TR 29113 WORKING DRAFT

WG5/N1849

31st May 2011 14:36

This is an internal working document of WG5.

Contents

List of Tables

Foreword

1 ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and nongovernmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

2 International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

3 The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

4 Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

5 ISO/IEC TR 29113:2011(E) was prepared by Joint Technical Committee ISO/IEC/JTC1, *Information technology*, Subcommittee SC22, *Programming languages, their environments and system software interfaces.*

6 This technical report specifies an enhancement of the C interoperability facilities of the programming language Fortran. Fortran is specified by the International Standard ISO/IEC 1539-1:2010.

7 It is the intention of ISO/IEC JTC1/SC22/WG5 that the semantics and syntax specified by this technical report be included in the next revision of the Fortran International Standard without change unless experience in the implementation and use of this feature identifies errors that need to be corrected, or changes are needed to achieve proper integration, in which case every reasonable effort will be made to minimize the impact of such changes on existing implementations.

Introduction

Technical Report on Further Interoperability of Fortran with C

1 The system for interoperability between the C language, as standardized by ISO/IEC 9899:1999, and Fortran, as standardized by ISO/IEC 1539-1:2010, provides for interoperability of procedure interfaces with arguments that are non-optional scalars, explicit-shape arrays, or assumed-size arrays. These are the cases where the Fortran and C data concepts directly correspond. Interoperability is not provided for important cases where there is not a direct correspondence between C and Fortran.

2 The existing system for interoperability does not provide for interoperability of interfaces with Fortran dummy arguments that are assumed-shape arrays, have assumed character length, or have the ALLOCATABLE, POINTER, or OPTIONAL attributes. As a consequence, a significant class of Fortran subprograms is not portably accessible from C, limiting the usefulness of the facility.

3 The provision in the existing system for interoperability with a C formal parameter that is a pointer to void is inconvenient to use and error-prone. C functions with such parameters are widely used.

4 This Technical Report extends the facility of Fortran for interoperating with C to provide for interoperability of procedure interfaces that specify dummy arguments that are assumed-shape arrays, have assumed character length, or have the ALLOCATABLE, POINTER, or OPTIONAL attributes. New Fortran concepts of assumed type and assumed rank are introduced. The former simplifies interoperation with formal parameters of type (void *). The latter facilitates interoperability with C functions that can accept arguments of arbitrary rank. An intrinsic function, RANK, is specified to obtain the rank of an assumed-rank variable.

5 The facility specified in this Technical Report is a compatible extension of Fortran as standardized by ISO/IEC 1539-1:2010. It does not require that any changes be made to the C language as standardized by ISO/IEC 9899:1999.

6 This Technical Report is organized in 6 clauses:

Overview	Clause 1
Type specifiers and attributes	Clause 2
Procedure	Clause 3
New intrinsic procedure	Clause 4
Interoperability with C	Clause 5
Required editorial changes to ISO/IEC 1539-1:2010(E)	Clause 6

7 It also contains the following nonnormative material:

Extended notes	Annex A

Technical Report — Further Interoperability of Fortran with C —

1 Overview

1.1 Scope

1 This Technical Report specifies the form and establishes the interpretation of facilities that extend the Fortran language defined by ISO/IEC 1539-1:2010. The purpose of this Technical Report is to promote portability, reliability, maintainability and efficient execution of programs containing parts written in Fortran and parts written in C, for use on a variety of computing systems.

1.2 Normative references

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

2 ISO/IEC 1539-1:2010, *Information technology—Programming languages—Fortran*

3 ISO/IEC 9899:1999, *Information technology—Programming languages—C*

1.3 Terms and definitions

1 For the purposes of this document, the following terms and definitions apply. Terms not defined in this Technical Report are to be interpreted according to ISO/IEC 1539-1:2010.

1 **1.3.1**
assumed-rank object
⟨dummy variable⟩ whose rank is assumed from its effective argument

1 **1.3.2**
assumed-type object
⟨dummy variable⟩ whose type and type parameters are assumed from its effective argument

1 **1.3.3**
C descriptor
struct of type CFI_cdesc_t

NOTE 1.1

A C descriptor is used to describe an object that has no exact analog in C.

1.4 Compatibility

1.4.1 New intrinsic procedures

1 This Technical Report defines an intrinsic procedure in addition to those specified in ISO/IEC 1539-1:2010. Therefore, a Fortran program conforming to ISO/IEC 1539-1:2010 might have a different interpretation under

this Technical Report if it invokes an external procedure having the same name as the new intrinsic procedure, unless that procedure is specified to have the EXTERNAL attribute.

1.4.2 Fortran 2008 compatibility

1 This Technical Report specifies an upwardly compatible extension to ISO/IEC 1539-1:2010.

2 Type specifiers and attributes

2.1 Assumed-type objects

1 The syntax rule R403 *declaration-type-spec* in subclause 4.3.1.1 of ISO/IEC 1539-1:2010 is replaced by

R403 *declaration-type-spec* **is** *intrinsic-type-spec*
or TYPE (*intrinsic-type-spec*)
or TYPE (*derived-type-spec*)
or CLASS (*derived-type-spec*)
or CLASS (*)
or TYPE (*)

2 An entity declared with a *declaration-type-spec* of TYPE (*) is an assumed-type entity. It has no declared type and its dynamic type and type parameters are assumed from its effective argument.

C407a An assumed-type entity shall be a dummy variable that does not have the ALLOCATABLE, CODIMENSION, POINTER, or VALUE attribute and is not an explicit-shape array.

C407b An assumed-type variable name shall not appear in a designator or expression except as an actual argument corresponding to a dummy argument that is assumed-type, or the first argument to the intrinsic and intrinsic module functions IS_CONTIGUOUS, LBOUND, PRESENT, RANK, SHAPE, SIZE, UBOUND, or C_LOC.

3 An assumed-type object is unlimited polymorphic.

NOTE 2.1

An assumed-type object that is not assumed-shape and not assumed-rank is passed as a simple pointer to the first address of the object. This means that there is insufficient information to construct an assumed-shape dope vector or C descriptor. As a consequence, there would be no functional difference between TYPE(*) explicit-shape and TYPE(*) assumed-size. Therefore TYPE(*) explicit-shape is not permitted.

NOTE 2.2

This Technical Report provides no mechanism within Fortran code to determine the actual type of an assumed-type argument.

2.2 Assumed-rank objects

1 The syntax rule R515 *array-spec* in subclause 5.3.8.1 of ISO/IEC 1539-1:2010 is replaced by

R515 *array-spec* **is** *explicit-shape-spec-list*
or *assumed-shape-spec-list*
or *deferred-shape-spec-list*
or *assumed-size-spec*
or *implied-shape-spec-list*
or *assumed-rank-spec*

2 An assumed-rank object is a dummy variable whose rank is assumed from its effective argument. An assumed-rank object is declared with an *array-spec* that is an *assumed-rank-spec*.

R522a *assumed-rank-spec* **is** ..

C535a An assumed-rank entity shall be a dummy variable that does not have the CODIMENSION or VALUE attribute.

3 An assumed-rank object may have the CONTIGUOUS attribute.

C535b An assumed-rank variable name shall not appear in a designator or expression except as an actual argument corresponding to a dummy argument that is assumed-rank, the argument of the C_LOC function in the ISO_C_BINDING intrinsic module, or the first argument in a reference to an intrinsic inquiry function.

4 The intrinsic inquiry function RANK can be used to inquire about the rank of a data object. The rank of an assumed-rank object is zero if the rank of the corresponding actual argument is zero.

5 The definition of TKR compatible in paragraph 2 of subclause 12.4.3.4.5 of ISO/IEC 1539-1:2010 is changed to:

> A dummy argument is type, kind, and rank compatible, or TKR compatible, with another dummy argument if the first is type compatible with the second, the kind type parameters of the first have the same values as the corresponding kind type parameters of the second, and both have the same rank or either is assumed-rank.

NOTE 2.3

Assumed rank is an attribute of a Fortran dummy argument. When a C function is invoked with an actual argument that corresponds to an assumed-rank dummy argument in a Fortran interface for that C function, the corresponding formal parameter is a pointer to a descriptor of type CFI_cdesc_t (5.2.8). The rank component of the descriptor provides the rank of the actual argument. The C function must therefore be able to handle any rank. On each invocation, the rank is available to it.

2.3 OPTIONAL attribute

1 The OPTIONAL attribute may be specified for a dummy argument in a procedure interface that has the BIND attribute.

2 The constraint C1255 in subclause 12.6.2.2 of ISO/IEC 1539-1:2010 is replaced by

C1255 (R1229) If *proc-language-binding-spec* is specified for a procedure, each dummy argument of the procedure shall be an interoperable procedure (15.3.7) or an interoperable variable (15.3.5, 15.3.6) that does not have both the OPTIONAL and VALUE attributes. If *proc-language-binding-spec* is specified for a function, the function result shall be an interoperable scalar variable.

3 Constraint C516 in subclause 5.3.1 of ISO/IEC 1539-1:2010 says "The ALLOCATABLE, POINTER, or OPTIONAL attribute shall not be specified for a dummy argument of a procedure that has a *proc-language-binding-spec*." This is replaced by the much less restrictive constraint:

C516 The ALLOCATABLE or POINTER attribute shall not be specified for a default-initialized dummy argument of a procedure that has a *proc-language-binding-spec*.

NOTE 2.4

It would be a severe burden to implementors to require that CFI_allocate initialize components of an object of a derived type with default initialization. The alternative of not requiring initialization would have been inconsistent with the effect of ALLOCATE in Fortran.

3 Procedures

3.1 Characteristics of dummy data objects

1 Additionally to the characteristics listed in subclause 12.3.2.2 of ISO/IEC 1539-1:2010, whether the type or rank of a dummy data object is assumed is a characteristic of the dummy data object.

