Fortran 2000

Steve Morgan
Computing Services Department
The University of Liverpool

Overview

- Parameterised Derived Types
- Derived Type I/O
- Other Input/output enhancements
- Some Miscellaneous enhancements

- Parameterised derived types

- d type I/O
  /O
  ss
  ding
  for preconnected units
- essages

- Int
- point

- Integration

- Not covered
  - named scratch fi
  - control of host asso
  - mixed component acc
  - public entities of private
  - renaming of defined ops

Only for my reference purposes

Parameterised Derived Types

Introduction

- In Fortran 90/95 the intrinsic types may be parameterised with :

  kind type parameters for numeric types, e.g.

```
INTEGER, PARAMETER :: wp=SELECTED_REAL_KIND(13)
REAL(KIND=wp) :: arr
COMPLEX(KIND=KIND(0D0)) :: z
```

  or LEN type parameters for character type, e.g.

```
CHARACTER(LEN=24)          :: ch
CHARACTER(KIND=3,LEN=20) :: ch1
```

Introduction

In Fortran 2000 derived types can be parameterised with any number of integer parameters which are **KIND** or **NONKIND** , e.g.

```
TYPE matrix(K,D)
    INTEGER,KIND :: K
    INTEGER, NONKIND :: D
    REAL(KIND=K):: M(D,D)
ENDTYPE
```

This allows declarations of the form,

```
TYPE( matrix(wp,10) )     :: m1
TYPE( matrix(K=wp,D=10) ):: m2
```

Type Parameters

- The declaration of the type parameters is mandatory

```
TYPE matrix(KIND,DIM)
  INTEGER, KIND    :: KIND
  INTEGER, NONKIND :: DIM
  REAL(KIND=KIND):: M(DIM,DIM)
ENDTYPE
```

Note that **INTEGER** in the above is redundant but is required by the current draft!

(Obvious!)Design

Parameterised derived types are designed, as far as possible, to behave like parameterised intrinsic types

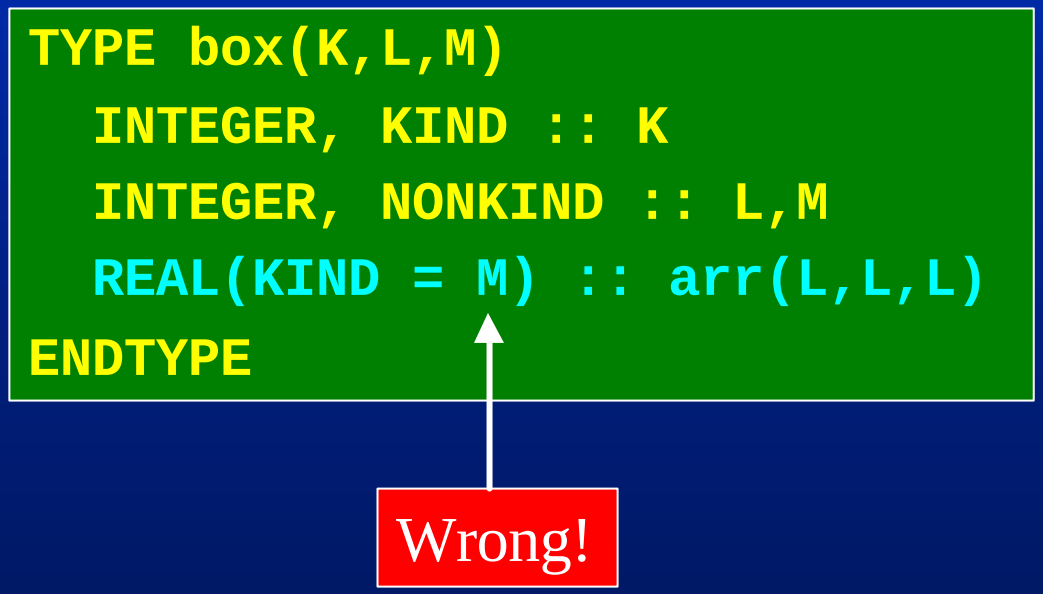
KIND Type Parameters

- A kind type parameter may be used in initialization and specification expressions within the derived-type definition

```
TYPE thing(K,L,M)
  INTEGER, KIND :: K,L,M
  INTEGER(KIND=K) :: arr(L,L,L)= M
ENDTYPE
```

NONKIND Type Parameters

A nonkind type parameter may be used in specification expressions within the derived-type definition for the type, but it may not be used in initialization expressions.



