Document Type: Defect Report


Document Source: WG 5 Convener (J. Reid)

Document Status: This document is circulated for review and consideration. Please take it into consideration when voting on SC 22 N 4345, Technical Corrigenda 4 for ISO/IEC 1539-1.

Action ID: ACT

Due Date: 

No. of Pages: 28
Defect reports that led to Corrigendum 4 and responses to them

Defect Type T:

<table>
<thead>
<tr>
<th>T number</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>I F95/0074</td>
<td>TARGET dummy arguments and POINTER expressions</td>
</tr>
<tr>
<td>I F95/0102</td>
<td>mask-expr evaluated only once</td>
</tr>
<tr>
<td>E F03/0003</td>
<td>Referencing deferred bindings</td>
</tr>
<tr>
<td>E F03/0004</td>
<td>Type-bound procedures and undefined association status</td>
</tr>
<tr>
<td>E F03/0049</td>
<td>Separators in list-directed output involving UDDTIO</td>
</tr>
<tr>
<td>I F03/0073</td>
<td>C interoper of dummy procedures</td>
</tr>
<tr>
<td>E F03/0074</td>
<td>Type mismatch for C character arguments</td>
</tr>
<tr>
<td>E F03/0075</td>
<td>C interoper of derived types with array components</td>
</tr>
<tr>
<td>I F03/0076</td>
<td>Scope of Fortran names of procedures with binding labels</td>
</tr>
<tr>
<td>E F03/0077</td>
<td>LBOUND of array structure component</td>
</tr>
<tr>
<td>I F03/0081</td>
<td>F edit descriptor with field width 0</td>
</tr>
<tr>
<td>I F03/0082</td>
<td>VALUE in place of INTENT for pure procedure dummy arguments</td>
</tr>
<tr>
<td>I F03/0087</td>
<td>Entry names as dummy procedure arguments</td>
</tr>
<tr>
<td>E F03/0098</td>
<td>Does allocate with source= define subcomponents?</td>
</tr>
<tr>
<td>E F03/0099</td>
<td>Clause 16 does not account for volatile variable</td>
</tr>
<tr>
<td>C F03/0102</td>
<td>Evaluation of bound-expr in data pointer assignment</td>
</tr>
<tr>
<td>E F03/0109</td>
<td>Referencing deferred binding via absent dummy argument</td>
</tr>
<tr>
<td>E F03/0111</td>
<td>Is defined assignment to a vector subscripted array allowed?</td>
</tr>
<tr>
<td>E F03/0113</td>
<td>Size and uniqueness considerations for ID=</td>
</tr>
<tr>
<td>E F03/0114</td>
<td>Inconsistent restrictions on i/o specifiers</td>
</tr>
<tr>
<td>E F03/0117</td>
<td>STOP executed via function in input/output list</td>
</tr>
</tbody>
</table>

---

NUMBER: F95/0074
TITLE: TARGET dummy arguments and POINTER expressions
KEYWORDS: TARGET, POINTER, dummy arguments
DEFECT TYPE: Interpretation
STATUS: Passed by WG5 ballot

see also F95 interp 000031

QUESTION:

Consider the following program.

```fortran
PROGRAM questionable
REAL, TARGET :: x
CALL s1(f(x))
CALL s2(f(x))
CALL s3(f(x))
CONTAINS
FUNCTION f(a)
REAL, POINTER :: f
REAL, TARGET :: a
f => a
END FUNCTION
SUBROUTINE s1(variable)
variable = 42 ! statement 1
END SUBROUTINE
SUBROUTINE s2(variable)
INTENT(OUT) variable
variable = 42 ! statement 2
END SUBROUTINE
```

Page 1
Is this program standard-conforming, and if so, what value is printed?

The real question is whether an expression argument that is a pointer function reference is treated as a variable (data-object) argument with the variable being the target of the pointer expression. (Or whether it is dereferenced as usual in the absence of POINTER dummy arguments).

Re (statement 1), the question is whether VARIABLE is definable when argument-associated with “F()”.

Re (statement 2), if the previous answer was Yes (VARIABLE is definable), then presumably it can be made INTENT(OUT). A random sample of 4 compilers revealed that they considered it not to be definable.

Re (statement 3), the question is whether P is pointer-associated with X, or processor-dependent. Of the same random sample 3 thought it was associated with X, 1 thought not.

ANSWER:

These answers are in the context of Fortran 2003.

1. The call to s1 is not standard conforming. 5.1.2.7 states
   "If no INTENT attribute is specified for a dummy argument, its use
   is subject to the limitations of the associated actual argument."
   The associated actual argument is the function reference f(x); this
   is an expression, not a variable, and therefore not definable,
   because only variables can be defined (in the sense of being given
   a value, see 2.4.3.1.1 and 2.5.5).

2. The call to s2 is not standard conforming. 12.4.1.2 says:
   "If a nonpointer dummy argument has INTENT(OUT) or INTENT(INOUT),
   the actual argument shall be definable."
   Again the actual argument is a function reference, which is not
   definable.

3. The call to s3 is standard conforming and the pointer assignment in
   s3 causes p and x to become associated. This may be deduced from
   12.4.1.2, which says [270:5-9]
   "If the dummy argument has the TARGET attribute, does not have
   the VALUE attribute, and is either a scalar or an assumed-shape
   array, and the corresponding actual argument has the TARGET
   attribute but is not an array section with a vector subscript
   then
   (1) Any pointers associated with the actual argument become
   associated with the corresponding dummy argument on
   invocation of the procedure ...”
QUESTION:

Is a mask-expr required to be evaluated in a WHERE such as:

WHERE ( (/ .TRUE., .TRUE. /) )
A = A + 1
ELSEWHERE (MASKF())
A = MAX
END WHERE

where an optimizing compiler might notice that the ELSEWHERE branch will never be executed?

Page 112, line 38 says of the mask expression in ELSEWHERE statements

"The mask-expression is evaluated only once."