3.2 Explicit interface

1 Additionally to the rules of subclause 12.4.2.2 of ISO/IEC 1539-1:2010, a procedure shall have an explicit interface if it has a dummy argument that is assumed-type or assumed-rank.

3.3 Argument association

1 An assumed-rank dummy argument may correspond to an actual argument of any rank. If the actual argument is scalar, the dummy argument has rank zero; the shape is a zero-sized array and the LBOUND and UBOUND intrinsic functions, with no DIM argument, return zero-sized arrays. If the actual argument is an array, the rank and bounds of the dummy argument are assumed from the actual argument. The value of the lower and upper bound of dimension N of the dummy argument are equal to the result of applying the LBOUND and UBOUND intrinsic inquiry functions to the actual argument with DIM=N specified.

2 An assumed-type dummy argument shall not correspond to an actual argument that is of a derived type that has type parameters, type-bound procedures, or final procedures.

3 If a Fortran procedure that has an INTENT(OUT) allocatable dummy argument is invoked by a C function, and the actual argument in the C function is a C descriptor that describes an allocated allocatable variable, the variable is deallocated on entry to the Fortran procedure.

4 When a C function is invoked from a Fortran procedure via an interface with an INTENT(OUT) allocatable dummy argument, and the actual argument in the reference to the C function is an allocated allocatable variable, the variable is deallocated on invocation (before execution of the C function begins).

3.4 Intrinsic procedures

3.4.1 SHAPE

1 The description of the intrinsic function SHAPE in ISO/IEC 1539-1:2010 is changed for an assumed-rank array that is associated with an assumed-size array; an assumed-size array has no shape, but in this case the result has a value of [(SIZE (ARRAY, I), I=1, RANK (ARRAY))]

3.4.2 SIZE

1 The description of the intrinsic function SIZE in ISO/IEC 1539-1:2010 is changed in the following cases:

(1) for an assumed-rank object that is associated with an assumed-size array, the result has a value of -1 if DIM is present and equal to the rank of ARRAY, and a negative value that is equal to PRODUCT ([(SIZE (ARRAY, I), I=1, RANK (ARRAY))]) if DIM is not present;

(2) for an assumed-rank object that is associated with a scalar, the result has a value of 1.

3.4.3 UBOUND

1 The description of the intrinsic function UBOUND in ISO/IEC 1539-1:2010 is changed for an assumed-rank object that is associated with an assumed-size array; the result has a value of LBOUND (ARRAY, RANK (ARRAY)) -2.

NOTE 3.1

If LBOUND or UBOUND is invoked for an assumed-rank object that is associated with a scalar and DIM is absent, the result is a zero-sized array. LBOUND or UBOUND cannot be invoked for an assumed-rank object that is associated with a scalar if DIM is present because the rank of a scalar is zero and DIM must be ≥ 1.

4 New intrinsic procedure

4.1 General

1 Detailed specification of the generic intrinsic function RANK is provided in 4.2. The types and type parameters of the RANK intrinsic procedure argument and function result are determined by this specification. The "Argument" paragraph specifies requirements on the actual arguments of the procedure. The intrinsic function RANK is pure.

4.2 RANK (A)

1 **Description.** Rank of a data object.

2 **Class.** Inquiry function.

3 **Arguments.**

A shall be a scalar or array of any type.

4 **Result Characteristics.** Default integer scalar.

5 **Result Value.** The result is the rank of A.

6 **Example.** For an array X declared REAL :: X(:,:,:), RANK(X) is 3.

5 Interoperability with C

5.1 C descriptors

1 A C descriptor is a struct of type CFI_cdesc_t. The C descriptor along with library functions with standard prototypes provide the means for describing an assumed-shape, assumed-rank, allocatable, or data pointer object within a C function. This struct is defined in the file `ISO_Fortran_binding.h`.

5.2 ISO_Fortran_binding.h

5.2.1 Summary of contents

1 The `ISO_Fortran_binding.h` file contains the definitions of the C structs CFI_cdesc_t and CFI_dim_t, typedef definitions for CFI_attribute_t, CFI_index_t, CFI_rank_t, and CFI_type_t, the definition of the macro CFI_CDESC_T, macro definitions that expand to integer constants, and C prototypes for the C functions CFI_address, CFI_allocate, CFI_deallocate, CFI_establish, CFI_is_contiguous, CFI_section, CFI_select_part, and CFI_setpointer. The contents of `ISO_Fortran_binding.h` can be used by a C function to interpret a C descriptor and allocate and deallocate objects represented by a C descriptor. These provide a means to specify a C prototype that interoperates with a Fortran interface that has an allocatable, assumed character length, assumed-rank, assumed-shape, or data pointer dummy argument.

2 `ISO_Fortran_binding.h` may be included in any order relative to the standard C headers, and may be included more than once in a given scope, with no effect different from being included only once, other than the effect on line numbers.

3 A C source file that includes the header `ISO_Fortran_binding.h` shall not use any names starting CFI_ that are not defined in the header. All names defined in the header begin with CFI_ or an underscore character, or are defined by a standard C header that it includes.

5.2.2 CFI_dim_t

1 CFI_dim_t is a named struct type defined by a typedef. It is used to represent lower bound, extent, and memory stride information for one dimension of an array. CFI_index_t is a typedef name for a standard signed integer type capable of representing the result of subtracting two pointers. CFI_dim_t contains at least the following members in any order:

CFI_index_t lower_bound; equal to the value of the lower bound for the dimension being described.

CFI_index_t extent; equal to the number of elements along the dimension being described.

CFI_index_t sm; equal to the memory stride for a dimension. The value is the distance in bytes between the beginnings of successive elements along the dimension being described.

5.2.3 CFI_cdesc_t

1 CFI_cdesc_t is a named struct type defined by a typedef, containing a flexible array member. It shall contain at least the following members. The first three members of the struct shall be `base_addr`, `elem_len`, and `version` in that order. The final member shall be `dim`, with the other members after `version` and before `dim` in any order.

void * base_addr; If the object is an unallocated allocatable or a pointer that is disassociated, the value is NULL. If the object has zero size, the value is not NULL but is otherwise processor-dependent. Otherwise, the value is the base address of the object being described. The base address of a scalar is its C address. The base address of an array is the C address of the element for which each Fortran subscript has the value of the corresponding lower bound.

size_t elem_len; If the object corresponds to a Fortran CHARACTER object, the value equals the length of the CHARACTER object times the `sizeof()` of a scalar of the character type; otherwise, the value equals the `sizeof()` of an element of the object.

int version; shall be set equal to the value of CFI_VERSION in the `ISO_Fortran_binding.h` header file that defined the format and meaning of this descriptor.

CFI_rank_t rank; equal to the number of dimensions of the Fortran object being described. If the object is a scalar, the value is zero. CFI_rank_t shall be a typedef name for a standard integer type capable of representing the largest supported rank.

CFI_type_t type; equal to the specifier for the type of the object. Each interoperable intrinsic C type has a specifier. A specifier is also provided to indicate that the type of the object is a struct type, or is unknown. Macros and the corresponding values for the specifiers are defined in the `ISO_Fortran_binding.h` file. CFI_type_t shall be a typedef name for a standard integer type capable of representing the values for the supported type specifiers.

CFI_attribute_t attribute; equal to the value of an attribute code that indicates whether the object described is a data pointer, allocatable, assumed-shape, or assumed-size. Macros and the corresponding values for the attribute codes are supplied in the `ISO_Fortran_binding.h` file. CFI_attribute_t shall be a typedef name for a standard integer type capable of representing the values of the attribute codes.

CFI_dim_t dim[ ]; Each element of the array contains the lower bound, extent, and memory stride information for the corresponding dimension of the Fortran object. The number of elements in the array shall be equal to the rank of the object.

2 For a descriptor of an assumed-shape array, the value of the `lower-bound` member of each element of the `dim` member of the descriptor shall be zero. For a descriptor of an allocatable or pointer array, the value of the `lower_bound` member of each element of the `dim` member of the descriptor is the Fortran lower bound.

3 There shall be an ordering of the dimensions such that the absolute value of the `sm` member of one dimension is not less than the absolute value of the `sm` member of the previous dimension multiplied by the extent of the previous dimension.

4 If any actual argument associated with the dummy argument is an assumed-size array, the array shall be simply contiguous, the `attribute` member shall be CFI_attribute_unknown_size, and the `extent` member of the last dimension of the `dim` member shall have the value −2.

NOTE 5.1

If the type of the Fortran object is character with kind C_CHAR, the value of the `elem_len` member will be equal to the character length.

5.2.4 Macros

1 The macros described in this subclause are defined in `ISO_Fortran_binding.h`. Except for CFI_CDESC_T, each expands to an integer constant expression suitable for use in `#if` preprocessing directives.

2 CFI_CDESC_T is a function-like macro that takes one argument, which is the rank of the descriptor to create, and evaluates to a type suitable for declaring a descriptor of that rank. A pointer to a variable declared using CFI_CDESC_T can be cast to CFI_cdesc_t *. A variable declared using CFI_CDESC_T shall not have an initializer.

NOTE 5.2

The following code uses CFI_CDESC_T to declare a descriptor of rank 5 and pass it to CFI_deallocate.

NOTE 5.2 (cont.)

```
CFI_CDESC_T(5) object;
... code to define and use descriptor ...
CFI_deallocate((CFI_cdesc_t *) &object);
```

3 CFI_MAX_RANK has a processor-dependent value equal to the largest rank supported. The value shall be greater than or equal to 15.

4 CFI_VERSION has a processor-dependent value that encodes the version of the `ISO_Fortran_binding.h` header file containing this macro.

NOTE 5.3

The intent is that the version should be increased every time that the header is incompatibly changed, and that the version in a descriptor may be used to provide a level of upwards compatibility, by using means not defined by this Technical Report.

