



ALBUQUERQUE  
High Performance Computing Center



# Workshop On Scientific Problem Solving In Fortran 90/95 — Parameterized Types And Floating Point Issues

Spring, 1999

Richard C. Allen, SNL

Paul M. Alsing, UNM/AHPCC

Andrew C. Pineda, UNM/AHPCC

Brian T. Smith, UNM/AHPCC/CS



## Topics To Be Covered

- ▶ Parameterized types
  - ◆ What is it and why have it?
  - ◆ General mechanism in Fortran -- kinds of a type
  - ◆ For non-numeric types
    - logical and character types -- very simple
- ▶ For numeric type (integer, real, complex)
  - ◆ Type declarations
  - ◆ Constants
  - ◆ Models For The Implementation
  - ◆ Inquiries about the model used
  - ◆ Expressions with mixed types and kinds



## Topics To Be Covered Continued

### ▲ Floating Point Issues

- ◆ **Specifying minimum precision for an application**
- ◆ **Porting codes**
- ◆ **Codes that adapt to their executing range and precision**
- ◆ **Manipulation of parts of the floating point representation**
  - **Efficient argument reduction (e.g. square root function)**



## Data Objects — Review

- ▶ Variables or constants
  - ◆ **a variable**
    - may be of any type and kind
    - is an identifier that is declared in a type statement
    - is given a value initially or during execution
    - may change its value during execution
    - may be a scalar or an array



## Examples of Variables

**real X**

**real :: A = -huge(X) ! Given a value initially**

**read \*, A**

**X = abs(A)**

**X = 0.5\*(X + A/X)**

- ◆ **Declared X, declared A with initial value**
  - only real above, but also integer, complex, logical, character
- ◆ **X and A are changed during execution**
- ◆ **All the above are scalars**
  - arrays will be described later but they may also variables



## Data Objects — Constants

- ▶ A constant
    - ◆ may be of any type and kind
    - ◆ may be a scalar or an array
    - ◆ may be a literal constant -- e.g. a digit string
    - ◆ may be a named constant (called a parameter)
      - an identifier that is declared, has the **PARAMETER** attribute, is given a value in a specification statement, and *never* changes
- real, parameter :: FUDGE = 1.998



## Literals and Named Constants

- ▶ Every type has literal constants
    - ◆ **real:** 1.0, 1.04e-10
    - ◆ **integer:** 1, -3
    - ◆ **complex:** (1.0, -1.0)
    - ◆ **character:** 'a', 'z', 'string', "string"
    - ◆ **logical:** .true., .false.
  - ▶ Every type has named constants:
    - ◆ **the programmer defines them**
      - using the **PARAMETER** attribute or statement
- real, parameter :: PI = 3.14159...



## Parameterized Types — What Are They?

- ▲ A means of specifying the various kinds of each of the intrinsic types
  - ◆ integers -- short, medium, and long
  - ◆ reals -- single and double precision, and extended
  - ◆ complex -- single, double, and extended precisions
  - ◆ logicals -- of size 1 bit, 1 byte, 1 word
  - ◆ characters -- 1 byte or multi-byte for large number of graphics (eg. Kanji)





## The Problem — CPUs Are Not Created Equal — The Why?

- ▶ CPUs made by different vendors use different kinds of integers and reals
  - ◆ some have single and double precision reals
  - ◆ some have extended precision reals
  - ◆ Cray uses a different representation than Intel
  - ◆ IBM uses base 16 on some machines and base 2 on others
- ▶ How does a program access the different kinds of reals and integers?
- ▶ How does a programming language adapt to these different machines and changes in them?



## Variations In Representations

- ▶ Different ways of representing integers and reals since the advent of electronic machines
  - ◆ different lengths: 32, 36, 48, 64, 80, and 128 bits for just a few
  - ◆ different radices for the representation: 2, 3, 8, 10, 16
  - ◆ different encodings:
    - integer: sign-magnitude, 1's complement, 2's complement
    - real: different sized exponents and fractions
- ▶ What is common denominator (nearly)?
  - ◆ power/sum form with fixed base approximates nearly all forms
- ▶ Define models for integers and reals based on this common denominator



## The Integer Model

- ▶ Power/sum form with a fixed base for the integers: Any integer  $i$

$$i = s \sum_{k=0}^{q-1} d_k r^k$$

where:

