub, Ben and Sjoberg, Daniel. 2025. “What’s in a message?”, *PHUSE US Connect 2025*, Paper OS04.

ACKNOWLEDGMENTS

The author would like to thank Nicole Zhang, Frédéric Coppin and Amy Gillespie for reviewing this paper.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Ben Gao
Statistical Programming
MSD China
peng.gao4@merck.com