Bloomberg

Link to this page: {BP Fortran For C Programmers<GO>}

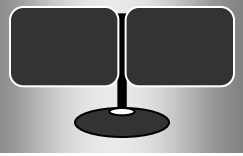
Add to MyBP Favorites

Fortran for C Programmers

Presented by R&D Training

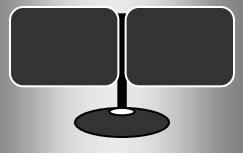
March 2012

Version 8.6



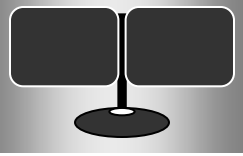
Overview

- Background
- Data types and control statements
- Interoperability between C and Fortran
- C wrappers to Fortran functions



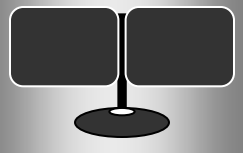
Motivations

- **A bulk of the Bloomberg system is written in Fortran.**
  - **See {ALLX SICC<GO>}**
- **You don’t need to write new Fortran code, but you may need to:**
  - **Understand the functionality of a module written in Fortran.**
  - **Modify a module written in Fortran.**
  - **Write C wrappers to call Fortran functions.**



What is Fortran, anyway

- **Fortran**
  - **The IBM Mathematical FORmula TRANslating System**
- **First ever high-level programming language**
  - **general purpose, procedural**
  - **especially suited to numeric computation and scientific computing**



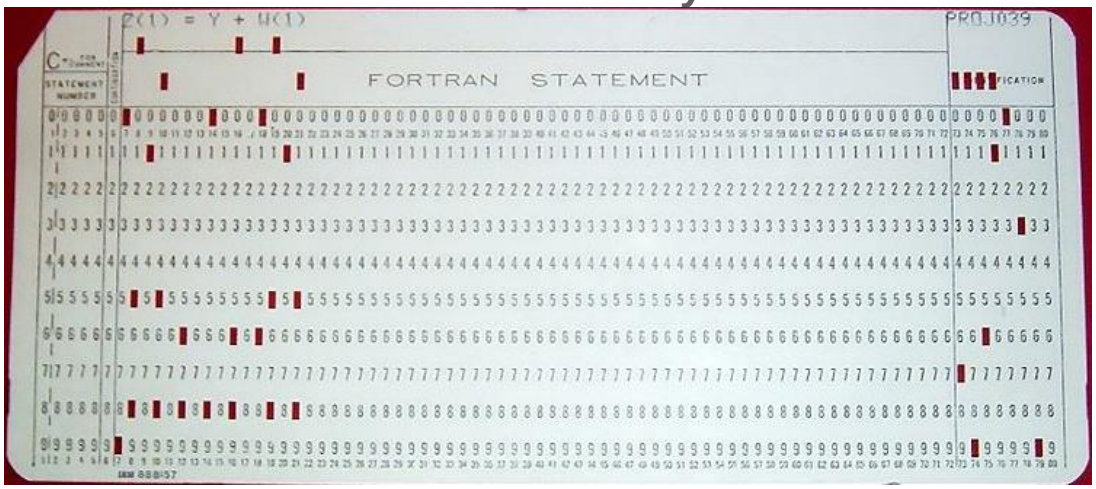
Fortran program: Hello, world

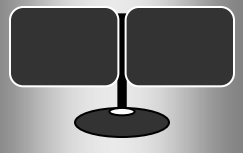
- The *main program* begins with a `PROGRAM` statement followed by a unique symbolic name
- There may be any number of subprograms (`SUBROUTINE`s and `FUNCTION`s).
  - These *program units* must end with an `END` statement.
- Building and running
  - `metamkmk -t other -f hello.mk hello.f`
  - `metalink hello.mk`
  - `./hello.sundev1.tsk`

```
      PROGRAM HELLO
        WRITE(*, *) 'Hello, world'
      END
```

