```f90
!< StringiFor, definition of `string` type.
module stringifor_string_t
!< StringiFor, definition of `string` type.
use, intrinsic :: iso_fortran_env, only : iostat_eor
! use befor64, only : b64_decode, b64_encode
use penf, only : I1P, I2P, I4P, I8P, R4P, R8P, R16P, str

implicit none
private
save
! expose StingiFor overloaded builtins and operators
! public :: adjustl, adjustr, count, index, len_trim, repeat, scan, trim, verify
public :: adjustl, adjustr, count, index, len_trim, repeat, scan, trim, verify
#ifndef __GFORTRAN__
public :: assignment(=), operator(//), operator(.cat.), operator(==), &
          operator(/=), operator(<), operator(<=), operator(>=), operator(>)
#endif
! expose StingiFor objects
public :: CK
public :: string

integer, parameter :: CK = selected_char_kind('DEFAULT') !< Default character kind.

type :: string
  !< OOP designed string class.
  private
  character(kind=CK, len=:), allocatable :: raw !< Raw data.
  contains
    ! public methods
    ! builtins replacements
    procedure, pass(self) :: adjustl  => sadjustl                  !< Adjustl replacement.
    procedure, pass(self) :: adjustr  => sadjustr                  !< Adjustr replacement.
    procedure, pass(self) :: count    => scount                    !< Count replacement.
    generic               :: index    => sindex_string_string, &
                                         sindex_string_character  !< Index replacement.
    procedure, pass(self) :: len      => slen                      !< Len replacement.
    procedure, pass(self) :: len_trim => slen_trim                 !< Len_trim replacement.
    generic               :: repeat   => srepeat_string_string, &
                                         srepeat_character_string !< Repeat replacement.
    generic               :: scan     => sscan_string_string,    &
                                         sscan_string_character   !< Scan replacement.
    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,&
```

```
                                to_real_R4P,   &
#ifdef _R16P_SUPPORTED
                                to_real_R8P,   &
                                to_real_R16P     !< Cast string to number.
#else
                                to_real_R8P      !< Cast string to number.
#endif
    procedure, pass(self) :: unescape          !< Unescape double backslashes (or custom escaped character).
    procedure, pass(self) :: unique            !< Reduce to one (unique) multiple occurrences of a substring into a string.
    procedure, pass(self) :: upper             !< Return a string with all uppercase characters.
    procedure, pass(self) :: write_file        !< Write a single string stream into file.
    procedure, pass(self) :: write_line        !< Write line (record) to a connected unit.
    procedure, pass(self) :: write_lines       !< Write lines (records) to a connected unit.
    ! inquire methods
    procedure, pass(self) :: end_with      !< Return true if a string ends with a specified suffix.
    procedure, pass(self) :: is_allocated  !< Return true if the string is allocated.
    procedure, pass(self) :: is_digit      !< Return true if all characters in the string are digits.
    procedure, pass(self) :: is_integer    !< Return true if the string contains an integer.
    procedure, pass(self) :: is_lower      !< Return true if all characters in the string are lowercase.
    procedure, pass(self) :: is_number     !< Return true if the string contains a number (real or integer).
    procedure, pass(self) :: is_real       !< Return true if the string contains an real.
    procedure, pass(self) :: is_upper      !< Return true if all characters in the string are uppercase.
    procedure, pass(self) :: start_with    !< Return true if a string starts with a specified prefix.
    ! operators
    generic :: assignment(=) => string_assign_string,       &
                                string_assign_character,    &
                                string_assign_integer_I1P,  &
                                string_assign_integer_I2P,  &
                                string_assign_integer_I4P,  &
                                string_assign_integer_I8P,  &
                                string_assign_real_R4P,     &
#ifdef _R16P_SUPPORTED
                                string_assign_real_R8P,     &
                                string_assign_real_R16P              !< Assignment operator overloading.
#else
                                string_assign_real_R8P               !< Assignment operator overloading.
#endif
    generic :: operator(//) => string_concat_string,    &
                               string_concat_character, &
                               character_concat_string               !< Concatenation operator overloading.
    generic :: operator(.cat.) => string_concat_string_string,    &
                                  string_concat_character_string, &
                                  character_concat_string_string     !< Concatenation operator (string output) overloading.
    generic :: operator(==) => string_eq_string,    &
```

```
                                string_eq_character, &
                                character_eq_string                 !< Equal operator overloading.
    generic :: operator(/=) => string_ne_string,    &
                                string_ne_character, &
                                character_ne_string                 !< Not equal operator overloading.
    generic :: operator(<) => string_lt_string,    &
                              string_lt_character, &
                              character_lt_string                   !< Lower than operator overloading.
    generic :: operator(<=) => string_le_string,    &
                               string_le_character, &
                               character_le_string                  !< Lower equal than operator overloading.
    generic :: operator(>=) => string_ge_string,    &
                               string_ge_character, &
                               character_ge_string                  !< Greater equal than operator overloading.
    generic :: operator(>) => string_gt_string,    &
                              string_gt_character, &
                              character_gt_string                   !< Greater than operator overloading.
    ! IO
#ifndef __GFORTRAN__
    generic :: read(formatted)    => read_formatted    !< Formatted input.
    generic :: write(formatted)   => write_formatted   !< Formatted output.
    generic :: read(unformatted)  => read_unformatted  !< Unformatted input.
    generic :: write(unformatted) => write_unformatted !< Unformatted output.
#endif
    ! private methods
    ! builtins replacements
    procedure, private, pass(self) :: sindex_string_string      !< Index replacement.
    procedure, private, pass(self) :: sindex_string_character   !< Index replacement.
    procedure, private, pass(self) :: srepeat_string_string     !< Repeat replacement.
    procedure, private, nopass     :: srepeat_character_string  !< Repeat replacement.
    procedure, private, pass(self) :: sscan_string_string       !< Scan replacement.
    procedure, private, pass(self) :: sscan_string_character    !< Scan replacement.
    procedure, private, pass(self) :: sverify_string_string     !< Verify replacement.
    procedure, private, pass(self) :: sverify_string_character  !< Verify replacement.
    ! auxiliary methods
    procedure, private, pass(self) :: glob_character    !< Glob search (character output).
    procedure, private, pass(self) :: glob_string       !< Glob search (string output).
    procedure, private, pass(self) :: insert_string     !< Insert substring into string at a specified position.
    procedure, private, pass(self) :: insert_character  !< Insert substring into string at a specified position.
    procedure, private, pass(self) :: join_strings      !< Return join string of an array of strings.
    procedure, private, pass(self) :: join_characters   !< Return join string of an array of characters.
    procedure, private, pass(self) :: to_integer_I1P    !< Cast string to integer.
    procedure, private, pass(self) :: to_integer_I2P    !< Cast string to integer.
    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.
```

```
    procedure, private, pass(dtv) :: read_undelimited              !< Read an undelimited input.
    procedure, private, pass(dtv) :: read_undelimited_listdirected !< Read an undelimited list directed input.
    procedure, private, pass(dtv) :: write_formatted               !< Formatted output.
    procedure, private, pass(dtv) :: read_unformatted              !< Unformatted input.
    procedure, private, pass(dtv) :: write_unformatted             !< Unformatted output.
#endif
    ! miscellanea
    procedure, private, pass(self) :: replace_one_occurrence !< Replace the first occurrence of substring old by new.
endtype string

! internal parameters
character(kind=CK, len=26), parameter :: UPPER_ALPHABET = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' !< Upper case alphabet.
character(kind=CK, len=26), parameter :: LOWER_ALPHABET = 'abcdefghijklmnopqrstuvwxyz' !< Lower case alphabet.
character(kind=CK, len=1),  parameter :: SPACE          = '␣'                          !< Space character.
character(kind=CK, len=1),  parameter :: TAB            = achar(9)                     !< Tab character.
character(kind=CK, len=1),  parameter :: UIX_DIR_SEP    = char(47)                     !< Unix/Linux directories separator (
character(kind=CK, len=1),  parameter :: BACKSLASH      = char(92)                     !< Backslash character.

! overloading string name
interface string
  !< Builtin adjustl overloading.
  module procedure string_
endinterface string

#ifndef __GFORTRAN__
! operators overloading interfaces
interface operator(//)
  !< Builtin // overloading.
  module procedure string_concat_string, string_concat_character, character_concat_string
endinterface
interface assignment(=)
  !< Builtin = overloading.
  module procedure string_assign_string, string_assign_character, string_assign_integer_I1P, string_assign_integer_I2P, &
                   string_assign_integer_I4P, string_assign_integer_I8P, string_assign_real_R4P,
&
#ifdef _R16P_SUPPORTED
                   string_assign_real_R8P, string_assign_real_R16P
#else
                   string_assign_real_R8P
#endif
endinterface
interface operator(==)
  !< Builtin == overloading.
  module procedure string_eq_string, string_eq_character, character_eq_string
```

```
endinterface
interface operator(/=)
  !< Builtin /= overloading.
  module procedure string_ne_string, string_ne_character, character_ne_string
endinterface
interface operator(<)
  !< Builtin < overloading.
  module procedure string_lt_string, string_lt_character, character_lt_string
endinterface
interface operator(<=)
  !< Builtin <= overloading.
  module procedure string_le_string, string_le_character, character_le_string
endinterface
interface operator(>=)
  !< Builtin >= overloading.
  module procedure string_ge_string, string_ge_character, character_ge_string
endinterface
interface operator(>)
  !< Builtin > overloading.
  module procedure string_gt_string, string_gt_character, character_gt_string
endinterface
interface operator(.cat.)
  !< .cat. overloading.
  module procedure string_concat_string_string, string_concat_character_string, character_concat_string_string
endinterface
#endif

! builtin overloading
interface adjustl
  !< Builtin adjustl overloading.
  module procedure sadjustl_character
endinterface adjustl

interface adjustr
  !< Builtin adjustr overloading.
  module procedure sadjustr_character
endinterface adjustr

interface count
  !< Builtin count overloading.
  module procedure count_substring
endinterface

interface index
```

```
   !< Builtin index overloading.
   module procedure sindex_string_string, sindex_string_character, sindex_character_string
endinterface index

!interface len
!   !< Builtin len overloading.
!   module procedure slen
!endinterface len

interface len_trim
   !< Builtin len_trim overloading.
   module procedure slen_trim
endinterface len_trim

interface repeat
   !< Builtin repeat overloading.
   module procedure srepeat_string_string
endinterface repeat

interface scan
   !< Builtin scan overloading.
   module procedure sscan_string_string, sscan_string_character, sscan_character_string
endinterface scan

interface trim
   !< Builtin trim overloading.
   module procedure strim
endinterface trim

interface verify
   !< Builtin verify overloading.
   module procedure sverify_string_string, sverify_string_character, sverify_character_string
endinterface verify

contains
    ! public non TBP

    ! creator
    pure function string_(c)
    !< Return a string given a character input.
    !<
    !<```fortran
    !< print "(L1)", string('Hello World')//''=='Hello World'
    !<```
```

```
    !=> T <<<
    character(*), intent(in) :: c        !< Character.
    type(string)             :: string_  !< String.

    string_%raw = c
    endfunction string_

    ! builtins replacements
    pure function sadjustl_character(s) result(adjusted)
    !< Left adjust a string by removing leading spaces (character output).
    !<
    !<```fortran
    !< type(string) :: astring
    !< astring = '   Hello World!'
    !< print "(L1)", adjustl(astring)=='Hello World!   '
    !<```
    !=> T <<<
    class(string), intent(in)               :: s         !< String.
    character(kind=CK, len=:), allocatable :: adjusted !< Adjusted string.

    if (allocated(s%raw)) adjusted = adjustl(s%raw)
    endfunction sadjustl_character

    pure function sadjustr_character(s) result(adjusted)
    !< Right adjust a string by removing leading spaces (character output).
    !<
    !<```fortran
    !< type(string) :: astring
    !< astring = 'Hello World!   '
    !< print "(L1)", adjustr(astring)=='   Hello World!'
    !<```
    !=> T <<<
    class(string), intent(in)               :: s         !< String.
    character(kind=CK, len=:), allocatable :: adjusted !< Adjusted string.

    if (allocated(s%raw)) adjusted = adjustr(s%raw)
    endfunction sadjustr_character

    elemental function count_substring(s, substring) result(No)
    !< Count the number of occurences of a substring into a string.
    !<
    !<```fortran
    !< print "(L1)", count('hello', substring='ll')==1
    !<```
```

```
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
```

```fortran

  elemental function sscan_character_string(s, set, back) result(i)
  !< Return the leftmost (if `back` is either absent or equals false, otherwise the rightmost) character of string that is
  !<
  !<```fortran
  !< type(string) :: string1
  !< logical       :: test_passed(2)
  !< string1 = 'llo'
  !< test_passed(1) = scan(s='Hello World Hello!', set=string1)==scan(string='Hello World Hello!', set='llo')
  !< test_passed(2) = scan(s='Hello World Hello!', set=string1, back=.true.)==scan(string='Hello World Hello!', &
  !<                                                                              set='llo', back=.true.)
