# Section 16: Interoperability with C

Fortran provides a means of referencing procedures that are defined by means of the C programming language or procedures that can be described by C prototypes as defined in 6.5.5.3 of the C standard, even if they are not actually defined by means of C. Conversely, there is a means of specifying that a procedure defined by a Fortran subprogram can be referenced from a function defined by means of C. In addition, there is a means of declaring global variables that are linked with C variables that have external linkage as defined in 6.1.2.2 of the C standard.

The ISO_C_BINDING module provides access to named constants that represent kind type parameters of data representations compatable with C types. Fortran also provides facilities for defining derived types (4.5), enumerations (4.7), and type aliases (4.6) that correspond to C types.

**J3 internal note**

Unresolved issue 94.

I can't find anything about C global variables or name mangling. Presumably to come.

## 16.1 The ISO_C_BINDING intrinsic module

The processor shall provide the intrinsic module ISO_C_BINDING. This module shall make accessible the following entities: C_INT, C_SHORT, C_LONG, C_LONG_LONG, C_SIGNED_CHAR, C_FLOAT, C_DOUBLE, C_LONG_DOUBLE, C_COMPLEX, C_DOUBLE_COMPLEX, C_LONG_DOUBLE_COMPLEX, C_CHAR, C_PTR, C_NULL_CHAR, and C_LOC. The ISO_C_BINDING module shall not make accessible any other entity.

The entities C_INT, C_SHORT, C_LONG, C_LONG_LONG, C_SIGNED_CHAR, C_FLOAT, C_DOUBLE, C_LONG_DOUBLE, C_COMPLEX, C_DOUBLE_COMPLEX, C_LONG_DOUBLE_COMPLEX, and C_CHAR shall be named constants of type default integer.

The values of C_INT, C_SHORT, C_LONG, C_LONG_LONG, and C_SIGNED_CHAR shall each be a valid value for an integer kind type parameter on the processor or shall be -1.

The values of C_FLOAT, C_DOUBLE, and C_LONG_DOUBLE shall each be a valid value for a real kind type parameter on the processor or shall be -1. The values of C_COMPLEX, C_DOUBLE_COMPLEX, and C_LONG_DOUBLE_COMPLEX shall be the same as those of C_FLOAT, C_DOUBLE, and C_LONG_DOUBLE, respectively.

The value of C_CHAR shall be a valid value for a character kind type parameter on the processor or shall be -1.

The entity C_NULL_CHAR shall be a named constant of type character with a length parameter of one and a kind parameter equal to the value of C_CHAR, unless the value of C_CHAR is -1, in which case the kind parameter of C_NULL_CHAR is the default character kind. If C_CHAR has a non-negative value, the value of C_NULL_CHAR shall be the same as the null value of the C character type; otherwise, the value of C_NULL_CHAR shall be the first character in the collating sequence for characters of default kind.

The entity C_PTR shall be a derived type or a type alias name. Its use is described in 16.2.2.

**J3 internal note**

Unresolved issue 97.

Don't we need to say a little more about C_PTR somewhere? Like that all of its components are private? And why even bring up the question of type alias name? Since this is defined in an intrinsic module, we don't need to worry about what the code inside of the module looks like - just what its visible aspect is. Hopefully, we aren't implying that the user could notice the difference between a derived type and a type alias name here.

And if it is a type alias name, then it might (quite plausibly) be a type alias for something like integer. Thus on some machines, it would be legal to assign an integer value to a variable of type C_PTR, whereas on other machines it would not be legal. This seems like it invites gratuitous incompatabilities. I'd say just require this to be a derived type with private components. If you want to allow a type alias, insist that it be a type alias for a derived type with private components. If the processor wants to internally implement it more like a type alias for integer (for example, if argument passing is different for integers and derived types), that's fine. The user, howver, should not be able to take advantage of this. We want the effect on the user to be just like a derived type with private components - he can't touch its innards directly.

