Fortran 2003

Bill Long, *Cray Inc*

ABSTRACT: *Fortran 2003 is the popular name for the latest revision of the Fortran programming language. This revision contains several new features to address shortcomings of the previous standard, as well as major new additions to the language in the areas of interoperability with C, object oriented programming, I/O, and support for the IEEE floating point arithmetic standard. Selected new features are reviewed along with the current implementation plan for a standard conforming compiler for the Cray X1. Proposed features for the following standard are also discussed.*

1. Introduction

The original programming language named Fortran was designed almost 50 years ago to be the language of choice for scientific programming. It continues to evolve through a series of revisions that incorporate more modern programming paradigms while retaining a focus on scientific computing and computational efficiency. Major milestones were the 1966 (f66), 1978 (f77) and 1991 (f90) standards, with a minor revision in 1997 (f95). The latest revision is complete and represents a major enhancement over f95. There was some discussion about whether to change the name, as was done for the transition from C to C++ or UPC, but tradition prevailed and the new language is informally known as Fortran 2003, or f03.

The completion of a new standard for Fortran involves the collaboration of two standards organizations. The ISO committee, named WG5, creates a list of features to be included in a new standard based on input from member countries and comments from the wider community. Once the specification of requirements is completed, it is transferred to the technical committee, named J3, which is charged with writing the document that defines Fortran. During development of a new standard, the document goes through a series of drafts.

A copy of the final Working Draft document for f03, document number 04-007, is available at the J3 committee web site: j3-fortran.org. You can download a copy in PDF format by clicking on the "Fortran 2003" label in the left column. This document was completed on May 10, 2004, and will be submitted to ISO for a final yes or no country vote this summer. The official ISO registration of the standard is expected in September of 2004 at which time it will replace the old f95 standard as the definition of Fortran. While the final ISO voting process allows for correction of very minor typographical errors, it does not allow for any technical changes. The May 10 version of the Working Draft of the J3 Fortran committee contains all the technical description of f03. Any non-trivial changes made before the next Fortran standard will take place though the same interpretations process that has been used with previous standards.

Fortran 2003 retains backward compatibility with f95, with very few exceptions, while including a significant number of changes. These changes were described in the paper "Fortran 2000" in the CUG 2003 proceedings. Apart from noting changes that were made to the f03 proposal since then, the complete list will not be replicated here.

The remainder of this paper is organized into three broad sections discussing a selected subset of the new features in f03, the plan for Cray's implementation, and a preview of the proposals already being discussed for the next revision of Fortran envisioned for 2008 or 2009.

2. Selected f03 features

Fortran 2003 contains many new features and enhancements. A recent, informal survey within DOE asked users to rank the importance of these features grouped into nine categories. The results suggested that C interoperability was far and away the most important new feature. Next in priority were procedure pointers, object oriented programming support, allocatable components and dummy arguments, and new I/O enhancements. These features are discussed in the following subsections.

C Interoperability

iso_c_binding module

The iso_c_binding intrinsic module contains definitions for constants and types that provide a way to portably link with programs written using the system's C compiler. KIND values are defined that tie Fortran intrinsic data types to corresponding C data types. For example, C_INT is defined to be the kind value for which an integer(c_int) declaration specifies a data object that has the same size as an int object in C. Constants are defined for all the C data types that have analogs in Fortran. If the Fortran processor does not support a particular combination of type and kind, the corresponding constant in the iso_c_binding module is –1. The module also defines certain standard character constants widely used in C programs, such as C_null_char, and C_new_line. Finally, the module defines new types. C_PTR and C_FUNPRT. These are used to specify variables that can be used as actual arguments or structure components corresponding to C data and function pointers.