  !< print '(L1)', all(test_passed)
  !<```
  !=> T <<<
  character(kind=CK, len=*), intent(in)           :: s    !< String.
  type(string),              intent(in)           :: set  !< Searched set.
  logical,                   intent(in), optional :: back !< Start of the last occurrence rather than the first.
  integer                                         :: i    !< Result of the search.

  if (allocated(set%raw)) then
    i = scan(string=s, set=set%raw, back=back)
  else
    i = 0
  endif
  endfunction sscan_character_string

  elemental function sverify_character_string(s, set, back) result(i)
  !< Return the leftmost (if `back` is either absent or equals false, otherwise the rightmost) character of string that is
  !< in `set`. If all characters of `string` are found in `set`, the result is zero.
  !<
  !<```fortran
  !< type(string) :: string1
  !< logical       :: test_passed(2)
  !< string1 = 'ell'
  !< test_passed(1) = verify(s='Hello World Hello!', set=string1)==verify(string='Hello World Hello!', set='llo')
  !< test_passed(2) = verify(s='Hello World Hello!', set=string1, back=.true.)==verify(string='Hello World Hello!', set='llo
  !<                                                                                 back=.true.)
  !< print '(L1)', all(test_passed)
  !<```
  !=> T <<<
  character(kind=CK, len=*), intent(in)           :: s    !< String.
  type(string),              intent(in)           :: set  !< Searched set.
  logical,                   intent(in), optional :: back !< Start of the last occurrence rather than the first.
  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 occurences of a substring into a string.
527     !<
528     !< @note If `ignore_isolated` is set to true the eventual "isolated" occurences 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)
```

```
#endif
         if (c2==0) return
         if (.not.ignore_isolated_) then
            No = No + 1
         else
            if (.not.((c1==1.and.c2==1) .or. (c1==len(self%raw)-len(substring)+1))) then
               No = No + 1
            endif
         endif
         c1 = c1 + c2 - 1 + len(substring)
      enddo
   endif
   endfunction scount

   elemental function sindex_string_string(self, substring, back) result(i)
   !< Return the position of the start of the first occurrence of string `substring` as a substring in `string`, counting fr
   !< If `substring` is not present in `string`, zero is returned. If the back argument is present and true, the return valu
   !< the start of the last occurrence rather than the first.
   !<
   !<```fortran
   !< type(string) :: string1
   !< type(string) :: string2
   !< logical       :: test_passed(2)
   !< string1 = 'Hello World Hello!'
   !< string2 = 'llo'
   !< test_passed(1) = string1%index(substring=string2)==index(string='Hello World Hello!', substring='llo')
   !< test_passed(2) = string1%index(substring=string2, back=.true.)==index(string='Hello World Hello!', substring='llo', &
   !<                                                                       back=.true.)
   !< print '(L1)', all(test_passed)
   !<```
   !=> T <<<
   class(string), intent(in)           :: self      !< The string.
   type(string),  intent(in)           :: substring !< Searched substring.
   logical,       intent(in), optional :: back      !< Start of the last occurrence rather than the first.
   integer                             :: i         !< Result of the search.

   if (allocated(self%raw)) then
      i = index(string=self%raw, substring=substring%raw, back=back)
   else
      i = 0
   endif
   endfunction sindex_string_string

   elemental function sindex_string_character(self, substring, back) result(i)
```

```
    !< Return the position of the start of the first occurrence of string `substring` as a substring in `string`, counting fr
    !< If `substring` is not present in `string`, zero is returned. If the back argument is present and true, the return valu
    !< the start of the last occurrence rather than the first.
    !<
    !<```fortran
    !< type(string) :: string1
    !< logical       :: test_passed(2)
    !< string1 = 'Hello World Hello!'
    !< test_passed(1) = string1%index(substring='llo ')==index(string='Hello World Hello!', substring='llo ')
    !< test_passed(2) = string1%index(substring='llo', back=.true.)==index(string='Hello World Hello!', substring='llo', back
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string),              intent(in)           :: self      !< The string.
    character(kind=CK, len=*), intent(in)           :: substring !< Searched substring.
    logical,                    intent(in), optional :: back      !< Start of the last occurrence rather than the first.
    integer                                          :: i         !< Result of the search.

    if (allocated(self%raw)) then
       i = index(string=self%raw, substring=substring, back=back)
    else
       i = 0
    endif
    endfunction sindex_string_character

    elemental function slen(self) result(l)
    !< Return the length of a string.
    !<
    !<```fortran
    !< type(string) :: astring
    !< astring = 'Hello World!    '
    !< print "(L1)", astring%len()==len('Hello World!    ')
    !<```
    !=> T <<<
    class(string), intent(in) :: self !< The string.
    integer                   :: l    !< String length.

    if (allocated(self%raw)) then
       l = len(string=self%raw)
    else
       l = 0
    endif
    endfunction slen

```

```fortran
   elemental function slen_trim(self) result(l)
   !< Return the length of a string, ignoring any trailing blanks.
   !<
   !<```fortran
   !< type(string) :: astring
   !< astring = 'Hello World!    '
   !< print "(L1)", astring%len_trim()==len_trim('Hello World!    ')
   !<```
   !=> T <<<
   class(string), intent(in) :: self !< The string.
   integer                   :: l    !< String length.

   if (allocated(self%raw)) then
      l = len_trim(string=self%raw)
   else
      l = 0
   endif
   endfunction slen_trim

   elemental function srepeat_string_string(self, ncopies) result(repeated)
   !< Concatenates several copies of an input string.
   !<
   !<```fortran
   !< type(string) :: astring
   !< astring = 'x'
   !< print "(L1)", astring%repeat(5)//''=='xxxxx'
   !<```
   !=> T <<<
   class(string), intent(in) :: self     !< String to be repeated.
   integer,       intent(in) :: ncopies  !< Number of string copies.
   type(string)              :: repeated !< Repeated string.

   repeated%raw = repeat(string=self%raw, ncopies=ncopies)
   endfunction srepeat_string_string

   elemental function srepeat_character_string(rstring, ncopies) result(repeated)
   !< Concatenates several copies of an input string.
   !<
   !<```fortran
   !< type(string) :: astring
   !< astring = 'y'
   !< print "(L1)", astring%repeat('x', 5)//''=='xxxxx'
   !<```
   !=> T <<<
```

```
    character(kind=CK, len=*), intent(in) :: rstring  !< String to be repeated.
    integer,                   intent(in) :: ncopies  !< Number of string copies.
    type(string)                          :: repeated !< Repeated string.

    repeated%raw = repeat(string=rstring, ncopies=ncopies)
    endfunction srepeat_character_string

    elemental function sscan_string_string(self, set, back) result(i)
    !< Return the leftmost (if `back` is either absent or equals false, otherwise the rightmost) character of string that is
    !<
    !<```fortran
    !< type(string) :: string1
    !< type(string) :: string2
    !< logical      :: test_passed(2)
    !< string1 = 'Hello World Hello!'
    !< string2 = 'llo'
    !< test_passed(1) = string1%scan(set=string2)==scan(string='Hello World Hello!', set='llo')
    !< test_passed(2) = string1%scan(set=string2, back=.true.)==scan(string='Hello World Hello!', set='llo', back=.true.)
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string), intent(in)           :: self  !< The string.
    type(string),  intent(in)           :: set   !< Searched set.
    logical,       intent(in), optional :: back  !< Start of the last occurrence rather than the first.
    integer                             :: i     !< Result of the search.

    if (allocated(self%raw).and.allocated(set%raw)) then
      i = scan(string=self%raw, set=set%raw, back=back)
    else
      i = 0
    endif
    endfunction sscan_string_string

    elemental function sscan_string_character(self, set, back) result(i)
    !< Return the leftmost (if `back` is either absent or equals false, otherwise the rightmost) character of string that is
    !<
    !<```fortran
    !< type(string) :: string1
    !< logical      :: test_passed(2)
    !< string1 = 'Hello World Hello!'
    !< test_passed(1) = string1%scan(set='llo')==scan(string='Hello World Hello!', set='llo')
    !< test_passed(2) = string1%scan(set='llo', back=.true.)==scan(string='Hello World Hello!', set='llo', back=.true.)
    !< print '(L1)', all(test_passed)
    !<```
```

```
    !=> T <<<
    class(string),              intent(in)           :: self   !< The string.
    character(kind=CK, len=*), intent(in)           :: set    !< Searched set.
    logical,                    intent(in), optional :: back   !< Start of the last occurrence rather than the first.
    integer                                          :: i      !< Result of the search.

    if (allocated(self%raw)) then
      i = scan(string=self%raw, set=set, back=back)
    else
      i = 0
    endif
    endfunction sscan_string_character

    elemental function strim(self) result(trimmed)
    !< Remove trailing spaces.
    !<
    !<```fortran
    !< type(string) :: astring
    !< astring = 'Hello World!    '
    !< print "(L1)", astring%trim()==trim('Hello World!    ')
    !<```
    !=> T <<<
    class(string), intent(in) :: self    !< The string.
    type(string)              :: trimmed !< Trimmed string.

    trimmed = self
    if (allocated(trimmed%raw)) trimmed%raw = trim(trimmed%raw)
    endfunction strim

    elemental function sverify_string_string(self, set, back) result(i)
    !< Return the leftmost (if `back` is either absent or equals false, otherwise the rightmost) character of string that is
    !< in `set`. If all characters of `string` are found in `set`, the result is zero.
    !<
    !<```fortran
    !< type(string) :: string1
    !< type(string) :: string2
    !< logical      :: test_passed(2)
    !< string1 = 'Hello World Hello!'
    !< string2 = 'llo'
    !< test_passed(1) = string1%verify(set=string2)==verify(string='Hello World Hello!', set='llo')
    !< test_passed(2) = string1%verify(set=string2, back=.true.)==verify(string='Hello World Hello!', set='llo', back=.true.)
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
```

```fortran
    class(string), intent(in)           :: self   !< The string.
    type(string),  intent(in)           :: set    !< Searched set.
    logical,       intent(in), optional :: back   !< Start of the last occurrence rather than the first.
    integer                             :: i      !< Result of the search.

    if (allocated(self%raw).and.allocated(set%raw)) then
      i = verify(string=self%raw, set=set%raw, back=back)
    else
      i = 0
    endif
    endfunction sverify_string_string

    elemental function sverify_string_character(self, set, back) result(i)
    !< Return the leftmost (if `back` is either absent or equals false, otherwise the rightmost) character of string that is
    !< in `set`. If all characters of `string` are found in `set`, the result is zero.
    !<
    !<```fortran
    !< type(string) :: string1
    !< logical       :: test_passed(2)
    !< string1 = 'Hello World Hello!'
    !< test_passed(1) = string1%verify(set='llo ')==verify(string='Hello World Hello!', set='llo ')
    !< test_passed(2) = string1%verify(set='llo', back=.true.)==verify(string='Hello World Hello!', set='llo', back=.true.)
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string),              intent(in)           :: self   !< The string.
    character(kind=CK, len=*), intent(in)           :: set    !< Searched set.
    logical,                    intent(in), optional :: back   !< Start of the last occurrence rather than the first.
    integer                                          :: i      !< Result of the search.

    if (allocated(self%raw)) then
      i = verify(string=self%raw, set=set, back=back)
    else
      i = 0
    endif
    endfunction sverify_string_character

    ! auxiliary methods
    elemental function basedir(self, sep)
    !< Return the base directory name of a string containing a file name.
    !<
    !<```fortran
    !< type(string) :: string1
    !< logical       :: test_passed(4)
```

```
!< string1 = '/bar/foo.tar.bz2'
!< test_passed(1) = string1%basedir()//''=='/bar'
!< string1 = './bar/foo.tar.bz2'
!< test_passed(2) = string1%basedir()//''=='./bar'
!< string1 = 'bar/foo.tar.bz2'
!< test_passed(3) = string1%basedir()//''=='bar'
!< string1 = '\bar\foo.tar.bz2'
!< test_passed(4) = string1%basedir(sep='\')//''=='\bar'
!< print '(L1)', all(test_passed)
!<```
!=> T <<<
class(string),              intent(in)           :: self     !< The string.
character(kind=CK, len=*), intent(in), optional :: sep      !< Directory separator.
type(string)                                    :: basedir  !< Base directory name.
character(kind=CK, len=:), allocatable          :: sep_     !< Separator, default value.
integer                                         :: pos      !< Character position.

if (allocated(self%raw)) then
  sep_ = UIX_DIR_SEP ; if (present(sep)) sep_ = sep
  basedir = self
  pos = index(self%raw, sep_, back=.true.)
  if (pos>0) basedir%raw = self%raw(1:pos-1)
endif
endfunction basedir

elemental function basename(self, sep, extension, strip_last_extension)
!< Return the base file name of a string containing a file name.
