

```

1  !< StringiFor, definition of `string` type.
2  module stringifor_string_t
3  !< StringiFor, definition of `string` type.
4  use, intrinsic :: iso_fortran_env, only : iostat_eor
5  ! use before64, only : b64_decode, b64_encode
6  use penf, only : I1P, I2P, I4P, I8P, R4P, R8P, R16P, str
7
8  implicit none
9  private
10 save
11 ! expose StingiFor overloaded builtins and operators
12 ! public :: adjustl, adjustr, count, index, len, len_trim, repeat, scan, trim, verify
13 public :: adjustl, adjustr, count, index, len_trim, repeat, scan, trim, verify
14 #ifndef __GFORTTRAN__
15 public :: assignment(=), operator(//), operator(.cat.), operator(==), &
16           operator(/=), operator(<), operator(<=), operator(>=), operator(>)
17 #endif
18 ! expose StingiFor objects
19 public :: CK
20 public :: string
21
22 integer, parameter :: CK = selected_char_kind('DEFAULT') !< Default character kind.
23
24 type :: string
25 !< OOP designed string class.
26 private
27 character(kind=CK, len=:), allocatable :: raw !< Raw data.
28 contains
29 ! public methods
30 ! builtins replacements
31 procedure, pass(self) :: adjustl => sadjustl !< Adjustl replacement.
32 procedure, pass(self) :: adjustr => sadjustr !< Adjustr replacement.
33 procedure, pass(self) :: count => scount !< Count replacement.
34 generic :: index => sindex_string_string, &
35             sindex_string_character !< Index replacement.
36 procedure, pass(self) :: len => slen !< Len replacement.
37 procedure, pass(self) :: len_trim => slen_trim !< Len_trim replacement.
38 generic :: repeat => srepeated_string_string, &
39             srepeated_character_string !< Repeat replacement.
40 generic :: scan => sscan_string_string, &
41             sscan_string_character !< Scan replacement.
42 procedure, pass(self) :: trim => strim !< Trim replacement.

```

```

43 generic                :: verify    => sverify_string_string, &
44                               sverify_string_character !< Verify replacement.
45 ! auxiliary methods
46 procedure, pass(self) :: basedir      !< Return the base directory name of a string containing a file name.
47 procedure, pass(self) :: basename     !< Return the base file name of a string containing a file name.
48 procedure, pass(self) :: camelcase    !< Return a string with all words capitalized without spaces.
49 procedure, pass(self) :: capitalize   !< Return a string with its first character capitalized and the rest lowercase.
50 procedure, pass(self) :: chars        !< Return the raw characters data.
51 ! procedure, pass(self) :: decode      !< Decode string.
52 ! procedure, pass(self) :: encode      !< Encode string.
53 procedure, pass(self) :: escape        !< Escape backslashes (or custom escape character).
54 procedure, pass(self) :: extension     !< Return the extension of a string containing a file name.
55 procedure, pass(self) :: fill          !< Pad string on the left (or right) with zeros (or other char) to fill width.
56 procedure, pass(self) :: free          !< Free dynamic memory.
57 generic                :: glob =>      &
58                               glob_character, &
59                               glob_string  !< Glob search, finds all the pathnames matching a given pattern.
60 generic                :: insert =>     &
61                               insert_string, &
62                               insert_character !< Insert substring into string at a specified position.
63 generic                :: join =>       &
64                               join_strings, &
65                               join_characters !< Return a string that is a join of an array of strings or characters.
66 procedure, pass(self) :: lower          !< Return a string with all lowercase characters.
67 procedure, pass(self) :: partition      !< Split string at separator and return the 3 parts (before, the separator and
68 procedure, pass(self) :: read_file       !< Read a file a single string stream.
69 procedure, pass(self) :: read_line      !< Read line (record) from a connected unit.
70 procedure, pass(self) :: read_lines     !< Read (all) lines (records) from a connected unit as a single ascii stream.
71 procedure, pass(self) :: replace        !< Return a string with all occurrences of substring old replaced by new.
72 procedure, pass(self) :: reverse        !< Return a reversed string.
73 procedure, pass(self) :: search         !< Search for *tagged* record into string.
74 procedure, pass(self) :: slice          !< Return the raw characters data sliced.
75 procedure, pass(self) :: snakecase      !< Return a string with all words lowercase separated by "_".
76 procedure, pass(self) :: split          !< Return a list of substring in the string, using sep as the delimiter string
77 procedure, pass(self) :: split_chunked !< Return a list of substring in the string, using sep as the delimiter string
78 procedure, pass(self) :: startcase      !< Return a string with all words capitalized, e.g. title case.
79 procedure, pass(self) :: strip          !< Return a string with the leading and trailing characters removed.
80 procedure, pass(self) :: swapcase       !< Return a string with uppercase chars converted to lowercase and vice versa.
81 procedure, pass(self) :: tempname      !< Return a safe temporary name suitable for temporary file or directories.
82 generic                :: to_number => &
83                               to_integer_I1P, &
84                               to_integer_I2P, &
85                               to_integer_I4P, &
86                               to_integer_I8P, &

```

```

87         to_real_R4P,      &
88 #ifdef _R16P_SUPPORTED
89         to_real_R8P,      &
90         to_real_R16P      !< Cast string to number.
91 #else
92         to_real_R8P      !< Cast string to number.
93 #endif
94     procedure, pass(self) :: unescape      !< Unescape double backslashes (or custom escaped character).
95     procedure, pass(self) :: unique      !< Reduce to one (unique) multiple occurrences of a substring into a string.
96     procedure, pass(self) :: upper      !< Return a string with all uppercase characters.
97     procedure, pass(self) :: write_file      !< Write a single string stream into file.
98     procedure, pass(self) :: write_line      !< Write line (record) to a connected unit.
99     procedure, pass(self) :: write_lines      !< Write lines (records) to a connected unit.
100     ! inquire methods
101     procedure, pass(self) :: end_with      !< Return true if a string ends with a specified suffix.
102     procedure, pass(self) :: is_allocated      !< Return true if the string is allocated.
103     procedure, pass(self) :: is_digit      !< Return true if all characters in the string are digits.
104     procedure, pass(self) :: is_integer      !< Return true if the string contains an integer.
105     procedure, pass(self) :: is_lower      !< Return true if all characters in the string are lowercase.
106     procedure, pass(self) :: is_number      !< Return true if the string contains a number (real or integer).
107     procedure, pass(self) :: is_real      !< Return true if the string contains a real.
108     procedure, pass(self) :: is_upper      !< Return true if all characters in the string are uppercase.
109     procedure, pass(self) :: start_with      !< Return true if a string starts with a specified prefix.
110     ! operators
111     generic :: assignment(=) => string_assign_string,      &
112         string_assign_character,      &
113         string_assign_integer_I1P, &
114         string_assign_integer_I2P, &
115         string_assign_integer_I4P, &
116         string_assign_integer_I8P, &
117         string_assign_real_R4P,      &
118 #ifdef _R16P_SUPPORTED
119         string_assign_real_R8P,      &
120         string_assign_real_R16P      !< Assignment operator overloading.
121 #else
122         string_assign_real_R8P      !< Assignment operator overloading.
123 #endif
124     generic :: operator(//) => string_concat_string,      &
125         string_concat_character, &
126         character_concat_string      !< Concatenation operator overloading.
127     generic :: operator(.cat.) => string_concat_string_string,      &
128         string_concat_character_string, &
129         character_concat_string_string      !< Concatenation operator (string output) overloading.
130     generic :: operator(==) => string_eq_string,      &

```

```

131         string_eq_character, &
132         character_eq_string      !< Equal operator overloading.
133 generic :: operator(/=) => string_ne_string, &
134         string_ne_character, &
135         character_ne_string      !< Not equal operator overloading.
136 generic :: operator(<) => string_lt_string, &
137         string_lt_character, &
138         character_lt_string      !< Lower than operator overloading.
139 generic :: operator(<=) => string_le_string, &
140         string_le_character, &
141         character_le_string      !< Lower equal than operator overloading.
142 generic :: operator(>=) => string_ge_string, &
143         string_ge_character, &
144         character_ge_string      !< Greater equal than operator overloading.
145 generic :: operator(>) => string_gt_string, &
146         string_gt_character, &
147         character_gt_string      !< Greater than operator overloading.
148 ! IO
149 #ifndef __GFORTRAN__
150 generic :: read(formatted) => read_formatted !< Formatted input.
151 generic :: write(formatted) => write_formatted !< Formatted output.
152 generic :: read(unformatted) => read_unformatted !< Unformatted input.
153 generic :: write(unformatted) => write_unformatted !< Unformatted output.
154 #endif
155 ! private methods
156 ! builtins replacements
157 procedure, private, pass(self) :: sindex_string_string !< Index replacement.
158 procedure, private, pass(self) :: sindex_string_character !< Index replacement.
159 procedure, private, pass(self) :: srepeated_string_string !< Repeat replacement.
160 procedure, private, pass(self) :: srepeated_character_string !< Repeat replacement.
161 procedure, private, pass(self) :: sscan_string_string !< Scan replacement.
162 procedure, private, pass(self) :: sscan_string_character !< Scan replacement.
163 procedure, private, pass(self) :: sverify_string_string !< Verify replacement.
164 procedure, private, pass(self) :: sverify_string_character !< Verify replacement.
165 ! auxiliary methods
166 procedure, private, pass(self) :: glob_character !< Glob search (character output).
167 procedure, private, pass(self) :: glob_string !< Glob search (string output).
168 procedure, private, pass(self) :: insert_string !< Insert substring into string at a specified position.
169 procedure, private, pass(self) :: insert_character !< Insert substring into string at a specified position.
170 procedure, private, pass(self) :: join_strings !< Return join string of an array of strings.
171 procedure, private, pass(self) :: join_characters !< Return join string of an array of characters.
172 procedure, private, pass(self) :: to_integer_I1P !< Cast string to integer.
173 procedure, private, pass(self) :: to_integer_I2P !< Cast string to integer.
174 procedure, private, pass(self) :: to_integer_I4P !< Cast string to integer.

```

```

175 procedure, private, pass(self) :: to_integer_I8P      !< Cast string to integer.
176 procedure, private, pass(self) :: to_real_R4P        !< Cast string to real.
177 procedure, private, pass(self) :: to_real_R8P        !< Cast string to real.
178 procedure, private, pass(self) :: to_real_R16P       !< Cast string to real.
179 ! assignments
180 procedure, private, pass(lhs) :: string_assign_string  !< Assignment operator from string input.
181 procedure, private, pass(lhs) :: string_assign_character !< Assignment operator from character input.
182 procedure, private, pass(lhs) :: string_assign_integer_I1P !< Assignment operator from integer input.
183 procedure, private, pass(lhs) :: string_assign_integer_I2P !< Assignment operator from integer input.
184 procedure, private, pass(lhs) :: string_assign_integer_I4P !< Assignment operator from integer input.
185 procedure, private, pass(lhs) :: string_assign_integer_I8P !< Assignment operator from integer input.
186 procedure, private, pass(lhs) :: string_assign_real_R4P   !< Assignment operator from real input.
187 procedure, private, pass(lhs) :: string_assign_real_R8P   !< Assignment operator from real input.
188 procedure, private, pass(lhs) :: string_assign_real_R16P  !< Assignment operator from real input.
189 ! concatenation operators
190 procedure, private, pass(lhs) :: string_concat_string    !< Concatenation with string.
191 procedure, private, pass(lhs) :: string_concat_character  !< Concatenation with character.
192 procedure, private, pass(rhs) :: character_concat_string  !< Concatenation with character (inverted).
193 procedure, private, pass(lhs) :: string_concat_string_string !< Concatenation with string (string output).
194 procedure, private, pass(lhs) :: string_concat_character_string !< Concatenation with character (string output).
195 procedure, private, pass(rhs) :: character_concat_string_string !< Concatenation with character (inverted, string output)
196 ! logical operators
197 procedure, private, pass(lhs) :: string_eq_string        !< Equal to string logical operator.
198 procedure, private, pass(lhs) :: string_eq_character     !< Equal to character logical operator.
199 procedure, private, pass(rhs) :: character_eq_string      !< Equal to character (inverted) logical operator.
200 procedure, private, pass(lhs) :: string_ne_string        !< Not equal to string logical operator.
201 procedure, private, pass(lhs) :: string_ne_character     !< Not equal to character logical operator.
202 procedure, private, pass(rhs) :: character_ne_string     !< Not equal to character (inverted) logical operator.
203 procedure, private, pass(lhs) :: string_lt_string        !< Lower than to string logical operator.
204 procedure, private, pass(lhs) :: string_lt_character     !< Lower than to character logical operator.
205 procedure, private, pass(rhs) :: character_lt_string     !< Lower than to character (inverted) logical operator.
206 procedure, private, pass(lhs) :: string_le_string        !< Lower equal than to string logical operator.
207 procedure, private, pass(lhs) :: string_le_character     !< Lower equal than to character logical operator.
208 procedure, private, pass(rhs) :: character_le_string     !< Lower equal than to character (inverted) logical operator.
209 procedure, private, pass(lhs) :: string_ge_string        !< Greater equal than to string logical operator.
210 procedure, private, pass(lhs) :: string_ge_character     !< Greater equal than to character logical operator.
211 procedure, private, pass(rhs) :: character_ge_string     !< Greater equal than to character (inverted) logical operator.
212 procedure, private, pass(lhs) :: string_gt_string        !< Greater than to string logical operator.
213 procedure, private, pass(lhs) :: string_gt_character     !< Greater than to character logical operator.
214 procedure, private, pass(rhs) :: character_gt_string     !< Greater than to character (inverted) logical operator.
215 ! IO
216 #ifndef __GFORTRAN__
217 procedure, private, pass(dtv) :: read_formatted          !< Formatted input.
218 procedure, private, pass(dtv) :: read_delimited          !< Read a delimited input.

```

```

219 procedure, private, pass(dtv) :: read_undelimited !< Read an undelimited input.
220 procedure, private, pass(dtv) :: read_undelimited_listdirected !< Read an undelimited list directed input.
221 procedure, private, pass(dtv) :: write_formatted !< Formatted output.
222 procedure, private, pass(dtv) :: read_unformatted !< Unformatted input.
223 procedure, private, pass(dtv) :: write_unformatted !< Unformatted output.
224 #endif
225 ! miscellanea
226 procedure, private, pass(self) :: replace_one_occurrence !< Replace the first occurrence of substring old by new.
227 endtype string
228
229 ! internal parameters
230 character(kind=CK, len=26), parameter :: UPPER_ALPHABET = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' !< Upper case alphabet.
231 character(kind=CK, len=26), parameter :: LOWER_ALPHABET = 'abcdefghijklmnopqrstuvwxyz' !< Lower case alphabet.
232 character(kind=CK, len=1), parameter :: SPACE = ' ' !< Space character.
233 character(kind=CK, len=1), parameter :: TAB = achar(9) !< Tab character.
234 character(kind=CK, len=1), parameter :: UNIX_DIR_SEP = char(47) !< Unix/Linux directories separator (
235 character(kind=CK, len=1), parameter :: BACKSLASH = char(92) !< Backslash character.
236
237 ! overloading string name
238 interface string
239 !< Builtin adjustl overloading.
240 module procedure string_
241 endinterface string
242
243 #ifndef __GFORTTRAN__
244 ! operators overloading interfaces
245 interface operator(//)
246 !< Builtin // overloading.
247 module procedure string_concat_string, string_concat_character, character_concat_string
248 endinterface
249 interface assignment(=)
250 !< Builtin = overloading.
251 module procedure string_assign_string, string_assign_character, string_assign_integer_I1P, string_assign_integer_I2P, &
252 string_assign_integer_I4P, string_assign_integer_I8P, string_assign_real_R4P,
253 &
254 string_assign_real_R8P, string_assign_real_R16P
255 #else
256 string_assign_real_R8P
257 #endif
258 endinterface
259 interface operator(==)
260 !< Builtin == overloading.
261 module procedure string_eq_string, string_eq_character, character_eq_string

```

```

262 endinterface
263 interface operator(/=)
264     !< Builtin /= overloading.
265     module procedure string_ne_string, string_ne_character, character_ne_string
266 endinterface
267 interface operator(<)
268     !< Builtin < overloading.
269     module procedure string_lt_string, string_lt_character, character_lt_string
270 endinterface
271 interface operator(<=)
272     !< Builtin <= overloading.
273     module procedure string_le_string, string_le_character, character_le_string
274 endinterface
275 interface operator(>=)
276     !< Builtin >= overloading.
277     module procedure string_ge_string, string_ge_character, character_ge_string
278 endinterface
279 interface operator(>)
280     !< Builtin > overloading.
281     module procedure string_gt_string, string_gt_character, character_gt_string
282 endinterface
283 interface operator(.cat.)
284     !< .cat. overloading.
285     module procedure string_concat_string_string, string_concat_character_string, character_concat_string_string
286 endinterface
287 #endif
288
289 ! builtin overloading
290 interface adjustl
291     !< Builtin adjustl overloading.
292     module procedure sadjustl_character
293 endinterface adjustl
294
295 interface adjustr
296     !< Builtin adjustr overloading.
297     module procedure sadjustr_character
298 endinterface adjustr
299
300 interface count
301     !< Builtin count overloading.
302     module procedure count_substring
303 endinterface
304
305 interface index

```

```

306      !< Builtin index overloading.
307      module procedure sindex_string_string, sindex_string_character, sindex_character_string
308      endinterface index
309
310      ! interface len
311      !      !< Builtin len overloading.
312      !      module procedure slen
313      ! endinterface len
314
315      interface len_trim
316          !< Builtin len_trim overloading.
317          module procedure slen_trim
318      endinterface len_trim
319
320      interface repeat
321          !< Builtin repeat overloading.
322          module procedure srepeat_string_string
323      endinterface repeat
324
325      interface scan
326          !< Builtin scan overloading.
327          module procedure sscan_string_string, sscan_string_character, sscan_character_string
328      endinterface scan
329
330      interface trim
331          !< Builtin trim overloading.
332          module procedure strim
333      endinterface trim
334
335      interface verify
336          !< Builtin verify overloading.
337          module procedure sverify_string_string, sverify_string_character, sverify_character_string
338      endinterface verify
339
340      contains
341          ! public non TBP
342
343          ! creator
344          pure function string(c)
345              !< Return a string given a character input.
346              !<
347              !<````fortran
348              !< print "(L1)", string('Hello World')// ' '=='Hello World '
349              !<````

```



```

350 ! => T <<<
351 character(*), intent(in) :: c !< Character.
352 type(string) :: string_ !< String.
353
354 string_%raw = c
355 endfunction string_
356
357 ! builtins replacements
358 pure function sadjustl_character(s) result(adjusted)
359 !< Left adjust a string by removing leading spaces (character output).
360 !<
361 !<````fortran
362 !< type(string) :: astring
363 !< astring = ' Hello World!'
364 !< print "(L1)", adjustl(astring)=='Hello World!'
365 !<````
366 ! => T <<<
367 class(string), intent(in) :: s !< String.
368 character(kind=CK, len=:), allocatable :: adjusted !< Adjusted string.
369
370 if (allocated(s%raw)) adjusted = adjustl(s%raw)
371 endfunction sadjustl_character
372
373 pure function sadjustr_character(s) result(adjusted)
374 !< Right adjust a string by removing leading spaces (character output).
375 !<
376 !<````fortran
377 !< type(string) :: astring
378 !< astring = 'Hello World!'
379 !< print "(L1)", adjustr(astring)==' Hello World!'
380 !<````
381 ! => T <<<
382 class(string), intent(in) :: s !< String.
383 character(kind=CK, len=:), allocatable :: adjusted !< Adjusted string.
384
385 if (allocated(s%raw)) adjusted = adjustr(s%raw)
386 endfunction sadjustr_character
387
388 elemental function count_substring(s, substring) result(No)
389 !< Count the number of occurrences of a substring into a string.
390 !<
391 !<````fortran
392 !< print "(L1)", count('hello ', substring='ll')==1
393 !<````

```

```

394 ! => T <<<
395 character(*), intent(in) :: s !< String.
396 character(*), intent(in) :: substring !< Substring.
397 integer(I4P) :: No !< Number of occurrences.
398 integer(I4P) :: c1 !< Counters.
399 integer(I4P) :: c2 !< Counters.
400
401 No = 0
402 if (len(substring) > len(s)) return
403 c1 = 1
404 do
405     c2 = index(string=s(c1:), substring=substring)
406     if (c2==0) return
407     No = No + 1
408     c1 = c1 + c2 + len(substring)
409 enddo
410 endfunction count_substring
411
412 elemental function sindex_character_string(s, substring, back) result(i)
413 !< Return the position of the start of the first occurrence of string `substring` as a substring in `string`, counting fr
414 !< If `substring` is not present in `string`, zero is returned. If the back argument is present and true, the return valu
415 !< the start of the last occurrence rather than the first.
416 !<
417 !<````fortran
418 !< type(string) :: string1
419 !< logical :: test_passed(2)
420 !< string1 = 'llo'
421 !< test_passed(1) = index(s='Hello World Hello!', substring=string1)==index(string='Hello World Hello!', substring='llo')
422 !< test_passed(2) = index(s='Hello World Hello!', substring=string1, back=.true.)==index(string='Hello World Hello!', &
423 !< substring='llo', back=.true.)
424 !< print '(L1)', all(test_passed)
425 !<````
426 ! => T <<<
427 character(kind=CK, len=*), intent(in) :: s !< String.
428 type(string), intent(in) :: substring !< Searched substring.
429 logical, intent(in), optional :: back !< Start of the last occurrence rather than the first.
430 integer :: i !< Result of the search.
431
432 if (allocated(substring%raw)) then
433     i = index(string=s, substring=substring%raw, back=back)
434 else
435     i = 0
436 endif
437 endfunction sindex_character_string

```

```

438
439 elemental function sscan_character_string(s, set, back) result(i)
440 !< Return the leftmost (if `back` is either absent or equals false, otherwise the rightmost) character of string that is
441 !<
442 !<````fortran
443 !< type(string) :: string1
444 !< logical      :: test_passed(2)
445 !< string1 = 'llo'
446 !< test_passed(1) = scan(s='Hello World Hello!', set=string1)==scan(string='Hello World Hello!', set='llo')
447 !< test_passed(2) = scan(s='Hello World Hello!', set=string1, back=.true.)==scan(string='Hello World Hello!', &
448 !< set='llo', back=.true.)
449 !< print '(L1)', all(test_passed)
450 !<````
451 !> T <<<
452 character(kind=CK, len=*), intent(in) :: s !< String.
453 type(string), intent(in) :: set !< Searched set.
454 logical, intent(in), optional :: back !< Start of the last occurrence rather than the first.
455 integer :: i !< Result of the search.
456
457 if (allocated(set%raw)) then
458 i = scan(string=s, set=set%raw, back=back)
459 else
460 i = 0
461 endif
462 endfunction sscan_character_string
463
464 elemental function sverify_character_string(s, set, back) result(i)
465 !< Return the leftmost (if `back` is either absent or equals false, otherwise the rightmost) character of string that is
466 !< in `set`. If all characters of `string` are found in `set`, the result is zero.
467 !<
468 !<````fortran
469 !< type(string) :: string1
470 !< logical      :: test_passed(2)
471 !< string1 = 'ell'
472 !< test_passed(1) = verify(s='Hello World Hello!', set=string1)==verify(string='Hello World Hello!', set='llo')
473 !< test_passed(2) = verify(s='Hello World Hello!', set=string1, back=.true.)==verify(string='Hello World Hello!', set='ll
474 !< back=.true.)
475 !< print '(L1)', all(test_passed)
476 !<````
477 !> T <<<
478 character(kind=CK, len=*), intent(in) :: s !< String.
479 type(string), intent(in) :: set !< Searched set.
480 logical, intent(in), optional :: back !< Start of the last occurrence rather than the first.
481 integer :: i !< Result of the search.

