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

Date: 2023-01-12

Reference number of document: **ISO/IEC TS 99999:2019(E)**

Committee identification: ISO/IEC JTC1/SC22

Secretariat: ANSI

Information technology — Programming languages — Fortran — Coroutines and Iterators

Technologies de l'information — Langages de programmation — Fortran — Coroutines et Iterators

Contents

0 Introduction

0.1 History

Fortran has historically been primarily but not exclusively used to solve problems in science and engineering. Solving problems in science and engineering primarily but not exclusively depends upon computational mathematics algorithms. Computational mathematics algorithms frequently require access to software that is provided by the user, to specify the problem. Examples include evaluating integrals, solving differential equations, minimization, and nonlinear parameter estimation.

Software to solve problems in science and engineering also benefits from the application of principles of software engineering, as explained, for example, in **Scientific Software Design: The Object-Oriented Way**, by our colleagues Damian Rouson and Jim Xia, and their coauthor Xiaofeng Xu. An important paradigm related to object-oriented programming is a *container*. Support to develop containers in Fortran is part of the work plan for the next revision. It is important to be able to iterate over the contents of a container, without exploiting the representation of the container. Examples include traversing a list or tree, or a row or column of a sparse matrix. The procedures of a container that iterate over its contents require access to software that is provided by the user, to perform actions using the members of the container.

In Fortran, access to software that is provided by the user has been provided in three ways.

- The procedure that implements the algorithm invokes a procedure of a specific name,
- The name of the procedure that defines the problem is passed to the procedure that implements the algorithm, or
- The procedure that implements the algorithm returns to the invoker whenever it requires a computation that defines the problem.

The first two of these methods are called *forward communication*; the last is called *reverse communication.*

Forward communication works well in the simple cases where the procedure that implements the algorithm can provide all the information needed by the procedure that defines the problem.

Before Fortran 2003, when additional information was needed, programs exploited methods known to reduce the reliability of programs or increase the cost of their development and maintenance: global data. Fortran 2003 provides type extension, which reduces the problem substantially, but can introduce other problems such as performance penalties caused by pointer components.

Programs developed in Fortran 2003 would probably use type extension to pass additional data to the procedure that defines the problem. Revising existing programs that use reverse communication to use type extension could be prohibitively expensive, especially if rigorous recertification is required, while revising them to use coroutines would be relatively inexpensive.

Reverse communication does not require information necessary to define the problem to be passed through the procedure that implements the algorithm, or require the procedure that defines the problem to access such information by using global data or type extension. There is, however, no structured support for reverse communication in Fortran. In order for the procedure to continue after the calculations that define the problem, it has to know it isn't starting a problem, and how to find its way to continue its process. This usually involves GO TO statements, or transformation of the procedure into an inscrutable "state machine." The state of the computation is usually represented in SAVE variables, which causes the procedure that implements the algorithm not to be thread safe.

A third alternative is mutual recursion with tail calls.

In some problems, it is desirable to preserve the activation record, primarily to avoid re-creating automatic variables. If a procedure is used to solve a large number of related problems, and it requires substantial "working storage," re-creating working storage as automatic variables, or allocating allocatable variables or pointers that do not have the SAVE attribute, can be a significant fraction of the total cost of solving one problem. Alternatives are allocatable variables or pointers with the SAVE attribute, which are not thread safe, and host association, which militates against reuse.

If coroutines had been available during the development of Fortran 2003, defined input/output would not have been needed. Instead, it could have been possible to specify a coroutine to process the input or output list, having an unlimited polymorphic argument to associate with each list item in turn.

0.2 What this technical specification proposes

This technical specification proposes two forms of procedures. They both have the property that they have a persistent internal state that is created by their initial invocation. They can be suspended and later resumed, to proceed from the point where they were suspended. The persistent internal state is represented by local entities. Local entities and the state of execution of the procedure are preserved in an *activation record*; local entities do not become undefined when the procedure is suspended.

A *coroutine* can be invoked in the same way as a subroutine. It can be resumed wherever and whenever necessary.

An *iterator* can be invoked in the same way as a function, but only in a new ITERATE construct. When a function is invoked, it returns a value. One would expect that when an iterator is resumed, it would return a value, but there is only one way to indicate which instance of the iterator is to be resumed, to provide a value: a looping construct. A Wikipedia article describes the iterator as

> ... one of the twenty-three well-known GoF design patterns that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.

The term "coroutine" first appeared in documentation of the language Simula. Tasks and protected variables in Ada are similar to coroutines.

Coroutines are supported directly in the following languages:

- Aikido
- AngelScript
- BCPL
- Pascal (Borland Turbo Pascal 7.0 with uThreads module)
- BETA
- BLISS
- C#
- ChucK
- CLU
- D
- Dynamic C
- Erlang
- F#
- Factor
- GameMonkey Script
- GDScript (Godot's scripting language)
- Go
- Haskell
- High Level Assembly
- Icon
- Io
- JavaScript (since 1.7, standardized in ECMAScript 6) ECMAScript 2017 also includes await support.
- Julia
- Kotlin (since 1.1)
- Limbo
- Lua
- Lucid
- μC++
- MiniD
- Modula-2
- Nemerle
- Perl 5 (using the Coro module)
- Perl 6
- PHP (with HipHop, native since PHP 5.5)
- Picolisp
- Prolog
- Python (since 2.5, with improved support since 3.3 and with explicit syntax since 3.5)
- Ruby
- Sather
- Scheme
- Self
- Simula 67
- Smalltalk
- Squirrel
- Stackless Python
- SuperCollider
- Tcl (since 8.6)
- urbiscript

Iterators are supported directly in the following programming languages:

- C++
- C# and other .NET languages
- Java
- JavaScript
- Matlab
- PHP
- Python
- Ruby
- Rust
- Scala

All of the alternatives that presently exist in Fortran, described in the previous subclause, require to invoke and return from a procedure to respond to a need to execute "user" code. In contrast, when a coroutine or iterator is suspended its activation record is not destroyed, and when it it is resumed its activation record is not reconstructed. Therefore, suspending and resuming a coroutine or iterator is generally more efficient than the alternatives.