Default Type Parameters

- Defaults for intrinsic types are
  - for **`REAL`** **`KIND=KIND(0.0)`**
  - for **`INTEGER`** **`KIND=KIND(0)`**
  - for **`CHARACTER`** **`KIND=KIND('A'), LEN=1`**
- Note that there are no defaults for derived type parameters, so, for **`matrix`** defined previously, ...

```
TYPE(matrix):: m !--- Compilation Error!
```

i.e. parameters must be specified, e.g.

```
TYPE(matrix(wp,8)) :: m
```

Generics

As for intrinsic types a kind type parameter participates in generic resolution

```
USE ex1
...
TYPE(blob(kind=KIND(0.0))  ::A1
TYPE(blob(kind=KIND(0.0D0))::A2
...
CALL sub(A1) !-- calls spec1
CALL sub(A2) !-- calls spec2
```

```
MODULE ex1
TYPE blob(kind)
  INTEGER, KIND :: kind
  REAL(KIND=kind):: A
ENDTYPE
INTERFACE sub
  MODULE PROCEDURE spec1
  MODULE PROCEDURE spec2
ENDINTERFACE
CONTAINS
  SUBROUTINE spec1(D)
    TYPE(blob(KIND=KIND(0.0))::D
    …
  END SUBROUTINE spec1
  SUBROUTINE spec2(D)
    TYPE(blob(KIND=KIND(0D0))::D
    …
  END SUBROUTINE spec2
END MODULE ex1
```

Example

If desired, a parameter used in a specification expression can be declared as KIND

```
USE ex1
...
TYPE(blob(len=1)):: C1
TYPE(blob(len=2)):: C2
...
CALL sub(C1) !-- calls spec1
CALL sub(C2) !-- calls spec2
```

```
MODULE ex1
TYPE blob(len)
  INTEGER,KIND :: len
  CHARACTER(LEN=len):: C
ENDTYPE
INTERFACE sub
  MODULE PROCEDURE spec1
  MODULE PROCEDURE spec2
ENDINTERFACE
CONTAINS
  SUBROUTINE spec1(D)
    TYPE(blob(LEN=1)):: D
    ...
  END SUBROUTINE spec1
  SUBROUTINE spec2(D)
    TYPE(blob(LEN=2)):: D
    ...
  END SUBROUTINE spec2
END MODULE ex1
```

Type parameter values

- In general type parameters can be any scalar integer expression (but KIND parameters must be known at compile time)
- In addition for NONKIND parameters they can be
  - an asterisk(*) (to indicate an assumed value)
  - a colon(:) (to indicate a deferred value)

Assumed type parameter

As for intrinsic types an assumed type parameter (*) is a nonkind type parameter for a dummy argument that assumes the type parameter value from the corresponding actual argument.

```
TYPE blob(LEN)
  INTEGER, NONKIND :: LEN
  CHARACTER(LEN=LEN):: blib
ENDTYPE
...
```

```
TYPE(blob(LEN=24)) :: A
...
CALL sub1(A)
...
SUBROUTINE sub1(D)
  TYPE(blob(LEN=*)) :: D
...
```