```

```

482
483 if (allocated(set%raw)) then
484     i = verify(string=s, set=set%raw, back=back)
485 else
486     i = 0
487 endif
488 endfunction sverify_character_string
489
490 ! public methods
491
492 ! builtins replacements
493 elemental function sadjustl(self) result(adjusted)
494 !< Left adjust a string by removing leading spaces.
495 !<
496 !<````fortran
497 !< type(string) :: astring
498 !< astring = '    Hello World!'
499 !< print "(L1)", astring%adjustl()/' '==' Hello World!
500 !<````
501 !> T <<<
502 class(string), intent(in) :: self !< The string.
503 type(string) :: adjusted !< Adjusted string.
504
505 adjusted = self
506 if (allocated(adjusted%raw)) adjusted%raw = adjustl(adjusted%raw)
507 endfunction sadjustl
508
509 elemental function sadjustr(self) result(adjusted)
510 !< Right adjust a string by removing leading spaces.
511 !<
512 !<````fortran
513 !< type(string) :: astring
514 !< astring = 'Hello World!'
515 !< print "(L1)", astring%adjustr()/' '=='    Hello World!
516 !<````
517 !> T <<<
518 class(string), intent(in) :: self !< The string.
519 type(string) :: adjusted !< Adjusted string.
520
521 adjusted = self
522 if (allocated(adjusted%raw)) adjusted%raw = adjustr(adjusted%raw)
523 endfunction sadjustr
524
525 elemental function scount(self, substring, ignore_isolated) result(No)

```

```

526      !< Count the number of occurrences of a substring into a string.
527      !<
528      !< @note If `ignore_isolated` is set to true the eventual "isolated" occurrences are ignored: an isolated occurrences are
529      !< occurrences happening at the start of string (thus not having a left companion) or at the end of the string (thus not
530      !< right companion).
531      !<
532      !< ``fortran
533      !< type(string) :: astring
534      !< logical      :: test_passed(4)
535      !< astring = '    Hello World  !    '
536      !< test_passed(1) = astring%count(substring=' ')==10
537      !< astring = 'Hello World  !    '
538      !< test_passed(2) = astring%count(substring=' ', ignore_isolated=.true.)==6
539      !< astring = '    Hello World  !'
540      !< test_passed(3) = astring%count(substring=' ', ignore_isolated=.true.)==6
541      !< astring = '    Hello World  !    '
542      !< test_passed(4) = astring%count(substring=' ', ignore_isolated=.true.)==8
543      !< print '(L1)', all(test_passed)
544      !< ``
545      !> T <<<
546      class(string), intent(in)                :: self                !< The string.
547      character(*), intent(in)                :: substring            !< Substring.
548      logical,      intent(in), optional      :: ignore_isolated    !< Ignore "isolated" occurrences.
549      integer                :: No                                !< Number of occurrences.
550      logical                :: ignore_isolated_ !< Ignore "isolated" occurrences, local variable.
551      integer                :: c1                                !< Counter.
552      integer                :: c2                                !< Counter.
553  #ifdef __GFORTRAN__
554      character(kind=CK, len=:), allocatable :: temporary          !< Temporary storage, workaround for GNU bug.
555  #endif
556
557      No = 0
558      if (allocated(self%raw)) then
559          if (len(substring)>len(self%raw)) return
560          ignore_isolated_ = .false. ; if (present(ignore_isolated)) ignore_isolated_ = ignore_isolated
561  #ifdef __GFORTRAN__
562      temporary = self%raw
563  #endif
564      c1 = 1
565      do
566  #ifdef __GFORTRAN__
567          c2 = index(string=temporary(c1:), substring=substring)
568  #else
569          c2 = index(string=self%raw(c1:), substring=substring)

```

```

570 #endif
571     if (c2==0) return
572     if (.not.ignore_isolated_) then
573         No = No + 1
574     else
575         if (.not.((c1==1.and.c2==1) .or. (c1==len(self%raw)-len(substring)+1))) then
576             No = No + 1
577         endif
578     endif
579     c1 = c1 + c2 - 1 + len(substring)
580 enddo
581 endif
582 endfunction scount
583
584 elemental function sindex_string_string(self, substring, back) result(i)
585     !< Return the position of the start of the first occurrence of string `substring` as a substring in `string`, counting fr
586     !< If `substring` is not present in `string`, zero is returned. If the back argument is present and true, the return valu
587     !< the start of the last occurrence rather than the first.
588     !<
589     !<````fortran
590     !< type(string) :: string1
591     !< type(string) :: string2
592     !< logical      :: test_passed(2)
593     !< string1 = 'Hello World Hello!'
594     !< string2 = 'llo '
595     !< test_passed(1) = string1%index(substring=string2)==index(string='Hello World Hello!', substring='llo ')
596     !< test_passed(2) = string1%index(substring=string2, back=.true.)==index(string='Hello World Hello!', substring='llo ', &
597     !<                                     back=.true.)
598     !< print '(L1)', all(test_passed)
599     !<````
600     !> T <<<
601     class(string), intent(in) :: self !< The string.
602     type(string), intent(in) :: substring !< Searched substring.
603     logical, intent(in), optional :: back !< Start of the last occurrence rather than the first.
604     integer :: i !< Result of the search.
605
606     if (allocated(self%raw)) then
607         i = index(string=self%raw, substring=substring%raw, back=back)
608     else
609         i = 0
610     endif
611 endfunction sindex_string_string
612
613 elemental function sindex_string_character(self, substring, back) result(i)

```

```

614 !< Return the position of the start of the first occurrence of string `substring` as a substring in `string`, counting fr
615 !< If `substring` is not present in `string`, zero is returned. If the back argument is present and true, the return valu
616 !< the start of the last occurrence rather than the first.
617 !<
618 !<````fortran
619 !< type(string) :: string1
620 !< logical      :: test_passed(2)
621 !< string1 = 'Hello World Hello!'
622 !< test_passed(1) = string1%index(substring='llo')==index(string='Hello World Hello!', substring='llo')
623 !< test_passed(2) = string1%index(substring='llo', back=.true.)==index(string='Hello World Hello!', substring='llo', back
624 !< print '(L1)', all(test_passed)
625 !<````
626 !> T <<<
627 class(string), intent(in) :: self !< The string.
628 character(kind=CK, len=*), intent(in) :: substring !< Searched substring.
629 logical, intent(in), optional :: back !< Start of the last occurrence rather than the first.
630 integer :: i !< Result of the search.
631
632 if (allocated(self%raw)) then
633     i = index(string=self%raw, substring=substring, back=back)
634 else
635     i = 0
636 endif
637 endfunction sindex_string_character
638
639 elemental function slen(self) result(l)
640 !< Return the length of a string.
641 !<
642 !<````fortran
643 !< type(string) :: astring
644 !< astring = 'Hello World! '
645 !< print "(L1)", astring%len()==len('Hello World! ')
646 !<````
647 !> T <<<
648 class(string), intent(in) :: self !< The string.
649 integer :: l !< String length.
650
651 if (allocated(self%raw)) then
652     l = len(string=self%raw)
653 else
654     l = 0
655 endif
656 endfunction slen
657

```

```

658 elemental function slen_trim(self) result(l)
659   !< Return the length of a string, ignoring any trailing blanks.
660   !<
661   !<````fortran
662   !< type(string) :: astring
663   !< astring = 'Hello World!'
664   !< print "(L1)", astring%len_trim()==len_trim('Hello World!')
665   !<````
666   !> T <<<
667   class(string), intent(in) :: self !< The string.
668   integer :: l !< String length.
669
670   if (allocated(self%raw)) then
671     l = len_trim(string=self%raw)
672   else
673     l = 0
674   endif
675 endfunction slen_trim
676
677 elemental function srepeat_string_string(self, ncopies) result(repeated)
678   !< Concatenates several copies of an input string.
679   !<
680   !<````fortran
681   !< type(string) :: astring
682   !< astring = 'x'
683   !< print "(L1)", astring%repeat(5)//' '=='xxxxx '
684   !<````
685   !> T <<<
686   class(string), intent(in) :: self !< String to be repeated.
687   integer, intent(in) :: ncopies !< Number of string copies.
688   type(string) :: repeated !< Repeated string.
689
690   repeated%raw = repeat(string=self%raw, ncopies=ncopies)
691 endfunction srepeat_string_string
692
693 elemental function srepeat_character_string(rstring, ncopies) result(repeated)
694   !< Concatenates several copies of an input string.
695   !<
696   !<````fortran
697   !< type(string) :: astring
698   !< astring = 'y'
699   !< print "(L1)", astring%repeat('x', 5)//' '=='xxxxx '
700   !<````
701   !> T <<<

```



```

702 character(kind=CK, len=*), intent(in) :: rstring !< String to be repeated.
703 integer, intent(in) :: ncopies !< Number of string copies.
704 type(string) :: repeated !< Repeated string.
705
706 repeated%raw = repeat(string=rstring, ncopies=ncopies)
707 endfunction srepeat_character_string
708
709 elemental function sscan_string_string(self, set, back) result(i)
710 !< Return the leftmost (if `back` is either absent or equals false, otherwise the rightmost) character of string that is
711 !<
712 !<````fortran
713 !< type(string) :: string1
714 !< type(string) :: string2
715 !< logical :: test_passed(2)
716 !< string1 = 'Hello World Hello!'
717 !< string2 = 'llo '
718 !< test_passed(1) = string1%scan(set=string2)==scan(string='Hello World Hello!', set='llo ')
719 !< test_passed(2) = string1%scan(set=string2, back=.true.)==scan(string='Hello World Hello!', set='llo ', back=.true.)
720 !< print '(L1)', all(test_passed)
721 !<````
722 !=> T <<<
723 class(string), intent(in) :: self !< The string.
724 type(string), intent(in) :: set !< Searched set.
725 logical, intent(in), optional :: back !< Start of the last occurrence rather than the first.
726 integer :: i !< Result of the search.
727
728 if (allocated(self%raw).and.allocated(set%raw)) then
729 i = scan(string=self%raw, set=set%raw, back=back)
730 else
731 i = 0
732 endif
733 endfunction sscan_string_string
734
735 elemental function sscan_string_character(self, set, back) result(i)
736 !< Return the leftmost (if `back` is either absent or equals false, otherwise the rightmost) character of string that is
737 !<
738 !<````fortran
739 !< type(string) :: string1
740 !< logical :: test_passed(2)
741 !< string1 = 'Hello World Hello!'
742 !< test_passed(1) = string1%scan(set='llo')==scan(string='Hello World Hello!', set='llo ')
743 !< test_passed(2) = string1%scan(set='llo ', back=.true.)==scan(string='Hello World Hello!', set='llo ', back=.true.)
744 !< print '(L1)', all(test_passed)
745 !<````

```

```

746 !> T <<<
747 class(string), intent(in) :: self !< The string.
748 character(kind=CK, len=*), intent(in) :: set !< Searched set.
749 logical, intent(in), optional :: back !< Start of the last occurrence rather than the first.
750 integer :: i !< Result of the search.
751
752 if (allocated(self%raw)) then
753   i = scan(string=self%raw, set=set, back=back)
754 else
755   i = 0
756 endif
757 endfunction sscan_string_character
758
759 elemental function strim(self) result(trimmed)
760 !< Remove trailing spaces.
761 !<
762 !<````fortran
763 !< type(string) :: astring
764 !< astring = 'Hello World! '
765 !< print "(L1)", astring%trim()==trim('Hello World! ')
766 !<````
767 !> T <<<
768 class(string), intent(in) :: self !< The string.
769 type(string) :: trimmed !< Trimmed string.
770
771 trimmed = self
772 if (allocated(trimmed%raw)) trimmed%raw = trim(trimmed%raw)
773 endfunction strim
774
775 elemental function sverify_string_string(self, set, back) result(i)
776 !< Return the leftmost (if `back` is either absent or equals false, otherwise the rightmost) character of string that is
777 !< in `set`. If all characters of `string` are found in `set`, the result is zero.
778 !<
779 !<````fortran
780 !< type(string) :: string1
781 !< type(string) :: string2
782 !< logical :: test_passed(2)
783 !< string1 = 'Hello World Hello!'
784 !< string2 = 'llo '
785 !< test_passed(1) = string1%verify(set=string2)==verify(string='Hello World Hello!', set='llo ')
786 !< test_passed(2) = string1%verify(set=string2, back=.true.)==verify(string='Hello World Hello!', set='llo ', back=.true.)
787 !< print '(L1)', all(test_passed)
788 !<````
789 !> T <<<

```

```

790 class(string), intent(in) :: self !< The string.
791 type(string), intent(in) :: set !< Searched set.
792 logical, intent(in), optional :: back !< Start of the last occurrence rather than the first.
793 integer :: i !< Result of the search.
794
795 if (allocated(self%raw).and.allocated(set%raw)) then
796 i = verify(string=self%raw, set=set%raw, back=back)
797 else
798 i = 0
799 endif
800 endfunction sverify_string_string
801
802 elemental function sverify_string_character(self, set, back) result(i)
803 !< Return the leftmost (if `back` is either absent or equals false, otherwise the rightmost) character of string that is
804 !< in `set`. If all characters of `string` are found in `set`, the result is zero.
805 !<
806 !<````fortran
807 !< type(string) :: string1
808 !< logical :: test_passed(2)
809 !< string1 = 'Hello World Hello!'
810 !< test_passed(1) = string1%verify(set='llo')==verify(string='Hello World Hello!', set='llo')
811 !< test_passed(2) = string1%verify(set='llo', back=.true.)==verify(string='Hello World Hello!', set='llo', back=.true.)
812 !< print '(L1)', all(test_passed)
813 !<````
814 !> T <<<
815 class(string), intent(in) :: self !< The string.
816 character(kind=CK, len=*), intent(in) :: set !< Searched set.
817 logical, intent(in), optional :: back !< Start of the last occurrence rather than the first.
818 integer :: i !< Result of the search.
819
820 if (allocated(self%raw)) then
821 i = verify(string=self%raw, set=set, back=back)
822 else
823 i = 0
824 endif
825 endfunction sverify_string_character
826
827 ! auxiliary methods
828 elemental function basedir(self, sep)
829 !< Return the base directory name of a string containing a file name.
830 !<
831 !<````fortran
832 !< type(string) :: string1
833 !< logical :: test_passed(4)

```

```

834 !< string1 = '/bar/foo.tar.bz2 '
835 !< test_passed(1) = string1%basedir()// ' '== '/bar '
836 !< string1 = './bar/foo.tar.bz2 '
837 !< test_passed(2) = string1%basedir()// ' '== './bar '
838 !< string1 = 'bar/foo.tar.bz2 '
839 !< test_passed(3) = string1%basedir()// ' '== 'bar '
840 !< string1 = '\\bar\\foo.tar.bz2 '
841 !< test_passed(4) = string1%basedir(sep='\\ ')// ' '== '\\bar '
842 !< print '(L1)', all(test_passed)
843 !< ``
844 !=> T <<<
845 class(string), intent(in) :: self !< The string.
846 character(kind=CK, len=*), intent(in), optional :: sep !< Directory separator.
847 type(string) :: basedir !< Base directory name.
848 character(kind=CK, len=:), allocatable :: sep_ !< Separator, default value.
849 integer :: pos !< Character position.
850
851 if (allocated(self%raw)) then
852     sep_ = UNIX_DIR_SEP ; if (present(sep)) sep_ = sep
853     basedir = self
854     pos = index(self%raw, sep_, back=.true.)
855     if (pos>0) basedir%raw = self%raw(1:pos-1)
856 endif
857 endfunction basedir
858
859 elemental function basename(self, sep, extension, strip_last_extension)
860 !< Return the base file name of a string containing a file name.
861 !<
862 !< Optionally, the extension is also stripped if provided or the last one if required, e.g.
863 !<
864 !< ``fortran
865 !< type(string) :: astring
866 !< logical :: test_passed(5)
867 !< astring = 'bar/foo.tar.bz2 '
868 !< test_passed(1) = astring%basename()// ' '== 'foo.tar.bz2 '
869 !< test_passed(2) = astring%basename(extension='.tar.bz2')// ' '== 'foo '
870 !< test_passed(3) = astring%basename(strip_last_extension=.true.)// ' '== 'foo.tar '
871 !< astring = '\\bar\\foo.tar.bz2 '
872 !< test_passed(4) = astring%basename(sep='\\ ')// ' '== 'foo.tar.bz2 '
873 !< astring = 'bar '
874 !< test_passed(5) = astring%basename(strip_last_extension=.true.)// ' '== 'bar '
875 !< print '(L1)', all(test_passed)
876 !< ``
877 !=> T <<<

```

```

878 class(string), intent(in) :: self !< The string.
879 character(kind=CK, len=*), intent(in), optional :: sep !< Directory separator.
880 character(kind=CK, len=*), intent(in), optional :: extension !< File extension.
881 logical, intent(in), optional :: strip_last_extension !< Flag to enable the stripping of last extension
882 type(string) :: basename !< Base file name.
883 character(kind=CK, len=:), allocatable :: sep_ !< Separator, default value.
884 integer :: pos !< Character position.
885 #ifdef __GFORTRAN__
886 character(kind=CK, len=:), allocatable :: temporary !< Temporary storage, workaround for GNU bug.
887 #endif
888
889 if (allocated(self%raw)) then
890 sep_ = UNIX_DIR_SEP ; if (present(sep)) sep_ = sep
891 basename = self
892 #ifdef __GFORTRAN__
893 temporary = basename%raw
894 pos = index(temporary, sep_, back=.true.)
895 if (pos>0) basename%raw = temporary(pos+1:)
896 #else
897 pos = index(basename%raw, sep_, back=.true.)
898 if (pos>0) basename%raw = self%raw(pos+1:)
899 #endif
900 if (present(extension)) then
901 #ifdef __GFORTRAN__
902 temporary = basename%raw
903 pos = index(temporary, extension, back=.true.)
904 if (pos>0) basename%raw = temporary(1:pos-1)
905 #else
906 pos = index(basename%raw, extension, back=.true.)
907 if (pos>0) basename%raw = basename%raw(1:pos-1)
908 #endif
909 elseif (present(strip_last_extension)) then
910 if (strip_last_extension) then
911 #ifdef __GFORTRAN__
912 temporary = basename%raw
913 pos = index(temporary, '.', back=.true.)
914 if (pos>0) basename%raw = temporary(1:pos-1)
915 #else
916 pos = index(basename%raw, '.', back=.true.)
917 if (pos>0) basename%raw = basename%raw(1:pos-1)
918 #endif
919 endif
920 endif
921 endif

```

```

922 endfunction basename
923
924 elemental function camelcase(self, sep)
925   !< Return a string with all words capitalized without spaces.
926   !<
927   !< @note Multiple subsequent separators are collapsed to one occurrence.
928   !<
929   !<````fortran
930   !< type(string) :: astring
931   !< astring = 'caMeL caSe var'
932   !< print '(L1)', astring%camelcase()/' '==' CamelCaseVar '
933   !<````
934   !> T <<<
935   class(string), intent(in) :: self !< The string.
936   character(kind=CK, len=*), intent(in), optional :: sep !< Separator.
937   type(string) :: camelcase !< Camel case string.
938   type(string), allocatable :: tokens(:) !< String tokens.
939
940   if (allocated(self%raw)) then
941     call self%split(tokens=tokens, sep=sep)
942     tokens = tokens%capitalize()
943     camelcase = camelcase%join(array=tokens)
944   endif
945 endfunction camelcase
946
947 elemental function capitalize(self) result(capitalized)
948   !< Return a string with its first character capitalized and the rest lowercased.
949   !<
950   !<````fortran
951   !< type(string) :: astring
952   !< astring = 'say all Hello WorLD!'
953   !< print '(L1)', astring%capitalize()/' '==' Say all hello world!'
954   !<````
955   !> T <<<
956   class(string), intent(in) :: self !< The string.
957   type(string) :: capitalized !< Upper case string.
958   integer :: c !< Character counter.
959
960   if (allocated(self%raw)) then
961     capitalized = self%lower()
962     c = index(LOWER_ALPHABET, capitalized%raw(1:1))
963     if (c>0) capitalized%raw(1:1) = UPPER_ALPHABET(c:c)
964   endif
965 endfunction capitalize

```

```

966
967 pure function chars(self) result(raw)
968   !< Return the raw characters data.
969   !<
970   !<````fortran
971   !< type(string) :: astring
972   !< astring = 'say all Hello WorLD!'
973   !< print '(L1)', astring%chars()=='say all Hello WorLD!'
974   !<````
975   !> T <<<
976   class(string), intent(in) :: self !< The string.
977   character(kind=CK, len=:), allocatable :: raw !< Raw characters data.
978
979   if (allocated(self%raw)) then
980     raw = self%raw
981   else
982     raw = ''
983   endif
984 endfunction chars
985
986   ! elemental function decode(self, codec) result(decoded)
987   ! !< Return a string decoded accordingly the codec.
988   ! !<
989   ! !< @note Only BASE64 codec is currently available.
990   ! !<
991   ! !<````fortran
992   ! !< type(string) :: astring
993   ! !< astring = 'SG93IGFyZSB5b3U/'
994   ! !< print '(L1)', astring%decode(codec='base64 ')//' '==' 'How are you?'
995   ! !<````
996   ! !> T <<<
997   ! class(string), intent(in) :: self !< The string.
998   ! character(kind=CK, len=*), intent(in) :: codec !< Encoding codec.
999   ! type(string) :: decoded !< Decoded string.
1000   ! type(string) :: codec_u !< Encoding codec in upper case string.
1001
1002   ! if (allocated(self%raw)) then
1003   !   decoded = self
1004   !   codec_u = codec
1005   !   select case (codec_u%upper()// ' ')
1006   !   case ('BASE64 ')
1007   !     call b64_decode(code=self%raw, s=decoded%raw)
1008   !   endselect
1009   !   decoded = decoded%strip(remove_nulls=.true.)

```

```

1010 ! endif
1011 ! endfunction decode
1012
1013 ! elemental function encode(self, codec) result(encoded)
1014 ! !< Return a string encoded accordingly the codec.
1015 ! !<
1016 ! !< @note Only BASE64 codec is currently available.
1017 ! !<
1018 ! !<````fortran
1019 ! !< type(string) :: astring
1020 ! !< astring = 'How are you?'
1021 ! !< print '(L1)', astring%encode(codec='base64 ')/' '==' 'SG93IGFyZSB5b3U/'
1022 ! !<````
1023 ! !> T <<<
1024 ! class(string), intent(in) :: self !< The string.
1025 ! character(kind=CK, len=*), intent(in) :: codec !< Encoding codec.
1026 ! type(string) :: encoded !< Encoded string.
1027
1028 ! if (allocated(self%raw)) then
1029 ! encoded = codec
1030 ! select case(encoded%upper()/' ')
1031 ! case('BASE64 ')
1032 ! call b64_encode(s=self%raw, code=encoded%raw)
1033 ! endselect
1034 ! endif
1035 ! endfunction encode
1036
1037 elemental function escape(self, to_escape, esc) result(escaped)
1038 !< Escape backslashes (or custom escape character).
1039 !<
1040 !<````fortran
1041 !< type(string) :: astring
1042 !< logical :: test_passed(2)
1043 !< astring = '~\s \d+\s*'
1044 !< test_passed(1) = astring%escape(to_escape='\ ')/' '==' '~\s \d+\s*'
1045 !< test_passed(2) = astring%escape(to_escape='\ ', esc='| ')/' '==' '~/\s /\d+/\s*'
1046 !< print '(L1)', all(test_passed)
1047 !<````
1048 !> T <<<
1049 class(string), intent(in) :: self !< The string.
1050 character(kind=CK, len=1), intent(in) :: to_escape !< Character to be escaped.
1051 character(kind=CK, len=*), intent(in), optional :: esc !< Character used to escape.
1052 type(string) :: escaped !< Escaped string.
1053 character(kind=CK, len=:), allocatable :: esc_ !< Character to escape, local variable.

