Implement algorithms in Modern Fortran.

Yuma Osada

2023-07-15.

目次

1 emacs-lisp

1.1 org babel settings

Modify org-babel function to avoid adding `program...end program` around `module...end module`.

```emacs-lisp
(defun org-babel-fortran-ensure-main-wrap (body params)
  "Wrap body in a \"program ... end program\" block if none exists."
  (if (or (string-match "^[ \t]*program\\>" (capitalize body))
          (string-match "^[ \t]*module\\>" (capitalize body)))
      (let ((vars (org-babel--get-vars params)))
        (when vars (error "Cannot use :vars if `program' statement is present"))
        body)
    (format "program main\n%s\nend program main\n" body)))
;; reference: https://stackoverflow.com/questions/40033843/show-the-name-of-a-code-block-in-org-mode-when-export.
(customize-set-variable 'org-babel-exp-code-template
                        "#+CAPTION: set-fortran-wrap(%lang)
                        ↪  label:set-fortran-wrap\n#+ATTR_LaTeX: :placement [H] :float
                        ↪  t\n#+BEGIN_SRC %lang\n%body\n#+END_SRC")
```

Listing 1: set-fortran-wrap(emacs-lisp)

```emacs-lisp
(customize-set-variable 'org-confirm-babel-evaluate nil)
```

Listing 2: disabled_org-confirm-babel-evaluate(emacs-lisp)

2 org-babel macros

2.1 error handling

```
cat <<EOF
  if (present(ierr)) then
    ierr = ${ierr}
    return
  end if
EOF
```

Listing 3: error-handling-return-ierr(bash)

```fortran
write(error_unit, '(a, i0, a)', advance = "no")&
     "Error in "//&
     __FILE__&
     //":", __LINE__, ":"
```

Listing 4: error-handling-filename(fortran)

```bash
cat <<EOF
  write(error_unit, '(a)')&
    "${string}"
  error stop ${err_num}
EOF
```

Listing 5: error-handling-error_message-exit(bash)

```bash
cat <<EOF
  error stop ${err_num}
EOF
```

Listing 6: error-handling-exit(bash)

2.2 assertion

```
if (${cond}) then
   write(error_unit, '(a, i0, a)', advance = "no")&
        "Error in "//&
        __FILE__&
        //":", __LINE__, ":"
   write(error_unit, '(a)') " Assertion '${cond_origin}' must be ${true_false}."
   if (len_trim("${message}") /= 0) then
      write(error_unit, '(a)') "Extra message: '${message}'"
   end if
   error stop ${code}
end if
```

Listing 7: assert-fortran(fortran)

```
if (${cond}) then
   write(error_unit, '(a, i0, a)', advance = "no")&
        "Error in "//&
        __FILE__&
        //":", __LINE__, ":"
   write(error_unit, '(a)') " Assertion '${cond_origin}' must be ${true_false}."
   write(error_unit, '(a)', advance = "no") "${eq1}: "
   write(error_unit, *) ${eq1}
   write(error_unit, '(a)', advance = "no") "${eq2}: "
   write(error_unit, *) ${eq2}
   if (len_trim("${message}") /= 0) then
      write(error_unit, '(a)') "Extra message: '${message}'"
   end if
   error stop ${code}
end if
```

Listing 8: assert-eq-fortran(fortran)

`${cond}` が `.false.` ならエラー.

```
true_false=false
cond_origin="${cond}"
cond=".not. (${cond})"
code=${code}
message="${message}"
cat << EOF
<<assert-fortran>>
EOF
```

Listing 9: assert(bash)

${cond} が .true. ならエラー.

```
true_false=false
cond_origin="${cond}"
cond="${cond}"
code=${code}
message="${message}"
cat << EOF
<<assert-fortran>>
EOF
```

Listing 10: assert-false(bash)

```
true_false=false
cond_origin="${eq1} == ${eq2}"
cond=".not. (${cond_origin})"
code=${code}
message="${message}"
cat << EOF
<<assert-eq-fortran>>
EOF
```

Listing 11: assert-eq(bash)

```fortran
program hi
  use, intrinsic :: iso_fortran_env
  implicit none
  integer(int32) :: a = 1, b = 2
  if (2<1) then
     write(error_unit, '(a, i0, a)', advance = "no")&
          "Error in "//&
          __FILE__&
          //":", __LINE__, ":"
     write(error_unit, '(a)') " Assertion '2<1' must be false."
     if (len_trim("in main") /= 0) then
        write(error_unit, '(a)') "Extra message: 'in main'"
     end if
     error stop 1
  end if

  if (.not. (a < b)) then
     write(error_unit, '(a, i0, a)', advance = "no")&
          "Error in "//&
          __FILE__&
          //":", __LINE__, ":"
     write(error_unit, '(a)') " Assertion 'a < b' must be false."
     if (len_trim("") /= 0) then
        write(error_unit, '(a)') "Extra message: ''"
     end if
     error stop 2
  end if

end program hi
```

Listing 12: assert-test(fortran)

3 utilities

3.1 swap

3.1.1 base code

We write a swap subroutine by `Fortran`. This takes two variables and swaps values of them. So, this is impure. We can expand `bash` variables that are expressed by `${variable}`, so decide the type of variables later src block.

- Let us explain `bash` variables.
  - `${type_arg}` is the type of `i, j`.
  - `${type_tmp}` is the type of `tmp` and is usually the same as `${type_arg}`. If `${type_arg}` is `character(len=*)`, `${type_tmp}` must be `character(len=:), allocatable`.
  - `${suffix}` is the suffix of name of subroutine for generic.

This is the whole subroutine. The algorithm of the swap is listing 14.

```
subroutine swap_${suffix}(i, j)
  ${type_arg}, intent(inout) :: i, j
  ${type_tmp} :: tmp
  <<swap-subroutine-body>>
end subroutine swap_${suffix}
```

Listing 13: swap-subroutine(fortran)

The algorithm of the swap is simple. We store the `i` in `tmp`, substitute `j` into `i` and `tmp` into `j`.

```
tmp = i
i   = j
j   = tmp
```

Listing 14: swap-subroutine-body(fortran)

3.1.2 process base code by bash

```
case "${type_arg}" in
    "character")
        suffix="character"
        type_tmp="character(len=max(len(i), len(j)))"
        type_arg="character(len=*)"
        ;;
    *)
        suffix="${type_kind}"
        type_tmp="${type_arg}(${type_kind})"
        type_arg="${type_tmp}"
        ;;
esac
cat <<EOF
<<swap-subroutine>>
EOF
```

Listing 15: swap-subroutine-var(bash)

3.1.3 module

```
module swap_m
  use, intrinsic :: iso_fortran_env
  implicit none
  private
  public :: swap
  !> ,swap: swap the two elements in the array.
  !> This is generic function for (int32, int64, real32, real64, character).
  interface swap
     module procedure :: swap_int32, swap_int64
     module procedure :: swap_real32, swap_real64
     module procedure :: swap_character
  end interface swap

contains

  <<swap-subroutine-var(type_arg="integer", type_kind="int32")>>
  <<swap-subroutine-var(type_arg="integer", type_kind="int64")>>
  <<swap-subroutine-var(type_arg="real", type_kind="real32")>>
  <<swap-subroutine-var(type_arg="real", type_kind="real64")>>
  <<swap-subroutine-var(type_arg="character")>>

end module swap_m
```

Listing 16: swap-module(fortran)

3.1.4 test

```f90
program test_swap
  use, intrinsic :: iso_fortran_env
  use swap_m
  implicit none
  integer      :: i
  integer      :: a(6) = [1, 2, 3, 4, 5, 6], a_init(6)
  integer      :: tmp_i
  real(real64) :: b(6), b_first(6)
  real(real64) :: epsilon = 1d-6
  character(len=3) :: strs(4) = [character(len=3)::"hi", "hoi", "hey", "hui"],
↪  strs_init(4)

  a_init(:) = a(:)
  call swap(a(1), a(1))
  if (sum(a_init - a) /= 0) then
     error stop 1
  end if
  ! print'(*(i0, " "))', (a(i), i = 1, size(a))
  call swap(a(2), a(1))
  ! print'(*(i0, " "))', (a(i), i = 1, size(a))
  if (a_init(2) /= a(1) .or. a_init(1) /= a(2)) then
     error stop 2
  end if

  call random_number(b)
  b_first(:) = b(:)
  ! print'(*(f5.3, " "))', (b(i), i = 1, size(b))
  call swap(b(3), b(4))
  ! print'(*(f5.3, " "))', (b(i), i = 1, size(b))
  if (abs(b_first(4) - b(3)) > epsilon .or. abs(b_first(3) - b(4)) > epsilon) then
     error stop 3
  end if

  strs_init = strs
  ! print'(4(a, ", "))', (strs(i), i = 1, size(strs))
  call swap(strs(4), strs(1))
  ! print'(4(a, ", "))', (strs(i), i = 1, size(strs))
  if (strs_init(4) /= strs(1) .or. strs_init(1) /= strs(4)) then
     error stop 4
  end if

end program test_swap
```

```
<<swap-module>>
<<swap-test>>
```

Listing 18: test-swap(fortran)

3.2 is-sorted

3.2.1 base

```
!> ,is_sorted: Check arr is sorted in the ${op} order.
!> arguments:
!> arr: array of ${type}.
!> return:
!> ${res}: logical, .true. if arr is sorted.
!> variables:
!> i: integer, loop counter.
pure logical function is_sorted_${suffix}(arr) result(${res})
  ${type}, intent(in) :: arr(:)
  integer(int32) :: i
  ${res} = .true.
  do i = 1, size(arr)-1
     if (.not. (arr(i) ${op} arr(i+1))) then
        ${res} = .false.
        return
     end if
  end do
end function is_sorted_${suffix}
```

Listing 19: is-sorted-function(fortran)

```
order=""
if [ "${op}" = ">=" ]; then
    order="descending_"
fi
case "${type}" in
    "character")
        type="${type}(len=*)"
        suffix="${order}character"
    ;;
    *)
        type="${type}(${type_kind})"
        suffix="${order}${type_kind}"
    ;;
esac
res="sorted"
cat <<EOF
!> ,is_sorted: Check arr is sorted in the ${op} order.
!> arguments:
!> arr: array of ${type}.
!> return:
!> ${res}: logical, .true. if arr is sorted.
!> variables:
!> i: integer, loop counter.
pure logical function is_sorted_${suffix}(arr) result(${res})
  ${type}, intent(in) :: arr(:)
  integer(int32) :: i
  ${res} = .true.
  do i = 1, size(arr)-1
     if (.not. (arr(i) ${op} arr(i+1))) then
        ${res} = .false.
        return
     end if
  end do
end function is_sorted_${suffix}
EOF
```

Listing 20: is-sorted-function-var(bash)

3.2.2 module

```
module is_sorted_m
  use, intrinsic :: iso_fortran_env
  implicit none
  private
  public :: is_sorted, is_sorted_descending
  !> ,is_sorted: Check arr is sorted and return logical value.
  !> This is generic function for (int32, int64, real32, real64, character).
  interface is_sorted
     module procedure :: is_sorted_int32, is_sorted_int64
     module procedure :: is_sorted_real32, is_sorted_real64
     module procedure :: is_sorted_character
  end interface is_sorted
  interface is_sorted_descending
     module procedure :: is_sorted_descending_int32, is_sorted_descending_int64
     module procedure :: is_sorted_descending_real32, is_sorted_descending_real64
     module procedure :: is_sorted_descending_character
  end interface is_sorted_descending

contains

!!! Check an array is sorted in the ascending order.
  !> ,is_sorted: Check arr is sorted in the <= order.
  !> arguments:
  !> arr: array of integer(int32).
  !> return:
  !> sorted: logical, .true. if arr is sorted.
  !> variables:
  !> i: integer, loop counter.
  pure logical function is_sorted_int32(arr) result(sorted)
    integer(int32), intent(in) :: arr(:)
    integer(int32) :: i
    sorted = .true.
    do i = 1, size(arr)-1
       if (.not. (arr(i) <= arr(i+1))) then
          sorted = .false.
          return
       end if
    end do
  end function is_sorted_int32

  !> ,is_sorted: Check arr is sorted in the <= order.
  !> arguments:
  !> arr: array of integer(int64).
```

3.2.3 test

```fortran
program is_sorted_test
  use, intrinsic :: iso_fortran_env
  use is_sorted_m
  use merge_sort_m
  implicit none
  integer(int64)    :: sorted_arr(4) = [1_int64, 10_int64, 10_int64, 100_int64]
  real(real32)      :: arr(10)
  character(len=10) :: strings(5) = [character(len=10) :: "apple", "apple", "banana",
↪  "brain", "brought"]
  if (.not. is_sorted(sorted_arr)) then
     error stop 1
  end if
  call random_number(arr)
  call merge_sort(arr)
  if (.not. is_sorted(arr)) then
     error stop 2
  end if
  if (.not. is_sorted(strings)) then
     error stop 3
  end if
end program is_sorted_test
```

Listing 22: is-sorted-test(fortran)

```
module is_sorted_m
  use, intrinsic :: iso_fortran_env
  implicit none
  private
  public :: is_sorted, is_sorted_descending
  !> ,is_sorted: Check arr is sorted and return logical value.
  !> This is generic function for (int32, int64, real32, real64, character).
  interface is_sorted
     module procedure :: is_sorted_int32, is_sorted_int64
     module procedure :: is_sorted_real32, is_sorted_real64
     module procedure :: is_sorted_character
  end interface is_sorted
  interface is_sorted_descending
     module procedure :: is_sorted_descending_int32, is_sorted_descending_int64
     module procedure :: is_sorted_descending_real32, is_sorted_descending_real64
     module procedure :: is_sorted_descending_character
  end interface is_sorted_descending

contains

!!! Check an array is sorted in the ascending order.
  !> ,is_sorted: Check arr is sorted in the <= order.
  !> arguments:
  !> arr: array of integer(int32).
  !> return:
  !> sorted: logical, .true. if arr is sorted.
  !> variables:
  !> i: integer, loop counter.
  pure logical function is_sorted_int32(arr) result(sorted)
    integer(int32), intent(in) :: arr(:)
    integer(int32) :: i
    sorted = .true.
    do i = 1, size(arr)-1
       if (.not. (arr(i) <= arr(i+1))) then
          sorted = .false.
          return
       end if
    end do
  end function is_sorted_int32

  !> ,is_sorted: Check arr is sorted in the <= order.
  !> arguments:
  !> arr: array of integer(int64).
  !> return:
  !> sorted: logical, .true. if arr is sorted.
  !> variables:
```

3.3 compress coordinate class

3.3.1 base

3.3.2 module

```
module compress_m
  use, intrinsic :: iso_fortran_env
  use merge_sort_m
  use binary_search_m
  implicit none
  private
  public :: compress
  type :: compress
     integer(int32) :: size_, ub_
     integer(int32), allocatable :: sorted_(:)
   contains
     procedure, pass :: init       => init_compress
     procedure, pass :: compress   => compress_compress
     procedure, pass :: decompress => decompress_compress
  end type compress
contains
  subroutine init_compress(this, arr)
    class(compress), intent(inout) :: this
    integer(int32), intent(inout) :: arr(:)
    integer(int32), allocatable :: tmp(:)
    integer(int32) :: i
    this%size_ = size(arr)
    allocate(this%sorted_(this%size_), tmp(this%size_))
    this%sorted_ = arr
    call merge_sort(this%sorted_)

    i = 1
    this%ub_ = 0
    unique:do
       if (i == this%size_) then
          this%ub_ = this%ub_ + 1
          this%sorted_(this%ub_) = this%sorted_(i)
          exit
       end if
       if (this%sorted_(i) == this%sorted_(i+1)) then
          i = i + 1
          cycle
       end if
       this%ub_ = this%ub_ + 1
       this%sorted_(this%ub_) = this%sorted_(i)
```

3.4 extend euclid

```fortran
module extend_euclid_m
  use, intrinsic :: iso_fortran_env
  implicit none
  private
  public :: extend_euclid
contains
  subroutine extend_euclid(a, b, g, x, y)
    integer(int32), intent(in)  :: a, b
    integer(int32), intent(out) :: g, x, y
    integer(int32) :: q, old, next
    integer(int32) :: zs(0:1), xs(0:1), ys(0:1)
    zs(0) = a; zs(1) = b
    xs(0) = 1; xs(1) = 0
    ys(0) = 0; ys(1) = 1
    old = 1
    do
       next = ieor(old, 1)
       if (zs(old) == 0) exit
       q = zs(next) / zs(old)
       zs(next) = zs(next) - q*zs(old)
       xs(next) = xs(next) - q*xs(old)
       ys(next) = ys(next) - q*ys(old)
       ! write(error_unit, '(*(i0, 1x))') zs(next), q, xs(next), ys(next)
       old = next
    end do
    x = xs(next)
    y = ys(next)
    g = a*x + b*y
  end subroutine extend_euclid
end module extend_euclid_m
```

Listing 25: (fortran) label:

3.5 polymorphic class(*)

Fortran has polymorphic type `class(*)`. We can store any values in a variable of `class(*) :: var` and extract value from it by `select type` statement.

