Standard intrinsic module ISO_C_BINDING

March 8, 2024

1 Name

iso_c_binding — standard intrinsic module

2 Usage

```
USE,INTRINSIC ::  ISO_C_BINDING
```

(The ',INTRINSIC ::' part is optional.)

3 Synopsis

ISO_C_BINDING provides named constants, types and procedures that are useful in a mixed-language (C and Fortran) program.

4 Parameter Descriptions

4.1 Kind Parameter Values

The following parameters are all KIND values for matching C types to a Fortran type and kind. If a particular C type has no matching Fortran kind, the parameter value will be negative.

```
INTEGER,PARAMETER :: c_int = ...
INTEGER,PARAMETER :: c_short = ...
INTEGER,PARAMETER :: c_long = ...
INTEGER,PARAMETER :: c_long_long = ...
INTEGER,PARAMETER :: c_signed_char = ...
INTEGER,PARAMETER :: c_size_t = ...
INTEGER,PARAMETER :: c_intmax_t = ...
INTEGER,PARAMETER :: c_intptr_t = ...
```

Kind parameter values for the C integral types `int`, `short`, `long`, `long long`, `signed char`, `c_size_t`, `intmax_t` and `c_intptr_t`. These are all kind values for type `INTEGER`.

```
INTEGER,PARAMETER :: c_int8_t = ...
INTEGER,PARAMETER :: c_int16_t = ...
INTEGER,PARAMETER :: c_int32_t = ...
INTEGER,PARAMETER :: c_int64_t = ...
INTEGER,PARAMETER :: c_int_least8_t = ...
INTEGER,PARAMETER :: c_int_least16_t = ...
INTEGER,PARAMETER :: c_int_least32_t = ...
INTEGER,PARAMETER :: c_int_least64_t = ...
INTEGER,PARAMETER :: c_int_fast8_t = ...
```

```
INTEGER,PARAMETER :: c_int_fast16_t = ...
INTEGER,PARAMETER :: c_int_fast32_t = ...
INTEGER,PARAMETER :: c_int_fast64_t = ...
```

Kind parameter values for the C integral types `int8_t` to `int_fast64_t`. These are all kind values for type `INTEGER`.

```
INTEGER,PARAMETER :: c_float = ...
INTEGER,PARAMETER :: c_double = ...
INTEGER,PARAMETER :: c_long_double = ...
```

Kind parameter values for the C floating-point types `float`, `double` and `long double`. These are all kind values for type `REAL`.

```
INTEGER,PARAMETER :: c_float_complex = c_float
INTEGER,PARAMETER :: c_double_complex = c_double
INTEGER,PARAMETER :: c_long_double_complex = c_long_double
```

Kind parameter values for the C `_Complex` family of types. These always have exactly the same values as `c_float` et al, and are included only for unnecessary redundancy.

```
INTEGER,PARAMETER :: c_bool = ...
```

Kind parameter value for the C type `_Bool`, for use with the `LOGICAL` type.

```
INTEGER,PARAMETER :: c_char = ..
```

Kind parameter value for the C type `char`, for use with the `CHARACTER` type.

4.2 Character Constants

The following parameters give Fortran values for all of the C "backslash" escape sequences.

```
CHARACTER,PARAMETER :: c_null_char = char(0)         ! C '\0'
CHARACTER,PARAMETER :: c_alert = achar(7)            ! C '\a'
CHARACTER,PARAMETER :: c_backspace = achar(8)        ! C '\b'
CHARACTER,PARAMETER :: c_form_feed = achar(12)       ! C '\f'
CHARACTER,PARAMETER :: c_new_line = achar(10)        ! C '\n'
CHARACTER,PARAMETER :: c_carriage_return = achar(13) ! C '\r'
CHARACTER,PARAMETER :: c_horizontal_tab = achar(9)   ! C '\t'
CHARACTER,PARAMETER :: c_vertical_tab = achar(11)    ! C '\v'
```

4.3 Pointer Constants

```
TYPE(c_ptr),PARAMETER :: c_null_ptr = c_ptr(...)
```

This is a C null pointer, equivalent to `(void *)0` in C.

```
TYPE(c_funptr),PARAMETER :: c_null_funptr = c_funptr(...)
```

This is a C null function pointer.