Information technology – Programming Languages – Fortran

Technical Specification: Coroutines and iterators

1 General

1.1 Scope

This technical specification specifies extensions to the programming language Fortran. The Fortran language is specified by International Standard ISO/IEC 1539-1:2019(E). The extensions are varieties of procedures known as *coroutines* and *iterators*. They have the property that an instance of one can be suspended, and later resumed to continue execution from the point where it was suspended. Local entities and the state of execution of the procedure are preserved in an *activation record*, and do not become undefined when the procedure is suspended. The invoking scope retains the activation record, and can have as many separate activation records for each procedure as necessary.

Clause 2 of this technical specification contains a general and informal but precise description of the extended functionalities. Clause 3 contains several illustrative examples. Clause 4 contains detailed instructions for editorial changes to ISO/IEC 1539-1:2019(E).

1.2 Normative References

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.

ISO/IEC 1539-1:2019(E) : *Information technology – Programming Languages – Fortran; Part 1: Base Language*

2 Requirements

2.1 General

The following subclauses contain a general description of the extensions to the syntax and semantics of the Fortran programming language to provide coroutines and iterators.

2.2 Summary

2.2.1 What is provided

This technical specification defines new forms of procedures, called *coroutines* and *iterators*, an instance of which can be suspended and later resumed to continue execution from the point where it was suspended. Local entities and the state of execution of the procedure are preserved in an *activation record*, and do not become undefined when the procedure is suspended. The invoking scope retains the activation record, and can have any number of activation records. There is presently nothing comparable in Fortran, but coroutines and iterators have been provided by numerous other programming languages.

This technical specification describes statements to define coroutines and iterators, statements to suspend, resume, and terminate coroutines, an inquiry function to determine whether a coroutine is suspended, and a looping control construct that invokes an iterator.

2.2.2 Coroutines

A coroutine is a procedure that is invoked similarly to the way a subroutine is invoked. Unlike a subroutine, it can be suspended, and later resumed to continue execution from the point where it was suspended. Local entities and the state of execution of a coroutine are preserved in an *activation record*, and do not become undefined when it is suspended. Each invocation of a coroutine creates a new instance, independently of whether an instance is already in a state of execution. The invoking scope retains the activation record, and can have as many activation records as necessary. A coroutine can be pure, but it cannot be elemental. A coroutine identifier shall have explicit interface where it is invoked or resumed.

2.2.3 Iterators

An iterator is a procedure that produces a result value, as does a function subprogram. It is intended to be used as an abstraction to produce the elements of a data structure, one at a time. It can be invoked or resumed only within the ITERATE statement of an ITERATE construct. Local entities and the state of execution of an iterator are preserved in an *activation record*, and do not become undefined when it is suspended. A different instance exists for each ITERATE construct. Nested ITERATE constructs can use the same iterator. An iterator identifier shall have explicit interface where it appears in an ITERATE construct.

2.2.4 ITERATE construct

The ITERATE construct uses an iterator to process the elements of a data structure, one at a time. When execution of the construct commences, the iterator is invoked and a new instance of it is created. Therefore, an ITERATE construct within another ITERATE construct can use the same iterator. Each time the iterator suspends it provides a value, or a pointer associated with a value, and the body of the construct is executed. After the construct body is executed, the iterator is resumed at the first executable construct after the SUSPEND statement that suspended execution of the iterator. Execution of the ITERATE construct completes, the activation record of the instance is destroyed, and the instance of the iterator ceases to exist when

- the iterator executes a RETURN, END, or STOP statement,

NOTE for J3

A STOP statement is included in the description in case exception handling is provided as described in J3/23-106. Execution of a STOP statement can raise an exception.

- an EXIT statement that belongs to the construct is executed,
- an EXIT or CYCLE statement that belongs to an outer construct and is within the range of the construct is executed,
- a branch occurs from a statement within the ITERATE construct to a statement that is neither the *end-iterate-stmt* nor within the range of the construct, or
- a RETURN or STOP statement within the range of the construct is executed.

2.2.5 SUSPEND statement

When an instance of a coroutine or iterator executes a SUSPEND statement, execution of the instance is suspended; local variables of the instance do not become undefined. For a coroutine, the sequence of execution continues after the CALL statement that invoked the coroutine, or after the RESUME statement that resumed execution of the same instance of the coroutine, whichever occurred most recently. For an iterator, the sequence of execution proceeds to the *block* of the ITERATE construct.

2.2.6 RESUME statement

When a RESUME statement is executed the procedure designator in the RESUME statement shall designate an instance variable of a suspended instance of a coroutine. Execution of the specified instance of the specified coroutine is resumed by re-establishing argument associations and transferring control to the first executable construct after the SUSPEND statement that most recently suspended execution of the specified instance of the coroutine. Expressions in the specification part are not re-evaluated, and the specification part is not elaborated again. Therefore, local variables of the instance, including automatic variables, retain the same bounds, length parameter values, definition status, and values if any, that they had when the instance was suspended.

NOTE 2.1

Because argument associations are re-established, dummy arguments might have different extents, length parameter values, allocation status, pointer association status, or values (if any).

2.2.7 The TERMINATE statement

When a TERMINATE statement is executed, the activation record of the specified instance of the specified coroutine is destroyed and that instance of the coroutine cannot thereafter be resumed. The procedure designator in the TERMINATE statement shall designate an instance variable of a suspended instance of the coroutine.

An instance of a coroutine that is not suspended shall not be terminated.