```



```

1054 integer :: c !< Character counter.
1055
1056 if (allocated(self%raw)) then
1057     esc_ = BACKSLASH ; if (present(esc)) esc_ = esc
1058     escaped%raw = ''
1059     do c=1, len(self%raw)
1060         if (self%raw(c:c)==to_escape) then
1061             escaped%raw = escaped%raw//esc_//to_escape
1062         else
1063             escaped%raw = escaped%raw//self%raw(c:c)
1064         endif
1065     enddo
1066 endif
1067 endfunction escape
1068
1069 elemental function extension(self)
1070     !< Return the extension of a string containing a file name.
1071     !<
1072     !<````fortran
1073     !< type(string) :: astring
1074     !< astring = '/bar/foo.tar.bz2 '
1075     !< print '(L1)', astring%extension()/' '=='.bz2 '
1076     !<````
1077     !> T <<<
1078     class(string), intent(in) :: self !< The string.
1079     type(string) :: extension !< Extension file name.
1080     integer :: pos !< Character position.
1081 #ifdef __GFORTRAN__
1082     character(kind=CK, len=:), allocatable :: temporary !< Temporary storage, workaround for GNU bug.
1083 #endif
1084
1085     if (allocated(self%raw)) then
1086         extension = ''
1087         pos = index(self%raw, '.', back=.true.)
1088 #ifdef __GFORTRAN__
1089         temporary = self%raw
1090         if (pos>0) extension%raw = temporary(pos:)
1091 #else
1092         if (pos>0) extension%raw = self%raw(pos:)
1093 #endif
1094     endif
1095 endfunction extension
1096
1097 elemental function fill(self, width, right, filling_char) result(filled)

```

```

1098 !< Pad string on the left (or right) with zeros (or other char) to fill width.
1099 !<
1100 !<````fortran
1101 !< type(string) :: astring
1102 !< logical      :: test_passed(4)
1103 !< astring = 'this is string example....wow!!!'
1104 !< test_passed(1) = astring%fill(width=40)//' '=='00000000this is string example....wow!!!'
1105 !< test_passed(2) = astring%fill(width=50)//' '=='000000000000000000this is string example....wow!!!'
1106 !< test_passed(3) = astring%fill(width=50, right=.true.)//' '=='this is string example....wow!!!00000000000000000000'
1107 !< test_passed(4) = astring%fill(width=40, filling_char='*')//' '=='*****this is string example....wow!!!'
1108 !< print '(L1)', all(test_passed)
1109 !<````
1110 !> T <<<
1111 class(string),           intent(in)           :: self           !< The string.
1112 integer,                 intent(in)           :: width          !< Final width of filled string.
1113 logical,                 intent(in), optional       :: right          !< Fill on the right instead of left.
1114 character(kind=CK, len=1), intent(in), optional :: filling_char   !< Filling character (default "0").
1115 type(string)             :: filled            !< Filled string.
1116 logical                  :: right_           !< Fill on the right instead of left, local variable.
1117 character(kind=CK, len=1) :: filling_char_  !< Filling character (default "0"), local variable.
1118
1119 if (allocated(self%raw)) then
1120   if (width>len(self%raw)) then
1121     right_ = .false. ; if (present(right)) right_ = right
1122     filling_char_ = '0' ; if (present(filling_char)) filling_char_ = filling_char
1123     if (.not.right_) then
1124       filled%raw = repeat(filling_char_, width-len(self%raw))//self%raw
1125     else
1126       filled%raw = self%raw//repeat(filling_char_, width-len(self%raw))
1127     endif
1128   endif
1129 endif
1130 endfunction fill
1131
1132 elemental subroutine free(self)
1133 !< Free dynamic memory.
1134 !<
1135 !<````fortran
1136 !< type(string) :: astring
1137 !< astring = 'this is string example....wow!!!'
1138 !< call astring%free
1139 !< print '(L1)', astring%is_allocated().eqv..false.
1140 !<````
1141 !> T <<<

```

```

1142 class(string), intent(inout) :: self !< The string.
1143
1144 if (allocated(self%raw)) deallocate(self%raw)
1145 endsubroutine free
1146
1147 subroutine glob_character(self, pattern, list)
1148 !< Glob search (character output), finds all the pathnames matching a given pattern according to the rules used by the Un
1149 !<
1150 !< @note Method not portable: works only on Unix/GNU Linux OS.
1151 !<
1152 !<````fortran
1153 !< type(string) :: astring
1154 !< character(len=:), allocatable :: alist_chr(:)
1155 !< integer, parameter :: Nf=5
1156 !< character(14) :: files(1:Nf)
1157 !< integer :: file_unit
1158 !< integer :: f
1159 !< integer :: ff
1160 !< logical :: test_passed
1161 !< do f=1, Nf
1162 !< files(f) = astring%tempname(prefix='foo-')
1163 !< open(newunit=file_unit, file=files(f))
1164 !< write(file_unit, *)f
1165 !< close(unit=file_unit)
1166 !< enddo
1167 !< call astring%glob(pattern='foo-*', list=alist_chr)
1168 !< do f=1, Nf
1169 !< open(newunit=file_unit, file=files(f))
1170 !< close(unit=file_unit, status='delete')
1171 !< enddo
1172 !< test_passed = .false.
1173 !< outer_chr: do f=1, size(alist_chr, dim=1)
1174 !< do ff=1, Nf
1175 !< test_passed = alist_chr(f) == files(ff)
1176 !< if (test_passed) cycle outer_chr
1177 !< enddo
1178 !< enddo outer_chr
1179 !< print '(L1)', test_passed
1180 !<````
1181 !> T <<<
1182 class(string), intent(in) :: self !< The string.
1183 character(*), intent(in) :: pattern !< Given pattern.
1184 character(len=:), allocatable, intent(out) :: list(:) !< List of matching pathnames.
1185 type(string), allocatable :: list_(:) !< List of matching pathnames.

```

```

1186 integer(I4P) :: max_len !< Maximum length.
1187 integer(I4P) :: matches_number !< Matches number.
1188 integer(I4P) :: m !< Counter.
1189
1190 call self%glob(pattern=pattern, list=list_)
1191 if (allocated(list_)) then
1192     matches_number = size(list_, dim=1)
1193     max_len = 0
1194     do m=1, matches_number
1195         max_len = max(max_len, list_(m)%len())
1196     enddo
1197     allocate(character(max_len) :: list(1:matches_number))
1198     do m=1, matches_number
1199         list(m) = list_(m)%chars()
1200     enddo
1201 endif
1202 endsubroutine glob_character
1203
1204 subroutine glob_string(self, pattern, list)
1205     !< Glob search (string output), finds all the pathnames matching a given pattern according to the rules used by the Unix
1206     !<
1207     !< @note Method not portable: works only on Unix/GNU Linux OS.
1208     !<
1209     !< ``fortran
1210     !< type(string) :: astring
1211     !< type(string), allocatable :: alist_str(:)
1212     !< integer, parameter :: Nf=5
1213     !< character(14) :: files(1:Nf)
1214     !< integer :: file_unit
1215     !< integer :: f
1216     !< integer :: ff
1217     !< logical :: test_passed
1218     !<
1219     !< do f=1, Nf
1220     !<     files(f) = astring%tempname(prefix='foo-')
1221     !<     open(newunit=file_unit, file=files(f))
1222     !<     write(file_unit, *)f
1223     !<     close(unit=file_unit)
1224     !< enddo
1225     !< call astring%glob(pattern='foo-*', list=alist_str)
1226     !< do f=1, Nf
1227     !<     open(newunit=file_unit, file=files(f))
1228     !<     close(unit=file_unit, status='delete')
1229     !< enddo

```

```

1230 !< test_passed = .false.
1231 !< outer_str: do f=1, size(alist_str, dim=1)
1232 !<     do ff=1, Nf
1233 !<         test_passed = alist_str(f) == files(ff)
1234 !<         if (test_passed) cycle outer_str
1235 !<     enddo
1236 !< enddo outer_str
1237 !< print '(L1)', test_passed
1238 !< ``
1239 !> T <<<
1240 class(string),          intent(in)  :: self      !< The string.
1241 character(*),          intent(in)  :: pattern   !< Given pattern.
1242 type(string), allocatable, intent(out) :: list(:) !< List of matching pathnames.
1243 type(string)          :: tempfile !< Safe temporary file.
1244 character(len=:), allocatable      :: tempname !< Safe temporary name.
1245 integer(I4P)          :: tempunit  !< Unit of temporary file.
1246
1247 tempname = self%tempname()
1248 call execute_command_line('ls_1_1_//trim(adjustl(pattern))//'_>_//tempname)
1249 call tempfile%read_file(file=tempname)
1250 call tempfile%split(sep=new_line('a'), tokens=list)
1251 open(newunit=tempunit, file=tempname)
1252 close(unit=tempunit, status='delete')
1253 endsubroutine glob_string
1254
1255 elemental function insert_character(self, substring, pos) result(inserted)
1256 !< Insert substring into string at a specified position.
1257 !<
1258 !< ``fortran
1259 !< type(string)          :: astring
1260 !< character(len=:), allocatable :: acharacter
1261 !< logical              :: test_passed(5)
1262 !< astring = 'this is string example wow!!!'
1263 !< acharacter = '... '
1264 !< test_passed(1) = astring%insert(substring=acharacter, pos=1)//' '=='... this is string example wow!!!'
1265 !< test_passed(2) = astring%insert(substring=acharacter, pos=23)//' '=='this is string example... wow!!!'
1266 !< test_passed(3) = astring%insert(substring=acharacter, pos=29)//' '=='this is string example wow!!!... '
1267 !< test_passed(4) = astring%insert(substring=acharacter, pos=-1)//' '=='... this is string example wow!!!'
1268 !< test_passed(5) = astring%insert(substring=acharacter, pos=100)//' '=='this is string example wow!!!... '
1269 !< print '(L1)', all(test_passed)
1270 !< ``
1271 !> T <<<
1272 class(string),          intent(in)  :: self      !< The string.
1273 character(len=*), intent(in)  :: substring !< Substring.

```

```

1274 integer,          intent(in) :: pos           !< Position from which insert substring.
1275 type(string)      :: inserted        !< Inserted string.
1276 integer           :: safepos         !< Safe position from which insert substring.
1277
1278 if (allocated(self%raw)) then
1279     inserted = self
1280     safepos = min(max(1, pos), len(self%raw))
1281     if (safepos==1) then
1282         inserted%raw = substring//self%raw
1283     elseif (safepos==len(self%raw)) then
1284         inserted%raw = self%raw//substring
1285     else
1286         inserted%raw = self%raw(1:safepos-1)//substring//self%raw(safepos:)
1287     endif
1288 else
1289     inserted%raw = substring
1290 endif
1291 endfunction insert_character
1292
1293 elemental function insert_string(self, substring, pos) result(inserted)
1294 !< Insert substring into string at a specified position.
1295 !<
1296 !<````fortran
1297 !< type(string) :: astring
1298 !< type(string) :: anotherstring
1299 !< logical      :: test_passed(5)
1300 !< astring = 'this is string example wow!!!'
1301 !< anotherstring = '... '
1302 !< test_passed(1) = astring%insert(substring=anotherstring, pos=1)//'=='...' this is string example wow!!!'
1303 !< test_passed(2) = astring%insert(substring=anotherstring, pos=23)//'=='this is string example... wow!!!'
1304 !< test_passed(3) = astring%insert(substring=anotherstring, pos=29)//'=='this is string example wow!!!... '
1305 !< test_passed(4) = astring%insert(substring=anotherstring, pos=-1)//'=='... this is string example wow!!!'
1306 !< test_passed(5) = astring%insert(substring=anotherstring, pos=100)//'=='this is string example wow!!!... '
1307 !< print '(L1)', all(test_passed)
1308 !<````
1309 !=> T <<<
1310 class(string), intent(in) :: self           !< The string.
1311 type(string),  intent(in) :: substring      !< Substring.
1312 integer,       intent(in) :: pos           !< Position from which insert substring.
1313 type(string)   :: inserted        !< Inserted string.
1314 integer       :: safepos         !< Safe position from which insert substring.
1315
1316 if (allocated(self%raw)) then
1317     inserted = self

```

```

1318 if (allocated(substring%raw)) then
1319     safepos = min(max(1, pos), len(self%raw))
1320     if (safepos==1) then
1321         inserted%raw = substring%raw//self%raw
1322     elseif (safepos==len(self%raw)) then
1323         inserted%raw = self%raw//substring%raw
1324     else
1325         inserted%raw = self%raw(1:safepos-1)//substring%raw//self%raw(safepos:)
1326     endif
1327 endif
1328 else
1329     if (allocated(substring%raw)) inserted%raw = substring%raw
1330 endif
1331 endfunction insert_string
1332
1333 pure function join_strings(self, array, sep) result(join)
1334 !< Return a string that is a join of an array of strings.
1335 !<
1336 !< The join-separator is set equals to self if self has a value or it is set to a null string ''. This value can be overr
1337 !< passing a custom separator.
1338 !<
1339 !<````fortran
1340 !< type(string) :: astring
1341 !< type(string) :: strings(3)
1342 !< logical :: test_passed(5)
1343 !< strings(1) = 'one '
1344 !< strings(2) = 'two '
1345 !< strings(3) = 'three '
1346 !< test_passed(1) = (astring%join(array=strings)//' '==strings(1)//strings(2)//strings(3))
1347 !< test_passed(2) = (astring%join(array=strings, sep='-')//' '==strings(1)//'- '//strings(2)//'- '//strings(3))
1348 !< call strings(1)%free
1349 !< strings(2) = 'two '
1350 !< strings(3) = 'three '
1351 !< test_passed(3) = (astring%join(array=strings, sep='-')//' '==strings(2)//'- '//strings(3))
1352 !< strings(1) = 'one '
1353 !< strings(2) = 'two '
1354 !< call strings(3)%free
1355 !< test_passed(4) = (astring%join(array=strings, sep='-')//' '==strings(1)//'- '//strings(2))
1356 !< strings(1) = 'one '
1357 !< call strings(2)%free
1358 !< strings(3) = 'three '
1359 !< test_passed(5) = (astring%join(array=strings, sep='-')//' '==strings(1)//'- '//strings(3))
1360 !< print '(L1)', all(test_passed)
1361 !<````

```

```

1362 ! => T <<<
1363 class(string), intent(in) :: self !< The string.
1364 type(string), intent(in) :: array(1:) !< Array to be joined.
1365 character(kind=CK, len=*), intent(in), optional :: sep !< Separator.
1366 type(string) :: join !< The join of array.
1367 character(kind=CK, len=:), allocatable :: sep_ !< Separator, default value.
1368 integer :: a !< Counter.
1369
1370 if (allocated(self%raw)) then
1371     sep_ = self%raw
1372 else
1373     sep_ = ''
1374 endif
1375 if (present(sep)) sep_ = sep
1376 join = ''
1377 do a=2, size(array, dim=1)
1378     if (allocated(array(a)%raw)) join%raw = join%raw//sep_//array(a)%raw
1379 enddo
1380 if (allocated(array(1)%raw)) then
1381     join%raw = array(1)%raw//join%raw
1382 else
1383     join%raw = join%raw(len(sep_)+1:len(join%raw))
1384 endif
1385 endfunction join_strings
1386
1387 pure function join_characters(self, array, sep) result(join)
1388 !< Return a string that is a join of an array of characters.
1389 !<
1390 !< The join-separator is set equals to self if self has a value or it is set to a null string ''. This value can be overr
1391 !< passing a custom separator.
1392 !<
1393 !<````fortran
1394 !< type(string) :: astring
1395 !< character(5) :: characters(3)
1396 !< logical :: test_passed(6)
1397 !< characters(1) = 'one'
1398 !< characters(2) = 'two'
1399 !< characters(3) = 'three'
1400 !< test_passed(1) = (astring%join(array=characters)//' '==characters(1)//characters(2)//characters(3))
1401 !< test_passed(2) = (astring%join(array=characters, sep='-')//' '==characters(1)//'- '//characters(2)//'- '//characters(3))
1402 !< characters(1) = ''
1403 !< characters(2) = 'two'
1404 !< characters(3) = 'three'
1405 !< test_passed(3) = (astring%join(array=characters, sep='-')//' '==characters(2)//'- '//characters(3))

```



```

1406 !< characters(1) = 'one '
1407 !< characters(2) = 'two '
1408 !< characters(3) = ''
1409 !< test_passed(4) = (astring%join(array=characters, sep='-')//' '==characters(1)//'- '//characters(2))
1410 !< characters(1) = 'one '
1411 !< characters(2) = ''
1412 !< characters(3) = 'three '
1413 !< test_passed(5) = (astring%join(array=characters, sep='-')//' '==characters(1)//'- '//characters(3))
1414 !< characters(1) = 'one '
1415 !< characters(2) = 'two '
1416 !< characters(3) = 'three '
1417 !< astring = '_ '
1418 !< test_passed(6) = (astring%join(array=characters)//' '==characters(1)//'_ '//characters(2)//'_ '//characters(3))
1419 !< print '(L1)', all(test_passed)
1420 !<````
1421 !> T <<<
1422 class(string), intent(in) :: self !< The string.
1423 character(kind=CK, len=*), intent(in) :: array(1:) !< Array to be joined.
1424 character(kind=CK, len=*), intent(in), optional :: sep !< Separator.
1425 type(string) :: join !< The join of array.
1426 character(kind=CK, len=:), allocatable :: sep_ !< Separator, default value.
1427 integer :: a !< Counter.
1428
1429 if (allocated(self%raw)) then
1430     sep_ = self%raw
1431 else
1432     sep_ = ''
1433 endif
1434 if (present(sep)) sep_ = sep
1435 join = ''
1436 do a=2, size(array, dim=1)
1437     if (array(a)/='') join%raw = join%raw//sep_//array(a)
1438 enddo
1439 if (array(1)/='') then
1440     join%raw = array(1)//join%raw
1441 else
1442     join%raw = join%raw(len(sep_)+1:len(join%raw))
1443 endif
1444 endfunction join_characters
1445
1446 elemental function lower(self)
1447 !< Return a string with all lowercase characters.
1448 !<
1449 !<````fortran

```

```

1450 !< type(string) :: astring
1451 !< logical      :: test_passed(1)
1452 !< astring = 'Hello WorLD!'
1453 !< test_passed(1) = astring%lower()/' '=='hello world!'
1454 !< print '(L1)', all(test_passed)
1455 !<````
1456 !> T <<<
1457 class(string), intent(in) :: self !< The string.
1458 type(string)      :: lower !< Upper case string.
1459 integer           :: n1 !< Characters counter.
1460 integer           :: n2 !< Characters counter.
1461
1462 if (allocated(self%raw)) then
1463     lower = self
1464     do n1=1, len(self%raw)
1465         n2 = index(UPPER_ALPHABET, self%raw(n1:n1))
1466         if (n2>0) lower%raw(n1:n1) = LOWER_ALPHABET(n2:n2)
1467     enddo
1468 endif
1469 endfunction lower
1470
1471 pure function partition(self, sep) result(partitions)
1472 !< Split string at separator and return the 3 parts (before, the separator and after).
1473 !<
1474 !<````fortran
1475 !< type(string) :: astring
1476 !< type(string) :: strings(3)
1477 !< logical      :: test_passed(3)
1478 !< astring = 'Hello WorLD!'
1479 !< strings = astring%partition(sep='lo Wo ')
1480 !< test_passed(1) = (strings(1)/' '=='Hel '.and.strings(2)/' '=='lo Wo '.and.strings(3)/' '=='rLD!')
1481 !< strings = astring%partition(sep='Hello ')
1482 !< test_passed(2) = (strings(1)/' '==' '.and.strings(2)/' '=='Hello '.and.strings(3)/' '==' WorLD!')
1483 !< astring = 'Hello WorLD!'
1484 !< strings = astring%partition()
1485 !< test_passed(3) = (strings(1)/' '=='Hello '.and.strings(2)/' '==' '.and.strings(3)/' '==' WorLD!')
1486 !< print '(L1)', all(test_passed)
1487 !<````
1488 !> T <<<
1489 class(string),           intent(in)           :: self !< The string.
1490 character(kind=CK, len=*), intent(in), optional :: sep !< Separator.
1491 type(string)           :: partitions(1:3) !< Partions: before the separator, the separator itself
1492                                     !< after the separator.
1493 character(kind=CK, len=:), allocatable       :: sep_ !< Separator, default value.

```

```

1494 integer :: c !< Character counter.
1495 #ifdef __GFORTRAN__
1496 character(kind=CK, len=:), allocatable :: temporary !< Temporary storage, workaround for GNU bug.
1497 #endif
1498
1499 if (allocated(self%raw)) then
1500     sep_ = SPACE ; if (present(sep)) sep_ = sep
1501
1502     partitions(1) = self
1503     partitions(2) = sep_
1504     partitions(3) = ''
1505     if (len(sep_)>=len(self%raw)) return
1506     c = index(self%raw, sep_)
1507     if (c>0) then
1508 #ifdef __GFORTRAN__
1509         temporary = self%raw
1510         partitions(1)%raw = temporary(1:c-1)
1511         partitions(2)%raw = temporary(c:c+len(sep_)-1)
1512         partitions(3)%raw = temporary(c+len(sep_):)
1513 #else
1514         partitions(1)%raw = self%raw(1:c-1)
1515         partitions(2)%raw = self%raw(c:c+len(sep_)-1)
1516         partitions(3)%raw = self%raw(c+len(sep_):)
1517 #endif
1518     endif
1519 endif
1520 endfunction partition
1521
1522 subroutine read_file(self, file, is_fast, form, iostat, iomsg)
1523 !< Read a file as a single string stream.
1524 !<
1525 !< @note All the lines are stored into the string self as a single ascii stream. Each line (record) is separated by a `ne
1526 !< character.
1527 !<
1528 !< @note For unformatted read only `access='stream'` is supported with new_line as line terminator.
1529 !<
1530 !< @note *Fast* file reading allows a very efficient reading of streamed file, but it dumps file as single streamed strin
1531 !<
1532 !<``fortran
1533 !< type(string) :: astring
1534 !< type(string), allocatable :: strings(:)
1535 !< type(string) :: line(3)
1536 !< integer :: iostat
1537 !< character(len=99) :: iomsg

```

```

1538 !< integer :: scratch
1539 !< integer :: l
1540 !< logical :: test_passed(9)
1541 !< line(1) = ' Hello World! '
1542 !< line(2) = 'How are you? '
1543 !< line(3) = ' All say: "Fine thanks" '
1544 !< open(newunit=scratch, file='read_file_test.tmp')
1545 !< write(scratch, "(A)") line(1)%chars()
1546 !< write(scratch, "(A)") line(2)%chars()
1547 !< write(scratch, "(A)") line(3)%chars()
1548 !< close(scratch)
1549 !< call astring%read_file(file='read_file_test.tmp', iostat=iostat, iomsg=iomsg)
1550 !< call astring%split(tokens=strings, sep=new_line('a'))
1551 !< test_passed(1) = (size(strings, dim=1)==size(line, dim=1))
1552 !< do l=1, size(strings, dim=1)
1553 !< test_passed(l+1) = (strings(l)==line(l))
1554 !< enddo
1555 !< open(newunit=scratch, file='read_file_test.tmp', form='UNFORMATTED', access='STREAM')
1556 !< write(scratch) line(1)%chars()//new_line('a')
1557 !< write(scratch) line(2)%chars()//new_line('a')
1558 !< write(scratch) line(3)%chars()//new_line('a')
1559 !< close(scratch)
1560 !< call astring%read_file(file='read_file_test.tmp', form='unformatted', iostat=iostat, iomsg=iomsg)
1561 !< call astring%split(tokens=strings, sep=new_line('a'))
1562 !< test_passed(5) = (size(strings, dim=1)==size(line, dim=1))
1563 !< do l=1, size(strings, dim=1)
1564 !< test_passed(l+5) = (strings(l)==line(l))
1565 !< enddo
1566 !< open(newunit=scratch, file='read_file_test.tmp', form='UNFORMATTED', access='STREAM')
1567 !< close(scratch, status='DELETE')
1568 !< call astring%read_file(file='read_file_test.tmp', iostat=iostat)
1569 !< test_passed(9) = (iostat/=0)
1570 !< print '(L1)', all(test_passed)
1571 !< ``
1572 !=> T <<<
1573 class(string), intent(inout) :: self !< The string.
1574 character(len=*), intent(in) :: file !< File name.
1575 logical, intent(in), optional :: is_fast !< Flag to enable (super) fast file reading.
1576 character(len=*), intent(in), optional :: form !< Format of unit.
1577 integer, intent(out), optional :: iostat !< IO status code.
1578 character(len=*), intent(inout), optional :: iomsg !< IO status message.
1579 logical :: is_fast_ !< Flag to enable (super) fast file reading, local variable.
1580 type(string) :: form_ !< Format of unit, local variable.
1581 integer :: iostat_ !< IO status code, local variable.