C global objects

Names of objects in the data part of a module can be linked to C global data using the bind(c) attribute. This allows Fortran and C routines to have access to shared data using standard syntax. The name of the corresponding global C object defaults to the Fortran name in lower case letters. Optionally, the user can specify a different name with a character constant. Data of any interoperable Fortran intrinsic type may be shared. In addition, a derived type may be specified to interoperate with C with certain restrictions. Interoperable derived types must not have the SEQUENCE attribute, allocatable or Fortran pointer components, or derived type components that are not interoperable. Derived types can be specified to replicate the form of a C structure. A mechanism is also provided to share data in Fortran common blocks with C. Examples illustrating the new syntax:

```
!  First example  ----------

module global_data
use,intrinsic :: iso_c_binding
   type,bind(c) :: flag_type
     integer(c_long) :: ioerror_num
     integer(c_long) :: fperror_num
   end type flag_type

   type(flag_type),bind(c):: error_flags

end module global_data


! The name of error_flags is specified
!  in C as

typedef  struct{
                 long ioerror_num;
                 long fperror_num;
               } flag_type

flag_type  error_flags;

!  Second example --------

module global_data2
use,intrinsic :: iso_c_binding

integer(c_int),bind(c,name='Fc')::fc

common /block/ r,s
common /tblock/ t
real(c_float) :: r,s,t
bind(c) :: /block/, /tblock/

end module global_data2

! The corresponding C declarations are:

int Fc;
struct {float r,s;} block;
float tblock;
```

The first example illustrates specification of an interoperable derived type and a data object of that type. The value of c_long is obtained from the iso_c_binding module.

The second example shows how to connect common block variables to C global variables. The global symbol is the name of the common block. The names of the entries in the common block are local, and may be different in different Fortran program units. As is the case with most attributes, the bind(c) attribute can be used either as a qualifier in a type declaration or as a separate statement. The separate statement form must be used for common blocks.

Interoperating with C functions.

Interoperation with C functions using standard syntax is a major new feature of Fortran 2003. To correctly link with a C function, as caller or callee, the compiler needs to know the correct interface information. This is specified by extensions to the interface block syntax. The bind(c) attribute identifies an external procedure as conforming to the C calling conventions. If there is no name clause in the bind(c) attribute, the C name of the procedure is the Fortran name in lower case letters. If there is an explicit name specified in bind(c,name='…') the leading and trailing blanks are removed from the character value and that name is used as the C name. In the special case that the character value is all blank or zero length, no binding name is specified. This can be used when the procedure might be passed to a C routine by means of a procedure pointer (as a call back routine, for example) and the C name is not actually needed. The external routine could be written in a language other than C, as long as its interface can be expressed in terms of C prototype. The constants from the iso_c_binding module are used in dummy argument declarations. A new attribute, VALUE, is optional for dummy arguments. If a dummy argument with the value attribute is defined within the subprogram, the corresponding actual argument is not changed. The value attribute effectively causes the argument to be passed by copy-in value. The dummy arguments in an interface for a bind(c) procedure must be interoperable with C data types. It is always possible to write a corresponding C prototype to describe the function interface. There is a syntax change since the CUG 2003 paper worth noting. The comma the preceded the bind(c) clause is no longer part of the syntax for subroutine and function statements. Cray's implementation will support the new syntax in the version 5.3 compiler and will continue to support the previous syntax, with the comma, as an extension. Example:

```
use,intrinsic :: iso_c_binding
interface
   function foo(ptr,val)       &
            bind(c,name='Foo') &
            result(bar)
      import :: c_int, c_long
      integer(c_int) :: ptr, bar
      integer(c_long),value :: val
   end function foo
end interface
integer(c_int) :: x,n
integer(c_long) :: y
   …
n = foo(x,y)

Corresponding C interface:

int Foo( int *ptr, long val);
```

C interoperability intrinsics

Five new intrinsic functions are provided as part of the iso_c_binding module. These are used to create and test C style pointers that are sometimes needed as actual arguments to C functions or as values of components of derived type objects interoperating with C structs.

C_LOC(fortran_data_arg) returns a type(C_PTR) pointer to the data argument.

C_ASSOCIATED(cp1 [,cp2]) returns true if the C pointer cp1 is associated, or if the two arguments are associated with the same target. This is analogous to the associated intrinsic function for Fortran pointers.

C_F_POINTER is a subroutine that associates the target of a C data pointer with a Fortran pointer.

C_FUNLOC(fortran_proc_arg) returns a type(C_FUNPTR) pointer to the Fortran procedure argument.

C_F_PROCPOINTER is a subroutine that associates the target of a C function pointer with a Fortran procedure pointer.