Fixed Format

- **Statement columns (a leftover from punch cards)**
  - **1–5: label field**
    - a sequence of digits
    - a ‘C’ in column 1 indicates that the entire line is a comment
  - **6: continuation field**
    - non-blank character indicates that the line is a continuation of the previous line
  - **7–72: statement**
  - **73–80: danger, stay out!**
    - reserved for identification (e.g., a sequence number to manually reorder the cards in case they are dropped)



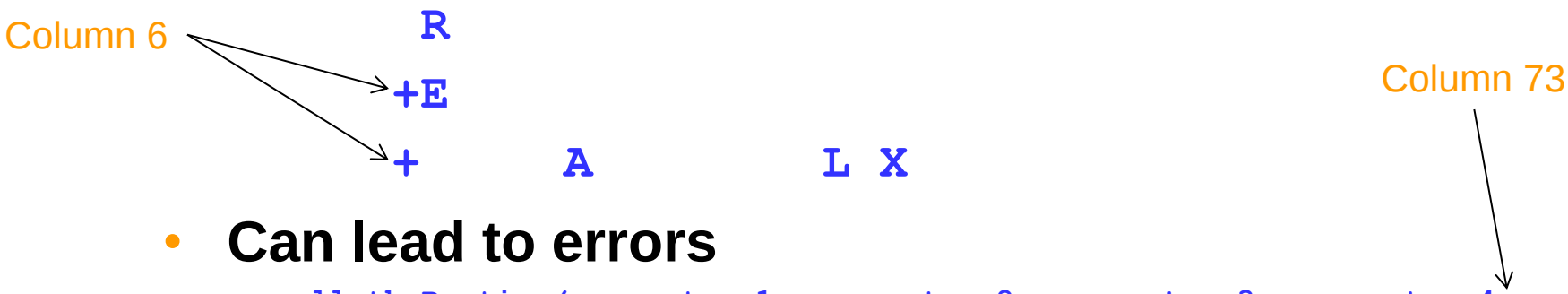


Fortran symbolic names

- Used for variables, arrays, constants, functions, subroutines, and common blocks.
  - Same as identifiers in C
- Must conform to the following rules:
  - the first character of each name must be a letter
  - letters and digits for subsequent characters
- Have an ‘_’ appended by the compiler
  - so they can be distinguished from C symbolic names.
- Converted to all lower case
  - case is ignored except for text strings.
- For interoperability, in C code
  - use all lowercase for C identifiers (C is case sensitive)
  - append an ‘_’ to the identifier

Spacing

- **Spaces are ignored**
  - `REAL X`, `REALX`, and `R E A L  X` are equivalent
- **Tokens can even be separated over several lines (not a good idea)**



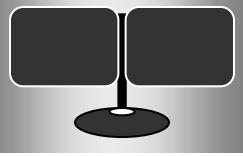
- **Can lead to errors**

```
call theRoutine(parameter_1, parameter_2, parameter_3, parameter_4,
+parameter_5)
```

  - Causes an error stating that a type is required for parameter_4parameter_5.

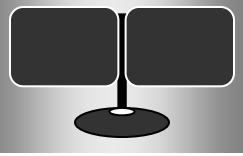
Consistently separating words by spaces became a general custom about the tenth century A.D., and lasted until about 1957, when FORTRAN abandoned the practice.