3.5.1 test

```
module polymorphic_class_m
  use, intrinsic :: iso_fortran_env
  implicit none
  ! interface operator(.as.)
  !    module procedure :: class_to_int32
  !    module procedure :: class_to_real32
  ! end interface operator(.as.)

contains
  pure integer(int32) function class_to_int32(v)
    class(*), intent(in) :: v
    select type(v)
    type is(integer(int32))
       class_to_int32 = v
    class default
       error stop 2
    end select
  end function class_to_int32

  pure integer(int32) function class_to_int32_dash(v, n)
    class(*), intent(in) :: v
    integer(int32), intent(in) :: n
    select type(v)
    type is(integer(int32))
       class_to_int32_dash = v
    class default
       error stop 2
    end select
  end function class_to_int32_dash
  pure real(real32) function class_to_real32(v, x)
    class(*), intent(in) :: v
    real(real32), intent(in) :: x
    select type(v)
    type is(real(real32))
       class_to_real32 = v
    class default
       error stop 3
    end select
  end function class_to_real32
end module polymorphic_class_m

program test_polymorphic_class
  use, intrinsic :: iso_fortran_env
```

4 sorting

4.1 insertion-sort

4.1.1 base code

The Fortran code for insertion sort, which is impure subroutine. The subroutine `insertion_sort_${suffix}` is generated once or more in the below org source block.

- `${bash variable}` will be expanded by bash.
  - `${type}` is the type of `arr(:)`.
  - `${type_key}` is the type of `key` and is usually the same as `${type}`.
  - `${suffix}` is the suffix of the name of the subroutine for avoiding name collision.
  - `${op}` is either `<` (ascending order) or `>` (descending order) .

The subroutine `insertion_sort` takes an argument `arr(:)` in line insertion-sort-arr, sorts it and `arr(:)` was sorted in the end. The type of the `key` is usually the same as type of an element in `arr(:)`, but if the type of `arr(:)` is `character(len=*)`, `key` must be `character(len=:), allocatable`. The type of `arr_size, i, j` is `integer(int32)`. `arr_size` is size of `arr(:)`. `i, j` is loop counter. If $size(arr) > 2^{31} - 1$, this subroutine goes fail, but in this case, we should use more effective sorting algorithms.

```
!> ,insertion_sort_${suffix}: Sort arr in the ${op} order by insertion-sort.
!> arguments:
!> arr: Array of ${type}.
!> In end of subroutine, arr is sorted.
!> variables:
!> key: ${type_key}, insert key into arr(1:i-1).
!> arr_size: integer, size of arr.
!> i, j: integer, loop counter.
subroutine insertion_sort_${suffix}(arr)
  $type, intent(inout) :: arr(:)
↪  (insertion-sort-arr)
  $type_key :: key
  integer(int32) :: arr_size, i, j
  arr_size = size(arr)
  do i = 2, arr_size
     key = arr(i)
     do j = i-1, 1, -1
        if (arr(j) $op key) exit
        arr(j+1) = arr(j)
     end do
     arr(j+1) = key
  end do
end subroutine insertion_sort_${suffix}
```

Listing 27: insertion-sort-subroutine(fortran)

4.1.2 process base code by bash

We want to expand the variables in the above base code by the various types. Pass the variables `type`, `type_kind` and `op` by org-babel :var.

```
order=""
if [ "${op}" = ">" ]; then
    order="descending_"
fi
case "${type}" in
    "character")
        type_key="character(len=:), allocatable"
        type="character(len=*)"
        suffix="${order}character"
        ;;
    *)
        type_key="${type}(${type_kind})"
        type="${type}(${type_kind})"
        suffix="${order}${type_kind}"
        ;;
esac
cat <<EOF
<<insertion-sort-subroutine>>
EOF
```

Listing 28: insertion-sort-subroutine-var(bash)

4.1.3 module

We want to expand the variables in the above base code by the various types. We can pass the arguments to the above org source block. So, our insertion sort is the generic subroutine for the array of `integer(int32)`, `integer(int64)`, `real(real32)`, `real(real64)`, and `character(len=*)` This module exports `insertion_sort` and `insertion_sort_descending`.

```fortran
module insertion_sort_m
  use, intrinsic :: iso_fortran_env
  implicit none
  private

  public :: insertion_sort, insertion_sort_descending
  !> ,insertion_sort: Sort arr in ascending order.
  !> This is generic subroutine for (int32, int64, real32, real64, character).
  interface insertion_sort
     module procedure :: insertion_sort_int32, insertion_sort_int64
     module procedure :: insertion_sort_real32, insertion_sort_real64
     module procedure :: insertion_sort_character
  end interface insertion_sort
  !> ,insertion_sort_descending: Sort arr in descending order.
  !> This is generic subroutine for (int32, int64, real32, real64, character).
  interface insertion_sort_descending
     module procedure :: insertion_sort_descending_int32,
↪  insertion_sort_descending_int64
     module procedure :: insertion_sort_descending_real32,
↪  insertion_sort_descending_real64
     module procedure :: insertion_sort_descending_character
  end interface insertion_sort_descending

contains

!!! Sort an array in the ascending order.
  <<insertion-sort-subroutine-var(type="integer", type_kind="int32", op="<")>>
  <<insertion-sort-subroutine-var(type="integer", type_kind="int64", op="<")>>
  <<insertion-sort-subroutine-var(type="real", type_kind="real32", op="<")>>
  <<insertion-sort-subroutine-var(type="real", type_kind="real64", op="<")>>
  <<insertion-sort-subroutine-var(type="character", op="<")>>
!!! Sort an array in the descending order.
  <<insertion-sort-subroutine-var(type="integer", type_kind="int32", op=">")>>
  <<insertion-sort-subroutine-var(type="integer", type_kind="int64", op=">")>>
  <<insertion-sort-subroutine-var(type="real", type_kind="real32", op=">")>>
  <<insertion-sort-subroutine-var(type="real", type_kind="real64", op=">")>>
  <<insertion-sort-subroutine-var(type="character", op=">")>>

end module insertion_sort_m
```

Listing 29: insertion-sort-module(fortran)

4.1.4 test

```fortran
program test_insertion_sort
  use, intrinsic :: iso_fortran_env
  use is_sorted_m
  use insertion_sort_m
  implicit none
  integer      :: i
  integer      :: a(6) = [31, 41, 59, 26, 41, 58]
  real(real64) :: b(100)
  character(len=42) :: c(5) = ["a    ", "zzz  ", "123  ", "0    ", "     "]

  ! print'(*(i0, " "))', (a(i), i = 1, size(a))
  call insertion_sort(a)
  if (.not. is_sorted(a)) error stop 1
  ! print'(*(i0, " "))', (a(i), i = 1, size(a))
  call random_number(b)
  ! print'(*(f5.3, " "))', (b(i), i = 1, size(b))
  call insertion_sort(b)
  if (.not. is_sorted(b)) error stop 2
  ! print'(*(f5.3, " "))', (b(i), i = 1, size(b))
  call insertion_sort(c)
  if (.not. is_sorted(c)) error stop 3
end program test_insertion_sort
```

Listing 30: insertion-sort-test(fortran)

```
<<is-sorted-module>>
<<insertion-sort-module>>
<<insertion-sort-test>>
```

Listing 31: test-insertion-sort(fortran)

4.2 selection-sort

4.2.1 base

```fortran
integer :: arr_size, mini_index, i, j
!> ,selection_sort: Sort arr of some type by selection-sort.
!> arguments:
!> arr: array of some type.
!> variables:
!> arr_size: integer, size of arr(:).
!> mini_index: integer, index of minimum value in arr(j:arr_size).
!> i, j: integer, loop counters.
arr_size = size(arr)
do j = 1, arr_size
   mini_index = j
   do i = j+1, arr_size
      if (arr(i) < arr(mini_index)) then
         mini_index = i
      end if
   end do
   call swap(arr(j), arr(mini_index))
end do
```

Listing 32: selection-sort(fortran)

4.2.2 module

```
module selection_sort_m
  use, intrinsic :: iso_fortran_env
  use swap_m
  implicit none
  private
  public :: selection_sort
  interface selection_sort
     module procedure :: selection_sort_int32, selection_sort_int64
     module procedure :: selection_sort_real32, selection_sort_real64
  end interface selection_sort

contains

  subroutine selection_sort_int32(arr)
    integer(int32), intent(inout) :: arr(:)
    <<selection-sort>>
  end subroutine selection_sort_int32
  subroutine selection_sort_int64(arr)
    integer(int64), intent(inout) :: arr(:)
    <<selection-sort>>
  end subroutine selection_sort_int64
  subroutine selection_sort_real32(arr)
    real(real32), intent(inout) :: arr(:)
    <<selection-sort>>
  end subroutine selection_sort_real32
  subroutine selection_sort_real64(arr)
    real(real64), intent(inout) :: arr(:)
    <<selection-sort>>
  end subroutine selection_sort_real64

end module selection_sort_m
```

Listing 33: selection-sort-module(fortran)

4.2.3 test

```
<<swap-module>>
<<selection-sort-module>>

program test_selection_sort
  use, intrinsic :: iso_fortran_env
  use selection_sort_m
  implicit none
  ! integer :: arr(9) = [8, 3, 1, 9, 5, 4, 2, 7, 6]
  integer :: arr(-2:6) = [9, 8, 7, 6, 5, 4, 3, 2, 1]
  integer :: i

  !    do i = -2, 6
  !       arr(i) = i
  !    end do

  print'(*(i0, " "))', (arr(i), i = lbound(arr, dim = 1), ubound(arr, dim = 1))
  call selection_sort(arr)
  print'(*(i0, " "))', (arr(i), i = lbound(arr, dim = 1), ubound(arr, dim = 1))

end program test_selection_sort
```

Listing 34: selection-sort-test(fortran)

```
9 8 7 6 5 4 3 2 1
1 2 3 4 5 6 7 8 9
```

4.3 bubble-sort

4.3.1 base

```fortran
integer(int32) :: size_arr, i, j
!> ,bubble_sort: Sort arr of some type by bubble-sort.
!> arguments:
!> arr: array of some type.
!> variables:
!> arr_size: integer, size of arr(:).
!> i, j: integer, loop counters.
size_arr = size(arr)
do i = 1, size_arr
   do j = size_arr, i+1, -1
      if (arr(j) < arr(j-1)) then
         call swap(arr(j), arr(j-1))
      end if
   end do
end do
```

Listing 35: bubble-sort(fortran)

4.3.2 module

```
module bubble_sort_m
  use, intrinsic :: iso_fortran_env
  use swap_m
  implicit none
  private
  public :: bubble_sort
  interface bubble_sort
     module procedure :: bubble_sort_int32, bubble_sort_int64
     module procedure :: bubble_sort_real32, bubble_sort_real64
  end interface bubble_sort

contains

  subroutine bubble_sort_int32(arr)
    integer(int32), intent(inout) :: arr(:)
    integer(int32) :: size_arr, i, j
    !> ,bubble_sort: Sort arr of some type by bubble-sort.
    !> arguments:
    !> arr: array of some type.
    !> variables:
    !> arr_size: integer, size of arr(:).
    !> i, j: integer, loop counters.
    size_arr = size(arr)
    do i = 1, size_arr
       do j = size_arr, i+1, -1
          if (arr(j) < arr(j-1)) then
             call swap(arr(j), arr(j-1))
          end if
       end do
    end do
  end subroutine bubble_sort_int32
  subroutine bubble_sort_int64(arr)
    integer(int64), intent(inout) :: arr(:)
    integer(int32) :: size_arr, i, j
    !> ,bubble_sort: Sort arr of some type by bubble-sort.
    !> arguments:
    !> arr: array of some type.
    !> variables:
    !> arr_size: integer, size of arr(:).
    !> i, j: integer, loop counters.
    size_arr = size(arr)
    do i = 1, size_arr
       do j = size_arr, i+1, -1
```

4.3.3 test

```
module swap_m
  use, intrinsic :: iso_fortran_env
  implicit none
  private
  public :: swap
  !> ,swap: swap the two elements in the array.
  !> This is generic function for (int32, int64, real32, real64, character).
  interface swap
     module procedure :: swap_int32, swap_int64
     module procedure :: swap_real32, swap_real64
     module procedure :: swap_character
  end interface swap

contains

  subroutine swap_int32(i, j)
    integer(int32), intent(inout) :: i, j
    integer(int32) :: tmp
    tmp = i
    i   = j
    j   = tmp
  end subroutine swap_int32

  subroutine swap_int64(i, j)
    integer(int64), intent(inout) :: i, j
    integer(int64) :: tmp
    tmp = i
    i   = j
    j   = tmp
  end subroutine swap_int64

  subroutine swap_real32(i, j)
    real(real32), intent(inout) :: i, j
    real(real32) :: tmp
    tmp = i
    i   = j
    j   = tmp
  end subroutine swap_real32

  subroutine swap_real64(i, j)
    real(real64), intent(inout) :: i, j
    real(real64) :: tmp
    tmp = i
```

4.4 merge-sort

4.4.1 module

```
module merge_sort_m
  use, intrinsic :: iso_fortran_env
  implicit none
  private
  public :: merge_sort, merge_sort_descending
  interface merge_sort
     module procedure :: merge_sort_int32
     module procedure :: merge_sort_with_key_int32
  end interface merge_sort
  interface merge_sort_descending
     module procedure :: merge_sort_int32_descending
     module procedure :: merge_sort_with_key_int32_descending
  end interface merge_sort_descending

  interface merge_sort
     module procedure :: merge_sort_int64
     module procedure :: merge_sort_with_key_int64
  end interface merge_sort
  interface merge_sort_descending
     module procedure :: merge_sort_int64_descending
     module procedure :: merge_sort_with_key_int64_descending
  end interface merge_sort_descending

  interface merge_sort
     module procedure :: merge_sort_real32
     module procedure :: merge_sort_with_key_real32
  end interface merge_sort
  interface merge_sort_descending
     module procedure :: merge_sort_real32_descending
     module procedure :: merge_sort_with_key_real32_descending
  end interface merge_sort_descending

  interface merge_sort
     module procedure :: merge_sort_real64
     module procedure :: merge_sort_with_key_real64
  end interface merge_sort
  interface merge_sort_descending
     module procedure :: merge_sort_real64_descending
     module procedure :: merge_sort_with_key_real64_descending
  end interface merge_sort_descending

contains
```

4.4.2 merge_sort モジュールの宣言

```
interface merge_sort${order}
   module procedure :: merge_sort_${suffix}
   module procedure :: merge_sort_with_key_${suffix}
end interface merge_sort${order}
```

Listing 39: declaration-merge_sort(fortran)

4.4.3 merge_sort 関連の関数

```
!> merge_sort_${suffix}: Sort arr(:) by sub function merge_sort_sub_${suffix}.
!> arguments:
!> arr: array of some type.
subroutine merge_sort_${suffix}(arr)
  ${type}, intent(inout) :: arr(:)
  call merge_sort_sub_${suffix}(arr, 1, size(arr))
end subroutine merge_sort_${suffix}
```

Listing 40: merge_sort(fortran)

■4.4.3.1 merge_sort

```fortran
!> merge: Algorithm for merge_sort, check if Left or Right is end in each loop.
!> arguments:
!> arr: array of some type, (out) arr(p:r) is sorted.
!> p, q, r: integer, indices p is start, r is end, q = floor( (p+q)/2 ).
!> variables:
!> Left, Right: array of typeof(arr), sorted
!> l_max, r_max: integer, max index of Left or Right.
subroutine merge_${suffix}(arr, p, q, r)
  ${type}, intent(inout) :: arr(:)
  integer(int32), intent(in) :: p, q, r
  ${type}                :: Left(1:q-p+1), Right(1:r-q)
  integer(int32)             :: l_max, r_max
  l_max = q-p+1
  r_max = r-q
  block
    !> i, j, k: integer, loop counters.
    integer(int32) :: i, j, k
    Left(1:l_max)  = arr(p:q)
    Right(1:r_max) = arr(q+1:r)
    i = 1
    j = 1
    do k = p, r
       if (Left(i) ${op} Right(j)) then
          arr(k) = Left(i)
          i = i + 1
          if (i > l_max) then
             arr(k+1:r) = Right(j:)
             return
          end if
       else
          arr(k) = Right(j)
          j = j + 1
          if (j > r_max) then
             arr(k+1:r) = Left(i:)
             return
          end if
       end if
    end do
  end block
end subroutine merge_${suffix}
```


Listing 41: merge(fortran)

■4.4.3.2 merge

```
!> merge_sort_sub: Recursive function used by merge_sort.
!> arguments:
!> arr: array of some type.
!> p, r: integer, p is start of arr, r is end of arr.
!> variables:
!> q: integer, q = floor( (p+q)/2 )
recursive subroutine merge_sort_sub_${suffix}(arr, p, r)
  ${type}, intent(inout) :: arr(:)
  integer(int32), intent(in) :: p, r
  integer(int32)             :: q
  if (p < r) then
     q = (p+r)/2
     call merge_sort_sub_${suffix}(arr, p, q)
     call merge_sort_sub_${suffix}(arr, q+1, r)
     call merge_${suffix}(arr, p, q, r)
  end if
end subroutine merge_sort_sub_${suffix}
```

Listing 42: merge_sort_sub(fortran)

■4.4.3.3 merge_sort_sub

```
!> merge_sort_with_key_${suffix}: Sort key(:) sub function
↪  merge_sort_sub_with_key_${suffix}.
!> arguments:
!> key: array of some type.
!> indices: array of some type.
subroutine merge_sort_with_key_${suffix}(key, indices)
  ${type}, intent(inout) :: key(:)
  integer(int32), intent(inout) :: indices(:)
  call merge_sort_sub_with_key_${suffix}(key, indices, 1, size(key))
end subroutine merge_sort_with_key_${suffix}
```