Enumerators

Enumerators are provided as a way to declare a set of integer constants that have the same kind as constants specified in a corresponding C enumeration type. Example:

```
enum,bind(c)
   enumerator :: red = 4, blue, yellow
end enum
```

This example defines three parameters, red=4, blue=5, and yellow=6. The kinds of these constants correspond to the interoperable kinds that would have resulted from an analogous enum statement in C.

Procedure Declarations and Pointers

PROCEDURE statement

The PROCEDURE statement is an extension of the module procedure statement from f90, used to define a generic interface. The specific procedures named in a procedure statement do not have to be contained in the module, as is the case with the module procedure statement. Interfaces for the procedures need to be visible. Example:

```
interface sgemm
   procedure sgemm_44, sgemm_48
   procedure sgemm_84, sgemm_88
   procedure cgemm_44, cgemm_48
   procedure cgemm_84, cgemm_88
end interface

interface dgemm
   procedure sgemm_44, sgemm_48
   procedure sgemm_84, sgemm_88
   procedure cgemm_44, cgemm_48
   procedure cgemm_84, cgemm_88
end interface
```

The example illustrates a mechanism for making the BLAS matrix multiply routine completely generic. The numbers at the ends of the specific routine names indicate the kind values for integer and real (or complex) arguments. Interfaces for the generic names cgemm and zgemm would be written in the same way. Interfaces for all of the specific routines need to be visible.

Procedure declarations and abstract interfaces

The procedure declaration statement can declare names to be of external procedures and identify an interface. An abstract interface specifies the interface information for a hypothetical procedure, and hence the procedure name itself is not made external. Abstract interfaces are used as templates for the interfaces of actual procedures. A procedure statement may reference either an abstract interface or an actual interface. Examples:

```
abstract interface
   function fun_r(x)
      real,intent(in) :: x
      real            :: fun_r
   end function fun_r
end interface

procedure(fun_r) :: gamma, Bessel

interface
   subroutine sub_r(x)
      real :: x
   end subroutine sub_r
end interface

procedure(sub_r) :: sub
procedure(real) :: psi
```

The declarations for gamma and Bessel use the abstract interface fun_r. The declaration for sub uses the explicit interface for sub_r. The declaration for psi uses an implicit interface, and is equivalent to real,external :: psi.

Procedure pointers

The procedure declaration statement may be used to declare procedure pointers. The pointer name may be used in place of the target name in CALL statements, function references, or as an actual argument. Procedure pointers may be components of derived types. Examples, assuming the abstract interface for fun_r above:

```
procedure(fun_r),pointer :: &
                 special_fun => null()

special_fun => gamma
```

The name special_fun is effectively an alias for gamma. In the declaration statement, special_fun was initialized to a status of disassociated.

```
type proc_ptr
   procedure(fun_r),pointer :: fun
end type proc_ptr

type(proc_ptr) :: special(10)

special(3)%fun => Bessel

ans = special(3)%fun(arg)
```

The second example declares a list of 10 procedure pointers, associates the third one with the Bessel function, and shows the syntax for referencing the target function Bessel by using the procedure pointer. This is equivalent to ans = Bessel(arg).

Object Oriented Programming

Extended types

Derived types are often extended from a general parent type to a larger type that contains additional variables for a more specific case. In f90 this was typically done be defining a new type for the specific case and including a component of the parent type. This technique requires a multiple part reference for the components of the base type. If the specific type is extended again, the complexity of references to the parent types increases. Fortran 2003 allows explicit extension of a type such that the parent components are also components of the extended type. The parent components are “inherited” by the extended type. This eliminates the reference part explosion, and is also more in keeping with the style of object oriented programming. Example:

```
type :: dna
   integer,allocatable :: ascii_text(:)
   integer  :: length
end type dna

type(extends(dna)) :: ocdna
   integer :: ssdid
   integer :: ssdsize
   integer :: state
end type ocdna
```

The derived extended type ocdna (out of core version of dna) contains five components, the three specified along with the two inherited from the parent type dna. There is also an implied component named dna that allows multi-part access to the parent types in the f90 style. This can be useful in cases where dummy argument type matching requires an object of the parent type.

Most derived types may be extended, though sequence and bind(c) types are not extendable. A type can inherit components from only one parent, commonly known as single inheritance. However, several extended types may have the same parent.