5 The macros in Table 5.1 are for use as attribute codes. The values shall be nonnegative and distinct.

Table 5.1: **Macros specifying attribute codes**

Macro	Code
CFI_attribute_assumed	assumed
CFI_attribute_allocatable	allocatable
CFI_attribute_pointer	pointer
CFI_attribute_unknown_size	assumed size

6 CFI_attribute_pointer specifies an object with the Fortran POINTER attribute. CFI_attribute_allocatable specifies an object with the Fortran ALLOCATABLE attribute. CFI_attribute_assumed specifies an assumed-shape object or a scalar that is not allocatable or a pointer. CFI_attribute_unknown_size specifies an object that is, or is argument-associated with, an assumed-size dummy argument.

7 The macros in Table 5.2 are for use as type specifiers. The value for CFI_type_other shall be distinct from all other type specifiers. If a C type is not interoperable with a Fortran type and kind supported by the Fortran processor, its macro shall evaluate to a negative value. Otherwise, the value for an intrinsic type shall be positive.

Table 5.2: **Macros specifying type codes**

Macro	C Type
CFI_type_signed_char	signed char
CFI_type_short	short int
CFI_type_int	int
CFI_type_long	long int
CFI_type_long_long	long long int
CFI_type_size_t	size_t
CFI_type_int8_t	int8_t
CFI_type_int16_t	int16_t
CFI_type_int32_t	int32_t
CFI_type_int64_t	int64_t
CFI_type_int_least8_t	int_least8_t
CFI_type_int_least16_t	int_least16_t
CFI_type_int_least32_t	int_least32_t
CFI_type_int_least64_t	int_least64_t
CFI_type_int_fast8_t	int_fast8_t
CFI_type_int_fast16_t	int_fast16_t

Macros specifying type codes (cont.)

Macro	C Type
CFI_type_int_fast32_t	int_fast32_t
CFI_type_int_fast64_t	int_fast64_t
CFI_type_intmax_t	intmax_t
CFI_type_intptr_t	intptr_t
CFI_type_float	float
CFI_type_double	double
CFI_type_long_double	long double
CFI_type_float_Complex	float _Complex
CFI_type_double_Complex	double _Complex
CFI_type_long_double_Complex	long double _Complex
CFI_type_Bool	_Bool
CFI_type_char	char
CFI_type_cptr	void *
CFI_type_cfunptr	pointer to a function
CFI_type_other	Any other type

NOTE 5.4

The specifiers for two intrinsic types can have the same value. For example, CFI_type_int and CFI_type_int32_t might have the same value.

8 The macros in Table 5.3 are for use as error codes. The macro CFI_SUCCESS shall be defined to be the integer constant 0.

9 The values of the error codes returned for the error conditions listed below are named by the indicated macros. The value of each macro other than CFI_SUCCESS shall be nonzero and shall be different from the values of the other macros specified in this subclause. Error conditions other than those listed in this subclause should be indicated by error codes different from the values of the macros named in this subclause.

10 The error codes that indicate the following error conditions are named by the associated macro name.

Table 5.3: **Macros specifying error codes**

Macro	Error
CFI_SUCCESS	No error detected.
CFI_ERROR_BASE_ADDR_NULL	The base address member of a C descriptor is NULL in a context that requires a non-null value.
CFI_ERROR_BASE_ADDR_NOT_NULL	The base address member of a C descriptor is not NULL in a context that requires a null value.
CFI_INVALID_ELEM_LEN	The value of the element length member of a C descriptor is not valid.
CFI_INVALID_RANK	The value of the rank member of a C descriptor is not valid.
CFI_INVALID_TYPE	The value of the type member of a C descriptor is not valid.
CFI_INVALID_ATTRIBUTE	The value of the attribute member of a C descriptor is not valid.

Macros specifying error codes (cont.)

Macro	Error
CFI_INVALID_EXTENT	The value of the extent member of a CFI_dim_t structure is not valid.
CFI_INVALID_SM	The value of the memory stride member of a CFI_dim_t structure is not valid.
CFI_INVALID_DESCRIPTOR	A general error condition for C descriptors.
CFI_ERROR_MEM_ALLOCATION	Memory allocation failed.
CFI_ERROR_OUT_OF_BOUNDS	A reference is out of bounds.

5.2.5 Functions

5.2.5.1 General

1 The functions described in this subclause and the structure of the C descriptor provide a C function with the capability to interoperate with a Fortran procedure that has an allocatable, assumed character length, assumed-rank, assumed-shape, or data pointer argument.

2 Within a C function, allocatable objects shall be allocated or deallocated only through execution of the CFI_allocate and CFI_deallocate functions. A Fortran pointer can become associated with a target by execution of the CFI_allocate function.

3 A C descriptor for a Fortran pointer can be constructed by execution of the functions described in this subclause. If a Fortran object without the TARGET attribute is associated with a formal parameter in a call to a C function and a C descriptor for a Fortran pointer to the formal parameter or a part of it exists on return, the `base_addr` member of the C descriptor becomes undefined on return.

4 Some of the functions described in 5.2.5 return an integer value that indicates if an error condition was detected. If no error condition was detected an integer zero is returned; if an error condition was detected, a nonzero integer is returned. A list of error conditions and macro names for the corresponding error codes is supplied in 5.2.4. A processor is permitted to detect other error conditions. If an invocation of a function defined in 5.2.5 could detect more than one error condition and an error condition is detected, which error condition is detected is processor dependent.

5 Prototypes for these functions are provided in the `ISO_Fortran_binding.h` file as follows:

5.2.5.2 void * CFI_address (const CFI_cdesc_t * dv, const CFI_index_t subscripts[]);

1 **Description.** Compute the address of an object described by a C descriptor.

2 **Formal Parameters.**

`dv` shall point to a C descriptor describing the object. The object shall not be an unallocated allocatable or a pointer that is not associated.

`subscripts` is ignored if the object is scalar. If the object is an array, `subscripts` points to a subscripts array. The number of elements shall be greater than or equal to the rank r of the object. The subscript values shall be within the bounds specified by the corresponding elements of the `dim` member of the C descriptor.

3 **Result Value.** If the object is an array, the result is the address of the element of the object that the first r

elements of the `subscripts` argument would specify if used as subscripts. If the object is scalar, the result is its address.

NOTE 5.5

When the subscripts argument is ignored, its value may be either NULL or a valid pointer value, but it need not point to an object.

4 **Example.** If `dv` points to a C descriptor for the Fortran array `a` declared as

```
real a(100,100)
```

5 the following code returns the address of `a(10,10)`

```
CFI_index_t subscripts[2];
long int address;
subscripts[0] = 10;
subscripts[1] = 10;
address = CFI_address( dv, subscripts );
```

5.2.5.3 int CFI_allocate (CFI_cdesc_t * dv, const CFI_index_t lower_bounds[], const CFI_index_t upper_bounds[], size_t elem_len) ;

1 **Description.** Allocates memory for an object described by a C descriptor.

2 **Formal Parameters.**

`dv` shall point to a C descriptor describing the object. The attribute member of the C descriptor shall have a value of CFI_attribute_allocatable or CFI_attribute_pointer.

`lower_bounds` points to a lower bounds array. The number of elements shall be greater than or equal to the rank r specified in the descriptor.

`upper_bounds` points to an upper bounds array. The number of elements shall be greater than or equal to the rank r specified in the descriptor.

`elem_len` is ignored unless the type specified in the descriptor is CFI_type_other or a character type. If the type is CFI_type_other, `elem_len` shall be greater than zero and equal to the `sizeof()` of an element of the object. If the object is of Fortran character type, the value of `elem_len` shall be the number of characters in an element of the object times the `sizeof()` of a scalar of the character type.

3 CFI_allocate allocates memory for the object described by the C descriptor pointed to by the `dv` argument using the same mechanism as the Fortran ALLOCATE statement. The first r elements of the `lower_bounds` and `upper_bounds` arguments provide the lower and upper Fortran bounds, respectively, for each corresponding dimension of the object. If the rank is zero, the `lower_bounds` and `upper_bounds` arguments are ignored.

4 On successful execution of CFI_allocate, the supplied lower and upper bounds override any current dimension information in the C descriptor and the C descriptor is updated. If an error is detected, the C descriptor is not modified.

5 **Result Value.** The result is an error indicator.

6 **Example.** If `dv` points to a C descriptor for the Fortran array `a` declared as

```
real, allocatable :: a(:,:)
```

7 and the array is not allocated, the following code allocates it to be of shape [100, 1000]

```
CFI_index_t lower[2], upper[2];
int flag;
size_t dummy;
lower[0] = 1; lower[1] = 1;
upper[0] = 100; upper[1] = 1000;
flag = CFI_allocate( dv, lower, upper, dummy );
```

5.2.5.4 int CFI_deallocate (CFI_cdesc_t * dv);

1 **Description.** Deallocates memory for an object described by a C descriptor.

2 **Formal Parameters.**

`dv` shall point to a C descriptor describing the object. It shall have been allocated using the same mechanism as the Fortran ALLOCATE statement. If the object is a pointer, it shall be associated with a target satisfying the conditions for successful deallocation by the Fortran DEALLOCATE statement (6.7.3.3 of ISO/IEC 1539-1:2010).

3 CFI_deallocate deallocates memory for the object. It uses the same mechanism as the Fortran DEALLOCATE statement.

4 On successful execution of CFI_deallocate, the C descriptor is updated. If an error is detected, the C descriptor is not modified.

5 **Result Value.** The result is an error indicator.

6 **Example.** If `dv` points to a C descriptor for the Fortran array `a` declared as

```
real, allocatable :: a(:,:)
```

7 and the array is allocated, the following code deallocates it

```
int flag;
flag = CFI_deallocate( dv );
```

5.2.5.5 int CFI_establish (CFI_cdesc_t * dv, void * base_addr, CFI_attribute_t attribute, CFI_type_t type, size_t elem_len, CFI_rank_t rank, const CFI_dim_t dim[]);

1 **Description.** Establishes a C descriptor for an object.