2.3 Coroutine syntax and semantics

2.3.1 Coroutine definition syntax

A coroutine is a subprogram. It can be an external subprogram, a module subprogram, an internal subprogram, or a separate module procedure. It can be bound to a type. It can be pure, but it cannot be elemental. Each invocation of a coroutine creates a new instance, independently of whether an instance is already in a state of execution. Suspending a coroutine does not destroy an instance. Resuming a coroutine does not create a new instance.

R1537a *coroutine-subprogram* **is** *coroutine-stmt*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-coroutine-stmt

R1537b *coroutine-stmt* **is** [*prefix*] COROUTINE *coroutine-name* ■
■ [([*dummy-arg-name-list*])]

R1537c *end-coroutine-stmt* **is** END COROUTINE [*coroutine-name*]

C1251a (R1537b) Neither *declaration-type-spec* nor ELEMENTAL shall appear in *prefix*.

C1251b (R1537a) An internal coroutine subprogram shall not contain an *internal-subprogram-part*.

C1251c (R1537c) If a *coroutine-name* appears in the *end-coroutine-stmt* it shall be identical to the *coroutine-name* in the *coroutine-stmt*.

NOTE 2.2

When a coroutine is invoked by a CALL statement, a new instance of its activation record is created, regardless whether it is invoked recursively. Therefore, whether RECURSIVE or NON_RECURSIVE appears in the prefix is irrelevant.

Unresolved Technical Issue Recursive Coroutine

The appearance of RECURSIVE or NON_RECURSIVE in the prefix could be prohibited instead of ignored.

2.3.2 Coroutine interface body

The interface of a coroutine can be declared by an interface body.

R1505 *interface-body* **is** ...
or *coroutine-stmt* ■
■ [*specification-part*]
■ *end-coroutine-stmt*

2.3.3 Coroutine reference

2.3.3.1 General

An identifier of a coroutine shall have explicit interface where it is invoked or resumed.

2.3.3.2 Coroutine instance variables

A *coroutine instance variable* represents an instance of a coroutine's activation record.

Within a scoping unit, if the *coroutine-name* of a coroutine, or a name associated with one by use or host association, appears as the *procedure-designator* in a CALL statement, or as an actual argument that corresponds to a dummy argument that does not have the VALUE attribute, a local instance variable identified by that *procedure-designator* exists and has a scope of that inclusive scope.

A coroutine procedure pointer, or a dummy procedure that has a coroutine interface, is an instance variable.

If an object is of a type that has a type-bound coroutine, that object contains an instance variable for that coroutine, identified by that binding.

An instance variable is not a local variable if it is

- a dummy coroutine without the VALUE attribute,
- accessed by use or host association, or
- represented within an object of derived type that has a binding to the coroutine, and the object is not a local variable.

Otherwise, it is a local variable.

An instance variable is an object of a private derived type defined by the processor, with private components. It identifies a coroutine and represents an instance of its activation record. The types of different instance variables are not necessarily the same, but they all have a private allocatable activation record component, and a private procedure pointer component that identifies the coroutine. If it is a dummy procedure with a coroutine interface, the association of the procedure pointer component is that of the corresponding actual argument. Otherwise, if it is a coroutine pointer, the procedure pointer component has default initialization of NULL(). Otherwise, the procedure pointer component is associated with the coroutine specified by the *procedure-designator*.

2.3.3.3 Coroutine activation records

An instance variable has a private allocatable component that represents the coroutine's activation record. It is allocated if and only if the instance of the coroutine is active. The activation record represents the state of execution of the instance, and its unsaved local variables. A local variable of a coroutine that has the SAVE attribute is shared by all instances; it is not part of an activation record. Variables accessed by use and host association are not part of an activation record.

The activation record component of a local instance variable is initially deallocated, even if it is a dummy coroutine with the VALUE attribute. A local instance variable does not initially represent an active instance when the procedure is invoked, even if it is a dummy coroutine with the VALUE attribute and the corresponding actual argument represents an active instance. Unlike a dummy data object with the VALUE attribute, the allocation status, and value if any, of the allocatable component that represents its activation record, is not copied from the actual argument that corresponds to a dummy coroutine with the VALUE attribute.

NOTE 2.3

Because the activation record component of an instance variable is allocatable, it is or becomes deallocated, and the instance it represents is terminated, under the same conditions that an allocatable component of a derived-type object is or becomes deallocated.

An instance of a coroutine is accessible if and only if is represented by an accessible instance variable that represents an active instance.

2.3.3.4 Creating an instance of a coroutine

When a coroutine is invoked by a CALL statement, the following occur in the order specified:

1. Arguments associations are established.

2. An instance of the coroutine is created.
3. The activation record component of its instance variable is allocated as if by an ALLOCATE statement.
4. Expressions within its specification part are evaluated and its specification part is elaborated, creating local variables of the instance that do not have the SAVE attribute.

When the instance executes a RETURN, END, STOP, or SUSPEND statement, or completes execution of the last executable construct of the coroutine's *execution-part*, execution of the CALL statement is completed.

2.3.3.5 Suspending a coroutine instance

When an instance of a coroutine executes a SUSPEND statement, execution of the instance of the coroutine is suspended and the execution sequence continues by executing the executable construct following the CALL statement that invoked that instance of that coroutine, or the RESUME statement that resumed execution of that instance of that coroutine, whichever occurred most recently. Local variables of the instance, within the activation record component of its instance variable, retain their bounds, length parameter values, definition status, and values if any.

2.3.3.6 Resuming a coroutine instance

An instance of a coroutine is resumed by executing a RESUME statement (2.2.6) with a designator that designates its instance variable. When it is resumed, argument associations are re-established and control is transferred to the first executable construct after the SUSPEND statement that most recently suspended execution of the instance of the coroutine represented by the instance variable used to resume it. Its activation record is not re-created. Expressions in the specification part are not re-evaluated, and the specification part is not elaborated again. Therefore, local variables of the instance, including automatic variables, retain the same bounds, length parameter values, definition status, and values if any, that they had when the instance was suspended.