```

```

1582 character(len=:), allocatable :: iomsg_ !< IO status message, local variable.
1583 integer :: unit !< Logical unit.
1584 logical :: does_exist !< Check if file exist.
1585 integer(I4P) :: filesize !< Size of the file for fast reading.
1586
1587 iomsg_ = repeat('␣', 99) ; if (present(iomsg)) iomsg_ = iomsg
1588 inquire(file=file, iomsg=iomsg_, iostat=iostat_, exist=does_exist)
1589 if (does_exist) then
1590   is_fast_ = .false. ; if (present(is_fast)) is_fast_ = is_fast
1591   if (is_fast_) then
1592     open(newunit=unit, file=file, access='STREAM', form='UNFORMATTED', iomsg=iomsg_, iostat=iostat_)
1593     inquire(file=file, size=filesize)
1594     if (allocated(self%raw)) deallocate(self%raw)
1595     allocate(character(len=filesize):: self%raw)
1596     read(unit=unit, iostat=iostat_, iomsg=iomsg_) self%raw
1597     close(unit)
1598   else
1599     form_ = 'FORMATTED' ; if (present(form)) form_ = form ; form_ = form_%upper()
1600     select case(form_%chars())
1601     case('FORMATTED')
1602       open(newunit=unit, file=file, status='OLD', action='READ', iomsg=iomsg_, iostat=iostat_, err=10)
1603     case('UNFORMATTED')
1604       open(newunit=unit, file=file, status='OLD', action='READ', form='UNFORMATTED', access='STREAM', &
1605         iomsg=iomsg_, iostat=iostat_, err=10)
1606     endselect
1607     call self%read_lines(unit=unit, form=form, iomsg=iomsg_, iostat=iostat_)
1608     10 close(unit)
1609   endif
1610 else
1611   iostat_ = 1
1612   iomsg_ = 'file␣not␣found'
1613 endif
1614 if (present(iostat)) iostat = iostat_
1615 if (present(iomsg)) iomsg = iomsg_
1616 endsubroutine read_file
1617
1618 subroutine read_line(self, unit, form, iostat, iomsg)
1619 !< Read line (record) from a connected unit.
1620 !<
1621 !< The line is read as an ascii stream read until the eor is reached.
1622 !<
1623 !< @note For unformatted read only `access='stream'` is supported with new_line as line terminator.
1624 !<
1625 !<````fortran

```

```

1626 !< type(string)      :: astring
1627 !< type(string)      :: line(3)
1628 !< integer           :: iostat
1629 !< character(len=99) :: iomsg
1630 !< integer           :: scratch
1631 !< integer           :: l
1632 !< logical            :: test_passed(6)
1633 !< line(1) = ' Hello World! '
1634 !< line(2) = 'How are you? '
1635 !< line(3) = ' All say: "Fine thanks" '
1636 !< open(newunit=scratch, status='SCRATCH')
1637 !< write(scratch, "(A)") line(1)%chars()
1638 !< write(scratch, "(A)") line(2)%chars()
1639 !< write(scratch, "(A)") line(3)%chars()
1640 !< rewind(scratch)
1641 !< l = 0
1642 !< iostat = 0
1643 !< do
1644 !<   l = l + 1
1645 !<   call astring%read_line(unit=scratch, iostat=iostat, iomsg=iomsg)
1646 !<   if (iostat/=0.and..not.is_iostat_eor(iostat)) then
1647 !<     exit
1648 !<   else
1649 !<     test_passed(l) = (astring==line(l))
1650 !<   endif
1651 !< enddo
1652 !< close(scratch)
1653 !< open(newunit=scratch, status='SCRATCH', form='UNFORMATTED', access='STREAM')
1654 !< write(scratch) line(1)%chars()//new_line('a')
1655 !< write(scratch) line(2)%chars()//new_line('a')
1656 !< write(scratch) line(3)%chars()//new_line('a')
1657 !< rewind(scratch)
1658 !< l = 0
1659 !< iostat = 0
1660 !< do
1661 !<   l = l + 1
1662 !<   call astring%read_line(unit=scratch, iostat=iostat, iomsg=iomsg, form='UnfORMatted')
1663 !<   if (iostat/=0.and..not.is_iostat_eor(iostat)) then
1664 !<     exit
1665 !<   else
1666 !<     test_passed(l+3) = (astring==line(l))
1667 !<   endif
1668 !< enddo
1669 !< close(scratch)

```

```

1670 !< print '(L1)', all(test_passed)
1671 !< ``
1672 !=> T <<<
1673 class(string), intent(inout) :: self !< The string.
1674 integer, intent(in) :: unit !< Logical unit.
1675 character(len=*), intent(in), optional :: form !< Format of unit.
1676 integer, intent(out), optional :: iostat !< IO status code.
1677 character(len=*), intent(inout), optional :: iomsg !< IO status message.
1678 type(string) :: form_ !< Format of unit, local variable.
1679 integer :: iostat_ !< IO status code, local variable.
1680 character(len=:), allocatable :: iomsg_ !< IO status message, local variable.
1681 character(kind=CK, len=:), allocatable :: line !< Line storage.
1682 character(kind=CK, len=1) :: ch !< Character storage.
1683
1684 form_ = 'FORMATTED' ; if (present(form)) form_ = form ; form_ = form_%upper()
1685 iomsg_ = repeat(' ', 99) ; if (present(iomsg)) iomsg_ = iomsg
1686 line = ''
1687 select case(form_%chars())
1688 case('FORMATTED')
1689 do
1690 read(unit, "(A)", advance='no', iostat=iostat_, iomsg=iomsg_, err=10, end=10, eor=10) ch
1691 line = line//ch
1692 enddo
1693 case('UNFORMATTED')
1694 do
1695 read(unit, iostat=iostat_, iomsg=iomsg_, err=10, end=10) ch
1696 if (ch==new_line('a')) then
1697 iostat_ = iostat_eor
1698 exit
1699 endif
1700 line = line//ch
1701 enddo
1702 endselect
1703 10 if (line/='') self%raw = line
1704 if (present(iostat)) iostat = iostat_
1705 if (present(iomsg)) iomsg = iomsg_
1706 endsubroutine read_line
1707
1708 subroutine read_lines(self, unit, form, iostat, iomsg)
1709 !< Read (all) lines (records) from a connected unit as a single ascii stream.
1710 !<
1711 !< @note All the lines are stored into the string self as a single ascii stream. Each line (record) is separated by a `ne
1712 !< character. The line is read as an ascii stream read until the eor is reached.
1713 !<

```

```

1714 !< @note The connected unit is rewinded. At a successful exit current record is at eof, at the beginning otherwise.
1715 !<
1716 !< @note For unformatted read only `access='stream'` is supported with new_line as line terminator.
1717 !<
1718 !<````fortran
1719 !< type(string) :: astring
1720 !< type(string), allocatable :: strings(:)
1721 !< type(string) :: line(3)
1722 !< integer :: iostat
1723 !< character(len=99) :: iomsg
1724 !< integer :: scratch
1725 !< integer :: l
1726 !< logical :: test_passed(8)
1727 !<
1728 !< line(1) = ' Hello World! '
1729 !< line(2) = 'How are you? '
1730 !< line(3) = ' All say: "Fine thanks" '
1731 !< open(newunit=scratch, status='SCRATCH')
1732 !< write(scratch, "(A)") line(1)%chars()
1733 !< write(scratch, "(A)") line(2)%chars()
1734 !< write(scratch, "(A)") line(3)%chars()
1735 !< call astring%read_lines(unit=scratch, iostat=iostat, iomsg=iomsg)
1736 !< call astring%split(tokens=strings, sep=new_line('a'))
1737 !< test_passed(1) = (size(strings, dim=1)==size(line, dim=1))
1738 !< do l=1, size(strings, dim=1)
1739 !< test_passed(l+1) = (strings(l)==line(l))
1740 !< enddo
1741 !< close(scratch)
1742 !< open(newunit=scratch, status='SCRATCH', form='UNFORMATTED', access='STREAM')
1743 !< write(scratch) line(1)%chars()//new_line('a')
1744 !< write(scratch) line(2)%chars()//new_line('a')
1745 !< write(scratch) line(3)%chars()//new_line('a')
1746 !< call astring%read_lines(unit=scratch, form='unformatted', iostat=iostat, iomsg=iomsg)
1747 !< call astring%split(tokens=strings, sep=new_line('a'))
1748 !< test_passed(5) = (size(strings, dim=1)==size(line, dim=1))
1749 !< do l=1, size(strings, dim=1)
1750 !< test_passed(l+5) = (strings(l)==line(l))
1751 !< enddo
1752 !< close(scratch)
1753 !< print '(L1)', all(test_passed)
1754 !<````
1755 !=> T <<<
1756 class(string), intent(inout) :: self !< The string.
1757 integer, intent(in) :: unit !< Logical unit.

```



```

1758 character(len=*), intent(in), optional :: form !< Format of unit.
1759 integer, intent(out), optional :: iostat !< IO status code.
1760 character(len=*), intent(inout), optional :: iomsg !< IO status message.
1761 integer :: iostat_ !< IO status code, local variable.
1762 character(len=:), allocatable :: iomsg_ !< IO status message, local variable.
1763 type(string) :: lines !< Lines storage.
1764 type(string) :: line !< Line storage.
1765
1766 iomsg_ = repeat(' ', 99) ; if (present(iomsg)) iomsg_ = iomsg
1767 rewind(unit)
1768 iostat_ = 0
1769 lines%raw = ''
1770 do
1771 line%raw = ''
1772 call line%read_line(unit=unit, form=form, iostat=iostat_, iomsg=iomsg_)
1773 if (iostat_ /= 0 .and..not.is_iostat_eor(iostat_)) then
1774 exit
1775 elseif (line /= '') then
1776 lines%raw = lines%raw//line%raw//new_line('a')
1777 endif
1778 enddo
1779 if (lines%raw /= '') self%raw = lines%raw
1780 if (present(iostat)) iostat = iostat_
1781 if (present(iomsg)) iomsg = iomsg_
1782 endsubroutine read_lines
1783
1784 elemental function replace(self, old, new, count) result(replaced)
1785 !< Return a string with all occurrences of substring old replaced by new.
1786 !<
1787 !<````fortran
1788 !< type(string) :: astring
1789 !< logical :: test_passed(3)
1790 !< astring = 'When YOU are sad YOU should think to me :-)'
1791 !< test_passed(1) = (astring%replace(old='YOU', new='THEY'))//''=='When THEY are sad THEY should think to me :-)'
1792 !< test_passed(2) = (astring%replace(old='YOU', new='THEY', count=1))//''=='When THEY are sad YOU should think to me :-)'
1793 !< astring = repeat(new_line('a')//'abcd', 20)
1794 !< astring = astring%replace(old=new_line('a'), new='|cr|')
1795 !< astring = astring%replace(old='|cr|', new=new_line('a')//''')
1796 !< test_passed(3) = (astring//''==repeat(new_line('a')//''//'abcd', 20))
1797 !< print '(L1)', all(test_passed)
1798 !<````
1799 !> T <<<
1800 class(string), intent(in) :: self !< The string.
1801 character(kind=CK, len=*), intent(in) :: old !< Old substring.

```

```

1802 character(kind=CK, len=*), intent(in) :: new !< New substring.
1803 integer, intent(in), optional :: count !< Number of old occurrences to be replaced.
1804 type(string) :: replaced !< The string with old replaced by new.
1805 integer :: r !< Counter.
1806
1807 if (allocated(self%raw)) then
1808     replaced = self
1809     r = 0
1810     do
1811         if (index(replaced%raw, old)>0) then
1812             replaced = replaced%replace_one_occurrence(old=old, new=new)
1813             r = r + 1
1814             if (present(count)) then
1815                 if (r>=count) exit
1816             endif
1817         else
1818             exit
1819         endif
1820     enddo
1821 endif
1822 endfunction replace
1823
1824 elemental function reverse(self) result(reversed)
1825 !< Return a reversed string.
1826 !<
1827 !<````fortran
1828 !< type(string) :: astring
1829 !< logical :: test_passed(2)
1830 !< astring = 'abcdefghijklmnopqrstuvz '
1831 !< test_passed(1) = (astring%reverse()// ' '== 'zvutsrqponmlihgfedcba ')
1832 !< astring = '0123456789 '
1833 !< test_passed(2) = (astring%reverse()// ' '== '9876543210 ')
1834 !< print '(L1)', all(test_passed)
1835 !<````
1836 !=> T <<<
1837 class(string), intent(in) :: self !< The string.
1838 type(string) :: reversed !< The reversed string.
1839 integer :: length !< Length of the string.
1840 integer :: c !< Counter.
1841
1842 if (allocated(self%raw)) then
1843     reversed = self
1844     length = len(self%raw)
1845     do c=1, length

```

```

1846         reversed%raw(c:c) = self%raw(length-c+1:length-c+1)
1847     enddo
1848 endif
1849 endfunction reverse
1850
1851 function search(self, tag_start, tag_end, in_string, in_character, istart, iend) result(tag)
1852     !< Search for *tagged* record into string, return the first record found (if any) matching the tags.
1853     !<
1854     !< Optionally, returns the indexes of tag start/end, thus this is not an `elemental` function.
1855     !<
1856     !< @note The tagged record is searched into self if allocated otherwise into `in_string` if passed or, eventually, into
1857     !< `in_character` is passed. If tag is not found the return string is not allocated and the start/end indexes (if request
1858     !< zero.
1859     !<
1860     !<````fortran
1861     !< type(string)                :: astring
1862     !< type(string)                :: anotherstring
1863     !< character(len=:), allocatable :: acharacter
1864     !< integer                    :: istart
1865     !< integer                    :: iend
1866     !< logical                    :: test_passed(5)
1867     !< astring = '<test> <first> hello </first> <first> not the first </first> </test>'
1868     !< anotherstring = astring%search(tag_start='<first>', tag_end='</first>')
1869     !< test_passed(1) = anotherstring//''=='<first> hello </first>'
1870     !< astring = '<test> <a> <a> <a> the nested a </a> </a> </a> </test>'
1871     !< anotherstring = astring%search(tag_start='<a>', tag_end='</a>')
1872     !< test_passed(2) = anotherstring//''=='<a> <a> <a> the nested a </a> </a> </a>'
1873     !< call astring%free
1874     !< anotherstring = '<test> <a> <a> <a> the nested a </a> </a> </a> </test>'
1875     !< astring = astring%search(in_string=anotherstring, tag_start='<a>', tag_end='</a>')
1876     !< test_passed(3) = astring//''=='<a> <a> <a> the nested a </a> </a> </a>'
1877     !< call astring%free
1878     !< acharacter = '<test> <a> <a> <a> the nested a </a> </a> </a> </test>'
1879     !< astring = astring%search(in_character=acharacter, tag_start='<a>', tag_end='</a>')
1880     !< test_passed(4) = astring//''=='<a> <a> <a> the nested a </a> </a> </a>'
1881     !< acharacter = '<test> <first> hello </first> <sec> <sec>not the first</sec> </sec> </test>'
1882     !< astring = astring%search(in_character=acharacter, tag_start='<sec>', tag_end='</sec>', istart=istart, iend=iend)
1883     !< test_passed(5) = astring//''==acharacter(31:67)
1884     !< print '(L1)', all(test_passed)
1885     !<````
1886     !> T <<<
1887     class(string),          intent(in)                :: self          !< The string.
1888     character(kind=CK, len=*), intent(in)            :: tag_start     !< Start tag.
1889     character(kind=CK, len=*), intent(in)            :: tag_end       !< End tag.

```

```

1890 type(string), intent(in), optional :: in_string !< Search into this string.
1891 character(kind=CK, len=*), intent(in), optional :: in_character !< Search into this character string.
1892 integer, intent(out), optional :: istart !< Starting index of tag inside the string.
1893 integer, intent(out), optional :: iend !< Ending index of tag inside the string.
1894 type(string) :: tag !< First tag found.
1895 character(kind=CK, len=:), allocatable :: raw !< Raw string into which search the tag.
1896 integer :: istart_ !< Starting index of tag inside the string, local variable
1897 integer :: iend_ !< Ending index of tag inside the string, local variable
1898 integer :: nested_tags !< Number of nested tags inside tag.
1899 integer :: t !< Counter.
1900
1901 raw = ''
1902 if (present(in_string)) then
1903     raw = in_string%raw
1904 elseif (present(in_character)) then
1905     raw = in_character
1906 else
1907     if (allocated(self%raw)) raw = self%raw
1908 endif
1909 istart_ = 0
1910 iend_ = 0
1911 if (raw/= '') then
1912     istart_ = index(raw, tag_start)
1913     iend_ = index(raw, tag_end)
1914     if (istart_>0.and.iend_>0) then
1915         iend_ = iend_ + len(tag_end) - 1
1916         tag%raw = raw(istart_:iend_)
1917         nested_tags = tag%count(tag_start)
1918         if (nested_tags>1) then
1919             do t=2, nested_tags
1920                 iend_ = iend_ + len(tag_end) - 1 + index(raw(iend_+1:), tag_end)
1921             enddo
1922             tag%raw = raw(istart_:iend_)
1923         endif
1924     endif
1925 endif
1926 if (present(istart)) istart = istart_
1927 if (present(iend)) iend = iend_
1928 endfunction search
1929
1930 pure function slice(self, istart, iend) result(raw)
1931 !< Return the raw characters data sliced.
1932 !<
1933 !<````fortran

```

```

1934 !< type(string) :: astring
1935 !< astring = 'the Quick Brown fox Jumps over the Lazy Dog.'
1936 !< print "(A)", astring%slice(11,25)
1937 !<```
1938 !> Brown fox Jumps <<<
1939 class(string), intent(in) :: self !< The string.
1940 integer, intent(in) :: istart !< Slice start index.
1941 integer, intent(in) :: iend !< Slice end index.
1942 character(kind=CK, len=:), allocatable :: raw !< Raw characters data.
1943
1944 if (allocated(self%raw)) then
1945     raw = self%raw(istart:iend)
1946 else
1947     raw = ''
1948 endif
1949 endfunction slice
1950
1951 elemental function snakecase(self, sep)
1952 !< Return a string with all words lowercase separated by "_".
1953 !<
1954 !< @note Multiple subsequent separators are collapsed to one occurrence.
1955 !<
1956 !<```fortran
1957 !< type(string) :: astring
1958 !< logical :: test_passed(1)
1959 !< astring = 'the Quick Brown fox Jumps over the Lazy Dog.'
1960 !< test_passed(1) = astring%snakecase()/' '=='the_quick_brown_fox_jumps_over_the_lazy_dog.'
1961 !< print '(L1)', all(test_passed)
1962 !<```
1963 !> T <<<
1964 class(string), intent(in) :: self !< The string.
1965 character(kind=CK, len=*), intent(in), optional :: sep !< Separator.
1966 type(string) :: snakecase !< Snake case string.
1967 type(string), allocatable :: tokens(:) !< String tokens.
1968
1969 if (allocated(self%raw)) then
1970     call self%split(tokens=tokens, sep=sep)
1971     tokens = tokens%lower()
1972     snakecase = snakecase%join(array=tokens, sep='_')
1973 endif
1974 endfunction snakecase
1975
1976 pure subroutine split(self, tokens, sep, max_tokens)
1977 !< Return a list of substring in the string, using sep as the delimiter string.