!<
!< Optionally, the extension is also stripped if provided or the last one if required, e.g.
!<
!<```fortran
!< type(string) :: astring
!< logical       :: test_passed(5)
!< astring = 'bar/foo.tar.bz2'
!< test_passed(1) = astring%basename()//''=='foo.tar.bz2'
!< test_passed(2) = astring%basename(extension='.tar.bz2')//''=='foo'
!< test_passed(3) = astring%basename(strip_last_extension=.true.)//''=='foo.tar'
!< astring = '\bar\foo.tar.bz2'
!< test_passed(4) = astring%basename(sep='\')//''=='foo.tar.bz2'
!< astring = 'bar'
!< test_passed(5) = astring%basename(strip_last_extension=.true.)//''=='bar'
!< print '(L1)', all(test_passed)
!<```
!=> 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_ = UIX_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
```

```fortran
  endfunction basename

  elemental function camelcase(self, sep)
  !< Return a string with all words capitalized without spaces.
  !<
  !< @note Multiple subsequent separators are collapsed to one occurence.
  !<
  !<```fortran
  !< type(string) :: astring
  !< astring = 'caMeL caSe var'
  !< print '(L1)', astring%camelcase()//''=='CamelCaseVar'
  !<```
  !=> T <<<
  class(string),              intent(in)           :: self      !< The string.
  character(kind=CK, len=*), intent(in), optional :: sep       !< Separator.
  type(string)                                    :: camelcase !< Camel case string.
  type(string), allocatable                       :: tokens(:) !< String tokens.

  if (allocated(self%raw)) then
    call self%split(tokens=tokens, sep=sep)
    tokens = tokens%capitalize()
    camelcase = camelcase%join(array=tokens)
  endif
  endfunction camelcase

  elemental function capitalize(self) result(capitalized)
  !< Return a string with its first character capitalized and the rest lowercased.
  !<
  !<```fortran
  !< type(string) :: astring
  !< astring = 'say all Hello WorLD!'
  !< print '(L1)', astring%capitalize()//''=='Say all hello world!'
  !<```
  !=> T <<<
  class(string), intent(in) :: self        !< The string.
  type(string)              :: capitalized !< Upper case string.
  integer                   :: c           !< Character counter.

  if (allocated(self%raw)) then
    capitalized = self%lower()
    c = index(LOWER_ALPHABET, capitalized%raw(1:1))
    if (c>0) capitalized%raw(1:1) = UPPER_ALPHABET(c:c)
  endif
  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.)
```

```
    !endif
    !endfunction decode

    !elemental function encode(self, codec) result(encoded)
    !!< Return a string encoded accordingly the codec.
    !!<
    !!< @note Only BASE64 codec is currently available.
    !!<
    !!<```fortran
    !!< type(string) :: astring
    !!< astring = 'How are you?'
    !!< print '(L1)', astring%encode(codec='base64')//''=='SG93IGFyZSB5b3U/'
    !!<```
    !!=> T <<<
    !class(string),               intent(in) :: self    !< The string.
    !character(kind=CK, len=*), intent(in) :: codec   !< Encoding codec.
    !type(string)                              :: encoded !< Encoded string.

    !if (allocated(self%raw)) then
    !  encoded = codec
    !  select case(encoded%upper()//'')
    !  case('BASE64')
    !    call b64_encode(s=self%raw, code=encoded%raw)
    !  endselect
    !endif
    !endfunction encode

    elemental function escape(self, to_escape, esc) result(escaped)
    !< Escape backslashes (or custom escape character).
    !<
    !<```fortran
    !< type(string) :: astring
    !< logical       :: test_passed(2)
    !< astring = '^\s \d+\s*'
    !< test_passed(1) = astring%escape(to_escape='\')//''=='^\\s \\d+\\s*'
    !< test_passed(2) = astring%escape(to_escape='\', esc='|')//''=='^|\s |\d+|\s*'
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string),              intent(in)           :: self      !< The string.
    character(kind=CK, len=1), intent(in)           :: to_escape !< Character to be escaped.
    character(kind=CK, len=*), intent(in), optional :: esc       !< Character used to escape.
    type(string)                                    :: escaped   !< Escaped string.
    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)
```

```
    !< Pad string on the left (or right) with zeros (or other char) to fill width.
    !<
    !<```fortran
    !< type(string) :: astring
    !< logical       :: test_passed(4)
    !< astring = 'this is string example....wow!!!'
    !< test_passed(1) = astring%fill(width=40)//''=='0000000this is string example....wow!!!'
    !< test_passed(2) = astring%fill(width=50)//''=='00000000000000000this is string example....wow!!!'
    !< test_passed(3) = astring%fill(width=50, right=.true.)//''=='this is string example....wow!!!0000000000000000'
    !< test_passed(4) = astring%fill(width=40, filling_char='*')//''=='********this is string example....wow!!!'
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string),              intent(in)           :: self           !< The string.
    integer,                    intent(in)           :: width          !< Final width of filled string.
    logical,                    intent(in), optional :: right          !< Fill on the right instead of left.
    character(kind=CK, len=1), intent(in), optional :: filling_char   !< Filling character (default "0").
    type(string)                                     :: filled         !< Filled string.
    logical                                          :: right_         !< Fill on the right instead of left, local variable.
    character(kind=CK, len=1)                        :: filling_char_  !< Filling character (default "0"), local variable.

    if (allocated(self%raw)) then
       if (width>len(self%raw)) then
          right_ = .false. ; if (present(right)) right_ = right
          filling_char_ = '0' ; if (present(filling_char)) filling_char_ = filling_char
          if (.not.right_) then
             filled%raw = repeat(filling_char_, width-len(self%raw))//self%raw
          else
             filled%raw = self%raw//repeat(filling_char_, width-len(self%raw))
          endif
       endif
    endif
    endfunction fill

    elemental subroutine free(self)
    !< Free dynamic memory.
    !<
    !<```fortran
    !< type(string) :: astring
    !< astring = 'this is string example....wow!!!'
    !< call astring%free
    !< print '(L1)', astring%is_allocated().eqv..false.
    !<```
    !=> T <<<
```

```fortran
    class(string), intent(inout) :: self !< The string.

    if (allocated(self%raw)) deallocate(self%raw)
    endsubroutine free

    subroutine glob_character(self, pattern, list)
    !< Glob search (character output), finds all the pathnames matching a given pattern according to the rules used by the Un
    !<
    !< @note Method not portable: works only on Unix/GNU Linux OS.
    !<
    !<```fortran
    !< type(string)                      :: astring
    !< character(len=:), allocatable     :: alist_chr(:)
    !< integer, parameter                :: Nf=5
    !< character(14)                     :: files(1:Nf)
    !< integer                           :: file_unit
    !< integer                           :: f
    !< integer                           :: ff
    !< logical                           :: test_passed
    !< do f=1, Nf
    !<     files(f) = astring%tempname(prefix='foo-')
    !<     open(newunit=file_unit, file=files(f))
    !<     write(file_unit, *)f
    !<     close(unit=file_unit)
    !< enddo
    !< call astring%glob(pattern='foo-*', list=alist_chr)
    !< do f=1, Nf
    !<     open(newunit=file_unit, file=files(f))
    !<     close(unit=file_unit, status='delete')
    !< enddo
    !< test_passed = .false.
    !< outer_chr: do f=1, size(alist_chr, dim=1)
    !<     do ff=1, Nf
    !<         test_passed = alist_chr(f) == files(ff)
    !<         if (test_passed) cycle outer_chr
    !<     enddo
    !< enddo outer_chr
    !< print '(L1)', test_passed
    !<```
    !=> T <<<
    class(string),                  intent(in)  :: self           !< The string.
    character(*),                   intent(in)  :: pattern        !< Given pattern.
    character(len=:), allocatable, intent(out) :: list(:)        !< List of matching pathnames.
    type(string), allocatable                   :: list_(:)       !< List of matching pathnames.
```

```
   integer(I4P)                                :: max_len         !< Maximum length.
   integer(I4P)                                :: matches_number  !< Matches number.
   integer(I4P)                                :: m               !< Counter.

   call self%glob(pattern=pattern, list=list_)
   if (allocated(list_)) then
      matches_number = size(list_, dim=1)
      max_len = 0
      do m=1, matches_number
         max_len = max(max_len, list_(m)%len())
      enddo
      allocate(character(max_len) :: list(1:matches_number))
      do m=1, matches_number
         list(m) = list_(m)%chars()
      enddo
   endif
   endsubroutine glob_character

   subroutine glob_string(self, pattern, list)
   !< Glob search (string output), finds all the pathnames matching a given pattern according to the rules used by the Unix
   !<
   !< @note Method not portable: works only on Unix/GNU Linux OS.
   !<
   !<```fortran
   !< type(string)                    :: astring
   !< type(string),      allocatable  :: alist_str(:)
   !< integer, parameter               :: Nf=5
   !< character(14)                    :: files(1:Nf)
   !< integer                          :: file_unit
   !< integer                          :: f
   !< integer                          :: ff
   !< logical                          :: test_passed
   !<
   !< do f=1, Nf
   !<     files(f) = astring%tempname(prefix='foo-')
   !<     open(newunit=file_unit, file=files(f))
   !<     write(file_unit, *)f
   !<     close(unit=file_unit)
   !< enddo
   !< call astring%glob(pattern='foo-*', list=alist_str)
   !< do f=1, Nf
   !<     open(newunit=file_unit, file=files(f))
   !<     close(unit=file_unit, status='delete')
   !< enddo
```

```
    !< test_passed = .false.
    !< outer_str: do f=1, size(alist_str, dim=1)
    !<     do ff=1, Nf
    !<         test_passed = alist_str(f) == files(ff)
    !<         if (test_passed) cycle outer_str
    !<     enddo
    !< enddo outer_str
    !< print '(L1)', test_passed
    !<```
    !=> T <<<
    class(string),              intent(in)  :: self     !< The string.
    character(*),               intent(in)  :: pattern  !< Given pattern.
    type(string), allocatable , intent(out) :: list(:)  !< List of matching pathnames.
    type(string)                            :: tempfile !< Safe temporary file.
    character(len=:), allocatable           :: tempname !< Safe temporary name.
    integer(I4P)                            :: tempunit !< Unit of temporary file.

    tempname = self%tempname()
    call execute_command_line('ls␣-1␣'//trim(adjustl(pattern))//'␣>␣'//tempname)
    call tempfile%read_file(file=tempname)
    call tempfile%split(sep=new_line('a'), tokens=list)
    open(newunit=tempunit, file=tempname)
    close(unit=tempunit, status='delete')
    endsubroutine glob_string

    elemental function insert_character(self, substring, pos) result(inserted)
    !< Insert substring into string at a specified position.
    !<
    !<```fortran
    !< type(string)                    :: astring
    !< character(len=:), allocatable :: acharacter
    !< logical                         :: test_passed(5)
    !< astring = 'this is string example wow!!!'
    !< acharacter = '... '
    !< test_passed(1) = astring%insert(substring=acharacter, pos=1)//''=='... this is string example wow!!!'
    !< test_passed(2) = astring%insert(substring=acharacter, pos=23)//''=='this is string example...  wow!!!'
    !< test_passed(3) = astring%insert(substring=acharacter, pos=29)//''=='this is string example wow!!!... '
    !< test_passed(4) = astring%insert(substring=acharacter, pos=-1)//''=='... this is string example wow!!!'
    !< test_passed(5) = astring%insert(substring=acharacter, pos=100)//''=='this is string example wow!!!... '
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string),    intent(in) :: self      !< The string.
    character(len=*), intent(in) :: substring !< Substring.
```

```fortran
   integer,           intent(in) :: pos       !< Position from which insert substring.
   type(string)                  :: inserted  !< Inserted string.
   integer                       :: safepos   !< Safe position from which insert substring.

   if (allocated(self%raw)) then
      inserted = self
      safepos = min(max(1, pos), len(self%raw))
      if (safepos==1) then
         inserted%raw = substring//self%raw
      elseif (safepos==len(self%raw)) then
         inserted%raw = self%raw//substring
      else
         inserted%raw = self%raw(1:safepos-1)//substring//self%raw(safepos:)
      endif
   else
      inserted%raw = substring
   endif
   endfunction insert_character

   elemental function insert_string(self, substring, pos) result(inserted)
   !< Insert substring into string at a specified position.
   !<
   !<```fortran
   !< type(string) :: astring
   !< type(string) :: anotherstring
   !< logical       :: test_passed(5)
   !< astring = 'this is string example wow!!!'
   !< anotherstring = '... '
   !< test_passed(1) = astring%insert(substring=anotherstring, pos=1)//''=='... this is string example wow!!!'
   !< test_passed(2) = astring%insert(substring=anotherstring, pos=23)//''=='this is string example...  wow!!!'
