Reference number of working document: **ISO/IEC JTC1/SC22/WG5 Nxxxx**

Date: 2018-10-31

Reference number of document:

Committee identification: ISO/IEC JTC1/SC22

Secretariat: ANSI

Information technology — Programming languages — Fortran — Accessor procedures

Technologies de l'information — Langages de programmation — Fortran — Procédures d'accès aux structures de données

(Blank page)

Contents

Foreword

This technical specification specifies an extension to the computational facilities of the programming language Fortran. Fortran is specified by the International Standard ISO/IEC 1539-1:2018(E).

(Blank page)

0 Introduction

0.1 History

1 Very early after the use of software began, it was realized that the cost of a change to a program is more likely to be proportional to the size of the program than the magnitude of the change.

2 John Backus sold Fortran to IBM president Tom Watson on the proposition that it would reduce labor costs. Tom Watson sold it to IBM customers on the proposition that it would reduce labor costs.

3 Within fifteen years, several people realized that the failure of high-level programming languages to reduce the cost of a program from being proportional to the size of the program to the magnitude of the change, was the result of the details of the implementation of each data structure being exposed in the syntax used to reference the representation of the data.

4 Two fundamentally different solutions were proposed for the problem.

5 In 1970 Douglas T. Ross proposed that the same syntax ought to be used to refer to every kind of data object, and to procedures.

6 Charles M. Geschke and James G. Mitchell repeated this proposal in 1975.

7 In 1972 David Parnas proposed that this could largely be achieved almost completely by encapsulating all operations on a data structure in a family of related procedures. Thereby, the difference to reference each kind of data object, or a procedure, would be isolated into the collection of procedures that implement operations on the data.

8 No major programming language has been revised to incorporate the principles advocated by Ross, Geschke and Mitchell.

9 Rather, it has apparently been judged that the problem can be adequately solved by program authors employing the principles advocated by Parnas.

1. Charles M. Geschke and James G. Mitchell, *On the problem of uniform references to data structures*, **IEEE Transactions on Software Engineering SE-2**, 1 (June 1975) 207-210.
2. David Parnas, *On the criteria to be used in decomposing systems into modules*, **Comm. ACM 15**, 12 (December 1972) 1053-1058.
3. D. T. Ross, *Uniform referents: An essential property for a software engineering language*, in **Software Engineering 1** (J. T. Tou, Ed.), Academic Press, (1970) 91-101.

10 Some languages have provided facilities that reduced the difference of syntax between different kinds of data, and procedures. Univac FORTRAN V implemented statement functions as macros. As a consequence, a reference to a statement function was permitted as the *variable* in an assignment statement if its body would have been permitted. This was used to provide some of the functionality of derived-type objects. The "component name" appeared in the syntactic position of a function name, and the "object" appeared in the syntactic position of an argument.

11 POP-2 had procedures called *updaters* that could be invoked to receive a value in a variable-definition context.

12 Python has procedures called *setters* that can be invoked to receive a value in a variable-definition context. Python setters can only be type-bound objects.

13 This technical specification proposes an abstraction mechanism related to POP-2 updaters, and python setters, called *accessors*.

0.2 The problems to be solved

1 There are two problems with the Parnas agenda.

2 First, it is difficult and costly to apply completely and consistently. If it hasn't been applied carefully and completely during the original development of a program, the program is difficult to modify.

3 Second, it is potentially inefficient, because all operations on data structures are encapsulated within procedures. Awareness of this potential is an incentive not to use it carefully and completely.

0.3 What this technical specification proposes

1 This technical specification extends the programming language Fortran so that the representation of a data abstraction can be changed between a data object and a procedure without changing the syntax of any references to it.

2 The facility specified by this technical specification is compatible to the computational facilities of Fortran as standardized by ISO/IEC 1539-1:2018(E).

Information technology – Programming Languages – Fortran

Technical Specification: Accessors

1 General

1.1 Scope

1 This technical specification specifies an extension to the programming language Fortran. The Fortran language is specified by International Standard ISO/IEC 1539-1:2018(E) : Fortran. The extension allows the representation of a data object to be changed between an array and a procedure, or between a structure component and a procedure, without changing the syntax of references to that data object.

2 Clause 2 of this technical specification contains a general and informal but precise description of the extended functionalities. Clause 3 contains an extended example of the use of facilities described in technical specification. Clause 4 contains detailed instructions for editorial changes to ISO/IEC 1539-1:2018(E).

1.2 Normative References

1 The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

2 ISO/IEC 1539-1:2018(E) : *Information technology – Programming Languages – Fortran; Part 1: Base Language*

2 Requirements

2.1 General

1 The following subclauses contain a general description of the extensions to the syntax and semantics of the Fortran programming language to provide that the representation of a data object can be changed between an array and a procedure, or between a structure component and a procedure, without changing the syntax of references to that data object.

2.2 Summary

2.2.1 General

1 This technical specification defines a new entity called an *accessor*. An accessor defines and activation record and at least two kinds of procedures: a function and a new kind of procedure called an *updater*. An updater can be invoked in a variable definition context, in which case the value is passed to the updater. An accessor can be invoked in a data reference context, in which case its function is executed and its result is a value. It can also be invoked in a variable definition context, in which case one of its updaters is executed and the value is passed to the updater. There is presently nothing comparable in Fortran, but updaters or accessors have been provided in other languages such as Mesa, POP-2, and Python. This dual nature of invocation allows the representation of a data abstraction to be changed from a data object to function and updater procedures, without changing the syntax of references to it.

2 In addition to a function and updaters, an accessor can provide subroutines, for initialization or other problem-specific purposes.

3 The type SECTION is defined. Objects of type SECTION have the same properties as section subscripts. The constructor for type SECTION has the same syntax as a section triplet. This allows variables and procedure dummy arguments that have those properties, which in turn allows the representation of an object to be changed between a function and updater, and an array, without changing the syntax of references to it.