NOTE 2.4

Because argument associations are re-established, dummy arguments might have different extents, length parameter values, allocation status, pointer association status, or values (if any).

If a coroutine is invoked before a DO CONCURRENT construct begins execution, the same instance of it shall not be resumed during more than one iteration of that execution of that construct. A coroutine shall not be invoked using the same instance variable during more than one iteration of a DO CONCURRENT construct. If a coroutine is invoked during an iteration of a DO CONCURRENT construct, that instance of it shall be terminated during that iteration, and it it shall not be terminated or resumed during a different iteration of that execution of that construct.

If a coroutine is invoked from within a CRITICAL construct or from within a procedure invoked during execution of a CRITICAL construct, the same instance of it shall be terminated during that execution of that construct, and it shall not be resumed after that execution of that construct completes. If a coroutine is invoked before execution of a CRITICAL construct begins, the same instance of it shall not be resumed from within that execution of that CRITICAL construct or from within a procedure invoked during that execution of that CRITICAL construct.

Unresolved Technical Issue Critical

The restrictions concerning critical sections might not be necessary or useful.

An instance of a coroutine that has ceased to exist shall not be resumed.

2.3.3.7 Terminating a coroutine instance

An instance of a coroutine is terminated, and the activation record component of the instance variable used to terminate the instance becomes deallocated, when

- a RETURN, STOP, or END statement is executed by the instance of the coroutine,
- the last executable construct of the *execution-part* of the coroutine completes execution,
- a TERMINATE statement that designates the instance variable is executed,
- a CALL statement invokes the coroutine using its instance variable,
- the instance variable is an unsaved local variable of a procedure that is not a coroutine, and execution of the procedure in which it is a local variable is terminated,
- the instance variable is an unsaved local variable of a BLOCK construct and execution of the construct is completed,
- the instance variable is an unsaved local variable of a coroutine and the instance of that coroutine is terminated,
- the instance variable is the *proc-pointer-object* in a pointer assignment statement that is executed,
- the instance variable is a *proc-pointer-object* in a NULLIFY statement that is executed, or
- the instance variable corresponds to a dummy procedure pointer that has INTENT(OUT) and the CALL statement or function reference is executed.

Unresolved Technical Issue Duplicate

> Executing a CALL statement that references a coroutine using a designator with which an instance is associated could alternatively be defined to be an error.

2.3.4 Coroutine procedure pointers

A coroutine procedure pointer is an instance variable. The ASSOCIATED intrinsic function inquires whether the procedure pointer component is associated with a coroutine. The SUSPENDED intrinsic function inquires whether its activation record component is allocated, that is, whether it represents an instance of a coroutine that has not terminated.

A coroutine procedure pointer shall not be a coindexed object or a subobject of a coindexed object.

2.3.5 SUSPEND statement

Execution of a suspend statement within a coroutine suspends execution of an instance of that coroutine (2.3.3.5).

Execution of a suspend statement within an iterator suspends execution of an instance of that iterator (2.4.4).

R1542a *suspend-stmt* **is** SUSPEND

C1276a (R1241a) A *suspend-stmt* shall appear only within the inclusive scope of a coroutine or iterator.

2.3.6 RESUME statement

Execution of a RESUME statement causes execution of an instance of a coroutine to be resumed (2.3.3.6).

R1525a *resume-stmt* **is** RESUME *procedure-designator* [([*actual-arg-spec-list*])]

C1537b (R1525a) The *procedure-designator* shall designate a coroutine instance variable.

C1537b (R1525a) The *procedure-designator* shall not be a coindexed object or a subobject of a coindexed object.

The *procedure-designator* shall designate a suspended instance of a coroutine.

When a RESUME statement is executed, argument associations are re-established, but expressions in the specification part of the coroutine are not re-evaluated and the specification part is not elaborated again. Therefore, local variables, including automatic variables, of the instance retain the same bounds, length parameter values, definition status, and values if any, that they had when the instance was suspended.

NOTE 2.5

Because argument associations are re-established, dummy arguments might have different extents, length parameter values, allocation status, pointer association status, or values (if any).

When the instance of the coroutine that is resumed by execution of a RESUME statement executes a SUSPEND, RETURN, or END statement, execution of the RESUME statement is completed.

2.3.7 SUSPENDED (PROC)

Description. Whether a coroutine is suspended.

Class. Transformational function.

Argument. PROC shall be a *procedure-designator* that designates a coroutine instance variable. It shall not be a coindexed object or a subobject of a coindexed object.

Result Characteristics. Default logical.

Result Value. The result has the value true if and only if the activation record component of PROC is allocated.

2.3.8 The TERMINATE statement

Execution of a TERMINATE statement causes an instance of a coroutine to be terminated (2.3.3.7).

R1525b	*terminate-stmt*	**is**	TERMINATE (*instance-variable* [*terminate-opt-list*]
R1525c	*terminate-opt*	**is**	STAT = *stat-variable*
		or	ERRMSG = *errmsg-variable*
R1525d	*instance-variable*	**is**	*procedure-name*
		or	*proc-pointer-object*
		or	*proc-component-ref*

C1537c (R1525c) The *instance-variable* shall designate a coroutine instance variable.

C1537d (R1525c) The *instance-variable* shall not be a subobject of a coindexed object.

The *procedure-designator* shall designate an instance variable of a coroutine, and its activation record component shall be allocated. A coroutine instance shall not terminate itself by executing a TERMINATE statement.