   !< test_passed(3) = astring%insert(substring=anotherstring, pos=29)//''=='this is string example wow!!!... '
   !< test_passed(4) = astring%insert(substring=anotherstring, pos=-1)//''=='... this is string example wow!!!'
   !< test_passed(5) = astring%insert(substring=anotherstring, pos=100)//''=='this is string example wow!!!... '
   !< print '(L1)', all(test_passed)
   !<```
   !=> T <<<
   class(string), intent(in) :: self      !< The string.
   type(string),  intent(in) :: substring !< Substring.
   integer,       intent(in) :: pos       !< Position from which insert substring.
   type(string)              :: inserted  !< Inserted string.
   integer                   :: safepos   !< Safe position from which insert substring.

   if (allocated(self%raw)) then
      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     !<```
```

```
!=> T <<<
class(string),              intent(in)           :: self       !< The string.
type(string),               intent(in)           :: array(1:) !< Array to be joined.
character(kind=CK, len=*), intent(in), optional :: sep        !< Separator.
type(string)                                     :: join       !< The join of array.
character(kind=CK, len=:), allocatable           :: sep_       !< Separator, default value.
integer                                          :: a          !< Counter.

if (allocated(self%raw)) then
   sep_ = self%raw
else
   sep_ = ''
endif
if (present(sep)) sep_ = sep
join = ''
do a=2, size(array, dim=1)
   if (allocated(array(a)%raw)) join%raw = join%raw//sep_//array(a)%raw
enddo
if (allocated(array(1)%raw)) then
   join%raw = array(1)%raw//join%raw
else
   join%raw = join%raw(len(sep_)+1:len(join%raw))
endif
endfunction join_strings

pure function join_characters(self, array, sep) result(join)
!< Return a string that is a join of an array of characters.
!<
!< 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
!< passing a custom separator.
!<
!<```fortran
!< type(string) :: astring
!< character(5) :: characters(3)
!< logical      :: test_passed(6)
!< characters(1) = 'one'
!< characters(2) = 'two'
!< characters(3) = 'three'
!< test_passed(1) = (astring%join(array=characters)//''==characters(1)//characters(2)//characters(3))
!< test_passed(2) = (astring%join(array=characters, sep='-')//''==characters(1)//'-'//characters(2)//'-'//characters(3))
!< characters(1) = ''
!< characters(2) = 'two'
!< characters(3) = 'three'
!< test_passed(3) = (astring%join(array=characters, sep='-')//''==characters(2)//'-'//characters(3))
```

```
!< characters(1) = 'one'
!< characters(2) = 'two'
!< characters(3) = ''
!< test_passed(4) = (astring%join(array=characters, sep='-')//''==characters(1)//'-'//characters(2))
!< characters(1) = 'one'
!< characters(2) = ''
!< characters(3) = 'three'
!< test_passed(5) = (astring%join(array=characters, sep='-')//''==characters(1)//'-'//characters(3))
!< characters(1) = 'one'
!< characters(2) = 'two'
!< characters(3) = 'three'
!< astring = '_'
!< test_passed(6) = (astring%join(array=characters)//''==characters(1)//'_'//characters(2)//'_'//characters(3))
!< print '(L1)', all(test_passed)
!<```
!=> T <<<
class(string),              intent(in)           :: self       !< The string.
character(kind=CK, len=*), intent(in)           :: array(1:) !< Array to be joined.
character(kind=CK, len=*), intent(in), optional :: sep        !< Separator.
type(string)                                    :: join       !< The join of array.
character(kind=CK, len=:), allocatable          :: sep_       !< Separator, default value.
integer                                         :: a          !< Counter.

if (allocated(self%raw)) then
   sep_ = self%raw
else
   sep_ = ''
endif
if (present(sep)) sep_ = sep
join = ''
do a=2, size(array, dim=1)
   if (array(a)/='') join%raw = join%raw//sep_//array(a)
enddo
if (array(1)/='') then
   join%raw = array(1)//join%raw
else
   join%raw = join%raw(len(sep_)+1:len(join%raw))
endif
endfunction join_characters

elemental function lower(self)
!< Return a string with all lowercase characters.
!<
!<```fortran
```

```
!< type(string) :: astring
!< logical       :: test_passed(1)
!< astring = 'Hello WorLD!'
!< test_passed(1) = astring%lower()//''=='hello world!'
!< print '(L1)', all(test_passed)
!<```
!=> T <<<
class(string), intent(in) :: self  !< The string.
type(string)              :: lower !< Upper case string.
integer                   :: n1    !< Characters counter.
integer                   :: n2    !< Characters counter.

if (allocated(self%raw)) then
   lower = self
   do n1=1, len(self%raw)
      n2 = index(UPPER_ALPHABET, self%raw(n1:n1))
      if (n2>0) lower%raw(n1:n1) = LOWER_ALPHABET(n2:n2)
   enddo
endif
endfunction lower

pure function partition(self, sep) result(partitions)
!< Split string at separator and return the 3 parts (before, the separator and after).
!<
!<```fortran
!< type(string) :: astring
!< type(string) :: strings(3)
!< logical       :: test_passed(3)
!< astring = 'Hello WorLD!'
!< strings = astring%partition(sep='lo Wo')
!< test_passed(1) = (strings(1)//''=='Hel'.and.strings(2)//''=='lo Wo'.and.strings(3)//''=='rLD!')
!< strings = astring%partition(sep='Hello')
!< test_passed(2) = (strings(1)//''==''.and.strings(2)//''=='Hello'.and.strings(3)//''==' WorLD!')
!< astring = 'Hello WorLD!'
!< strings = astring%partition()
!< test_passed(3) = (strings(1)//''=='Hello'.and.strings(2)//''==' '.and.strings(3)//''=='WorLD!')
!< print '(L1)', all(test_passed)
!<```
!=> T <<<
class(string),              intent(in)           :: self             !< The string.
character(kind=CK, len=*), intent(in), optional :: sep              !< Separator.
type(string)                                    :: partitions(1:3) !< Partions: before the separator, the separator itself
                                                                    !< after the separator.
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
```

```
!< integer                       :: scratch
!< integer                       :: l
!< logical                       :: test_passed(9)
!< line(1) = ' Hello World!    '
!< line(2) = 'How are you?   '
!< line(3) = '    All say: "Fine thanks"'
!< open(newunit=scratch, file='read_file_test.tmp')
!< write(scratch, "(A)") line(1)%chars()
!< write(scratch, "(A)") line(2)%chars()
!< write(scratch, "(A)") line(3)%chars()
!< close(scratch)
!< call astring%read_file(file='read_file_test.tmp', iostat=iostat, iomsg=iomsg)
!< call astring%split(tokens=strings, sep=new_line('a'))
!< test_passed(1) = (size(strings, dim=1)==size(line, dim=1))
!< do l=1, size(strings, dim=1)
!<    test_passed(l+1) = (strings(l)==line(l))
!< enddo
!< open(newunit=scratch, file='read_file_test.tmp', form='UNFORMATTED', access='STREAM')
!< write(scratch) line(1)%chars()//new_line('a')
!< write(scratch) line(2)%chars()//new_line('a')
!< write(scratch) line(3)%chars()//new_line('a')
!< close(scratch)
!< call astring%read_file(file='read_file_test.tmp', form='unformatted', iostat=iostat, iomsg=iomsg)
!< call astring%split(tokens=strings, sep=new_line('a'))
!< test_passed(5) = (size(strings, dim=1)==size(line, dim=1))
!< do l=1, size(strings, dim=1)
!<    test_passed(l+5) = (strings(l)==line(l))
!< enddo
!< open(newunit=scratch, file='read_file_test.tmp', form='UNFORMATTED', access='STREAM')
!< close(scratch, status='DELETE')
!< call astring%read_file(file='read_file_test.tmp', iostat=iostat)
!< test_passed(9) = (iostat/=0)
!< print '(L1)', all(test_passed)
!<```
!=> T <<<
class(string),     intent(inout)           :: self       !< The string.
character(len=*), intent(in)               :: file       !< File name.
logical,          intent(in),    optional :: is_fast    !< Flag to enable (super) fast file reading.
character(len=*), intent(in),    optional :: form       !< Format of unit.
integer,          intent(out),   optional :: iostat     !< IO status code.
character(len=*), intent(inout), optional :: iomsg      !< IO status message.
logical                                   :: is_fast_   !< Flag to enable (super) fast file reading, local variable.
type(string)                              :: form_      !< Format of unit, local variable.
integer                                   :: iostat_    !< IO status code, local variable.
```

```
   character(len=:), allocatable             :: iomsg_     !< IO status message, local variable.
   integer                                   :: unit       !< Logical unit.
   logical                                   :: does_exist !< Check if file exist.
   integer(I4P)                              :: filesize   !< Size of the file for fast reading.

   iomsg_ = repeat(' ', 99) ; if (present(iomsg)) iomsg_ = iomsg
   inquire(file=file, iomsg=iomsg_, iostat=iostat_, exist=does_exist)
   if (does_exist) then
      is_fast_ = .false. ; if (present(is_fast)) is_fast_ = is_fast
      if (is_fast_) then
         open(newunit=unit, file=file, access='STREAM', form='UNFORMATTED', iomsg=iomsg_, iostat=iostat_)
         inquire(file=file, size=filesize)
         if (allocated(self%raw)) deallocate(self%raw)
         allocate(character(len=filesize):: self%raw)
         read(unit=unit, iostat=iostat_, iomsg=iomsg_) self%raw
         close(unit)
      else
         form_ = 'FORMATTED' ; if (present(form)) form_ = form ; form_ = form_%upper()
         select case(form_%chars())
         case('FORMATTED')
            open(newunit=unit, file=file, status='OLD', action='READ', iomsg=iomsg_, iostat=iostat_, err=10)
         case('UNFORMATTED')
            open(newunit=unit, file=file, status='OLD', action='READ', form='UNFORMATTED', access='STREAM', &
                 iomsg=iomsg_, iostat=iostat_, err=10)
         endselect
         call self%read_lines(unit=unit, form=form, iomsg=iomsg_, iostat=iostat_)
         10 close(unit)
      endif
   else
      iostat_ = 1
      iomsg_ = 'file not found'
   endif
   if (present(iostat)) iostat = iostat_
   if (present(iomsg)) iomsg = iomsg_
   endsubroutine read_file

   subroutine read_line(self, unit, form, iostat, iomsg)
   !< Read line (record) from a connected unit.
   !<
   !< The line is read as an ascii stream read until the eor is reached.
   !<
   !< @note For unformatted read only `access='stream'` is supported with new_line as line terminator.
   !<
   !<```fortran
```

```
!< type(string)        :: astring
!< type(string)        :: line(3)
!< integer             :: iostat
!< character(len=99)   :: iomsg
!< integer             :: scratch
!< integer             :: l
!< logical             :: test_passed(6)
!< line(1) = ' Hello World!    '
!< line(2) = 'How are you?   '
!< line(3) = '    All say: "Fine thanks"'
!< open(newunit=scratch, status='SCRATCH')
!< write(scratch, "(A)") line(1)%chars()
!< write(scratch, "(A)") line(2)%chars()
!< write(scratch, "(A)") line(3)%chars()
!< rewind(scratch)
!< l = 0
!< iostat = 0
!< do
!<    l = l + 1
!<    call astring%read_line(unit=scratch, iostat=iostat, iomsg=iomsg)
!<    if (iostat/=0.and..not.is_iostat_eor(iostat)) then
!<       exit
!<    else
!<       test_passed(l) = (astring==line(l))
!<    endif
!< enddo
!< close(scratch)
!< open(newunit=scratch, status='SCRATCH', form='UNFORMATTED', access='STREAM')
!< write(scratch) line(1)%chars()//new_line('a')
!< write(scratch) line(2)%chars()//new_line('a')
!< write(scratch) line(3)%chars()//new_line('a')
!< rewind(scratch)
!< l = 0
!< iostat = 0
!< do
!<    l = l + 1
!<    call astring%read_line(unit=scratch, iostat=iostat, iomsg=iomsg, form='UnfORMatteD')
!<    if (iostat/=0.and..not.is_iostat_eor(iostat)) then
!<       exit
!<    else
!<       test_passed(l+3) = (astring==line(l))
!<    endif
!< enddo
!< close(scratch)
```

```
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string),    intent(inout)           :: self    !< The string.
    integer,          intent(in)              :: unit    !< Logical unit.
    character(len=*), intent(in),    optional :: form    !< Format of unit.
    integer,          intent(out),   optional :: iostat  !< IO status code.
    character(len=*), intent(inout), optional :: iomsg   !< IO status message.
    type(string)                              :: form_   !< Format of unit, local variable.
    integer                                   :: iostat_ !< IO status code, local variable.
    character(len=:),          allocatable    :: iomsg_  !< IO status message, local variable.
    character(kind=CK, len=:), allocatable    :: line    !< Line storage.
    character(kind=CK, len=1)                 :: ch      !< Character storage.