Which reminds me of a comment I forgot to make in the type alias section. I would not say that a type alias very effectively "hides" what it is an alias for. If its purpose was "hiding" then it does a poor job, because what is legal to do with a type alias depends directly on what it is an alias for. For example, if it is an alias for a numeric type, then I can assign it a numeric value; otherwise I can't. I think "hiding" is a completely misleading term in this context.

The C_LOC function is described in 16.2.3.

## 16.2 Interoperation between Fortran and C entities

A Fortran entities is said to **interoperate** with a C entity if some correspondence exists between the entities. That correspondence is defined by this section.

A Fortran entity with the POINTER attribute or the ALLOCATABLE attribute cannot interoperate with any C entity. The following sections describe situations in which a Fortran entity can interoperate with a C entity. If an entity is not expressly defined as interoperating with a C entity, it cannot interoperate with any C entity.

**J3 internal note**

Unresolved issue 165

The definition of "interoperate" in 16.2 doesn't tell me anything. It is pretty much self referential. It says that something interoperates if it is described as interoperating in this section, but it still doesn't say a thing about what this *MEANS*. In a definition of a term like "interoperate", I don't care about all the rules for what interoperates with what; that comes later. I want to know what it means to say that 2 things interoperate. There is probably a reason why we didn't say that a Fortran namelist interoperates with a C struct (and its probably a pretty good reason); you'd never guess why we didn't from the definition here. "Some correspondence exists" doesn't hack it, particularly when it just then says that the correspondence is defined by this section. I might paraphrase the definition given here as "a Fortran entity interoperates with a C entity if we say it does.". We could simplify a lot of the definitions in section 2 if we replaced them all by statements like "a Fortran entity is an XXX if it is defined to be one by this standard".

And what kinds of entities does the term apply to. I'm not clear on this point. I think you are mostly talking data entities and procedures, but I find it a bit confusing. At times it almost seems like we are talking about types as being interoperable; at other times it is more like data entities of those types. I think we mostly mean data entities; but at times it seems confused. For example, 16.2.1 starts off talking about the correspondence between Fortran and C types. Its not until the 3rd sentence that it mentions entities of such types. And then in the 4th sentence we specifically talk about types as beinginteroperable. This jumping back and forth between types and entities of the types is confusing.

Another matter of terminology that is inconsistent is whether we say that something "can" interoperate or "does" interoperate. We often talk about whether a Fortran entity can interoperate with a C entity. From this phrasing, I would expect that its interoperability is just established as a possibility. For example, one might think that a Fortran entity can interoperate with a C entity as long as the Fortran entity is of a plausible type, but it only *DOES* interoperate with a C entity if it is actually linked (or associated or whatever we call it) with some C entity. I'm not sure whether that was the idea or not. It mostly just looks to me like we randomly switch between saying that something can or does interoperate without meaning anything different. If we mean one thing, we should use one term. If we mean two things, then we need to be much more upfront about defining what they mean. We don't seem to have a definition of even one meaning yet, much less two of them.

### 16.2.1 Fortran scalar intrinsic entities and C entities

Table 16.1 shows the correspondence between Fortran and C types. The second column of the table refers to the names made accessible by the ISO_C_BINDING intrinsic module. A scalar Fortran entity of the indicated type and kind type parameter, that is not expressly prohibited from interoperating with any C entity, is interoperable with a scalar C entity of any type compatible with the indicated C type in the same row of the table. If the value of any of the kind type parameters in this table is -1, then there is no Fortran type and type parameter that is interoperable with the C types specified in that row of the table. In the case of character type, this table applies only to Fortran entities with a length parameter of 1.

**J3 internal note**

Unresolved issue 166

I've inserted the phrase ", that is not expressly prohibited from interoperating with any C entity," exactly as specified by 99-118r1. I'll let someone else fix the fact that its placement in the sentence makes it completely confusing (or am I wrong in assuming that you probably wanted it to modify "entity" instead of "type parameter"?). I'll get on with typing instead of rewriting.

**NOTE 16.1**

The C programming language defines null-terminated strings, which are actually arrays of the C type char that have a C null character in them to indicate the last valid element. A C string can interoperate with a Fortran array of type character with a kind type parameter equal to C_CHAR.