Type-bound procedures

Procedures can be bound to a type, automatically carrying along interface information with each variable of that type. Type-bound procedures are part of the overall OOP features of Fortran 2003. Procedures are declared with PROCEDURE, GENERIC, or FINAL statements. The type contains only the declaration for the procedure. The actual procedure is defined elsewhere. Only the interface for the procedure must be visible to the type definition. A type-bound procedure may have an implied argument of the containing type, specified with the PASS attribute. Example:

```
type strange_int
    integer :: n
contains
    generic :: operator(+)=> strange_add
end type
```

The interface for strange_add must be either supplied by an interface block, or by defining the function in an accessible module.

Polymorphic objects

The CLASS type specifier is used to declare polymorphic objects. These declarations must be for dummy arguments, or for objects with the allocatable or pointer attribute. The primary use of polymorphic objects is as dummy arguments. Actual arguments of the type specified, or any extension of that type, are type compatible with the corresponding dummy argument. Assuming the subprogram uses only components from the declared type, all extensions of that type will also have those components and hence be a reasonable type for actual arguments. The specification of a polymorphic dummy argument allows the routine to be called with arguments of the declared type or any of the extended types. It is possible to declare something CLASS(*), or unlimited polymorphic. Such an object is type compatible with any type object. Use of an unlimited polymorphic object is limited to allocate statements or statements within a select type construct, where more information about the actual type can be determined. Example:

```
function strange_add (a,b) result (c)
   class(strange_int),intent(in) :: a,b
   type(strange_int)             ::c

   c%n = iand(a%n+b%n, 1)
end function strange_add
```

This function is assumed to be in the same module that defines the type strange_int above.

Select Type construct

The select type construct allows alternate execution paths based on the actual type of a polymorphic object. The selection clauses are TYPE IS, CLASS IS, or CLASS DEFAULT. If the type of the argument specified in the select type statement matches one of the types specified in a TYPE IS clause statement, then the code block following that statement is executed. If none of the TYPE IS types match the type of the selector, then the CLASS IS clauses are tried. The most extended type that matches is selected. If none of the CLASS IS statements has a compatible type, the CLASS DEFAULT block is executed. CLASS IS (*) is not allowed because it is redundant with CLASS DEFUALT. Example, assuming the definition of strange_int from above:

```
type,extends(strange_int)::strange_mint
   integer :: m
end type strange_mint

class(strange_int) :: a,b,c

select type(a)
type is (strange_int)
   c%n = iand(a%n+b%n,1)
class is (strange_int)
   i = min(a%m,b%m)
   c%n = iand(a%n + b%n, 2**i-1)
   c%m = i
end select
```

The select type construct above represents a way to implement the computation in the function strange_add for more than one type in the class strange_int.

Finalizers

A finalizer is a special type of type-bound procedure that is executed when an object of the containing derived type becomes undefined. A variable may become undefined by various means, including the initial state of an intent(out) dummy argument, or the state of a unsaved local variable at procedure exit. Finalizers are specified with the FINAL declaration. Example:

```
type foo
    real,pointer :: bar(:)
contains
   final :: foo_cleanup
end type

subroutine foo_cleanup(x)
   class(foo) :: x
   deallocate(x%bar)
end subroutine foo_cleanup
```

Allocatable components and arguments

Allocatable components

One of the least satisfactory aspects of f95 is the requirement that dynamically sized components of a derived type be declared as a pointer. Because a compiler cannot determine all the possible aliases for pointer target data, optimization of expressions involving such data is restricted. The new standard allows allocatable components, which do not have this performance problem. Example:

```
type :: foo
   real,allocatable :: bar(:)
end type foo
```

Allocatable dummy arguments

The size needed for an actual argument associated with a dummy argument may be computed inside the called procedure. With f95, such an argument had to be a pointer, resulting in the disadvantages of pointers being forced on the programmer. Fortran 2003 allows allocatable dummy arguments, resolving this shortcoming of f95. The storage for an allocatable dummy argument is not automatically deallocated at the end of the procedure. Example:

```
integer,allocatable :: db(:)
call sub(db,nwords)

subroutine sub(db,n)
   integer,allocatable :: db
   integer             :: n

   read *, n
   allocate(db(n))
   read *, db
end subroutine sub
```