Deferred type parameters (by example)

```
TYPE blob(LEN)
  INTEGER, NONKIND :: LEN
  REAL :: blib(LEN)
ENDTYPE

TYPE(blob(LEN=:)), ALLOCATABLE :: A
TYPE(blob(LEN=:)), POINTER     :: P
...
ALLOCATE(TYPE(blob(LEN=100)) :: A)
ALLOCATE(TYPE(blob(LEN=50)) :: P)
```

```
TYPE(matrix(KIND(0.0D0),m = 10,n = 20):: a
TYPE(matrix(KIND(0.0D0),m =:,n =:), ALLOCATABLE :: b,c
ALLOCATE(b,SOURCE=a)
ALLOCATE(c,SOURCE=a)
```

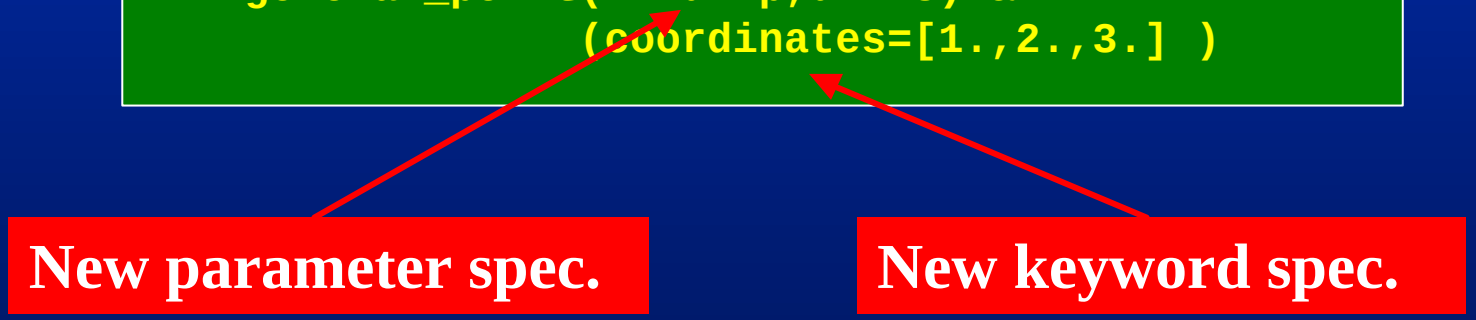
```
ALLOCATE(TYPE(matrix(KIND(0.0D0),m = 10,n = 20)):: b,c )
```

Parameters in derived type constructors

```
TYPE general_point(kind,dim)
  INTEGER, KIND :: kind
  INTEGER, NONKIND :: dim
  REAL(KIND=kind) :: coordinates(dim)
ENDTYPE
```

```
TYPE(general_point(wp,3)) :: P

P = general_point(kind=wp,dim=3) &
                  (coordinates=[1.,2.,3.] )
```



Type Parameter Inquiry

type-param-inquiry is *designator* % *type-param-name*

```
TYPE(general_point(KIND=wp,dim=3)) :: P
Pkind = P%KIND    ! Has value of 1
Pdim  = P%dim     ! Has value of 3
```

Note: Inquiry has same syntax as a structure component reference but has different semantics

Note: For consistency, inquiry syntax can be used with intrinsic types

```
A%KIND !-- A is real.      Same value as KIND(A)
S%len  !-- S is character. Same value as LEN(S)
```

Type Parameter Inquiry

- Note that the inquiry syntax is **not** the same as component selection

- E.g. `P%dim` could **not** appear on the LHS of an assignment statement

rnative proposal

- e syntax
- Instead of
- Main objection - "in e"

Censored

Visibility

- Type parameters are **not** components...
- ... but, when viewed that way, they are effectively always public