Fortran's rules of sequence association (12.4.1.5) permit a character scalar actual argument to be associated with a dummy argument array. This makes it possible to argument associate a Fortran character string with a C string.

**J3 internal note**

Unresolved issue 120

Note 16.1 answers the request to say at least something about C strings. It leaves me with one question, however. (Particularly when I was trying to think of cases to counter its proposed wording that it was now "a simple matter" to argument associate a Fortran character string with a C string.

I didn't notice a restriction against assumed-length character dummy arguments for procedure interoperability. Perhaps I just missed it (which is quite possible - I'm skimming). Without such a restriction, we could declare such a Fortran procedure with BIND(C) and I wouldn't know how to actually call it portably from C.

**Table 16.1 Correspondence between Fortran and C types**

| Fortran type | Kind type parameter | C type |
|---|---|---|
| INTEGER | C_INT | int<br>signed int |
| | C_SHORT | short int<br>signed short int |
| | C_LONG | long int<br>signed long int |
| | C_LONG_LONG | long long int<br>signed long long int |
| | C_SIGNED_CHAR | signed char<br>unsigned char |
| REAL | C_FLOAT | float |
| | C_DOUBLE | double |
| | C_LONG_DOUBLE | long double |
| COMPLEX | C_COMPLEX | _Complex |
| | C_DOUBLE_COMPLEX | double _Complex |
| | C_LONG_DOUBLE_COMPLEX | long double _Complex |
| CHARACTER | C_CHAR | char |

**NOTE 16.2**

For example, a scalar object of type integer with a kind type parameter of C_SHORT interoperates with a scalar object of the C type short or of any C type derived (via typedef) from short.

**NOTE 16.3**

The C standard specifies that the representations for positive signed integers are the same as the corresponding values of unsigned integers. Because Fortran does not provide direct support for unsigned kinds of integers, the ISO_C_BINDING module does not make accessible named constants C_UNSIGNED_INT, C_UNSIGNED_SHORT, C_UNSIGNED_LONG, C_UNSIGNED_LONG_LONG, or C_UNSIGNED_CHAR. Instead a user can use the signed kinds of integers to interoperate with the unsigned types as well. This has the potentially surprising side effect that a scalar of the C type unsigned char interoperates with scalars of type integer with a kind type parameter of C_SIGNED_CHAR.

### 16.2.2 Interoperation with C pointer types

A scalar Fortran entity of type C_PTR that is not expressly prohibited from interoperating with any C entity (16.2) is interoperable with a scalar C entity of any C pointer type.

**J3 internal note**

Unresolved issue 167

The bit in 16.2.2 about "that is not expressly prohibitted" seems a bit vague and potentially circular. What do you really mean here? Do you just mean that it can't be a pointer or allocatable? If so, much simpler to say that. I see circularity in that 16.2 says that anything that is not allowed is prohibited. I also note that we usually use the term "explicitly" instead of "expressly". Nothing really wrong with the word "expressly", except that its not the one we've mostly used. (I see only one exception - in section 2 on free form source).

Same problem in the first sentence of 16.2.3 on structs, which has the additional problem that the phrase appears to modify "type" instead of entity.

**NOTE 16.4**

This implies that a C processor is required to have the same representation method for all C pointer types if the C processor is to be the target of interoperability of a Fortran processor. The C standard does not impose this requirement.

**NOTE 16.5**

No facility for dereferencing of C pointers within Fortran is provided.

### 16.2.3 The C address operator

Many C interfaces are defined in terms of ''addresses''. The C_LOC function is provided so that Fortran applications can determine the appropriate value to use with C facilities.

#### C_LOC (X)

**Description.** Returns the ''C address'' of the argument.

**Class.** Inquiry function.

**Argument.** X shall be either a variable that has the TARGET attribute and interoperates with some C object or a procedure that has the BIND(C) attribute. It shall not be an array pointer, an assumed shape array, or an array section.