ANSWER:

The above quote is from the Fortran 95 standard. In Fortran 2003 the mask expression in an ELSEWHERE is not required to be evaluated, and this is made clear by the fourth paragraph of 7.4.3.2 (at [147:7]) which, referring to a masked ELSEWHERE statement, states

"The <mask-expr> is evaluated at most once."

This also applies to the mask expressions of nested WHERE construct statements; this is made clear by the last sentence of the seventh paragraph of 7.4.3.2 (at [147:16]) which also states that it is evaluated "at most once."

However, this does not apply to the mask expression of the outermost WHERE construct statement, which must be evaluated exactly once; this is made clear by the last statement of the first paragraph of 7.4.3.2 (at [147:1]) which states that this mask expression is evaluated "only once." That does not guarantee invocation of functions in that expression if they are not required to determine its value.

EDITS:
QUESTION:

I thought that the intent was that it would be impossible to reference a deferred binding. However, it doesn't appear to me that this intent was achieved. Consider the following program (Sorry, but I don't have any compilers up to syntax-checking this).

```fortran
module defer
  type, abstract :: t
  contains
    procedure (sub), nopass, deferred :: deferred_proc
  end type t
  type, extends(t) :: t2
  contains
    procedure, nopass :: deferred_proc => sub2
  end type t2
contains
  subroutine sub
    write (*, *) 'Hello.'
  end subroutine sub
  subroutine sub2
    write (*, *) 'Goodbye.'
  end subroutine sub2
end module defer

program p
  use defer
  class(t), pointer :: x
  nullify(x)
call x%deferred_proc
end program p
```

Is this a valid program? If not, what restriction of the standard does it violate?

Note that x%deferred_proc does not require the value of x (4.5.7) and thus is not a reference to x (2.5.6). Therefore, [83:23-24] does not prohibit this. Nor is it clear that there is an intent to prohibit invocation of type-bound procedures for disassociated pointer objects; except in the case of deferred bindings, this
seems well-defined and potentially useful.

Because x is disassociated, its dynamic type is the same as its declared type, thus making the interpretation of x%nondeferred_proc reasonably clear.

ANSWER:

No, this was not intended to be a valid program. A type-bound procedure may not be invoked through an undefined pointer, a disassociated pointer, or an unallocated allocatable variable. An edit is supplied to clarify this situation. The same answer and edit also apply to F03/0004.

EDITS:

All edits refer to 04-007.

Insert after [04-007: 266:24]:

"The <data-ref> shall not be an unallocated allocatable variable or a pointer whose association status is disassociated or undefined."

Note: this is the same edit as interp F03/0004.

SUBMITTED BY: Richard Maine

HISTORY: 04-322 m169 F03/0003 Submitted
04-322r1 m169 Passed by J3 meeting
04-418r1 m170 Subsumed by interp F03/0004
05-180 m172 Failed WG5 ballot N1617 - the edit is subsumed by F03/0004
07-280 m182 Revised
07-280r1 m182 Passed by J3 meeting
08-133r2 m183 Passed by letter ballot #15 08-101
08-164r1 m184 Passed WG5 ballot #5 N1722-N1726 with a changed edit

NUMBER: F03/0004
TITLE: Type-bound procedures and undefined association status
KEYWORDS: Type-bound procedure, dynamic type
DEFECT TYPE: Erratum
STATUS: Passed by WG5 ballot

QUESTION:

It appears that the dynamic type is undefined for a pointer with undefined association status. This impacts type-bound procedures. Consider the following program.

```fortran
module undefined

  type :: t
  contains
    procedure, nopass :: nondeferred_proc => sub
  end type t

  type, extends(t) :: t2
  contains
    procedure, nopass :: nondeferred_proc => sub2
  end type t2

end module undefined
```
contains
 subroutine sub
  write (*,*) 'Hello.'
 end subroutine sub
 subroutine sub2
  write (*,*) 'Goodbye.'
 end subroutine sub2
end module undefined

program p
  use undefined
  class(t), pointer :: x
  call x%nondeferred_proc
end program p

Is this a valid program? If not, what restriction of the standard does it violate? If so, what does it print.

Note that x%nondeferred_proc does not require the value of x (4.5.7) and thus is not a reference to x (2.5.6). Therefore, [83:23-24] does not prohibit this.

If x were disassociated, its dynamic type would be t and the interpretation of this would be reasonably clear.

However, the standard does not appear to specify the dynamic type of x when its association status is undefined. Nor can I find any prohibition that applies to this case.

ANSWER:

No, the program is not valid, because the standard does not establish an interpretation of it. An edit is supplied to clarify this.

Furthermore, the case with a disassociated pointer was not intended to be valid. An edit is supplied to correct this oversight.

DISCUSSION:

Access to object-bound procedures (a.k.a. procedure pointer components) always require there to be an object. Access to type-bound procedures of an object was intended to require this too, but the effect of the NOPASS attribute on this was overlooked.

EDITS:

All edits refer to 04-007.

Insert after [04-007: 266:24]:

"The <data-ref> shall not be an unallocated allocatable variable or a pointer whose association status is disassociated or undefined."

Note: this is the same edit as interp F03/0003.