  i.e. they are always visible whenever an object of the type is visible

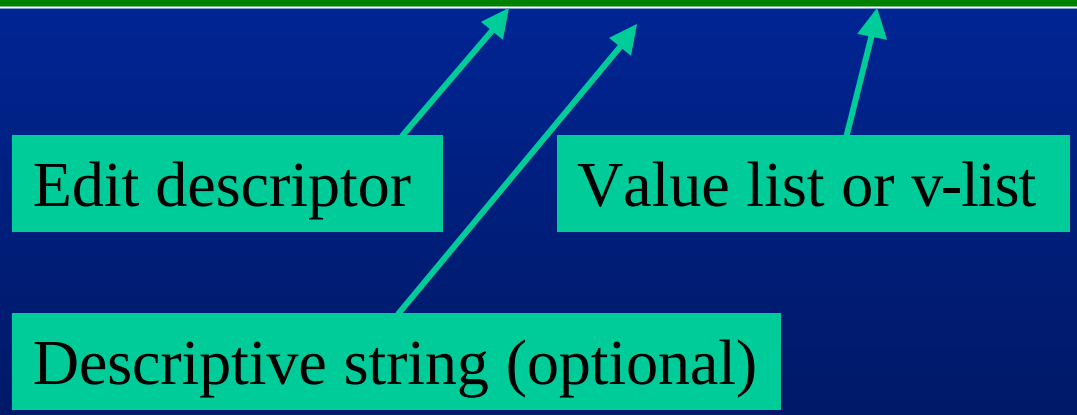
Derived Type I/O (*DTIO*)

Overview

- *User -defined DTIO procedures allow a program to override the default handling of derived type objects and values in I/O statements.*

Example

```
USE Example_module ! On next slide
...
TYPE(array) :: arr
...
WRITE(UNIT=1, FMT= "( DT'string'(10,3) )" ) arr
```



Example module

```
MODULE Example_module
TYPE array
  REAL :: A(10,10)
ENDTYPE
INTERFACE WRITE(FORMATTED)
  MODULE PROCEDURE my_write_formatted
END INTERFACE
CONTAINS
SUBROUTINE my_write_formatted &
           (dtv,unit,iotype,v_list,iostat,iomsg)
...
END SUBROUTINE my_write_formatted
END MODULE Example_module
```

Specific subroutine characteristics

```
SUBROUTINE my_write_formatted &
           (dtv,unit,iotype,v_list,iostat,iomsg)
  TYPE(array), INTENT(IN)          :: dtv
  INTEGER,INTENT(IN)               :: unit
  CHARACTER(LEN=*), INTENT(IN)     :: iotype
  INTEGER,INTENT(IN)               :: v_list(:)
  INTEGER,INTENT(OUT)              :: iostat
  CHARACTER(LEN=*), INTENT(INOUT) :: iomsg
  ...
END SUBROUTINE my_write_formatted
```

Note that the specific names used are arbitrary

Definition of arguments

```
SUBROUTINE my_write_formatted &
           (dtv,unit,iotype,v_list,iostat,iomsg)
```

dtv derived type variable (used for generic resolution)

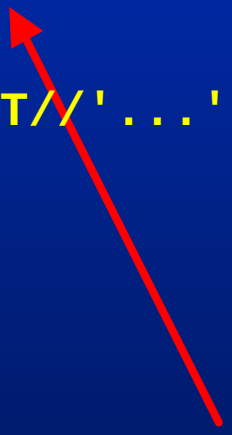
unit unit number in **WRITE/PRINT** statement
or (if **UNIT=***) same value as **OUTPUT_UNIT**
or (if internal file) negative value

iotype **"LISTDIRECTED"** or **"NAMELIST"** or **DT//'...'**

v_list **(/ *list of integer values* /)**

iostat value to be returned in **IOSTAT=** variable

iomsg explanatory message



See next slide

ISO_FORTRAN_ENV

- This intrinsic module provides:

INPUT_UNIT	processor-dependent pre-connected external unit as identified by an asterisk in a READ statement
OUTPUT_UNIT	
ERROR_UNIT	
IOSTAT_END	
IOSTAT_EOR	

In general...