2.2.2 Type SECTION

1 The type SECTION is defined in the intrinsic module ISO_Fortran_Env.

2 The type SECTION has one kind type parameter and five protected components. The effect is as if it were declared using the following type declaration, which assumes existence of the PROTECTED attribute:

3
```
  type :: SECTION ( Kind )
    integer, kind :: Kind
    integer(kind), protected :: LBOUND = -huge(0_kind)
    logical, protected :: LOWER_BOUNDED
    integer(kind), protected :: UBOUND = huge(0_kind)
    logical, protected :: UPPER_BOUNDED
    integer(kind), protected :: STRIDE
  end type SECTION
```

4 The integer parts are the lower bound, the upper bound, and the stride. The logical parts indicate whether a value appeared in the type constructor, for the lower bound or the upper bound. The type SECTION is not a sequence derived type. Therefore, objects of type SECTION cannot be storage associated. A processor might represent it differently from a derived type.

2.2.3 Constructor for values of type SECTION

1 An object of type SECTION can be constructed using the same syntax as *subscript-triplet*. If the stride is not specified its value is 1.

NOTE 2.1

Although an object of type SECTION can be constructed using the same syntax as a constructor for an object of derived type, it is important that the constructor for objects of type SECTION be the same as *subscript-triplet*, not the same as a constructor for an object of derived type.

2.2.4 Constructing an array from a SECTION object

1 The type SECTION has a type-bound procedure named SECTION_AS_ARRAY that produces a rank-one integer array whose values are the same as would be denoted by an equivalent subscript triplet. It shall not be invoked if the LOWER_BOUNDED or UPPER_BOUNDED component has the value false.

2.2.5 Definition of accessor subprograms

1 A new program unit called an ACCESSOR is defined. An accessor defines an activation record, one or more functions to reference the activation record, one or more updaters to modify the activation record, and it may define subroutines to initialize the activation record, or for other problem-dependent purposes.

2 When an accessor is referenced to provide the value of a primary during evaluation of a expression, one of its functions is invoked, and the result value is provided in the same way as by a function subprogram. When an accessor is referenced in a variable definition context, one of its updaters is invoked, and the value to be defined is transferred to the updater in its acceptor variable.

2.2.6 Syntax to reference accessor procedures

1 A reference to an accessor is permitted where a reference to or definition of a variable is permitted.

NOTE 2.2

For example, an accessor reference can appear within an expression, as the *variable* in an intrinsic assignment statement, in an input/output list in either a READ or WRITE statement, in place of a *variable* in a control information list. . . .

2 An accessor is referenced using an extension of the syntax to reference a function. The extended syntax is the same as is used to reference an array or a character substring, which in turn allows the representation of an object to be changed between a function and updater, and a character scalar or an array, without changing the syntax of references to it.

3 Where a reference appears in a value reference context, an accessor function is invoked to produce a value. Where it appears in a variable definition context, an accessor updater is invoked to accept a value. Where it appears as an actual argument associated with a dummy argument with INTENT(IN), an accessor function is invoked to produce a value before the procedure to which it is an actual argument is invoked. Where it appears as an actual argument associated with a dummy argument with INTENT(OUT), an accessor updater is invoked to accept a value after the invoked procedure completes execution. Where it appears as an actual argument associated with a dummy argument with INTENT(INOUT) or unspecified intent, an accessor function function is invoked to produce a value before the procedure to which it is an actual argument is invoked, and an accessor updater is invoked to accept a value after the invoked procedure completes execution.

4 An accessor function or updater is not required to have nonoptional dummy arguments. Unlike the syntax

for functions, where it is referenced without actual arguments, it need not include empty parentheses. This permits an accessor and scalar variable to be interchanged.

5 The result and acceptor variables of accessor functions and updaters cannot be procedure pointers. Therefore,

- where a procedure is referenced with an instance variable as an actual argument that corresponds to an instance variable dummy argument, the instance variable is the argument; the accessor is not invoked to produce or receive a value corresponding to the dummy argument, and
- where an instance variable appears as the *instance-target* in a pointer assignment statement, the function part of the accessor is not invoked.

NOTE 2.3

Where an accessor reference appears as an actual argument, a processor might use copy-in, copy-out, or copy-in/copy-out argument passing. Alternatively, a processor might create an anonymous temporary variable into which the result value of invocation of an accessor function is placed if the dummy argument does not have INTENT(OUT). That anonymous temporary variable is then the actual argument. After the procedure completes execution, if the dummy argument does not have INTENT(IN), that anonymous temporary variable is passed to an invocation of an accessor updater.

2.3 Expressions of type SECTION

1 The syntax of *expr* is extended to include the constructor for objects of type SECTION.

R1022	*expr*	**is**	*level-6-expr*
		or	*section-constructor*
R1022a	*level-6-expr*	**is**	[*level-6-expr defined-binary-op*] *level-5-expr*
R758a	*section-constructor*	**is**	[*scalar-int-expr*] : [*scalar-int-expr*] [: *scalar-int-expr*]

2 A *section-constructor* constructs an object of type SECTION. The value of the kind type parameter of the object is the kind type parameter of the *scalar-int-expr* that has the greatest number of decimal digits, if any *scalar-int-expr* appears. Otherwise, the value of the kind type parameter is the value of default integer kind. The first *scalar-int-expr* provides the lower bound for the section. If it does not appear, the value of the LBOUND component of the value is −HUGE(0_*kind*). The second *scalar-int-expr* provides the upper bound for the section. If it does not appear, the UBOUND component of the value is HUGE(0_*kind*). The third provides the stride. If it does not appear the value of the stride is 1. Its value shall not be zero. The LOWER_BOUNDED component of the value is true if and only if the first *scalar-int-expr* appears. The UPPER_BOUNDED component of the value is true if and only if the second *scalar-int-expr* appears.

3 No intrinsic operations are defined for objects of type SECTION.

4 Intrinsic assignment is defined for objects of type SECTION. The kind type parameter values of the *variable* and *expr* are not required to be the same. The parts of *expr* are assigned to corresponding parts of *variable* as if by intrinsic assignment, except that if the LOWER_BOUNDED (UPPER_-BOUNDED) part of *expr* is false, the LBOUND (UBOUND) part of *variable* is assigned the value −HUGE(*variable*%LBOUND) (HUGE(*variable*%UBOUND)).