SUBMITTED BY: Richard Maine

HISTORY: 04-323 m169 F03/0004 submitted
04-323r1 m169 Passed by J3 meeting
04-418r1 m170 Passed J3 letter ballot #9
05-180 m172 Failed WG5 ballot N1617
Consider the following program:

```fortran
MODULE m
  TYPE t
    INTEGER i
  END TYPE
  INTERFACE WRITE(FORMATTED)
    MODULE PROCEDURE formattedWriteT
  END INTERFACE
  CONTAINS
    SUBROUTINE formattedWriteT(dtv, unit, iotype, v_list, iostat, iomsg)
      CLASS(t),     INTENT(IN)    :: dtv
      INTEGER,     INTENT(IN)    :: unit
      CHARACTER(LEN=*) , INTENT(IN) :: iotype
      INTEGER,     INTENT(IN)    :: v_list(:)
      INTEGER,     INTENT(OUT)   :: iostat
      CHARACTER(LEN=*) , INTENT(INOUT) :: iomsg
      WRITE(unit, *) dtv%i, 'a'
    END SUBROUTINE
  END INTERFACE
END MODULE

PROGRAM foo
  USE m
  TYPE(t) :: t1 = t(5)
  OPEN(10, FILE='foo.txt', ACCESS='SEQUENTIAL', FORM='FORMATTED', &
       DELIM='NONE')
  WRITE(10, *) 'xyz', t1, 'zyx'
END PROGRAM
```

10.9.2 of Fortran 2003 states that character sequences produced for list-directed output are not separated from each other by value separators when the delimiter mode is NONE. The implication of this is obvious when the adjacent effective output list items are both of character type. But when user-defined derived-type input/output is involved, it is much less clear whether a separator should be included in the output.

In the example given, it is unclear whether the output should be:

```
xyz 5 azyx
```

or:

```
xyz 5 a zyx
```

1. Should a separator be inserted between two non-delimited character sequences when one of the character sequences is written by a child data transfer statement, and the other is written by a parent data transfer statement, where both statements are list-directed?
2. Should a separator be inserted between two non-delimited character sequences when the character sequences are written by two different child data transfer statements, where both statements are list-directed?

3. Should a separator be inserted between two character sequences when one of the character sequences is written by a child data transfer statement, and the other is written by a parent data transfer statement, where one of the statements is list-directed and the other is format-directed?

4. Should a separator be inserted between two character sequences when the character sequences are written by two different child data transfer statements, where one of the statements is list-directed and the other is format-directed?

There is also the further question of whether, in the case of recursive i/o, this is affected by intervening i/o operations to a different unit. For example, in the modified program:

```
MODULE m2
  TYPE t
    INTEGER i
  END TYPE

  INTERFACE WRITE(FORMATTED)
    MODULE PROCEDURE formattedWriteT
  END INTERFACE

  CONTAINS

  SUBROUTINE formattedWriteT(dtv, unit, iotype, v_list, iostat, iomsg)
    CLASS(t), INTENT(IN) :: dtv
    INTEGER, INTENT(IN) :: unit
    CHARACTER(LEN=*) , INTENT(IN) :: iotype
    INTEGER, INTENT(IN) :: v_list(:)
    INTEGER, INTENT(INOUT) :: iostat
    CHARACTER(LEN=*) , INTENT(INOUT) :: iomsg

    CHARACTER(100) temp
    WRITE (temp,'(I0)') dtv%i
    WRITE (unit,*) TRIM(temp)
  END SUBROUTINE

END MODULE

PROGRAM foo2
  USE m2
  TYPE(t) :: t1 = t(5)
  OPEN(10, FILE='foo.txt', ACCESS='SEQUENTIAL', FORM='FORMATTED', DELIMIT='NONE')
  WRITE(10, *) 'xyz', t1, 'zyx'
END PROGRAM
```

5. Does the second example write
   xyz 5xyz
or
   xyz5xyz

That is, in this case, does the data transfer which is an internal
6. Similarly, if the outer list-directed WRITE were to an internal file does an intervening external file i/o operation affect whether the character sequences are considered adjacent (i.e. the opposite way around to the second example)?

ANSWER:

1) No. It is the intent of the standard (10.9.2) that when both the parent and child data transfer statements are both list-directed output statements, or both are namelist output statements, the processor treats the first list item appearing in a child data transfer statement as if that list item had immediately followed the last list item processed by the parent data transfer statement, as long as no other data transfers to that unit occurred in between the processing of those two list items. Therefore, in this case, the two character sequences are considered adjacent.

2) No. It is the intent of the standard (10.9.2) that when two different child data transfer statements are both list-directed output statements, or both namelist output statements, they write to the same unit, and no other data transfers to that unit occur in between the two child data transfer statements, the processor treats the first list item appearing in the second child data transfer statement as if that list item had immediately followed the last list item processed by the first child data transfer statement. Therefore, in this case, the two character sequences are considered adjacent.

3) It is processor dependent whether or not a separator appears between two such character sequences. In section 10.9.2, the phrase "adjacent undelimited character sequences" refers to character sequences produced by list-directed output. When one of the sequences is written by a child or parent output statement that is not list-directed, the exception described in the first paragraph of 10.9.2 does not apply. The other rules for inserting optional blanks around values in list-directed output allow the processor to insert optional leading and trailing blanks around a list item. The standard does not specify when optional blanks are written; therefore, when two adjacent list items (the values thereof) are written to an output record, and only one of them was written by list-directed input/output, the standard does not specify whether or not any optional blanks appear between those values in the output record.

4) It is processor dependent whether or not a separator appears between two such character sequences. See answer 3.

5) The result should be "xyz5xyz", that is, the recursive i/o to the internal file does not affect whether the character sequences being written by the outer data transfer statement are considered adjacent.

6) No, if an outer data transfer is to an internal file, then a recursive i/o operation to a different internal file or an external file does not affect whether the character sequences are considered adjacent.

NOTE TO J3:

In Fortran 2008, recursive i/o operations to a different external unit...
are allowed and these also ought not to affect whether the character sequences written by the outer data transfer statement are considered to be adjacent. The edit to F2003 below will produce that result when applied to the (draft) F2008.

EDITS:

[241:5] In Section 10.9.2, add the following to the end of the first paragraph:

"Two undelimited character sequences are considered adjacent when both were written using list-directed input/output, no intervening data transfer or input/output file positioning operations on that unit occurred, and both were written either by a single data transfer statement, or during the execution of a parent data transfer statement along with its child data transfer statements."