The **INTERFACE** statement is extended to include...

```
INTERFACE READ(FORMATTED)
INTERFACE READ(UNFORMATTED)
INTERFACE WRITE(FORMATTED)
INTERFACE WRITE(UNFORMATTED)
```

... with corresponding characteristics for specific subroutines, e.g.

```
SUBROUTINE my_read_formatted(...)
SUBROUTINE my_read_unformatted(...)
SUBROUTINE my_write_formatted(...)
SUBROUTINE my_write_unformatted(...)
```

Note again: specific names are arbitrary

Interpretation of DTIO

The effect of executing the user-defined derived-type input/output procedure is *similar* to that of substituting the list items from any child data transfer statements into the parent data transfer statement's list items, along with similar substitutions in the format specification.

```
WRITE(*,FMT='(2F10.3,DT'point'(8,2),F10.3)') a, b, point1, c
```

parent

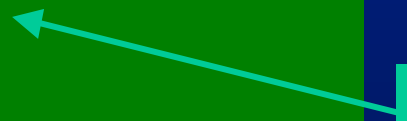
Equivalent to:

```
WRITE(*,FMT='(2F10.3,2F8.2,F10.3)') a, b, point1%x, point1%y, c
```

The above example assumes that point1 is written as 2F8.2 in the user-defined derived-type input/output procedure.

Example of `my_write_formatted`...

```
SUBROUTINE my_write_formatted &
           (dtv,unit,iotype,v_list,iostat,iomsg)
  TYPE(array), INTENT(IN)          :: dtv
  INTEGER,INTENT(IN)               :: unit
  CHARACTER(LEN=*), INTENT(IN)     :: iotype
  INTEGER,INTENT(IN)               :: v_list(:)
  INTEGER,INTENT(OUT)              :: iostat
  CHARACTER(LEN=*), INTENT(INOUT) :: iomsg
  ...
  ! Manipulations to extract v_list values
  ! And construct a format (F8.2)
  ...
  WRITE(UNIT=unit,FMT=...) dtv%x,dtv%y
  ...
END SUBROUTINE my_write_formatted
```



DTIO

- Note that the procedures defining DTIO can also be bound to an object (see later talk?)

```
TYPE something
   REAL :: height
   CHARACTER(LEN=20) :: name
   ...
CONTAINS
   GENERIC :: READ(FORMATTED)  => my_r_f
   GENERIC :: WRITE(FORMATTED) => my_w_f
ENDTYPE something
```

`my_r_f` and `my_w_f` must be module procedures or external procedures with explicit interfaces

Other I/O Enhancements

Asynchronous I/O

Asynchronous I/O

- **`ASYNCHRONOUS=`** on **`OPEN`** and **`READ`**/**`WRITE`**
- **`WAIT`** statement
- **`ID=`** on **`READ`**/**`WRITE`**, **`INQUIRE`**, **`WAIT`**
- **`ASYNCHRONOUS`** attribute and statement
- **`PENDING=`** on **`INQUIRE`**

Example 1

```
REAL    :: arr(large)
...
OPEN(UNIT=10,ASYNCHRONOUS='YES')
...
READ(10, ASYNCHRONOUS='YES') arr
...
<...Code executed while I/O is pending... >
...
WAIT(10)  ! Terminates pending operation
```

- Compiler can "ignore" **ASYNCHRONOUS** I/O (limiting case)
- **CLOSE** or *file positioning* statements cause pending operations to finish.
- **INQUIRE** can also cause pending operations to finish.

Example 2 (ID=)

```
REAL    :: arr1(large),arr2(large)
INTEGER :: async1, async2
...
OPEN(UNIT=10,ASYNCHRONOUS='YES')

...
READ(10, ASYNCHRONOUS='YES',ID=async1) arr1
...
<...Code executed while I/O is pending... >
...
READ(10, ASYNCHRONOUS='YES',ID=async2) arr2
...
<...Code executed while I/O is pending... >
...
WAIT(10,ID=async1)  ! Terminates pending operation
...
WAIT(10,ID=async2)  ! Terminates pending operation
```