    form_ = 'FORMATTED' ; if (present(form)) form_ = form ; form_ = form_%upper()
    iomsg_ = repeat('␣', 99) ; if (present(iomsg)) iomsg_ = iomsg
    line = ''
    select case(form_%chars())
    case('FORMATTED')
       do
          read(unit, "(A)", advance='no', iostat=iostat_, iomsg=iomsg_, err=10, end=10, eor=10) ch
          line = line//ch
       enddo
    case('UNFORMATTED')
       do
          read(unit, iostat=iostat_, iomsg=iomsg_, err=10, end=10) ch
          if (ch==new_line('a')) then
             iostat_ = iostat_eor
             exit
          endif
          line = line//ch
       enddo
    endselect
    10 if (line/='') self%raw = line
    if (present(iostat)) iostat = iostat_
    if (present(iomsg)) iomsg = iomsg_
    endsubroutine read_line

    subroutine read_lines(self, unit, form, iostat, iomsg)
    !< Read (all) lines (records) from a connected unit as a single ascii stream.
    !<
    !< @note All the lines are stored into the string self as a single ascii stream. Each line (record) is separated by a `ne
    !< character. The line is read as an ascii stream read until the eor is reached.
    !<
```

```
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.
```

```fortran
   character(len=*), intent(in),    optional :: form    !< Format of unit.
   integer,          intent(out),   optional :: iostat  !< IO status code.
   character(len=*), intent(inout), optional :: iomsg   !< IO status message.
   integer                                   :: iostat_ !< IO status code, local variable.
   character(len=:), allocatable             :: iomsg_  !< IO status message, local variable.
   type(string)                              :: lines   !< Lines storage.
   type(string)                              :: line    !< Line storage.

   iomsg_ = repeat(' ', 99) ; if (present(iomsg)) iomsg_ = iomsg
   rewind(unit)
   iostat_ = 0
   lines%raw = ''
   do
      line%raw = ''
      call line%read_line(unit=unit, form=form, iostat=iostat_, iomsg=iomsg_)
      if (iostat_/=0.and..not.is_iostat_eor(iostat_)) then
         exit
      elseif (line/='') then
         lines%raw = lines%raw//line%raw//new_line('a')
      endif
   enddo
   if (lines%raw/='') self%raw = lines%raw
   if (present(iostat)) iostat = iostat_
   if (present(iomsg)) iomsg = iomsg_
   endsubroutine read_lines

   elemental function replace(self, old, new, count) result(replaced)
   !< Return a string with all occurrences of substring old replaced by new.
   !<
   !<```fortran
   !< type(string) :: astring
   !< logical       :: test_passed(3)
   !< astring = 'When YOU are sad YOU should think to me :-)'
   !< test_passed(1) = (astring%replace(old='YOU', new='THEY')//''=='When THEY are sad THEY should think to me :-)')
   !< test_passed(2) = (astring%replace(old='YOU', new='THEY', count=1)//''=='When THEY are sad YOU should think to me :-)')
   !< astring = repeat(new_line('a')//'abcd', 20)
   !< astring = astring%replace(old=new_line('a'), new='|cr|')
   !< astring = astring%replace(old='|cr|', new=new_line('a')//'    ')
   !< test_passed(3) = (astring//''==repeat(new_line('a')//'    '//'abcd', 20))
   !< print '(L1)', all(test_passed)
   !<```
   !=> T <<<
   class(string),             intent(in)           :: self     !< The string.
   character(kind=CK, len=*), intent(in)           :: old      !< Old substring.
```

```
   character(kind=CK, len=*), intent(in)           :: new      !< New substring.
   integer,                   intent(in), optional :: count    !< Number of old occurences to be replaced.
   type(string)                                    :: replaced !< The string with old replaced by new.
   integer                                         :: r        !< Counter.

   if (allocated(self%raw)) then
      replaced = self
      r = 0
      do
         if (index(replaced%raw, old)>0) then
            replaced = replaced%replace_one_occurrence(old=old, new=new)
            r = r + 1
            if (present(count)) then
               if (r>=count) exit
            endif
         else
            exit
         endif
      enddo
   endif
   endfunction replace

   elemental function reverse(self) result(reversed)
   !< Return a reversed string.
   !<
   !<```fortran
   !< type(string) :: astring
   !< logical       :: test_passed(2)
   !< astring = 'abcdefghilmnopqrstuvz'
   !< test_passed(1) = (astring%reverse()//''=='zvutsrqponmlihgfedcba')
   !< astring = '0123456789'
   !< test_passed(2) = (astring%reverse()//''=='9876543210')
   !< print '(L1)', all(test_passed)
   !<```
   !=> T <<<
   class(string), intent(in) :: self     !< The string.
   type(string)              :: reversed !< The reversed string.
   integer                   :: length   !< Length of the string.
   integer                   :: c        !< Counter.

   if (allocated(self%raw)) then
      reversed = self
      length = len(self%raw)
      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.
```

```
type(string),               intent(in),  optional :: in_string     !< Search into this string.
character(kind=CK, len=*), intent(in),  optional :: in_character !< Search into this character string.
integer,                    intent(out), optional :: istart        !< Starting index of tag inside the string.
integer,                    intent(out), optional :: iend          !< Ending index of tag inside the string.
type(string)                                      :: tag           !< First tag found.
character(kind=CK, len=:), allocatable            :: raw           !< Raw string into which search the tag.
integer                                           :: istart_       !< Starting index of tag inside the string, local variab
integer                                           :: iend_         !< Ending index of tag inside the string, local variable
integer                                           :: nested_tags   !< Number of nested tags inside tag.
integer                                           :: t             !< Counter.

raw = ''
if (present(in_string)) then
   raw = in_string%raw
elseif (present(in_character)) then
   raw = in_character
else
   if (allocated(self%raw)) raw = self%raw
endif
istart_ = 0
iend_ = 0
if (raw/='') then
   istart_ = index(raw, tag_start)
   iend_ = index(raw, tag_end)
   if (istart_>0.and.iend_>0) then
      iend_ = iend_ + len(tag_end) - 1
      tag%raw = raw(istart_:iend_)
      nested_tags = tag%count(tag_start)
      if (nested_tags>1) then
         do t=2, nested_tags
            iend_ = iend_ + len(tag_end) - 1 + index(raw(iend_+1:), tag_end)
         enddo
         tag%raw = raw(istart_:iend_)
      endif
   endif
endif
if (present(istart)) istart = istart_
if (present(iend)) iend = iend_
endfunction search

pure function slice(self, istart, iend) result(raw)
!< Return the raw characters data sliced.
!<
!<```fortran
```

```
    !< type(string) :: astring
    !< astring = 'the Quick Brown fox Jumps over the Lazy Dog.'
    !< print "(A)", astring%slice(11,25)
    !<```
    !=> Brown fox Jumps <<<
    class(string), intent(in)                :: self   !< The string.
    integer,       intent(in)                :: istart !< Slice start index.
    integer,       intent(in)                :: iend   !< Slice end   index.
    character(kind=CK, len=:), allocatable :: raw    !< Raw characters data.

    if (allocated(self%raw)) then
       raw = self%raw(istart:iend)
    else
       raw = ''
    endif
    endfunction slice

    elemental function snakecase(self, sep)
    !< Return a string with all words lowercase separated by "_".
    !<
    !< @note Multiple subsequent separators are collapsed to one occurence.
    !<
    !<```fortran
    !< type(string) :: astring
    !< logical       :: test_passed(1)
    !< astring = 'the Quick Brown fox Jumps over the Lazy Dog.'
    !< test_passed(1) = astring%snakecase()//''=='the_quick_brown_fox_jumps_over_the_lazy_dog.'
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string),             intent(in)           :: self      !< The string.
    character(kind=CK, len=*), intent(in), optional :: sep       !< Separator.
    type(string)                                    :: snakecase !< Snake case string.
    type(string), allocatable                       :: tokens(:) !< String tokens.

    if (allocated(self%raw)) then
       call self%split(tokens=tokens, sep=sep)
       tokens = tokens%lower()
       snakecase = snakecase%join(array=tokens, sep='_')
    endif
    endfunction snakecase

    pure subroutine split(self, tokens, sep, max_tokens)
    !< Return a list of substring in the string, using sep as the delimiter string.
```

```
!<
!< @note Multiple subsequent separators are collapsed to one occurrence.
!<
!< @note If `max_tokens` is passed the returned number of tokens is either `max_tokens` or `max_tokens + 1`.
!<
!<```fortran
!< type(string)                 :: astring
!< type(string), allocatable    :: strings(:)
!< logical                      :: test_passed(11)
!< astring = '+ab-++cre-++cre-ab+'
!< call astring%split(tokens=strings, sep='+')
!< test_passed(1) = (strings(1)//''=='ab-'.and.strings(2)//''=='cre-'.and.strings(3)//''=='cre-ab')
!< astring = 'ab-++cre-++cre-ab+'
!< call astring%split(tokens=strings, sep='+')
!< test_passed(2) = (strings(1)//''=='ab-'.and.strings(2)//''=='cre-'.and.strings(3)//''=='cre-ab')
!< astring = 'ab-++cre-++cre-ab'
!< call astring%split(tokens=strings, sep='+')
!< test_passed(3) = (strings(1)//''=='ab-'.and.strings(2)//''=='cre-'.and.strings(3)//''=='cre-ab')
!< astring = 'Hello  '//new_line('a')//'World!'
!< call astring%split(tokens=strings, sep=new_line('a'))
!< test_passed(4) = (strings(1)//''=='Hello  '.and.strings(2)//''=='World!')
!< astring = 'Hello  World!'
!< call astring%split(tokens=strings)
!< test_passed(5) = (strings(1)//''=='Hello '.and.strings(2)//''=='World!')
!< astring = '+ab-'
!< call astring%split(tokens=strings, sep='+')
!< test_passed(6) = (strings(1)//''=='ab-')
!< astring = '+ab-'
!< call astring%split(tokens=strings, sep='-')
!< test_passed(7) = (strings(1)//''=='+ab')
!< astring = '+ab-+cd-'
!< call astring%split(tokens=strings, sep='+')
!< test_passed(8) = (strings(1)//''=='ab-'.and.strings(2)//''=='cd-')
!< astring = 'ab-+cd-+'
!< call astring%split(tokens=strings, sep='+')
!< test_passed(9) = (strings(1)//''=='ab-'.and.strings(2)//''=='cd-')
!< astring = '+ab-+cd-+'
!< call astring%split(tokens=strings, sep='+')
!< test_passed(10) = (strings(1)//''=='ab-'.and.strings(2)//''=='cd-')
!< astring = '1-2-3-4-5-6-7-8'
!< call astring%split(tokens=strings, sep='-', max_tokens=3)
!< test_passed(11) = (strings(1)//''=='1'.and.strings(2)//''=='2'.and.strings(3)//''=='3'.and.strings(4)//''=='4-5-6-7-8'
!< print '(L1)', all(test_passed)
!<```
```

```
    !=> T <<<
    class(string),              intent(in)           :: self            !< The string.
    type(string), allocatable , intent(out)          :: tokens(:)       !< Tokens substring.
    character(kind=CK, len=*), intent(in), optional :: sep             !< Separator.
    integer,                    intent(in), optional :: max_tokens     !< Fix the maximum number of returned tokens.
    character(kind=CK, len=:), allocatable           :: sep_            !< Separator, default value.
    integer                                          :: No              !< Number of occurrences of sep.
    integer                                          :: t               !< Character counter.
    type(string)                                     :: temporary       !< Temporary storage.
    type(string), allocatable                        :: temp_toks(:,:) !< Temporary tokens substring.

    if (allocated(self%raw)) then
      sep_ = SPACE ; if (present(sep)) sep_ = sep

      temporary = self%unique(sep_)
      No = temporary%count(sep_)

      if (No>0) then
        if (present(max_tokens)) then
          if (max_tokens < No.and.max_tokens > 0) No = max_tokens
        endif
        allocate(temp_toks(3, No))
        temp_toks(:, 1) = temporary%partition(sep_)
        if (No>1) then
          do t=2, No
            temp_toks(:, t) = temp_toks(3, t-1)%partition(sep_)
          enddo
        endif

        if (temp_toks(1, 1)%raw/='').and.temp_toks(3, No)%raw/='') then
          allocate(tokens(No+1))
          do t=1, No
            if (t==No) then
              tokens(t  ) = temp_toks(1, t)
              tokens(t+1) = temp_toks(3, t)
            else
              tokens(t) = temp_toks(1, t)
            endif
          enddo
        elseif (temp_toks(1, 1)%raw/='') then
          allocate(tokens(No))
          do t=1, No
            tokens(t) = temp_toks(1, t)
          enddo
```

```
      elseif (temp_toks(3, No)%raw/='') then
        allocate(tokens(No))
        do t=1, No-1
          tokens(t) = temp_toks(1, t+1)
        enddo
        tokens(No) = temp_toks(3, No)
      else
        allocate(tokens(No-1))
        do t=2, No
          tokens(t-1) = temp_toks(1, t)
        enddo
      endif

    else
      allocate(tokens(1))
      tokens(1) = self
    endif
  endif
  endsubroutine split

  pure subroutine split_chunked(self, tokens, chunks, sep)
  !< Return a list of substring in the string, using sep as the delimiter string, chunked (memory-efficient) algorithm.