- $s$  is a sign (plus or minus 1) for  $i$
- $d_k$  are digits of  $i$  with  $0 \leq d_k < r$
- $q$  is the number of digits to hold  $i$  -- a CPU property
- $r$  is the radix or base for the representation -- a CPU property

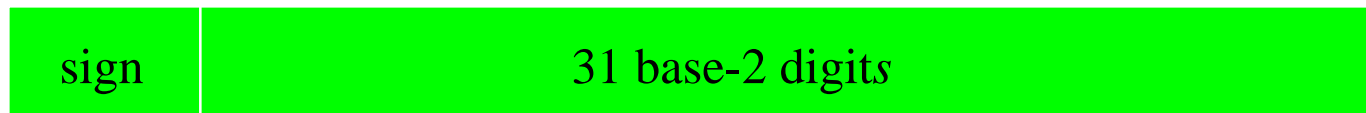


## Fitting The Integer Model To A Machine

- ▶ Given the model, find the parameters  $r$  and  $q$  so that the model “best” fits the machine:
  - ◆ **For example:**
    - 32 bit word for integers
    - integers are represented in base 2 --  $r = 2$
    - 1 bit denotes the sign of the integer number, usually the first bit
    - the remaining 31 bits are for the digits of the integer --  $q = 31$
  - ◆ **This works with 2's complement and sign/magnitude**
    - 1's complement has two zeros and so it is treated as if it had only 30 bits for the digits of the integer --  $q = 30$

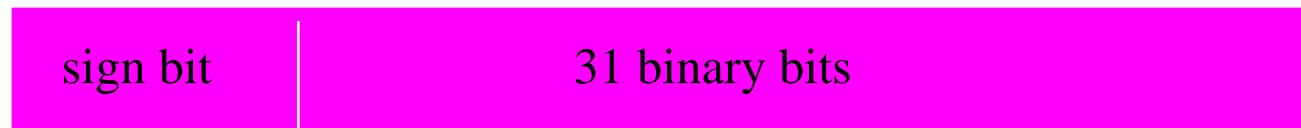
# An Picture Of Machine And Model Integers

Model:  $q = 31, r = 2$

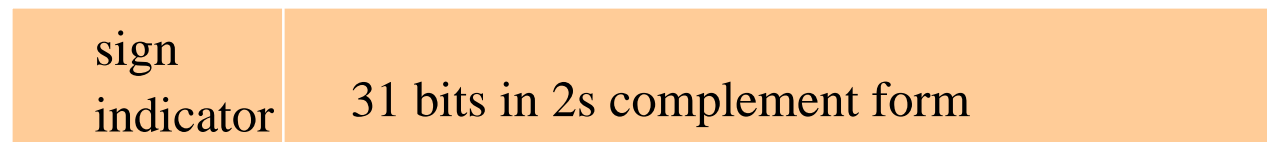


Machine:

- ♦ **sign magnitude**



- ♦ **2s complement**





## Parameterization For The Integers

- ▶ From the model, only  $r$  and  $q$  are machine dependent
- ▶ For a particular CPU, make a list of all of the different kinds of integers
- ▶ Number the different pattern with positive integers in any order.

- ◆ **For example, PC Salford Compiler (Nag From End)**

size in bits:	8	16	32	
base $r$ :	2	2	2	
$q$ :	7	15	31	
kind number:	1	2	3	(Salford)
kind number:	1	2	4	(IBM)



## RealModel

- ▶ Power/sum model with an exponent of fixed range. The nonzero real number  $x$  is:

$$x = sb^e \sum_{k=1}^p f_k b^k$$

where:

- $s$  is the sign (+1 or -1)
- $f_k$  is the  $k$ -th digit in the mantissa with  $0 \leq f_k < b$  with  $f_1 > 0$
- $b$  is the base or radix, and is an integer -- a CPU property
- $p$  is the number of mantissa digits -- a CPU property
- $e$  is an integer with  $e_{min} \leq e \leq e_{max}$
- $e_{min}$  and  $e_{max}$  are specified integers -- CPU properties