SUBMITTED BY: Rob James

HISTORY: 05-140  m171  F03/0049 submitted
05-140r1 m171 Passed by J3 meeting
05-170  m172 Failed J3 letter ballot #11
06-367r1 m178 Passed by J3 meeting
07-272  m181 Passed as changed by J3 letter ballot #13
08-155  m184 Failed WG5 ballot #4 N1711-N1721
08-182r1 m184 Revised answer - Passed by J3 meeting
08-259  m185 Passed by J3 letter ballot #17 08-213
N1770  m187 Passed WG5 ballot 6 N1764-N1765 as revised: this edit should come before the Corrigendum 3 edit in the same place

---------------------------------------------------------------------

NUMBER: F03/0073
TITLE: C interop of dummy procedures
KEYWORDS: Interoperability, dummy procedures
DEFECT TYPE: Interpretation
STATUS: Passed by WG5 ballot

QUESTION:

Is the following interface interoperable, and if yes, what is the interoperable C prototype?

```c
subroutine my_sub(my_fun) bind(c, name="My_Proc")
  interface
    function my_fun(x) bind(c)
      use iso_c_binding
      integer(c_int) :: my_fun
      real(c_double), value :: x
    end function
  end interface
end subroutine
```

ANSWER:

Yes, this interface is interoperable with the following prototype:

```c
void My_Proc(int (*his_fun)(double));
```

Its formal parameter is of a pointer type, and the referenced type is a function that is interoperable with my fun, thus satisfying the requirements in item 5 in the list in subclause 15.2.6.
NUMBER: F03/0074
TITLE: Type mismatch for C character arguments
KEYWORDS: Interoperability, dummy procedures
DEFECT TYPE: Erratum
STATUS: Passed by WG5 ballot

QUESTION:
Is the following conforming if the C character kind (C_CHAR) is not the same as the default character kind (KIND('A'))?

```
use iso_c_binding
interface
  subroutine sub(string) bind(c)
    import C_Char
    character(kind=C_Char) :: string(*)
  end subroutine
end interface

character(kind=C_Char, len=10) :: string
call sub(string)
```

The second and 3rd paras of 12.4.1.2 do not treat C character kind the same as default character kind, and seem to require matching type parameter values between the actual and dummy. This makes Note 15.23 nonconforming if C_CHAR is not the same as default character kind, and is also not in sync with 270:25-26.

ANSWER:
Yes, the program fragment is standard-conforming. An edit is provided to correct the oversight in 12.4.1.2.

EDITS:

[269:3] After "of type default character",
  Insert "or of type character with the C character kind (15.1)".

[269:5,7-8] After each "of type default character",
  Insert "or of type character with the C character kind", twice.
Is the following C struct:

typedef struct
{
  float x[5];
} array;

 interoperable with this Fortran type:

type, bind(c) :: array
  real(c_float) :: x(3)
end type

ANSWER:

No. According to the words in 15.2.3, the answer is yes, since only interoperability of the types of the C and Fortran components is required. This is, however, an oversight. An edit is provided to correct this.

EDITS:

In the first sentence of the first paragraph after NOTE 15.12 in subclause 15.2.3 [04-007:398:11-12], replace “have types and type parameters that are interoperable with the types of the corresponding components of the struct type” with “would interoperate with corresponding components of the C struct type as described in 15.2.4 and 15.2.5 if the components were variables”.

SUBMITTED BY: Aleksandar Donev

HISTORY: 06-106 m175 F03/0075 submitted
  08-191r1 m184 Revised answer - Passed by J3 meeting
  08-259 m185 Passed by J3 letter ballot #17 08-213
N1770 m187 Passed WG5 ballot 6 N1764-N1765

------------------------------------------------------------------------

NUMBER: F03/0076
TITLE: Scope of Fortran names of procedures with binding labels
KEYWORDS: Interoperability, dummy procedures
DEFECT TYPE: Interpretation
STATUS: Passed by WG5 ballot

QUESTION:

Is the following conforming:

module A
  interface
    subroutine CProc() bind(C, name="AnExternalCProcedure")
  end subroutine
  end interface
end module


Page 12
module B
    interface
        subroutine CSub() bind(C, name="AnExternalCProcedure")
    end subroutine
    end interface
end module

ANSWER:

No, the fragment is not conforming. The name of an external procedure is a global entity and is required to be unique. That is, the external C procedure AnExternalCProcedure, must be given a unique Fortran name inside a Fortran program.

EDITS:

None.

SUBMITTED BY: Aleksandar Donev

HISTORY: 06-107 m175 F03/0076 submitted
08-187 m184 Passed by J3 meeting
08-259 m185 Passed by J3 letter ballot #17 08-213
N1770 m187 Passed WG5 ballot 6 N1764-N1765

------------------------------------------------------------------

NUMBER: F03/0077
TITLE: LBOUND of array structure component
KEYWORDS: LBOUND, lower bounds, bounds, structure component, array sections
DEFECT TYPE: Erratum
STATUS: Passed by WG5 ballot

QUESTION:
Given the declarations

```
TYPE t
    REAL x
    REAL y(-5:5)
END TYPE
```

```
TYPE(t) z(10:20)
```

What is the result of the following LBOUND references:

```
LBOUND(z(10)%y)
LBOUND(z%x)
```

The confusion arises because the existing definition of LBOUND uses the undefined term "array structure component" in determining whether to return the lower bound in the <array-spec> or 1. It seems clear that Z(10)%Y must be an array structure component (so the answer ought to be -5) but it is not clear whether Z%x is one and therefore whether the result of that LBOUND ought to be 1 or 10.

ANSWER:

Yes, the result of the first LBOUND is indeed -5.
The result of the second LBOUND is 1.
Clarifying edits are provided.

The description of the intrinsic function UBOUND suffers from the same
EDITS:

To avoid the undefined term, or any long phrase, the edit changes the definition of "whole array" to include the case of a whole array component of a structure.

[107:2-3] Replace the existing definition which is
"A <<whole array>> is a named array, which may be either a named constant (5.1.2.10, 5.2.9) or a variable; no subscript list is appended to the name."

with
"A <<whole array>> is a named array or a structure component whose final <part-ref> is an array component name; no subscript list is appended."