  !<
  !< @note Multiple subsequent separators are collapsed to one occurrence.
  !<
  !< @note The split is performed in chunks of `#chunks` to avoid excessive memory consumption.
  !<
  !<```fortran
  !< type(string)                :: astring
  !< type(string), allocatable   :: strings(:)
  !< logical                     :: test_passed(1)
  !< astring = '-1-2-3-4-5-6-7-8-'
  !< call astring%split_chunked(tokens=strings, sep='-', chunks=3)
  !< test_passed(1) = (strings(1)//''=='1'.and.strings(2)//''=='2'.and.strings(3)//''=='3'.and.strings(4)//''=='4'.and. &
  !<                   strings(5)//''=='5'.and.strings(6)//''=='6'.and.strings(7)//''=='7'.and.strings(8)//''=='8')
  !< print '(L1)', all(test_passed)
  !<```
  !=> T <<<
  class(string),              intent(in)           :: self      !< The string.
  type(string), allocatable, intent(out)           :: tokens(:) !< Tokens substring.
  integer,                    intent(in)           :: chunks    !< Number of chunks.
  character(kind=CK, len=*), intent(in), optional :: sep       !< Separator.
  character(kind=CK, len=:), allocatable           :: sep_      !< Separator, default value.
  integer                                          :: Nt        !< Number of actual tokens.
```

```
    integer                                             :: t          !< Counter.
    logical                                             :: isok

    if (allocated(self%raw)) then
      sep_ = SPACE ; if (present(sep)) sep_ = sep

      Nt = self%count(sep_)
      if (self%start_with(prefix=sep_)) Nt = Nt - 1
      if (self%end_with(suffix=sep_)) Nt = Nt - 1
      t = 0
      call self%split(tokens=tokens, sep=sep_, max_tokens=chunks)
      do
        t = size(tokens, dim=1)
        if (t > Nt) exit
        call split_last_token(tokens=tokens, max_tokens=chunks,isok=isok)
        if(isok)then
        else
             exit
        endif
      enddo

      t = size(tokens, dim=1)
      if (tokens(t)%count(sep_) > 0) then
         call split_last_token(tokens=tokens,isok=isok)
      endif
    endif

    contains
       pure subroutine split_last_token(tokens, max_tokens,isok)
       !< Split last token.
       type(string), allocatable, intent(inout)        :: tokens(:)       !< Tokens substring.
       integer,                   intent(in), optional :: max_tokens      !< Max tokens returned.
       type(string), allocatable                       :: tokens_(:)      !< Temporary tokens.
       type(string), allocatable                       :: tokens_swap(:)  !< Swap tokens.
       integer                                         :: Nt_             !< Number of last created tokens.
       logical,intent(out)                             :: isok

       isok=.true.
       call tokens(t)%split(tokens=tokens_, sep=sep_, max_tokens=max_tokens)
       if (allocated(tokens_)) then
         Nt_ = size(tokens_, dim=1)
         if (Nt_ >= 1) then
           allocate(tokens_swap(1:t-1+Nt_))
           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
```

```
    !< type(string) :: astring
    !< logical       :: test_passed(1)
    !< astring = '  Hello World!   '
    !< test_passed(1) = astring%strip()//''=='Hello World!'
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string), intent(in)           :: self          !< The string.
    logical,       intent(in), optional :: remove_nulls  !< Remove null characters at the end.
    type(string)                        :: strip         !< The stripped string.
    integer                             :: c             !< Counter.

    if (allocated(self%raw)) then
       strip = self%adjustl()
       strip = strip%trim()
       if (present(remove_nulls)) then
          if (remove_nulls) then
             c = index(self%raw, char(0))
             if (c>0) strip%raw = strip%raw(1:c-1)
          endif
       endif
    endif
    endfunction strip

    elemental function swapcase(self)
    !< Return a copy of the string with uppercase characters converted to lowercase and vice versa.
    !<
    !<```fortran
    !< type(string) :: astring
    !< logical       :: test_passed(1)
    !< astring = '  Hello World!   '
    !< test_passed(1) = astring%swapcase()//''=='  hELLO wORLD!   '
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string), intent(in) :: self     !< The string.
    type(string)              :: swapcase !< Upper case string.
    integer                   :: n1       !< Characters counter.
    integer                   :: n2       !< Characters counter.

    if (allocated(self%raw)) then
       swapcase = self
       do n1=1, len(self%raw)
          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      !< The string.
2279      logical,       intent(in), optional :: is_file   !< True if tempname should be used for file (the default).
2280      character(*),  intent(in), optional :: prefix    !< Name prefix, otherwise self is used (if allocated).
2281      character(*),  intent(in), optional :: path      !< Path where file/directory should be used, default `./`.
2282      character(len=:), allocatable       :: tempname  !< Safe (unique) temporary name.
2283      logical                             :: is_file_  !< True if tempname should be used for file (the default).
2284      character(len=:), allocatable       :: prefix_   !< Name prefix, otherwise self is used (if allocated).
2285      character(len=:), allocatable       :: path_     !< 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'
```

```
    !< integer_ = astring%to_number(kind=1_I4P)
    !< test_passed(1) = integer_ ==127_I4P
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string), intent(in) :: self       !< The string.
    integer(I4P),  intent(in) :: kind       !< Mold parameter for kind detection.
    integer(I4P)              :: to_number  !< The number into the string.

    if (allocated(self%raw)) then
      if (self%is_integer()) read(self%raw, *) to_number
    endif
    endfunction to_integer_I4P

    elemental function to_integer_I8P(self, kind) result(to_number)
    !< Cast string to integer (I8P).
    !<
    !<```fortran
    !< use penf
    !< type(string) :: astring
    !< integer(I8P) :: integer_
    !< logical      :: test_passed(1)
    !< astring = '127'
    !< integer_ = astring%to_number(kind=1_I8P)
    !< test_passed(1) = integer_ ==127_I8P
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string), intent(in) :: self       !< The string.
    integer(I8P),  intent(in) :: kind       !< Mold parameter for kind detection.
    integer(I8P)              :: to_number  !< The number into the string.

    if (allocated(self%raw)) then
      if (self%is_integer()) read(self%raw, *) to_number
    endif
    endfunction to_integer_I8P

    elemental function to_real_R4P(self, kind) result(to_number)
    !< Cast string to real (R4P).
    !<
    !<```fortran
    !< use penf
    !< type(string) :: astring
    !< real(R4P)    :: real_
```

```
    !< logical       :: test_passed(1)
    !< astring = '3.4e9'
    !< real_ = astring%to_number(kind=1._R4P)
    !< test_passed(1) = real_==3.4e9_R4P
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string), intent(in) :: self      !< The string.
    real(R4P),     intent(in) :: kind      !< Mold parameter for kind detection.
    real(R4P)                 :: to_number !< The number into the string.

    if (allocated(self%raw)) then
      if (self%is_real()) read(self%raw, *) to_number
    endif
    endfunction to_real_R4P

    elemental function to_real_R8P(self, kind) result(to_number)
    !< Cast string to real (R8P).
    !<
    !<```fortran
    !< use penf
    !< type(string) :: astring
    !< real(R8P)    :: real_
    !< logical       :: test_passed(1)
    !< astring = '3.4e9'
    !< real_ = astring%to_number(kind=1._R8P)
    !< test_passed(1) = real_==3.4e9_R8P
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string), intent(in) :: self      !< The string.
    real(R8P),     intent(in) :: kind      !< Mold parameter for kind detection.
    real(R8P)                 :: to_number !< The number into the string.

    if (allocated(self%raw)) then
      if (self%is_real()) read(self%raw, *) to_number
    endif
    endfunction to_real_R8P

    elemental function to_real_R16P(self, kind) result(to_number)
    !< Cast string to real (R16P).
    !<
    !<```fortran
    !< use penf
```

```
    !< type(string) :: astring
    !< real(R16P)    :: real_
    !< logical       :: test_passed(1)
    !< astring = '3.4e9'
    !< real_ = astring%to_number(kind=1._R16P)
    !< test_passed(1) = real_==3.4e9_R16P
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string), intent(in) :: self       !< The string.
    real(R16P),    intent(in) :: kind       !< Mold parameter for kind detection.
    real(R16P)                :: to_number  !< The number into the string.

    if (allocated(self%raw)) then
      if (self%is_real()) read(self%raw, *) to_number
    endif
    endfunction to_real_R16P

    elemental function unescape(self, to_unescape, unesc) result(unescaped)
    !< Unescape double backslashes (or custom escaped character).
    !<
    !<```fortran
    !< type(string) :: astring
    !< logical       :: test_passed(2)
    !< astring = '^\\s \\d+\\s*'
    !< test_passed(1) = (astring%unescape(to_unescape='\')//''=='^\s \d+\s*')
    !< test_passed(2) = (astring%unescape(to_unescape='s')//''=='^\s \\d+\s*')
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string),              intent(in)           :: self        !< The string.
    character(kind=CK, len=1),  intent(in)           :: to_unescape !< Character to be unescaped.
    character(kind=CK, len=*),  intent(in), optional :: unesc       !< Character used to unescape.
    type(string)                                     :: unescaped   !< Escaped string.
    character(kind=CK, len=:),  allocatable          :: unesc_      !< Character to unescape, local variable.
    integer                                          :: c           !< Character counter.

    if (allocated(self%raw)) then
       unesc_ = '' ; if (present(unesc)) unesc_ = unesc
       unescaped%raw = ''
       c = 1
       do
          if (c>len(self%raw)) exit
          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 occurences 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"'
```

```
!< anotherstring = anotherstring%join(array=line, sep=new_line('a'))
!< call anotherstring%write_file(file='write_file_test.tmp', iostat=iostat, iomsg=iomsg)
!< call astring%read_file(file='write_file_test.tmp', iostat=iostat, iomsg=iomsg)
!< call astring%split(tokens=strings, sep=new_line('a'))
!< test_passed(1) = (size(strings, dim=1)==size(line, dim=1))
!< do l=1, size(strings, dim=1)
!<    test_passed(l+1) = (strings(l)==line(l))
!< enddo
!< call anotherstring%write_file(file='write_file_test.tmp', form='unformatted', iostat=iostat, iomsg=iomsg)
!< call astring%read_file(file='write_file_test.tmp', form='unformatted', iostat=iostat, iomsg=iomsg)
!< call astring%split(tokens=strings, sep=new_line('a'))
!< test_passed(5) = (size(strings, dim=1)==size(line, dim=1))
!< do l=1, size(strings, dim=1)
!<    test_passed(l+5) = (strings(l)==line(l))
!< enddo
!< open(newunit=scratch, file='write_file_test.tmp')
!< close(unit=scratch, status='delete')
!< print '(L1)', all(test_passed)
!<```
!=> T <<<
class(string),     intent(in)                :: self     !< The string.
character(len=*), intent(in)                 :: file     !< File name.
character(len=*), intent(in),     optional   :: form     !< Format of unit.
integer,          intent(out),    optional   :: iostat   !< IO status code.
character(len=*), intent(inout), optional    :: iomsg    !< IO status message.
type(string)                                 :: form_    !< Format of unit, local variable.
integer                                      :: iostat_  !< IO status code, local variable.
character(len=:), allocatable                :: iomsg_   !< IO status message, local variable.
integer                                      :: unit     !< Logical unit.

iomsg_ = repeat('␣', 99) ; if (present(iomsg)) iomsg_ = iomsg
form_ = 'FORMATTED' ; if (present(form)) form_ = form ; form_ = form_%upper()
select case(form_%chars())
case('FORMATTED')
   open(newunit=unit, file=file, action='WRITE', iomsg=iomsg_, iostat=iostat_, err=10)
case('UNFORMATTED')
   open(newunit=unit, file=file, action='WRITE', form='UNFORMATTED', access='STREAM', iomsg=iomsg_, iostat=iostat_, err=1
endselect
call self%write_lines(unit=unit, form=form, iomsg=iomsg_, iostat=iostat_)
10 close(unit)
if (present(iostat)) iostat = iostat_
if (present(iomsg)) iomsg = iomsg_
endsubroutine write_file

```

```
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!').equ..true.
2705    !< test_passed(2) = astring%end_with(suffix='lD!').equ..false.
2706    !< test_passed(3) = astring%end_with(suffix='orLD!', start=5).equ..true.
2707    !< test_passed(4) = astring%end_with(suffix='orLD!', start=8, end=12).equ..true.
2708    !< test_passed(5) = astring%end_with(suffix='!').equ..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().equ..false.