Listing 43: merge_sort_with_key(fortran)

■4.4.3.4 merge_sort_with_key

```
!> merge_with_key: Algorithm for merge_sort, check if Left or Right is end in each loop.
!> arguments:
!> indices: array of indices.
!> key: array of some type, (out) key(p:r) is sorted.
!> p, q, r: integer, indices p is start, r is end, q = floor( (p+q)/2 ).
!> variables:
!> Left, Right: array of typeof(indices), sorted
!> l_max, r_max: integer, max index of Left or Right.
subroutine merge_with_key_${suffix}(key, indices, p, q, r)
  ${type}, intent(inout) :: key(:)
  integer(int32), intent(inout) :: indices(:)
  integer(int32), intent(in) :: p, q, r
  integer(int32) :: Left(1:q-p+1), Right(1:r-q)
  ${type} :: Left_key(1:q-p+1), Right_key(1:r-q)
  integer(int32) :: l_max, r_max
  l_max = q-p+1
  r_max = r-q
  block
    !> i, j, k: integer, loop counters.
    integer(int32) :: i, j, k
    Left(1:l_max)  = indices(p:q)
    Right(1:r_max) = indices(q+1:r)
    Left_key(1:l_max)  = key(p:q)
    Right_key(1:r_max) = key(q+1:r)
    i = 1
    j = 1
    do k = p, r
       if (Left_key(i) ${op} Right_key(j)) then
          key(k) = Left_key(i)
          indices(k) = Left(i)
          i = i + 1
          if (i > l_max) then
             key(k+1:r) = Right_key(j:)
             indices(k+1:r) = Right(j:)
             return
          end if
       else
          key(k) = Right_key(j)
          indices(k) = Right(j)
          j = j + 1
          if (j > r_max) then
             key(k+1:r) = Left_key(i:)
             indices(k+1:r) = Left(i:)
             return
```

■4.4.3.5 merge_with_key

```
!> merge_sort_sub_with_key: Recursive function used by merge_sort_with_key.
!> arguments:
!> indices: array of indices.
!> key: array of some type.
!> p, r: integer, p is start of arr, r is end of arr.
!> variables:
!> q: integer, q = floor( (p+q)/2 )
recursive subroutine merge_sort_sub_with_key_${suffix}(key, indices, p, r)
  ${type}, intent(inout) :: key(:)
  integer(int32), intent(inout) :: indices(:)
  integer(int32), intent(in) :: p, r
  integer(int32)             :: q
  if (p < r) then
     q = (p+r)/2
     call merge_sort_sub_with_key_${suffix}(key, indices, p, q)
     call merge_sort_sub_with_key_${suffix}(key, indices, q+1, r)
     call merge_with_key_${suffix}(key, indices, p, q, r)
  end if
end subroutine merge_sort_sub_with_key_${suffix}
```

Listing 45: merge_sort_sub_with_key(fortran)

■4.4.3.6 merge_sort_sub_with_key

4.4.4 merge_sort 関連の変数の展開

```
case "${type_base}" in
    "character")
        type="character"
        suffix="character"
        ;;
    *)
        type="${type_base}(${type_kind})"
        suffix="${type_kind}"
        ;;
esac
case "${op}" in
    "<=")
        order=""
        ;;
    ">=")
        order="_descending"
        ;;
esac
suffix="${suffix}${order}"
```

Listing 46: merge_sort-var(bash)

```
for op in "<=" ">="
do
<<merge_sort-var>>
cat <<EOF
<<declaration-merge_sort>>
EOF
done
```

Listing 47: declaration-merge_sort-var(bash)

```
for op in "<=" ">="
do
<<merge_sort-var>>
cat <<EOF
<<procedures-merge_sort>>
EOF
done
```

Listing 48: procedures-merge_sort-var(bash)

4.4.5 test

```
program test_merge
  use, intrinsic :: iso_fortran_env
  use merge_sort_m
  implicit none
  integer(int32) :: arr(9) = [8, 3, 1, 9, 5, 4, 2, 7, 6]
  integer(int32), allocatable :: indices(:)
  integer(int32) :: i

  indices = [(i, i = 1, 9)]
  call merge_sort(arr, indices)
  if (.not. (all(arr(:) == [1,2,3,4,5,6,7,8,9]))) then
     write(error_unit, '(a, i0, a)', advance = "no")&
          "Error in "//&
          __FILE__&
          //":", __LINE__, ":"
     write(error_unit, '(a)') " Assertion 'all(arr(:) == [1,2,3,4,5,6,7,8,9])' must be
↪  false."
     if (len_trim("merge_sort with key is illegal.") /= 0) then
        write(error_unit, '(a)') "Extra message: 'merge_sort with key is illegal.'"
     end if
     error stop 11
  end if

  if (.not. (all(indices(:) == [3,7,2,6,5,9,8,1,4]))) then
     write(error_unit, '(a, i0, a)', advance = "no")&
          "Error in "//&
          __FILE__&
          //":", __LINE__, ":"
     write(error_unit, '(a)') " Assertion 'all(indices(:) == [3,7,2,6,5,9,8,1,4])' must
↪  be false."
     if (len_trim("merge_sort with key is illegal.") /= 0) then
        write(error_unit, '(a)') "Extra message: 'merge_sort with key is illegal.'"
     end if
     error stop 12
  end if

  call merge_sort_descending(arr)
  if (.not. (all(arr(:) == [9,8,7,6,5,4,3,2,1]))) then
     write(error_unit, '(a, i0, a)', advance = "no")&
          "Error in "//&
          __FILE__&
          //":", __LINE__, ":"
     write(error_unit, '(a)') " Assertion 'all(arr(:) == [9,8,7,6,5,4,3,2,1])' must be
```

```
module merge_sort_m
  use, intrinsic :: iso_fortran_env
  implicit none
  private
  public :: merge_sort, merge_sort_descending
  interface merge_sort
     module procedure :: merge_sort_int32
     module procedure :: merge_sort_with_key_int32
  end interface merge_sort
  interface merge_sort_descending
     module procedure :: merge_sort_int32_descending
     module procedure :: merge_sort_with_key_int32_descending
  end interface merge_sort_descending

  interface merge_sort
     module procedure :: merge_sort_int64
     module procedure :: merge_sort_with_key_int64
  end interface merge_sort
  interface merge_sort_descending
     module procedure :: merge_sort_int64_descending
     module procedure :: merge_sort_with_key_int64_descending
  end interface merge_sort_descending

  interface merge_sort
     module procedure :: merge_sort_real32
     module procedure :: merge_sort_with_key_real32
  end interface merge_sort
  interface merge_sort_descending
     module procedure :: merge_sort_real32_descending
     module procedure :: merge_sort_with_key_real32_descending
  end interface merge_sort_descending

  interface merge_sort
     module procedure :: merge_sort_real64
     module procedure :: merge_sort_with_key_real64
  end interface merge_sort
  interface merge_sort_descending
     module procedure :: merge_sort_real64_descending
     module procedure :: merge_sort_with_key_real64_descending
  end interface merge_sort_descending

contains
  !> merge_sort_int32: Sort arr(:) by sub function merge_sort_sub_int32.
  !> arguments:
  !> arr: array of some type.
  subroutine merge_sort_int32(arr)
```

4.5 heap-sort(未完成)

4.5.1 base code

```
subroutine heap_sort${suffix}(arr)
  ${type_arg}, intent(inout) :: arr(:)
  integer(int32) :: size, i
  size = size(arr)
  do i = 1, size
     call shift_up(arr, i)
  end do
  do i = size-1, 1, -1
     call swap(1, arr(i))
     call shift_down(arr, i)
  end do
end subroutine heap_sort${suffix}
```

Listing 51: heap_sort(fortran)

```
subroutine shift_up${suffix}(arr, n)
  ${type_arg}, intent(inout) :: arr(:)
  integer(int32), intent(in) :: n
  integer(int32) :: pos
  pos = n
  do
     if (pos == 1) exit
     if (arr(pos) > arr(pos/2)) exit
     call swap(arr(pos), arr(pos/2))
     pos = pos/2
  end do
end subroutine shift_up${suffix}
```

Listing 52: shift_up(fortran)

```
subroutine shift_down${suffix}(arr, n)
  ${type_arg}, intent(inout) :: arr(:)
  integer(int32), intent(in) :: n
  integer(int32) :: pos
  pos = 1
  do
     if (pos*2 > n) exit
     if (arr(pos*2) > arr(pos)) then
        pos = pos*2
        if (pos == n) then
           call swap(arr(pos), arr(pos/2))
           exit
        end if
        if (arr(pos*2+1) > arr(pos*2)) pos = pos+1
        call swap(arr(pos), arr(pos/2))
     end if
  end do
end subroutine shift_down${suffix}
```

Listing 53: shift_down(fortran)

4.5.2 test

4.6 radix-sort

```f90
module radix_sort_m
  use, intrinsic :: iso_fortran_env
  use unwrapped_vector_m
  implicit none
  private
  integer(int32), parameter :: ten_pow(10) = [1, 10, 10**2, 10**3, 10**4, 10**5, 10**6, 10**7, 10**8, 10**9]
  public :: radix_sort
  interface radix_sort
     module procedure :: radix_sort_int32
  end interface radix_sort
contains
  subroutine radix_sort_int32(arr, pow_max)
    integer(int32), intent(inout) :: arr(:)
    integer(int32), intent(in) :: pow_max
    integer(int32) :: n, i, p, r, idx
    integer(int32), allocatable :: arr_tmp(:, :)
    integer(int32) :: old, next
    type(unwrapped_vector_int32) :: radix(-9:9)
    n = size(arr)
    allocate(arr_tmp(n, 0:1))
    old = 0
    arr_tmp(:, old) = arr(:)
    do p = 0, pow_max
       next = ieor(old, 1)
       do r = -9, 9
          call radix(r)%resize(0)
       end do
       do i = 1, n
          r = mod(arr_tmp(i, old) / ten_pow(p+1), 10)
          ! write(error_unit, '(*(i0, 1x))') p, i, r
          call radix(r)%push_back(i)
       end do
       idx = 0
       do r = -9, 9
          do i = 1, radix(r)%size()
             idx = idx + 1
             arr_tmp(idx, next) = arr_tmp(radix(r)%arr_(i), old)
          end do
       end do
       old = next
    end do
```

5 search

5.1 binary-search

5.1.1 base

```fortran
integer(int32), intent(in) :: lb, ub
integer(int32) :: p, q, r
!> ,binary_search: Search v from arr
!> arguments:
!> v: typeof(v).
!> arr: array of some type.
!> lb, ub: integer, lower bound and upper bound of arr.
!> return:
!> pos: position of v in arr if lb <= pos <= ub.
!> v does not exist in arr if pos = lb-1.
!> variables:
!> p, r: integer, range of search [p, r]
!> q: integer, q = floor( (p+r)/2 ).
p = lb
r = ub
do
   if (p > r) then
      pos = lb-1
      return
   end if
   q = int((p+r)/2, int32)
   if (arr(q) == v) then
      pos = q
      return
   else if (arr(q) < v) then
      p = q + 1
   else
      r = q - 1
   end if
end do
```

Listing 55: binary-search(fortran)

5.1.2 module

```
module binary_search_m
  use, intrinsic :: iso_fortran_env
  implicit none
  private
  public :: binary_search
  interface binary_search
     module procedure :: binary_search_int32, binary_search_int64
  end interface binary_search

contains

  pure integer function binary_search_int32(v, arr, lb, ub) result(pos)
    integer(int32), intent(in) :: v
    integer(int32), intent(in) :: arr(lb:ub)
    integer(int32), intent(in) :: lb, ub
    integer(int32) :: p, q, r
    !> ,binary_search: Search v from arr
    !> arguments:
    !> v: typeof(v).
    !> arr: array of some type.
    !> lb, ub: integer, lower bound and upper bound of arr.
    !> return:
    !> pos: position of v in arr if lb <= pos <= ub.
    !> v does not exist in arr if pos = lb-1.
    !> variables:
    !> p, r: integer, range of search [p, r]
    !> q: integer, q = floor( (p+r)/2 ).
    p = lb
    r = ub
    do
       if (p > r) then
          pos = lb-1
          return
       end if
       q = int((p+r)/2, int32)
       if (arr(q) == v) then
          pos = q
          return
       else if (arr(q) < v) then
          p = q + 1
       else
          r = q - 1
       end if
```

5.1.3 test

```fortran
program test_binary_search
  use, intrinsic :: iso_fortran_env
  use binary_search_m
  implicit none
  integer :: arr(-1:7) = [1, 2, 3, 4, 4, 6, 7, 8, 9]
  integer :: i
  if (binary_search(2, arr, -1, 7) /= 0) then
     error stop 1
  else if (binary_search(5, arr, -1, 7) /= lbound(arr, dim = 1)-1) then
     error stop 2
  else if (binary_search(9, arr, -1, 7) /= 7) then
     error stop 3
  end if
end program test_binary_search
```

Listing 57: binary-search-test(fortran)

```
module binary_search_m
  use, intrinsic :: iso_fortran_env
  implicit none
  private
  public :: binary_search
  interface binary_search
     module procedure :: binary_search_int32, binary_search_int64
  end interface binary_search

contains

  pure integer function binary_search_int32(v, arr, lb, ub) result(pos)
    integer(int32), intent(in) :: v
    integer(int32), intent(in) :: arr(lb:ub)
    integer(int32), intent(in) :: lb, ub
    integer(int32) :: p, q, r
    !> ,binary_search: Search v from arr
    !> arguments:
    !> v: typeof(v).
    !> arr: array of some type.
    !> lb, ub: integer, lower bound and upper bound of arr.
    !> return:
    !> pos: position of v in arr if lb <= pos <= ub.
    !> v does not exist in arr if pos = lb-1.
    !> variables:
    !> p, r: integer, range of search [p, r]
    !> q: integer, q = floor( (p+r)/2 ).
    p = lb
    r = ub
    do
       if (p > r) then
          pos = lb-1
          return
       end if
       q = int((p+r)/2, int32)
       if (arr(q) == v) then
          pos = q
          return
       else if (arr(q) < v) then
          p = q + 1
       else
          r = q - 1
       end if
    end do
  end function binary_search_int32
  pure integer function binary_search_int64(v, arr, lb, ub) result(pos)
```

5.2 lower_bound

5.2.1 whole module of the lower_bound

This is whole module of the ‘lower_bound‘. There are several types for ‘lower_bound‘.

```
module lower_bound_m
  use, intrinsic :: iso_fortran_env
  implicit none
  private
  <<declaration-lower_bound-var(type="integer", type_kind="int8")>>
  <<declaration-lower_bound-var(type="integer", type_kind="int16")>>
  <<declaration-lower_bound-var(type="integer", type_kind="int32")>>
  <<declaration-lower_bound-var(type="integer", type_kind="int64")>>
  <<declaration-lower_bound-var(type="real", type_kind="real32")>>
  <<declaration-lower_bound-var(type="real", type_kind="real64")>>
  public :: lower_bound
contains
  <<procedures-lower_bound-var(type="integer", type_kind="int8")>>
  <<procedures-lower_bound-var(type="integer", type_kind="int16")>>
  <<procedures-lower_bound-var(type="integer", type_kind="int32")>>
  <<procedures-lower_bound-var(type="integer", type_kind="int64")>>
  <<procedures-lower_bound-var(type="real", type_kind="real32")>>
  <<procedures-lower_bound-var(type="real", type_kind="real64")>>
end module lower_bound_m
```

Listing 59: lower_bound-module(fortran)

5.2.2 declaration of the lower_bound

```
interface lower_bound
   module procedure :: lower_bound_${suffix}
end interface lower_bound
```

Listing 60: declaration-lower_bound(fortran)

5.2.3 procedures of the lower_bound

`function lower_bound` searches the index that has the element that is higher than or equal to the ‘val‘. Index starts from 1.

```
!> lower_bound_${suffix}: Search
pure integer(int32) function lower_bound_${suffix}(arr, val) result(res)
  ${type}, intent(in) :: arr(:)
  ${type}, intent(in) :: val
  integer(int32) :: p, q, r
  p = 1
  r = size(arr)
  if (arr(p) >= val) then
     res = p
     return
  else if (arr(r) < val) then
     res = r + 1
     return
  end if
  !> a, b, ..., k, `val`, l, ..., z
  !> arr(p) < val
  !> arr(r) >= val
  do
     q = (p+r)/2
     if (p + 1 >= r) exit
     if (arr(q) >= val) then
        r = q
     else
        p = q
     end if
  end do
  res = r
end function lower_bound_${suffix}
```

Listing 61: procedures-lower_bound(fortran)

5.2.4 process definition and procedures of the lower_bound

```
case "${type}" in
    "character")
        type="character"
        suffix="character"
        ;;
    *)
        type="${type}(${type_kind})"
        suffix="${type_kind}"
        ;;
esac
cat <<EOF
<<declaration-lower_bound>>
EOF
```

Listing 62: declaration-lower_bound-var(bash)

```
case "${type}" in
    "character")
        type="character"
        suffix="character"
        ;;
    *)
        type="${type}(${type_kind})"
        suffix="${type_kind}"
        ;;
esac
cat <<EOF
<<procedures-lower_bound>>
EOF
```

Listing 63: procedures-lower_bound-var(bash)

5.2.5 test

- Test the array that has several values.