Allocatable function results

Function results can be considered equivalent to an additional argument to a subroutine. A natural extension of the allocatable dummy argument feature is the allocatable function result. This is included in Fortran 2003. Example:

```
function foo(x) result (foo_r)
   real,dimension(:),intent(in)  :: x
   real,dimension(:),allocatable ::foo_r
… ! foo_r must be allocated in foo
end function foo
```

I/O Features

Asynchhronous I/O

Fortran 2003 contains syntax support for asynchronous input and output operations. An asynchronous read or write statement initiates the operation but allows the program to continue before the operation is finished. A separate WAIT statement forces the program to wait until the operation is completed. The functionality is essentially the same as the old buffer in and buffer out statements.

```
open(10,…,asynchronous='yes',…)

read(10,…,asynchronous='yes',id=idw) …

wait(10, id=idw)
```

Without the ID clause in the WAIT statement all currently outstanding operations on the unit must complete. Executing a CLOSE or INQUIRE operation on the unit has an implied wait if the file was opened for asynchronous data transfer.

Stream I/O

Part of the improved interoperability with C includes support for stream I/O. Files opened for stream I/O do not have internal record structure information. Formatted files may have embedded newline characters, matching the convention used by C programs to delimit records. Unformatted files do not contain internal record size information. The current location within the file can be obtained or specified with a POS= keyword in the I/O statement. Example:

```
open (unit=10, … access = 'stream', …)
```

IEEE features

Support for IEEE floating point arithmetic is a major new feature in Fortran 2003. This is optional in the sense that the features are not required on systems that do not have hardware support for particular modes or functions. The IEEE_FEATURES intrinsic module contains constants that are defined if the processor supports the indicated feature. The full list of constants is

```
ieee_datatype
ieee_nan
ieee_inf
ieee_denormal
ieee_rounding
ieee_sqrt
ieee_halting
ieee_inexact_flag
ieee_invalid_flag
ieee_underflow_flag
```

Constants omitted from the module correspond to unsupportable features. A USE of the module with an ONLY clause can detect the absence of a feature at compile time.

IEEE arithmetic control

The IEEE_ARITHMETIC intrinsic module defines a type, ieee_class_type, and constants of that type corresponding to the possible values of ieee floating point numbers:

```
ieee_signaling_nan
ieee_quiet_nan
ieee_negative_inf
ieee_negative_normal
ieee_negative_denormal
ieee_negative_zero
ieee_positive_zero
ieee_positive_denromal
ieee_positive_normal
ieee_positive_inf
ieee_other_value
```

The module also defines a type, ieee_round_type, and constants of that type corresponding to the ieee rounding modes:

```
ieee_nearest
ieee_up
ieee_down
ieee_to_zero
ieee_other
```

IEEE arithmetic functions

The IEEE_ARITHMETIC intrinsic module also defines a set of functions to inquire about support for various features, get and set rounding modes, and perform ieee conforming operations. If an ieee_support_* routine returns false, referencing other routines that depend on support for that feature may not be meaningful. The functions defined in the module are:

```
ieee_support_datatype
ieee_support_denromal
ieee_support_divide
ieee_support_inf
ieee_support_io
ieee_support_nan
ieee_support_rounding
ieee_support_sqrt
ieee_support_standard
ieee_support_underflow_control

ieee_class
ieee_copy_sign
ieee_is_finite
ieee_is_nan
ieee_is_normal
ieee_is_negative
ieee_logb
ieee_rem
ieee_rint
ieee_scalb
ieee_unordered
ieee_value

ieee_selected_real_kind

ieee_get_rounding_mode
ieee_set_rounding_mode
ieee_get_underflow_mode
ieee_set_underflow_mode
```

IEEE exception control

The IEEE_EXCEPTIONS intrinsic module defines two new data types: ieee_status_type, and ieee_flag_type. The ieee_status_type should be used to declare a variable that holds the current value of the floating point status. The constants of type ieee_flag_type defined in the module are:

```
ieee_overflow
ieee_divide_by_zero
ieee_invalid
ieee_underflow
ieee_inexact
```

The module also includes several routines to get and set values of exception flags:

```
ieee_support_flag
ieee_support_halting
ieee_get_flag
ieee_set_flag
ieee_get_halting_mode
ieee_set_halting_mode
ieee_get_status
ieee_set_status
```

If the ieee_support_flag or ieee_support_halting routines return false for a particular flag, referencing the corresponding get and set routines is not meaningful.