When a TERMINATE statement is executed, the activation record component of the instance variable becomes deallocated, as if by execution of a DEALLOCATE statement. The effects of STAT= and ERRMSG= specifiers include the same effects as in a DEALLOCATE statement, including the case when the *instance-variable* designates an inactive instance. In addition, if a coroutine instance terminates itself by executing a TERMINATE statement, a processor-dependent nonzero value shall be assigned to *stat-variable*, and that value shall be different from any value that might be assigned by a DEALLOCATE statement. If the activation record component of the instance variable is not allocated or a coroutine instance terminates itself by executing a TERMINATE statement, and STAT= does not appear, an error condition exists.

2.3.9 Coroutine to process input or output statement

The READ and WRITE statements are revised to include an optional PROCESSOR=*coroutine-name* specifier. The PROCESSOR=specifier shall not appear in a statement that specifies namelist or list-directed formatting, or that has both ASYNCHRONOUS='YES' and SIZE= specifiers. The specified coroutine shall have the following interface:

```
coroutine coroutine-name ( unit, item, format, iostat, iomsg, size )
  integer, intent(in) :: unit
  class(*), INTENT(intent-spec), optional :: item(..)
  character(*), intent(in), optional :: format
  integer, intent(out), optional :: iostat
  character(*), intent(inout), optional :: iomsg
  integer, intent(out), optional :: size
end coroutine coroutine-name
```

Unresolved Technical Issue Item argument

Instead of requiring the `item` argument to be unlimited polymorphic, it could be required to be type compatible with every data transfer list item.

If the statement is a READ statement, the *intent-spec* of its `item` argument shall be OUT. If it is a WRITE statement, the *intent-spec* of its `item` argument shall be IN.

When a data transfer statement with a PROCESSOR=*coroutine-name* specifier is executed, the specified coroutine is invoked even if there is no first list item. The processor resumes the coroutine if and only if there is another list item, to process each list item. The `item` argument is present if and only if there is another list item.

The `format` argument is present if and only if the data transfer statement is a formatted data transfer statement. The value of the `format` argument begins and ends with parentheses, and corresponds to the `item` argument, as if the item and format were processed without using the coroutine. It might contain edit descriptors even if the `item` argument is not present; for example, it might contain control or character string edit descriptors.

If a list item is of a derived type that has a pointer or allocatable direct component, and the data transfer statement is a formatted data transfer statement, the corresponding format item shall be a DT edit descriptor. If the corresponding format item is a DT edit descriptor, or the list item is of a derived type that has a pointer or allocatable direct component, the list item is associated with the `item` argument. Otherwise, the list item is expanded as specified in subclause 12.6.3 of ISO/IEC 1539-1:2019(E)

The `iostat` or `iomsg` argument is present if and only if the corresponding specifier appears in the data transfer statement; it is associated with the specified entity.

If an error, end-of-file, or end-of-record condition occurs, and the `iostat` argument is present, the coroutine shall assign the appropriate value to that argument, as specified in subclause 12.11 of ISO/IEC 1539-1:2019(E). If the `iomsg` argument is present, a value may be assigned to it. If the `iostat` argument is absent, the coroutine shall return rather than suspending. If no error occurs and the `iostat` argument is present, the value zero shall be assigned to it. A value shall not be assigned to the `iomsg` argument unless a nonzero value is or would be assigned to the `iostat` argument. If no error, end-of-file, or end-of-record condition occurs the coroutine shall suspend.

The `size` argument is present if and only if the data transfer statement is a READ statement in which a SIZE= specifier appears. If it is present, a value shall be assigned to it, to specify the number of characters transferred from the file.

If the data transfer statement is a formatted data transfer statement, data transfer statements other than those that specify an internal file that are executed while the coroutine is active are processed as if ADVANCE='NO' were specified, even if ADVANCE='YES' is specified in the statement that caused the coroutine to be executed.

After processing the last list item, or if the coroutine assigns a nonzero value to the `iostat` argument, the processor terminates the coroutine. Because the coroutine might use asynchronous data transfer statements, after terminating the coroutine, the processor performs a wait operation if the statement that caused the coroutine to be executed is not an asynchronous data transfer statement.

If the coroutine terminates instead of suspending, an error condition occurs in the statement that caused the coroutine to be executed.

2.4 ITERATOR and ITERATE construct syntax

2.4.1 ITERATOR syntax

An iterator is a subprogram. It can be an external subprogram, a module subprogram, an internal subprogram, or a separate module procedure. It can be bound to a type. It can be pure, but it cannot be elemental.

R1532a *iterator-subprogram* **is** *iterator-stmt*
    [*specification-part*]
    [*execution-part*]
    [*internal-subprogram-part*]
    end-iterator-stmt

R1532b *iterator-stmt* **is** [*prefix*] ITERATOR *iterator-name* ■
    ■ ([*dummy-arg-name-list*]) [RESULT (*result-name*)]

R1532c *end-iterator-stmt* **is** END ITERATOR [*iterator-name*]

C1564a (R1532b) If RESULT appears, *result-name* shall not be the same as *iterator-name*.

C1564b (R1532b) If RESULT appears, the *iterator-name* shall not appear in any specification statements in the scoping unit of the iterator subprogram.

C1564c (R1532b) ELEMENTAL shall not appear in *prefix*.

C1564d (R1532a) An internal iterator subprogram shall not contain an *internal-subprogram-part*.

C1564e (R1532c) If an *iterator-name* appears in the *end-iterator-stmt* it shall be identical to the *iterator-name* in the *iterator-stmt*.

The result variable name of an iterator is the *result-name* if one appears; otherwise it is the *iterator-name*.

NOTE 2.6

When an iterator is invoked by an ITERATE construct, a new activation record is created, even if it is invoked recursively. Therefore, whether RECURSIVE or NON_RECURSIVE appears in the prefix is irrelevant.

Unresolved Technical Issue Recursive Iterator

The appearance of RECURSIVE or NON_RECURSIVE in the prefix could be prohibited instead of ignored.

2.4.2 Iterator interface body

An iterator interface can be declared by an interface body.