5 Where an object of type SECTION appears as an actual argument, the value of its kind type parameter shall be the same as the value of the kind type parameter of the corresponding dummy argument if the dummy argument does not have the VALUE attribute. If the dummy argument has the VALUE attribute, the actual argument is assigned to the dummy argument as if by intrinsic assignment.

2.4 Input/output of objects of type SECTION

1 When an object of type SECTION appears as a *variable* in an *input-item-list* or *output-item-list*, it is processed as a derived-type object using a type-bound defined input/output procedure. If it is a *variable* in an *input-item-list* and the value of the LOWER_BOUNDED (UPPER_BOUNDED) component of the input value is false, the LBOUND (UBOUND) component is assigned the value −HUGE(*variable*%LBOUND) (HUGE(*variable*%UBOUND)), regardless of the value of any input item that is provided. The value of the stride component shall not be zero. If an input item is processed by list-directed formatting and no value is provided for the LBOUND (UBOUND) component, it is assigned the value −HUGE(*variable*%LBOUND) (HUGE(*variable*%UBOUND)). If no value is provided for the STRIDE component, it is assigned the value 1.

2.5 Updater subprogram

2.5.1 Syntax

1 An updater subprogram defines a procedure for which references can appear in variable-definition contexts.

R1537a	*updater-subprogram*	**is**	*updater-stmt*
			[*specification-part*]
			[*execution-part*]
			[*internal-subprogram-part*]
			end-updater-stmt
R1537b	*updater-stmt*	**is**	[*prefix*] UPDATER *updater-name* ■
			■ [([*dummy-arg-name-list*]) [(*aux-dummy-arg-name*)]] ■
			■ [ACCEPT (*acceptor-arg-name*]
R1537c	*acceptor-arg-name*	**is**	*name*
R1537d	*end-updater-stmt*	**is**	END [UPDATER [*updater-name*]]

C1567a (R1537c) *acceptor-arg-name* shall not be the same as *updater-name*. It shall have the VALUE or INTENT(IN) attributes. It shall not have the ALLOCATABLE or POINTER attribute.

C1537b (R1537d) If an *updater-name* appears in the *end-updater-stmt*, it shall be identical to the *updater-name* specified in the *updater-stmt*.

C1567c (R1537a) An ENTRY statement shall not appear within an updater subprogram.

R503	*external-subprogram*	**is**	...
		or	*updater-subprogram*
R1408	*module-subprogram*	**is**	...
		or	*updater-subprogram*

2 The type and type parameters (if any) of the value accepted by the updater may be specified by a type specification in the UPDATER statement or by the name of the acceptor variable appearing in a type declaration statement in the specification part of the updater subprogram. They shall not be specified both ways. If they are not specified either way, they are determined by the implicit typing rules in effect within the updater subprogram. If the acceptor variable is an array, this shall be specified by specifications of the acceptor variable name within the specification part of the updater subprogram. The specifications of the acceptor variable attributes, the specifications of the dummy argument and auxiliary dummy argument attributes, and the information in the UPDATER statement collectively define the characteristics of the updater.

3 If ACCEPT appears, the name of the acceptor variable is *acceptor-arg-name* and all occurrences of the updater name in the *execution-part* statements in its scope refer to the updater itself. If ACCEPT does not appear, the name of the acceptor variable is *updater-name* and all occurrences of the updater name in the *execution-part* statements in its scope refer to the acceptor variable.

4 The acceptor variable is considered to be a dummy argument. Unless it has the VALUE attribute, it is assumed to have the INTENT(IN) attribute, and this may be confirmed by explicit specification.

2.5.2 Updater reference

1 The syntax to reference an updater is similar to the syntax to reference a function. The difference is that if an updater is referenced without arguments, empty parentheses are not required to appear.

R1521b	*updater-reference*	**is**	*procedure-designator* [([*actual-arg-spec-list*]) ■
			■ [(*aux-actual-arg*)]]

C1525c (R1521b) *procedure-designator* shall designate an updater procedure.

2 An updater can only be invoked in a variable-definition context.

R902	*variable*	**is**	...
		or	*updater-reference*

C902a (R902) A *variable* shall not be *updater-reference* except where it appears in a variable-definition context.

3 When an updater is invoked, the following events occur, in the order specified:

1. Actual arguments, if any, are evaluated.
2. Actual arguments are associated to corresponding dummy arguments. The value to define is considered to be an actual argument, and is associated to the updater's acceptor variable.
3. The updater is invoked.
4. Execution of the updater is completed by execution of a RETURN statement or by execution of the last executable construct in the *execution-part*.

NOTE 2.4

One way to think about an updater reference is that it is a time-reversed function reference.

2.5.3 Generic interface

1 Generic interfaces are extended to allow updaters. Further, a function and an updater can have the same generic name. A generic identifier for an updater shall be a generic name. See subclause 2.6.

2.5.4 Example

NOTE 2.5

This example illustrates the use of a function and updater together to access a complex variable.

Assume that a program contains a complex variable represented in Cartesian form, i.e., using the intrinsic COMPLEX type.