- Test the array that has only one value.
- Test the array that has some same values.
- Test the array that has all same values.

```
program test_lower_bound
  use, intrinsic :: iso_fortran_env
  use lower_bound_m
  implicit none
  integer(int32) :: i, j
  integer(int32), parameter :: n = 10
  integer(int32) :: arr(n), arr2(1), dup_arr(n), allsame_arr(n)
  do i = 1, n
     arr(i) = i
  end do
  ! arr
  do i = 0, n
     j = lower_bound(arr, i)
     <<assert(cond="j == max(1, i)",code=11,message="`lower_bound` does not work well...")>>
  end do
  j = lower_bound(arr, n+1)
  <<assert(cond="j == size(arr)+1",code=12,message="`lower_bound` does not work well...")>>
  ! arr2
  arr2(1) = 7
  <<assert(cond="lower_bound(arr2, 6) == 1",code=13,message="`lower_bound` does not work well for one element array...")>>
  <<assert(cond="lower_bound(arr2, 7) == 1",code=14,message="`lower_bound` does not work well for one element array...")>>
  <<assert(cond="lower_bound(arr2, 8) == 2",code=15,message="`lower_bound` does not work well for one element array...")>>
  ! dup_arr
  dup_arr = [1, 1, 2, 3, 3, 3, 3, 5, 5, 5]
  <<assert(cond="lower_bound(dup_arr, 0) == 1",code=21,message="`lower_bound` does not work well for the array that has same values...")>>
  <<assert(cond="lower_bound(dup_arr, 2) == 3",code=22,message="`lower_bound` does not work well for the array that has same values...")>>
  <<assert(cond="lower_bound(dup_arr, 3) == 4",code=23,message="`lower_bound` does not work well for the array that has same values...")>>
  <<assert(cond="lower_bound(dup_arr, 5) == 8",code=24,message="`lower_bound` does not work well for the array that has same values...")>>
  <<assert(cond="lower_bound(dup_arr, 7) > size(dup_arr)",code=25,message="`lower_bound` does not work well for the array that has same values...")>>
  ! allsame_arr
  allsame_arr = [(1, i = 1, n)]
  <<assert(cond="lower_bound(allsame_arr, 0) == 1",code=31,message="`lower_bound` does not work well for the array that has all same values...")>>
  <<assert(cond="lower_bound(allsame_arr, 1) == 1",code=32,message="`lower_bound` does
```

```
<<lower_bound-module>>
<<lower_bound-test>>
```

Listing 65: test-lower_bound(fortran)

5.3 upper_bound

5.3.1 whole module of the upper_bound

This is whole module of the ‘upper_bound‘. There are several types for ‘upper_bound‘.

```
module upper_bound_m
  use, intrinsic :: iso_fortran_env
  implicit none
  private
  <<declaration-upper_bound-var(type="integer", type_kind="int8")>>
  <<declaration-upper_bound-var(type="integer", type_kind="int16")>>
  <<declaration-upper_bound-var(type="integer", type_kind="int32")>>
  <<declaration-upper_bound-var(type="integer", type_kind="int64")>>
  <<declaration-upper_bound-var(type="real", type_kind="real32")>>
  <<declaration-upper_bound-var(type="real", type_kind="real64")>>
  public :: upper_bound
contains
  <<procedures-upper_bound-var(type="integer", type_kind="int8")>>
  <<procedures-upper_bound-var(type="integer", type_kind="int16")>>
  <<procedures-upper_bound-var(type="integer", type_kind="int32")>>
  <<procedures-upper_bound-var(type="integer", type_kind="int64")>>
  <<procedures-upper_bound-var(type="real", type_kind="real32")>>
  <<procedures-upper_bound-var(type="real", type_kind="real64")>>
end module upper_bound_m
```

Listing 66: upper_bound-module(fortran)

5.3.2 declaration of the upper_bound

```
interface upper_bound
   module procedure :: upper_bound_${suffix}
end interface upper_bound
```

Listing 67: declaration-upper_bound(fortran)

5.3.3 procedures of the upper_bound

`function upper_bound` searches the index that has the element that is higher than the ‘val‘. Index starts from 1.

```fortran
!> upper_bound_${suffix}: Search
pure integer(int32) function upper_bound_${suffix}(arr, val) result(res)
  ${type}, intent(in) :: arr(:)
  ${type}, intent(in) :: val
  integer(int32) :: p, q, r
  p = 1
  r = size(arr)
  if (arr(p) > val) then
     res = p
     return
  else if (arr(r) <= val) then
     res = r + 1
     return
  end if
  !> a, b, ..., k, `val`, l, ..., z
  !> arr(p) <= val
  !> arr(r) >  val
  do
     q = (p+r)/2
     if (p + 1 >= r) exit
     if (arr(q) > val) then
        r = q
     else
        p = q
     end if
  end do
  res = r
end function upper_bound_${suffix}
```

Listing 68: procedures-upper_bound(fortran)

5.3.4 process definition and procedures of the upper_bound

```
case "${type}" in
    "character")
        type="character"
        suffix="character"
        ;;
    *)
        type="${type}(${type_kind})"
        suffix="${type_kind}"
        ;;
esac
```

Listing 69: upper_bound-var(bash)

```
<<upper_bound-var>>
cat <<EOF
<<declaration-upper_bound>>
EOF
```

Listing 70: declaration-upper_bound-var(bash)

```
<<upper_bound-var>>
cat <<EOF
<<procedures-upper_bound>>
EOF
```

Listing 71: procedures-upper_bound-var(bash)

5.3.5 test

- Test the array that has several values.
- Test the array that has only one value.
- Test the array that has some same values.
- Test the array that has all same values.

```
program test_upper_bound
  use, intrinsic :: iso_fortran_env
  use upper_bound_m
  implicit none
  integer(int32) :: i, j
  integer(int32), parameter :: n = 10
  integer(int32) :: arr(n), arr2(1), dup_arr(n), allsame_arr(n)
  do i = 1, n
     arr(i) = i
  end do
  ! arr
  do i = 0, n
     j = upper_bound(arr, i)
     <<assert(cond="j == i+1",code=11,message="`upper_bound` does not work well...")>>
  end do
  j = upper_bound(arr, n+1)
  <<assert(cond="j == size(arr)+1",code=12,message="`upper_bound` does not work
  ↪  well...")>>
  ! arr2
  arr2(1) = 7
  <<assert(cond="upper_bound(arr2, 6) == 1",code=13,message="`upper_bound` does not
  ↪  work well for one element array...")>>
  <<assert(cond="upper_bound(arr2, 7) == 2",code=14,message="`upper_bound` does not
  ↪  work well for one element array...")>>
  <<assert(cond="upper_bound(arr2, 8) == 2",code=15,message="`upper_bound` does not
  ↪  work well for one element array...")>>
  ! dup_arr
  dup_arr = [1, 1, 2, 3, 3, 3, 3, 5, 5, 5]
  <<assert(cond="upper_bound(dup_arr, 0) == 1",code=21,message="`upper_bound` does not
  ↪  work well for the array that has same values...")>>
  <<assert(cond="upper_bound(dup_arr, 1) == 3",code=22,message="`upper_bound` does not
  ↪  work well for the array that has same values...")>>
  <<assert(cond="upper_bound(dup_arr, 2) == 4",code=23,message="`upper_bound` does not
  ↪  work well for the array that has same values...")>>
  <<assert(cond="upper_bound(dup_arr, 4) == 8",code=24,message="`upper_bound` does not
  ↪  work well for the array that has same values...")>>
  <<assert(cond="upper_bound(dup_arr, 5) ==
  ↪  size(dup_arr)+1",code=25,message="`upper_bound` does not work well for the array
  ↪  that has same values...")>>
  <<assert(cond="upper_bound(dup_arr, 7) ==
  ↪  size(dup_arr)+1",code=26,message="`upper_bound` does not work well for the array
  ↪  that has same values...")>>
  ! allsame_arr
  allsame_arr = [(1, i = 1, n)]
  <<assert(cond="upper_bound(allsame_arr, 0) == 1",code=31,message="`upper_bound` does
  ↪  not work well for the array that has all same values...")>>
```

```
1 <<upper_bound-module>>
2 <<upper_bound-test>>
```

Listing 73: test-upper_bound(fortran)

6 math

6.1 prime

6.1.1 prime factorization

7 data structure

7.1 String

```
module string_m
  use, intrinsic :: iso_fortran_env
  implicit none
  private string_row
  type :: string_row
     integer(int32) :: ref_cnt_ = 0_int32
     character, allocatable :: str_(:)
   contains
     final :: destroy_string_row
  end type string_row
  type :: string
     private
     integer(int32) :: size_ = 0_int32
     type(string_row), pointer :: ptr_ => null()
   contains
     procedure, pass :: assign_string, assign_chars
     generic :: assignment(=) => assign_string
     final :: destroy_string
  end type string
  interface assignment(=)
     module procedure :: assign_string_to_chars
  end interface assignment(=)
  private equal_string, not_equal_string
  private less_string, less_equal_string, greater_string, greater_equal_string
  interface operator(==)
     module procedure :: equal_string
  end interface operator(==)
  interface operator(/=)
     module procedure :: not_equal_string
  end interface operator(/=)
  interface operator(<)
     module procedure :: less_string
  end interface operator(<)
  interface operator(<=)
     module procedure :: less_equal_string
```

7.2 Tuple

7.2.1 Tuple2

```
module tuple2_m
  use, intrinsic :: iso_fortran_env
  implicit none
  <<declaration-tuple2-var(type1="integer", type1_kind="int32", type2="integer",
↪  type2_kind="int32")>>
  <<declaration-tuple2-var(type1="integer", type1_kind="int64", type2="integer",
↪  type2_kind="int64")>>
contains
  <<procedures-tuple2-var(type1="integer", type1_kind="int32", type2="integer",
↪  type2_kind="int32")>>
  <<procedures-tuple2-var(type1="integer", type1_kind="int64", type2="integer",
↪  type2_kind="int64")>>
end module tuple2_m
```

Listing 75: tuple2-module(fortran)

■7.2.1.1 Tuple2 モジュール全体

■7.2.1.2 **Tuple2 の宣言** まず, 型 `${tuple2}` の宣言を行う. `${tuple2}` は 2 つの要素を持つ. 2 つの要素の型は異なる型でも構わない.

変数 `${variable}` は `bash` で展開される.

- `${tuple2}` は Tuple2 型.
- `${type1}` は Tuple2 型の一番目の型.
- `${type2}` は Tuple2 型の二番目の型.

```
public :: ${tuple2}
type :: ${tuple2}
   private
   ${type1} :: fst_
   ${type2} :: snd_
 contains
   procedure, pass :: fst  => fst_${tuple2}
   procedure, pass :: snd  => snd_${tuple2}
end type ${tuple2}
private :: construct_${tuple2}
interface ${tuple2}
   module procedure :: construct_${tuple2}
end interface ${tuple2}
interface operator(<)
   module procedure :: less_${tuple2}
end interface operator(<)
interface operator(<=)
   module procedure :: less_equal_${tuple2}
end interface operator(<=)
interface operator(>)
   module procedure :: greater_${tuple2}
end interface operator(>)
interface operator(>=)
   module procedure :: greater_equal_${tuple2}
end interface operator(>=)
interface operator(==)
   module procedure :: equal_${tuple2}
end interface operator(==)
interface operator(/=)
   module procedure :: not_equal_${tuple2}
end interface operator(/=)
```

Listing 76: declaration-tuple2(fortran)

■7.2.1.3　Tuple2 の関数とか

- constructor `function construct` は Tuple2 型を生成する.

```
!> construct_${tuple2}_by_size: Construct ${tuple2}.
impure function construct_${tuple2}(val1, val2) result(res)
  type(${tuple2}) :: res
  ${type1}, intent(in) :: val1
  ${type2}, intent(in) :: val2
  res%fst_ = val1
  res%snd_ = val2
end function construct_${tuple2}
```

Listing 77: construct-tuple2(fortran)

- fst `function fst` は Tuple2 の一番目の要素を返す.

```
!> fst_${tuple2}: Return the first element of ${tuple2}.
${type1} function fst_${tuple2}(this) result(res)
  class(${tuple2}), intent(in) :: this
  res = this%fst_
end function fst_${tuple2}
```

Listing 78: fst-tuple2(fortran)

- snd `function snd` は Tuple2 の二番目の要素を返す.

```
!> snd_${tuple2}: Return the second element of ${tuple2}.
${type1} function snd_${tuple2}(this) result(res)
  class(${tuple2}), intent(in) :: this
  res = this%snd_
end function snd_${tuple2}
```

Listing 79: snd-tuple2(fortran)

- compare_operator 比較演算子たち.

```
!> less_${tuple2}: Compare the first elements.
!> Compare the second elements if the first elements are same.
logical function less_${tuple2}(lhs, rhs) result(res)
  type(${tuple2}), intent(in) :: lhs, rhs
  res = lhs%fst_ < rhs%fst_
  if (lhs%fst_ == rhs%fst_) then
     res = lhs%snd_ < rhs%snd_
  end if
end function less_${tuple2}
logical function less_equal_${tuple2}(lhs, rhs) result(res)
  type(${tuple2}), intent(in) :: lhs, rhs
  res = lhs%fst_ < rhs%fst_
  if (lhs%fst_ == rhs%fst_) then
     res = lhs%snd_ <= rhs%snd_
  end if
end function less_equal_${tuple2}
logical function greater_${tuple2}(lhs, rhs) result(res)
  type(${tuple2}), intent(in) :: lhs, rhs
  res = lhs%fst_ > rhs%fst_
  if (lhs%fst_ == rhs%fst_) then
     res = lhs%snd_ > rhs%snd_
  end if
end function greater_${tuple2}
logical function greater_equal_${tuple2}(lhs, rhs) result(res)
  type(${tuple2}), intent(in) :: lhs, rhs
  res = lhs%fst_ > rhs%fst_
  if (lhs%fst_ == rhs%fst_) then
     res = lhs%snd_ >= rhs%snd_
  end if
end function greater_equal_${tuple2}
logical function equal_${tuple2}(lhs, rhs) result(res)
  type(${tuple2}), intent(in) :: lhs, rhs
  res = lhs%fst_ == rhs%fst_ .and. lhs%snd_ == rhs%snd_
end function equal_${tuple2}
logical function not_equal_${tuple2}(lhs, rhs) result(res)
  type(${tuple2}), intent(in) :: lhs, rhs
  res = lhs%fst_ /= rhs%fst_ .or. lhs%snd_ /= rhs%snd_
end function not_equal_${tuple2}
```

Listing 80: compare-tuple2(fortran)

```
suffix=""
case "${type1}" in
    "character")
        type1="character"
        suffix="${suffix}_character"
        ;;
    *)
        type1="${type1}(${type1_kind})"
        suffix="${suffix}_${type1_kind}"
        ;;
esac
case "${type2}" in
    "character")
        type2="character"
        suffix="${suffix}_character"
        ;;
    *)
        type2="${type2}(${type2_kind})"
        suffix="${suffix}_${type2_kind}"
        ;;
esac
tuple2="tuple2${suffix}"
```

Listing 81: tuple2-var(bash)

```
<<tuple2-var>>
cat <<EOF
<<declaration-tuple2>>
EOF
```

Listing 82: declaration-tuple2-var(bash)

```
<<tuple2-var>>
cat <<EOF
<<procedures-tuple2>>
EOF
```

Listing 83: procedures-tuple2-var(bash)

■7.2.1.4　Tuple2 の展開

```fortran
program test_tuple2
  use, intrinsic :: iso_fortran_env
  use tuple2_m
  implicit none
  type(tuple2_int32_int32) :: t1, t2
  t1 = tuple2_int32_int32(1, 1)
  <<assert(cond="t1 == t1",       code=10, message="`==` for Tuple2 is illegal.")>>
  <<assert-false(cond="t1 /= t1", code=11, message="`/=` for Tuple2 is illegal.")>>
  <<assert-false(cond="t1 < t1",  code=12, message="`<` for Tuple2 is illegal.")>>
  <<assert(cond="t1 >= t1",       code=13, message="`>=` for Tuple2 is illegal.")>>
  <<assert-false(cond="t1 > t1",  code=14, message="`>` for Tuple2 is illegal.")>>
  <<assert(cond="t1 <= t1",       code=15, message="`<=` for Tuple2 is illegal.")>>
  t2 = tuple2_int32_int32(1, 2)
  <<assert-false(cond="t1 == t2", code=20, message="`==` for Tuple2 is illegal.")>>
  <<assert(cond="t1 /= t2",       code=21, message="`/=` for Tuple2 is illegal.")>>
  <<assert(cond="t1 < t2",        code=22, message="`<` for Tuple2 is illegal.")>>
  <<assert-false(cond="t1 >= t2", code=23, message="`>=` for Tuple2 is illegal.")>>
  <<assert-false(cond="t1 > t2",  code=24, message="`>` for Tuple2 is illegal.")>>
  <<assert(cond="t1 <= t2",       code=25, message="`<=` for Tuple2 is illegal.")>>
  t2 = tuple2_int32_int32(100, 2)
  <<assert-false(cond="t1 == t2", code=30, message="`==` for Tuple2 is illegal.")>>
  <<assert(cond="t1 /= t2",       code=31, message="`/=` for Tuple2 is illegal.")>>
  <<assert(cond="t1 < t2",        code=32, message="`<` for Tuple2 is illegal.")>>
  <<assert-false(cond="t1 >= t2", code=33, message="`>=` for Tuple2 is illegal.")>>
  <<assert-false(cond="t1 > t2",  code=34, message="`>` for Tuple2 is illegal.")>>
  <<assert(cond="t1 <= t2",       code=35, message="`<=` for Tuple2 is illegal.")>>
  t2 = tuple2_int32_int32(1, -100)
  <<assert-false(cond="t1 == t2", code=40, message="`==` for Tuple2 is illegal.")>>
  <<assert(cond="t1 /= t2",       code=41, message="`/=` for Tuple2 is illegal.")>>
  <<assert-false(cond="t1 < t2",  code=42, message="`<` for Tuple2 is illegal.")>>
  <<assert(cond="t1 >= t2",       code=43, message="`>=` for Tuple2 is illegal.")>>
  <<assert(cond="t1 > t2",        code=44, message="`>` for Tuple2 is illegal.")>>
  <<assert-false(cond="t1 <= t2", code=45, message="`<=` for Tuple2 is illegal.")>>
end program test_tuple2
```