3. Implementation Status and Plans

Many of the new f03 features are already supported by the Cray Fortran compiler for the X1 series systems. Additional support is planned with each major release of the compiler. The list of features already supported and those planned for the next major compiler release is detailed in the next two subsections.

Fortran 2003 features in ftn 5.2

allocatable components: Components of user defined types may be allocatable arrays or scalars. If a derived type variable with an allocatable component appears as the variable in an intrinsic assignment statement, the components are automatically allocated to match the corresponding components of the right hand side expression.

allocatable dummy arguments: Dummy arguments may be allocatable. If they are allocated in the procedure, they remain allocated on return.

allocatable function results: The returned result from a function may be allocatable. The result must be allocated during execution of the function.

automatic array allocation: If an allocatable whole array (not a section) appears as the variable in an assignment statement and is not allocated, or is allocated but with a shape different from the expression, the array is automatically allocated or reallocated with the correct shape. Because of the checking overhead involved, this feature is active only if the –ew compilation is specified.

mixed component accessibility: Component names in a user defined type may have their accessibility, either PUBLIC or PRIVATE, specified separately for each component.

public entities of private type: Objects declared in a module may have PUBLIC accessibility even if their type is PRIVATE.

keywords in structure constructors: A structure constructor provides a means to specify the values of all the components of a user defined type object. The component names may be used as keywords to specify the values in any order, or to allow writing self-documenting code. This capability parallels the use of dummy argument names as keywords in procedure references.

VOLATILE attribute: A variable with the VOLATILE attribute may have its value changed by some mechanism not visible to the routine where the variable is declared. This is usually associated with global data shared by more than one process. The compiler will reload values of volatile variables from memory before each use, rather than relying on values that may be in a register.

PROTECTED attribute: A module variable with the PROTECTED attribute may be referenced in a program unit using the module, but may not be defined except by initialization in the module or by a procedure contained in the module.

INTENT for pointer arguments: Dummy arguments that are pointers may also have an INTENT specified. The intent refers to the pointer association status of the pointer, and not the definition status of the target of the pointer.

Character MIN and MAX: The MIN and MAX intrinsics are extended to allow character arguments.

Lower bounds in pointer assignment: Explicit specification of the lower bound for a pointer array in pointer assignment is allowed. Previously the lower bound was always 1.

Parameters in complex constants: The real and imaginary parts of a complex constant may be named constants rather than literal constants, such as eye=(zero,one) where one and zero are declared with the parameter attribute.

Enumerators: Enumerators create a sequence of values for a set of names, effectively shorthand for a series of parameter declarations. This is partly aimed at C interoperability.

FLUSH: A FLUSH statement is added that causes internal I/O buffers to be flushed for a specified unit. This is equivalent to the old flush library routine.

Names for I/O units: The iso_fortran_env intrinsic module contains definitions for several run time environment values Included are INPUT_UNIT, OUTPUT_UNIT, and ERROR_UNIT, which are the unit numbers for the input and output units corresponding to *, and the standard error unit.

Carriage Control: The first character in a formatted write output record is no longer interpreted as a carriage control character.

Command line: The parcels of the command line that initiated execution of the program are available through new intrinsic routines COMMAND_ARGUMENT_COUNT, GET_COMMAND, and GET_COMMAND_ARGUMENT.

Environment variables: The values of external environment variables are available through the new intrinsic routine GET_ENVIRONMENT_VARIABLE.

Continuation lines: The old limit of 39 continuation lines is increased to 255. The Cray compiler does not limit the number of continuation lines.

Optional KIND arguments: Several intrinsic functions, such as COUNT, allow optional KIND arguments to specify the kind of the result value. This is very useful if the default size if 32 bits and the values that could be returned are very large.

C Interoperability: All components of the C interoperability feature in Fortran 2003 are supported except in two areas. The current implementation does not support the form of bind(c) on a subroutine or function statement without the preceding comma. The c_funloc() and c_f_procptr() intrinsic procedures are not supported yet because they require support for procedure pointers.