NOTE 2.5 (cont.)

```
    complex :: Z
    z = 0.75 * sqrt(2.0) * cmplx ( 1.0, 1.0 ) ! Modulus is 1.5
    print *, 'Z = ', z
    print *, 'Abs(Z) = ', z ! Prints approximately 1.5
```

Assume it is necessary somewhere to change the modulus of the variable, but not its phase:

```
    z = newModulus * ( z / abs(z) )
    print *, 'Revised Z = ' )
```

This would be clearer using an updater:

```
    pure real updater Set_Complex_Abs ( Z ), accept ( V )
      complex, intent(inout) :: Z
      z = v * ( z / abs(z) )
    end updater Set_Complex_Abs

    generic :: Abs => Set_Complex_Abs

    print *, 'Abs(Z) = ', z ! Prints approximately 1.5
    abs(z) = 0.5 * sqrt(2.0)
    print *, 'Revised Z = ', z ! prints approximately 1.0, 1.0
```

Assume this happens sufficiently frequently that the program would be more efficient if complex numbers were represented in polar form:

```
    type :: Polar_Complex
      real :: Modulus
      real :: Phase ! Radians
    end type Polar_Complex

    pure real function Get_Polar_Abs ( Z )
      type(polar_complex), intent(in) :: Z
      get_polar_abs = z%modulus
    end function Set_Polar_Abs

    pure real updater Set_Polar_Abs ( Z ) accept ( V )
      type(polar_complex), intent(inout) :: Z
      z%modulus = v
    end updater Set_Polar_Abs

    pure real updater Set_Cartesian_Abs ( Z ) accept ( V )
      complex, intent(inout) :: Z
      z = v * ( z / abs(z) )
    end updater Set_Cartesian_Abs

    generic :: Abs => Get_Polar_Abs, Set_Polar_Abs, Set_Cartesian_Abs

    type(polar_complex) :: Z
    complex :: Z
```

NOTE 2.5 (cont.)

```
    z = polar_complex ( modulus=1.5, phase=atan(1.0) )
    print *, 'Z = ', z
    print *, 'Abs(Z) = ', abs(z)         ! prints 1.5
    abs(z) = 0.5 * sqrt(2.0) ! Change the modulus but not the phase
    print *, 'Revised Abs(z) = ', abs(z) ! prints approximiately 0.7071
```

Notice that the statement

`abs(z) = 0.5 * sqrt(2.0)`

is the same in both cases. This is a simple example of the principles described by Ross, and by Geschke and Mitchell. Providing the UPDATER subprogram allows to illustrate the principles described by Parnas. In all cases, the cost to modify the program is more likely to be proportional to the magnitude of the change than to the size of the program.

An example illustrating the application to a sparse matrix appears in clause 3.

2.6 Interfaces

2.6.1 Revised syntax

1 The syntax of interface blocks is extended to allow updater interface bodies.

R1505	*interface-body*	**is**	*function-interface*
		or	*subroutine-interface*
		or	*updater-interface*
R1505a	*function-interface*	**is**	*function-stmt* [*specification-part*] *end-function-stmt*
R1505b	*subroutine-interface*	**is**	*subroutine-stmt* [*specification-part*] *end-subroutine-stmt*
R1505c	*updater-interface*	**is**	*updater-stmt* [*specification-part*] *end-updater-stmt*

2.6.2 Generic interfaces

1 Generic interfaces are extended to allow updaters and instance references (2.8.2).

R1507	*specific-procedure*	**is**	*procedure-name*
		or	*instance-reference*

C1509 (R1501) An *interface-specification* in a generic interface block shall not specify a procedure or instance reference that was previously specified in any accessible interface with the same generic identifier.

C1509a (R1508) A *generic-spec* that is not *generic-name* shall not identify an instance reference.

C1510 (R1510) A *specific-procedure* in a GENERIC statement shall not specify a procedure or instance reference that was specified in any accessible interface with the same generic identifier.

2 If a generic interface specifies an instance reference for an accessor, it specifies that all of the specific procedures in the accessor identified by the instance variable name are accessible using that generic identifier.

3 A generic interface can specify specific procedures that are subroutines, functions, updaters, and instance references. Two procedures in a generic interface are distinguishable if they are not defined by the same kind of program unit. For example, a function and updater are distinguishable. The form of reference specifies the kind of procedure to be invoked. For example, a function or updater cannot be invoked by a CALL statement.

NOTE 2.6

The only way to specify a generic identifier for an instance reference is by using a GENERIC statement. There is no syntax to specify a generic identifier for an instance reference using an interface block.

Even though an instance reference identifies an accessor (2.7), which specifies a generic interface for all its contained procedures, an instance reference is nonetheless allowed as a specific name in a generic interface. This allows, for example, to combine instance references for accessors that have different kind type parameters. This is especially important in conjunction with facilities for generic programming.

2.6.3 Explicit interface

1 A subprogram shall have explicit interface where it is referenced if it has a dummy procedure argument that is an updater or an instance reference.

2.7 Accessor definition

2.7.1 Accessor definition

1 An accessor defines an activation record, and may define subroutines, functions, and updaters.

2 An accessor is similar to a derived type definition. One important difference is that procedures are defined within it, not bound to it. Its specification part declares and defines entities within instance variables. Procedures defined within its *accessor-subprogram-part* have access to instance variables, and entities within them, using host association.

R1526a *accessor-def* **is** *accessor-stmt*
    [*specification-part*]
    accessor-subprogram-part
    end-accessor-stmt

R1526b *accessor-stmt* **is** ACCESSOR [, *access-spec*] [::] *accessor-name* ■
    ■ [(*type-param-name-list*)]

R1526c *accessor-subprogram-part* **is** *contains-stmt*
    accessor-subprogram [*accessor-subprogram* . . .]

R1526d *accessor-subprogram* **is** *function-subprogram*
    or *subroutine-subprogram*
    or *updater-subprogram*

R1526e *end-accessor-stmt* **is** END [ACCESSOR [*accessor-name*]]

C1551a (R1526c) If *accessor-name* appears in the *end-accessor-stmt*, it shall be identical to the *accessor-*

name specified in the *accessor-stmt*.

C1551b (R1526a) The SAVE attribute shall not be specified for a variable declared within the *specification-part* of an accessor.

C1551c (R1526a) The *internal-subprogram-part* shall contain at least one function and at least one updater.

3 An accessor can be declared within the specification part of a main program, subprogram, accessor, or module.

R508	*specification-construct*	**is**	. . .
		or	*accessor-def*

2.7.2 PUBLIC and PRIVATE entities of an accessor

1 An *access-stmt* may appear within an accessor.

C869 An *access-stmt* shall appear only in the *specification-part* of a module or accessor. Only one accessibility statement with an omitted *access-id* is permitted in the *specification-part* of a module or accessor.

C869a If an *access-stmt* appears in an accessor, the entities specified by *access-id* shall be accessor procedures or ASSIGNMENT(=) generic identifiers.