5 Type Definitions

```
TYPE c_funptr
  PRIVATE
  ...
END TYPE
```

This type represents a C function pointer, and is used when passing procedure arguments to a C routine. The interface to the C routine is declared with a `TYPE(c_funptr)` dummy argument, and values of this type can be created by using the function `c_funloc` on a procedure name (see below for restrictions).

```
TYPE c_ptr
  PRIVATE
  ...
END TYPE
```

This type represents a '`(void *)`' C data pointer, and is used when passing pointer arguments to a C routine. The interface to the C routine is declared with a `TYPE(c_ptr)` dummy argument; values of this type are created using the `c_loc` function (Fortran) pointer or target (see below for restrictions). A C pointer can be turned into a Fortran pointer using the `c_f_pointer` function (see below for the full description).

6 Procedure Descriptions

All the procedures provided are generic and not specific. The `c_associated` and `c_sizeof` functions are pure.

In the descriptions below, `TYPE(*)` means any type (including intrinsic types), and `INTEGER(*)` means any kind of INTEGER type.

```
INTERFACE c_associated
  PURE LOGICAL FUNCTION c_associated...(c_ptr_1,c_ptr_2) ! Specific name not visible
    TYPE(c_ptr),INTENT(IN) :: c_ptr_1,c_ptr_2
    OPTIONAL c_ptr_2
  END
  PURE LOGICAL FUNCTION c_associated...(c_ptr_1,c_ptr_2) ! Specific name not visible
    TYPE(c_funptr),INTENT(IN):: c_ptr_1,c_ptr_2
    OPTIONAL c_ptr_2
  END
END INTERFACE
```

Returns true if and only if `c_ptr_1` is not a null pointer and, if `c_ptr_2` is present, the same as `c_ptr_2`.

```
INTERFACE c_f_pointer
  SUBROUTINE c_f_pointer...(cptr,fptr) ! Specific name not visible
    TYPE(c_ptr),INTENT(IN) :: cptr
    TYPE(*),INTENT(OUT),POINTER :: fptr
  END
  SUBROUTINE c_f_pointer...(cptr,fptr,shape) ! Specific name not visible
    TYPE(c_ptr),INTENT(IN) :: cptr
```

```
    TYPE(*),INTENT(OUT),POINTER :: fptr(...)
    INTEGER(*),INTENT(IN) :: shape(:)
  END
END INTERFACE
```

Converts a C address to a Fortran pointer. If `fptr` is an array, `shape` must be an array whose size is equal to the rank of `fptr`.

```
INTERFACE c_f_procpointer
  ...
END INTERFACE
```

It converts `TYPE(c_funptr)` into Fortran procedure pointers.

```
INTERFACE c_funloc
  TYPE(c_funptr) FUNCTION c_funloc...(x) ! Specific name not visible
    EXTERNAL x
  END
END INTERFACE
```

Returns the C address of a Fortran procedure, which must be a dummy procedure, external procedure or module procedure, and must have the BIND(C) attribute.

```
INTERFACE c_loc
  TYPE(c_ptr) FUNCTION c_loc...(x) ! Specific name not visible
    TYPE(*),TARGET :: x
  END
END INTERFACE
```

Returns the C address of a Fortran variable, which must have the TARGET attribute and must not be polymorphic (i.e. it must not declared with the `CLASS` keyword). If `x` is a pointer, it must be associated with a target; if `x` is allocatable, it must be allocated with non-zero size. If `x` is an array, it must have interoperable type and type parameters.

```
INTERFACE c_sizeof
  PURE INTEGER(c_size_t) FUNCTION c_sizeof...(x) ! Specific name not visible
    TYPE(*) :: x(..)
  END FUNCTION
END INTERFACE
```

The actual argument `x` must be interoperable. The result is the same as the C `sizeof` operator applied to the conceptually corresponding C entity; that is, the size of `x` in bytes.

For scalars, this will be equal to `STORAGE_SIZE(x)/STORAGE_SIZE(C_char_'A')`; for an array, the scalar value is multiplied by `SIZE(x)`.

7 See Also

nag_modules(3).

8 Bugs

Please report any bugs found to 'support@nag.co.uk' or 'support@nag.com', along with any suggestions for improvements.

9 Author

Malcolm Cohen, Nihon Numerical Algorithms Group KK, Tokyo, Japan.