Asynchronous I/O

- Not permitted with DTIO (possible, but J3's collective brain hurt when trying to deal with the complications!)
- A variable is an ***affector*** if any part of it is associated with any part of an item in an I/O list of a pending asynchronous operation
- While an asynchronous I/O operation is pending an affector is not permitted to be redefined
- **`ASYNCHRONOUS`** attribute and statement
  - Warns compiler that some code motions across **`WAIT`** statements might lead to incorrect results

Stream I/O

Background

- A file is composed of either a sequence of file storage units or a sequence of records, which provide an extra level of organization to the file.
- A file composed of records is called a record file.
- A file composed of file storage units is called a stream file.
- A processor may allow a file to be viewed both as a record file and as a stream file; in this case the relationship between the file storage units when viewed as a stream file and the records when viewed as a record file is processor dependent.

Background

- A file storage unit is the basic unit of storage in a stream file or an unformatted record file.
- It is the unit of file position for stream access, the unit of record length for unformatted files and the unit of file size for all external files
- Every value in a stream file shall occupy an integer number of file storage units.
- It is recommended that the file storage unit be an 8-bit octet where this choice is practical

Facilities 1

Stream I/O access treats a data file as a continuous sequence of file storage units (usually bytes), addressable by a positive integer starting from 1.

A stream I/O file is opened, e.g.

```
OPEN(..., ACCESS='STREAM',...)
```

A stream file can be positioned to a byte address, e.g.

```
READ(...,POS=<integer>,...)
```

Facilities 2

Inquiries can be made, e.g.

INQUIRE(...,STREAM= *character*

character can be **'YES'|'NO'|'UNKNOWN'.**

(Indicates availability of file for stream access)

INQUIRE(...,ACCESS= *character*

character can be **'STREAM'|'SEQUENTIAL'|'DIRECT'|'UNDEFINED'**

INQUIRE(...,POS= *integer* **)**

integer is number of the file storage unit immediately following the current position

INQUIRE(IOLENGTH= *integer* **)** *<output-item-list>*

integer is the number of *file storage units* required to store the output data

Example

Stream I/O can be very useful when interoperating with files created or read by C programs, as is shown in the following example...

The C program writes 1024 32-bit integers to a file using C fwrite().
The Fortran 2k reader reads them once as an array, and then reads them individually going backwards through the file.
The pos= specifier in the second read statement illustrates that positions are in bytes, starting from byte 1 (as opposed to C, where they start from byte 0).

```
C program writes to a file
 #include <stdio.h>
 int bin_data[1024];
/*Create a file with 1024 32-bit integers*/
int main(void)
{ int i;
 FILE *fp;
for (i = 0; i < 1024; ++i) bin_data[i] = i;
fp = fopen("test", "w");
fwrite(bin_data, sizeof(bin_data), 1, fp);
fclose(fp);
}
```

```
Fortran 2k program reads files
created by C fwrite()

program reader
 integer:: a(1024), i, result
 open(file="test", unit=8,&
   access="stream",&
   form="unformatted")
! read all of a
 read(8) a
 do i = 1,1024
   if (a(i) .ne. i-1)
   print *,'error at ', I
 enddo
! read the file backward
 do i = 1024,1,-1
   read(8, pos=(i-1)*4+1) result
   if (result .ne. i-1)then
     print *,'error at ', i
   endif
 enddo
 close(8)
end program reader
```

Miscellaneous

Control of Rounding in I/O

Control of Rounding in I/O

- **ROUND=** specifier on **OPEN** statement
- **UP**, **DOWN**, **ZERO**, **NEAREST**, **COMPATIBLE**, **PROCESSOR_DEFINED**
- Override with **ROUND=** in **READ**/**WRITE**
- Rounding mode may be changed temporarily with **RU**, **RD**, **RZ**, **RN**, **RC** and **RP** edit descriptors

Access to input/output error messages.

Access to input/output error messages.