2 The default accessibility of accessor procedures and ASSIGNMENT(=) generic identifiers in an accessor is PUBLIC.

2.7.3 Accessor subroutine

1 An accessor subroutine is a subroutine subprogram that is defined within the accessor definition. It can be used to initialize an instance variable of the accessor, to which it has access by host association, or for other problem-dependent purposes.

2.7.4 Accessor function

1 An accessor function is a function subprogram that is defined within the accessor definition. Its usual but not exclusive purpose is to access an instance variable of the accessor, to which it has access by host association. The function subprogram dummy argument list is extended by one additional argument so that the syntax to reference a function can be compatible with the syntax to reference a character variable.

R1530	*function-stmt*	**is**	[*prefix*] FUNCTION *function-name* ■
			■ ([*dummy-arg-name-list*]) [(*aux-dummy-arg-name*)] ■
			■ [*suffix*]
R1530a	*aux-dummy-arg-name*	**is**	*name*

C1561a (R1530a) *aux-dummy-arg-name* shall be a scalar of type SECTION and have the INTENT(IN) or VALUE attributes. It shall not have the POINTER or ALLOCATABLE attribute.

NOTE about C1563

An accessor function within an internal accessor may have an internal subprogram.

2.7.5 Accessor updater

1 An accessor updater is an accessor that is defined within the accessor program unit. Its usual but not exclusive purpose is to modify an instance variable of the accessor, to which it has access by host association.

2.7.6 Pure accessor subprograms

1 Variables that are declared in the specification part of the accessor, which are accessible by host association, may appear in variable definition contexts within pure accessor subprograms. Variables in the specification part of the accessor that are accessed by host or use association shall not appear in variable definition contexts within pure accessor subprograms.

2.8 Instance variable and activation record

2.8.1 Activation record

1 The internal state of an accessor is represented by an activation record.

2 The activation record is declared by the variable declarations within the specification part of the accessor, as if it were the definition of a private non-sequence derived type with private components.

2.8.2 Instance variable declaration

1 An instance variable represents an activation record.

NOTE 2.7

Instance variables allow a single collection of functions, updaters, and subroutines to have independent persistent states that are not represented only by their arguments, and variables accessed by host or use association. Without instance variables, functions and updaters are limited to operations on their arguments, or variables they access by host or use association. Without instance variables, to have independent persistent states, other than by argument association, it is necessary to copy functions, updaters, and subroutines to different scoping units, either physically or by using INCLUDE statements. A function or updater that is accessed by host or use association is the same entity in every scoping unit, and accesses the same scope by host association.

If it is necessary to revise a program so that several similar objects that need independent persistent states are represented by functions and updaters instead of variables, it is necessary either to copy the procedures to different scoping units, one for each persistent state, provide the persistent state explicitly using additional arguments, or use instance variables. Scoping units are not allocatable, so the ”copy” strategy is limited to fixed numbers of persistent states. If the persistent states were to be provided using argument association, each persistent state, a sparse matrix for example, would need to be provided as an argument in addition to, for example, subscripts. This would require all references to and definitions of each original entity to be revised.

2 An instance variable is not a derived-type object. Entities within the specification part of the accessor are not accessible as if they were components.

Unresolved Technical Issue concerning access to accessor specification parts

It is possible in principle to expose entities within the specification part of the accessor as if they were components of an object of derived type. Whether this additional complication is desirable can be decided in due course.

R703	*declaration-type-spec*	**is**	. . .
		or	*instance-declaration*
R1518a	*instance-declaration*	**is**	ACCESSOR (*accessor-name*) [[, *instance-attr-spec*] ::] ■ ■ *instance-variable-name*
R1518b	*instance-attr-spec*	**is**	GENERIC (*generic-name*)
		or	*attr-spec*

C1521a (R1518b) An *attr-spec* shall not be ASYNCHRONOUS, CODIMENSION, CONTIGUOUS, DIMENSION, EXTERNAL, INTRINSIC, *language-binding-spec*, PARAMETER, or VOLATILE.

NOTE 2.8

An instance variable is always a noncoarray scalar.

3 If GENERIC (*generic-name*) appears it is a generic identifier for the procedures defined within the accessor, and *instance-variable-name* identifies the activation record. If GENERIC (*generic-name*) does not appear, *instance-variable-name* is a generic identifier for the procedures defined within the accessor and the instance variable is not accessible by *instance-variable-name*.

4 An instance variable shall not be a subobject of a coarray or coindexed object.

2.8.3 Instance variable reference

1 An instance variable may be referenced directly using the *instance-variable-name* in its *instance-declaration* if a generic identifier is specified in the declaration.

2 If a generic identifier is not specified in the declaration, the *instance-variable-name* is a generic identifier for the procedures in the instance variable's accessor, not for the instance variable itself. It specifies the instance variable that is to be accessible by host association when an accessor procedure is invoked. The instance variable is not accessible except by host association within accessor procedures for the specified *accessor-name*.

R1523b	*instance-reference*	**is**	*instance-variable-name*
		or	*instance-generic-name*

C1529b (R1523b) The *instance-variable-name* shall be the name of an instance variable that does not declare a generic identifier.

C1529c (R1523b) The *instance-generic-name* shall be the name generic identifier specified in the declaration of an instance variable.

2.9 Reference to accessors

2.9.1 Syntax

1 An accessor is referenced using an instance reference, which is either an instance variable for the accessor, or a generic identifier specified in the declaration of an instance variable for the accessor.

2 A reference to an accessor is permitted where a reference to or definition of a variable is permitted. Where an accessor reference appears as an expression it is considered to be a reference to an accessor function subprogram. Where an accessor appears in a variable definition context it is considered to be a reference to an accessor updater subprogram. The specific function or updater is determined using the rules for generic resolution specified in subclause 15.5.5.2 in ISO/IEC 1539-1:2018(E).

3 The syntax of an accessor reference is an extension of the syntax of a function reference. The extension makes it compatible with a scalar reference, an array element reference, a whole array reference, or a character substring reference, which in turn allows the representation of an object to be changed between an accessor and a data object without changing the syntax of references to it.