```

```

1978 !<
1979 !< @note Multiple subsequent separators are collapsed to one occurrence.
1980 !<
1981 !< @note If `max_tokens` is passed the returned number of tokens is either `max_tokens` or `max_tokens + 1`.
1982 !<
1983 !< ``fortran
1984 !< type(string) :: astring
1985 !< type(string), allocatable :: strings(:)
1986 !< logical :: test_passed(11)
1987 !< astring = '+ab-++cre-++cre-ab+'
1988 !< call astring%split(tokens=strings, sep='+')
1989 !< test_passed(1) = (strings(1)//' '=='ab-'.and.strings(2)//' '=='cre-'.and.strings(3)//' '=='cre-ab')
1990 !< astring = 'ab-++cre-++cre-ab+'
1991 !< call astring%split(tokens=strings, sep='+')
1992 !< test_passed(2) = (strings(1)//' '=='ab-'.and.strings(2)//' '=='cre-'.and.strings(3)//' '=='cre-ab')
1993 !< astring = 'ab-++cre-++cre-ab'
1994 !< call astring%split(tokens=strings, sep='+')
1995 !< test_passed(3) = (strings(1)//' '=='ab-'.and.strings(2)//' '=='cre-'.and.strings(3)//' '=='cre-ab')
1996 !< astring = 'Hello '//new_line('a')//'World!'
1997 !< call astring%split(tokens=strings, sep=new_line('a'))
1998 !< test_passed(4) = (strings(1)//' '=='Hello '.and.strings(2)//' '=='World!')
1999 !< astring = 'Hello World!'
2000 !< call astring%split(tokens=strings)
2001 !< test_passed(5) = (strings(1)//' '=='Hello '.and.strings(2)//' '=='World!')
2002 !< astring = '+ab-'
2003 !< call astring%split(tokens=strings, sep='+')
2004 !< test_passed(6) = (strings(1)//' '=='ab-')
2005 !< astring = '+ab-'
2006 !< call astring%split(tokens=strings, sep='-')
2007 !< test_passed(7) = (strings(1)//' '=='+ab')
2008 !< astring = '+ab-+cd-'
2009 !< call astring%split(tokens=strings, sep='+')
2010 !< test_passed(8) = (strings(1)//' '=='ab-'.and.strings(2)//' '=='cd-')
2011 !< astring = 'ab-+cd-+'
2012 !< call astring%split(tokens=strings, sep='+')
2013 !< test_passed(9) = (strings(1)//' '=='ab-'.and.strings(2)//' '=='cd-')
2014 !< astring = '+ab-+cd-+'
2015 !< call astring%split(tokens=strings, sep='+')
2016 !< test_passed(10) = (strings(1)//' '=='ab-'.and.strings(2)//' '=='cd-')
2017 !< astring = '1-2-3-4-5-6-7-8'
2018 !< call astring%split(tokens=strings, sep='-', max_tokens=3)
2019 !< test_passed(11) = (strings(1)//' '=='1'.and.strings(2)//' '=='2'.and.strings(3)//' '=='3'.and.strings(4)//' '=='4-5-6-7-8')
2020 !< print '(L1)', all(test_passed)
2021 !< ``

```

```

2022 ! => T <<<
2023 class(string), intent(in) :: self !< The string.
2024 type(string), allocatable, intent(out) :: tokens(:) !< Tokens substring.
2025 character(kind=CK, len=*), intent(in), optional :: sep !< Separator.
2026 integer, intent(in), optional :: max_tokens !< Fix the maximum number of returned tokens.
2027 character(kind=CK, len=:), allocatable :: sep_ !< Separator, default value.
2028 integer :: No !< Number of occurrences of sep.
2029 integer :: t !< Character counter.
2030 type(string) :: temporary !< Temporary storage.
2031 type(string), allocatable :: temp_toks(:, :) !< Temporary tokens substring.
2032
2033 if (allocated(self%raw)) then
2034     sep_ = SPACE ; if (present(sep)) sep_ = sep
2035
2036     temporary = self%unique(sep_)
2037     No = temporary%count(sep_)
2038
2039     if (No>0) then
2040         if (present(max_tokens)) then
2041             if (max_tokens < No.and.max_tokens > 0) No = max_tokens
2042         endif
2043         allocate(temp_toks(3, No))
2044         temp_toks(:, 1) = temporary%partition(sep_)
2045         if (No>1) then
2046             do t=2, No
2047                 temp_toks(:, t) = temp_toks(3, t-1)%partition(sep_)
2048             enddo
2049         endif
2050
2051         if (temp_toks(1, 1)%raw/=''.and.temp_toks(3, No)%raw/='') then
2052             allocate(tokens(No+1))
2053             do t=1, No
2054                 if (t==No) then
2055                     tokens(t) = temp_toks(1, t)
2056                     tokens(t+1) = temp_toks(3, t)
2057                 else
2058                     tokens(t) = temp_toks(1, t)
2059                 endif
2060             enddo
2061         elseif (temp_toks(1, 1)%raw/='') then
2062             allocate(tokens(No))
2063             do t=1, No
2064                 tokens(t) = temp_toks(1, t)
2065             enddo

```

```

2066     elseif (temp_toks(3, No)%raw/='') then
2067         allocate(tokens(No))
2068         do t=1, No-1
2069             tokens(t) = temp_toks(1, t+1)
2070         enddo
2071         tokens(No) = temp_toks(3, No)
2072     else
2073         allocate(tokens(No-1))
2074         do t=2, No
2075             tokens(t-1) = temp_toks(1, t)
2076         enddo
2077     endif
2078
2079     else
2080         allocate(tokens(1))
2081         tokens(1) = self
2082     endif
2083 endif
2084 endsubroutine split
2085
2086 pure subroutine split_chunked(self, tokens, chunks, sep)
2087     !< Return a list of substring in the string, using sep as the delimiter string, chunked (memory-efficient) algorithm.
2088     !<
2089     !< @note Multiple subsequent separators are collapsed to one occurrence.
2090     !<
2091     !< @note The split is performed in chunks of `#chunks` to avoid excessive memory consumption.
2092     !<
2093     !<````fortran
2094     !< type(string)                :: astring
2095     !< type(string), allocatable :: strings(:)
2096     !< logical                    :: test_passed(1)
2097     !< astring = '-1-2-3-4-5-6-7-8-'
2098     !< call astring%split_chunked(tokens=strings, sep='-', chunks=3)
2099     !< test_passed(1) = (strings(1)//' '=='1'.and.strings(2)//' '=='2'.and.strings(3)//' '=='3'.and.strings(4)//' '=='4'.and. &
2100     !<                      strings(5)//' '=='5'.and.strings(6)//' '=='6'.and.strings(7)//' '=='7'.and.strings(8)//' '=='8')
2101     !< print '(L1)', all(test_passed)
2102     !<````
2103     !> T <<<
2104     class(string),          intent(in)                :: self          !< The string.
2105     type(string), allocatable, intent(out)          :: tokens(:)    !< Tokens substring.
2106     integer,          intent(in)                :: chunks      !< Number of chunks.
2107     character(kind=CK, len=*), intent(in), optional :: sep          !< Separator.
2108     character(kind=CK, len=:), allocatable          :: sep_      !< Separator, default value.
2109     integer          :: Nt          !< Number of actual tokens.

```



```

2110 integer :: t !< Counter.
2111 logical :: isok
2112
2113 if (allocated(self%raw)) then
2114   sep_ = SPACE ; if (present(sep)) sep_ = sep
2115
2116   Nt = self%count(sep_)
2117   if (self%start_with(prefix=sep_)) Nt = Nt - 1
2118   if (self%end_with(suffix=sep_)) Nt = Nt - 1
2119   t = 0
2120   call self%split(tokens=tokens, sep=sep_, max_tokens=chunks)
2121   do
2122     t = size(tokens, dim=1)
2123     if (t > Nt) exit
2124     call split_last_token(tokens=tokens, max_tokens=chunks, isok=isok)
2125     if(isok)then
2126       else
2127         exit
2128       endif
2129     enddo
2130
2131   t = size(tokens, dim=1)
2132   if (tokens(t)%count(sep_) > 0) then
2133     call split_last_token(tokens=tokens, isok=isok)
2134   endif
2135 endif
2136
2137 contains
2138   pure subroutine split_last_token(tokens, max_tokens, isok)
2139     !< Split last token.
2140     type(string), allocatable, intent(inout) :: tokens(:) !< Tokens substring.
2141     integer, intent(in), optional :: max_tokens !< Max tokens returned.
2142     type(string), allocatable :: tokens_(:) !< Temporary tokens.
2143     type(string), allocatable :: tokens_swap(:) !< Swap tokens.
2144     integer :: Nt_ !< Number of last created tokens.
2145     logical, intent(out) :: isok
2146
2147     isok=.true.
2148     call tokens(t)%split(tokens=tokens_, sep=sep_, max_tokens=max_tokens)
2149     if (allocated(tokens_)) then
2150       Nt_ = size(tokens_, dim=1)
2151       if (Nt_ >= 1) then
2152         allocate(tokens_swap(1:t-1+Nt_))
2153         tokens_swap(1:t-1) = tokens(1:t-1)

```

```

2154         tokens_swap(t:)      = tokens_(:)
2155         call move_alloc(from=tokens_swap, to=tokens)
2156     endif
2157     if (Nt_ == 1) then
2158         isok=.false.
2159     end if
2160     deallocate(tokens_)
2161 endif
2162 endsubroutine split_last_token
2163 endsubroutine split_chunked
2164
2165 elemental function startcase(self, sep)
2166     !< Return a string with all words capitalized, e.g. title case.
2167     !<
2168     !< @note Multiple subsequent separators are collapsed to one occurence.
2169     !<
2170     !<````fortran
2171     !< type(string) :: astring
2172     !< logical      :: test_passed(1)
2173     !< astring = 'the Quick Brown fox Jumps over the Lazy Dog.'
2174     !< test_passed(1) = astring%startcase()/' '=='The Quick Brown Fox Jumps Over The Lazy Dog.'
2175     !< print '(L1)', all(test_passed)
2176     !<````
2177     !> T <<<
2178     class(string),          intent(in)          :: self          !< The string.
2179     character(kind=CK, len=*), intent(in), optional :: sep          !< Separator.
2180     type(string)           :: startcase        !< Start case string.
2181     character(kind=CK, len=:), allocatable         :: sep_         !< Separator, default value.
2182     type(string), allocatable         :: tokens(:) !< String tokens.
2183
2184     if (allocated(self%raw)) then
2185         sep_ = SPACE ; if (present(sep)) sep_ = sep
2186         call self%split(tokens=tokens, sep=sep_)
2187         tokens = tokens%capitalize()
2188         startcase = startcase%join(array=tokens, sep=sep_)
2189     endif
2190 endfunction startcase
2191
2192 elemental function strip(self, remove_nulls)
2193     !< Return a copy of the string with the leading and trailing characters removed.
2194     !<
2195     !< @note Multiple subsequent separators are collapsed to one occurence.
2196     !<
2197     !<````fortran

```

```

2198 !< type(string) :: astring
2199 !< logical      :: test_passed(1)
2200 !< astring = '   Hello World!   '
2201 !< test_passed(1) = astring%strip()/' '==' Hello World! '
2202 !< print '(L1)', all(test_passed)
2203 !< ````
2204 !> T <<<
2205 class(string), intent(in) :: self !< The string.
2206 logical, intent(in), optional :: remove_nulls !< Remove null characters at the end.
2207 type(string) :: strip !< The stripped string.
2208 integer :: c !< Counter.
2209
2210 if (allocated(self%raw)) then
2211   strip = self%adjustl()
2212   strip = strip%trim()
2213   if (present(remove_nulls)) then
2214     if (remove_nulls) then
2215       c = index(self%raw, char(0))
2216       if (c>0) strip%raw = strip%raw(1:c-1)
2217     endif
2218   endif
2219 endif
2220 endfunction strip
2221
2222 elemental function swapcase(self)
2223 !< Return a copy of the string with uppercase characters converted to lowercase and vice versa.
2224 !<
2225 !< ````fortran
2226 !< type(string) :: astring
2227 !< logical      :: test_passed(1)
2228 !< astring = '   Hello World!   '
2229 !< test_passed(1) = astring%swapcase()/' '==' hELLO wORLD! '
2230 !< print '(L1)', all(test_passed)
2231 !< ````
2232 !> T <<<
2233 class(string), intent(in) :: self !< The string.
2234 type(string) :: swapcase !< Upper case string.
2235 integer :: n1 !< Characters counter.
2236 integer :: n2 !< Characters counter.
2237
2238 if (allocated(self%raw)) then
2239   swapcase = self
2240   do n1=1, len(self%raw)
2241     n2 = index(UPPER_ALPHABET, self%raw(n1:n1))

```

```

2242         if (n2>0) then
2243             swapcase%raw(n1:n1) = LOWER_ALPHABET(n2:n2)
2244         else
2245             n2 = index(LOWER_ALPHABET, self%raw(n1:n1))
2246             if (n2>0) swapcase%raw(n1:n1) = UPPER_ALPHABET(n2:n2)
2247         endif
2248     enddo
2249 endif
2250 endfunction swapcase
2251
2252 function tempname(self, is_file, prefix, path)
2253     !< Return a safe temporary name suitable for temporary file or directories.
2254     !<
2255     !< ````fortran
2256     !< type(string) :: astring
2257     !< character(len=:), allocatable :: tmpname
2258     !< logical :: test_passed(5)
2259     !< tmpname = astring%tempname()
2260     !< inquire(file=tmpname, exist=test_passed(1))
2261     !< test_passed(1) = .not.test_passed(1)
2262     !< tmpname = astring%tempname(is_file=.false.)
2263     !< inquire(file=tmpname, exist=test_passed(2))
2264     !< test_passed(2) = .not.test_passed(2)
2265     !< tmpname = astring%tempname(path='./')
2266     !< inquire(file=tmpname, exist=test_passed(3))
2267     !< test_passed(3) = .not.test_passed(3)
2268     !< astring = 'me-'
2269     !< tmpname = astring%tempname()
2270     !< inquire(file=tmpname, exist=test_passed(4))
2271     !< test_passed(4) = .not.test_passed(4)
2272     !< tmpname = astring%tempname(prefix='you-')
2273     !< inquire(file=tmpname, exist=test_passed(5))
2274     !< test_passed(5) = .not.test_passed(5)
2275     !< print '(L1)', all(test_passed)
2276     !< ````
2277     !> T <<<
2278     class(string), intent(in) :: self
2279     logical, intent(in), optional :: is_file
2280     character(*), intent(in), optional :: prefix
2281     character(*), intent(in), optional :: path
2282     character(len=:), allocatable :: tmpname
2283     logical :: is_file_
2284     character(len=:), allocatable :: prefix_
2285     character(len=:), allocatable :: path_
2286
2287     !< The string.
2288     !< True if tempname should be used for file (the default).
2289     !< Name prefix, otherwise self is used (if allocated).
2290     !< Path where file/directory should be used, default './'.
2291     !< Safe (unique) temporary name.
2292     !< True if tempname should be used for file (the default).
2293     !< Name prefix, otherwise self is used (if allocated).
2294     !< Path where file/directory should be used, default './'.

```

```

2286 logical, save :: is_initialized=.false. !< Status of random seed initialization.
2287 real(R4P) :: random_real !< Random number (real).
2288 integer(I4P) :: random_integer !< Random number (integer).
2289 logical :: is_hold !< Flag to check if a safe tempname has been found.
2290
2291 is_file_ = .true. ; if (present(is_file)) is_file_ = is_file
2292 path_ = '' ; if (present(path)) path_ = path
2293 prefix_ = ''
2294 if (present(prefix)) then
2295     prefix_ = prefix
2296 elseif (allocated(self%raw)) then
2297     prefix_ = self%raw
2298 endif
2299 if (.not.is_initialized) then
2300     call random_seed
2301     is_initialized = .true.
2302 endif
2303 tempname = repeat('␣', len(path_) + len(prefix_) + 10) ! [path_] + [prefix_] + 6 random chars + [.tmp]
2304 do
2305     call random_number(random_real)
2306     random_integer = transfer(random_real, random_integer)
2307     random_integer = iand(random_integer, 16777215_I4P)
2308     if (is_file_) then
2309         write(tempname, '(A,Z6.6,A)') path_//prefix_, random_integer, '.tmp'
2310     else
2311         write(tempname, '(A,Z6.6)') path_//prefix_, random_integer
2312         tempname = trim(tempname)
2313     endif
2314     inquire(file=tempname, exist=is_hold)
2315     if (.not.is_hold) exit
2316 enddo
2317 endfunction tempname
2318
2319 elemental function to_integer_I1P(self, kind) result(to_number)
2320 !< Cast string to integer (I1P).
2321 !<
2322 !<````fortran
2323 !< use penf
2324 !< type(string) :: astring
2325 !< integer(I1P) :: integer_
2326 !< logical :: test_passed(1)
2327 !< astring = '127'
2328 !< integer_ = astring%to_number(kind=1_I1P)
2329 !< test_passed(1) = integer_==127_I1P

```

```

2330 !< print '(L1)', all(test_passed)
2331 !<````
2332 !> T <<<
2333 class(string), intent(in) :: self !< The string.
2334 integer(I1P), intent(in) :: kind !< Mold parameter for kind detection.
2335 integer(I1P) :: to_number !< The number into the string.
2336
2337 if (allocated(self%raw)) then
2338   if (self%is_integer()) read(self%raw, *) to_number
2339 endif
2340 endfunction to_integer_I1P
2341
2342 elemental function to_integer_I2P(self, kind) result(to_number)
2343 !< Cast string to integer (I2P).
2344 !<
2345 !<````fortran
2346 !< use penf
2347 !< type(string) :: astring
2348 !< integer(I2P) :: integer_
2349 !< logical :: test_passed(1)
2350 !< astring = '127'
2351 !< integer_ = astring%to_number(kind=1_I2P)
2352 !< test_passed(1) = integer_==127_I2P
2353 !< print '(L1)', all(test_passed)
2354 !<````
2355 !> T <<<
2356 class(string), intent(in) :: self !< The string.
2357 integer(I2P), intent(in) :: kind !< Mold parameter for kind detection.
2358 integer(I2P) :: to_number !< The number into the string.
2359
2360 if (allocated(self%raw)) then
2361   if (self%is_integer()) read(self%raw, *) to_number
2362 endif
2363 endfunction to_integer_I2P
2364
2365 elemental function to_integer_I4P(self, kind) result(to_number)
2366 !< Cast string to integer (I4P).
2367 !<
2368 !<````fortran
2369 !< use penf
2370 !< type(string) :: astring
2371 !< integer(I4P) :: integer_
2372 !< logical :: test_passed(1)
2373 !< astring = '127'

```

```

2374 !< integer_ = astring%to_number(kind=1_I4P)
2375 !< test_passed(1) = integer_==127_I4P
2376 !< print '(L1)', all(test_passed)
2377 !<````
2378 !> T <<<
2379 class(string), intent(in) :: self !< The string.
2380 integer(I4P), intent(in) :: kind !< Mold parameter for kind detection.
2381 integer(I4P) :: to_number !< The number into the string.
2382
2383 if (allocated(self%raw)) then
2384   if (self%is_integer()) read(self%raw, *) to_number
2385 endif
2386 endfunction to_integer_I4P
2387
2388 elemental function to_integer_I8P(self, kind) result(to_number)
2389 !< Cast string to integer (I8P).
2390 !<
2391 !<````fortran
2392 !< use penf
2393 !< type(string) :: astring
2394 !< integer(I8P) :: integer_
2395 !< logical :: test_passed(1)
2396 !< astring = '127'
2397 !< integer_ = astring%to_number(kind=1_I8P)
2398 !< test_passed(1) = integer_==127_I8P
2399 !< print '(L1)', all(test_passed)
2400 !<````
2401 !> T <<<
2402 class(string), intent(in) :: self !< The string.
2403 integer(I8P), intent(in) :: kind !< Mold parameter for kind detection.
2404 integer(I8P) :: to_number !< The number into the string.
2405
2406 if (allocated(self%raw)) then
2407   if (self%is_integer()) read(self%raw, *) to_number
2408 endif
2409 endfunction to_integer_I8P
2410
2411 elemental function to_real_R4P(self, kind) result(to_number)
2412 !< Cast string to real (R4P).
2413 !<
2414 !<````fortran
2415 !< use penf
2416 !< type(string) :: astring
2417 !< real(R4P) :: real_

```

```

2418      !< logical          :: test_passed(1)
2419      !< astring = '3.4e9'
2420      !< real_ = astring%to_number(kind=1._R4P)
2421      !< test_passed(1) = real_==3.4e9_R4P
2422      !< print '(L1)', all(test_passed)
2423      !<````
2424      !> T <<<
2425      class(string), intent(in) :: self          !< The string.
2426      real(R4P),      intent(in) :: kind          !< Mold parameter for kind detection.
2427      real(R4P)          :: to_number !< The number into the string.
2428
2429      if (allocated(self%raw)) then
2430          if (self%is_real()) read(self%raw, *) to_number
2431      endif
2432      endfunction to_real_R4P
2433
2434      elemental function to_real_R8P(self, kind) result(to_number)
2435      !< Cast string to real (R8P).
2436      !<
2437      !<````fortran
2438      !< use penf
2439      !< type(string) :: astring
2440      !< real(R8P)      :: real_
2441      !< logical          :: test_passed(1)
2442      !< astring = '3.4e9'
2443      !< real_ = astring%to_number(kind=1._R8P)
2444      !< test_passed(1) = real_==3.4e9_R8P
2445      !< print '(L1)', all(test_passed)
2446      !<````
2447      !> T <<<
2448      class(string), intent(in) :: self          !< The string.
2449      real(R8P),      intent(in) :: kind          !< Mold parameter for kind detection.
2450      real(R8P)          :: to_number !< The number into the string.
2451
2452      if (allocated(self%raw)) then
2453          if (self%is_real()) read(self%raw, *) to_number
2454      endif
2455      endfunction to_real_R8P
2456
2457      elemental function to_real_R16P(self, kind) result(to_number)
2458      !< Cast string to real (R16P).
2459      !<
2460      !<````fortran
2461      !< use penf

```



```

2462 !< type(string) :: astring
2463 !< real(R16P) :: real_
2464 !< logical :: test_passed(1)
2465 !< astring = '3.4e9'
2466 !< real_ = astring%to_number(kind=1._R16P)
2467 !< test_passed(1) = real_==3.4e9_R16P
2468 !< print '(L1)', all(test_passed)
2469 !<````
2470 !> T <<<
2471 class(string), intent(in) :: self !< The string.
2472 real(R16P), intent(in) :: kind !< Mold parameter for kind detection.
2473 real(R16P) :: to_number !< The number into the string.
2474
2475 if (allocated(self%raw)) then
2476 if (self%is_real()) read(self%raw, *) to_number
2477 endif
2478 endfunction to_real_R16P
2479
2480 elemental function unescape(self, to_unescape, unesc) result(unescaped)
2481 !< Unescape double backslashes (or custom escaped character).
2482 !<
2483 !<````fortran
2484 !< type(string) :: astring
2485 !< logical :: test_passed(2)
2486 !< astring = '^\\s \\d+\\s*'
2487 !< test_passed(1) = (astring%unescape(to_unescape='\\')/' '=='^\\s \\d+\\s*')
2488 !< test_passed(2) = (astring%unescape(to_unescape='s')/' '=='^\\s \\d+\\s*')
2489 !< print '(L1)', all(test_passed)
2490 !<````
2491 !> T <<<
2492 class(string), intent(in) :: self !< The string.
2493 character(kind=CK, len=1), intent(in) :: to_unescape !< Character to be unescaped.
2494 character(kind=CK, len=*), intent(in), optional :: unesc !< Character used to unescape.
2495 type(string) :: unescaped !< Escaped string.
2496 character(kind=CK, len=:), allocatable :: unesc_ !< Character to unescape, local variable.
2497 integer :: c !< Character counter.
2498
2499 if (allocated(self%raw)) then
2500 unesc_ = ' ' ; if (present(unesc)) unesc_ = unesc
2501 unescaped%raw = ' '
2502 c = 1
2503 do
2504 if (c>len(self%raw)) exit
2505 if (c==len(self%raw)) then

```

```

2506         unescaped%raw = unescaped%raw//self%raw(c:c)
2507         exit
2508     else
2509         if (self%raw(c:c+1)==BACKSLASH//to_unescape) then
2510             unescaped%raw = unescaped%raw//to_unescape
2511             c = c + 2
2512         else
2513             unescaped%raw = unescaped%raw//self%raw(c:c)
2514             c = c + 1
2515         endif
2516     endif
2517 enddo
2518 endif
2519 endfunction unescape
2520
2521 elemental function unique(self, substring) result(uniq)
2522     !< Reduce to one (unique) multiple (sequential) occurrences of a substring into a string.
2523     !<
2524     !< For example the string ' ab-cre-cre-ab ' is reduce to 'ab-cre-ab' if the substring is '-cre '.
2525     !< @note Eventual multiple trailing white space are not reduced to one occurrence.
2526     !<
2527     !<````fortran
2528     !< type(string) :: astring
2529     !< logical      :: test_passed(1)
2530     !< astring = '+++ab-++cre-++cre-ab+++++'
2531     !< test_passed(1) = astring%unique(substring='+')// '+' == '+ab-+cre-+cre-ab+'
2532     !< print '(L1)', all(test_passed)
2533     !<````
2534     !> T <<<
2535     class(string),          intent(in)          :: self          !< The string.
2536     character(kind=CK, len=*), intent(in), optional :: substring    !< Substring which multiple occurrences must be reduced to o
2537     character(kind=CK, len=:), allocatable          :: substring_  !< Substring, default value.
2538     type(string)           :: uniq              !< String parsed.
2539
2540     if (allocated(self%raw)) then
2541         substring_ = SPACE ; if (present(substring)) substring_ = substring
2542
2543         uniq = self
2544         do
2545             if (.not.uniq%index(repeat(substring_, 2))>0) exit
2546             uniq = uniq%replace(old=repeat(substring_, 2), new=substring_)
2547         enddo
2548     endif
2549 endfunction unique