- Sun FORTRAN Reference Manual



Data types and control statements

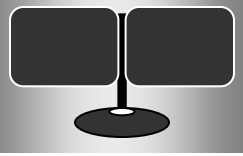
- Basic data types
- Constants and the `PARAMETER` statement
- Expressions
- Control statements (`IF`, `DO`, `WHILE`, and `GO TO`)
- Derived data types (strings, arrays, `EQUIVALENCE`, and `STRUCTURE`)
- Variable initialization (`DATA` statement)



Basic Data Types (1)

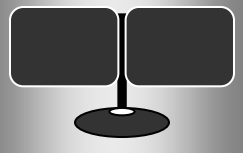
Fortran 77 versus C data types

Fortran 77	C
`integer` (4 bytes in our environment)	`int` (4 bytes in our environment)
`integer*2`	`short`
`integer*4`	`long`
`real`	`float`
`real*4`	`float`
`real*8`	`double`
`real*16`	`long double`



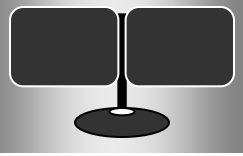
Basic Data Types (2)

Fortran	C
`double precision`	`double`
`character`	`char`
`character*n`	`char[n]`
`byte` (non standard)	`char`
`logical`	`int`
`logical*1`	`char`
`logical*2`	`short`
`logical*4`	`int`



Fortran variables

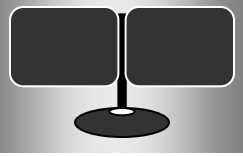
- Variable scope is the program unit (program, function, subroutine) in which it is defined
- Variables need not be declared but it is good programming practice to declare them.
- When a variable is not explicitly declared, the data type is determined implicitly by the first letter of its name
  - `A – H` `REAL`
  - `I – N` `INTEGER`
  - `O – Z` `REAL`
- God is Real, unless declared Integer.
  - J. Allan Toogood, Fortran programmer



Implicit data types

- **Default implicit types can be overridden by**
  `IMPLICIT TypeName (CharacterRange)`
- **For example,**

  `IMPLICIT CHARACTER*40(C-D)`
  - **Identifiers that start with '`C`' or '`D`' will have type `CHARACTER*40`**
- **To enforce explicit types for all variables**
  - **Insert the statement `IMPLICIT NONE` at the beginning of each program unit (i.e., `PROGRAM`, `FUNCTION`, and `SUBROUTINE`)**
  - **Use the `-u` option with the compiler**
    - `f77 -u cool_program.f`
  - **Use metalink/plink, which compiles with the `-u` option**



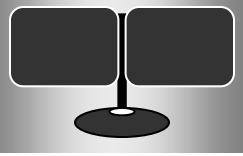
Constants in Fortran (1)

- There are two types of constants
  - Named and Unnamed
- Named Constants
  - To declare, use a data type declaration.
  - To assign a value, use the `PARAMETER` statement.
- `PARAMETER` statement syntax:

```
PARAMETER (cname1 = cexp1, cname2 = cexp2, …)
```

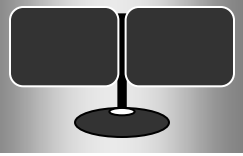
  - For example:

```
INTEGER SIZE
REAL PI
PARAMETER (SIZE = 4, PI = 3.14159265)
```



Input/Output

- **Input/Output**
  - `READ(unit_number, format) arguments`
  - `WRITE (unit_number, format) arguments`
- **Unit_number**
  - A number associated with a device/file
  - `OPEN`/`CLOSE` for file access
  - `UNIT=*` or `*` for standard input/output (`READ`/`WRITE`)
- **Format**
  - A label, format string, or `*` (unformatted)
- **Common Format Descriptors**
  - `rIw` – `integer`(s), repeated `r` times, and `w` digits wide
  - `rFw.d` – `real`(s), with `d` digits to the right of the `.`
  - `Aw` – `character` string, `w` characters wide
  - `rx` – skip spaces



Input/Output

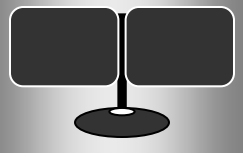
- **Examples**

```
      OPEN(UNIT=15,FILE='input.txt', STATUS='old')
      READ(15, *) value
      CLOSE(15)
      WRITE(*, *) 'Integers: ', 10, 12, ' Reals: ', 33.3, 44.4, 55.5
      WRITE(*, '(2I3, 3F6.2)') 10, 12, 33.3, 44.4, 55.5
      WRITE(*, 10) 'Integers:', 10, 12, ' Reals: ', 33.3, 44.4, 55.5
10    FORMAT(A, 2I3, A, 3F6.2)
```