**Result Characteristics.** Scalar of type C_PTR.

**Result Value.** The value that the target C processor returns as the result of a unary "**&**" operator, as defined in the C standard, 6.5.3.2.

**NOTE 16.6**

The following example illustrates the use of C_LOC.

```
USE ISO_C_BINDING
REAL, TARGET, DIMENSION(10) :: A
TYPE(C_PTR) :: C
C = C_LOC(A)
CALL FOO(C) ! FOO is a C routine.
```

**J3 internal note**

Unresolved issue 170

If I read correctly, we require explicit interfaces for all C procedures. It would therefore seem appropriate to have the sample code for C_LOC in 16.2.3 follow the requirements of the standard rather than trying to substitute a comment.

### 16.2.4 Interoperation with C struct types

A scalar Fortran entity of derived type that is not expressly prohibited from interoperating with any C entity is interoperable with a scalar C entity of a struct type if the derived type definition of the Fortran type specifies BIND(C) (4.5.1), the Fortran derived type and the C struct type have the same number of components, and the components of the Fortran derived type interoperate with the corresponding components of the struct type. A component of a Fortran derived type and a component of a C struct type correspond if they are declared in the same relative position in their respective type definitions.

**NOTE 16.7**

The names of the corresponding components of the derived type and the C struct type need not be the same.

There is no Fortran entity that is interoperable with a C entity of a struct type that contains a bit field or that contains a flexible array member. There is no Fortran entity that is interoperable with a C entity of a union type. There is no C entity that interoperates with a Fortran entity of parameterized derived type.

**J3 internal note**

Unresolved issue 100

Need a reference/glossary entry for flexible array member.

**NOTE 16.8**

For example, a scalar C object of type myctype, declared below, interoperates with a scalar Fortran object of type myftype, declared below.

```
typedef struct {
  int m, n;
  float r;
} myctype

TYPE MYFTYPE
  BIND(C)
  INTEGER(C_INT) :: I, J
  REAL(C_FLOAT) :: S
END TYPE MYFTYPE
```

The names of the types and the names of the components are not significant for the purposes of determining whether the types are interoperable.

**J3 internal note**

Unresolved issue 101

Note 16.5 looks substantially better than it used to. However, I think it a bit odd that it now seems to imply that BIND(C) in a derived type definition automatically makes the ISO_C_BINDING module accessible.

**NOTE 16.9**

The C standard requires the names and component names of two struct types to be the same in order for the types to be considered to be the same. This is simillar to Fortran's rule describing when sequence derived type are considered to be the same type. This rule is not followed for the purposes of determining whether a Fortran entity of derived type interoperates with a C entity of a struct type.

**J3 internal note**

Unresolved issue 102

Note 16.6 refers to the conditions of the C standard for two types to be the same. My quick glance at the index of the C draft makes me wonder whether this is accurate. The C draft section 6.1.2.6 lists conditions sounding quite simillar to these as being necessary for 2 types to be compatible. That section also uses the term "same type" in a much more strict sense. Do you mean "the same" or do you mean "compatible"? We appear to use the term "same type" for this in Fortran (I had to check, though), but I'm not sure that is the C terminology. Also, if we are going to have a note about this anyway, might it not be appropriate to give a hint as to why the difference instead of just mentioning the difference and letting the user guess why?

### 16.2.5 Interoperation with C array types

An explict-shape or assumed-size array of rank *r*, with a shape of $[e_1 \ldots e_r]$ interoperates with a C array if either

(1) the array is assumed size, and the C array does not specify a size or specifies a size of *, or

(2) the array is an explicit shape array, and the extent of the last dimension ($e_r$) is the same as the size of the C array,

and either

(1) *r* is equal to one, and an element of the array interoperates with an element of the C array, or

(2) *r* is greater than one, and an explicit-shape array with shape of $[e_1 \ldots e_{r-1}]$, with the same type and type parameters as the original array, would interoperate with a C array of a type equal to the element type of the original C array.

**NOTE 16.10**

An element of a multi-dimensional C array is an array type, so a Fortran array of rank one cannot interoperate with a multidimensional C array.