{Make "whole array" include the whole array component case.}

[107:7-8] Replace "whole array name" with "whole array designator", twice.

[326:8] After "a whole array" delete "or array structure component". {No longer need kludgy wording in LBOUND.}

[358:6-7] After "a whole array" delete "or array structure component". {No longer need kludgy wording in UBOUND.}

[436:36] After "named array" insert "or array component of a structure, with no subscript list."
{Fix the glossary.}

SUBMITTED BY: Aleksandar Donev

HISTORY: 06-118 m175 F03/0077 submitted 08-174 m184 Revised with edits 08-174r1 m184 Fixed typo - Passed by J3 meeting 08-259 m185 Passed as modified by J3 LB #17 08-213 N1770 m187 Passed WG5 ballot 6 N1764-N1765

---

NUMBER: F03/0081
TITLE: F edit descriptor with field width 0
KEYWORDS: Data edit descriptors, F edit descriptor, field width
DEFECT TYPE: Interpretation
STATUS: Passed by WG5 ballot

QUESTION:

Does the Fortran 2003 standard allow the output field produced by an F edit descriptor with a field width w whose value is zero for a real value whose magnitude is less than one to include an optional leading zero?

For example, is the PRINT statement

```
PRINT '(F0.2)', 0.01
```

allowed to write "0.01"?

ANSWER:

No. This is clear from Section 10.6.1 of the Fortran 2003 standard,
NUMBER: F03/0082
TITLE: VALUE in place of INTENT for pure procedure dummy arguments
KEYWORDS: VALUE, INTENT, PURE
DEFECT TYPE: Interpretation
STATUS: Passed by WG5 ballot
QUESTION:

C1266 requires all nonpointer dummy arguments of a pure function to be
data objects that have INTENT(IN).  C1267 requires all dummy arguments
of a pure subroutine to have specified intents.

Was it intended that VALUE not be allowed in place of INTENT(IN), that
is, was it intended that if a dummy argument of a pure procedure has
the VALUE attribute, it is nonetheless also required to have the
INTENT(IN) attribute explicitly specified?

ANSWER:
The possible extension of the VALUE attribute to allow omission of the
INTENT(IN) attribute in the specific case of pure procedures was not
considered during the development of Fortran 2003. Nonetheless, the
standard is consistent and not in error, and so this restriction
should remain in Fortran 2003.

EDITS:
None.

SUBMITTED BY: Van Snyder
HISTORY: 06-153  m176  F03/0082 submitted, fix edit, needs more
          08-176  m184  Reversed answer + revised alternative edits
          Passed by J3 meeting
          08-259  m185  Passed as modified by J3 LB #17 08-213
          N1770  m187  Passed WG5 ballot 6 N1764-N1765

---------------------------------------------------------------------
Does the following subprogram fragment conflict with the 2003 Fortran standard? If so, was the conflict intended?

entry A ( X )
interface
  subroutine X ( A )
  abstract interface
  subroutine I ( ... )
  ... end subroutine I
  end interface
  procedure (I) A
end subroutine X
end interface
...

It seems that constraint C1255 in subclause 12.5.2.4 [283:10-12] prohibits the entry name A to appear in the PROCEDURE statement in the interface body. There shouldn’t be a problem, however, since the PROCEDURE statement is in a different scoping unit.

ANSWER:

No, this program fragment conforms to the 2003 Fortran standard.

EDITS:

None

NOTE:

C1255 is pointless because Clause 16 applies. It could fruitfully be removed from a future revision of the Fortran standard.

SUBMITTED BY: Van Snyder

HISTORY: 07-105 m179 F03/0087 submitted
          07-105 m179 Passed by J3 meeting
          07-272 m181 Failed J3 letter ballot #13 07-250r1
          08-192r1 m184 Revised answer - Passed by J3 meeting
          08-259 m185 Passed by J3 letter ballot #17 08-213
          N1770 m187 Passed WG5 ballot 6 N1764-N1765

---------------------------------------------------------------------
NUMBER: F03/0098
TITLE: Does allocate with source= define subcomponents?
KEYWORDS: allocate, source, define
DEFECT TYPE: Erratum
STATUS: Passed by WG5 ballot

QUESTION:
Was it intended that an allocate with a source= clause define subcomponents?

Bullet 11 on 422 says “Successful execution of an ALLOCATE statement...
...causes the subcomponent to become undefined.”

ANSWER:

An Allocate with a SOURCE= specifier was intended to define subcomponents.
In fact, none of the lists in clause 16 account for a SOURCE= specifier. Edits are supplied to clarify this.

EDITS:

[113:21] At the end of the last sentence in 6.3.1.1 insert "unless the SOURCE= specifier appears and the corresponding component of the <source-expr> is allocated".

[421:27] 16.5.5, list item 19, after "default-initialized subcomponent", insert ", except by an ALLOCATE statement with a SOURCE= specifier,"

[421:28+] 16.5.5, insert new list item after (19) "(19a) Successful execution of an ALLOCATE statement with a SOURCE= specifier causes a subobject of the allocated object to become defined if the corresponding subobject of the SOURCE= expression is defined."

[422:41-43] 16.5.6, list item (11), replace list item with "Successful allocation of an ALLOCATE statement with no SOURCE= specifier causes a subcomponent of an allocated object to become undefined if default initialization has not been specified for that subcomponent."

[422:43+] 16.5.6, add a new list item after (11) "(11a) Successful execution of an ALLOCATE statement with a SOURCE= specifier causes a subobject of the allocated object to become undefined if the corresponding subobject of the SOURCE= expression is undefined."