R1505 *interface-body* **is** ...
or *iterator-stmt*
[*specification-part*]
end-iterator-stmt

2.4.3 ITERATE construct syntax

An ITERATE construct is used to iterate over the elements of a data structure, which elements are provided by invoking and resuming an iterator.

R1139a *iterate-construct* **is** *iterate-stmt*
block
end-iterate-stmt

R1139b *iterate-stmt* **is** [*iterate-construct-name:*] ITERATE [CONCURRENT] ■
■ (*iteration-control*)

R1139c *iteration-control* **is** *variable* = *iterator-reference*
or *data-pointer-object* => *iterator-reference*
or *declaration-type-spec* [, *iterate-attrib-list*] :: ■
■ *variable-name* [(*array-spec*)] = *iterator-reference*
or *declaration-type-spec* [, POINTER] :: ■
■ *variable-name* [(*array-spec*)] => *iterator-reference*

R1139d *iterate-attrib* **is** ALLOCATABLE
or TARGET

R1139e *end-iterate-stmt* **is** END ITERATE [*iterate-construct-name*]

C1143a (R1139a) If the *iterate-stmt* of an *iterate-construct* specifies an *iterate-construct-name*, the corresponding *end-iterate-stmt* shall specify the same *iterate-construct-name*. If the *iterate-stmt* of an *iterate-construct* does not specify an *iterate-construct-name*, the corresponding *end-iterate-stmt* shall not specify an *iterate-construct-name*.

C1143b (R1139c) If = appears and ALLOCATABLE does not appear, *array-spec* shall specify explicit shape. If ALLOCATABLE appears or => appears, *array-spec* shall specify deferred shape.

C1143c (R1139c) If = appears, the type, type parameters, and rank of *variable* or *variable-name* shall conform to those of the result of *iterator-reference* in the same way that those of *variable* and *expr* are required to conform in an intrinsic *assignment-stmt*.

C1143d (R1139c) If => appears, the type, type parameters, and rank of *data-pointer-object* or *variable-name* shall conform to those of the result of *iterator-reference* in the same way that those of *data-pointer-object* and *data-target* are required to conform in a *pointer-assignment-stmt*.

C1143e (R1139c) The *variable* shall not be a coindexed object or a subobject of a coindexed object.

C1143f (R1139c) If *declaration-type-spec* appears it shall specify the same declared type and kind type parameters as the result of *iterator-reference*, and shall not specify any assumed length type parameters.

C1143g (R1139c) If => appears, either *declaration-type-spec* shall appear, or *data-pointer-object* shall have the POINTER attribute.

C1143h (R1139c) If CONCURRENT appears, *declaration-type-spec* shall appear.

C1143j (R1139a) If CONCURRENT appears, the construct shall not contain an EXIT statement that belongs to the construct or an outer construct, a CYCLE statement that belongs to an outer construct, or a branching statement that has a branch target that is not the END ITERATE statement or a statement within the block of the construct.

C1143k (R1139d) The same *iterate-attrib* shall not appear more than once.

R1520a *iterator-reference* **is** *procedure-designator* ([*actual-arg-spec-list*])

C1524a (R1520a) The *procedure-designator* shall designate an iterator.

C1524b (R1520a) The *procedure-designator* shall not be a coindexed object or a subobject of a coindexed object.

If *declaration-type-spec* appears, it specifies the type and type parameter values of the *variable-name*, and *variable-name* is a construct entity of the ITERATE construct. If => also appears it has the pointer attribute, and this may be confirmed by the appearance of POINTER. If = appears the *variable-name* may be declared to have the ALLOCATABLE or TARGET attribute. It does not have any additional attributes.

2.4.4 ITERATE construct and iterator execution semantics

When the *iterate-stmt* of an ITERATE construct is executed the construct becomes active. If the *procedure-designator* in *iterator-reference* is a pointer, it shall be associated with an iterator. The values of the nondeferred length parameters of *variable*, *variable-name*, or *data-pointer-object* shall be the same as corresponding parameters of the result of *iterator-reference*.

When an *iterate-stmt* is executed, the following occur in the specified order:

1. Argument associations are established.
2. An instance of the iterator is associated with the *iterate-stmt*; it is not represented by an instance variable
3. The iterator is invoked.
4. An activation record is created for the instance by evaluating expressions within the specification part of the iterator and elaborating the specification part.
5. Execution of the iterator begins with its first executable construct.

While the construct is active, the following occur in the specified order:

1. If = appears the iterator result value is assigned to *variable* or *variable-name* as if by an assignment statement; if => appears the result value is assigned to *data-pointer-object* or *variable-name* as if by pointer assignment.

   NOTE 2.7

   Because the assignment of the result of *iterator-reference* to *variable* or *variable-name* is as if by an assignment statement, it might cause finalization of *variable*, invocation of defined assignment, or allocation or reallocation of an allocatable *variable*.

2. The *block* of the ITERATE construct is executed.

3. The instance of the iterator is resumed by re-establishing argument associations and transferring control to the first executable construct after the SUSPEND statement whose execution suspended its execution. Expressions in the specification part are not re-evaluated and the specification part is not elaborated again. Therefore, local variables, including automatic variables, of the instance retain the same bounds, length parameter values, definition status, and values if any, that they had when the instance was suspended.

   NOTE 2.8

   Because argument associations are re-established, dummy arguments might have different extents, length parameter values, allocation status, pointer association status, or values (if any).

Invoking or resuming the iterator, assigning or associating its result, and executing the *block*, is an iteration. If *declaration-type-spec* appears, each iteration has a different instance of *variable-name*.

An iterator terminates when it executes a RETURN, END, or SUSPEND statement, or completes execution of the final executable construct of its *execution-part*.