R1523a	*accessor-reference*	**is**	*instance-reference* [([*actual-arg-spec-list*]) ■
			■ [(*aux-actual-arg*)]]
R1523c	*aux-actual-arg*	**is**	*actual-arg-spec*

C1529a (R1523a) The *procedure-designator* shall designate an accessor.

C1529d (R1523c) The *aux-actual-arg* shall be of type SECTION.

4 Unlike a reference to a function, if an *instance-reference* appears without either *actual-args* or *aux-actual-arg* it nonetheless specifies invocation of the accessor unless it is an actual argument associated with a dummy accessor, or a *data-target* in a pointer assignment statement. For this reason, a procedure shall have explicit interface where it is invoked if it has an accessor dummy argument.

5 The syntax of *designator* is extended to allow references to accessors in value-providing and variable definition contexts.

R901	*designator*	**is**	...
		or	*accessor-reference*

6 The syntax of intrinsic assignment already allows reference to an updater part of an accessor in its variable-definition context.

R1032	*assignment-stmt*	**is**	*variable* = *expr*

7 If the *variable* in *assignment-stmt* is *accessor-reference*, the specific updater is determined according to specifications in subclause 15.5.5.2 of ISO/IEC 1539-1:2018(E), with *expr* as an actual argument corresponding to an accessor acceptor variable.

8 If the *variable* in *assignment-stmt* is *instance-reference* and the specified instance variable does not declare a generic identifier, the *variable* specifies the instance variable, not a generic identifier of the updaters of the accessor. The assignment is intrinsic assignment, as if for *variable* and *expr* of derived type, if the accessor does not define assignment, and defined assignment otherwise.

2.9.2 Execution of an accessor

1 When an accessor is invoked, the following events occur in the order specified.

(1) The actual arguments are evaluated, and associated with their corresponding dummy arguments. If the accessor is invoked to accept a value the value to be accepted is considered to be an actual argument.

(2) The instance variable is made accessible to the invoked procedure by host association.

NOTE 2.9

Making the instance variable available by host association uses the same mechanism at that used to make the host environment of a dummy procedure available by host association.

(3) If the accessor is invoked

- to produce a value an accessor function is executed, or
- to accept a value an accessor updater is executed.

(4) Execution of the function or updater is completed when a RETURN statement is executed, or execution of the last executable construct in the *execution-part* of the accessor function or updater is completed.

2 When the accessor is invoked to accept a value, the accepted object is argument associated with the acceptor variable.

NOTE 2.10

Because an updater's acceptor variable has the VALUE or INTENT(IN) attribute, it is not possible for an updater to change the value associated to its acceptor variable.

2.10 Executing accessor procedures

2.10.1 Procedure designators

1 The syntax for procedure designators is extended to facilitate reference to accessor procedures.

R1520	*procedure-designator*	**is**	. . .
		or	*instance-reference* [%*accessor-procedure-name*]
R1520a	*instance-reference*	**is**	*name*

C1525d (R1521c) If %*accessor-procedure-name* appears, *instance-reference* shall be the name of an accessor instance variable that does not specify a generic identifier in its declaration. If %*accessor-procedure-name* does not appear, *instance-reference* shall be the generic identifier specified in the declaration of an instance variable.

C1525b (R1520) *accessor-procedure-name* shall be an accessible name of an accessor procedure defined within the accessor for which *instance-reference* identifies an instance variable.

2 If *procedure-designator* is *instance-reference*, the invoked procedure has access to the specified instance variable using host association.

3 Where an accessor subroutine is invoked using *instance-reference* without %*accessor-procedure-name*, *instance-reference* is a generic identifier for the specific accessor procedures within the accessor for which *instance-reference* is a generic identifier specified in the declaration of an instance variable, and the invoked procedure is a specific accessor procedure for that generic interface.

4 Where an accessor subroutine is invoked using *instance-reference*%*accessor-procedure-name*, the invoked procedure is the specific accessor procedure of the accessor for which *instance-reference* is an instance variable.

5 When an accessor procedure is invoked, the activation record that it accesses by host association is the one designated by the *instance-reference*.

2.11 Relationship to DO CONCURRENT

1 An *instance-reference* that identifies an instance variable that has SHARED or unspecified locality shall not be invoked in more than one iteration of a DO CONCURRENT construct.

2.12 Argument association of instance variables

1 An *instance-reference* shall be an actual argument if and only if the corresponding dummy argument is an *instance-reference* for the same accessor. The type parameters of the actual and dummy arguments, if any, shall have the same values. The actual argument shall have the GENERIC attribute if and only

if the corresponding dummy argument has the GENERIC attribute. It is not necessary that the actual and dummy argument GENERIC attributes specify the same generic name.

2.13 Pointer association of instance variables

1 Pointer association for instance pointer objects is defined.

R1033 *pointer-assignment-stmt* **is** . . .
or *instance-pointer-object* => *instance-target*

1031a (R1033) An *instance-pointer-object* shall be an *instance-variable-name* that has the POINTER attribute if and only if *instance-target* is an *instance-variable-name* that has the TARGET attribute. The *instance-pointer-object* and *instance-target* shall be *instance-variable-names* for instance variables for the same accessor, and their kind type parameters, if any, shall have the same values. The *instance-pointer-object* shall have the GENERIC attribute if and only if the *instance-target* has the GENERIC attribute.

2 The length parameters of the *instance-pointer-object* and *instance-target*, if any, shall have the same values.

3 It is not necessary that the GENERIC attributes of the *instance-pointer-object* and *instance-target*, if any, specify the same generic name.

2.14 Compatible extension of substring range

1 The type SECTION is provided to allow a dummy argument of type SECTION, so that an accessor can replace an array or character variable without requiring change to the references. It seems pointless to restrict this only to actual arguments, so it makes sense to allow variables other than dummy arguments of type SECTION. Having a variable of type section and not allowing it to be used as a *substring-range* would be silly.

R910 *substring-range* **is** *scalar-section-expr*

R910a *scalar-section-expr* **is** *scalar-expr*

C908a (R910a) The *scalar-expr* shall be an expression of type SECTION.