2741     !< astring = 'hello'
2742     !< test_passed(2) = astring%is_allocated().equ..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().equ..false.
2760     !< astring = '12121123'
2761     !< test_passed(2) = astring%is_digit().equ..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().eqv..true.
2806    !< astring = '    -1212112'
2807    !< test_passed(2) = astring%is_integer(allow_spaces=.false.).eqv..false.
2808    !< astring = '-1212112    '
2809    !< test_passed(3) = astring%is_integer(allow_spaces=.false.).eqv..false.
2810    !< astring = '+2e20'
2811    !< test_passed(4) = astring%is_integer().eqv..true.
2812    !< astring = ' -2E13 '
2813    !< test_passed(5) = astring%is_integer().eqv..true.
```

```
    !< astring = ' -2 E13 '
    !< test_passed(6) = astring%is_integer().equ..false.
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string), intent(in)           :: self          !< The string.
    logical,       intent(in), optional :: allow_spaces  !< Allow leading-trailing spaces.
    logical                             :: is_integer    !< Result of the test.
    logical                             :: allow_spaces_ !< Allow leading-trailing spaces, local variable.
    integer                             :: stage         !< Stages counter.
    integer                             :: c             !< Character counter.

    if (allocated(self%raw)) then
       allow_spaces_ = .true. ; if (present(allow_spaces)) allow_spaces_ = allow_spaces
       stage = 0
       is_integer = .true.
       do c=1, len(self%raw)
          select case(self%raw(c:c))
          case(SPACE, TAB)
             select case(stage)
             case(0, 6)
                is_integer = allow_spaces_
             case(2, 5)
                is_integer = allow_spaces_
                stage = 6
             case default
                is_integer = .false.
             endselect
          case('-')
             select case(stage)
             case(0)
                stage = 1
             case default
                is_integer = .false.
             end select
          case('+')
             select case(stage)
             case(0)
                stage = 1
             case(3)
                stage = 4
             case default
                is_integer = .false.
             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().equ..false.
2898      !< astring = ' HELLO WORLD'
2899      !< test_passed(2) = astring%is_lower().equ..false.
2900      !< astring = ' hello world'
2901      !< test_passed(3) = astring%is_lower().equ..true.
```

```
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string), intent(in) :: self      !< The string.
    logical                   :: is_lower  !< Result of the test.
    integer                   :: c         !< Character counter.

    is_lower = .false.
    if (allocated(self%raw)) then
       is_lower = .true.
       do c=1, len(self%raw)
          if (index(UPPER_ALPHABET, self%raw(c:c))>0) then
             is_lower = .false.
             exit
          endif
       enddo
    endif
    endfunction is_lower

    elemental function is_number(self, allow_spaces)
    !< Return true if the string contains a number (real or integer).
    !<
    !<```fortran
    !< type(string) :: astring
    !< logical       :: test_passed(7)
    !< astring = '    -1212112 '
    !< test_passed(1) = astring%is_number().eqv..true.
    !< astring = '    -121.2112 '
    !< test_passed(2) = astring%is_number().eqv..true.
    !< astring = '    -1212112'
    !< test_passed(3) = astring%is_number(allow_spaces=.false.).eqv..false.
    !< astring = '-12121.12    '
    !< test_passed(4) = astring%is_number(allow_spaces=.false.).eqv..false.
    !< astring = '+2e20'
    !< test_passed(5) = astring%is_number().eqv..true.
    !< astring = ' -2.4E13 '
    !< test_passed(6) = astring%is_number().eqv..true.
    !< astring = ' -2 E13 '
    !< test_passed(7) = astring%is_number().eqv..false.
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string), intent(in)           :: self         !< The string.
    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().eqv..true.
2974    !< astring = '    -1212112.d0'
2975    !< test_passed(2) = astring%is_real(allow_spaces=.false.).eqv..false.
2976    !< astring = '-1212112.d0    '
2977    !< test_passed(3) = astring%is_real(allow_spaces=.false.).eqv..false.
2978    !< astring = '+2.e20'
2979    !< test_passed(4) = astring%is_real().eqv..true.
2980    !< astring = ' -2.01E13 '
2981    !< test_passed(5) = astring%is_real().eqv..true.
2982    !< astring = ' -2.01 E13 '
2983    !< test_passed(6) = astring%is_real().eqv..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().equ..false.
3074      !< astring = ' HELLO WORLD'
3075      !< test_passed(2) = astring%is_upper().equ..true.
3076      !< astring = ' hello world'
3077      !< test_passed(3) = astring%is_upper().equ..false.
```

```
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string), intent(in) :: self      !< The string.
    logical                   :: is_upper  !< Result of the test.
    integer                   :: c         !< Character counter.

    is_upper = .false.
    if (allocated(self%raw)) then
       is_upper = .true.
       do c=1, len(self%raw)
          if (index(LOWER_ALPHABET, self%raw(c:c))>0) then
             is_upper = .false.
             exit
          endif
       enddo
    endif
    endfunction is_upper

    elemental function start_with(self, prefix, start, end)
    !< Return true if a string starts with a specified prefix.
    !<
    !<```fortran
    !< type(string) :: astring
    !< logical       :: test_passed(4)
    !< astring = 'Hello WorLD!'
    !< test_passed(1) = astring%start_with(prefix='Hello ').equ..true.
    !< test_passed(2) = astring%start_with(prefix='hell ').equ..false.
    !< test_passed(3) = astring%start_with(prefix='llo Wor', start=3).equ..true.
    !< test_passed(4) = astring%start_with(prefix='lo W', start=4, end=7).equ..true.
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string),              intent(in)           :: self       !< The string.
    character(kind=CK, len=*), intent(in)           :: prefix     !< Searched prefix.
    integer,                    intent(in), optional :: start      !< Start position into the string.
    integer,                    intent(in), optional :: end        !< End position into the string.
    logical                                          :: start_with !< Result of the test.
    integer                                          :: start_     !< Start position into the string, local variable.
    integer                                          :: end_       !< End position into the string, local variable.

    start_with = .false.
    if (allocated(self%raw)) then
       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
```

```
    !< logical       :: test_passed(1)
    !< astring = 127_I4P
    !< test_passed(1) = astring%to_number(kind=1_I4P)==127_I4P
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string), intent(inout) :: lhs !< Left hand side.
    integer(I4P),  intent(in)    :: rhs !< Right hand side.

    lhs%raw = trim(str(rhs))
    endsubroutine string_assign_integer_I4P

    elemental subroutine string_assign_integer_I8P(lhs, rhs)
    !< Assignment operator from integer input.
    !<
    !<```fortran
    !< use penf
    !< type(string) :: astring
    !< logical       :: test_passed(1)
    !< astring = 127_I8P
    !< test_passed(1) = astring%to_number(kind=1_I8P)==127_I8P
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string), intent(inout) :: lhs !< Left hand side.
    integer(I8P),  intent(in)    :: rhs !< Right hand side.

    lhs%raw = trim(str(rhs))
    endsubroutine string_assign_integer_I8P

    elemental subroutine string_assign_real_R4P(lhs, rhs)
    !< Assignment operator from real input.
    !<
    !<```fortran
    !< use penf
    !< type(string) :: astring
    !< logical       :: test_passed(1)
    !< astring = 3.021e6_R4P
    !< test_passed(1) = astring%to_number(kind=1._R4P)==3.021e6_R4P
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string), intent(inout) :: lhs !< Left hand side.
    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    !<
```

```
!<```fortran
!< type(string) :: astring
!< type(string) :: anotherstring
!< logical       :: test_passed(1)
!< astring = 'Hello '
!< anotherstring = 'Bye bye'
!< test_passed(1) = astring//anotherstring=='Hello Bye bye'
!< print '(L1)', all(test_passed)
!<```
!=> T <<<
class(string), intent(in)                :: lhs     !< Left hand side.
type(string),  intent(in)                :: rhs     !< Right hand side.
character(kind=CK, len=:), allocatable :: concat !< Concatenated string.

concat = ''
if (allocated(lhs%raw)) concat = lhs%raw
if (allocated(rhs%raw)) concat = concat//rhs%raw
endfunction string_concat_string

pure function string_concat_character(lhs, rhs) result(concat)
!< Concatenation with character.
!<
!<```fortran
!< type(string)                    :: astring
!< character(len=:), allocatable :: acharacter
!< logical                         :: test_passed(1)
!< astring = 'Hello '
!< acharacter = 'World!'
!< test_passed(1) = astring//acharacter=='Hello World!'
!< print '(L1)', all(test_passed)
!<```
!=> T <<<
class(string),              intent(in)  :: lhs     !< Left hand side.
character(kind=CK, len=*), intent(in)  :: rhs     !< Right hand side.
character(kind=CK, len=:), allocatable :: concat !< Concatenated string.

if (allocated(lhs%raw)) then
   concat = lhs%raw//rhs
else
   concat = rhs
endif
endfunction string_concat_character

pure function character_concat_string(lhs, rhs) result(concat)
```

```
    !< Concatenation with character (inverted).
    !<
    !<```fortran
    !< type(string)                      :: astring
    !< character(len=:), allocatable :: acharacter
    !< logical                           :: test_passed(1)
    !< astring = 'Hello '
    !< acharacter = 'World!'
    !< test_passed(1) = acharacter//astring=='World!Hello '
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    character(kind=CK, len=*), intent(in)  :: lhs     !< Left hand side.
    class(string),             intent(in)  :: rhs     !< Right hand side.
    character(kind=CK, len=:), allocatable :: concat !< Concatenated string.

    if (allocated(rhs%raw)) then
       concat = lhs//rhs%raw
    else
       concat = lhs
    endif
    endfunction character_concat_string

    elemental function string_concat_string_string(lhs, rhs) result(concat)
    !< Concatenation with string.
    !<
    !<```fortran
    !< type(string) :: astring
    !< type(string) :: anotherstring
    !< type(string) :: yetanotherstring
    !< logical       :: test_passed(1)
    !< astring = 'Hello '
    !< anotherstring = 'Bye bye'
    !< yetanotherstring = astring.cat.anotherstring
    !< test_passed(1) = yetanotherstring%chars()=='Hello Bye bye'
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string), intent(in)                :: lhs       !< Left hand side.
    type(string),  intent(in)                :: rhs       !< Right hand side.
    type(string)                             :: concat    !< Concatenated string.
    character(kind=CK, len=:), allocatable :: temporary !< Temporary concatenated string.

    temporary = ''
```

```
   if (allocated(lhs%raw)) temporary = lhs%raw
   if (allocated(rhs%raw)) temporary = temporary//rhs%raw
   if (temporary/='') concat%raw = temporary
   endfunction string_concat_string_string

   elemental function string_concat_character_string(lhs, rhs) result(concat)
   !< Concatenation with character.
   !<
   !<```fortran
   !< type(string)                    :: astring
   !< type(string)                    :: yetanotherstring
   !< character(len=:), allocatable :: acharacter
   !< logical                         :: test_passed(1)
   !< astring = 'Hello '
   !< acharacter = 'World!'
   !< yetanotherstring = astring.cat.acharacter
   !< test_passed(1) = yetanotherstring%chars()=='Hello World!'
   !< print '(L1)', all(test_passed)
   !<```
   !=> T <<<
   class(string),              intent(in)  :: lhs     !< Left hand side.
   character(kind=CK, len=*), intent(in)  :: rhs     !< Right hand side.
   type(string)                            :: concat  !< Concatenated string.

   if (allocated(lhs%raw)) then
      concat%raw = lhs%raw//rhs
   else
      concat%raw = rhs
   endif
   endfunction string_concat_character_string

   elemental function character_concat_string_string(lhs, rhs) result(concat)
   !< Concatenation with character (inverted).
   !<
   !<```fortran
   !< type(string)                    :: astring
   !< type(string)                    :: yetanotherstring
   !< character(len=:), allocatable :: acharacter
   !< logical                         :: test_passed(1)
   !< astring = 'Hello '
   !< acharacter = 'World!'
   !< yetanotherstring = acharacter.cat.astring
   !< test_passed(1) = yetanotherstring%chars()=='World!Hello '
   !< print '(L1)', all(test_passed)
```

```
    !<```
    !=> T <<<
    character(kind=CK, len=*), intent(in)  :: lhs     !< Left hand side.
    class(string),             intent(in)  :: rhs     !< Right hand side.
    type(string)                           :: concat  !< Concatenated string.

    if (allocated(rhs%raw)) then
      concat%raw = lhs//rhs%raw
    else
      concat%raw = lhs
    endif
    endfunction character_concat_string_string

    ! logical operators
    elemental function string_eq_string(lhs, rhs) result(is_it)
    !< Equal to string logical operator.
    !<
    !<```fortran
    !< type(string) :: astring
    !< type(string) :: anotherstring
    !< logical       :: test_passed(2)
    !< astring = '  one '
    !< anotherstring = 'two'
    !< test_passed(1) = ((astring==anotherstring).eqv..false.)