Listing 84: tuple2-test(fortran)

```
<<tuple2-module>>
<<tuple2-test>>
```

Listing 85: test-tuple2(fortran)

■7.2.1.5 test

```
module tuple2_priority_queue_m
  use, intrinsic :: iso_fortran_env
  use tuple2_m
  implicit none
  <<declaration-priority_queue-var(type_base="type", type_kind="tuple2_int32_int32")>>
  <<declaration-priority_queue-var(type_base="type", type_kind="tuple2_int64_int64")>>
contains
  <<procedures-priority_queue-var(type_base="type", type_kind="tuple2_int32_int32")>>
  <<procedures-priority_queue-var(type_base="type", type_kind="tuple2_int64_int64")>>
end module tuple2_priority_queue_m
```

Listing 86: tuple2-priority_queue-module(fortran)

■7.2.1.6 Tuple2 の priority_queue

7.2.2 Tuple3

```
module tuple3_m
  use, intrinsic :: iso_fortran_env
  implicit none
  <<declaration-tuple3-var(type1="integer", type1_kind="int32", type2="integer", type2_kind="int32", type3="integer", type3_kind="int32")>>
  <<declaration-tuple3-var(type1="integer", type1_kind="int64", type2="integer", type2_kind="int64", type3="integer", type3_kind="int64")>>
contains
  <<procedures-tuple3-var(type1="integer", type1_kind="int32", type2="integer", type2_kind="int32", type3="integer", type3_kind="int32")>>
  <<procedures-tuple3-var(type1="integer", type1_kind="int64", type2="integer", type2_kind="int64", type3="integer", type3_kind="int64")>>
end module tuple3_m
```

Listing 87: tuple3-module(fortran)

■7.2.2.1 Tuple3 モジュール全体

■7.2.2.2 Tuple3 の宣言 まず, 型 `${tuple3}` の宣言を行う. `${tuple3}` は 3 つの要素を持つ. 3 つの要素の型は異なる型でも構わない.

変数 `${variable}` は `bash` で展開される.

- `${tuple3}` は Tuple3 型.
- `${type1}` は Tuple3 型の一番目の型.
- `${type2}` は Tuple3 型の二番目の型.
- `${type3}` は Tuple3 型の三番目の型.

```
public :: ${tuple3}
type :: ${tuple3}
   private
   ${type1} :: fst_
   ${type2} :: snd_
   ${type3} :: thr_
 contains
   procedure, pass :: fst  => fst_${tuple3}
   procedure, pass :: snd  => snd_${tuple3}
   procedure, pass :: thr  => thr_${tuple3}
end type ${tuple3}
public :: construct_${tuple3}
interface ${tuple3}
   module procedure :: construct_${tuple3}
end interface ${tuple3}
interface operator(<)
   module procedure :: less_${tuple3}
end interface operator(<)
interface operator(<=)
   module procedure :: less_equal_${tuple3}
end interface operator(<=)
interface operator(>)
   module procedure :: greater_${tuple3}
end interface operator(>)
interface operator(>=)
   module procedure :: greater_equal_${tuple3}
end interface operator(>=)
interface operator(==)
   module procedure :: equal_${tuple3}
end interface operator(==)
interface operator(/=)
   module procedure :: not_equal_${tuple3}
end interface operator(/=)
```

Listing 88: declaration-tuple3(fortran)

■7.2.2.3　Tuple3 の関数とか

- constructor `function construct` は Tuple3 型を生成する.

```
!> construct_${tuple3}_by_size: Construct ${tuple3}.
impure function construct_${tuple3}(val1, val2, val3) result(res)
  type(${tuple3}) :: res
  ${type1}, intent(in) :: val1
  ${type2}, intent(in) :: val2
  ${type3}, intent(in) :: val3
  res%fst_ = val1
  res%snd_ = val2
  res%thr_ = val3
end function construct_${tuple3}
```

Listing 89: construct-tuple3(fortran)

- fst `function fst` は Tuple3 の一番目の要素を返す.

```
!> fst_${tuple3}: Return the first element of ${tuple3}.
${type1} function fst_${tuple3}(this) result(res)
  class(${tuple3}), intent(in) :: this
  res = this%fst_
end function fst_${tuple3}
```

Listing 90: fst-tuple3(fortran)

- snd `function snd` は Tuple3 の二番目の要素を返す.

```
!> snd_${tuple3}: Return the second element of ${tuple3}.
${type1} function snd_${tuple3}(this) result(res)
  class(${tuple3}), intent(in) :: this
  res = this%snd_
end function snd_${tuple3}
```

Listing 91: snd-tuple3(fortran)

- thr `function thr` は Tuple3 の三番目の要素を返す.

```
!> thr_${tuple3}: Return the second element of ${tuple3}.
${type1} function thr_${tuple3}(this) result(res)
  class(${tuple3}), intent(in) :: this
  res = this%thr_
end function thr_${tuple3}
```

Listing 92: snd-tuple3(fortran)

- compare_operator　比較演算子たち.

```
!> less_${tuple3}: Compare the first elements.
!> Compare the second elements if the first elements are same.
logical function less_${tuple3}(lhs, rhs) result(res)
  type(${tuple3}), intent(in) :: lhs, rhs
  res = lhs%fst_ < rhs%fst_
  if (lhs%fst_ == rhs%fst_) then
     res = lhs%snd_ < rhs%snd_
     if (lhs%snd_ == rhs%snd_) then
        res = lhs%thr_ < rhs%thr_
     end if
  end if
end function less_${tuple3}
logical function less_equal_${tuple3}(lhs, rhs) result(res)
  type(${tuple3}), intent(in) :: lhs, rhs
  res = lhs%fst_ < rhs%fst_
  if (lhs%fst_ == rhs%fst_) then
     res = lhs%snd_ < rhs%snd_
     if (lhs%snd_ == rhs%snd_) then
        res = lhs%thr_ <= rhs%thr_
     end if
  end if
end function less_equal_${tuple3}
logical function greater_${tuple3}(lhs, rhs) result(res)
  type(${tuple3}), intent(in) :: lhs, rhs
  res = lhs%fst_ > rhs%fst_
  if (lhs%fst_ == rhs%fst_) then
     res = lhs%snd_ > rhs%snd_
     if (lhs%snd_ == rhs%snd_) then
        res = lhs%thr_ > rhs%thr_
     end if
  end if
end function greater_${tuple3}
logical function greater_equal_${tuple3}(lhs, rhs) result(res)
  type(${tuple3}), intent(in) :: lhs, rhs
  res = lhs%fst_ > rhs%fst_
  if (lhs%fst_ == rhs%fst_) then
     res = lhs%snd_ > rhs%snd_
     if (lhs%snd_ == rhs%snd_) then
        res = lhs%thr_ >= rhs%thr_
     end if
  end if
end function greater_equal_${tuple3}
logical function equal_${tuple3}(lhs, rhs) result(res)
  type(${tuple3}), intent(in) :: lhs, rhs
  res = lhs%fst_ == rhs%fst_ .and. lhs%snd_ == rhs%snd_ .and. lhs%thr_ == rhs%thr_
end function equal_${tuple3}
```

```bash
suffix=""
case "${type1}" in
    "character")
        type1="character"
        suffix="${suffix}_character"
        ;;
    *)
        type1="${type1}(${type1_kind})"
        suffix="${suffix}_${type1_kind}"
        ;;
esac
case "${type2}" in
    "character")
        type2="character"
        suffix="${suffix}_character"
        ;;
    *)
        type2="${type2}(${type2_kind})"
        suffix="${suffix}_${type2_kind}"
        ;;
esac
case "${type3}" in
    "character")
        type3="character"
        suffix="${suffix}_character"
        ;;
    *)
        type3="${type3}(${type3_kind})"
        suffix="${suffix}_${type3_kind}"
        ;;
esac
tuple3="tuple3${suffix}"
```

Listing 94: tuple3-var(bash)

```
<<tuple3-var>>
cat <<EOF
<<declaration-tuple3>>
EOF
```

Listing 95: declaration-tuple3-var(bash)

```
<<tuple3-var>>
cat <<EOF
<<procedures-tuple3>>
EOF
```

Listing 96: procedures-tuple3-var(bash)

■7.2.2.4　Tuple3 の展開

```
program test_tuple3
  use, intrinsic :: iso_fortran_env
  use tuple3_m
  implicit none
  type(tuple3_int32_int32_int32) :: t1, t2
  t1 = tuple3_int32_int32_int32(1, 1, 1)
  <<assert(cond="t1 == t1",       code=10, message="`==` for Tuple3 is illegal.")>>
  <<assert-false(cond="t1 /= t1", code=11, message="`/=` for Tuple3 is illegal.")>>
  <<assert-false(cond="t1 < t1",  code=12, message="`<` for Tuple3 is illegal.")>>
  <<assert(cond="t1 >= t1",       code=13, message="`>=` for Tuple3 is illegal.")>>
  <<assert-false(cond="t1 > t1",  code=14, message="`>` for Tuple3 is illegal.")>>
  <<assert(cond="t1 <= t1",       code=15, message="`<=` for Tuple3 is illegal.")>>
  t2 = tuple3_int32_int32_int32(1, 1, 2)
  <<assert-false(cond="t1 == t2", code=20, message="`==` for Tuple3 is illegal.")>>
  <<assert(cond="t1 /= t2",       code=21, message="`/=` for Tuple3 is illegal.")>>
  <<assert(cond="t1 < t2",        code=22, message="`<` for Tuple3 is illegal.")>>
  <<assert-false(cond="t1 >= t2", code=23, message="`>=` for Tuple3 is illegal.")>>
  <<assert-false(cond="t1 > t2",  code=24, message="`>` for Tuple3 is illegal.")>>
  <<assert(cond="t1 <= t2",       code=25, message="`<=` for Tuple3 is illegal.")>>
  t2 = tuple3_int32_int32_int32(1, 2, 2)
  <<assert-false(cond="t1 == t2", code=30, message="`==` for Tuple3 is illegal.")>>
  <<assert(cond="t1 /= t2",       code=31, message="`/=` for Tuple3 is illegal.")>>
  <<assert(cond="t1 < t2",        code=32, message="`<` for Tuple3 is illegal.")>>
  <<assert-false(cond="t1 >= t2", code=33, message="`>=` for Tuple3 is illegal.")>>
  <<assert-false(cond="t1 > t2",  code=34, message="`>` for Tuple3 is illegal.")>>
  <<assert(cond="t1 <= t2",       code=35, message="`<=` for Tuple3 is illegal.")>>
  t2 = tuple3_int32_int32_int32(100, 1, 2)
  <<assert-false(cond="t1 == t2", code=40, message="`==` for Tuple3 is illegal.")>>
  <<assert(cond="t1 /= t2",       code=41, message="`/=` for Tuple3 is illegal.")>>
  <<assert(cond="t1 < t2",        code=42, message="`<` for Tuple3 is illegal.")>>
  <<assert-false(cond="t1 >= t2", code=43, message="`>=` for Tuple3 is illegal.")>>
  <<assert-false(cond="t1 > t2",  code=44, message="`>` for Tuple3 is illegal.")>>
  <<assert(cond="t1 <= t2",       code=45, message="`<=` for Tuple3 is illegal.")>>
  t2 = tuple3_int32_int32_int32(0, 1, 2)
  <<assert-false(cond="t1 == t2", code=50, message="`==` for Tuple3 is illegal.")>>
  <<assert(cond="t1 /= t2",       code=51, message="`/=` for Tuple3 is illegal.")>>
  <<assert-false(cond="t1 < t2",        code=52, message="`<` for Tuple3 is
  ↪  illegal.")>>
  <<assert(cond="t1 >= t2", code=53, message="`>=` for Tuple3 is illegal.")>>
  <<assert(cond="t1 > t2",  code=54, message="`>` for Tuple3 is illegal.")>>
  <<assert-false(cond="t1 <= t2",       code=55, message="`<=` for Tuple3 is
  ↪  illegal.")>>
  t2 = tuple3_int32_int32_int32(1, 1, -100)
  <<assert-false(cond="t1 == t2", code=50, message="`==` for Tuple3 is illegal.")>>
```

```
<<tuple3-module>>
<<tuple3-test>>
```

Listing 98: test-tuple3(fortran)

■7.2.2.5　test

```
module tuple3_priority_queue_m
  use, intrinsic :: iso_fortran_env
  use tuple3_m
  implicit none
  <<declaration-priority_queue-var(type_base="type",
↪  type_kind="tuple3_int32_int32_int32")>>
  <<declaration-priority_queue-var(type_base="type",
↪  type_kind="tuple3_int64_int64_int64")>>
contains
  <<procedures-priority_queue-var(type_base="type",
↪  type_kind="tuple3_int32_int32_int32")>>
  <<procedures-priority_queue-var(type_base="type",
↪  type_kind="tuple3_int64_int64_int64")>>
end module tuple3_priority_queue_m
```

Listing 99: tuple3-priority_queue-module(fortran)

■7.2.2.6　Tuple3 の priority_queue

7.3 linked list

7.3.1 by pointer

```
module linked_list_m
  use, intrinsic :: iso_fortran_env
  implicit none
  <<declaration-linked_list-var(type="integer", type_kind="int32")>>
  <<declaration-linked_list-var(type="integer", type_kind="int64")>>
  <<declaration-linked_list-var(type="real", type_kind="real32")>>
  <<declaration-linked_list-var(type="real", type_kind="real64")>>
contains
  <<procedures-linked_list-var(type="integer", type_kind="int32")>>
  <<procedures-linked_list-var(type="integer", type_kind="int64")>>
  <<procedures-linked_list-var(type="real", type_kind="real32")>>
  <<procedures-linked_list-var(type="real", type_kind="real64")>>
end module linked_list_m
```

Listing 100: linked_list-module(fortran)

■7.3.1.1 whole module of linked list

■7.3.1.2 declaration of linked list

First, We define type of linked list. This linked list is implemented by a head of list and some lists (0 or more than). The list can add new values, delete, and search some elements.

Variables like ${variable} are expanded by bash.

- ${type} is type of elements in the list.

```
private :: linked_list_${suffix}
type :: linked_list_${suffix}
   private
   ${type} :: val_
   type(linked_list_${suffix}), pointer :: next_ => null()
end type linked_list_${suffix}

public :: linked_list_${suffix}_head
type :: linked_list_${suffix}_head
   private
   type(linked_list_${suffix}), pointer :: head_ => null()
 contains
   procedure, pass :: add    => add_linked_list_${suffix}_head
   procedure, pass :: delete => delete_linked_list_${suffix}_head
   procedure, pass :: search => search_linked_list_${suffix}_head
end type linked_list_${suffix}_head

interface linked_list_${suffix}
   module procedure :: init_linked_list_${suffix}
end interface linked_list_${suffix}
interface linked_list_${suffix}_head
   module procedure :: init_linked_list_${suffix}_head
   module procedure :: init_linked_list_${suffix}_head_by_array
end interface linked_list_${suffix}_head
```

Listing 101: declaration-linked_list(fortran)

■7.3.1.3 procedures of linked list There are four procedures for the linked list.