2 **Formal Parameters.**

`dv` shall point to a C object large enough to hold a C descriptor of the appropriate rank. It shall not point to a C descriptor that describes an object that is described by a C descriptor pointed to by a formal parameter that corresponds to a Fortran dummy argument. If it points to a C descriptor that describes an allocatable object, the object shall be unallocated.

`base_addr` shall be NULL or the base address of the object. If it is not NULL it shall be appropriately aligned (ISO/IEC 9899:1999 3.2) for an object of the specified type.

`attribute` shall be one of CFI_attribute_assumed, CFI_attribute_allocatable, or CFI_attribute_pointer. If it is CFI_attribute_assumed, `base_addr` shall not be NULL. If it is CFI_attribute_allocatable, `base_addr` shall be NULL.

`type` shall be one of the type names in Table 5.2.

`elem_len` is ignored unless `type` is CFI_type_other or a character type. If the type is CFI_type_other, `elem_len` shall be greater than zero and equal to the `sizeof()` for an element of the object. If the object is of Fortran character type, the value of `elem_len` shall be the number of characters in an element of the object times the `sizeof()` for a scalar of the character type.

rank is the rank of the object. It shall be between 0 and CFI_MAX_RANK inclusive.

`dim` is ignored if the rank r is not greater than zero or if `base_addr` is NULL. Otherwise, it shall point to an array with r elements specifying the Fortran dimension information.

3 CFI_establish establishes a C descriptor for an assumed-shape array, an assumed-length character object, an unallocated allocatable, or a pointer. If `base_addr` is derived from the C address of a Fortran object, CFI_establish establishes a C descriptor for that object or a subobject of it. If `base_addr` is NULL, the established C descriptor is for an unallocated allocatable, or a disassociated pointer. The properties of the object are given by the other arguments.

4 On successful execution of CFI_establish, the object pointed to by `dv` is updated to the C descriptor established. If an error is detected, that object is not modified.

5 **Result Value.** The function returns an error indicator.

6 **Example 1.** The following code fragment establishes a C descriptor for an unallocated rank-one allocatable array to pass to Fortran for allocation there.

```
CFI_rank_t rank;
CFI_dim_t dim[1];
CFI_CDESC_T(1) field;
int ind;
rank = 1;
ind = CFI_establish ( (CFI_cdesc_t *)  &field, NULL, CFI_attribute_allocatable,
                      CFI_type_double, 0, rank, dim );
```

7 **Example 2.** If `source` already points to a C descriptor for the Fortran array `a` declared thus:

```
type,bind(c) :: t
  REAL(C_DOUBLE) x
  complex(C_DOUBLE_COMPLEX) y
end type
type(t) a(100)
```

8 the following code fragment establishes a C descriptor for the array `a(:)%y`.

```
typedef struct { double x; double complex y;} t;
CFI_dim_t dim[1];
CFI_CDESC_T(1) component;
int ind;
dim[0].lower_bound = 0;
dim[0].extent = 100;
dim[0].sm = sizeof(t);
ind = CFI_establish ( (CFI_cdesc_t *) &component,
            (char *)source->base_addr+offsetof(t, y),
            CFI_attribute_assumed, CFI_type_double_Complex,
            0, source->rank, dim );
```

5.2.5.6 int CFI_is_contiguous (const CFI_cdesc_t * dv);

1 **Description.** Test contiguity of an array.

2 **Formal Parameter.**

`dv` shall point to a C descriptor describing the object.

3 **Result Value.** CFI_is_contiguous returns 1 if the descriptor pointed to by the `dv` argument is a valid C descriptor and the object described is determined to be contiguous, and 0 otherwise.

5.2.5.7 int CFI_section (CFI_cdesc_t * result, const CFI_cdesc_t * source, CFI_attribute_t attribute, CFI_rank_t rank, const CFI_dim_t dim[]);

1 **Description.** Establishes a C descriptor for an array section.

2 **Formal Parameters.**

`result` shall point to a C object large enough to hold a C descriptor of the appropriate rank. It shall not point to a C descriptor that describes an object that is described by a C descriptor pointed to by a formal parameter that corresponds to a Fortran dummy argument. If it points to a C descriptor that describes an allocatable object, the object shall be unallocated.

`source` shall point to a C descriptor that describes an assumed-shape array, an allocated allocatable array, or an associated pointer.

`attribute` shall be CFI_attribute_assumed or CFI_attribute_pointer.

`rank` specifies the rank of the array section. It shall be between 1 and the rank of the object described by the C descriptor pointed to by `source` inclusive.

`dim` points to an array specifying the Fortran dimension information for the array section. The number of elements shall be the rank of the array section. The values of the elements shall be such that they specify an array that could have been obtained by associating the `source` argument with a Fortran assumed-shape array and applying array section notation in Fortran.

3 CFI_section establishes a C descriptor to refer to a section of an array described by the C descriptor pointed to by the `source` argument. The `attribute` argument determines whether the C descriptor describes an assumed-shape array or pointer object.

4 On successful execution of CFI_section, the object pointed to by `result` is updated to the C descriptor established. If an error is detected, that object is not modified.

5 **Result Value.** The function returns an error indicator.

6 **Example.** If `source` already points to a C descriptor for the rank-one Fortran array `A`, the following code fragment establishes a C descriptor for the array section `A(1:10:5)`.

```
CFI_dim_t dim[1];
CFI_rank_t rank = 1;
CFI_CDESC_T(1) section;
int ind;
dim[0].lower_bound = 0;
dim[0].extent = 2;
dim[0].sm = 5*source->dim[0].sm;
ind = CFI_section ( (CFI_cdesc_t *) &section, source, CFI_attribute_assumed, rank, dim );
```

5.2.5.8 int CFI_select_part (CFI_cdesc_t * result, const CFI_cdesc_t * source, CFI_attribute_t attribute, CFI_type_t type, size_t displacement, size_t elem_len);

1 **Description.** CFI_select_part establishes a C descriptor for an array for which each element is a part of the corresponding element of an array.

2 **Formal Parameters.**

`result` shall point to a C object large enough to hold a C descriptor of the appropriate rank. It shall not point to a C descriptor that describes an object that is described by a C descriptor pointed to by a formal parameter that corresponds to a Fortran dummy argument. If it points to a C descriptor that describes an allocatable object, the object shall be unallocated.

`source` shall point to a C descriptor for an assumed-shape array, an allocated allocatable array, or an associated pointer array.

`attribute` shall be CFI_attribute_assumed or CFI_attribute_pointer.

`type` shall be one of the type names in Table 5.2. It specifies the type of the array section described by the C descriptor pointed to by `result`.

`displacement` is the value to be added to the base address of the array described by the C descriptor pointed to by `source` to give the base address of the array section described by the C descriptor pointed to by `result`. The resulting base address shall be appropriately aligned (ISO/IEC 9899:1999 3.2) for an object of the specified type. The value of `displacement` shall be between 0 and `source->elem_len - 1` inclusive.

`elem_len` is ignored unless `type` is CFI_type_other or a character type. If the type is CFI_type_other, `elem_len` shall be greater than zero and equal to the `sizeof()` for an element of the object. If the object is of Fortran character type, the value of `elem_len` shall be the number of characters in an element of the object times the `sizeof()` for a scalar of the character type. The value of `elem_len` shall be between 1 and `source->elem_len` inclusive.

3 CFI_select_part establishes a C descriptor for an array for which each element is a part of the corresponding element of the array described by the C descriptor pointed to by `source`. The part may be a component of a structure, a substring, or the real or imaginary part of a complex value. The `attribute` argument determines whether the C descriptor describes an assumed-shape array or array pointer.

4 On successful execution of CFI_select_part, the object pointed to by `result` is updated to the C descriptor established. If an error is detected, that object is not modified.

5 **Result Value.** The function returns an error indicator.

6 **Example.** If `source` already points to a C descriptor for the Fortran array `a` declared thus:

```
type,bind(c):: t
  real(C_DOUBLE) :: x
  complex(C_DOUBLE_COMPLEX) :: y
end type
type(t) a(100)
```

7 the following code fragment establishes a C descriptor for the array `a(:)%y`.

```
typedef struct { double x; double complex y;} t;
CFI_CDESC_T(1) component;
int ind;

ind = CFI_select_part ( (CFI_cdesc_t *) &component, source, CFI_attribute_assumed,
      CFI_type_double_complex, offsetof(t, y), 0 );
```

5.2.5.9 int CFI_setpointer (CFI_cdesc_t * result, CFI_cdesc_t * source, const CFI_index_t lower_bounds[]);

1 **Description.** CFI_setpointer updates a C descriptor for a Fortran pointer.

2 **Formal Parameters.**

`result` shall point to a C descriptor for a Fortran pointer. It is updated using information from the `source` and `lower_bounds` arguments.

`source` shall be NULL or point to a C descriptor for an assumed-shape array, an allocatable object, or a data pointer object.

`lower_bounds` is ignored if the rank is not greater than zero. Otherwise, the number of elements in the array `lower_bounds` shall be greater than or equal to the rank specified in the `source` C descriptor. The elements provide the lower bounds for each corresponding dimension of the `result` C descriptor. The extents and memory strides are copied from the `source` C descriptor.

3 CFI_setpointer updates the C descriptor pointed to by `result` with information in the C descriptor pointed to by `source` and the `lower_bounds` argument.

4 If `source` is NULL or points to a C descriptor for an allocatable object that is not allocated or a pointer that is not associated, the updated C descriptor describes a disassociated pointer.

5 If `source` is not NULL, the `elem_len`, `rank`, and `type` members of the C descriptor pointed to by `source` determine the corresponding members of the updated C descriptor.

6 If `lower_bounds` is NULL or the rank is zero, the C descriptor pointed to by `result` becomes a C descriptor for the object described by the C descriptor pointed to by `source`.

7 On successful execution of CFI_setpointer, the C descriptor pointed to by `result` is updated. If an error is detected, that C descriptor is not modified.