SUBMITTED BY: Dick Hendrickson

HISTORY: 07-268 m181 F03/0098 submitted 07-268r2 m181 Passed by J3 meeting 07-279 m182 Passed as changed by J3 LB #14 07-321 08-155 m184 Failed WG5 ballot #4 N1711-N1721 08-183 m184 Revised edits - Passed by J3 meeting 08-259 m185 Passed as modified by J3 LB #17 08-213 N1770 m187 Passed WG5 ballot 6 N1764-N1765

------------------------------------------------------------------------

NUMBER: F03/0099
TITLE: Clause 16 does not account for volatile variable
KEYWORDS: volatile, defined, undefined
DEFECT TYPE: Erratum
STATUS: Passed by WG5 ballot

QUESTION:

Should the various lists in clause 16 that describe definition and association be modified to include the effects of volatile variables?

In particular, 16.4.2.1.1 through 16.4.2.1.2 do not mention the fact that pointer association status can be volatile and change by other means.

16.5.4 says "all other variables are initially undefined." Can a volatile variable be initially defined by other means?

16.5 (26) says volatile variables "become defined", but they also can...
become undefined, by other means.

Allocatable volatile variables can become allocated or unallocated by other means also.

ANSWER:

Yes, the lists in clause 16 should be modified to include the effects of volatility. In addition, the effects of volatile on pointer objects are not completely specified in clause 5. The effect on allocatable objects is not complete in clause 6.

EDITS:

[85:10] In the paragraph between note 5.21 and note 5.22, change "association status and array bounds" to "association status, dynamic type and type parameters, and array bounds"

[415:27] Add a new paragraph at the end of 16.4.2.1.4

"The association status of a pointer object with the VOLATILE attribute might change by means not specified by the program."

[421:43] In 16.5.5 list item (26) change "becomes" to "might become".

[423:28+] In 16.5.6 after the last list item insert a new list item

"(19) An object with the VOLATILE attribute (5.1.2.16) might become undefined by means not specified by the program."

SUBMITTED BY: Dick Hendrickson

HISTORY: 07-269 m181 F03/0099 submitted
07-269r2 m181 Passed by J3 meeting
07-279/07-321 Failed letter ballot
07-339 m182 Passed by J3 meeting
08-133r2 m183 Failed letter ballot #15 08-101
08-137 m183 Passed by J3 meeting
08-163 m184 Passed by J3 letter ballot #16 08-141
N1770 m187 Passed WG5 ballot 6 N1764-N1765

------------------------------------------------------------------

NUMBER: F03/0102
TITLE: Evaluation of bound-expr in data pointer assignment
KEYWORDS: pointer, pointer assignment, bounds, expression
DEFECT TYPE: Clarification
STATUS: Passed by WG5 ballot

DISCUSSION:

Currently there are no rules in 7.4.2.1 to prohibit changing of a pointer's association status during evaluation of bound expressions in a data pointer assignment (pointer with either bounds-spec or bounds-remapping specified). This may lead to ambiguous code with regard to the evaluation orders between the bound expression and the data pointer assignment itself. Consider the following code,

```fortran
integer, target :: tar2(100, 100)
integer, target :: tar1(100)
integer, pointer :: ptr(:, :)
```
ptr(-2:, f1(ptr, tar1, 1, 1):) => tar2
print*, lbounds(ptr)
print*, ubounds(ptr)
print*, size(ptr)
contains
function f1(ptr, arr, i, j)
integer :: i, j, f1
integer, pointer :: ptr(:,:)
integer, target :: arr(:)

ptr(i:j, i:j) => arr
f1 = -1
end function
end
end

In 7.4.1.3 for interpretation of intrinsic assignments, there are rules explicitly requesting evaluation of all expressions in variable occurred before the variable is defined [139:14-19]. It appears that data pointer assignment should also follow similar rules.

Note the similar problem also exists for evaluating the <data-target> if it references a function that returns a data pointer.

QUESTION:
(a) Is this program intended to be standard conforming?
(b) If it is standard conforming, then what would it print?

ANSWER:
This program does not conform to the Fortran Standard.

The assertion that there are no rules about pointer association status is false because changes in pointer association status necessarily cause changes in definition status, as explained in the opening paragraph of “16.4.2 Pointer association status” which states “If a pointer is associated with a target, the definition status of the pointer is either defined or undefined, depending on the definition status of the target.”

The reference to F1(PTR, TAR1, 1, 1) executes the pointer assignment PTR(I:J, I:J) => ARR; this causes its actual argument PTR to become undefined (using the quoted rule from 16.4.2).

In 7.1.8 paragraph 3, it says “If a function reference causes definition or undefinition of an actual argument of the function, that argument or any associated entities shall not appear elsewhere in the same statement.”

However, PTR appears elsewhere in that statement (as the base object of the variable in the assignment), violating this requirement. Therefore the program is not standard-conforming.

EDITS:
None.
NOTE:

This would be clearer if the undefined case were also added to the list of "Events that cause variables to become undefined" in clause 16, e.g.

"(n) When a pointer becomes associated with a target that is undefined, the pointer becomes undefined."

This is recommended for a future revision of the Standard.

SUBMITTED BY: Jim Xia

HISTORY: 07-297r1 m182 F03/0102 submitted
07-297r2 m182 Passed by J3 meeting
08-133r2 m183 Failed J3 letter ballot
08-135 m183 Passed by J3 meeting
08-163 m184 Passed by J3 letter ballot #16 08-141
N1770 m187 Passed WG5 ballot 6 N1764-N1765

NUMBER: F03/0109
TITLE: Referencing deferred binding via absent dummy argument
KEYWORDS: Type-bound procedure, deferred binding
DEFECT TYPE: Erratum
STATUS: Passed by WG5 ballot

QUESTION:

The intent was that it would be impossible to reference a deferred binding. However, it doesn't appear to me that this intent was achieved. Consider the following program

```plaintext
program P
  type, abstract :: T
  contains
    procedure(sub), nopass, deferred :: deferred_proc
  end type T

  call sub
contains

  subroutine Sub ( X )
    class(t), optional :: X
    call x%deferred_proc
  end subroutine Sub
end program P
```

Is this a valid program? If not, what restriction of the standard does it violate?