- init `function init` initialize linked_list and linked_list

```fortran
!> init_linked_list_${suffix}: Initialize the linked_list_${suffix} by val.
impure function init_linked_list_${suffix}(val) result(lst)
  type(linked_list_${suffix}), pointer :: lst
  ${type} :: val
  allocate(lst)
  lst%val_ = val
  return
end function init_linked_list_${suffix}
```

Listing 102: init-linked_list(fortran)

```fortran
!> init_linked_list_${suffix}_head: Initialize the empty linked_list_${suffix}_head.
impure function init_linked_list_${suffix}_head() result(lst_head)
  type(linked_list_${suffix}_head) :: lst_head
  lst_head%head_ => null()
  return
end function init_linked_list_${suffix}_head
!> init_linked_list_${suffix}_head_by_array: Initialize the empty
↪  linked_list_${suffix}_head by array.
impure function init_linked_list_${suffix}_head_by_array(arr) result(lst_head)
  type(linked_list_${suffix}_head) :: lst_head
  ${type} :: arr(:)
  integer(int32) :: s, i
  s = size(arr)
  do i = s, 1, -1
     call lst_head%add(arr(i))
  end do
  return
end function init_linked_list_${suffix}_head_by_array
```

Listing 103: init-linked_list_head(fortran)

- add `Subroutine add` adds value into the linked list.

```fortran
!> add_linked_list_${suffix}: Add val into head of linked list.
subroutine add_linked_list_${suffix}_head(lst_head, val)
  class(linked_list_${suffix}_head), intent(inout) :: lst_head
  ${type}, intent(in) :: val
  type(linked_list_${suffix}), pointer :: lst_elem
  lst_elem => linked_list_${suffix}(val)
  lst_elem%next_ => lst_head%head_
  lst_head%head_ => lst_elem
end subroutine add_linked_list_${suffix}_head
```

Listing 104: add-linked_list_head(fortran)

- delete `Subroutine delete` delete elements in linked list.

```fortran
!> delete_linked_list_${suffix}: Delete val from element of linked list.
!> Do nothing if lst does not elem val.
subroutine delete_linked_list_${suffix}_head(lst_head, val)
  class(linked_list_${suffix}_head), intent(inout) :: lst_head
  ${type}, intent(in) :: val
  type(linked_list_${suffix}), pointer :: lst_elem, lst_del
  if (.not. associated(lst_head%head_)) return
  lst_elem => lst_head%head_
  if (lst_elem%val_ == val) then
     lst_head%head_ => lst_elem%next_
     deallocate(lst_elem)
     return
  end if
  do
     if (.not. associated(lst_elem%next_)) return
     if (lst_elem%next_%val_ == val) then
        lst_del => lst_elem%next_
        lst_elem%next_ => lst_elem%next_%next_
        deallocate(lst_del)
        return
     end if
  end do
end subroutine delete_linked_list_${suffix}_head
```

Listing 105: delete-linked_list_head(fortran)

- search `Subroutine search` search value from linked list and return `.true.` if success.

```
!> search_linked_list_${suffix}: Search val from element of linked list.
!> Return .true. if success.
logical function search_linked_list_${suffix}_head(lst_head, val) result(find)
  class(linked_list_${suffix}_head), intent(in) :: lst_head
  ${type}, intent(in) :: val
  type(linked_list_${suffix}), pointer :: lst_elem
  if (.not. associated(lst_head%head_)) return
  lst_elem => lst_head%head_
  find = .false.
  do
     if (.not. associated(lst_elem)) return
     if (lst_elem%val_ == val) then
        find = .true.
        return
     end if
     lst_elem => lst_elem%next_
  end do
end function search_linked_list_${suffix}_head
```

Listing 106: search-linked_list_head(fortran)

```
case "${type}" in
    "character")
        type="character(len=:), allocatable"
        suffix="character"
        ;;
    *)
        type="${type}(${type_kind})"
        suffix="${type_kind}"
        ;;
esac
cat <<EOF
<<declaration-linked_list>>
EOF
```

Listing 107: declaration-linked_list-var(bash)

```
case "${type}" in
    "character")
        type="character(len=:), allocatable"
        suffix="character"
        ;;
    *)
        type="${type}(${type_kind})"
        suffix="${type_kind}"
        ;;
esac
cat <<EOF
<<procedures-linked_list>>
EOF
```

Listing 108: procedures-linked_list-var(bash)

■7.3.1.4 process definition and procedures of linked list

```fortran
program test_linked_list
  use, intrinsic :: iso_fortran_env
  use linked_list_m
  implicit none
  integer(int32) :: i
  type(linked_list_int32_head) :: lst_i32
  do i = 1, 10
     call lst_i32%add(i)
  end do
  print*, lst_i32%search(3)
  print*, lst_i32%search(-1)
end program test_linked_list
```

Listing 109: linked_list-test(fortran)

```
<<linked_list-module>>
<<linked_list-test>>
```

Listing 110: test-linked_list(fortran)

■7.3.1.5 test

7.4 Vector (Variable array)

7.4.1 Unwrapped Vector

```
1  module unwrapped_vector_m
2    use, intrinsic :: iso_fortran_env
3    implicit none
4    private
5    <<declaration-unwrapped_vector-var(type="integer", type_kind="int32")>>
6    <<declaration-unwrapped_vector-var(type="integer", type_kind="int64")>>
7    <<declaration-unwrapped_vector-var(type="real", type_kind="real32")>>
8    <<declaration-unwrapped_vector-var(type="real", type_kind="real64")>>
9    <<declaration-unwrapped_vector-var(type="character")>>
10 contains
11   <<procedures-unwrapped_vector-var(type="integer", type_kind="int32")>>
12   <<procedures-unwrapped_vector-var(type="integer", type_kind="int64")>>
13   <<procedures-unwrapped_vector-var(type="real", type_kind="real32")>>
14   <<procedures-unwrapped_vector-var(type="real", type_kind="real64")>>
15   <<procedures-unwrapped_vector-var(type="character")>>
16 end module unwrapped_vector_m
```

Listing 111: unwrapped_vector-module(fortran)

■7.4.1.1 whole module of the unwrapped_vector

■7.4.1.2 declaration of the unwrapped_vector

First, We define the type of the unwrapped_vector. These unwrapped_vectors are implemented by the array that shrink and expand. The list can add new values, delete, and search some elements. We can access the member `arr_(:)` directory, so we should take care of the consistency of data.

Variables like ${variable} are expanded by bash.

- ${uwvec} is the name of the type for the unwrapped vector .
- ${type} is the type of the

```
public :: ${uwvec}
type :: ${uwvec}
   private
   ${type}, allocatable, public :: arr_(:)
   integer(int32) :: size_ = 0, capa_ = 0
 contains
   procedure, pass :: init      => init_${uwvec}
   procedure, pass :: push_back_${uwvec}, push_back_array_${uwvec}
   generic         :: push_back => push_back_${uwvec}, push_back_array_${uwvec}
   procedure, pass :: pop_back  => pop_back_${uwvec}
   procedure, pass :: back      => back_${uwvec}
   procedure, pass :: size      => size_${uwvec}
   procedure, pass :: resize    => resize_${uwvec}
   procedure, pass :: lower_bound => lower_bound_${uwvec}
end type ${uwvec}
interface ${uwvec}
   module procedure :: construct_${uwvec}_by_size, &
        construct_${uwvec}_by_arr, &
        construct_${uwvec}_by_init_val
end interface ${uwvec}
```

Listing 112: declaration-unwrapped_vector(fortran)

■7.4.1.3 procedures of the unwrapped vector

- constructor `function construct` constructs unwrapped_vector by size or value.

```
!> construct_${uwvec}_by_size: Construct ${uwvec} by the size, the initial values is
↪  unknown.
impure function construct_${uwvec}_by_size(size) result(res)
  type(${uwvec}) :: res
  integer(int32), intent(in) :: size
  call res%init(size)
end function construct_${uwvec}_by_size
!> construct_${uwvec}_by_arr: Construct ${uwvec} by the array of ${type}.
impure function construct_${uwvec}_by_arr(arr) result(res)
  type(${uwvec}) :: res
  ${type}, intent(in) :: arr(:)
  integer(int32) :: n
  n = size(arr)
  call res%init(n)
  res%arr_(1:n) = arr(1:n)
end function construct_${uwvec}_by_arr
!> construct_${uwvec}_by_init_val: Construct ${uwvec} by size and the initial values.
impure function construct_${uwvec}_by_init_val(size, val) result(res)
  type(${uwvec}) :: res
  integer(int32), intent(in) :: size
  ${type}, intent(in) :: val
  call res%init(size)
  res%arr_(1:size) = val
end function construct_${uwvec}_by_init_val
```

Listing 113: construct-unwrapped_vector(fortran)

- init `subroutine init` initialize unwrapped_vector by size.

```
!> init_${uwvec}: Initialize the ${uwvec} by size.
subroutine init_${uwvec}(this, n)
  class(${uwvec}), intent(inout) :: this
  integer(int32), intent(in) :: n
  if (.not. allocated(this%arr_)) then
     allocate(this%arr_(n))
     this%size_ = n
     this%capa_ = n
  end if
end subroutine init_${uwvec}
```

Listing 114: init-unwrapped_vector(fortran)

- push_back `subroutine push_back` insert value to the tail of elements of the unwrapped vector.

```
!> push_back_${uwvec}: Insert value to the tail of elements of the ${uwvec}.
subroutine push_back_${uwvec}(this, val)
  class(${uwvec}), intent(inout) :: this
  ${type}, intent(in) :: val
  if (.not. allocated(this%arr_)) call this%resize(0)
  if (this%size_ == this%capa_) then
     call this%resize(2*this%capa_)
  end if
  this%size_ = this%size_ + 1
  this%arr_(this%size_) = val
end subroutine push_back_${uwvec}
!> push_back_array_${uwvec}: Insert elemeents of array to the tail of elements of the
↪  ${uwvec}.
subroutine push_back_array_${uwvec}(this, arr)
  class(${uwvec}), intent(inout) :: this
  ${type}, intent(in) :: arr(:)
  integer(int32) :: s
  s = size(arr)
  if (.not. allocated(this%arr_)) call this%init(s)
  if (this%size_ + s > this%capa_) then
     call this%resize(this%size_ + s)
  end if
  this%arr_(this%size_+1:this%size_+s) = arr(:)
  this%size_ = this%size_ + s
end subroutine push_back_array_${uwvec}
```

Listing 115: push_back-unwrapped_vector(fortran)

- pop_back `function pop_back` deletes the value in the end of arr_(:) of the unwrapped vector and returns it.

```
!> pop_back_${uwvec}: Delete the value in the end of arr_(:) of the ${uwvec} and return
↪  it.
${type} function pop_back_${uwvec}(this)
  class(${uwvec}), intent(inout) :: this
  pop_back_${uwvec} = this%arr_(this%size_)
  this%size_ = this%size_ - 1
end function pop_back_${uwvec}
```

Listing 116: pop_back-unwrapped_vector(fortran)

- back `function back` returns the value in the end of arr_(:) of the unwrapped vector.

```
!> back_${uwvec}: Delete the value in the end of arr_(:) of the ${uwvec} and return it.
${type} function back_${uwvec}(this)
  class(${uwvec}), intent(inout) :: this
  back_${uwvec} = this%arr_(this%size_)
end function back_${uwvec}
```

Listing 117: back-unwrapped_vector(fortran)

- size `function size` return current size of the unwrapped vector.

```
!> size_vector_${suffix}: Return current size of the ${uwvec}.
pure integer(int32) function size_${uwvec}(this)
  class(${uwvec}), intent(in) :: this
  size_${uwvec} = this%size_
end function size_${uwvec}
```

Listing 118: size-unwrapped_vector(fortran)

- resize `subroutine resize` shrinks or expands arr_(:) of the unwrapped vector.

```
!> resize_${uwvec}: Shrink or expand arr_(:) of the ${uwvec}.
subroutine resize_${uwvec}(this, resize)
  class(${uwvec}), intent(inout) :: this
  integer(int32), intent(in) :: resize
  ${type}, allocatable :: tmp(:)
  if (resize < 1) then
     this%size_ = 0
     allocate(tmp(1))
     call move_alloc(from = tmp, to = this%arr_)
     this%capa_ = 1
  else
     if (this%capa_ == resize) return
     allocate(tmp(resize))
     this%size_ = min(this%size_, resize)
     tmp(1:this%size_) = this%arr_(1:this%size_)
     call move_alloc(from = tmp, to = this%arr_)
     this%capa_ = resize
  end if
end subroutine resize_${uwvec}
```

Listing 119: resize-unwrapped_vector(fortran)

- lower_bound `function lower_bound` returns the minimum index that is higher than or equal to ‘val‘.

```
!> lower_bound_vector_${suffix}: Return the minimum index that is higher than or equal
↪  to `val`.
integer(int32) function lower_bound_${uwvec}(this, val)
  class(${uwvec}), intent(in) :: this
  ${type}, intent(in) :: val
  integer(int32) :: p, q, r
  p = 1
  r = this%size_
  if (this%arr_(r) < val) then
     lower_bound_${uwvec} = r + 1
     return
  end if
  do
     q = (p+r)/2
     if (p + 1 > r) exit
     if (this%arr_(q) >= val) then
        r = q
     else
        p = q+1
     end if
  end do
  lower_bound_${uwvec} = q
end function lower_bound_${uwvec}
```

Listing 120: lower_bonud-unwrapped_vector(fortran)

```
case "${type}" in
    "character")
        type="character"
        suffix="character"
        ;;
    *)
        type="${type}(${type_kind})"
        suffix="${type_kind}"
        ;;
esac
uwvec="unwrapped_vector_${suffix}"
```

Listing 121: unwrapped_vector-var(bash)

```
<<unwrapped_vector-var>>
cat <<EOF
<<declaration-unwrapped_vector>>
EOF
```

Listing 122: declaration-unwrapped_vector-var(bash)

```
<<unwrapped_vector-var>>
cat <<EOF
<<procedures-unwrapped_vector>>
EOF
```

Listing 123: procedures-unwrapped_vector-var(bash)

■7.4.1.4 process definition and procedures of the vector

```
program test_unwrapped_vector
  use, intrinsic :: iso_fortran_env
  use unwrapped_vector_m
  implicit none
  integer(int32) :: i, j
  integer(int32) :: ierr
  integer(int32), parameter :: n = 10, low = 5, high = low+n-1
  type(unwrapped_vector_int32) :: v, v2
  store:do i = 1, n
     call v%push_back(i)
     <<assert(cond="v%arr_(i) == i", code=10, message="Stored value in `v%arr_(i)` is illegal in loop.")>>
  end do store
  test_lower_bound:do i = 0, v%size()+1
     j = v%lower_bound(i)
     <<assert(cond="j == max(1, i)", code=11, message="Return value of `lower_bound` is illegal in loop.")>>
  end do test_lower_bound
  do i = 1, n
     j = v%pop_back()
  end do

  v2 = unwrapped_vector_int32(5)
  v2%arr_(:) = 1
  do i = 1, 5
     <<assert(cond="v2%arr_(i) == 1", code=20, message="Initialization by size of `v2` is illegal.")>>
  end do
  v2 = unwrapped_vector_int32([(i, i = 1,5)])
  do i = 1, 5
     <<assert(cond="v2%arr_(i) == i", code=21, message="Initialization by array of `v2` is illegal.")>>
  end do
  v2 = unwrapped_vector_int32(size = 5, val = 2)
  do i = 1, 5
     <<assert(cond="v2%arr_(i) == 2", code=22, message="Initialization by init_val of `v2` is illegal.")>>
  end do

  call v2%resize(0)
  do i = 1, 5
     call v2%push_back(i)
     <<assert(cond="v2%back() == i", code=23, message="Resize or back for `v2` is illegal.")>>
```

```
<<unwrapped_vector-module>>
<<unwrapped_vector-test>>
```

Listing 125: test-unwrapped_vector(fortran)

■7.4.1.5 test

7.5 queue(未完成)

7.5.1 imp

```
module queue_m
  use, intrinsic :: iso_fortran_env
  use unwrapped_vector_m
  implicit none
  private
  public :: queue
  type :: queue
     private
     integer(int32) :: head_, tail_
     type(unwrapped_vector_int32) :: q_
   contains
     procedure, pass :: init => init_queue
     procedure, pass :: push_back => push_back_queue
     procedure, pass :: pop_front => pop_front_queue
     procedure, pass :: size => size_front_queue
     procedure, pass :: empty => empty_front_queue
  end type queue
contains
  subroutine init_queue(this)
    class(queue), intent(inout) :: this
    this%head_ = 1
    this%tail_ = 0
  end subroutine init_queue
  subroutine push_back_queue(this, val)
    class(queue), intent(inout) :: this
    integer(int32), intent(in) :: val
    integer(int32) :: s
    if (this%head_ == this%q_%size()) then
       s = this%tail_ - (this%head_-1)
       this%q_%arr_(1:s) = eoshift(this%q_%arr_(:), shift = this%head_-1)
       this%tail_ = s
       this%head_ = 1
       call this%q_%resize(this%size())
    end if
    this%tail_ = this%tail_ + 1
    call this%q_%push_back(val)
  end subroutine push_back_queue
  integer(int32) function pop_front_queue(this) result(res)
    class(queue), intent(inout) :: this
    res = this%q_%arr_(this%head_)
    this%head_ = this%head_ + 1
```

7.6 priority queue

7.6.1 whole module of the priority queue

priority queue モジュール全体は以下のとおり. 型毎に noweb マクロを展開する.

```
module priority_queue_m
  use, intrinsic :: iso_fortran_env
  implicit none
  private
  <<declaration-priority_queue-var(type_base="integer", type_kind="int32")>>
  <<declaration-priority_queue-var(type_base="integer", type_kind="int64")>>
  <<declaration-priority_queue-var(type_base="real", type_kind="real32")>>
  <<declaration-priority_queue-var(type_base="real", type_kind="real64")>>
  <<declaration-priority_queue-var(type_base="character")>>
contains
  <<procedures-priority_queue-var(type_base="integer", type_kind="int32")>>
  <<procedures-priority_queue-var(type_base="integer", type_kind="int64")>>
  <<procedures-priority_queue-var(type_base="real", type_kind="real32")>>
  <<procedures-priority_queue-var(type_base="real", type_kind="real64")>>
  <<procedures-priority_queue-var(type_base="character")>>
end module priority_queue_m
```

Listing 127: priority_queue-module(fortran)

7.6.2 declaration of the priority queue

まず, `priority_queue` 型を宣言する. 配列を伸長させて優先度付きキューを実装する. 挿入と削除操作は `O(log n)` で, 先頭の参照は `O(1)` で可能.

`bash` を用いて `${variable}` を展開する.