    !< astring = 'the same '
    !< anotherstring = 'the same '
    !< test_passed(2) = ((astring==anotherstring).eqv..true.)
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string), intent(in) :: lhs    !< Left hand side.
    type(string),  intent(in) :: rhs    !< Right hand side.
    logical                   :: is_it  !< Opreator test result.

    is_it = lhs%raw == rhs%raw
    endfunction string_eq_string

    elemental function string_eq_character(lhs, rhs) result(is_it)
    !< Equal to character logical operator.
    !<
    !<```fortran
    !< type(string)                    :: astring
    !< character(len=:), allocatable :: acharacter
    !< logical                         :: test_passed(2)
```

```
    !< astring = '  one '
    !< acharacter = 'three'
    !< test_passed(1) = ((astring==acharacter).eqv..false.)
    !< astring = 'the same '
    !< acharacter = 'the same '
    !< test_passed(2) = ((astring==acharacter).eqv..true.)
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string),              intent(in) :: lhs    !< Left hand side.
    character(kind=CK, len=*), intent(in) :: rhs    !< Right hand side.
    logical                                :: is_it  !< Opreator test result.

    is_it = lhs%raw == rhs
    endfunction string_eq_character

    elemental function character_eq_string(lhs, rhs) result(is_it)
    !< Equal to character (inverted) logical operator.
    !<
    !<```fortran
    !< type(string)                    :: astring
    !< character(len=:), allocatable :: acharacter
    !< logical                         :: test_passed(2)
    !< astring = '  one '
    !< acharacter = 'three'
    !< test_passed(1) = ((acharacter==astring).eqv..false.)
    !< astring = 'the same '
    !< acharacter = 'the same '
    !< test_passed(2) = ((acharacter==astring).eqv..true.)
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    character(kind=CK, len=*), intent(in) :: lhs    !< Left hand side.
    class(string),              intent(in) :: rhs    !< Right hand side.
    logical                                :: is_it  !< Opreator test result.

    is_it = rhs%raw == lhs
    endfunction character_eq_string

    elemental function string_ne_string(lhs, rhs) result(is_it)
    !< Not equal to string logical operator.
    !<
    !<```fortran
    !< type(string) :: astring
```

```
    !< type(string) :: anotherstring
    !< logical       :: test_passed(2)
    !< astring = '  one '
    !< anotherstring = 'two'
    !< test_passed(1) = ((astring/=anotherstring).eqv..true.)
    !< astring = 'the same '
    !< anotherstring = 'the same '
    !< test_passed(2) = ((astring/=anotherstring).eqv..false.)
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string), intent(in) :: lhs    !< Left hand side.
    type(string),  intent(in) :: rhs    !< Right hand side.
    logical                   :: is_it  !< Opreator test result.

    is_it = lhs%raw /= rhs%raw
    endfunction string_ne_string

    elemental function string_ne_character(lhs, rhs) result(is_it)
    !< Not equal to character logical operator.
    !<
    !<```fortran
    !< type(string)                    :: astring
    !< character(len=:), allocatable :: acharacter
    !< logical                         :: test_passed(2)
    !< astring = '  one '
    !< acharacter = 'three'
    !< test_passed(1) = ((astring/=acharacter).eqv..true.)
    !< astring = 'the same '
    !< acharacter = 'the same '
    !< test_passed(2) = ((astring/=acharacter).eqv..false.)
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string),              intent(in) :: lhs    !< Left hand side.
    character(kind=CK, len=*), intent(in) :: rhs    !< Right hand side.
    logical                                :: is_it  !< Opreator test result.

    is_it = lhs%raw /= rhs
    endfunction string_ne_character

    elemental function character_ne_string(lhs, rhs) result(is_it)
    !< Not equal to character (inverted) logical operator.
    !<
```

```
    !<```fortran
    !< type(string)                    :: astring
    !< character(len=:), allocatable :: acharacter
    !< logical                         :: test_passed(2)
    !< astring = '  one '
    !< acharacter = 'three'
    !< test_passed(1) = ((acharacter/=astring).eqv..true.)
    !< astring = 'the same '
    !< acharacter = 'the same '
    !< test_passed(2) = ((acharacter/=astring).eqv..false.)
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    character(kind=CK, len=*), intent(in) :: lhs    !< Left hand side.
    class(string),             intent(in) :: rhs    !< Right hand side.
    logical                               :: is_it  !< Opreator test result.

    is_it = rhs%raw /= lhs
    endfunction character_ne_string

    elemental function string_lt_string(lhs, rhs) result(is_it)
    !< Lower than to string logical operator.
    !<
    !<```fortran
    !< type(string) :: astring
    !< type(string) :: anotherstring
    !< logical       :: test_passed(2)
    !< astring = 'one'
    !< anotherstring = 'ONE'
    !< test_passed(1) = ((astring<anotherstring).eqv..false.)
    !< astring = 'ONE'
    !< anotherstring = 'one'
    !< test_passed(2) = ((astring<anotherstring).eqv..true.)
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string), intent(in) :: lhs    !< Left hand side.
    type(string),  intent(in) :: rhs    !< Right hand side.
    logical                   :: is_it  !< Opreator test result.

    is_it = lhs%raw < rhs%raw
    endfunction string_lt_string

    elemental function string_lt_character(lhs, rhs) result(is_it)
```

```
    !< Lower than to character logical operator.
    !<
    !<```fortran
    !< type(string)                      :: astring
    !< character(len=:), allocatable     :: acharacter
    !< logical                           :: test_passed(2)
    !< astring = 'one'
    !< acharacter = 'ONE'
    !< test_passed(1) = ((astring<acharacter).eqv..false.)
    !< astring = 'ONE'
    !< acharacter = 'one'
    !< test_passed(2) = ((astring<acharacter).eqv..true.)
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string),              intent(in) :: lhs    !< Left hand side.
    character(kind=CK, len=*), intent(in) :: rhs    !< Right hand side.
    logical                               :: is_it  !< Opreator test result.

    is_it = lhs%raw < rhs
    endfunction string_lt_character

    elemental function character_lt_string(lhs, rhs) result(is_it)
    !< Lower than to character (inverted) logical operator.
    !<
    !<```fortran
    !< type(string)                      :: astring
    !< character(len=:), allocatable     :: acharacter
    !< logical                           :: test_passed(2)
    !< astring = 'one'
    !< acharacter = 'ONE'
    !< test_passed(1) = ((acharacter<astring).eqv..true.)
    !< astring = 'ONE'
    !< acharacter = 'one'
    !< test_passed(2) = ((acharacter<astring).eqv..false.)
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    character(kind=CK, len=*), intent(in) :: lhs    !< Left hand side.
    class(string),              intent(in) :: rhs    !< Right hand side.
    logical                               :: is_it  !< Opreator test result.

    is_it = lhs < rhs%raw
    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).eqv..false.)
3661    !< astring = 'ONE'
3662    !< anotherstring = 'one'
3663    !< test_passed(2) = ((astring<=anotherstring).eqv..true.)
3664    !< astring = 'ONE'
3665    !< anotherstring = 'ONE'
3666    !< test_passed(3) = ((astring<=anotherstring).eqv..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  !< Opreator 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).eqv..false.)
3687    !< astring = 'ONE'
3688    !< acharacter = 'one'
3689    !< test_passed(2) = ((astring<=acharacter).eqv..true.)
3690    !< astring = 'ONE'
3691    !< acharacter = 'ONE'
3692    !< test_passed(3) = ((astring<=acharacter).eqv..true.)
3693    !< print '(L1)', all(test_passed)
```

```
    !<```
    !=> T <<<
    class(string),              intent(in) :: lhs    !< Left hand side.
    character(kind=CK, len=*), intent(in) :: rhs    !< Right hand side.
    logical                                :: is_it !< Opreator test result.

    is_it = lhs%raw <= rhs
    endfunction string_le_character

    elemental function character_le_string(lhs, rhs) result(is_it)
    !< Lower equal than to character (inverted) logical operator.
    !<
    !<```fortran
    !< type(string)                    :: astring
    !< character(len=:), allocatable :: acharacter
    !< logical                         :: test_passed(3)
    !< astring = 'one'
    !< acharacter = 'ONE'
    !< test_passed(1) = ((acharacter <= astring).eqv..true.)
    !< astring = 'ONE'
    !< acharacter = 'one'
    !< test_passed(2) = ((acharacter <= astring).eqv..false.)
    !< astring = 'ONE'
    !< acharacter = 'ONE'
    !< test_passed(3) = ((acharacter <= astring).eqv..true.)
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    character(kind=CK, len=*), intent(in) :: lhs    !< Left hand side.
    class(string),              intent(in) :: rhs    !< Right hand side.
    logical                                :: is_it !< Opreator test result.

    is_it = lhs <= rhs%raw
    endfunction character_le_string

    elemental function string_ge_string(lhs, rhs) result(is_it)
    !< Greater equal than to string logical operator.
    !<
    !<```fortran
    !< type(string) :: astring
    !< type(string) :: anotherstring
    !< logical       :: test_passed(3)
    !< astring = 'one'
    !< anotherstring = 'ONE'
```

```
!< test_passed(1) = ((astring>=anotherstring).eqv..true.)
!< astring = 'ONE'
!< anotherstring = 'one'
!< test_passed(2) = ((astring>=anotherstring).eqv..false.)
!< astring = 'ONE'
!< anotherstring = 'ONE'
!< test_passed(3) = ((astring>=anotherstring).eqv..true.)
!< print '(L1)', all(test_passed)
!<```
!=> T <<<
class(string), intent(in) :: lhs    !< Left hand side.
type(string),  intent(in) :: rhs    !< Right hand side.
logical                   :: is_it  !< Opreator test result.

is_it = lhs%raw >= rhs%raw
endfunction string_ge_string

elemental function string_ge_character(lhs, rhs) result(is_it)
!< Greater equal than to character logical operator.
!<
!<```fortran
!< type(string)                    :: astring
!< character(len=:), allocatable   :: acharacter
!< logical                         :: test_passed(3)
!< astring = 'one'
!< acharacter = 'ONE'
!< test_passed(1) = ((astring>=acharacter).eqv..true.)
!< astring = 'ONE'
!< acharacter = 'one'
!< test_passed(2) = ((astring>=acharacter).eqv..false.)
!< astring = 'ONE'
!< acharacter = 'ONE'
!< test_passed(3) = ((astring>=acharacter).eqv..true.)
!< print '(L1)', all(test_passed)
!<```
!=> T <<<
class(string),              intent(in) :: lhs    !< Left hand side.
character(kind=CK, len=*), intent(in) :: rhs    !< Right hand side.
logical                                :: is_it  !< Opreator test result.

is_it = lhs%raw >= rhs
endfunction string_ge_character

elemental function character_ge_string(lhs, rhs) result(is_it)
```

```
    !< Greater equal than to character (inverted) logical operator.
    !<
    !<```fortran
    !< type(string)                      :: astring
    !< character(len=:), allocatable :: acharacter
    !< logical                           :: test_passed(3)
    !< astring = 'one'
    !< acharacter = 'ONE'
    !< test_passed(1) = ((acharacter>=astring).eqv..false.)
    !< astring = 'ONE'
    !< acharacter = 'one'
    !< test_passed(2) = ((acharacter>=astring).eqv..true.)
    !< astring = 'ONE'
    !< acharacter = 'ONE'
    !< test_passed(3) = ((acharacter>=astring).eqv..true.)
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    character(kind=CK, len=*), intent(in) :: lhs    !< Left hand side.
    class(string),             intent(in) :: rhs    !< Right hand side.
    logical                               :: is_it  !< Opreator test result.

    is_it = lhs >= rhs%raw
    endfunction character_ge_string

    elemental function string_gt_string(lhs, rhs) result(is_it)
    !< Greater than to string logical operator.
    !<
    !<```fortran
    !< type(string) :: astring
    !< type(string) :: anotherstring
    !< logical       :: test_passed(2)
    !< astring = 'one'
    !< anotherstring = 'ONE'
    !< test_passed(1) = ((astring>anotherstring).eqv..true.)
    !< astring = 'ONE'
    !< anotherstring = 'one'
    !< test_passed(2) = ((astring>anotherstring).eqv..false.)
    !< print '(L1)', all(test_passed)
    !<```
    !=> T <<<
    class(string), intent(in) :: lhs    !< Left hand side.
    type(string),  intent(in) :: rhs    !< Right hand side.
    logical                   :: is_it  !< Opreator 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).eqv..true.)
3840    !< astring = 'ONE'
3841    !< acharacter = 'one'
3842    !< test_passed(2) = ((astring>acharacter).eqv..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 !< Opreator 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).eqv..false.)
3863    !< astring = 'ONE'
3864    !< acharacter = 'one'
3865    !< test_passed(2) = ((acharacter>astring).eqv..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  !< Opreator 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=ioms
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-zer
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 spcify 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
```