Since x%deferred_proc has the NOPASS attribute, this does not require the value of x (4.5.7) and thus is not a reference to x (2.5.6). Therefore, the first item in the second list in 12.4.1.2 (at [04-007:272:32-33]) does not prohibit this.

ANSWER:

This was not intended to be a valid program. A type-bound procedure shall not be invoked through an absent dummy argument. An edit is supplied to clarify this situation.
The same flaw is present for procedure pointer component invocation.

EDITS:

Add new items to the second list in 12.4.1.6, [273:12+]

'(11) It shall not be supplied as the <data-ref> in a
<procedure-designator>.
(12) It shall not be supplied as the <variable> in a
<proc-component-ref>.'

SUBMITTED BY: Van Snyder

HISTORY: 07-338 m182 F03/0109 submitted - Passed by J3 meeting
08-133r2 m183 Failed J3 letter ballot
08-136 m183 Passed by J3 meeting
08-163 m184 Passed by J3 letter ballot #16 08-141
N1770 m187 Passed WG5 ballot 6 N1764-N1765

Note to Editor:
The edit for bullet (12) should say "<scalar-variable>" instead of
"<variable>" when applied to Fortran 2008. Compare R741 on page
157 of 08-007r2 with [143:35] of 04-007.

---------------------------------------------------------------------
NUMBER: F03/0111
TITLE: Is defined assignment to a vector subscripted array allowed?
KEYWORDS: defined assignment, vector-valued subscripts, elemental
DEFECT TYPE: Clarification
STATUS: Passed by WG5 ballot

QUESTION: Is the assignment statement in subroutine cg1018 standard
conforming?

It does an elemental defined assignment to an array with a vector
valued subscript. Several compilers reject this assignment,
claiming that an INTENT(OUT) argument can't be associated with
a vector-valued actual argument.

According to 142:6-7, this is a defined elemental assignment.

Lines 26 to 30 of 142 are
"The interpretation of a defined assignment is provided by the
subroutine that defines it.

If the defined assignment is an elemental assignment and the variable
in the assignment is an array, the defined assignment is performed
element-by-element, in any order, on corresponding elements of
variable and expr."

The first line looks like a general definition of defined assignment
and the second line looks like a qualification for the specific case
of elemental assignment. The qualification has to mean that the
assignments are performed on an element-by-element basis and this
surely must mean that the statement is processed as if it were
expanded into something like

TLA1L(7) = UDA1R(7)
TLA1L(1) = UDA1R(1)
TLA1L(3) = UDA1R(3)

and then the assignment subroutine is invoked on an
element-by-element basis following the rules in chapter 12.

Page 140, lines 4-5 have essentially identical words for intrinsic assignment and intrinsic assignment to a vector valued array, including derived type arrays, is allowed (if not many-to-one).

The processors that reject the test program apparently interpret the assignment as

\[
\text{Call } \text{U\_TO\_T}(\ TLA1L(NFV1), \ (UDA1R))
\]

without doing the assignment on an element-by-element basis.

If that interpretation is correct, then we have the unusual situation where

\[
\text{TLA1L(NFV1) = TLA1L}
\]

is standard conforming if the assignment is intrinsic and non-standard if the assignment is defined.

```
MODULE c_TESTS
  integer, save :: nfv1(10) = [1,2,3,4,5,6,7,8,9,10]
  TYPE UNSEQ
    REAL                           R
  END TYPE UNSEQ
  TYPE SEQ
    sequence
    REAL                           R
  END TYPE SEQ
  INTERFACE ASSIGNMENT(=)
    MODULE PROCEDURE U_TO_T
  END INTERFACE ASSIGNMENT(=)
  CONTAINS
    ELEMENTAL PURE SUBROUTINE U_TO_T(T,U)
      TYPE(SEQ), INTENT(IN) :: U
      TYPE(UNSEQ), INTENT(OUT) :: T
      T%R = U%R
    END SUBROUTINE U_TO_T
    SUBROUTINE CG1018(TLA1L,UDA1R)
      TYPE(UNSEQ) TLA1L(10)
      TYPE(SEQ) UDA1R(10)
      TLA1L(NFV1) = UDA1R             !???????
    END SUBROUTINE
  END MODULE c_TESTS
```

This is not standard conforming. According to [271:3-5]

If the actual argument is an array section having a vector subscript, the dummy argument is not definable and shall not have the INTENT (OUT), INTENT (INOUT), VOLATILE, or ASYNCHRONOUS attributes.

EDITS:
QUESTION:

The ID= specifier returns an identifier (or "handle") for an async i/o request. Must this fit inside the smallest integer kind supported by the processor?

Consider:

Integer(Selected_Int_Kind(1)) x(1000)
Character(80) :: test(1000) = (/ ... /)

Do i=1,1000
  Write (17,'(a)',Asynchronous='Yes',Id=x(i)) test(i)
  End Do

Do i=1,1000
  Wait (17,Id=x(i))
  End Do

X is only guaranteed to have approximately 20 values available, and in practice will probably only have 256 such values, so we are certainly not going to have 1000 unique handles.

Without loss of generality, suppose that one such duplicate value is 4. After WAIT(17,ID=4), the async i/o request with ID=4 is no longer pending. This means that the subsequent WAIT(17,ID=4) is not conforming to the standard since it requires that the ID=value is one that is pending.

(1) Is the processor permitted to require a minimum size for ID= (e.g. such as default integer)?

Clearly 1000 values do not fit into a 256-valued object, but apart from that,

(2) Is the processor required to produce unique ID= values for each multiple asynchronous i/o operation?

One might conjecture that if the processor returned ID=4 for two or more async i/o requests on the same unit, the first WAIT for ID=4 would wait for the first such async i/o request and the second one would wait for the second such async i/o request.