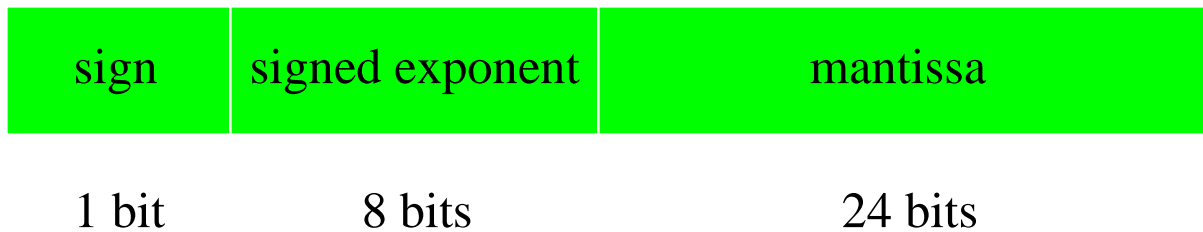
## Fitting The Real Model To A Machine

- ▲ Given the model, find the parameters  $r$ ,  $b$ ,  $e_{min}$  and  $e_{max}$  so that the model “best” fits the machine:
  - ◆ For example: Intel IEEE Floating Point P-754
    - 32 bit word for reals
    - reals are represented in base 2 --  $r = 2$
    - 1 bit denotes the sign of the real number, the first bit
    - 8 bits for the exponent --  $e_{min} = -125$ ,  $e_{max} = 128$
    - 23 bits + 1 implied bit for the mantissa --  $p = 24$

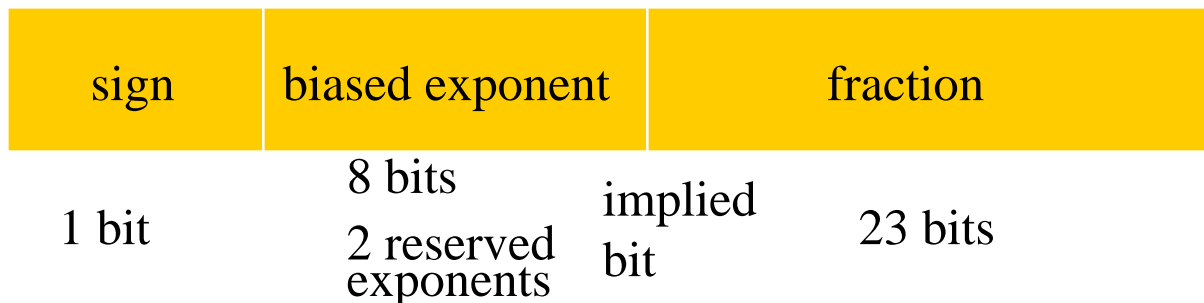


## An Picture Of Machine And Model Reads

- Model:  $q = 31, r = 2, e_{min} = -125, e_{max} = 128$



- Machine:





## What Are Parameterized Types?

- ▶ A way to specify the different kinds of reals or integers that your compiler/machine supports
  - ◆ a CPU may have integers that are 8 bits long, 16 bits long, 32 bits long or even 64 bits long
  - ◆ a CPU may have reals of length 32 bits, 64 bits, 128 bit
- ▶ The parameterization provides static but readily adaptable way to specify and to modify with recompilation a particular specification
  - provides a portable, general, and flexible technique for specification of kinds of these intrinsic types



## Topics To Cover

- ▶ How do we specify kinds in a program?
- ▶ How do we know which ones the compiler/machine supports?
- ▶ How do we write a programs so that the program can adapt to whatever it provided?
- ▶ How do we transfer our programs to different machines -- portability



## Specification of Integer Kinds

- ▲ The kinds of a type (say integer) are designated by positive integers, say 1, 2, 3
  - ◆ In a declaration, the kind number appears in parenthesis after the type name
    - For example, for integer type specification statements:  

```
integer(1) I, J  
integer(2) :: K  
integer(3), dimension(10,10) :: P
```
    - If no kind number is specified, a default kind number is provided by the processor; the Salford compiler selects 3; the IBM compiler selects 4.