8 **Result Value.** The function returns an error indicator.

9 **Example.** If `ptr` already points to a C descriptor for an array pointer of rank 1, the following code makes it point to this with lower bound 0.

```
CFI_index_t lower_bounds[1];
int ind;
lower_bounds[0] = 0;
ind = CFI_setpointer ( ptr, ptr, lower_bounds );
```

5.2.6 Use of C descriptors

1 A C descriptor shall not be initialized, updated or copied other than by calling the functions specified here. A C descriptor that is pointed to by a formal parameter that corresponds to a Fortran dummy argument with the INTENT(IN) attribute shall not be updated.

2 Calling CFI_allocate or CFI_deallocate for a C descriptor changes the allocation status of the Fortran variable it describes and causes the allocation status of any associated allocatable variable to change accordingly (6.7.1.3 of ISO/IEC 1539-1:2010).

3 A C descriptor that is pointed to by a formal parameter or actual argument that corresponds to a Fortran dummy argument in a BIND(C) interface shall describe an object that is acceptable to both Fortran and C with the type specified in its type member.

5.2.7 Restrictions on lifetimes

1 When a Fortran object is deallocated, execution of its host instance is completed, or its association status becomes undefined, all C descriptors and C pointers to any part of it become undefined, and any further use of them is undefined behavior (ISO/IEC 9899:1999 3.4.3).

2 A C descriptor that is pointed to by a formal parameter that corresponds to a Fortran dummy argument becomes undefined on return from a call to the function from Fortran. If the dummy argument does not have any of the TARGET, ASYNCHRONOUS or VOLATILE attributes, all C pointers to any part of the object it describes become undefined on return from the call, and any further use of them is undefined behavior.

3 If a pointer to a C descriptor is passed as an actual argument, the lifetime of the C descriptor and that of the object it describes (ISO/IEC 9899:1999 6.2.4) shall not end before the return from the function call. A Fortran pointer variable that is associated with the object described by a C descriptor shall not be accessed beyond the end of the lifetime of the C descriptor and the object it describes.

5.2.8 Interoperability of procedures and procedure interfaces

1 The rules in this subclause replace the contents of paragraphs one and two of subclause 15.3.7 of ISO/IEC 1539-1:2010 entirely.

2 A Fortran procedure is interoperable if it has the BIND attribute, that is, if its interface is specified with a *proc-language-binding-spec*.

3 A Fortran procedure interface is interoperable with a C function prototype if

- (1) the interface has the BIND attribute,
- (2) either
  - (a) the interface describes a function whose result variable is a scalar that is interoperable with the result of the prototype or
  - (b) the interface describes a subroutine and the prototype has a result type of void,
- (3) the number of dummy arguments of the interface is equal to the number of formal parameters of the prototype,
- (4) the prototype does not have variable arguments as denoted by the ellipsis (...),
- (5) any dummy argument with the VALUE attribute is interoperable with the corresponding formal parameter of the prototype, and
- (6) any dummy argument without the VALUE attribute corresponds to a formal parameter of the prototype that is of a pointer type, and either
  - (a) the dummy argument is interoperable with an entity of the referenced type (ISO/IEC 9899:1999, 6.2.5, 7.17, and 7.18.1) of the formal parameter,
  - (b) the dummy argument is a nonallocatable, nonpointer variable of type CHARACTER with assumed length, and corresponds to a formal parameter of the prototype that is a pointer to CFI_cdesc_t,
  - (c) the dummy argument is allocatable, assumed-shape, assumed-rank, or a pointer, and corresponds to a formal parameter of the prototype that is a pointer to CFI_cdesc_t, or
  - (d) the dummy argument is assumed-type and not assumed-shape or assumed-rank, and corresponds to a formal parameter of the prototype that is a pointer to void.

4 If a dummy argument in an interoperable interface is of type CHARACTER and is allocatable or a pointer, its character length shall be deferred.

5 If a dummy argument in an interoperable interface is allocatable, assumed-shape, assumed-rank, or a pointer, the corresponding formal parameter is interpreted as a pointer to a C descriptor for the effective argument in a reference to the procedure. The C descriptor shall describe an object of interoperable type and type parameters

with the same characteristics as the effective argument; the type member shall have a value from Table 5.2 that depends on the effective argument as follows:

- if the dynamic type of the effective argument is an interoperable type listed in Table 5.2, the corresponding value for that type;
- otherwise, CFI_type_other.

6 An absent actual argument in a reference to an interoperable procedure is indicated by a corresponding formal parameter with the value NULL.

6 Required editorial changes to ISO/IEC 1539-1:2010(E)

6.1 General

1 The following editorial changes, if implemented, would provide the facilities described in foregoing clauses of this Technical Report. Descriptions of how and where to place the new material are enclosed in braces . Edits to different places within the same clause are separated by horizontal lines.

2 In the edits, except as specified otherwise by the editorial instructions, underwave (underwave) and strike-out (~~strike-out~~) are used to indicate insertion and deletion of text.

6.2 Edits to Introduction

1 {In paragraph 1 of the Introduction }

2 After "informally known as Fortran 2008"
insert ", plus the facilities defined in ISO/IEC TR 29113:2011".

---

3 {After paragraph 3 of the Introduction, insert new paragraph}

4 ISO/IEC TR 29113 provides additional facilities with the purpose of improving interoperability with the C programming language:

- assumed-type objects provide more convenient interoperability with C pointers;
- assumed-rank objects provide more convenient interoperability with the C memory model;
- it is now possible for a C function to interoperate with a Fortran procedure that has an allocatable, assumed character length, assumed-shape, optional, or pointer dummy data object.

6.3 Edits to clause 1

1 {Insert new term definitions before term **1.3.9 attribute**}

2 **1.3.8a**
assumed rank
⟨dummy variable⟩ the property of assuming the rank from its effective argument (5.3.8.7, 12.5.2.4)

3 **1.3.8b**
assumed type
⟨dummy variable⟩ being declared as TYPE (*) and therefore assuming the type and type parameters from its effective argument (4.3.1)

---

4 {Insert new term definition before **1.3.20 character context**}

5 **1.3.19a**
C descriptor
struct of type CFI_cdesc_t defined in the header `ISO_Fortran_binding.h` (15.5)

---

6 {Insert new subclause before 1.6.2 Fortran 2003 compatibility}

7 1.6.1a Fortran 2008 compatibility

8 This part of ISO/IEC 1539 is an upward compatible extension to the preceding Fortran International Standard,

ISO/IEC 1539-1:2010(E). Any standard-conforming Fortran 2008 program remains standard-conforming under this part of ISO/IEC 1539.

6.4 Edits to clause 4

1 {In 4.3.1.1 Type specifier syntax, insert additional production for R403 *declaration-type-spec* after the one for CLASS (*)}

or TYPE (*)

---

2 {In 4.3.1.2 TYPE, edit the first paragraph as follows}

3 A TYPE type specifier is used to declare entities that are of assumed type, or of an intrinsic or derived type.

---

4 {In 4.3.1.2 TYPE, insert new paragraphs at the end of the subclause}

5 An entity that is declared using the TYPE(*) type specifier has assumed type and is an unlimited polymorphic entity (4.3.1.3). Its dynamic type and type parameters are assumed from its associated effective argument.

C407a An assumed-type entity shall be a dummy variable that does not have the ALLOCATABLE, CODIMENSION, POINTER or VALUE attributes.

C407b An assumed-type variable name shall not appear in a designator or expression except as an actual argument corresponding to a dummy argument that is assumed-type, or the first argument to the intrinsic and intrinsic module functions IS_CONTIGUOUS, LBOUND, PRESENT, RANK, SHAPE, SIZE, UBOUND, or C_LOC.

6.5 Edits to clause 5

1 {In 5.3.1 Constraints, replace C516 with}

C516 The ALLOCATABLE or POINTER attribute shall not be specified for a default-initialized dummy argument of a procedure that has a *proc-language-binding-spec*.

---

2 {In 5.3.7 CONTIGUOUS attribute, edit C530 as follows}

C530 An entity with the CONTIGUOUS attribute shall be an array pointer, ~~or~~ an assumed-shape array, or have assumed rank.

---

3 {In 5.3.7 CONTIGUOUS attribute, edit paragraph 1 as follows}

4 The CONTIGUOUS attribute specifies that an assumed-shape array can only be argument associated with a contiguous effective argument, ~~or~~ that an array pointer can only be pointer associated with a contiguous target, or that an assumed-rank object can only be argument associated with a scalar or contiguous effective argument.

---

5 {In 5.3.7 CONTIGUOUS attribute, paragraph 2, item (3)}

6 Change first “array” to “or assumed-rank dummy argument”,
change second “array” to “object”.

---

7 {In 5.3.8.1 General, edit paragraph 1 as follows}

8 The DIMENSION attribute specifies that an entity has assumed rank or is an array. An assumed-rank entity has the rank and shape of its associated actual argument; otherwise, the~~The~~ rank or rank and shape is specified by its *array-spec*.

9 {In 5.3.8.1 General, insert additional production for R515 *array-spec*, after *implied-shape-spec-list*}

or *assumed-rank-spec*

---

10 {At the end of 5.3.8, immediately before 5.3.9, insert new subclause}

11 5.3.8.7 Assumed-rank entity

12 An assumed-rank entity is a dummy variable whose rank is assumed from its effective argument; this rank may be zero. An assumed-rank entity is declared with an *array-spec* that is an *assumed-rank-spec*.

R522a *assumed-rank-spec* **is** ..

C535a An assumed-rank entity shall be a dummy variable that does not have the CODIMENSION or VALUE attribute.

C535b An assumed-rank variable name shall not appear in a designator or expression except as an actual argument corresponding to a dummy argument that is assumed-rank, the argument of the C_LOC function in the ISO_C_BINDING intrinsic module, or the first argument in a reference to an intrinsic inquiry function.