- **IOMSG=** specifier
- Identifies a scalar **CHARACTER** variable into which the processor places a message if an error, end-of-file, or end-of-record condition occurs
- Available with **OPEN**, **CLOSE**, **READ**, **WRITE**

Access to Host Environment

Intrinsics provided

```
GET_COMMAND ([COMMAND, LENGTH, STATUS])

COMMAND_ARGUMENT_COUNT ()

GET_COMMAND_ARGUMENT (NUMBER [, VALUE, LENGTH, STATUS])

GET_ENVIRONMENT_VARIABLE (NAME [, VALUE, LENGTH, STATUS,TRIM_NAME])
```

GET_COMMAND

```
SUBROUTINE GET_COMMAND ([COMMAND, LENGTH, STATUS])
  CHARACTER(LEN=*), OPTIONAL, INTENT(OUT) :: COMMAND
  INTEGER, OPTIONAL, INTENT(OUT)   :: LENGTH
  INTEGER, OPTIONAL, INTENT(OUT)   :: STATUS
```

COMMAND entire command by which the program was invoked (character)

If the command cannot be determined, it is assigned all blanks.

LENGTH length of the command by which the program was invoked.

If the command length cannot be determined, a length of 0 is assigned.

STATUS -1 if the COMMAND argument is present and has a length less than the significant length of the command.

It is assigned a processor-dependent positive value if the command retrieval fails. Otherwise it is assigned the value 0

```
SUBROUTINE GET_COMMAND_ARGUMENT (NUMBER [, VALUE, LENGTH,
                                                STATUS])
```

NUMBER the number of the command argument that the other arguments give information about.

Useful values of **NUMBER** are those between **0** and the argument count returned by the **COMMAND_ARGUMENT_COUNT** intrinsic.

VALUE the value of the command argument specified by **NUMBER**(character)

LENGTH the significant length of the command argument specified by **NUMBER**.

If the command argument length cannot be determined, a length of **0** is assigned

STATUS It is assigned the value **-1** if the **VALUE** argument is present and has a length less than the significant length of the command argument specified by **NUMBER**. It is assigned a processor-dependent positive value if the argument retrieval fails. Otherwise it is assigned the value **0**

Example

```
Program echo
integer :: i
character :: command*32, arg*128

call get_command_argument(0, command)
write (*,*) "Program name is: ", command

do i = 1 , command_argument_count()
  call get_command_argument(i, arg)
  write (*,*) "Argument ", i, " is ", arg
end do

end program echo
```

```
GET_ENVIRONMENT_VARIABLE (NAME [, VALUE, LENGTH, STATUS, TRIM_NAME])
```

NAME	name of environment variable
VALUE	value of environment variable (character)
LENGTH	length of value (integer)
STATUS	success or failure (integer)
TRIM_NAME	significance of trailing blanks in **NAME** (logical)

International Usage

Character sets

- **`SELECTED_CHAR_KIND(name)`**
  - returns the kind value of the character set specified by **`name`** as a default integer (-1 if not supported)
  - **`name`** is scalar of type default character which has one of the values:

    `DEFAULT  ASCII  ISO_10646`

    (ISO 10646 is the standard for 4 byte characters)

- Characters added to Fortran character set

  `~ \ [ ] ` ^ { } | # @`

- Only **`[`** and **`]`** **for** array constructor are used in the Fortran syntax

Decimal point representation

Specify when opening a file

```
OPEN(UNIT=10, DECIMAL= 'COMMA')
OPEN(UNIT=11, DECIMAL= 'POINT') ! the default
```

Can be overridden with

```
READ(UNIT=10,... DECIMAL= 'POINT'...)
WRITE(UNIT=11,... DECIMAL= 'COMMA'...)
```

Intended for use in countries where decimal numbers are usually written with commas instead of decimal points

E.g. **`321,123`** instead of **`321.123`**

Summary

Topics Covered

Parameterised derived types

Derived type I/O

Asynch I/O

Stream access

Control of rounding

Named constants for preconnected units

Access to I/O error messages

International usage

Point/comma support

Integration with host operating system

Slides available as PDF at :

http://www.liv.ac.uk/~qq42/BCSSeminar/f2ktalk.htm