## Kind Numbers Are Potentially Non-portable

- ▶ Because the numbering scheme for kinds is processor-dependent and the defaults are processor-dependent, kind number specifications are potentially non-portable
- ▶ There are two mechanisms to provide a portable specification
  - ◆ **use of module (global-like) named constants for the kind numbers**
  - ◆ **use of certain intrinsic functions to specify the kind numbers**

## Using Decimal Ranges

- Consider classifying the kinds of integers by the decimal ranges of their representable values. Consider the Salford compiler:

Kind	Size	q	r	huge	Dec.Range	RANGE
1	8	7	2	$2^7-1$	$[-10^2, 10^2]$	2
2	16	15	2	$2^{15}-1$	$[-10^4, 10^4]$	4
3	32	31	2	$2^{31}-1$	$[-10^{10}, 10^{10}]$	10

- $\text{RANGE} = \lfloor \log_{10}(\text{huge}(x)) \rfloor$



## Certain Intrinsic Functions

- ▶ For integers, there is an intrinsic function  
**selected\_int\_kind(<integer\_range>)**  
where <integer\_range> is an *integer constant* representing the decimal range of the integers whose kind number is to be returned
  - ◆ The <integer\_range> is a minimum specification
- ▶ For example, on the Salford compiler,
  - ◆ **selected\_int\_kind(2)** returns 1
  - ◆ **selected\_int\_kind(4)** returns 2
  - ◆ **selected\_int\_kind(8)** returns 3

## Using Decimal Precisions

- Consider classifying the kinds of reals by the decimal precisions of their representable values. Consider the Salford compiler:

Kind	Size	p	r	epsilon	Dec. Precision
1	32	24	2	$2^{-23}$	6
2	64	53	2	$2^{-53}$	15

- The decimal precision is:
  - PRECISION** =  $-\log_{10}(\text{epsilon}(x))$





## Using Decimal Ranges For Reals

- ▶ Same definition as integers:
  - ◆ **RANGE** =  $\lfloor \log_{10}(\text{huge}(x)) \rfloor$

## Certain Intrinsic Functions Continued

- ▲ For reals, there is an intrinsic function  
**`selected_real_kind(<precision>,<range>)`**  
where `<precision>` and `<range>` are *integer constants* representing the decimal precision and range of the reals whose kind number is to be returned
  - ◆ The `<precision>` and `<range>` are minimum specifications
- ▲ For example, on the Salford compiler,
  - ◆ `selected_real_kind(5)` returns 1
  - ◆ `selected_real_kind(10)` returns 2
  - ◆ `selected_real_kind(5,100)` returns 3



## Examples of Specifications

- Consider the following declarations:

`real( 1) x`

**! Specification of kind = 1**

`real(selected_real_kind(4)) y`

**! Specification of kind = 1**

**! At least precision of 4**

`real(selected_real_kind(4,100))`

**! Specification of kind = 2**

**! At least precision 4 and**

**! with at least range 100**



## Another Specification Intrinsic

- ▶ Another intrinsic is available for integers and reals
  - ◆ **kind(x)** returns the kind number of its argument
    - **kind(0.0)** is the kind number for default real kind
    - **kind(0.0d0)** is the kind number for double precision
    - **kind(0)** is the kind number for default integers



## Inquiry Intrinsics

- ▲ The language provides intrinsics that inquiry about these parameters, related values, and useful algorithmic “constants”
  - ◆ **Assume  $x$  is a declared object with type  $\text{real}(\text{kind}\#)$** 
    - **$\text{radix}(x)$**  returns the radix used for  $x$
    - **$\text{precision}(x)$**  returns the decimal precision for  $x$
    - **$\text{range}(x)$**  returns the decimal range for  $x$
    - **$\text{digits}(x)$**  returns the base- $r$  digits used for  $x$
    - **$\text{radix}(x)$**  returns the base  $r$  for  $x$
    - **$\text{minexponent}$**  returns the minimum exponent for  $x$
    - **$\text{maxexponent}$**  returns the maximum exponent for  $x$



## Inquiry Intrinsic Continued

- **huge(x)** returns the largest value x can have
- **tiny(x)** returns the smallest positive value x can have
- **epsilon(x)** returns the smallest number relative to 1 that changes 1
-



## Using Module Constants

- ▶ The second way to handle the portability issue of processor-dependent constants is to use module constants
  - ◆ Suppose WP is to be the kind number for working precision
  - ◆ Suppose SP is to be the kind number for single precision
  - ◆ Suppose DP is to be the kind number for double precision
  - ◆ Suppose DWP is to be the kind number for double working precision
- ▶ Then place the following declarations in a module