```

```

2550
2551 elemental function upper(self)
2552   !< Return a string with all uppercase characters.
2553   !<
2554   !<````fortran
2555   !< type(string) :: astring
2556   !< logical :: test_passed(1)
2557   !< astring = 'Hello WorLD!'
2558   !< test_passed(1) = astring%upper()/' '=='HELLO WORLD!'
2559   !< print '(L1)', all(test_passed)
2560   !<````
2561   !> T <<<
2562   class(string), intent(in) :: self !< The string.
2563   type(string) :: upper !< Upper case string.
2564   integer :: n1 !< Characters counter.
2565   integer :: n2 !< Characters counter.
2566
2567   if (allocated(self%raw)) then
2568     upper = self
2569     do n1=1, len(self%raw)
2570       n2 = index(LOWER_ALPHABET, self%raw(n1:n1))
2571       if (n2>0) upper%raw(n1:n1) = UPPER_ALPHABET(n2:n2)
2572     enddo
2573   endif
2574 endfunction upper
2575
2576 subroutine write_file(self, file, form, iostat, iomsg)
2577   !< Write a single string stream into file.
2578   !<
2579   !< @note For unformatted read only `access='stream'` is supported with new_line as line terminator.
2580   !<
2581   !<````fortran
2582   !< type(string) :: astring
2583   !< type(string) :: anotherstring
2584   !< type(string), allocatable :: strings(:)
2585   !< type(string) :: line(3)
2586   !< integer :: iostat
2587   !< character(len=99) :: iomsg
2588   !< integer :: scratch
2589   !< integer :: l
2590   !< logical :: test_passed(8)
2591   !< line(1) = ' Hello World! '
2592   !< line(2) = 'How are you? '
2593   !< line(3) = ' All say: "Fine thanks" '

```

```

2594 !< anotherstring = anotherstring%join(array=line, sep=new_line('a'))
2595 !< call anotherstring%write_file(file='write_file_test.tmp', iostat=iostat, iomsg=iomsg)
2596 !< call astring%read_file(file='write_file_test.tmp', iostat=iostat, iomsg=iomsg)
2597 !< call astring%split(tokens=strings, sep=new_line('a'))
2598 !< test_passed(1) = (size(strings, dim=1)==size(line, dim=1))
2599 !< do l=1, size(strings, dim=1)
2600 !<   test_passed(l+1) = (strings(l)==line(l))
2601 !< enddo
2602 !< call anotherstring%write_file(file='write_file_test.tmp', form='unformatted', iostat=iostat, iomsg=iomsg)
2603 !< call astring%read_file(file='write_file_test.tmp', form='unformatted', iostat=iostat, iomsg=iomsg)
2604 !< call astring%split(tokens=strings, sep=new_line('a'))
2605 !< test_passed(5) = (size(strings, dim=1)==size(line, dim=1))
2606 !< do l=1, size(strings, dim=1)
2607 !<   test_passed(l+5) = (strings(l)==line(l))
2608 !< enddo
2609 !< open(newunit=scratch, file='write_file_test.tmp')
2610 !< close(unit=scratch, status='delete')
2611 !< print '(L1)', all(test_passed)
2612 !< ``
2613 !> T <<<
2614 class(string),      intent(in)           :: self      !< The string.
2615 character(len=*),  intent(in)           :: file       !< File name.
2616 character(len=*),  intent(in),          optional :: form       !< Format of unit.
2617 integer,           intent(out),         optional :: iostat      !< IO status code.
2618 character(len=*),  intent(inout),       optional :: iomsg       !< IO status message.
2619 type(string)       :: form_             !< Format of unit, local variable.
2620 integer            :: iostat_           !< IO status code, local variable.
2621 character(len=:),  allocatable          :: iomsg_       !< IO status message, local variable.
2622 integer           :: unit              !< Logical unit.
2623
2624 iomsg_ = repeat(' ', 99) ; if (present(iomsg)) iomsg_ = iomsg
2625 form_ = 'FORMATTED' ; if (present(form)) form_ = form ; form_ = form_%upper()
2626 select case(form_%chars())
2627 case('FORMATTED')
2628   open(newunit=unit, file=file, action='WRITE', iomsg=iomsg_, iostat=iostat_, err=10)
2629 case('UNFORMATTED')
2630   open(newunit=unit, file=file, action='WRITE', form='UNFORMATTED', access='STREAM', iomsg=iomsg_, iostat=iostat_, err=10)
2631 endselect
2632 call self%write_lines(unit=unit, form=form, iomsg=iomsg_, iostat=iostat_)
2633 10 close(unit)
2634 if (present(iostat)) iostat = iostat_
2635 if (present(iomsg)) iomsg = iomsg_
2636 endsubroutine write_file
2637

```

```

2638 subroutine write_line(self, unit, form, iostat, iomsg)
2639 !< Write line (record) to a connected unit.
2640 !<
2641 !< @note If the connected unit is unformatted a `new_line()` character is added at the end (if necessary) to mark the end
2642 !<
2643 !< @note There is no doctests, this being tested by means of [[string:write_file]] doctests.
2644 class(string), intent(in) :: self !< The string.
2645 integer, intent(in) :: unit !< Logical unit.
2646 character(len=*), intent(in), optional :: form !< Format of unit.
2647 integer, intent(out), optional :: iostat !< IO status code.
2648 character(len=*), intent(inout), optional :: iomsg !< IO status message.
2649 type(string) :: form_ !< Format of unit, local variable.
2650 integer :: iostat_ !< IO status code, local variable.
2651 character(len=:), allocatable :: iomsg_ !< IO status message, local variable.
2652
2653 iostat_ = 0
2654 iomsg_ = repeat(' ', 99) ; if (present(iomsg)) iomsg_ = iomsg
2655 if (allocated(self%raw)) then
2656 form_ = 'FORMATTED' ; if (present(form)) form_ = form ; form_ = form_%upper()
2657 select case(form_%chars())
2658 case('FORMATTED')
2659 write(unit, "(A)", iostat=iostat_, iomsg=iomsg_) self%raw
2660 case('UNFORMATTED')
2661 if (self%end_with(new_line('a')) then
2662 write(unit, iostat=iostat_, iomsg=iomsg_) self%raw
2663 else
2664 write(unit, iostat=iostat_, iomsg=iomsg_) self%raw//new_line('a')
2665 endif
2666 endselect
2667 endif
2668 if (present(iostat)) iostat = iostat_
2669 if (present(iomsg)) iomsg = iomsg_
2670 endsubroutine write_line
2671
2672 subroutine write_lines(self, unit, form, iostat, iomsg)
2673 !< Write lines (records) to a connected unit.
2674 !<
2675 !< This method checks if self contains more than one line (records) and writes them as lines (records).
2676 !<
2677 !< @note If the connected unit is unformatted a `new_line()` character is added at the end (if necessary) to mark the end
2678 !<
2679 !< @note There is no doctests, this being tested by means of [[string:write_file]] doctests.
2680 class(string), intent(in) :: self !< The string.
2681 integer, intent(in) :: unit !< Logical unit.

```

```

2682 character(len=*), intent(in), optional :: form !< Format of unit.
2683 integer, intent(out), optional :: iostat !< IO status code.
2684 character(len=*), intent(inout), optional :: iomsg !< IO status message.
2685 type(string), allocatable :: lines(:) !< Lines.
2686 integer :: l !< Counter.
2687
2688 if (allocated(self%raw)) then
2689 call self%split(tokens=lines, sep=new_line('a'))
2690 do l=1, size(lines, dim=1)
2691 call lines(l)%write_line(unit=unit, form=form, iostat=iostat, iomsg=iomsg)
2692 enddo
2693 endif
2694 endsubroutine write_lines
2695
2696 ! inquire
2697 elemental function end_with(self, suffix, start, end, ignore_null_eof)
2698 !< Return true if a string ends with a specified suffix.
2699 !<
2700 !<````fortran
2701 !< type(string) :: astring
2702 !< logical :: test_passed(5)
2703 !< astring = 'Hello WorLD!'
2704 !< test_passed(1) = astring%end_with(suffix='LD!').equiv..true.
2705 !< test_passed(2) = astring%end_with(suffix='ld!').equiv..false.
2706 !< test_passed(3) = astring%end_with(suffix='orLD!', start=5).equiv..true.
2707 !< test_passed(4) = astring%end_with(suffix='orLD!', start=8, end=12).equiv..true.
2708 !< test_passed(5) = astring%end_with(suffix='!').equiv..true.
2709 !< print '(L1)', all(test_passed)
2710 !<````
2711 !=> T <<<
2712 class(string), intent(in) :: self !< The string.
2713 character(kind=CK, len=*), intent(in) :: suffix !< Searched suffix.
2714 integer, intent(in), optional :: start !< Start position into the string.
2715 integer, intent(in), optional :: end !< End position into the string.
2716 logical, intent(in), optional :: ignore_null_eof !< Ignore null character at the end of file.
2717 logical :: end_with !< Result of the test.
2718 integer :: start_ !< Start position into the string, local variable.
2719 integer :: end_ !< End position into the string, local variable.
2720 logical :: ignore_null_eof_ !< Ignore null character at the end of file, local va
2721
2722 end_with = .false.
2723 if (allocated(self%raw)) then
2724 start_ = 1 ; if (present(start)) start_ = start
2725 end_ = len(self%raw) ; if (present(end)) end_ = end

```

```

2726     ignore_null_eof_ = .false. ; if (present(ignore_null_eof)) ignore_null_eof_ = ignore_null_eof
2727     if (ignore_null_eof_ and (self%raw(end_:end_) == char(0))) end_ = end_ - 1
2728     if (len(suffix) <= len(self%raw(start_:end_))) then
2729         end_with = self%raw(end_ - len(suffix) + 1: end_) == suffix
2730     endif
2731 endif
2732 endfunction end_with
2733
2734 elemental function is_allocated(self)
2735     !< Return true if the string is allocated.
2736     !<
2737     !<````fortran
2738     !< type(string) :: astring
2739     !< logical :: test_passed(2)
2740     !< test_passed(1) = astring%is_allocated().equiv..false.
2741     !< astring = 'hello'
2742     !< test_passed(2) = astring%is_allocated().equiv..true.
2743     !< print '(L1)', all(test_passed)
2744     !<````
2745     !> T <<<
2746     class(string), intent(in) :: self !< The string.
2747     logical :: is_allocated !< Result of the test.
2748
2749     is_allocated = allocated(self%raw)
2750 endfunction is_allocated
2751
2752 elemental function is_digit(self)
2753     !< Return true if all characters in the string are digits.
2754     !<
2755     !<````fortran
2756     !< type(string) :: astring
2757     !< logical :: test_passed(2)
2758     !< astring = ' -1212112.3 '
2759     !< test_passed(1) = astring%is_digit().equiv..false.
2760     !< astring = '12121123'
2761     !< test_passed(2) = astring%is_digit().equiv..true.
2762     !< print '(L1)', all(test_passed)
2763     !<````
2764     !> T <<<
2765     class(string), intent(in) :: self !< The string.
2766     logical :: is_digit !< Result of the test.
2767     integer :: c !< Character counter.
2768
2769     is_digit = .false.

```

```

2770 if (allocated(self%raw)) then
2771   do c=1, len(self%raw)
2772     select case (self%raw(c:c))
2773       case ('0':'9')
2774         is_digit = .true.
2775       case default
2776         is_digit = .false.
2777       exit
2778     end select
2779   enddo
2780 endif
2781 endfunction is_digit
2782
2783 elemental function is_integer(self, allow_spaces)
2784   !< Return true if the string contains an integer.
2785   !<
2786   !< The regular expression is `s*[\+|-]?d+([eE]\+?d+)?s*`. The parse algorithm is done in stages:
2787   !<
2788   !< | S0 | S1 | S2 | S3 | S4 | S5 | S6 |
2789   !< |-----|-----|-----|-----|-----|-----|
2790   !< | `s*` | `[+|-]?` | `d+` | `[eE]` | `+?` | `d+` | `s*` |
2791   !<
2792   !< Exit on stages-parsing results in:
2793   !<
2794   !< | S0 | S1 | S2 | S3 | S4 | S5 | S6 |
2795   !< |----|----|----|----|----|----|----|
2796   !< | F | F | T | F | F | T | T |
2797   !<
2798   !< @note This implementation is courtesy of
2799   !< [tomedunn](https://github.com/tomedunn/fortran-string-utility-module/blob/master/src/string_utility_module.f90#L294)
2800   !<
2801   !< ``fortran
2802   !< type(string) :: astring
2803   !< logical :: test_passed(6)
2804   !< astring = ' -1212112 '
2805   !< test_passed(1) = astring%is_integer().equiv..true.
2806   !< astring = ' -1212112 '
2807   !< test_passed(2) = astring%is_integer(allow_spaces=.false.).equiv..false.
2808   !< astring = ' -1212112 '
2809   !< test_passed(3) = astring%is_integer(allow_spaces=.false.).equiv..false.
2810   !< astring = '+2e20 '
2811   !< test_passed(4) = astring%is_integer().equiv..true.
2812   !< astring = ' -2E13 '
2813   !< test_passed(5) = astring%is_integer().equiv..true.

```



```

2814 !< astring = ' -2 E13 '
2815 !< test_passed(6) = astring%is_integer().equiv..false.
2816 !< print '(L1)', all(test_passed)
2817 !<````
2818 !=> T <<<
2819 class(string), intent(in) :: self !< The string.
2820 logical, intent(in), optional :: allow_spaces !< Allow leading-trailing spaces.
2821 logical :: is_integer !< Result of the test.
2822 logical :: allow_spaces_ !< Allow leading-trailing spaces, local variable.
2823 integer :: stage !< Stages counter.
2824 integer :: c !< Character counter.
2825
2826 if (allocated(self%raw)) then
2827   allow_spaces_ = .true. ; if (present(allow_spaces)) allow_spaces_ = allow_spaces
2828   stage = 0
2829   is_integer = .true.
2830   do c=1, len(self%raw)
2831     select case(self%raw(c:c))
2832     case(SPACE, TAB)
2833       select case(stage)
2834       case(0, 6)
2835         is_integer = allow_spaces_
2836       case(2, 5)
2837         is_integer = allow_spaces_
2838         stage = 6
2839       case default
2840         is_integer = .false.
2841       endselect
2842     case(' - ')
2843       select case(stage)
2844       case(0)
2845         stage = 1
2846       case default
2847         is_integer = .false.
2848       end select
2849     case(' + ')
2850       select case(stage)
2851       case(0)
2852         stage = 1
2853       case(3)
2854         stage = 4
2855       case default
2856         is_integer = .false.
2857       endselect

```

```

2858     case('0':'9')
2859         select case(stage)
2860             case(0:1)
2861                 stage = 2
2862             case(3:4)
2863                 stage = 5
2864             case default
2865                 continue
2866             endselect
2867     case ('e','E')
2868         select case(stage)
2869             case(2)
2870                 stage = 3
2871             case default
2872                 is_integer = .false.
2873             endselect
2874     case default
2875         is_integer = .false.
2876     endselect
2877     if (.not.is_integer) exit
2878 enddo
2879 endif
2880 if (is_integer) then
2881     select case(stage)
2882         case(2, 5, 6)
2883             is_integer = .true.
2884         case default
2885             is_integer = .false.
2886         end select
2887 endif
2888 endfunction is_integer
2889
2890 elemental function is_lower(self)
2891     !< Return true if all characters in the string are lowercase.
2892     !<
2893     !<````fortran
2894     !< type(string) :: astring
2895     !< logical      :: test_passed(3)
2896     !< astring = ' Hello World '
2897     !< test_passed(1) = astring%is_lower().equiv..false.
2898     !< astring = ' HELLO WORLD '
2899     !< test_passed(2) = astring%is_lower().equiv..false.
2900     !< astring = ' hello world '
2901     !< test_passed(3) = astring%is_lower().equiv..true.

```

```

2902 !< print '(L1)', all(test_passed)
2903 !<``
2904 !> T <<<
2905 class(string), intent(in) :: self !< The string.
2906 logical :: is_lower !< Result of the test.
2907 integer :: c !< Character counter.
2908
2909 is_lower = .false.
2910 if (allocated(self%raw)) then
2911   is_lower = .true.
2912   do c=1, len(self%raw)
2913     if (index(UPPER_ALPHABET, self%raw(c:c))>0) then
2914       is_lower = .false.
2915       exit
2916     endif
2917   enddo
2918 endif
2919 endfunction is_lower
2920
2921 elemental function is_number(self, allow_spaces)
2922 !< Return true if the string contains a number (real or integer).
2923 !<
2924 !<````fortran
2925 !< type(string) :: astring
2926 !< logical :: test_passed(7)
2927 !< astring = ' -1212112 '
2928 !< test_passed(1) = astring%is_number().equiv..true.
2929 !< astring = ' -121.2112 '
2930 !< test_passed(2) = astring%is_number().equiv..true.
2931 !< astring = ' -1212112 '
2932 !< test_passed(3) = astring%is_number(allow_spaces=.false.).equiv..false.
2933 !< astring = ' -12121.12 '
2934 !< test_passed(4) = astring%is_number(allow_spaces=.false.).equiv..false.
2935 !< astring = '+2e20 '
2936 !< test_passed(5) = astring%is_number().equiv..true.
2937 !< astring = ' -2.4E13 '
2938 !< test_passed(6) = astring%is_number().equiv..true.
2939 !< astring = ' -2 E13 '
2940 !< test_passed(7) = astring%is_number().equiv..false.
2941 !< print '(L1)', all(test_passed)
2942 !<``
2943 !> T <<<
2944 class(string), intent(in) :: self !< The string.
2945 logical, intent(in), optional :: allow_spaces !< Allow leading-trailing spaces.

```

```

2946 logical                                :: is_number      !< Result of the test.
2947
2948 is_number = (self%is_integer(allow_spaces=allow_spaces).or.self%is_real(allow_spaces=allow_spaces))
2949 endfunction is_number
2950
2951 elemental function is_real(self, allow_spaces)
2952 !< Return true if the string contains a real.
2953 !<
2954 !< The regular expression is `s*[\+|-]?d*(/\.?d*([deDE][\+|-]?d+)?)\s*`. The parse algorithm is done in stages:
2955 !<
2956 !< | S0 | S1 | S2 | S3 | S4 | S5 | S6 | S7 | S8 |
2957 !< |-----|-----|-----|-----|-----|-----|-----|-----|
2958 !< | `s*` | `[+|-]?` | `d*` | `\.?` | `d*` | `[deDE]` | `[+|-]?` | `d+` | `s*` |
2959 !<
2960 !< Exit on stages-parsing results in:
2961 !<
2962 !< | S0 | S1 | S2 | S3 | S4 | S5 | S6 | S7 | S8 |
2963 !< |-----|-----|-----|-----|-----|-----|-----|-----|
2964 !< | F | F | T | T | T | F | F | T | T |
2965 !<
2966 !< @note This implementation is courtesy of
2967 !< [tomedunn](https://github.com/tomedunn/fortran-string-utility-module/blob/master/src/string_utility_module.f90#L614)
2968 !<
2969 !<````fortran
2970 !< type(string) :: astring
2971 !< logical :: test_passed(6)
2972 !< astring = ' -1212112.d0 '
2973 !< test_passed(1) = astring%is_real().equiv..true.
2974 !< astring = ' -1212112.d0 '
2975 !< test_passed(2) = astring%is_real(allow_spaces=.false.).equiv..false.
2976 !< astring = ' -1212112.d0 '
2977 !< test_passed(3) = astring%is_real(allow_spaces=.false.).equiv..false.
2978 !< astring = '+2.e20 '
2979 !< test_passed(4) = astring%is_real().equiv..true.
2980 !< astring = ' -2.01E13 '
2981 !< test_passed(5) = astring%is_real().equiv..true.
2982 !< astring = ' -2.01 E13 '
2983 !< test_passed(6) = astring%is_real().equiv..false.
2984 !< print '(L1)', all(test_passed)
2985 !<````
2986 !<=> T <<<
2987 class(string), intent(in) :: self !< The string.
2988 logical, intent(in), optional :: allow_spaces !< Allow leading-trailing spaces.
2989 logical :: is_real !< Result of the test.

```

```

2990 logical :: allow_spaces_ !< Allow leading-trailing spaces, local variable.
2991 logical :: has_leading_digit !< Check the presence of leading digits.
2992 integer :: stage !< Stages counter.
2993 integer :: c !< Character counter.
2994
2995 if (allocated(self%raw)) then
2996   allow_spaces_ = .true. ; if (present(allow_spaces)) allow_spaces_ = allow_spaces
2997   stage = 0
2998   is_real = .true.
2999   has_leading_digit = .false.
3000   do c=1, len(self%raw)
3001     select case(self%raw(c:c))
3002       case(SPACE, TAB)
3003         select case(stage)
3004           case(0, 8)
3005             is_real = allow_spaces_
3006             continue
3007           case(2:4, 7)
3008             is_real = allow_spaces_
3009             stage = 8
3010           case default
3011             is_real = .false.
3012           endselect
3013       case('+', '-')
3014         select case(stage)
3015           case(0)
3016             stage = 1
3017           case(5)
3018             stage = 6
3019           case default
3020             is_real = .false.
3021           endselect
3022       case('0':'9')
3023         select case(stage)
3024           case(0:1)
3025             stage = 2
3026             has_leading_digit = .true.
3027           case(3)
3028             stage = 4
3029           case(5:6)
3030             stage = 7
3031           case default
3032             continue
3033           endselect

```

```

3034     case('.')
3035         select case(stage)
3036             case(0:2)
3037                 stage = 3
3038             case default
3039                 is_real = .false.
3040             endselect
3041     case('e','E','d','D')
3042         select case(stage)
3043             case(2:4)
3044                 stage = 5
3045             case default
3046                 is_real = .false.
3047             endselect
3048         case default
3049             is_real = .false.
3050         endselect
3051         if (.not.is_real) exit
3052     enddo
3053 endif
3054 if (is_real) then
3055     select case(stage)
3056         case(2, 4, 7, 8)
3057             is_real = .true.
3058         case(3)
3059             is_real = has_leading_digit
3060         case default
3061             is_real = .false.
3062         endselect
3063 endif
3064 endfunction is_real
3065
3066 elemental function is_upper(self)
3067     !< Return true if all characters in the string are uppercase.
3068     !<
3069     !<````fortran
3070     !< type(string) :: astring
3071     !< logical      :: test_passed(3)
3072     !< astring = ' Hello World '
3073     !< test_passed(1) = astring%is_upper().equiv..false.
3074     !< astring = ' HELLO WORLD '
3075     !< test_passed(2) = astring%is_upper().equiv..true.
3076     !< astring = ' hello world '
3077     !< test_passed(3) = astring%is_upper().equiv..false.