`old`, `new`, or `unknown`

- **Output**

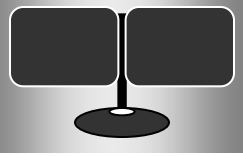
```
 Integers:   10  12 Reals:     33.3000    44.4000    55.5000
 10 12 33.30 44.40 55.50
 Integers: 10 12 Reals:  33.30 44.40 55.50
```



PARAMETER Statement

- **May precede or follow type declarations.**
- **Must follow all type (or `IMPLICIT`) statements which affect the constant data type or length**
  - **Example:**

```
PARAMETER (LENCD = 40, LENE = 2*LENCD)
IMPLICIT CHARACTER*(LENCD)(C-D), CHARACTER*(LENE)(E)
PARAMETER (DEMO = 'This is exactly 40 chars long')
PARAMETER (EXTENDED = '80 chars long')
```

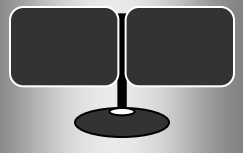


Constants in Fortran (2)

- **Unnamed Constants**
  - **Examples:**

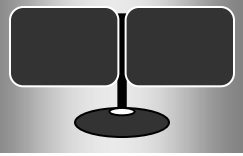
```
1234
12.34
.1234D2
1.0E-02
```

  - **Used explicitly in the source code.**
  - **The compiler infers the data type from the constant's form.**
    - **Allows context-dependent considerations in inferring the data-type. For example if a constant is assigned to a variable, it gets the data type of this variable.**



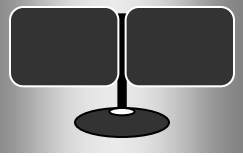
Expressions in Fortran (1)

- **Relational Expressions**
  - Compares the values of two arithmetic expressions or two character expressions.
  - The result is a logical value, either `.TRUE.` or `.FALSE.`.
  - Relational operators
    - `.EQ.` – equal to
    - `.GE.` – greater than or equal to
    - `.GT.` – greater than
    - `.LE.` – less than or equal to
    - `.LT.` – less than
    - `.NE.` – not equal to



Expressions in Fortran (2)

- **Logical Expressions**
  - Can be used in logical assignment statements or control statements.
  - Can have any of the following forms:
    - logical-term
    - `.NOT.` logical-term
    - logical-expression <logical-operator> logical-term
  - Logical operators
    - `.AND.` – logical and
    - `.OR.` – logical inclusive or
    - `.EQV.` – logical equivalence
    - `.NEQV.` – logical non-equivalence aka exclusive or



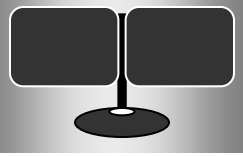
Control Statements: IF

- **`IF` block example:**

```
IF (N .NE. 0) THEN
  AVG = SUM / N
ENDIF
```

- **`IF ELSE` and `ELSE IF` example:**

```
IF (N .LT. 10) THEN
  NEWTOTAL = SUM / N
ELSE IF (N .EQ. 10) THEN
  NEWTOTAL = SUM / N + 10
ELSE
  NEWTOTAL = SUM / N + 100
ENDIF
```



Control Statements: nested IF-ELSE

- **Nested `IF ELSE` Example:**

```
IF (POWER .GT. LIMIT) THEN
  IF (.NOT. WARNED) THEN
    CALL SET('WARNED')
    WARNED = .TRUE.
  ELSE
    CALL SET('ALARM')
  ENDIF
ENDIF
```