2 The value of the stride of *scalar-section-expr* shall be 1.

Unresolved Technical Issue 1

Does this introduce a syntax ambiguity?

2.15 Compatible extension of subscript triplet

1 Having a variable of type section and not allowing it to be used as a *subscript-triplet* would be silly.

R921 *subscript-triplet* **is** *scalar-section-expr*

2 If the LOWER_BOUNDED part of the *scalar-section-expr* is false, the effect is as if the LBOUND part were the lower bound of the array. If the UPPER_BOUNDED part of the *scalar-section-expr* is false, the effect is as if the UBOUND part were the upper bound of the array.

NOTE 2.11

Since no operations are defined on objects of type SECTION, the only possible expressions of type SECTION are section constructors, variables of type SECTION, references to functions or accessors of type SECTION, or such an expression enclosed in parentheses. Thus `A((1:10))` is a newly-allowed syntax having the same meaning as `A(1:10)`.

2.16 Compatible extension of vector subscript

1 Having an array of type section and not allowing it to be used as a *vector-subscript* would be silly.

R923 *vector-subscript* **is** *expr*

C927 (R923) A *vector-subscript* shall be an array expression of rank one, and of type integer or SECTION.

2 If *vector-subscript* is of type SECTION and the LOWER_BOUNDED part of any element is false, the effect is as if the LBOUND part were the lower bound of the array. If the UPPER_BOUNDED part of any element is false, the effect is as if the UBOUND part were the upper bound of the array. The resulting vector subscript is then computed as if the elements appeared as arguments to a sequence of references to the SECTION_AS_ARRAY intrinsic function in an array constructor, in array element order.

NOTE 2.12

For example, if A is an array of type SECTION with two elements having values 1:5:2 and 5:1:-2, the effect is as if the subscript were [SECTION_AS_ARRAY(A(1)), SECTION_AS_ARRAY(A(2))], which has the value [1, 3, 5, 5, 3, 1].

2.17 SECTION_AS_ARRAY (A)

1 **Description.** An array having element values of all elements of a section.

2 **Class.** Transformational function bound to the type SECTION from the intrinsic module ISO_Fortran_-Env.

3 **Argument.** A shall be a scalar of type SECTION. Neither A%LOWER_BOUNDED nor A%UPPER_-BOUNDED shall be false. The value of A%STRIDE shall not be zero.

4 **Result Characteristics.** Rank one array of type integer and the same kind as A. The size of the result is the number of elements denoted by the section, which is MAX(0, (A%UBOUND − A%LBOUND + A%STRIDE) / A%STRIDE).

5 **Result Value.** The result value is the same as the expression [(I, I = A%LBOUND, A%UBOUND, A%STRIDE)] where I is an integer of the same kind as A.

NOTE 2.13

The description of the result value makes it clear that SECTION_AS_ARRAY is not really needed; it is pure syntactic sugar.

6 **Examples.** The value of SECTION_AS_ARRAY (5:1:-2) is [5, 3, 1]. The value of SECTION_AS_ARRAY (5:1:2) is [ ] and the size of the result value is zero.

2.18 Existing intrinsic functions as updaters

1 The following genberic intrinsic functions should be defined to be both functions and updaters. When a reference appears in a variable-definition context

- REAL(X) with complex X is equivalent to X%RE,
- AIMAG(X) with complex X is equivalent to X%IM,
- ABS(X)
  - with integer or real X changes the magnitude of X without changing the sign, or
  - with complex X changes the modulus without changing the phase,
- FRACTION(X) with real X changes the fraction but not the exponent, and
- EXPONENT(X) with real X changes the exponent but not the fraction.

3 Extended example

3.1 General

1 The use of an accessor to replace a dense matrix by a sparse matrix is illustrated.

2 If a matrix is represented by a rank-2 Fortran array, references to elements of the array use the array name, followed by two subscripts. Definitions of elements of the array use the array name, followed by two subscripts.

3 It is not unusual for the requirements of a problem to change in such a way that a rank-2 Fortran array cannot be used because the matrix is too large, but the matrix is sparse. One could replace the array by a derived-type object, and provide a function that has the same name as the array to reference elements of the array.

4 But definitions of elements of the array would need to be replaced by calls to a subroutine.

5 Defined assignment cannot be used as an "updater" because the subroutine that defines assignment is allowed to have only two arguments – the first is associated with the *variable* in a defined assignment statement, and the second is associated with the *expr*. There is no provision for arguments that play the rôle of subscripts.

6 A function that returns a pointer associated with an element cannot be used. If it is used in a reference context, and the designated element is zero (and therefore not represented), the function could return a pointer associated with a save variable that has the value zero. But if the same function is to be used as a "left-hand function" in a variable-definition context, if the designated element does not exist, it needs to create one so that it can return a pointer associated with an array element into which the value, for example the value of the *expr* in the assignment statement, can be stored. It cannot decide not to store a nonzero value because (a) it does not receive the value to be stored, and (b) it does not store the value – it merely provides a pointer associated with a place where the assignment statement can store a value.