```

```

3078      !< print '(L1)', all(test_passed)
3079      !<````
3080      !> T <<<
3081      class(string), intent(in) :: self      !< The string.
3082      logical :: is_upper !< Result of the test.
3083      integer :: c      !< Character counter.
3084
3085      is_upper = .false.
3086      if (allocated(self%raw)) then
3087          is_upper = .true.
3088          do c=1, len(self%raw)
3089              if (index(LOWER_ALPHABET, self%raw(c:c))>0) then
3090                  is_upper = .false.
3091                  exit
3092              endif
3093          enddo
3094      endif
3095      endfunction is_upper
3096
3097      elemental function start_with(self, prefix, start, end)
3098      !< Return true if a string starts with a specified prefix.
3099      !<
3100      !<````fortran
3101      !< type(string) :: astring
3102      !< logical :: test_passed(4)
3103      !< astring = 'Hello WorLD!'
3104      !< test_passed(1) = astring%start_with(prefix='Hello ').equiv..true.
3105      !< test_passed(2) = astring%start_with(prefix='hell ').equiv..false.
3106      !< test_passed(3) = astring%start_with(prefix='llo Wor ', start=3).equiv..true.
3107      !< test_passed(4) = astring%start_with(prefix='lo W ', start=4, end=7).equiv..true.
3108      !< print '(L1)', all(test_passed)
3109      !<````
3110      !> T <<<
3111      class(string), intent(in) :: self      !< The string.
3112      character(kind=CK, len=*), intent(in) :: prefix !< Searched prefix.
3113      integer, intent(in), optional :: start !< Start position into the string.
3114      integer, intent(in), optional :: end !< End position into the string.
3115      logical :: start_with !< Result of the test.
3116      integer :: start_ !< Start position into the string, local variable.
3117      integer :: end_ !< End position into the string, local variable.
3118
3119      start_with = .false.
3120      if (allocated(self%raw)) then
3121          start_ = 1 ; if (present(start)) start_ = start

```

```

3122     end_    = len(self%raw) ; if (present(end))    end_    = end
3123     if (len(prefix)<=len(self%raw(start_:end_))) then
3124         start_with = index(self%raw(start_:end_), prefix)==1
3125     endif
3126 endif
3127 endfunction start_with
3128
3129     ! private methods
3130
3131     ! assignments
3132 elemental subroutine string_assign_string(lhs, rhs)
3133     !< Assignment operator from string input.
3134     !<
3135     !<````fortran
3136     !< type(string) :: astring
3137     !< type(string) :: anotherstring
3138     !< logical      :: test_passed(1)
3139     !< astring = 'hello '
3140     !< anotherstring = astring
3141     !< test_passed(1) = astring%chars()==anotherstring%chars()
3142     !< print '(L1)', all(test_passed)
3143     !<````
3144     !> T <<<
3145     class(string), intent(inout) :: lhs !< Left hand side.
3146     type(string), intent(in)    :: rhs !< Right hand side.
3147
3148     if (allocated(rhs%raw)) lhs%raw = rhs%raw
3149 endsubroutine string_assign_string
3150
3151 elemental subroutine string_assign_character(lhs, rhs)
3152     !< Assignment operator from character input.
3153     !<
3154     !<````fortran
3155     !< type(string) :: astring
3156     !< logical      :: test_passed(1)
3157     !< astring = 'hello '
3158     !< test_passed(1) = astring%chars()='hello '
3159     !< print '(L1)', all(test_passed)
3160     !<````
3161     !> T <<<
3162     class(string),           intent(inout) :: lhs !< Left hand side.
3163     character(kind=CK, len=*), intent(in)    :: rhs !< Right hand side.
3164
3165     lhs%raw = rhs

```



```

3166 endsubroutine string_assign_character
3167
3168 elemental subroutine string_assign_integer_I1P(lhs, rhs)
3169   !< Assignment operator from integer input.
3170   !<
3171   !<````fortran
3172   !< use penf
3173   !< type(string) :: astring
3174   !< logical      :: test_passed(1)
3175   !< astring = 127_I1P
3176   !< test_passed(1) = astring%to_number(kind=1_I1P)==127_I1P
3177   !< print '(L1)', all(test_passed)
3178   !<````
3179   !> T <<<
3180   class(string), intent(inout) :: lhs !< Left hand side.
3181   integer(I1P), intent(in)      :: rhs !< Right hand side.
3182
3183   lhs%raw = trim(str(rhs))
3184 endsubroutine string_assign_integer_I1P
3185
3186 elemental subroutine string_assign_integer_I2P(lhs, rhs)
3187   !< Assignment operator from integer input.
3188   !<
3189   !<````fortran
3190   !< use penf
3191   !< type(string) :: astring
3192   !< logical      :: test_passed(1)
3193   !< astring = 127_I2P
3194   !< test_passed(1) = astring%to_number(kind=1_I2P)==127_I2P
3195   !< print '(L1)', all(test_passed)
3196   !<````
3197   !> T <<<
3198   class(string), intent(inout) :: lhs !< Left hand side.
3199   integer(I2P), intent(in)      :: rhs !< Right hand side.
3200
3201   lhs%raw = trim(str(rhs))
3202 endsubroutine string_assign_integer_I2P
3203
3204 elemental subroutine string_assign_integer_I4P(lhs, rhs)
3205   !< Assignment operator from integer input.
3206   !<
3207   !<````fortran
3208   !< use penf
3209   !< type(string) :: astring

```

```

3210 !< logical      :: test_passed(1)
3211 !< astring = 127_I4P
3212 !< test_passed(1) = astring%to_number(kind=1_I4P)==127_I4P
3213 !< print '(L1)', all(test_passed)
3214 !<````
3215 !> T <<<
3216 class(string), intent(inout) :: lhs !< Left hand side.
3217 integer(I4P), intent(in)      :: rhs !< Right hand side.
3218
3219 lhs%raw = trim(str(rhs))
3220 endsubroutine string_assign_integer_I4P
3221
3222 elemental subroutine string_assign_integer_I8P(lhs, rhs)
3223 !< Assignment operator from integer input.
3224 !<
3225 !<````fortran
3226 !< use penf
3227 !< type(string) :: astring
3228 !< logical      :: test_passed(1)
3229 !< astring = 127_I8P
3230 !< test_passed(1) = astring%to_number(kind=1_I8P)==127_I8P
3231 !< print '(L1)', all(test_passed)
3232 !<````
3233 !> T <<<
3234 class(string), intent(inout) :: lhs !< Left hand side.
3235 integer(I8P), intent(in)      :: rhs !< Right hand side.
3236
3237 lhs%raw = trim(str(rhs))
3238 endsubroutine string_assign_integer_I8P
3239
3240 elemental subroutine string_assign_real_R4P(lhs, rhs)
3241 !< Assignment operator from real input.
3242 !<
3243 !<````fortran
3244 !< use penf
3245 !< type(string) :: astring
3246 !< logical      :: test_passed(1)
3247 !< astring = 3.021e6_R4P
3248 !< test_passed(1) = astring%to_number(kind=1._R4P)==3.021e6_R4P
3249 !< print '(L1)', all(test_passed)
3250 !<````
3251 !> T <<<
3252 class(string), intent(inout) :: lhs !< Left hand side.
3253 real(R4P), intent(in)      :: rhs !< Right hand side.

```

```

3254
3255 lhs%raw = trim(str(rhs))
3256 endsubroutine string_assign_real_R4P
3257
3258 elemental subroutine string_assign_real_R8P(lhs, rhs)
3259 !< Assignment operator from real input.
3260 !<
3261 !<````fortran
3262 !< use penf
3263 !< type(string) :: astring
3264 !< logical :: test_passed(1)
3265 !< astring = 3.021e6_R8P
3266 !< test_passed(1) = astring%to_number(kind=1._R8P)==3.021e6_R8P
3267 !< print '(L1)', all(test_passed)
3268 !<````
3269 !> T <<<
3270 class(string), intent(inout) :: lhs !< Left hand side.
3271 real(R8P), intent(in) :: rhs !< Right hand side.
3272
3273 lhs%raw = trim(str(rhs))
3274 endsubroutine string_assign_real_R8P
3275
3276 elemental subroutine string_assign_real_R16P(lhs, rhs)
3277 !< Assignment operator from real input.
3278 !<
3279 !<````fortran
3280 !< use penf
3281 !< type(string) :: astring
3282 !< logical :: test_passed(1)
3283 !< astring = 3.021e6_R8P
3284 !< test_passed(1) = astring%to_number(kind=1._R8P)==3.021e6_R8P
3285 !< print '(L1)', all(test_passed)
3286 !<````
3287 !> T <<<
3288 class(string), intent(inout) :: lhs !< Left hand side.
3289 real(R16P), intent(in) :: rhs !< Right hand side.
3290
3291 lhs%raw = trim(str(rhs))
3292 endsubroutine string_assign_real_R16P
3293
3294 ! contatenation operators
3295 pure function string_concat_string(lhs, rhs) result(concat)
3296 !< Concatenation with string.
3297 !<

```

```

3298 !<```fortran
3299 !< type(string) :: astring
3300 !< type(string) :: anotherstring
3301 !< logical      :: test_passed(1)
3302 !< astring = 'Hello '
3303 !< anotherstring = 'Bye bye '
3304 !< test_passed(1) = astring//anotherstring=='Hello Bye bye '
3305 !< print '(L1)', all(test_passed)
3306 !<```
3307 !=> T <<<
3308 class(string), intent(in)          :: lhs      !< Left hand side.
3309 type(string), intent(in)          :: rhs      !< Right hand side.
3310 character(kind=CK, len=:), allocatable :: concat !< Concatenated string.
3311
3312 concat = ''
3313 if (allocated(lhs%raw)) concat = lhs%raw
3314 if (allocated(rhs%raw)) concat = concat//rhs%raw
3315 endfunction string_concat_string
3316
3317 pure function string_concat_character(lhs, rhs) result(concat)
3318 !< Concatenation with character.
3319 !<
3320 !<```fortran
3321 !< type(string)          :: astring
3322 !< character(len=:), allocatable :: acharacter
3323 !< logical              :: test_passed(1)
3324 !< astring = 'Hello '
3325 !< acharacter = 'World!'
3326 !< test_passed(1) = astring//acharacter=='Hello World!'
3327 !< print '(L1)', all(test_passed)
3328 !<```
3329 !=> T <<<
3330 class(string),          intent(in)  :: lhs      !< Left hand side.
3331 character(kind=CK, len=*), intent(in) :: rhs      !< Right hand side.
3332 character(kind=CK, len=:), allocatable :: concat !< Concatenated string.
3333
3334 if (allocated(lhs%raw)) then
3335     concat = lhs%raw//rhs
3336 else
3337     concat = rhs
3338 endif
3339 endfunction string_concat_character
3340
3341 pure function character_concat_string(lhs, rhs) result(concat)

```

```

3342 !< Concatenation with character (inverted).
3343 !<
3344 !<````fortran
3345 !< type(string) :: astring
3346 !< character(len=:), allocatable :: acharacter
3347 !< logical :: test_passed(1)
3348 !< astring = 'Hello '
3349 !< acharacter = 'World!'
3350 !< test_passed(1) = acharacter//astring=='World!Hello '
3351 !< print '(L1)', all(test_passed)
3352 !<````
3353 !=> T <<<
3354 character(kind=CK, len=*), intent(in) :: lhs !< Left hand side.
3355 class(string), intent(in) :: rhs !< Right hand side.
3356 character(kind=CK, len=:), allocatable :: concat !< Concatenated string.
3357
3358 if (allocated(rhs%raw)) then
3359     concat = lhs//rhs%raw
3360 else
3361     concat = lhs
3362 endif
3363 endfunction character_concat_string
3364
3365 elemental function string_concat_string_string(lhs, rhs) result(concat)
3366 !< Concatenation with string.
3367 !<
3368 !<````fortran
3369 !< type(string) :: astring
3370 !< type(string) :: anotherstring
3371 !< type(string) :: yetanotherstring
3372 !< logical :: test_passed(1)
3373 !< astring = 'Hello '
3374 !< anotherstring = 'Bye bye '
3375 !< yetanotherstring = astring.cat.anotherstring
3376 !< test_passed(1) = yetanotherstring%chars()=='Hello Bye bye '
3377 !< print '(L1)', all(test_passed)
3378 !<````
3379 !=> T <<<
3380 class(string), intent(in) :: lhs !< Left hand side.
3381 type(string), intent(in) :: rhs !< Right hand side.
3382 type(string) :: concat !< Concatenated string.
3383 character(kind=CK, len=:), allocatable :: temporary !< Temporary concatenated string.
3384
3385 temporary = ''

```

```

3386 if (allocated(lhs%raw)) temporary = lhs%raw
3387 if (allocated(rhs%raw)) temporary = temporary//rhs%raw
3388 if (temporary/='') concat%raw = temporary
3389 endfunction string_concat_string_string
3390
3391 elemental function string_concat_character_string(lhs, rhs) result(concat)
3392   !< Concatenation with character.
3393   !<
3394   !<````fortran
3395   !< type(string) :: astring
3396   !< type(string) :: yetanotherstring
3397   !< character(len=:), allocatable :: acharacter
3398   !< logical :: test_passed(1)
3399   !< astring = 'Hello '
3400   !< acharacter = 'World!'
3401   !< yetanotherstring = astring.cat.acharacter
3402   !< test_passed(1) = yetanotherstring%chars()== 'Hello World!'
3403   !< print '(L1)', all(test_passed)
3404   !<````
3405   !> T <<<
3406   class(string), intent(in) :: lhs !< Left hand side.
3407   character(kind=CK, len=*), intent(in) :: rhs !< Right hand side.
3408   type(string) :: concat !< Concatenated string.
3409
3410 if (allocated(lhs%raw)) then
3411   concat%raw = lhs%raw//rhs
3412 else
3413   concat%raw = rhs
3414 endif
3415 endfunction string_concat_character_string
3416
3417 elemental function character_concat_string_string(lhs, rhs) result(concat)
3418   !< Concatenation with character (inverted).
3419   !<
3420   !<````fortran
3421   !< type(string) :: astring
3422   !< type(string) :: yetanotherstring
3423   !< character(len=:), allocatable :: acharacter
3424   !< logical :: test_passed(1)
3425   !< astring = 'Hello '
3426   !< acharacter = 'World!'
3427   !< yetanotherstring = acharacter.cat.astring
3428   !< test_passed(1) = yetanotherstring%chars()== 'World!Hello '
3429   !< print '(L1)', all(test_passed)

```

```

3430 !<``
3431 !=> T <<<
3432 character(kind=CK, len=*), intent(in) :: lhs !< Left hand side.
3433 class(string), intent(in) :: rhs !< Right hand side.
3434 type(string) :: concat !< Concatenated string.
3435
3436 if (allocated(rhs%raw)) then
3437   concat%raw = lhs//rhs%raw
3438 else
3439   concat%raw = lhs
3440 endif
3441 endfunction character_concat_string_string
3442
3443 ! logical operators
3444 elemental function string_eq_string(lhs, rhs) result(is_it)
3445 !< Equal to string logical operator.
3446 !<
3447 !<``fortran
3448 !< type(string) :: astring
3449 !< type(string) :: anotherstring
3450 !< logical :: test_passed(2)
3451 !< astring = ' one '
3452 !< anotherstring = 'two '
3453 !< test_passed(1) = ((astring==anotherstring).equiv..false.)
3454 !< astring = 'the same '
3455 !< anotherstring = 'the same '
3456 !< test_passed(2) = ((astring==anotherstring).equiv..true.)
3457 !< print '(L1)', all(test_passed)
3458 !<``
3459 !=> T <<<
3460 class(string), intent(in) :: lhs !< Left hand side.
3461 type(string), intent(in) :: rhs !< Right hand side.
3462 logical :: is_it !< Operator test result.
3463
3464 is_it = lhs%raw == rhs%raw
3465 endfunction string_eq_string
3466
3467 elemental function string_eq_character(lhs, rhs) result(is_it)
3468 !< Equal to character logical operator.
3469 !<
3470 !<``fortran
3471 !< type(string) :: astring
3472 !< character(len=:), allocatable :: acharacter
3473 !< logical :: test_passed(2)

```

```

3474 !< astring = ' one '
3475 !< acharacter = 'three '
3476 !< test_passed(1) = ((astring==acharacter).equiv..false.)
3477 !< astring = 'the same '
3478 !< acharacter = 'the same '
3479 !< test_passed(2) = ((astring==acharacter).equiv..true.)
3480 !< print '(L1)', all(test_passed)
3481 !<```
3482 !> T <<<
3483 class(string), intent(in) :: lhs !< Left hand side.
3484 character(kind=CK, len=*), intent(in) :: rhs !< Right hand side.
3485 logical :: is_it !< Operator test result.
3486
3487 is_it = lhs%raw == rhs
3488 endfunction string_eq_character
3489
3490 elemental function character_eq_string(lhs, rhs) result(is_it)
3491 !< Equal to character (inverted) logical operator.
3492 !<
3493 !<```fortran
3494 !< type(string) :: astring
3495 !< character(len=:), allocatable :: acharacter
3496 !< logical :: test_passed(2)
3497 !< astring = ' one '
3498 !< acharacter = 'three '
3499 !< test_passed(1) = ((acharacter==astring).equiv..false.)
3500 !< astring = 'the same '
3501 !< acharacter = 'the same '
3502 !< test_passed(2) = ((acharacter==astring).equiv..true.)
3503 !< print '(L1)', all(test_passed)
3504 !<```
3505 !> T <<<
3506 character(kind=CK, len=*), intent(in) :: lhs !< Left hand side.
3507 class(string), intent(in) :: rhs !< Right hand side.
3508 logical :: is_it !< Operator test result.
3509
3510 is_it = rhs%raw == lhs
3511 endfunction character_eq_string
3512
3513 elemental function string_ne_string(lhs, rhs) result(is_it)
3514 !< Not equal to string logical operator.
3515 !<
3516 !<```fortran
3517 !< type(string) :: astring

```



```

3518 !< type(string) :: anotherstring
3519 !< logical :: test_passed(2)
3520 !< astring = ' one '
3521 !< anotherstring = 'two '
3522 !< test_passed(1) = ((astring/=anotherstring).equiv..true.)
3523 !< astring = 'the same '
3524 !< anotherstring = 'the same '
3525 !< test_passed(2) = ((astring/=anotherstring).equiv..false.)
3526 !< print '(L1)', all(test_passed)
3527 !<```
3528 !> T <<<
3529 class(string), intent(in) :: lhs !< Left hand side.
3530 type(string), intent(in) :: rhs !< Right hand side.
3531 logical :: is_it !< Operator test result.
3532
3533 is_it = lhs%raw /= rhs%raw
3534 endfunction string_ne_string
3535
3536 elemental function string_ne_character(lhs, rhs) result(is_it)
3537 !< Not equal to character logical operator.
3538 !<
3539 !<```fortran
3540 !< type(string) :: astring
3541 !< character(len=:), allocatable :: acharacter
3542 !< logical :: test_passed(2)
3543 !< astring = ' one '
3544 !< acharacter = 'three '
3545 !< test_passed(1) = ((astring/=acharacter).equiv..true.)
3546 !< astring = 'the same '
3547 !< acharacter = 'the same '
3548 !< test_passed(2) = ((astring/=acharacter).equiv..false.)
3549 !< print '(L1)', all(test_passed)
3550 !<```
3551 !> T <<<
3552 class(string), intent(in) :: lhs !< Left hand side.
3553 character(kind=CK, len=*), intent(in) :: rhs !< Right hand side.
3554 logical :: is_it !< Operator test result.
3555
3556 is_it = lhs%raw /= rhs
3557 endfunction string_ne_character
3558
3559 elemental function character_ne_string(lhs, rhs) result(is_it)
3560 !< Not equal to character (inverted) logical operator.
3561 !<

```

```

3562 !<```fortran
3563 !< type(string) :: astring
3564 !< character(len=:), allocatable :: acharacter
3565 !< logical :: test_passed(2)
3566 !< astring = ' one '
3567 !< acharacter = 'three '
3568 !< test_passed(1) = ((acharacter/=astring).equiv..true.)
3569 !< astring = 'the same '
3570 !< acharacter = 'the same '
3571 !< test_passed(2) = ((acharacter/=astring).equiv..false.)
3572 !< print '(L1)', all(test_passed)
3573 !<```
3574 !=> T <<<
3575 character(kind=CK, len=*), intent(in) :: lhs !< Left hand side.
3576 class(string), intent(in) :: rhs !< Right hand side.
3577 logical :: is_it !< Opreator test result.
3578
3579 is_it = rhs%raw /= lhs
3580 endfunction character_ne_string
3581
3582 elemental function string_lt_string(lhs, rhs) result(is_it)
3583 !< Lower than to string logical operator.
3584 !<
3585 !<```fortran
3586 !< type(string) :: astring
3587 !< type(string) :: anotherstring
3588 !< logical :: test_passed(2)
3589 !< astring = 'one '
3590 !< anotherstring = 'ONE '
3591 !< test_passed(1) = ((astring<anotherstring).equiv..false.)
3592 !< astring = 'ONE '
3593 !< anotherstring = 'one '
3594 !< test_passed(2) = ((astring<anotherstring).equiv..true.)
3595 !< print '(L1)', all(test_passed)
3596 !<```
3597 !=> T <<<
3598 class(string), intent(in) :: lhs !< Left hand side.
3599 type(string), intent(in) :: rhs !< Right hand side.
3600 logical :: is_it !< Opreator test result.
3601
3602 is_it = lhs%raw < rhs%raw
3603 endfunction string_lt_string
3604
3605 elemental function string_lt_character(lhs, rhs) result(is_it)

```

```

3606      !< Lower than to character logical operator.
3607      !<
3608      !<````fortran
3609      !< type(string)                :: astring
3610      !< character(len=:), allocatable :: acharacter
3611      !< logical                      :: test_passed(2)
3612      !< astring = 'one '
3613      !< acharacter = 'ONE '
3614      !< test_passed(1) = ((astring<acharacter).equiv..false.)
3615      !< astring = 'ONE '
3616      !< acharacter = 'one '
3617      !< test_passed(2) = ((astring<acharacter).equiv..true.)
3618      !< print '(L1)', all(test_passed)
3619      !<````
3620      !> T <<<
3621      class(string),          intent(in) :: lhs    !< Left hand side.
3622      character(kind=CK, len=*), intent(in) :: rhs  !< Right hand side.
3623      logical                :: is_it !< Operator test result.
3624
3625      is_it = lhs%raw < rhs
3626      endfunction string_lt_character
3627
3628      elemental function character_lt_string(lhs, rhs) result(is_it)
3629      !< Lower than to character (inverted) logical operator.
3630      !<
3631      !<````fortran
3632      !< type(string)                :: astring
3633      !< character(len=:), allocatable :: acharacter
3634      !< logical                      :: test_passed(2)
3635      !< astring = 'one '
3636      !< acharacter = 'ONE '
3637      !< test_passed(1) = ((acharacter<astring).equiv..true.)
3638      !< astring = 'ONE '
3639      !< acharacter = 'one '
3640      !< test_passed(2) = ((acharacter<astring).equiv..false.)
3641      !< print '(L1)', all(test_passed)
3642      !<````
3643      !> T <<<
3644      character(kind=CK, len=*), intent(in) :: lhs    !< Left hand side.
3645      class(string),          intent(in) :: rhs  !< Right hand side.
3646      logical                :: is_it !< Operator test result.
3647
3648      is_it = lhs < rhs%raw
3649      endfunction character_lt_string

```

```

3650
3651 elemental function string_le_string(lhs, rhs) result(is_it)
3652   !< Lower equal than to string logical operator.
3653   !<
3654   !<````fortran
3655   !< type(string) :: astring
3656   !< type(string) :: anotherstring
3657   !< logical      :: test_passed(3)
3658   !< astring = 'one'
3659   !< anotherstring = 'ONE'
3660   !< test_passed(1) = ((astring <= anotherstring).equiv..false.)
3661   !< astring = 'ONE'
3662   !< anotherstring = 'one'
3663   !< test_passed(2) = ((astring <= anotherstring).equiv..true.)
3664   !< astring = 'ONE'
3665   !< anotherstring = 'ONE'
3666   !< test_passed(3) = ((astring <= anotherstring).equiv..true.)
3667   !< print '(L1)', all(test_passed)
3668   !<````
3669   !> T <<<
3670   class(string), intent(in) :: lhs    !< Left hand side.
3671   type(string), intent(in) :: rhs    !< Right hand side.
3672   logical                :: is_it !< Operator test result.
3673
3674   is_it = lhs%raw <= rhs%raw
3675 endfunction string_le_string
3676
3677 elemental function string_le_character(lhs, rhs) result(is_it)
3678   !< Lower equal than to character logical operator.
3679   !<
3680   !<````fortran
3681   !< type(string)                :: astring
3682   !< character(len=:), allocatable :: acharacter
3683   !< logical                    :: test_passed(3)
3684   !< astring = 'one'
3685   !< acharacter = 'ONE'
3686   !< test_passed(1) = ((astring <= acharacter).equiv..false.)
3687   !< astring = 'ONE'
3688   !< acharacter = 'one'
3689   !< test_passed(2) = ((astring <= acharacter).equiv..true.)
3690   !< astring = 'ONE'
3691   !< acharacter = 'ONE'
3692   !< test_passed(3) = ((astring <= acharacter).equiv..true.)
3693   !< print '(L1)', all(test_passed)