(3) Does WAIT-ing on an ID= value wait for all async i/o operations that have that as the handle, or are these queued up?
Unlike other i/o-related specifiers, there does not seem to be any constraint on the value returned for ID= (such as being non-zero or non-negative). In the case of the i/o being completed synchronously, there does not appear to be any way of returning a "non-handle" to the user program (one might have conjectured that zero was not a handle, but this is not supported by the text of the standard).

(4) Was there intended to be a "non-handle" return for ID=?

DISCUSSION:

It is understood that resource considerations may limit the number of outstanding asynchronous i/o requests, but 19 (the limit implied by the minimum-size integer that follows the model in c13) seems awfully small; for that matter 256 seems pretty small for today's operating systems.

ANSWER:

(1) Yes, the standard should require the ID= specifier to be default integer or bigger. An edit is provided.

(2) Yes, all pending data transfer operation identifiers on a particular unit are required to be unique. An edit is provided to clarify this.

(3) ID= values are required to be unique, so this question does not arise.

(4) Yes, the value zero should not be a handle but an indication the request was completed synchronously. An edit is provided to add this.

EDITs:

In 9.5.1 Control information list,
[187:2] "ID = <scalar-int-variable>" -> "ID = <id-variable>"

[187:10+] Insert new BNF rule and constraint
"R913a <id-variable> <<is>> <scalar-int-variable>

C908a (R913a) The <scalar-int-variable> shall have a decimal range no smaller than that of default integer."
{Require default integer or larger.}

In 9.5.1.8 ID= specifier in a data transfer statement
[190:17] Change "This value to "If this value is zero, the data transfer operation has been completed. A nonzero value"
{Zero = "it was done synchronously already" value.}

[190:18] After "operation." insert
"This identifier is different from the identifier of any other pending data transfer operation for this unit."
{Require uniqueness.}

[206:18] Before "the identifier" insert "zero or".
[206:20] After "operation" insert "if any,"
{Make ID=zero do nothing in WAIT.}

SUBMITTED BY: Malcolm Cohen
QUESTION:
All of the i/o specifiers that return INTEGER values permit any kind of integer to be used, even though this is likely to cause problems with small integers. But all of the i/o specifiers that return LOGICAL values require "full-size" (default) logical variables, even though every kind of LOGICAL can represent every LOGICAL value, viz .TRUE. and .FALSE. .

Should these be consistent, i.e. should every kind of LOGICAL be allowed for EXIST=, NAMED=, OPENED=, and PENDING=?

ANSWER:
Yes, these should have been consistent. Edits are provided.

EDITS:
All edits are against 04-007.

After the first paragraph in Clause 6, delete the BNF definition of and constraint for \si{default-logical-variable}:

\[103:17-18\] delete R605 and C604

Replace every occurrence of \si{scalar-default-logical-variable} in clause 9 with \si{scalar-logical-variable}:

\[210:18\] in 9.9.1 EXIST=
\[210:25\] in 9.9.1 NAMED=
\[210:28\] in 9.9.1 OPENED=
\[210:30\] in 9.9.1 PENDING=
\[212:24\] in 9.9.1.10 EXIST=
\[213:9\] in 9.9.1.15 NAMED=
\[213:23\] in 9.9.1.18 OPENED=
\[213:25\] in 9.9.1.18 OPENED=

(note: \si{scalar-default-logical-variable} is not used in 9.9.1.20 PENDING= (page 214))

SUBMITTED BY: Malcolm Cohen
Interpretation F90/000209 considered the question Section 9.7. May a function referenced in an I/O statement execute a STOP or PAUSE statement? If STOP statements are permitted, what is the state of the file at the end of execution?

The answer (w.r.t. STOP), included in F95 corrigendum 2, replied No, a STOP statement may not be executed in this situation. Executing a STOP statement causes normal termination of execution, which closes all files (9.3.5), and this is equivalent to executing a CLOSE statement, which is prohibited by 9.7. A clarifying edit is supplied.

However, on further consideration of the reasoning behind this answer, it seems unhelpful to rule out STOP, which is often used for user signalling of error conditions, during I/O operations. For one thing, if the data transfer statement being executed is READ, the question about file state has a very straightforward answer. It might also be considered unhelpful in the WRITE case for STOP to cause the user’s entire file system to be potentially reformatted instead of just making the contents of the relevant file undefined.

Furthermore, the user who wants to signal an error can cause error termination via one of the “approved” methods such as DEALLOCATE of an unallocated array. It seems less than unhelpful to force the user to use such a circumlocution simply for standards conformance, and the question about file state can equally be asked of this situation.

Q1. Is execution of a STOP statement during internal I/O allowed?
Q2. Is execution of a STOP statement during a READ statement allowed?
Q3. Is execution of a STOP statement during a WRITE statement allowed?
Q4. What is the effect on the file state of executing the following code fragment during a WRITE statement to an external file:

    REAL, ALLOCATABLE :: array(:)
    DEALLOCATE(array)

    (And if the answer to Q3 is yes, does a STOP statement have the same effect.)

ANSWER:

The answer to the first three questions is yes, this should have been allowed. An edit is provided to correct the standard.

The answer to Q4 is that the effects on the file currently being written by that WRITE statement are not standardised. (This was, in fact, the situation before the adoption of the previous interpretation.) A clarifying edit is provided to indicate that the contents of the file become undefined.

EDITS:
In 9.5.3 "Execution of a data transfer input/output statement", immediately before Note 9.38 insert a new paragraph

"If execution of the program is terminated during execution of a WRITE or PRINT statement, the contents of the file become undefined."

Delete [219:30] the last paragraph of 9.11 which currently reads

"A STOP statement shall not be executed during execution of an input/output statement."

SUBMITTED BY: Malcolm Cohen

HISTORY: 08-173 m184 F03/0117 submitted - Passed by J3 meeting
08-259 m185 Passed as modified by J3 LB #17 08-213
N1770 m187 Passed WG5 ballot 6 N1764-N1765