- **Logical `IF` Example:**

```
IF (POWER .LT. LIMIT) POWER = 100
```

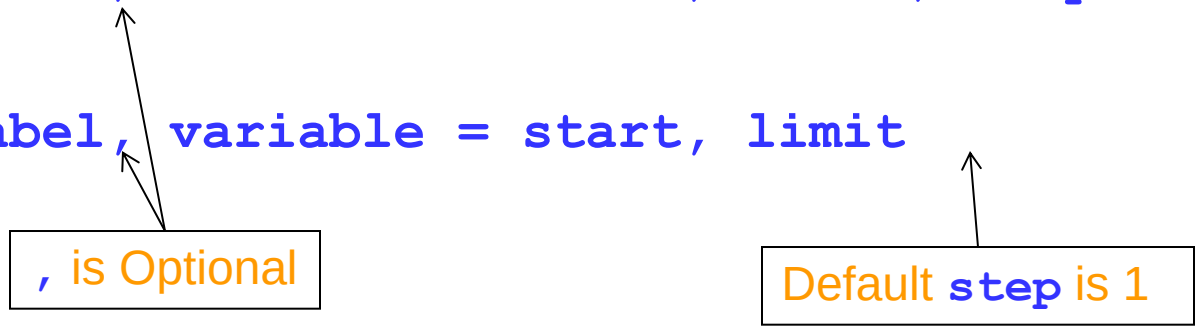
Control Statements: DO

- **Syntax:**

```
DO label, variable = start, limit, step
```

or

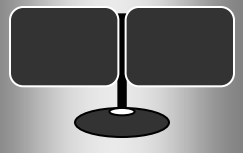
```
DO label, variable = start, limit
```



- **DO example:**

```
        SUM = 0
        DO 15, I = 1, N
           SUM = SUM + I
15      CONTINUE
        OTHER_SUM = 0
        DO 25, I = 1, N
25         OTHER_SUM = OTHER_SUM + I
```

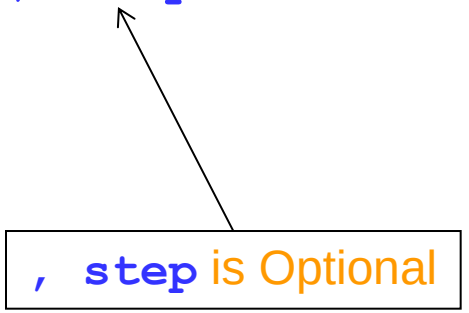
The label is on the last statement in the body of the `DO` loop. A separate `CONTINUE` statement is not required, but is preferred.



Control Statements: DO

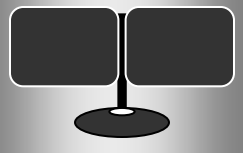
- **Better Syntax:**

```
DO variable = start, limit, step
   …
END DO
```



- **DO-END DO example:**

```
SUM = 0
DO I = 1, N
   SUM = SUM + I
END DO
```



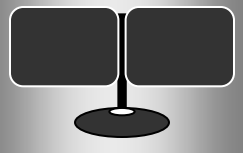
Control Statements: DO WHILE

- **Syntax:**

```
DO WHILE (condition)
   …
END DO
```

- **DO WHILE example:**

```
SUM = 0
I = 1
DO WHILE (I <= N)
   SUM = SUM + I
   I = I + 1
END DO
```



Control Statements: GO TO

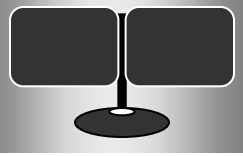
- **Syntax:**

```
GO TO label
```

- **GO TO example:**

```
      SUM = 0
      I = 1
50    IF (I > 5) GO TO 100
      SUM = SUM + I
      I = I + 1
      GO TO 50
100   WRITE (*,*) "SUM = ", SUM
```

- **Used frequently at Bloomberg**
- **Necessary before DO WHILE was added to Fortran**
- **Avoid**