Fortran 2003 features planned for ftn 5.3

Several additional Fortran 2003 features are planned for the next major release of the Cray Fortran compiler, version 5.3, schedules for release in October, 2004. There are summarized below.

Procedure declarations: The new procedure declaration statement specifies interface and attribute information for procedures.

Procedure pointers: Procedure pointers are dynamic aliases for procedure names. They may be components of a structure.

C Interoperability: Add the c_funloc() and c_f_procptr() intrinsic procedures to complete the C interoperability feature. Remove the requirement for the comma before bind(c) in subroutine and function statements. This will complete the implementation of all the C interoperability features in f03.

New array constructor syntax: In addition to the f95 syntax of (/ … /) for array constructors, the cleaner […] syntax is added.

Type specification in array constructors: The ability to specify a type in an array constructor eliminates the need to attach explicit KIND parameters to each element in the array, and allows character array constructors without the need to pad the elements to the same length.

Pointer rank remapping: A rank-1 target array can be associated with a pointer of higher rank by specifying shape information in the pointer assignment statement.

Asynchronous and Stream I/O: New syntax for Open, Read, and Write statements, and the new Wait statement, provide portable support for asynchronous and stream I/O.

International real I/O: The form for a real value in formatted I/O can have a comma instead of the default decimal point.

Access to I/O error messages: The user has the capability to capture the text of an I/O error message in a character variable. This could be used for diagnostic output.

Operator renaming: User defined operator names can be renamed on a USE statement.

Longer names: The maximum length for names is increased from 31 to 63 characters.

Optional KIND arguments: The remaining intrinsic functions that take optional KIND arguments will be implemented.

SYSTEM_CLOCK: The system_clock intrinsic subroutine will be generic, allowing integer arguments that are larger than default. This addresses a long standing problem of using default integers on 32-bit machines for these arguments.

4. The next Fortran standard

Future directions

Over 100 suggestions already been submitted for new features to be added to the next revision of Fortran. Many of these will eventually rejected by WG5, and most of the ones that are accepted will be relatively minor. Four fairly significant features are being actively discussed for the next revision of Fortran. These are Submodules, Co-arrays, Typless data, and Generic Programming.

A proposal to add SUBMODULES to Fortran is already well developed. The basic idea of submodules is to separate the interface from the actual definition of a module procedure. This will allow for the definitions of the procedures to be in submodules that are in separate files. The main benefits of this structure are a better environment for large-scale projects with many developers, and as a mechanism to avoid compilation cascades common with the current module structure. The submodule facility is already on the form of a Technical Report that has been sent to ISO for a final vote. Once passed, vendors may implement this feature well before the next standard is finalized.

Parallel execution dominates large-scale scientific computations, which is the traditional focus of Fortran. The most attractive candidate for parallel structures within Fortran is the Co-Array Fortran (CAF) model. A proposal to include Co-Arrays in the next standard has been submitted by the UK national body. Wider adoption of Co-Arrays by the user community would enhance the chances of it being adopted as a required part of the next standard.

Fortran has traditionally focused on numeric computing and has strong support for numeric data types. Emerging fields like bioinformatics have a significant computational component that involves non-numeric bit manipulation. The addition of typeless data to Fortran has been formally proposed for the next standard. This feature would also simplify some procedure interface issues, better handle hexadecimal, octal, and binary constants, and standardize some of the bit manipulation intrinsics such as popcnt.

Generic Programming, or genericity, has been proposed in several different forms for the next standard. This facility would allow writing reusable, type-independent code similar to what is possible with templates in C++. The ideas at this point are not sufficiently focused to describe a particular approach, but the general idea has strong support.

Acknowledgments

The author would like to thank the Cray Fortran compiler group for early implementation of many of the Fortran 2003 features as well as substantial input on the future implementation schedule.

About the Author

Bill Long represents Cray as a primary member of the J3 Fortran standard committee. He is also the primary author of the Cray Bioinformatics Library, most of which is written in Fortran 2003. Bill can be reached at Cray Inc., 1340 Mendota Heights Road, Mendota Heights, MN 55120, Email: longb@cray.com.