## Using Module Constants Continued

Module precision\_module

! Kind parameters for reals

integer, parameter :: WP = selected\_real\_kind(10)

integer, parameter :: SP = kind(1.0)

integer, parameter :: DP = kind(1.0d0)

integer, parameter :: DWP = &  
selected\_real\_kind(2\*precision(1.0\_WP))

! Kind parameters for integers

integer, parameter :: IR = selected\_int\_kind(4)

integer, parameter :: IDR = kind(0)

end module precision\_module





## An Example: Newton's Method

- ▲ Consider writing a procedure to find the square root of a number  $a$  using Newton's method
  - ◆ The iteration technique is:  $x_{i+1} = 0.5*(x_i + a/x_i)$
  - ◆ Start the iteration at  $a / 2$  -- there are better values
  - ◆ Stop the iteration when  $|x_{i+1} - x_i| \leq \text{"small"}$
  - ◆ What is small?
    - Use the function  $\text{epsilon}(x)$  to determine small
    - It measures a small number relative to 1
    - It measures a unit change in the last digit of precision of the number



## The SQR T Program — Specifications

```
Function my_sqrt( a )  
  use precision_module  
  implicit none  
  real(WP) my_sqrt  
  real(WP), intent(in) :: a  
  intrinsic abs, epsilon  
  ...  
  
end function my_sqrt
```



## The S Q R T Program — Execution Part

```
if( x == 0.0_WP ) then
    my_sqrt = 0.0_WP
else
    x_old = a; x_new = a/2.0_WP
    do while( abs(x_new-x_old) <= abs(x_old)*epsilon(x_old) )
        x_old = x_new
        x_new = 0.5_WP*(x_old + x/x_old)
    end do
    my_sqrt = x_new
endif
```



## A Better Starting Value

- ▶ The problem with this starting value is that when  $a$  is very large or very small in magnitude, the starting value  $a / 2$  is too far away from the root so that the iteration is slow
- ▶ We can use other manipulation intrinsic functions to break  $a$  into its exponent and fractional parts to get a better starting value



## Manipulation Intrinsic

- **exponent(x)** returns the exponent of x in terms of the model
- **fraction(x)** returns the fraction of x in terms of the model
- **set\_exponent(x,i)** returns a number whose fraction is that of x and whose exponent is the value i
- **scale(x,i)** returns x scaled by  $\text{radix}(x)^i$

▲ A better value for the initial  $x_{\text{new}}$  is:

$$x_{\text{new}} = \text{set\_exponent}(1.0\_WP, \text{exponent}(a) / 2 + 1)$$

▲ Use a linear least square approximation to the square root function over the interval (0.5,2.0)



## Generalize To Array Arguments

- ▲ Find the square root of an array, element-by-element
  - ◆ **Use a masked array assignment using the**
    - **WHERE statement or construct**
    - **WHERE-ELSEWHERE block construct**
- ▲ After defining such functions, they can be made generic
  - ◆ **allows my\_sqrt to work on arrays**
  - ◆ **the scalar code generalizes to array code in a simple way**



## The Euclidean Norm of a Vector

- ▶ The Euclidean norm of a vector is:

$$norm = \sqrt{\sum_{j=1}^n x_j^2}$$

- ▶ This computation can overflow or underflow unnecessarily (that is, an intermediate result may get too large or small and yet the result is representable)
- ▶ The solution is to use scaling and epsilon to make the computation more robust



## Algorithm

- ▶ Determine the largest value in magnitude
- ▶ Scale all values by this maximum (or an approximation to it to avoid rounding errors -- use the scale intrinsic function)
  - ◆ **avoid n divisions as they are very expensive**
- ▶ Compute the sum of squares for only those values that are larger than  $\epsilon(x)$
- ▶ Compute the norm (taking cognizance of the scaling)



## Exercises

- ▶ Write a program to determine the supported kind numbers for integers, reals, logicals, complex, and character on IBM compiler using:
  - ◆ **a program fails at compilation time when a specified kind value is not supported by the compiler**
- ▶ Write a computer program to generate the tables on slides 22 and 24 for all supported kinds of integers and reals on the IBM compiler
  - ◆ **On your first cut, print the values as integer or floating point values without using the formulas**
  - ◆ **On a second try, represent the formulas with parameters and print the values of the parameters**