```

```

3694 !<```
3695 !=> T <<<
3696 class(string), intent(in) :: lhs !< Left hand side.
3697 character(kind=CK, len=*), intent(in) :: rhs !< Right hand side.
3698 logical :: is_it !< Operator test result.
3699
3700 is_it = lhs%raw <= rhs
3701 endfunction string_le_character
3702
3703 elemental function character_le_string(lhs, rhs) result(is_it)
3704 !< Lower equal than to character (inverted) logical operator.
3705 !<
3706 !<```fortran
3707 !< type(string) :: astring
3708 !< character(len=:), allocatable :: acharacter
3709 !< logical :: test_passed(3)
3710 !< astring = 'one'
3711 !< acharacter = 'ONE'
3712 !< test_passed(1) = ((acharacter <= astring).equiv..true.)
3713 !< astring = 'ONE'
3714 !< acharacter = 'one'
3715 !< test_passed(2) = ((acharacter <= astring).equiv..false.)
3716 !< astring = 'ONE'
3717 !< acharacter = 'ONE'
3718 !< test_passed(3) = ((acharacter <= astring).equiv..true.)
3719 !< print '(L1)', all(test_passed)
3720 !<```
3721 !=> T <<<
3722 character(kind=CK, len=*), intent(in) :: lhs !< Left hand side.
3723 class(string), intent(in) :: rhs !< Right hand side.
3724 logical :: is_it !< Operator test result.
3725
3726 is_it = lhs <= rhs%raw
3727 endfunction character_le_string
3728
3729 elemental function string_ge_string(lhs, rhs) result(is_it)
3730 !< Greater equal than to string logical operator.
3731 !<
3732 !<```fortran
3733 !< type(string) :: astring
3734 !< type(string) :: anotherstring
3735 !< logical :: test_passed(3)
3736 !< astring = 'one'
3737 !< anotherstring = 'ONE'

```

```

3738 !< test_passed(1) = ((astring >= anotherstring).equiv..true.)
3739 !< astring = 'ONE '
3740 !< anotherstring = 'one '
3741 !< test_passed(2) = ((astring >= anotherstring).equiv..false.)
3742 !< astring = 'ONE '
3743 !< anotherstring = 'ONE '
3744 !< test_passed(3) = ((astring >= anotherstring).equiv..true.)
3745 !< print '(L1)', all(test_passed)
3746 !<```
3747 !> T <<<
3748 class(string), intent(in) :: lhs !< Left hand side.
3749 type(string), intent(in) :: rhs !< Right hand side.
3750 logical :: is_it !< Operator test result.
3751
3752 is_it = lhs%raw >= rhs%raw
3753 endfunction string_ge_string
3754
3755 elemental function string_ge_character(lhs, rhs) result(is_it)
3756 !< Greater equal than to character logical operator.
3757 !<
3758 !<```fortran
3759 !< type(string) :: astring
3760 !< character(len=:), allocatable :: acharacter
3761 !< logical :: test_passed(3)
3762 !< astring = 'one '
3763 !< acharacter = 'ONE '
3764 !< test_passed(1) = ((astring >= acharacter).equiv..true.)
3765 !< astring = 'ONE '
3766 !< acharacter = 'one '
3767 !< test_passed(2) = ((astring >= acharacter).equiv..false.)
3768 !< astring = 'ONE '
3769 !< acharacter = 'ONE '
3770 !< test_passed(3) = ((astring >= acharacter).equiv..true.)
3771 !< print '(L1)', all(test_passed)
3772 !<```
3773 !> T <<<
3774 class(string), intent(in) :: lhs !< Left hand side.
3775 character(kind=CK, len=*), intent(in) :: rhs !< Right hand side.
3776 logical :: is_it !< Operator test result.
3777
3778 is_it = lhs%raw >= rhs
3779 endfunction string_ge_character
3780
3781 elemental function character_ge_string(lhs, rhs) result(is_it)

```

```

3782 !< Greater equal than to character (inverted) logical operator.
3783 !<
3784 !<````fortran
3785 !< type(string) :: astring
3786 !< character(len=:), allocatable :: acharacter
3787 !< logical :: test_passed(3)
3788 !< astring = 'one'
3789 !< acharacter = 'ONE'
3790 !< test_passed(1) = ((acharacter >= astring).equiv..false.)
3791 !< astring = 'ONE'
3792 !< acharacter = 'one'
3793 !< test_passed(2) = ((acharacter >= astring).equiv..true.)
3794 !< astring = 'ONE'
3795 !< acharacter = 'ONE'
3796 !< test_passed(3) = ((acharacter >= astring).equiv..true.)
3797 !< print '(L1)', all(test_passed)
3798 !<````
3799 !> T <<<
3800 character(kind=CK, len=*), intent(in) :: lhs !< Left hand side.
3801 class(string), intent(in) :: rhs !< Right hand side.
3802 logical :: is_it !< Operator test result.
3803
3804 is_it = lhs >= rhs%raw
3805 endfunction character_ge_string
3806
3807 elemental function string_gt_string(lhs, rhs) result(is_it)
3808 !< Greater than to string logical operator.
3809 !<
3810 !<````fortran
3811 !< type(string) :: astring
3812 !< type(string) :: anotherstring
3813 !< logical :: test_passed(2)
3814 !< astring = 'one'
3815 !< anotherstring = 'ONE'
3816 !< test_passed(1) = ((astring > anotherstring).equiv..true.)
3817 !< astring = 'ONE'
3818 !< anotherstring = 'one'
3819 !< test_passed(2) = ((astring > anotherstring).equiv..false.)
3820 !< print '(L1)', all(test_passed)
3821 !<````
3822 !> T <<<
3823 class(string), intent(in) :: lhs !< Left hand side.
3824 type(string), intent(in) :: rhs !< Right hand side.
3825 logical :: is_it !< Operator test result.

```

```

3826
3827 is_it = lhs%raw > rhs%raw
3828 endfunction string_gt_string
3829
3830 elemental function string_gt_character(lhs, rhs) result(is_it)
3831 !< Greater than to character logical operator.
3832 !<
3833 !<````fortran
3834 !< type(string) :: astring
3835 !< character(len=:), allocatable :: acharacter
3836 !< logical :: test_passed(2)
3837 !< astring = 'one'
3838 !< acharacter = 'ONE'
3839 !< test_passed(1) = ((astring>acharacter).equiv..true.)
3840 !< astring = 'ONE'
3841 !< acharacter = 'one'
3842 !< test_passed(2) = ((astring>acharacter).equiv..false.)
3843 !< print '(L1)', all(test_passed)
3844 !<````
3845 !=> T <<<
3846 class(string), intent(in) :: lhs !< Left hand side.
3847 character(kind=CK, len=*), intent(in) :: rhs !< Right hand side.
3848 logical :: is_it !< Operator test result.
3849
3850 is_it = lhs%raw > rhs
3851 endfunction string_gt_character
3852
3853 elemental function character_gt_string(lhs, rhs) result(is_it)
3854 !< Greater than to character (inverted) logical operator.
3855 !<
3856 !<````fortran
3857 !< type(string) :: astring
3858 !< character(len=:), allocatable :: acharacter
3859 !< logical :: test_passed(2)
3860 !< astring = 'one'
3861 !< acharacter = 'ONE'
3862 !< test_passed(1) = ((acharacter>astring).equiv..false.)
3863 !< astring = 'ONE'
3864 !< acharacter = 'one'
3865 !< test_passed(2) = ((acharacter>astring).equiv..true.)
3866 !< print '(L1)', all(test_passed)
3867 !<````
3868 !=> T <<<
3869 character(kind=CK, len=*), intent(in) :: lhs !< Left hand side.

```



```

3870 class(string),           intent(in) :: rhs      !< Right hand side.
3871 logical                  :: is_it !< Operator test result.
3872
3873 is_it = lhs > rhs%raw
3874 endfunction character_gt_string
3875
3876 ! IO
3877 #ifndef __GFORTRAN__
3878 subroutine read_formatted(dtv, unit, iotype, v_list, iostat, iomsg)
3879 !< Formatted input.
3880 !<
3881 !< @bug Change temporary acks: find a more precise length of the input string and avoid the trimming!
3882 !<
3883 !< @bug Read listdirected with and without delimiters does not work.
3884 class(string),           intent(inout) :: dtv          !< The string.
3885 integer,                intent(in)    :: unit          !< Logical unit.
3886 character(len=*),       intent(in)    :: iotype        !< Edit descriptor.
3887 integer,                intent(in)    :: v_list(:)     !< Edit descriptor list.
3888 integer,                intent(out)   :: iostat        !< IO status code.
3889 character(len=*),       intent(inout) :: iomsg         !< IO status message.
3890 character(len=len(iomsg)) :: local_iomsg !< Local variant of iomsg, so it doesn't get inappropriately rede
3891 character(kind=CK, len=1) :: delim      !< String delimiter, if any.
3892 character(kind=CK, len=100) :: temporary !< Temporary storage string.
3893
3894 if (iotype == 'LISTDIRECTED') then
3895   call get_next_non_blank_character_any_record(unit=unit, ch=delim, iostat=iostat, iomsg=iomsg)
3896   if (iostat /= 0) return
3897   if (delim == '' .OR. delim == '"') then
3898     call dtv%read_delimited(unit=unit, delim=delim, iostat=iostat, iomsg=local_iomsg)
3899   else
3900     ! step back before the non-blank
3901     read(unit, "(TL1)", iostat=iostat, iomsg=iomsg)
3902     if (iostat /= 0) return
3903     call dtv%read_undelimited_listdirected(unit=unit, iostat=iostat, iomsg=local_iomsg)
3904   endif
3905   if (is_iostat_eor(iostat)) then
3906     ! suppress IOSTAT_EOR
3907     iostat = 0
3908   elseif (iostat /= 0) then
3909     iomsg = local_iomsg
3910   endif
3911   return
3912 else
3913   read(unit, "(A)", iostat=iostat, iomsg=iomsg) temporary

```

```

3914     dtv%raw = trim(temporary)
3915 endif
3916 endsubroutine read_formatted
3917
3918 subroutine read_delimited(dtv, unit, delim, iostat, iomsg)
3919     !< Read a delimited string from a unit connected for formatted input.
3920     !<
3921     !< If the closing delimiter is followed by end of record, then we return end of record.
3922     !<
3923     !< @note This does not need a doctest, it being tested by [[string::read_formatted]].
3924     class(string),           intent(out)    :: dtv           !< The string.
3925     integer,                intent(in)     :: unit          !< Logical unit.
3926     character(kind=CK, len=1), intent(in)  :: delim         !< String delimiter.
3927     integer,                intent(out)    :: iostat        !< IO status code.
3928     character(kind=CK, len=*), intent(inout) :: iomsg        !< IO status message.
3929     character(kind=CK, len=1)  :: ch         !< A character read.
3930     logical                  :: was_delim  !< Indicates that the last character read was a delimiter.
3931
3932     was_delim = .false.
3933     dtv%raw = ''
3934     do
3935         read(unit, "(A)", iostat=iostat, iomsg=iomsg) ch
3936         if (is_iostat_eor(iostat)) then
3937             if (was_delim) then
3938                 ! end of delimited string followed by end of record is end of the string. Pass back the
3939                 ! end of record condition to the caller
3940                 return
3941             else
3942                 ! end of record without terminating delimiter - move along
3943                 cycle
3944             endif
3945         elseif (iostat /= 0) THEN
3946             return
3947         endif
3948         if (ch == delim) then
3949             if (was_delim) then
3950                 ! doubled delimiter is one delimiter in the value
3951                 dtv%raw = dtv%raw // ch
3952                 was_delim = .false.
3953             else
3954                 ! need to test next character to see what is happening
3955                 was_delim = .true.
3956             endif
3957         elseif (was_delim) then

```

```

3958      ! the previous character was actually the delimiter for the end of the string. Put back this character
3959      read(unit, "(TL1)", iostat=iostat, iomsg=iomsg)
3960      return
3961      else
3962          dtv%raw = dtv%raw // ch
3963      endif
3964      enddo
3965      endsubroutine read_delimited
3966
3967      subroutine read_undelimited_listdirected(dtv, unit, iostat, iomsg)
3968      !< Read an undelimited (no leading apostrophe or double quote) character value according to the rules for list directed in
3969      !<
3970      !< A blank, comma/semicolon (depending on the decimal mode), slash or end of record terminates the string.
3971      !<
3972      !< If input is terminated by end of record, then this procedure returns an end-of-record condition.
3973      class(string),      intent(inout) :: dtv                !< The string.
3974      integer,           intent(in)   :: unit                !< Logical unit.
3975      integer,           intent(out)  :: iostat              !< IO status code.
3976      character(len=*), intent(inout) :: iomsg               !< IO status message.
3977      logical           :: decimal_point !< True if DECIMAL=POINT in effect.
3978
3979      call get_decimal_mode(unit=unit, decimal_point=decimal_point, iostat=iostat, iomsg=iomsg)
3980      if (iostat /= 0) return
3981      call dtv%read_undelimited(unit=unit, terminators='□'// '/'// '/'//merge(CK_',', CK_';', decimal_point), iostat=iostat, iomsg=iomsg)
3982      endsubroutine read_undelimited_listdirected
3983
3984      subroutine read_undelimited(dtv, unit, terminators, iostat, iomsg)
3985      !< Read an undelimited string up until end of record or a character from a set of terminators is encountered.
3986      !<
3987      !< If a terminator is encountered, the file position will be at that terminating character. If end of record is encountered
3988      !< file remains at end of record.
3989      class(string),      intent(inout) :: dtv                !< The string.
3990      integer,           intent(in)   :: unit                !< Logical unit.
3991      character(kind=CK, len=*), intent(in) :: terminators !< Characters that are considered to terminate the string.
3992      !< Blanks in this string are meaningful.
3993      integer,           intent(out)  :: iostat              !< IO status code.
3994      character(len=*), intent(inout) :: iomsg               !< IO status message.
3995      character(kind=CK, len=1)       :: ch                 !< A character read.
3996
3997      dtv%raw = ''
3998      do
3999          read(unit, "(A)", iostat=iostat, iomsg=iomsg) ch
4000          if (is_iostat_eor(iostat)) then
4001              ! end of record just means end of string. We pass on the condition

```

```

4002     return
4003 elseif (iostat /= 0) then
4004     ! something odd happened
4005     return
4006 endif
4007 if (scan(ch, terminators) /= 0) then
4008     ! change the file position so that the next read sees the terminator
4009     read(unit, "(TL1)", iostat=iostat, iomsg=iomsg)
4010     if (iostat /= 0) return
4011     iostat = 0
4012     return
4013 endif
4014     ! we got a character - append it
4015     dtv%raw = dtv%raw // ch
4016 enddo
4017 endsubroutine read_undelimited
4018
4019 subroutine write_formatted(dtv, unit, iotype, v_list, iostat, iomsg)
4020     !< Formatted output.
4021     class(string),           intent(in)      :: dtv           !< The string.
4022     integer,                intent(in)      :: unit          !< Logical unit.
4023     character(kind=CK, len=*), intent(in)    :: iotype        !< Edit descriptor.
4024     integer,                intent(in)      :: v_list(:)     !< Edit descriptor list.
4025     integer,                intent(out)     :: iostat        !< IO status code.
4026     character(kind=CK, len=*), intent(inout) :: iomsg        !< IO status message.
4027
4028     if (allocated(dtv%raw)) then
4029         write(unit, "(A)", iostat=iostat, iomsg=iomsg)dtv%raw
4030     else
4031         write(unit, "(A)", iostat=iostat, iomsg=iomsg)''
4032     endif
4033 endsubroutine write_formatted
4034
4035 subroutine read_unformatted(dtv, unit, iostat, iomsg)
4036     !< Unformatted input.
4037     !<
4038     !< @bug Change temporary acks: find a more precise length of the input string and avoid the trimming!
4039     class(string),           intent(inout) :: dtv           !< The string.
4040     integer,                intent(in)      :: unit          !< Logical unit.
4041     integer,                intent(out)     :: iostat        !< IO status code.
4042     character(kind=CK, len=*), intent(inout) :: iomsg        !< IO status message.
4043     character(kind=CK, len=100)                :: temporary !< Temporary storage string.
4044
4045     read(unit, iostat=iostat, iomsg=iomsg)temporary

```

```

4046 dtv%raw = trim(temporary)
4047 endsubroutine read_unformatted
4048
4049 subroutine write_unformatted(dtv, unit, iostat, iomsg)
4050 !< Unformatted output.
4051 class(string), intent(in) :: dtv !< The string.
4052 integer, intent(in) :: unit !< Logical unit.
4053 integer, intent(out) :: iostat !< IO status code.
4054 character(kind=CK, len=*), intent(inout) :: iomsg !< IO status message.
4055
4056 if (allocated(dtv%raw)) then
4057   write(unit, iostat=iostat, iomsg=iomsg)dtv%raw
4058 else
4059   write(unit, iostat=iostat, iomsg=iomsg)''
4060 endif
4061 endsubroutine write_unformatted
4062 #endif
4063
4064 ! miscellanea
4065 elemental function replace_one_occurrence(self, old, new) result(replaced)
4066 !< Return a string with the first occurrence of substring old replaced by new.
4067 !<
4068 !< @note The doctest is not necessary, this being tested by [[string:replace]].
4069 class(string), intent(in) :: self !< The string.
4070 character(kind=CK, len=*), intent(in) :: old !< Old substring.
4071 character(kind=CK, len=*), intent(in) :: new !< New substring.
4072 type(string) :: replaced !< The string with old replaced by new.
4073 integer :: pos !< Position from which replace old.
4074 #ifdef __GFORTRAN__
4075   character(kind=CK, len=:), allocatable :: temporary !< Temporary storage, workaround for GNU bug.
4076 #endif
4077
4078 if (allocated(self%raw)) then
4079   replaced = self
4080   pos = index(string=self%raw, substring=old)
4081   if (pos>0) then
4082 #ifdef __GFORTRAN__
4083     temporary = self%raw
4084     if (pos==1) then
4085       replaced%raw = new//temporary(len(old)+1:)
4086     else
4087       replaced%raw = temporary(1:pos-1)//new//temporary(pos+len(old):)
4088     endif
4089 #else

```

```

4090         if (pos==1) then
4091             replaced%raw = new//self%raw(len(old)+1:)
4092         else
4093             replaced%raw = self%raw(1:pos-1)//new//self%raw(pos+len(old):)
4094         endif
4095 #endif
4096         endif
4097     endif
4098     endfunction replace_one_occurrence
4099
4100     ! non type-bound-procedures
4101     subroutine get_delimiter_mode(unit, delim, iostat, iomsg)
4102     !< Get the DELIM changeable connection mode for the given unit.
4103     !<
4104     !< If the unit is connected to an internal file, then the default value of NONE is always returned.
4105     use, intrinsic :: iso_fortran_env, only : iostat_inquire_internal_unit
4106     integer,          intent(in)      :: unit          !< The unit for the connection.
4107     character(len=1, kind=CK), intent(out) :: delim      !< Represents the value of the DELIM mode.
4108     integer,          intent(out)    :: iostat         !< IOSTAT error code, non-zero on error.
4109     character(*),      intent(inout)  :: iomsg          !< IOMSG explanatory message - only defined if iostat is non-zero
4110     character(10)      :: delim_buffer !< Buffer for INQUIRE about DELIM, sized for APOSTROHPE.
4111     character(len(iomsg)) :: local_iomsg !< Local variant of iomsg, so it doesn't get inappropriately red
4112
4113     ! get the string representation of the changeable mode
4114     inquire(unit, delim=delim_buffer, iostat=iostat, iomsg=local_iomsg)
4115     if (iostat == iostat_inquire_internal_unit) then
4116         ! no way of determining the DELIM mode for an internal file
4117         iostat = 0
4118         delim = ''
4119         return
4120     elseif (iostat /= 0) then
4121         iomsg = local_iomsg
4122         return
4123     endif
4124     ! interpret the DELIM string
4125     if (delim_buffer == 'QUOTE') then
4126         delim = '"'
4127     elseif (delim_buffer == 'APOSTROPHE') then
4128         delim = "'"
4129     else
4130         delim = ''
4131     endif
4132     endsubroutine get_delimiter_mode
4133

```

```

4134 subroutine get_next_non_blank_character_this_record(unit, ch, iostat, iomsg)
4135 !< Get the next non-blank character in the current record.
4136 integer, intent(in) :: unit !< Logical unit.
4137 character(kind=CK, len=1), intent(out) :: ch !< The non-blank character read. Not valid if IOSTAT is non-zero.
4138 integer, intent(out) :: iostat !< IO status code.
4139 character(kind=CK, len=*), intent(inout) :: iomsg !< IO status message.
4140
4141 do
4142 ! we specify non-advancing, just in case we want this callable outside the context of a child input statement
4143 ! the PAD specifier simply saves the need for the READ statement to define ch if EOR is hit
4144 ! read(unit, "(A)", iostat=iostat, iomsg=iomsg, advance='NO') ch
4145 ! ...but that causes ifort to blow up at runtime
4146 read(unit, "(A)", iostat=iostat, iomsg=iomsg, pad='NO') ch
4147 if (iostat /= 0) return
4148 if (ch /= '') exit
4149 enddo
4150 endsubroutine get_next_non_blank_character_this_record
4151
4152 subroutine get_next_non_blank_character_any_record(unit, ch, iostat, iomsg)
4153 !< Get the next non-blank character, advancing records if necessary.
4154 integer, intent(in) :: unit !< Logical unit.
4155 character(kind=CK, len=1), intent(out) :: ch !< The non-blank character read. Not valid if IOSTAT is non-zero.
4156 integer, intent(out) :: iostat !< IO status code.
4157 character(kind=CK, len=*), intent(inout) :: iomsg !< IO status message.
4158 character(len(iomsg)) :: local_iomsg !< Local variant of iomsg, so it doesn't get inappropriately rede
4159
4160 do
4161 call get_next_non_blank_character_this_record(unit=unit, ch=ch, iostat=iostat, iomsg=local_iomsg)
4162 if (is_iostat_eor(iostat)) then
4163 ! try again on the next record
4164 read (unit, "(/)", iostat=iostat, iomsg=iomsg)
4165 if (iostat /= 0) return
4166 elseif (iostat /= 0) then
4167 ! some sort of problem
4168 iomsg = local_iomsg
4169 return
4170 else
4171 ! got it
4172 exit
4173 endif
4174 enddo
4175 endsubroutine get_next_non_blank_character_any_record
4176
4177 subroutine get_decimal_mode(unit, decimal_point, iostat, iomsg)

```

```

4178      !< Get the DECIMAL changeable connection mode for the given unit.
4179      !<
4180      !< If the unit is connected to an internal file, then the default value of DECIMAL is always returned. This may not be th
4181      !< actual value in force at the time of the call to this procedure.
4182      use, intrinsic :: iso_fortran_env, only : iostat_inquire_internal_unit
4183      integer,           intent(in)      :: unit           !< Logical unit.
4184      logical,           intent(out)     :: decimal_point  !< True if the decimal mode is POINT, false otherwise.
4185      integer,           intent(out)     :: iostat         !< IO status code.
4186      character(kind=CK, len=*), intent(inout) :: iomsg      !< IO status message.
4187      character(5)       :: decimal_buffer !< Buffer for INQUIRE about DECIMAL, sized for POINT or COMMA.
4188      character(len(iomsg)) :: local_iomsg !< Local iomsg, so it doesn't get inappropriately redefined.
4189
4190      inquire(unit, decimal=decimal_buffer, iostat=iostat, iomsg=local_iomsg)
4191      if (iostat == iostat_inquire_internal_unit) then
4192          ! no way of determining the decimal mode for an internal file
4193          iostat = 0
4194          decimal_point = .true.
4195          return
4196      else if (iostat /= 0) then
4197          iomsg = local_iomsg
4198          return
4199      endif
4200      decimal_point = decimal_buffer == 'POINT'
4201      endsubroutine get_decimal_mode
4202  endmodule stringifor_string_t

```