**NOTE 16.11**

For example, a Fortran array declared as

```
INTEGER :: A(18, 3:7, *)
```

interoperates with a C array declared as

```
int b[][5][18]
```

### 16.2.6 Interoperation with C functions

A formal parameter of a C function corresponds to a dummy argument of a Fortran procedure if they are in the same relative positions in the C parameter list and the dummy argument list, respectively.

The **reference type** of a C pointer type is the C type of the object that the C pointer type points to.

**NOTE 16.12**

For example, the reference type of the pointer type `int *` is `int`.

A Fortran procedure interoperates with a C function if

(1) the procedure is declared with the BIND(C) attribute;

(2) the Fortran procedure is a function, the result variable of the procedure and the C function interoperate, or if the Fortran procedure is a subroutine, the result type of the C function is compatible with the C type void;

(3) the number of dummy arguments of the Fortran procedure is equal to the number of formal parameters of the C function;

(4) all the dummy arguments are dummy data objects, none of which have either the POINTER or ALLOCATABLE attribute;

(5) any dummy argument with the VALUE attribute interoperates with the corresponding formal parameter of the C function; and

(6) any dummy argument without the VALUE attribute corresponds to a formal parameter of the C function that is of a pointer type, and the type of the dummy argument interoperates with the reference type of the formal parameter.

**NOTE 16.13**

The VALUE attribute may not be applied to arrays (5.1.2.14). It would have been unhelpful to have permitted it, because arrays in C are always passed by reference.

The BIND(C) attribute shall not be specified for a procedure that has an asterisk dummy argument. The BIND(C) attribute shall not be specified for a procedure that requires an explicit interface, unless the procedure is defined by a Fortran subprogram.

**NOTE 16.14**

The requirement that the Fortran procedure not require an explicit interface prohibits its dummy arguments from having the POINTER attribute, having the ALLOCATABLE attribute, or being assumed shape arrays. It also prohibits the Fortran procedure from being elemental or having array results.

**J3 internal note**

Unresolved issue 110

Presumably there will be a syntax somewhere that allows one to specify the BIND(C) attribute for a procedure. I can't find it. And whenever it is written, I assume that the restrictions from 16.2.4 will go to the appropriate place. They seem out of place here.

It should be a separate issue, but I long ago ran out of toes to count on, so I'll lump it in here. What is the "unless" clause supposed to be about? It is quite surprising. So I can do all of the funny stuff as long as it is in a Fortran procedure called from C? I doubt it. And if I have only an interface body in the calling routine, how am I even going to know whether or not the procedure is defined by a Fortran subprogram? I don't understand this at all. Maybe I need more caffeine...or maybe I've had too much.

This is an attempt to prevent things that have dummy arguments with the pointer or allocatable attribute, array results, etc.. However, if the BIND(C) attribute is specified on a subprogram, any procedure defined by that subprogram must have an explicit interface. So we want to prohibit the former, but not the later. We'll work on it in a subsequent paper.

**NOTE 16.15**

For example, a Fortran function with an interface described by

```
BIND(C) INTEGER(C_SHORT) FUNCTION FUNC(I, J, K, L, M)
  USE ISO_C_BINDING
  INTEGER(C_INT), VALUE :: I
  REAL(C_DOUBLE) :: J
  INTEGER(C_INT) :: K, L(10)
  TYPE(C_PTR), VALUE :: M
END FUNCTION FUNC
```

interoperates with a C function with an interface described by

```
short func(int i; double *j; int *k; int l[10]; void *m)
```

**NOTE 16.16**

A C pointer may correspond to a Fortran dummy argument of type C_PTR or to a Fortran scalar that does not have the VALUE attribute.

**J3 internal note**

Unresolved issue 112

I'd think the last note in 16.2.4 a lot more clear with an example. Its a bit short and cryptic. Indeed so short that it doesn't mention that the types need to agree in the nonvalue case. For that matter, it doesn't specify that the Fortran C_PTR case has to specify VALUE. It does have to, doesn't it?