3.2 A derived type to represent a sparse matrix

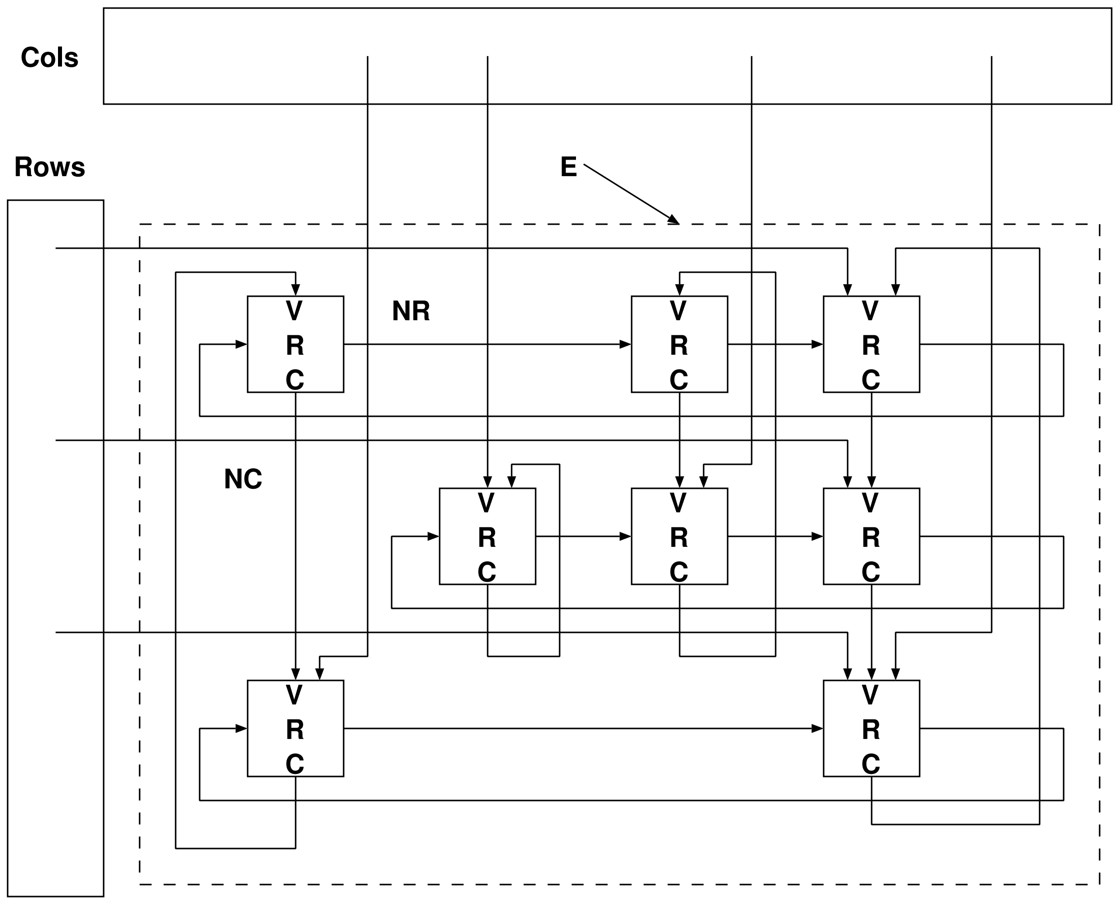
1 A derived type to represent a sparse matrix might be defined by

```
  type :: Sparse_Element_t ! One element in a sparse matrix
    real :: V      ! Element value
    integer :: R   ! In which row is the element
    integer :: C   ! In which col is the element
    integer :: NR  ! Next element in same row
    integer :: NC  ! Next element in same col
  end type Sparse_Element_t

  type :: Sparse_t ! Representation for a sparse matrix
    integer :: NE = 0   ! Number of elements actually used,<= size(E)
    ! Rows and columns are circular lists, so last element points to first:
    integer, allocatable :: Rows(:) ! Last element in each row
    integer, allocatable :: Cols(:) ! Last element in each col
    type(sparse_element_t), allocatable :: E(:) ! nonzero elements
  contains
    procedure :: Create   ! Allocate rows, cols, set to zero
    procedure :: Add_Element_Value
    procedure :: Destroy  ! Deallocate everything
    procedure :: Empty    ! set rows, cols, NE to zero
    procedure :: Find_Element_Value
    ....
    final :: Destroy
  end type Sparse_t
```

2 Each element in the `cols` component is the index in the `E` component of the last nonzero element in the corresponding column. Each element in the `rows` component is the index in the E component of the last nonzero element in the corresponding row.

3 Each element in the `E` component includes the value of the element, the row and column subscripts of the element, the index in `E` of the next element in the same row, and the next element in the same column. A graphical representation of a matrix with seven nonzero elements follows.



4 An accessor to reference and define elements of the array might be

```
  module Sparse_m
    ! Type definitions above
  contains
    accessor Sparse_Matrix_Element
      type(sparse_t) :: Matrix
      public
    contains
      real function Element_Ref ( I, J ) result ( R )
        integer, intent(in) :: I, J ! Row, column of the element
        r = matrix%find_element_value ( i, j )
      end function Element_Ref
      real updater Element_Def_Real ( I, J ) accept ( V )
        call matrix%add_element_value ( v, i, j ) ! doesn't add zeroes
      end updater Element_Def_Real
      integer updater Element_Def_Int ( I, J ) accept ( V )
        call matrix%add_element_value ( real(v), i, j ) ! doesn't add zeroes
      end updater Element_Def_Int
      subroutine Initialize ( nRows, nCols, InitNumElements )
        integer, intent(in) :: nRows, nCols
        integer, intent(in), optional :: InitNumElements
        call matrix%create nRows, nCols, InitNumElements )
      end subroutine Initialize
      subroutine CleanUp
        call matrix%destroy
      end subroutine CleanUp
      subroutine Empty
        call matrix%empty
      end subroutine Empty
    end accessor Sparse_Matrix_Element
  end module Sparse_m
```

5 The accessor might be accessed, and an instance variable declared for it, using

```
use Sparse_m, only:  Sparse_Matrix_Element
```

```
accessor ( Sparse_Matrix_Element ) ::  A
```

6 The instance variable might then be initialized using

```
call a ( nRows=1000000, nCols=100000 )
```

7 which references the `Initialize` accessor subroutine using generic resolution, or

```
call a%initialize ( nRows=1000000, nCols=100000 )
```

8 To fill the array, one could use

```
   do
     read ( *, *, end=9 ) i, j, v
     a ( i, j ) = v
   end do
 9 continue
```

9 To reference an element of the array, one could use

```
print '("Element = ", 1pg15.8)', a(i,j)
```

10 If one later needs a new set of values for an array with the same shape, one could use

```
call a%empty
```

11 To destroy the array without destroying the accessor's activation record, one could use

```
call a%cleanUp
```

4 Required editorial changes to ISO/IEC 1539-1:2018(E)

To be provided in due course.