13 The intrinsic function RANK can be used to inquire about the rank of a data object.

6.6 Edits to clause 6

1 {In 6.5.4 Simply contiguous array designators, paragraph 2, edit the second bullet item as follows}

- an *object-name* that is not a pointer, not ~~or~~ assumed-shape, and not assumed-rank,

---

2 {In 6.7.3.2 Deallocation of allocatable variables, append to paragraph 6}

3 If a Fortran procedure that has an INTENT (OUT) allocatable dummy argument is invoked by a C function and the corresponding argument in the C function call is a C descriptor that describes an allocated allocatable variable, the variable is deallocated on entry to the Fortran procedure. When a C function is invoked from a Fortran procedure via an interface with an INTENT (OUT) allocatable dummy argument and the corresponding actual argument in the reference of the C function is an allocated allocatable variable, the variable is deallocated on invocation (before execution of the C function begins).

6.7 Edits to clause 12

1 {In 12.3.2.2, edit paragraph 1 as follows}

2 The characteristics of a dummy data object are its type, its type parameters (if any), its shape (unless it is assumed-rank), its corank, its codimensions, its intent (5.3.10, 5.4.10), whether it is optional (5.3.12, 5.4.10), whether it is allocatable (5.3.3), whether it has the ASYNCHRONOUS (5.3.4), CONTIGUOUS (5.3.7), VALUE (5.3.18), or VOLATILE (5.3.19) attributes, whether it is polymorphic, and whether it is a pointer (5.3.14, 5.4.12) or a target (5.3.17, 5.4.15). If a type parameter of an object or a bound of an array is not a constant expression, the exact dependence on the entities in the expression is a characteristic. If a rank, shape, size, type, or type parameter is assumed or deferred, it is a characteristic.

---

3 {In 12.4.2.2 Explicit interface, after item (2)(c) insert new item}

4 (c2) has assumed rank,

---

5 {In 12.5.2.4 Ordinary dummy variables, append to paragraph 2}

6 If the actual argument is of a derived type that has type parameters, type-bound procedures, or final subroutines, the dummy argument shall not be assumed type.

---

7 {In 12.5.2.4 Ordinary dummy variables, paragraphs 3 and 4}

8 Change "not assumed shape" to "explicit-shape or assumed-size" (twice).

---

9 {In 12.5.2.4 Ordinary dummy variables, paragraph 9}

10 After "dummy argument is a scalar"
Change "or" to ", has assumed rank, or is".

---

11 {In 12.5.2.4 Ordinary dummy variables, insert new paragraph after paragraph 14}

12 An actual argument of any rank may correspond to an assumed-rank dummy argument. The rank and shape of the dummy argument are the rank and shape of the corresponding actual argument. If the rank is nonzero, the lower and upper bounds of the dummy argument are those that would be given by the intrinsic functions LBOUND and UBOUND respectively if applied to the actual argument, except that when the actual argument is assumed size, the upper bound of the last dimension of the dummy argument is 2 less than the lower bound of that dimension.

---

13 {In 12.6.2.2 Function subprogram, edit C1255 as follows}

C1255 (R1229) If *proc-language-binding-spec* is specified for a procedure, each of the procedure's dummy arguments shall be an ~~nonoptional~~ interoperable variable (15.3.5, 15.3.6) that does not have both the OPTIONAL and VALUE attributes, or an ~~nonoptional~~ interoperable procedure (15.3.7). If *proc-language-binding-spec* is specified for a function, the function result shall be an interoperable scalar variable.

6.8 Edits to clause 13

1 {In 13.5 Standard generic intrinsic procedures, Table 13.1, LBOUND and UBOUND intrinsic functions}

2 Delete " of an array" (twice).

---

3 {In 13.5 Standard generic intrinsic procedures, Table 13.1}

4 Insert new entry into the table, alphabetically

5 RANK (A) I Rank of a data object.

---

6 {In 13.7.86, IS_CONTIGUOUS, edit paragraph 3 as follows}

7 **Argument.** ARRAY may be of any type. It shall be an array or an assumed-rank object. If it is a pointer it shall be associated.

---

8 {In 13.7.86, IS_CONTIGUOUS, edit paragraph 5 as follows}

9 **Result Value.** The result has the value true if ARRAY has rank zero or is contiguous, and false otherwise.

---

10 {In 13.7.90 LBOUND, edit paragraph 1 as follows}

11 **Description.** Lower bound(s) ~~of an array~~.

---

12 {In 13.7.90 LBOUND, edit paragraph 3, ARRAY argument, as follows}

ARRAY shall be an array or assumed-rank object of any type. It shall not be an unallocated allocatable variable or a pointer that is not associated.

---

13 {In 13.7.93 LEN, paragraph 3}

14 Change "a type character scalar or array"
to "of type character".

---

15 {Immediately before subclause 13.8.138 REAL, insert new subclause}

16 **13.7.137a RANK (A)**

17 **Description.** Rank of a data object.

18 **Class.** Inquiry function.

19 **Argument.** A shall be a data object of any type.

20 **Result Characteristics.** Default integer scalar.

21 **Result Value.** The result is the rank of A.

22 **Example.** If X is declared as REAL X (:, :, :), the result has the value 3.

---

23 {In 13.7.149 SHAPE, replace paragraph 5 with}

24 **Result Value.** The result has a value equal to [(SIZE(SOURCE, i, KIND), i=1, RANK(SOURCE))].

---

25 {In 13.7.156 SIZE, edit paragraph 3, argument ARRAY, as follows}

ARRAY shall be an array or assumed-rank object of any type. It shall not be an unallocated allocatable variable or a pointer that is not associated. If ARRAY is an assumed-size array, DIM shall be present with a value less than the rank of ARRAY.

---

26 {In 13.7.156 SIZE, replace paragraph 5 with}

27 **Result Value.** If ARRAY is an assumed-rank object associated with an assumed-size array and DIM is present with a value equal to the rank of ARRAY, the result is −1; otherwise, if DIM is present, the result has a value equal to the extent of dimension DIM of ARRAY. If DIM is not present, the result has a value equal to PRODUCT([(SIZE(ARRAY, i, KIND), i=1, RANK(ARRAY))]).

---

28 {In 13.7.160 STORAGE_SIZE, paragraph 3}

29 Change "a scalar or array of any type"
to "a data object of any type".

---

30 {In 13.7.171 UBOUND, paragraph 1}

31 Delete " of an array".

---

32 {In 13.7.171 UBOUND, paragraph 3, ARRAY argument}

33 After "shall be an array"
insert "or assumed-rank object".

34 {In 13.7.171 UBOUND, edit paragraph 5 as follows}

35 **Result Value.**

Case (i): For an array section or for an array expression, other than a whole array, UBOUND (ARRAY, DIM) has a value equal to the number of elements in the given dimension~~; otherwise,~~.

Case (ii): For an assumed-rank object associated with an assumed-size array, UBOUND(ARRAY, n) where n is the rank of ARRAY has a value equal to LBOUND(ARRAY, n) − 2.

Case (iii): Otherwise, UBOUND(ARRAY, DIM) has a value equal to the upper bound for subscript DIM of ARRAY if dimension DIM of ARRAY does not have size zero and has the value zero if dimension DIM has size zero.

Case (iv): UBOUND (ARRAY) has a value whose i^{th} element is equal to UBOUND (ARRAY, i), for $i = 1, 2, \ldots, n$, where n is the rank of ARRAY.

6.9 Edits to clause 15

1 {In 15.1 General, at the end of the subclause, insert new paragraph}

2 The header `ISO_Fortran_binding.h` provides definitions and prototypes to enable a C function to interoperate with a Fortran procedure with an allocatable, assumed character length, assumed-shape, assumed-rank, or pointer dummy data object.

---

3 {In 15.3.7 Interoperability of procedures and procedure interfaces, paragraph 2, edit item (5) as follows}

(5) any dummy argument without the VALUE attribute corresponds to a formal parameter of the prototype that is of pointer type, and either

- (a) the dummy argument is interoperable with an entity of the referenced type (ISO/IEC 9899:1999, 6.25, 7.17, and 7.18.1) of the formal parameter,
- (b) the dummy argument is a nonallocatable, nonpointer variable of type CHARACTER with assumed length, and corresponds to a formal parameter of the prototype that is a pointer to CFI_desc_t,
- (c) the dummy argument is allocatable, assumed-shape, assumed-rank, or a pointer, and corresponds to a formal parameter of the prototype that is a pointer to CFI_cdesc_t, or
- (d) the dummy argument is assumed-type and not allocatable, assumed-shape, assumed-rank, or a pointer, and corresponds to a formal parameter of the prototype that is a pointer to void,

(5a) each allocatable or pointer dummy argument of type CHARACTER has deferred character length, and,

---

4 {In 15.3.7 Interoperability of procedures and procedure interfaces, insert new paragraphs at the end of the subclause}

5 If a dummy argument in an interoperable interface is allocatable, assumed-shape, assumed-rank, or a pointer, the corresponding formal parameter is interpreted as a pointer to a C descriptor for the effective argument in a reference to the procedure. The C descriptor shall describe an object of interoperable type and type parameters with the same characteristics as the effective argument.

6 An absent actual argument in a reference to an interoperable procedure is indicated by a corresponding formal parameter with the value NULL.

---

7 {At the end of clause 15}

8 Insert subclause 5.2 of this Technical Report as subclause 15.5, including subclauses 5.2.1 to 5.2.8 as subclauses 15.5.1 to 15.5.8.

6.10 Edits for annex C

1 {In C.11 Clause 15 notes, at the end of the subclause}

2 Insert subclauses A.1.1 to A.1.6 as subclauses C.11.6 to C.11.11.

3 Insert subclause A.2.1 as C.11.12 with the revised title "Processing assumed-shape arrays in C".

4 Insert subclauses A.2.2 to A.2.4 as subclauses C.11.13 to C.11.15.

Annex A

(Informative)

Extended notes

A.1 Clause 2 notes

A.1.1 Using assumed type in the context of interoperation with C

1 The mechanism for handling unlimited polymorphic entities whose dynamic type is interoperable with C is designed to handle the following two situations:

(1) An entity corresponding to a C pointer to void. This is a start address, and no further information about the entity is available via the language rules. This situation occurs if the entity is a nonallocatable nonpointer scalar or is an array of assumed size.