## Exercises Continued

- ▲ Write a test program for `my_sqrt`
  - ◆ **put `my_sqrt` in a module procedure called `my_sqrts`**
  - ◆ **test the square root program in the module**
- ▲ Write a version of `my_sqrt` for kind of SP
  - ◆ **place it in a module and call it `my_sqrt_sp`**
  - ◆ **test it with your test program**
- ▲ Write a version of `my_sqrt` for kind of DP
  - ◆ **place it in the same module and call it `my_sqrt_dp`**
  - ◆ **test it with your test program**



## Generic Procedures — Modules And Interfaces

- ▶ In the example of the square root program, it was written for WP kind numbers
- ▶ We want versions of square root for single and double precision arguments and we want to reference it by the single name `my_sqrt`
- ▶ How do we do it?
  - ◆ **Write 2 versions, one for SP kind and one for DP kind**
  - ◆ **Use an interface to specify the generic name `my_sqrt`**
  - ◆ **All references are to the generic name `my_sqrt`**



## Module My\_sqrt

Module my\_sqrt

use precision\_module

implicit none

interface my\_sqrt

module procedure my\_sqrt\_sp, my\_sqrt\_dp

end interface

CONTAINS

function my\_sqrt\_sp( a )

real(SP) my\_sqrt\_sp, a

...

end function my\_sqrt\_sp



## Module My\_sqrt Continued

```
function my_sqrt_dp( a )  
    real(SP) my_sqrt_dp, a  
    ...  
end function my_sqrt_dp  
end module my_sqrt
```

- ▶ The module procedures can now be referenced by the name `my_sqrt`
  - ◆ **The type of the argument determines which version of square root is called**
    - `my_sqrt(4.0)` calls the single precision version
    - `my_sqrt(4.0_d0)` calls the double precision version
    - `my_sqrt(9.0_WP)` calls the appropriate version



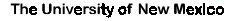
## Exercise

- ▶ Create a module with all supported precisions on the IBM compiler for the Euclidean norm program
- ▶ Write a test program for these norm functions and test the results



## Mixed Mode Expressions

- ▶ The arithmetic and relational operators can combined operands of different types and different kinds
  - ◆ What are the kinds and types of the result?
- ▶ Simplified rule:
  - ◆ order the types from simplest to most powerful
    - integer, then real, then complex
  - ◆ order the kinds within the arithmetic type by
    - increasing ranges for integers
    - increasing precisions for reals
    - increasing precisions for complex



## Mixed Mode Expressions Continued

- ▶ The result of an operation is the most powerful type, precision, and range
- ▶ Examples (assume <operator> is an arithmetic operator):
  - integer <operator> real                  -- real (e.g. I + X)**
  - integer <operator> complex              -- complex (e.g. I\*C)**
  - SP real <operator> DP real                -- DP real (e.g. X\*\*D)**
  - integer <rel\_operator> real                -- default logical with the  
comparison on DP reals  
(e.g. I <= X )**





## Specifying The Kind Of A Constant

- ▶ Literal constants have default kinds, defined by the compiler
  - ◆ **1.0, 2.0e10 -- default real kind**
  - ◆ **1.1d-5 -- default double precision kind**
  - ◆ **1 -- default integer kind**
  - ◆ **(1.0,-1.0) -- default complex kind, same as default real kind**
  - ◆ **.false. -- default logical kind**
  - ◆ **“string” or ‘string’ -- default character kind**

## Literal Constants on Non-Default Kind

- ▶ The kind values for literals of type other than character are specified with:

- ◆ **after the constant after an underscore (\_)**

- a literal integer constant, or
- an integer named constant

1.1_WP	-- real of kind value WP (the value of WP)
3.14159e0_DP	-- real of kind value DP
2.7_4	-- real of kind value of 4
1_IP	-- integer of kind value IP
.false._LP	-- logical of kind value LP
(1.0_WP, 1.0_WP)	-- complex of kind value WP



## Non-Default Kind Values

- ▲ For characters, the kind specification is before the constant

**CK\_'math\_symbols' -- character of kind CK**

- ▲ Named constants have the kind of their declaration

**real(WP), parameter :: tenth = 0.1\_WP**

**complex(DP), parameter :: j = ( 0.0\_DP, 1.0\_DP )**

**character(10,MATH), parameter :: pi = MATH\_' $\pi$ '**

**logical(BIT), parameter :: T = .true.\_BIT**

- where BIT and MATH are named integer constants whose values are logical and character kinds