If CONCURRENT appears, the processor may invoke and resume the iterator, and assign its value, in the sequence of execution that began execution of the construct, and then execute each corresponding block in a separate sequence of execution. Alternatively, it may invoke and resume the iterator, assign its value, and execute the corresponding block, in a separate sequence of execution for each iteration. The processor shall ensure that when the iterator is invoked or resumed, no other iteration of the same execution of the construct resumes the construct's instance of the iterator until it terminates. In either case, the separate sequences of execution may be executed in any order, or concurrently.

NOTE 2.9

If the processor chooses to invoke or resume the iterator, assign values to instances of *variable-name*, and execute corresponding blocks, independently within separate sequences of execution, instead of invoking and resuming the iterator within the sequence of execution that initiated the construct, this effectively requires an iterator to be a monitor procedure, or that invoking or resuming it is protected as if by a critical section.

Because the *variable-name* is a construct entity, if it is allocatable, it is not allocated before the iterator is invoked, and it becomes deallocated at the end of each iteration. The *variable* is not a construct entity.

When the iterator terminates, a value is not assigned to *variable* or *variable-name*, or associated with *data-pointer-object*. If the result variable is allocatable, it shall be deallocated before the iterator terminates. Whether a non-allocatable result variable is finalized is processor dependent.

NOTE 2.10

Because an iterator is allowed but not required to have assigned a value to its result variable when it terminates, requiring a processor to finalize the result variable would require the processor to keep track of its definition status.

If CONCURRENT does not appear, execution of an ITERATE construct completes, the activation record of the iterator instance is destroyed, the iterator instance ceases to exist, and the construct becomes inactive when

- the iterator terminates,
- an EXIT statement that belongs to the ITERATE construct is executed,
- an EXIT or CYCLE statement that belongs to an outer construct and is within the range of the ITERATE construct is executed,
- a branch occurs from a statement within the range of the ITERATE construct to a statement that is neither the *end-iterate-stmt* nor within the range of the ITERATE construct, or
- a RETURN or STOP statement within the ITERATE construct is executed.

If CONCURRENT appears, execution of an ITERATE construct completes, the activation record of the iterator instance is destroyed, the iterator instance ceases to exist, and the construct becomes inactive when the iterator terminates and execution of all iterations is completed.

When execution of the ITERATE construct completes, if *declaration-type-spec* does not appear

- if = appears and *block* was executed, the value of *variable* is the value assigned by the ITERATE statement before the final execution of *block*, or assigned during the final execution of *block*; otherwise its definition status and value (if any) are the same as before execution of the ITERATE construct, or
- if => appears and *block* was executed, the association status of *data-pointer-object* is as established by the ITERATE statement before the final execution of *block*, or established during the final execution of *block*; otherwise its association status is the same as before execution of the ITERATE construct.

NOTE 2.11

The *variable* might become undefined during the final execution of *block*. The association status of *data-pointer-object* might become undefined during the final execution of *block*.

2.4.5 Restrictions on DO CONCURRENT constructs

Subclause 11.1.7.5 of ISO/IEC 1539-1:2019(E) concerning restrictions on DO CONCURRENT constructs is revised to apply to ITERATE CONCURRENT constructs as well.

2.5 VALUE attribute

The VALUE attribute shall be allowed for a dummy coroutine or iterator.

2.6 PRESENT (A)

The PRESENT intrinsic function inquires whether an optional dummy argument is associated with an actual argument in a function or iterator reference, a CALL statement, or a RESUME statement.

3 Examples

3.1 Quadrature example

This subclause presents four examples of a simple quadrature procedure. One uses forward communication, two use reverse communication without coroutine syntax, and the fourth uses reverse communication with coroutine syntax.

3.1.1 Forward communication example

```
  subroutine INTEGRATE ( A, B, ANSWER, ERROR, FUNC )
    real, intent(in) :: A, B ! Bounds of the integral
    real, intent(out) :: ANSWER, ERROR
    interface
      real function FUNC ( X )
        real, intent(in) :: X
      end function FUNC
    end interface
    real, parameter :: ABSCISSAE(...) = [ ... ]
    real, parameter :: WEIGHTS(...) = [...]
    integer :: I
    answer = weights(1) * func( 0.5*(b+a) )
    do i = 2, size(weights)
      answer = answer + weights(i) * func( 0.5*(b+a) + (b-a) * abscissae(i) )
      answer = answer + weights(i) * func( 0.5*(b+a) - (b-a) * abscissae(i) )
    end do
    answer = ( b - a ) * answer
    error = ...
  end subroutine INTEGRATE
```

3.1.2 First reverse communication example

This example uses computed GO TO to resume computation after each integrand value is computed. Notice that the DO construct cannot be used because computation needs to be resumed within the construct. Further, this subroutine is not thread safe.

```
  subroutine INTEGRATE ( A, B, ANSWER, ERROR, WHAT )
    real, intent(in) :: A, B ! Bounds of the integral
    real, intent(inout) :: ANSWER, ERROR
    integer, intent(inout) :: WHAT
    real, parameter :: ABSCISSAE(...) = [ ... ]
    real, parameter :: WEIGHTS(...) = [...]
    real, save :: RESULT
    integer, save :: I
    go to ( 10, 20, 30 ), what
    i = 1
    answer = 0.5 * ( a + b )
    what = 1
    return
10  result = answer * weights(1)
11    i = i + 1
      if ( i > size(weights) ) then
        what = 0
```

```
    answer = ( a - b ) * result
    error = ...
    return
   end if
   answer = 0.5*(b+a) + (b-a) * abscissae(i)
   what = 2
   return
20 result = result + weights(i) * answer
   answer = 0.5*(b+a) - (b-a) * abscissae(i)
   what = 3
   return
30 result = result + weights(i) * answer
  go to 11
 end subroutine INTEGRATE
```

This subroutine is used as follows:

```
 what = 0
 do
  call integrate ( a, b, answer, error, what )
  if ( what == 0 ) exit
  ! evaluate the integrand at ANSWER and put the value into ANSWER
 end do
 ! Integral is in ANSWER here
```