(2) An entity of interoperable dynamic type for which additional information on state, type and size is implicitly provided with the entity. All assumed-type entities of assumed shape or rank fall into this category.

2 For entities in the first category, it is the programmer's responsibility to explicitly provide additional information on the size (e.g., in units of bytes) and possibly also the type of the object pointed to.

3 Within C, entities in the second category require the use of a C descriptor. The rules of the language ensure that, within Fortran, entities of the first category cannot be used in a context where the additional information needed for the second category is required but unavailable. However, it is possible to use entities of the second category in a context where the Fortran processor simply needs to extract the starting address from the entity to convert it to the first category. Within C, the programmer must explicitly perform this extraction.

4 The examples A.1.2 - A.1.4 illustrate some uses of assumed type entities.

A.1.2 Example for mapping of interfaces with void * C parameters to Fortran

1 A C interface for message passing or I/O functionality could be provided in the form

```
int EXAMPLE_send(const void *buffer, size_t buffer_size, const HANDLE_t *handle);
```

2 where the `buffer_size` argument is given in units of bytes, and the `handle` argument (which is of a type aliased to `int`) provides information about the target the buffer is to be transferred to. In this example, type resolution is not required.

3 The first method provides a thin binding; a call to `EXAMPLE_send` from Fortran directly invokes the C function.

```
interface
  integer(c_int) function EXAMPLE_send(buffer, buffer_size, handle)  &
                  bind(c,name='EXAMPLE_send')
    use,intrinsic :: iso_c_binding
    type(*), dimension(*), intent(in) :: buffer
    integer(c_size_t), value :: buffer_size
    integer(c_int), intent(in) :: handle
  end function EXAMPLE_send
end interface
```

4 It is assumed that this interface is declared in the specification part of a module `mod_EXAMPLE_old`. Example invocations from Fortran then are

```
use, intrinsic :: iso_c_binding
use mod_EXAMPLE_old

real(c_float) :: x(100)
integer(c_int) :: y(10,10)
real(c_double) :: z
integer(c_int) :: status, handle
:
! assign values to x, y, z and initialize handle
:
! send values in x, y, and z using EXAMPLE_send:
status = EXAMPLE_send(x, c_sizeof(x), handle)
status = EXAMPLE_send(y, c_sizeof(y), handle)
status = EXAMPLE_send((/ z /), c_sizeof(z), handle)
```

5 In these invocations, `x` and `y` are passed by address, and for `y` the sequence association rules (12.5.2.11 of ISO/IEC 1539-1:2010) allow this. For `z`, it is necessary to explicitly create an array expression.

```
status = EXAMPLE_send(y, c_sizeof(y(:,1)), handle)
```

6 passes the first column of `y` (again by address).

```
status = EXAMPLE_send(y(1,5), c_sizeof(y(:,5)), handle)
```

7 passes the fifth column of `y` using the sequence association rules.

8 The second method provides a Fortran interface which is easier to use, but requires writing a separate C wrapper routine; this is commonly called a "fat binding". In this implementation, a C descriptor is created because the buffer is declared with assumed rank in the Fortran interface; the use of an optional argument is also demonstrated.

```
interface
  subroutine example_send(buffer, handle, status) &
             BIND(C, name='EXAMPLE_send_fortran')
    use,intrinsic :: iso_c_binding
    type(*), dimension(..), contiguous, intent(in) :: buffer
    integer(c_int), intent(in) :: handle
    integer(c_int), intent(out), optional :: status
  end subroutine example_send
end interface
```

9 It is assumed that this interface is declared in the specification part of a module `mod_EXAMPLE_new`. Example invocations from Fortran then are

```
use, intrinsic :: iso_c_binding
use mod_EXAMPLE_new

type, bind(c) :: my_derived
  integer(c_int) :: len_used
  real(c_float) ::  stuff(100)
end type
type(my_derived) :: w(3)
real(c_float) :: x(100)
integer(c_int) :: y(10,10)
real(c_double) :: z
integer(c_int) :: status, handle
```

```
:
! assign values to w, x, y, z and initialize handle
:
! send values in w, x, y, and z using EXAMPLE_send
call EXAMPLE_send(w, handle, status)
call EXAMPLE_send(x, handle)
call EXAMPLE_send(y, handle)
call EXAMPLE_send(z, handle)

call EXAMPLE_send(y(:,5), handle) ! fifth column of y
call EXAMPLE_send(y(1,5), handle) ! scalar y(1,5) passed by descriptor
```

10 However, the following call from Fortran is not allowed

```
type(*) :: d(*) ! is a dummy argument
:
call EXAMPLE_send(d(1:4), handle, status)
```

11 The wrapper routine implemented in C reads

```
#include "ISO_Fortran_binding.h"

void EXAMPLE_send_fortran(const CFI_cdesc_t *buffer,
                          const HANDLE_t *handle, int *status) {
  int status_local;
  size_t buffer_size;
  int i;

  buffer_size = buffer->elem_len;
  for (i=0; i<buffer->rank; i++) {
    buffer_size *= buffer->dim[i].extent;
  }
  status_local = EXAMPLE_send(buffer->base_addr,buffer_size, handle);
  if (status != NULL) *status = status_local;
}
```

A.1.3 A constructor for an interoperable unlimited polymorphic entity

1 Leave space for RB replacement example.

A.1.4 Using assumed-type dummy arguments

Example of TYPE (*) for an abstracted message passing routine with two arguments.

1 The first argument is a data buffer of type (void *) and the second argument is an integer indicating the size of the buffer to be transferred. The generic interface accepts both 32-bit and 64-bit integers as the buffer size, converting them to “C int” since the caller will probably want to use default integer and the size of default integer varies depending on the compiler and option used.

2 The C prototype is:

```
   void EXAMPLE_send ( void * buffer, int n);
```

3 and it is assumed that an implementation exists.

4 The Fortran module has the public generic interface:

```
   interface EXAMPLE_send
      subroutine EXAMPLE_send (buffer, n) bind(c,name="EXAMPLE_send")
         use,intrinsic :: iso_c_binding
         type(*),dimension(*) :: buffer
         integer(c_int),value :: n
      end subroutine EXAMPLE_send
      module procedure EXAMPLE_send_i8
   end interface EXAMPLE_send
```

5 and the module procedure

```
   subroutine EXAMPLE_send_i8 (buffer, n)
      use,intrinsic :: iso_c_binding
      type(*),dimension(*) :: buffer
      integer(selected_int_kind(17)) :: n
      call EXAMPLE_send(buffer, int(n,c_int))
   end subroutine EXAMPLE_send_i8
```

A.1.5 Casting TYPE (*) in Fortran

Example of how to gain access to a TYPE (*) argument

1 It is possible to "cast" a TYPE (*) object to a usable type, exactly as is done for void * objects in C. For example, this code fragment casts a block of memory to be used as an integer array.

```
   subroutine process(block, nbytes)
      type(*), target :: block(*)
      integer, intent(in) :: nbytes ! Number of bytes in block(*)

      integer :: nelems
      integer, pointer :: usable(:)

      nelems=nbytes/(bit_size(usable)/8)
      call c_f_pointer (c_loc(block), usable, [nelems] )
      usable=0 ! Instead of the disallowed block=0
   end subroutine
```

A.1.6 Simplifying interfaces for arbitrary rank procedures

Example of assumed-rank usage in Fortran

1 Assumed-rank variables are not restricted to be assumed-type. For example, many of the IEEE intrinsic procedures in Clause 14 of ISO/IEC 1539-1:2010 could be written using an assumed-rank dummy argument instead of writing 16 separate specific routines, one for each possible rank.

2 An example of an assumed-rank dummy argument for the specific procedures for the IEEE_SUPPORT_DIVIDE function.

```
   interface ieee_support_divide
       module procedure ieee_support_divide_noarg
       module procedure ieee_support_divide_onearg_r4
       module procedure ieee_support_divide_onearg_r8
   end interface ieee_support_divide

   ...
```

```
   logical function ieee_support_divide_noarg ()
      ieee_support_divide_noarg = .true.
   end function ieee_support_divide_noarg

   logical function ieee_support_divide_onearg_r4 (x)
      real(4),dimension(..) :: x
      ieee_support_divide_onearg_r4 = .true.
   end function ieee_support_divide_onearg_r4

   logical function ieee_support_divide_onearg_r8 (x)
      real(8),dimension(..) :: x
      ieee_support_divide_onearg_r8 = .true.
   end function ieee_support_divide_onearg_r8
```

A.2 Clause 5 notes

A.2.1 Dummy arguments of any type and rank

1 The example shown below calculates the product of individual elements of arrays A and B and returns the result in array C. The Fortran interface of `elemental_mult` will accept arguments of any type and rank. However, the C function will return an error code if any argument is not a two-dimensional `int` array. Note that the arguments are permitted to be array sections, so the C function does not assume that any argument is contiguous.