- `${pq}` は優先度付きキューの型の名前である.
- `${type}` は優先度付きキューの要素の型の名前である.
- `${op}` は ‘<‘ か ‘>‘.

```
public :: ${pq}
type :: ${pq}
   private
   integer(int32) :: size_ = 0_int32, capa_ = 0_int32
   ${type}, allocatable :: arr_(:)
 contains
   procedure, pass :: push  => push_${pq}
   procedure, pass :: pop   => pop_${pq}
   procedure, pass :: front => front_${pq}
   procedure, pass :: size => size_${pq}
   ! procedure, pass :: dump => dump_${pq}
end type ${pq}
```

Listing 128: declaration-priority_queue(fortran)

7.6.3 procedures of the priority queue

■7.6.3.1 push push はヒープへ要素を追加し, ヒープを再構成する.

```fortran
!> push_${pq}: adds an element to the heap and reconstructs the heap by ${op} order.
subroutine push_${pq}(this, val)
  class(${pq}), intent(inout) :: this
  ${type}, intent(in) :: val
  ${type} :: tmp
  integer(int32) :: i
  if (this%size_ == this%capa_) then
     if (this%capa_ == 0) then
        this%capa_ = 1
        allocate(this%arr_(1))
     else
        this%capa_ = 2*this%capa_
        block
          ${type}, allocatable :: tmp_arr(:)
          allocate(tmp_arr(this%capa_))
          tmp_arr(1:this%size_) = this%arr_(1:this%size_)
          call move_alloc(from = tmp_arr, to = this%arr_)
        end block
     end if
  end if
  this%size_ = this%size_ + 1
  ! add `val` to heap.
  this%arr_(this%size_) = val
  i = this%size_
  tmp = val
  upheap:do ! reconstruct the heap by ${op}.
     if (i == 1) then ! top of the heap
        this%arr_(1) = tmp
        exit
     else if (tmp ${op} this%arr_(i/2)) then ! move the element up in the heap
        this%arr_(i) = this%arr_(i/2)
     else ! move the element up in the heap
        this%arr_(i) = tmp
        tmp = this%arr_(i/2)
     end if
     i = i / 2
  end do upheap
end subroutine push_${pq}
```

Listing 129: push_priority_queue(fortran)

■7.6.3.2　pop　pop はヒープへ要素を追加し, ヒープを再構成する.

```
!> pop_${pq}: extracts the ${op} element from the heap.
${type} function pop_${pq}(this) result(res)
  class(${pq}), intent(inout) :: this
  integer(int32) :: n, prev, next
  ! add `val` to heap.
  ! swap `arr(1)` and `arr(n)` and delete ${op} element, `arr(1)`.
  res = this%arr_(1)
  this%arr_(1) = this%arr_(this%size_)
  this%size_ = this%size_ - 1
  n = this%size_
  ! reconstruct the heap by moving the element `arr(n)` downwards.
  next = 1
  downheap:do ! reconstruct the heap by ${op}.
     prev = next
     if (2*prev > n) exit
     if (this%arr_(2*prev)      ${op} this%arr_(next)) &
          next = 2*prev
     if (2*prev+1 <= n) then
        if (this%arr_(2*prev+1) ${op} this%arr_(next)) &
             next = 2*prev+1
     end if
     if (prev == next) exit ! arr(next) < arr(2*prev) .and. arr(next) < arr(2*prev+1)
     call swap(this%arr_(prev), this%arr_(next))
  end do downheap
contains
  subroutine swap(x, y)
    ${type}, intent(inout) :: x, y
    ${type} :: tmp
    tmp = x
    x   = y
    y   = tmp
  end subroutine swap
end function pop_${pq}
```

Listing 130: pop_priority_queue(fortran)

■7.6.3.3　front　front は ヒープの一番上 (minimum or maximum) の要素を返す.

```
!> front_${pq}: returns the top of the element of the heap, which has either the minimum
↪  or maximum value depending on the type of heap.
pure ${type} function front_${pq}(this) result(res)
  class(${pq}), intent(in) :: this
  res = this%arr_(1)
end function front_${pq}
```

Listing 131: front_priority_queue(fortran)

■7.6.3.4　size　size はヒープの要素数を返す.

```
!> size_${pq}: returns the size of the heap.
pure integer(int32) function size_${pq}(this) result(res)
  class(${pq}), intent(in) :: this
  res = this%size_
end function size_${pq}
```

Listing 132: size_priority_queue(fortran)

■7.6.3.5　dump　dump

```
! !> dump_${pq}: output the heap.
! subroutine dump_${pq}(this)
!   class(${pq}), intent(in) :: this
!   write(error_unit, '(*(g0, 1x))') this%arr_(1:this%size_)
! end subroutine dump_${pq}
```

Listing 133: dump_priority_queue(fortran)

7.6.4 process definition and procedures of the priority queue

```
case "${type_base}" in
    "character")
        type="${type_base}"
        suffix="${type_base}"
        ;;
    "type")
        type="type(${type_kind})"
        suffix="${type_kind}"
        ;;
    *)
        type="${type_base}(${type_kind})"
        suffix="${type_kind}"
        ;;
esac
pq="priority_queue"
case "${op}" in
    "<")
        pq="${pq}_min_${suffix}"
    ;;
    ">")
        pq="${pq}_max_${suffix}"
    ;;
esac
```

Listing 134: priority_queue-var(bash)

```
for op in "<" ">"
do
<<priority_queue-var>>
cat <<EOF
<<declaration-priority_queue>>
EOF
done
```

Listing 135: declaration-priority_queue-var(bash)

```
for op in "<" ">"
do
<<priority_queue-var>>
cat <<EOF
<<procedures-priority_queue>>
EOF
done
```

Listing 136: procedures-priority_queue-var(bash)

7.6.5 test

```fortran
program test_priority_queue
  use, intrinsic :: iso_fortran_env
  use priority_queue_m
  implicit none
  integer(int32), parameter :: n = 20, arr(n) = [10, 1, 11, 2, 12, 3, 13, 4, 14, 5, 15, 6, 16, 7, 17, 8, 18, 9, 19, 20]
  integer(int32) :: i
  type(priority_queue_min_int32) :: pq_min
  type(priority_queue_max_int32) :: pq_max
  do i = 1, n
     call pq_min%push(arr(i))
     call pq_max%push(arr(i))
  end do
  <<assert-eq(eq1="n", eq2="pq_min%size()", code=10, message="The size of pq_min is illegal.")>>
  <<assert-eq(eq1="n", eq2="pq_max%size()", code=11, message="The size of pq_max is illegal.")>>
  do i = 1, n
     block
       integer(int32) :: val
       val = pq_min%pop()
       <<assert-eq(eq1="i", eq2="val", code=12, message="The value of pq_min%pop() is illegal.")>>
       val = pq_max%pop()
       <<assert-eq(eq1="n-i+1", eq2="val", code=13, message="The value of pq_max%pop() is illegal.")>>
     end block
  end do
end program test_priority_queue
```

Listing 137: priority_queue-test(fortran)

```
<<priority_queue-module>>
<<priority_queue-test>>
```

Listing 138: test-priority_queue(fortran)

7.7 double ended queue

```
module vec_deque_m
  use, intrinsic :: iso_fortran_env
  implicit none
  integer(int32), parameter :: init_size = 4
  type :: vec_dequeue
     private
     integer(int32) :: size_ = 0, capa_ = 0
     integer(int32) :: head_, tail_
     integer(int32), allocatable :: arr_(:)
   contains
     procedure, pass :: init => init_vec_dequeue
     procedure, pass :: resize => resize_vec_dequeue
     procedure, pass :: push_front => push_front_vec_dequeue
     procedure, pass :: push_back  => push_back_vec_dequeue
     procedure, pass :: pop_front  => pop_front_vec_dequeue
     procedure, pass :: pop_back   => pop_back_vec_dequeue
     procedure, pass :: to_array => to_array_vec_dequeue
     procedure, pass :: debug_print => debug_print_vec_dequeue
  end type vec_dequeue
contains
  subroutine init_vec_dequeue(this)
    class(vec_dequeue), intent(inout) :: this
    if (allocated(this%arr_)) return
    allocate(this%arr_(init_size))
    this%size_ = 0
    this%capa_ = init_size
    this%head_ = this%capa_
    this%tail_ = 1
  end subroutine init_vec_dequeue
  subroutine resize_vec_dequeue(this, capa)
    class(vec_dequeue), intent(inout) :: this
    integer(int32), intent(in) :: capa
    integer(int32) :: s
    integer(int32), allocatable :: tmp(:)
    if (capa <= this%size_) return
    allocate(tmp(capa))
    if (this%head_ < this%tail_) then
       !> (1???h...t???c), ... が意味のあるデータ, ???が意味のないデータ.
       tmp(this%head_+1:this%tail_-1) = this%arr_(this%head_+1:this%tail_-1)
       call move_alloc(from = tmp, to = this%arr_)
    else !> this%head_ >= this%tail_
       !> (1...t????h...c).
       tmp(1:this%tail_-1) = this%arr_(1:this%tail_-1)
```

7.8 Hash table

7.8.1 open addressing hash by double hash

We implement open addressing hash table that use double hash by `Fortran`. The size of hash table is m. We cannot store the number of elements that is over m.

■7.8.1.1 The whole module of the hash table

This is the whole module. The element of hash table is below.

```fortran
module hash_table_m
  use, intrinsic :: iso_fortran_env
  implicit none
  private
  integer, parameter :: max_elem = 701, small_m = 700, cardinal = 128
  public :: size

  type :: variable_char
     character(len=:), allocatable :: s
  end type variable_char
  <<expand-declaration-hash_table(type="integer", type_kind="int32")>>
  <<expand-declaration-hash_table(type="integer", type_kind="int64")>>
  <<expand-declaration-hash_table(type="real", type_kind="real32")>>
  <<expand-declaration-hash_table(type="real", type_kind="real64")>>
contains

  pure integer(int32) function hash1(key)
    character(len=*), intent(in) :: key
    integer(int32) :: h, i
    h = 0_int32
    do i = len(key), 1, -1
       h = mod(h * cardinal + ichar(key(i:i)), max_elem)
    end do
    hash1 = h
  end function hash1
  pure integer(int32) function hash2(key)
    character(len=*), intent(in) :: key
    integer(int32) :: h, i
    h = 0_int32
    do i = len(key), 1, -1
       h = mod(h * cardinal + ichar(key(i:i)), small_m)
    end do
    hash2 = h + 1
  end function hash2

  <<expand-procedures-hash_table(type="integer", type_kind="int32")>>
  <<expand-procedures-hash_table(type="integer", type_kind="int64")>>
  <<expand-procedures-hash_table(type="real", type_kind="real32")>>
  <<expand-procedures-hash_table(type="real", type_kind="real64")>>
end module hash_table_m
```


Listing 140: hash_table-module(fortran)

```
public :: ${hash_table}
type :: ${hash_table}
   private
   integer(int32) :: num_elems_
   type(variable_char), allocatable :: keys_(:)
   ${type_elements}, allocatable :: elems_(:)
   logical, allocatable :: vacancy_(:)
   logical, allocatable :: deleted_(:)
 contains
   procedure, pass :: insert => insert_${hash_table}
   procedure, pass :: delete => delete_${hash_table}
   procedure, pass :: search => search_${hash_table}
   procedure, pass :: write_${hash_table}
   generic :: write(formatted) => write_${hash_table}
end type ${hash_table}

interface ${hash_table}
   module procedure :: init_${hash_table}
end interface ${hash_table}
interface size
   module procedure :: size_${hash_table}
end interface
```

Listing 141: declaration-hash_table(fortran)

■7.8.1.2 The type declaration of the hash table

```
<<init-hash_table>>
<<size-hash_table>>
<<insert-hash_table>>
<<delete-hash_table>>
<<search-hash_table>>
<<write-hash_table>>
```

Listing 142: procedures-hash_table(fortran)

■7.8.1.3 The procedures of the hash table

- initialize

```
impure type(${hash_table}) function init_${hash_table}() result(res)
  res%num_elems_ = 0
  allocate(res%elems_(0:max_elem-1))
  allocate(res%keys_(0:max_elem-1))
  allocate(res%vacancy_(0:max_elem-1), source = .true.)
  allocate(res%deleted_(0:max_elem-1), source = .false.)
end function init_${hash_table}
```

Listing 143: init-hash_table(fortran)

- size

```
pure integer(int32) function size_${hash_table}(ht) result(res)
  type(${hash_table}), intent(in) :: ht
  res = ht%num_elems_
end function size_${hash_table}
```

Listing 144: size-hash_table(fortran)

- insert-hash_table Insert `val` into hash table. If `key` is already in the hash table, change to new `val` corresponding to `key`.

```
subroutine insert_${hash_table} (this, key, val, ierr)
  class(${hash_table}), intent(inout) :: this
  character(len=*), intent(in) :: key
  ${type_elements}, intent(in) :: val
  integer(int32), optional, intent(out) :: ierr
  integer(int32) :: h1, h2, pos, i
  <<error-handing-initialize-ierr-hash_table>>
  h1 = hash1(key)
  h2 = hash2(key)
  pos = h1
  do i = 1, max_elem
     if (this%vacancy_(pos)) then
        this%keys_(pos)%s = key
        this%elems_(pos) = val
        this%vacancy_(pos) = .false.
        this%num_elems_ = this%num_elems_ + 1
        return
     else if (this%keys_(pos)%s == key) then
        this%elems_(pos) = val
        return
     end if
     pos = mod(pos + h2, max_elem)
  end do
  <<error-handing-capacity-over-hash_table>>
end subroutine insert_${hash_table}
```

Listing 145: insert-hash_table(fortran)

```
if (present(ierr)) ierr = 0
```

Listing 146: error-handing-initialize-ierr-hash_table(fortran)

```fortran
write(error_unit, '(a)') "Size limit: Hash table is too large."
write(error_unit, '(a, i0)') __FILE__//": ", __LINE__
if (present(ierr)) then
   ierr = 1
else
   error stop 1
end if
```

Listing 147: error-handing-capacity-over-hash_table(fortran)

- delete-hash_table

```fortran
subroutine delete_${hash_table} (this, key, found)
  class(${hash_table}), intent(inout) :: this
  character(len=*), intent(in) :: key
  logical, optional, intent(out) :: found
  integer(int32) :: h1, h2, pos, i
  h1 = hash1(key)
  h2 = hash2(key)
  pos = h1
  do i = 1, max_elem
     if (this%vacancy_(pos) .and. (.not. this%deleted_(pos))) exit
     if (this%keys_(pos)%s == key) then
        this%vacancy_(pos) = .true.
        this%deleted_(pos) = .true.
        this%num_elems_ = this%num_elems_ - 1
        if (present(found)) found = .true.
        return
     end if
     pos = mod(pos + h2, max_elem)
  end do
  if (present(found)) found = .false.
end subroutine delete_${hash_table}
```

Listing 148: delete-hash_table(fortran)

- search-hash_table

```
${type_elements} function search_${hash_table} (this, key, found) result(res)
  class(${hash_table}), intent(in) :: this
  character(len=*), intent(in) :: key
  logical, optional, intent(out) :: found
  integer(int32) :: h1, h2, pos, i
  res = -1
  h1 = hash1(key)
  h2 = hash2(key)
  pos = h1
  do i = 1, max_elem
     if (this%vacancy_(pos) .and. (.not. this%deleted_(pos))) exit
     if (this%keys_(pos)%s == key) then
        res = this%elems_(pos)
        if (present(found)) found = .true.
        return
     end if
     pos = mod(pos + h2, max_elem)
  end do
  if (present(found)) found = .false.
end function search_${hash_table}
```

Listing 149: search-hash_table(fortran)

- write-hash_table

```
subroutine write_${hash_table}(this, unit, iotype, v_list, iostat, iomsg)
  class(${hash_table}), intent(in) :: this
  integer             , intent(in)    :: unit
  character(len=*)    , intent(in)    :: iotype
  integer             , intent(in)    :: v_list(:)
  integer             , intent(out)   :: iostat
  character(len=*)    , intent(inout) :: iomsg
  integer(int32) :: i
  do i = 0, max_elem-1
     if (.not. this%vacancy_(i)) then
        write(unit, fmt='(a, i0, a, g18.10)', advance = "No", iostat=iostat,
↪  iomsg=iomsg) &
             "|", i, ": ht["//this%keys_(i)%s//"] => ", this%elems_(i)
     end if
  end do
end subroutine write_${hash_table}
```

Listing 150: write-hash_table(fortran)

```
 case "${type}" in
    "character")
        type_elements="type(variable_char)"
        type_val="character(len=:), allocatable"
        hash_table="hash_table_character"
        ;;
    *)
        type_elements="${type}(${type_kind})"
        type_val="${type_elements}"
        hash_table="hash_table_${type_kind}"
        ;;
esac
```

Listing 151: expand-hash_table(bash)

```
<<expand-hash_table>>
cat <<EOF
<<declaration-hash_table>>
EOF
```

Listing 152: expand-declaration-hash_table(bash)

```
<<expand-hash_table>>
cat <<EOF
<<procedures-hash_table>>
EOF
```

Listing 153: expand-procedures-hash_table(bash)