3.1.3 Second reverse communication example

This example avoids GO TO statements and statement labels by structuring the quadrature subroutine as a “state machine.” The state indicates how to resume computation after each integrand value is computed. Although a DO construct can be used, control flow is difficult to follow because it is controlled by the state variable. This subroutine is also not thread safe.

```
 subroutine INTEGRATE ( A, B, ANSWER, ERROR, WHAT )
  real, intent(in) :: A, B ! Bounds of the integral
  real, intent(inout) :: ANSWER, ERROR
  integer, intent(inout) :: WHAT
  real, parameter :: ABSCISSAE(...) = [ ... ]
  real, parameter :: WEIGHTS(...) = [...]
  real, save :: RESULT
  integer, save :: I
  do
   select case ( what )
   case ( 0 )
    i = 1
    answer = 0.5 * ( a + b )
    what = 1
    return
   case ( 1 )
    result = weights(1) * answer
    what = 2
   case ( 2 )
    i = i + 1
    if ( i > size(weights) ) then
```

```
          what = 0
          answer = ( a - b ) * result
          error = ...
          return
        end if
        answer = 0.5*(b+a) + (b-a) * abscissae(i)
        what = 3
        return
      case ( 3 )
        result = result + weights(i) * answer
        answer = 0.5*(b+a) - (b-a) * abscissae(i)
        what = 4
        return
      case ( 4 )
        result = result + weights(i) * answer
        what = 2
      end select
    end do
  end subroutine INTEGRATE
```

This example is used the same way as the previous example.

3.1.4 Example using a coroutine

The coroutine organization is much clearer than the previous two examples.

```
  coroutine INTEGRATE ( A, B, ANSWER, ERROR )
    real, intent(in) :: A, B ! Bounds of the integral
    real, intent(out) :: ANSWER, ERROR
    real, parameter :: ABSCISSAE(...) = [ ... ]
    real, parameter :: WEIGHTS(...) = [...]
    integer :: I
    answer = 0.5*(b+a)
    suspend
    result = answer * weights(1)
    do i = 2, size(weights)
      answer = 0.5*(b+a) + (b-a) * abscissae(i)
      suspend
      result = result + answer * weights(i)
      answer = 0.5*(b+a) - (b-a) * abscissae(i)
      suspend
      result = result + answer * weights(i)
    end do
    answer = ( b - a ) * result
    error = ...
  end subroutine INTEGRATE
```

This coroutine is used as follows:

```
  call integrate ( a, b, answer, error )
  do while ( suspended(integrate) )
    ! Evaluate the integrand at ANSWER and put the value into ANSWER
    resume integrate ( a, b, answer, error )
```

```
  end do
  ! Integral is in ANSWER here
```

3.2 Iterator for a queue

This example performs a breadh-first traversal of a binary tree. It illustrates that the *block* of an ITERATE construct might change the object that is the attention of its iterator. Whether this “makes sense” in the general case is the responsibility of the iterator and other procedures that act on its arguments, or variables to which it has access by use or host association; it is not the responsibility of the processor or the standard.

```
  type :: Tree_Node_t
    class(tree_node_t), pointer :: LeftSon => NULL(), RightSon => NULL()
  end type Tree_Node_t

  class(tree_node_t), pointer :: Root => NULL()

  type :: Queue_Element_t
    class(*), pointer :: Thing => NULL()
    class(queue_element_t), pointer :: Next => NULL()
  end type Queue_Element_t

  type :: Queue_t
    class(queue_element_t), pointer :: Head => NULL(), Tail => NULL()
  contains
    procedure :: DeQueue
    procedure :: EnQueue
  end type Queue_t

  type(queue_t) :: MyQueue

  call Fill_The_Tree ( root )
  call myQueue%enQueue ( root ) ! Doesn't enqueue if root is NULL()
  iterate ( class(*) :: node => myQueue%deQueue() )
    ! This is an example where it ought to be possible to invoke (or resume) a
    ! type-bound iterator (or function) that has no arguments other than
    ! the passed-object argument without ().
    select type ( node )
    class ( tree_node_t )
      call node%processIt
      call myQueue%enQueue ( node%leftSon )
      call myQueue%enQueue ( node%rightSon )
    end select
  end iterate

  contains

    iterator DeQueue ( TheQueue ) result ( Thing )
      class(queue_t), intent(inout) :: TheQueue
      class(*), pointer :: Thing
      class(queue_element_t), pointer :: This
      do
        this => theQueue%head
        if ( .not. associated(this) ) return ! terminate ITERATE construct
```

```
        thing => this%thing
        theQueue%head => this%next
        deallocate ( this )
        suspend ! Process Thing and come back here
      end do
    end iterator DeQueue

    subroutine Enqueue ( TheQueue, Thing )
      class(queue_t), intent(inout) :: TheQueue
      class(*), intent(in), pointer :: Thing
      class(queue_element_t), pointer :: This
      if ( associated(thing) ) then
        allocate ( this )
        this%thing => thing
        if ( associated(theQueue%tail) ) then
          theQueue%tail%next => this
        else
          theQueue%head => this
        end if
        theQueue%tail => this
      end if
    end subroutine Enqueue
```

3.3 Preserving automatic variables

If one needs to invoke a procedure to solve several differently-sized problems, and the expense of creating local automatic variables is significant, it can be posed as a coroutine and then invoked initially in such a way as to create its automatic variables with the maximum extents necessary for the entire spectrum of problems to be solved. It can then be suspended, which does not destroy its automatic variables. When it is resumed to solve each problem, the automatic variables are intact.

3.4 Relationship to exception handling

If exception handling is provided as described in J3/23-106, using an exception type defined in the ISO_Fortran_Env module, additional exception identifiers will be needed.

4 Required editorial changes to ISO/IEC 1539-1:2019(E)

To be provided in due course.