2 The Fortran interface is:

```
   interface
      function elemental_mult(A, B, C) bind(C,name="elemental_mult_c"), result(err)
         use,intrinsic  :: iso_c_binding
         integer(c_int) :: err
         type(*), dimension(..) :: A, B, C
      end function elemental_mult
   end interface
```

3 The definition of the C function is:

```
  #include "ISO_Fortran_binding.h"

  int elemental_mult_c(CFI_cdesc_t * a_desc,
                       CFI_cdesc_t * b_desc, CFI_cdesc_t * c_desc) {
   size_t i, j, ni, nj;

   int err = 1;  /* this error code represents all errors */

   char * a_col = (char*) a_desc->base_addr;
   char * b_col = (char*) b_desc->base_addr;
   char * c_col = (char*) c_desc->base_addr;
   char *a_elt, *b_elt, *c_elt;

   /* only support integers */
   if (a_desc->type != CFI_type_int || b_desc->type != CFI_type_int ||
       c_desc->type != CFI_type_int) {
```

```
        return err;
     }

     /* only support two dimensions */
     if (a_desc->rank != 2 || b_desc->rank != 2 || c_desc->rank != 2) {
        return err;
     }

     ni = a_desc->dim[0].extent;
     nj = a_desc->dim[1].extent;

     /* ensure the shapes conform */
     if (ni != b_desc->dim[0].extent || ni != c_desc->dim[0].extent) return err;
     if (nj != b_desc->dim[1].extent || nj != c_desc->dim[1].extent) return err;

     /* multiply the elements of the two arrays */
     for (j = 0; j < nj; j++) {
       a_elt = a_col;
       b_elt = b_col;
       c_elt = c_col;
       for (i = 0; i < ni; i++) {
         *(int*)a_elt = *(int*)b_elt * *(int*)c_elt;
         a_elt += a_desc->dim[0].sm;
         b_elt += b_desc->dim[0].sm;
         c_elt += c_desc->dim[0].sm;
       }
       a_col += a_desc->dim[1].sm;
       b_col += b_desc->dim[1].sm;
       c_col += c_desc->dim[1].sm;
     }
     return 0;
   }
```

4 The following example provides functions that can be used to copy an array described by a CFI_cdesc_t descriptor to a contiguous buffer. The input array need not be contiguous.

5 The C functions are:

```
#include "ISO_Fortran_binding.h"
/* other necessary includes omitted */

/*
 * Returns the number of elements in the object described by desc.
 * If it is an array, it need not be contiguous.
 * (The number of elements could be zero).
 */
size_t numElements(const CFI_cdesc_t * desc) {
   CFI_rank_t r;
   size_t num = 1;

   for (r = 0; r < desc->rank; r++) {
      num *= desc->dim[r].extent;
   }
   return num;
}
```

```
/*
 * Auxiliary routine to loop over a particular rank.
 */
static void * _copyToContiguous (const CFI_cdesc_t * vald,
               void * output, const void * input, CFI_rank_t rank) {
   CFI_index_t e;

   if (rank == 0) {
      /* copy scalar element */
      memcpy (output, input, vald->elem_len);
      output = (void *)((char *)output + vald->elem_len);
   }
   else {
      for (e = 0; e < vald->dim[rank-1].extent; e++) {
         /* recurse on subarrays of lesser rank */
         output = _copyToContiguous (vald, output, input, rank-1);
         input = (void *) ((char *)input + vald->dim[rank].sm);
      }
   }
   return output;
}

/*
 * General routine to copy the elements in the array described by vald
 * to buffer, as done by sequence association.  The array itself may
 * be non-contiguous.  This is not the most efficient approach.
 */
void copyToContiguous (void * buffer, const CFI_cdesc_t * vald) {
   _copyToContiguous (vald, buffer, vald->base_addr, vald->rank);
}

/*
 * Send the data described by vald using the function send_contig, which
 * requires a contiguous buffer.  If needed, copy the data to a contiguous
 * buffer before calling send_contig.
 */
void send_data (CFI_cdesc_t * vald) {
  size_t num_bytes = numElements(vald)*vald->elem_len;
  if (CFI_is_contiguous(vald)) {
     /* the data described by vald is already contiguous, just send it */
     send_contig(vald->base_addr, num_bytes);
  }
  else if (num_bytes) {
     void * buffer = malloc(num_bytes);
     copyToContiguous(buffer, vald);

     /* send the contiguous copy of data described by vald */
     send_contig(buffer, num_bytes);

     free(buffer);
  }
}
```

A.2.2 Changing the attributes of an array

1 A C programmer might want to call more than one Fortran procedure and the attributes of an array involved might differ between the procedures. In this case, it is necessary to set up more than one C descriptor for the array. For example, this code fragment initializes two C descriptors of rank 2, calls a procedure that allocates the array described by the first descriptor, copies the `base_addr` pointer and `dim` array to the second descriptor, then calls a procedure that expects an assumed-shape array.

```
CFI_CDESC_T(2) loc_alloc, loc_assum;
CFI_cdesc_t * desc_alloc = (CFI_cdesc_t *)&loc_alloc,
            * desc_assum = (CFI_cdesc_t *)&loc_assum;
CFI_dim_t dims[2];
CFI_rank_t rank = 2;
int flag;

flag = CFI_establish(desc_alloc,
                     NULL,
                     CFI_attribute_allocatable,
                     CFI_type_double,
                     sizeof(double),
                     rank,
                     dims);

Fortran_factor (desc_alloc, ...); /* Allocates array described by desc_alloc */

/* Use dim information from the allocated array in the assumed shape one */
flag = CFI_establish(desc_assum,
                     desc_alloc->base_addr,
                     CFI_attribute_assumed,
                     CFI_type_double,
                     sizeof(double),
                     rank,
                     desc_alloc->dim);

Fortran_solve (desc_assum, ...); /* Uses array allocated in Fortran_factor */
```

A.2.3 Example for creating an array slice in C

1 Given the Fortran subprogram

```
subroutine set_all(int_array, val) bind(c)
  integer(c_int) :: int_array(:)
  integer(c_int), value :: val
  int_array = val
end subroutine
```

2 that sets all the elements of an array and the Fortran interface

```
interface
  subroutine set_odd(int_array, val) bind(c)
    use, intrinsic :: iso_c_binding, only : c_int
    integer(c_int) :: int_array(:)
    integer(c_int), value :: val
  end subroutine
end interface
```

3 for a C function that sets every second array element, beginning with the first one, the implementation in C reads

```
#include "ISO_Fortran_binding.h"

void set_odd(CFI_cdesc_t *int_array, int val) {
   CFI_dim_t dims[1];
   CFI_CDESC_T(1) array;
   int status;
   /* the following is equivalent to saying int_array(1::2) in Fortran */
   dims[0].lower_bound = 0;
   dims[0].extent      = (int_array->dim[0].extent + 1)/2;
   dims[0].sm          = 2*int_array->dim[0].sm;
   /* Update the descriptor with the new information */
   status = CFI_establish( (CFI_cdesc_t *) &array,
                            int_array->base_addr,
                            CFI_attribute_assumed,
                            int_array->type,
                            int_array->elem_len,
                            /* rank */ 1,
                            dims);

   set_all( (CFI_cdesc_t *) &array, val);

   /* here one could make use of int_array and access all its data */

}
```

4 A copy of the incoming descriptor is created because the call to CFI_establish() irreversibly modifies the descriptor. At least the `extent` and `sm` members of `int_array->dim[0]` will be modified: `sm` will be doubled, and the value of the `extent` member will be changed to (`extent` + 1)/2.

5 Without such a copy, it would not be possible to access all the data of the incoming descriptor after the invocation of CFI_establish(), which may be a problem for the remaining part of the implementation, or – after the call site – for a C function which invokes `set_odd()` (see below).

6 Let invocation of `set_odd()` from a Fortran program be done as follows:

```
integer(c_int) :: d(5)
d = (/ 1, 2, 3, 4, 5 /)
call set_odd(d, -1)
write(*, *) d
```

7 Then, the program will print

```
   -1    2   -1    4   -1
```

8 During execution of the subprogram `set_all()`, its dummy object `int_array` would appear to be an array of size 3 with lower bound 1 and upper bound 3.

9 It is also possible to invoke `set_odd()` from C. However, it is the C programmer's responsibility to make sure that all members of the descriptor have the correct value on entry to the function. Inserting additional checking into the function's implementation could alleviate this problem.

```
/* necessary includes omitted */
#define ARRAY_SIZE 5

```

```
CFI_CDESC_T(1) d;
CFI_dim_t dims[1];
CFI_index_t subscripts[1];
void *base;
int i, status;

base = malloc(ARRAY_SIZE*sizeof(int));
dims[0].lower_bound = 0;
dims[0].extent      = ARRAY_SIZE;
dims[0].sm          = sizeof(int);
/* different from CFI_allocate, stride must be specified here */
status = CFI_establish( (CFI_cdesc_t *) &d,
                         base,
                         CFI_attribute_assumed,
                         CFI_type_int,
                         sizeof(int),
                         /* rank */ 1,
                         dims);

set_odd( (CFI_cdesc_t *) &d, -1);

for (i=0; i<ARRAY_SIZE; i++) {
  subscripts[1] = i;
  printf("   %d",*((int *)CFI_address( (CFI_cdesc_t *) &d, subscripts)));
}
printf("\n");
free(base);
```

10 This C program will print (apart from formatting) the same output as the Fortran program above. It also demonstrates how an assumed shape entity is dynamically generated within C.

A.2.4 Example for handling objects with the POINTER attribute

1 The following C function modifies a pointer to an integer variable to point at a global variable defined inside C:

```
#include "ISO_Fortran_binding.h"

int y = 2;

void change_target(CFI_cdesc_t *ip) {
  if (ip->attribute == CFI_attribute_pointer && ip->rank == 0) {
    CFI_establish(ip,
                  &y,
                  CFI_attribute_pointer,
                  CFI_type_int,
                  sizeof(int),
                  /* rank */ 0,
                  /* dim  */ NULL);
  }
}
```

2 The following Fortran code

```
use, intrinsic :: iso_c_binding

```

```
interface
  subroutine change_target(ip) bind(c)
    import :: c_int
    integer(c_int), pointer :: ip
  end subroutine
end interface

integer(c_int), target :: it = 1
integer(c_int), pointer :: it_ptr

it_ptr => it
write(*,*) it_ptr
call change_target(it_ptr)
write(*,*) it_ptr
```

3 will then print

```
1
2
```