■7.8.1.4 process definition and procedures of hash table

```
program test_hash_table
  use, intrinsic :: iso_fortran_env
  use hash_table_m
  implicit none
  integer(int32) :: v, i, j, k, ierr
  logical :: found
  character(len=:), allocatable :: s
  type(hash_table_int32) :: ht_i32, ht_i32_2
  ht_i32 = hash_table_int32()
  !> check empty character.
  call ht_i32%insert("", 0, ierr=ierr)
  v = ht_i32%search("", found=found)
  if (.not. found) then
     write(error_unit, *) "Empty string '' not found or not inserted..."
     error stop 2
  end if
  if (v /= 0) then
     write(error_unit, *) "Value of arr[''] must be 0"
     error stop 3
  end if
  !> check size
  !> insert 701 elements
  !> first, insert 10*10*7 elements
  do i = ichar("a"), ichar("a")+10-1
     do j = ichar("A"), ichar("A")+10-1
        do k = ichar(" "), ichar(" ")+7-1
           s = achar(i)//achar(j)//achar(k)
           call ht_i32%insert(s, 128**2*i+128*j+k, ierr)
        end do
     end do
  end do
  call ht_i32%insert("abcde", 0, ierr) ! size of hash table is maximum
  if (ierr == 0) then
     write(error_unit, *) "Insert in fully hash table must fail...", size(ht_i32)
     error stop 4
  end if
  call ht_i32%delete("aB$", found) ! delete elements in hash table.
  if (.not. found) then
     write(error_unit, *) "Delete failed...", size(ht_i32)
     error stop 5
  end if
  call ht_i32%insert("abcdef", 0, ierr) ! be able to insert
  if (ierr /= 0) then
     write(error_unit, *) "Delete or insert failed..."
```

```
<<hash_table-module>>
<<hash_table-test>>
```

Listing 155: test-hash_table(fortran)

■7.8.1.5 test

7.9 B-Tree

7.9.1 B 木

Rust に習って B 木を実装する. $t = 6$ でノード内の内部ノードの数は $2t - 1 = 11$ とする.

```
module btree_m
  use, intrinsic :: iso_fortran_env
  implicit none
  private
  !> `t-1` must be the least number of elements in `btree_node` without root (minimum
  ↪  degree).
  integer(int32), parameter :: t = 6
  !> the number of internal node in `btree_node`.
  integer(int32), parameter :: inode = 2*t-1
  integer(int32), parameter :: iter_max_depth = 30

  !> pointer to btree_node.
  type :: btree_node_ptr
     type(btree_node), pointer :: p_ => null()
   contains
     procedure, pass :: size => size_btree_node_ptr
     procedure, pass :: is_leaf => is_leaf_btree_node_ptr
     procedure, pass :: get => get_btree_node_ptr
     procedure, pass :: split_child => split_child_btree_node_ptr
     procedure, pass :: insert => insert_btree_node_ptr
     procedure, pass :: remove => remove_btree_node_ptr
     procedure, pass :: shrink_left  => shrink_left_btree_node_ptr
     procedure, pass :: expand_right => expand_right_btree_node_ptr
     procedure, pass :: print => print_btree_node_ptr
     procedure, pass :: check_invariant => check_invariant_btree_node_ptr
  end type btree_node_ptr
  !> node of B-Tree.
  type :: btree_node
     integer(int32) :: nelem_ = 0
     integer(int32) :: key_(inode)
     integer(int32) :: val_(inode)
     type(btree_node_ptr) :: children_(inode+1)
     logical :: is_leaf_ = .true.
  end type btree_node

  public :: btree
  !> `btree` has pointer to root of B-Tree.
  type :: btree
     private
     type(btree_node_ptr) :: root_
     integer(int32) :: size_   = 0
     integer(int32) :: height_ = 0
   contains
     procedure, pass :: size   => size_btree
     procedure, pass :: height => height_btree
     procedure, pass :: init   => init_btree
```

7.10 modint

7.10.1 type declaration

```
module modint_m
  use, intrinsic :: iso_fortran_env
  implicit none
  private
  integer(int64), parameter :: modulo = 10**9 + 7
  public :: modint
  public :: assignment(=), operator(+), operator(-), operator(*), operator(/), inv,
↪  operator(**), combination
  type :: modint
     integer(int64) :: val_
   contains
     procedure, pass :: to_i64 => to_i64_modint
  end type modint
  interface modint
     module procedure :: init_modint_i32, init_modint_i64
  end interface modint
  interface assignment(=)
     module procedure :: assign_m_from_m, assign_m_from_i32, assign_m_from_i64
  end interface assignment(=)
  interface operator(+)
     module procedure :: add_m_m, add_i32_m, add_i64_m, add_m_i32, add_m_i64
  end interface operator(+)
  interface operator(-)
     module procedure :: sub_m_m, sub_i32_m, sub_i64_m, sub_m_i32, sub_m_i64
  end interface operator(-)
  interface operator(*)
     module procedure :: mul_m_m, mul_i32_m, mul_i64_m, mul_m_i32, mul_m_i64
  end interface operator(*)
  interface inv
     module procedure :: inv_modint, inv_i32, inv_i64
  end interface inv
  interface operator(/)
     module procedure :: div_m_m, div_i32_m, div_i64_m, div_m_i32, div_m_i64
  end interface operator(/)
  interface operator(**)
     module procedure :: pow_m_i32, pow_m_i64
  end interface operator(**)
  interface combination
     module procedure :: combination_m_m, combination_m_i32, combination_m_i64,
↪  combination_i32_m, combination_i64_m
  end interface combination
```

7.10.2 procedures

7.11 Binary Indexed Tree(BIT)

7.11.1 BIT モジュールの全容

i32, i64, r32, r64 でユーザ定義型とその実装を定義. BIT は n 個の値の区間 [1, r] までの部分和を O(\log n) で求めることができる. 更新も O(\log n) である.

```
module binary_indexed_tree_m
  use, intrinsic :: iso_fortran_env
  implicit none
  private
  <<declaration-binary_indexed_tree-var(type="integer", type_kind="int32")>>
  <<declaration-binary_indexed_tree-var(type="integer", type_kind="int64")>>
  <<declaration-binary_indexed_tree-var(type="real", type_kind="real32")>>
  <<declaration-binary_indexed_tree-var(type="real", type_kind="real64")>>
contains
  <<procedures-binary_indexed_tree-var(type="integer", type_kind="int32")>>
  <<procedures-binary_indexed_tree-var(type="integer", type_kind="int64")>>
  <<procedures-binary_indexed_tree-var(type="real", type_kind="real32")>>
  <<procedures-binary_indexed_tree-var(type="real", type_kind="real64")>>
end module binary_indexed_tree_m
```

Listing 158: binary_indexed_tree-module(fortran)

7.11.2 BIT 型の宣言

まず, BIT 型の宣言をする. BIT 型の振舞いとして

- subroutine init(n) でサイズ n , 値 0 で初期化する.
- subroutine init(arr(:)) で配列 arr(:) で初期化する.
- subroutine reset() でサイズを変えずに値を 0 にする.
- i32 size() でサイズを返す.
- subroutine add(i, v) でインデックス i へ v を加算する.
- ${type} sum1(r) で 閉区間 [1, r] の和を返す.
- ${type} sum_range(l, r) で 閉区間 [l, r] の和を返す.
- i32 lower_bound(w) は和が w 以上になる最小のインデックスを返す.

- デコンストラクタ destroy_${BIT} は BIT 型の配列を開放する.

とする. 配列のインデックスは Fortran らしく, 1 始まりとする.

bash 変数を後で展開して, ソースコードを生み出す. bash 変数一覧.

- ${BIT} は BIT 型の名前, binary_indexed_tree_int32 など.
- ${type} は配列の要素の型, integer(int64) など.
- ${zero} は ${type} 型での 0, 0.0_real32 など.

```
public :: ${BIT}
!> ${BIT}: can calculate the range sum O(log n).
type :: ${BIT}
   private
   ${type}, allocatable :: arr_(:)
   integer(int32) :: size_ = 0
 contains
   procedure, pass :: init_${BIT}_by_size, init_${BIT}_by_arr
   generic         :: init  => init_${BIT}_by_size, init_${BIT}_by_arr
   procedure, pass :: reset => reset_${BIT}
   procedure, pass :: size  => size_${BIT}
   procedure, pass :: add       => add_${BIT}
   procedure, pass :: sum1      => sum1_${BIT}
   procedure, pass :: sum_range => sum_range_${BIT}
   procedure, pass :: lower_bound => lower_bound_${BIT}
   final :: destroy_${BIT}
end type ${BIT}
```

Listing 159: declaration-binary_indexed_tree(fortran)

7.11.3 BIT の procedures

- init subroutine init は BIT を初期化する. BIT のサイズ n を渡すと, 全ての要素が 0 の BIT ができる. BIT へ配列 arr(:) を渡すと, その配列の BIT ができる.

```
!> init_${BIT}_by_size: Initialize the ${BIT} by size.
!> All elements of ${BIT} is ${zero}.
subroutine init_${BIT}_by_size(this, n)
  class(${BIT}), intent(inout) :: this
  integer(int32), intent(in) :: n
  !> Error exist if already allocated.
  if (allocated(this%arr_)) then
     <<error-handling-filename>>
     <<error-handling-error_message-exit(err_num=1,string="This ${BIT} is already
↪  allocated.")>>
  end if
  allocate(this%arr_(n), source = ${zero})
  this%size_ = n
end subroutine init_${BIT}_by_size

!> init_${BIT}_by_arr: Initialize the ${BIT} by array.
subroutine init_${BIT}_by_arr(this, arr)
  class(${BIT}), intent(inout) :: this
  ${type}, intent(in) ::arr(:)
  integer(int32) :: i, arr_size
  arr_size = size(arr)
  call this%init(arr_size)
  do i = 1, arr_size
     call this%add(i, arr(i))
  end do
end subroutine init_${BIT}_by_arr
```

Listing 160: init-binary_indexed-tree(fortran)

- reset `subroutine reset` は BIT の配列を全て `0` にする.

```
!> reset_${BIT}: Replace `this%arr_(:)` with `0`.
subroutine reset_${BIT}(this)
  class(${BIT}), intent(inout) :: this
  if (allocated(this%arr_)) then
     this%arr_(:) = ${zero}
  end if
end subroutine reset_${BIT}
```

Listing 161: reset-binary_indexed-tree(fortran)

- size `function size` はサイズを返す.

```
!> size_${BIT}: Return current size of the ${BIT}.
pure integer(int32) function size_${BIT}(this) result(res)
  class(${BIT}), intent(in) :: this
  res = this%size_
end function size_${BIT}
```

Listing 162: size-binary_indexed_tree(fortran)

- add `subroutine add` は配列 `arr(:)` の `idx` 番目に `val` を足すことと同義である.

```
!> add_${BIT}: Add the value `val` into the index `idx` of `arr(:)`.
subroutine add_${BIT}(this, idx, val)
  class(${BIT}), intent(inout) :: this
  integer(int32), intent(in) :: idx
  ${type}, intent(in) :: val
  integer(int32) :: i
  i = idx
  do
     if (i > this%size_) exit
     this%arr_(i) = this%arr_(i) + val
     i = i + iand(i, -i)
  end do
end subroutine add_${BIT}
```

Listing 163: add-binary_indexed_tree(fortran)

- sum1 `function sum1` は 閉区間 `[1, r]` の和を返す.

```
!> sum1_${BIT}: Return the summation of `arr(1:r)`.
!> Return ${zero} if r < 0.
${type} function sum1_${BIT}(this, r) result(res)
  class(${BIT}), intent(in) :: this
  integer(int32), intent(in) :: r
  integer(int32) :: i
  res = ${zero}
  i = r
  do
     if (i < 1) return
     res = res + this%arr_(i)
     i = i - iand(i, -i)
  end do
end function sum1_${BIT}
```

Listing 164: sum-binary_indexed_tree(fortran)

- sum_range `=function sum_range` は 閉区間 `[l, r]` の和を返す.

```
!> sum_range_${BIT}: Return the summation of `arr(l:r)`
!> Returun ${zero} if r < l.
${type} function sum_range_${BIT}(this, l, r) result(res)
  class(${BIT}), intent(in) :: this
  integer(int32), intent(in) :: l, r
  res = ${zero}
  if (r < l) return
  res = this%sum1(r) - this%sum1(l-1)
end function sum_range_${BIT}
```

Listing 165: sum_range-binary_indexed_tree(fortran)

- lower_bound `function lower_bound` は 和が `w` になるような最小のインデックスを返す.

```
!> lower_bound_${BIT}: Return the minimum index, which `x1 + x2 + ... + xres >= w`.
!> Returun 0 if w <= ${zero}.
integer(int32) function lower_bound_${BIT}(this, w) result(res)
  class(${BIT}), intent(in) :: this
  ${type}, intent(in) :: w
  ${type} :: w_tmp
  integer(int32) :: x, r, l
  if (w <= ${zero}) then
     res = 0_int32
     return
  end if
  w_tmp = w
  x = 0
  r = 1
  do while (r < this%size_)
     r = ishft(r, 1)
  end do
  l = r
  do while (l > 0)
     if (x + l <= this%size_) then
        if (this%arr_(x+l) < w_tmp) then
           w_tmp = w_tmp - this%arr_(x+l)
           x = x + l
        end if
     end if
     l = ishft(l, -1)
  end do
  res = x + 1
end function lower_bound_${BIT}
```

Listing 166: lower_bound-binary_indexed_tree(fortran)

- final `subroutine destroy_${BIT}` は BIT の配列を開放する.

```
!> destroy_${BIT}: Replace `this%arr_(:)` with `0`.
subroutine destroy_${BIT}(this)
  type(${BIT}), intent(inout) :: this
  if (allocated(this%arr_)) then
     deallocate(this%arr_)
  end if
end subroutine destroy_${BIT}
```

Listing 167: destroy-binary_indexed-tree(fortran)

7.11.4 bash で展開

上で定義したものを NOWEB (この場合は bash) で展開する. 型を `case` 分で `Fortran` 用に処理する.

```
case "${type}" in
    "real")
        zero="0.0_${type_kind}"
        ;;
    "integer")
        zero="0_${type_kind}"
        ;;
esac
type="${type}(${type_kind})"
suffix="${type_kind}"
BIT="binary_indexed_tree_${suffix}"
```

Listing 168: binary_indexed_tree-var(bash)

```
<<binary_indexed_tree-var>>
cat <<EOF
<<declaration-binary_indexed_tree>>
EOF
```

Listing 169: declaration-binary_indexed_tree-var(bash)

```
<<binary_indexed_tree-var>>
cat <<EOF
<<procedures-binary_indexed_tree>>
EOF
```

Listing 170: procedures-binary_indexed_tree-var(bash)

7.11.5 test

```
program test_binary_indexed_tree
  use, intrinsic :: iso_fortran_env
  use binary_indexed_tree_m
  implicit none
  integer(int32), parameter :: n = 10
  call check_summation(n)
  call check_inversion(n)
  call check_kth_element(n)
contains
  subroutine check_summation(n)
    integer(int32), intent(in) :: n
    integer(int32), allocatable :: arr(:)
    integer(int32) :: i
    type(binary_indexed_tree_int32) :: bit
    allocate(arr, source = [(i, i = 1, n)])
    call bit%init(arr)
    <<assert(eq1="bit%size()", eq2="n", code=2, message="Size of `bit` is wrong.")>>
    do i = 1, n
       <<assert-eq(eq1="bit%sum1(i)", eq2="i*(i+1)/2", code=3, message="The summation
↪  of bit is wrong.")>>
    end do
  end subroutine check_summation
  subroutine check_inversion(n)
    integer(int32), intent(in) :: n
    integer(int32), allocatable :: arr(:)
    integer(int32) :: i, cnts
    type(binary_indexed_tree_int32) :: bit
    allocate(arr, source = [(i, i = n, 1, -1)])
    call bit%init(n)
    cnts = 0_int32
    do i = 1, n
       cnts = cnts + (i-1) - bit%sum1(arr(i))
       call bit%add(arr(i), 1)
       <<assert-eq(eq1="cnts", eq2="i*(i-1)/2", code=3, message="The inversion number
↪  of bit is wrong.")>>
    end do
    <<assert(eq1="bit%size()", eq2="n", code=2, message="Size of `bit` is wrong.")>>
  end subroutine check_inversion
  subroutine check_kth_element(n)
    integer(int32), intent(in) :: n
    integer(int32), allocatable :: arr(:)
    integer(int32) :: i, idx
    type(binary_indexed_tree_int32) :: bit
```

```
<<binary_indexed_tree-module>>
<<binary_indexed_tree-test>>
```

Listing 172: test-binary_indexed_tree(fortran)

7.12 segment tree

7.12.1 Implementation

```
module segment_tree_m
  use, intrinsic :: iso_fortran_env
  implicit none
  private
  public :: segment_tree
  public :: monoid_op
  public :: plus_int32_op, min_int32_op
  type :: segment_tree
     private
     integer(int32) :: arr_size_, tree_size_, depth_
     integer(int32), allocatable :: arr_(:)
     class(monoid_op), allocatable :: monoid
   contains
     procedure, pass :: init   => init_segment_tree
     procedure, pass :: dump   => dump_segment_tree
     procedure, pass :: update => update_segment_tree
     procedure, pass :: query => query_segment_tree
  end type segment_tree
  type, abstract :: monoid_op
     private
   contains
     procedure(identity_int32), nopass, deferred :: identity
     procedure(bin_op_int32)  , nopass, deferred :: bin_op
  end type monoid_op
  abstract interface
     pure integer(int32) function identity_int32() result(res)
       import int32
     end function identity_int32
     pure integer(int32) function bin_op_int32(x, y) result(res)
       import int32
       integer(int32), intent(in) :: x, y
     end function bin_op_int32
  end interface

  type, extends(monoid_op) :: plus_int32_op
     private
   contains
     procedure, nopass :: identity => identity_plus_int32_op
     procedure, nopass :: bin_op   => bin_op_plus_int32_op
  end type plus_int32_op
  type, extends(monoid_op) :: min_int32_op
     private
```