Control Statements: Computed GO TO

- **Syntax:**

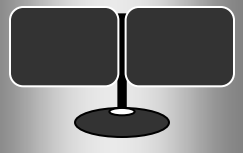
  `GO TO (label_list) expr`

  – `expr` is evaluated and control is transferred to the label at position `expr`

- **Computed `GO TO` example:**

```
      V = someFunction()
      GO TO (3, 4, 15, 10, 10) V
3     WRITE (*,*) "V <= 1 or 6 <= V"
      GO TO 100
4     WRITE (*,*) "V = 2"
10    WRITE (*,*) "V = 2, 4, or 5"
      GO TO 100
15    WRITE (*,*) "V = 3"
100   WRITE (*,*) "Done"
```

- **Used frequently at Bloomberg**
- **Avoid – use an `IF-ELSE` instead**



Control Statements: DO

- **Try it:**
  - Using two `DO` loops, write a program that outputs the odd numbers between 0 and 10, 5 times.
    - Write the loops using one `CONTINUE` statement, two `CONTINUE` statements, and using the `END DO` syntax.
  - Write a program to determine the factors of a number. E.g., the factors of 12 are 1, 2, 3, 4, 6, and 12
    - Hint: `REAL(I)` converts the integer `I` to a real

Fortran character strings

- In C, strings are NUL-terminated character arrays.
- In Fortran, strings are fixed-length blank-padded character variables (no NUL character)

```
character*15 LongString
character*5 ShortString
```

- Truncated if it's assigned to a shorter string

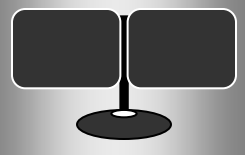
```
LongString = "a long string"
ShortString = LongString
```

Contains "a lon"

- Space-padded if it's assigned to a longer string

```
ShortString = "short"
LongString = ShortString
```

Contains "short "

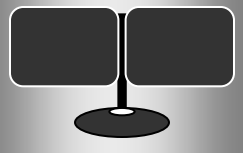


Convert between Fortran and C strings

- **Several routines for converting between Fortran fixed-length blank-padded character variables and C NUL-terminated strings.**
  - **See {BP C from Fortran<GO>}**
  - **For example, to copy the Fortran `fstr` into `cstr`, you can use**

```
int cfrmf(char *cstr, int clen,
          const char *fstr, int flen);
```

  - **Trims the spaces from the right of `fstr`, truncates if necessary, and copies to `cstr` (including the NUL character)**



Arrays in Fortran

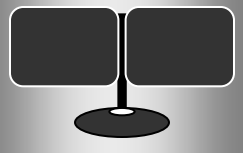
- **Array declaration examples:**
  - **One-dimensional: an array of 8 integers**

    ```
    integer A(8)
    ```

  - **Two-dimensional: an array of 2x7 integers**

    ```
    integer B(2, 7)
    ```

  - **Multi-dimensional arrays: limited to maximum of 7 dimensions.**
- **In Fortran array indexes start at 1**

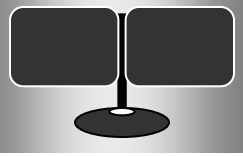


Fortran multi-dimensional arrays

- **Storage sequence:**
  - rows and columns between C and Fortran are switched.
  - Fortran uses column-wise storage of matrices while C uses row-wise storage.

	Fortran	C
Declaration	`integer A(2, 3)`	`int A[2][3]`
Storage layout	<table><tr><th></th><th>1</th><th>2</th><th>3</th></tr><tr><th>1</th><td>10</td><td>20</td><td>30</td></tr><tr><th>2</th><td>40</td><td>50</td><td>60</td></tr></table> 10 40 20 50 30 60	<table><tr><th></th><th>0</th><th>1</th><th>2</th></tr><tr><th>0</th><td>10</td><td>20</td><td>30</td></tr><tr><th>1</th><td>40</td><td>50</td><td>60</td></tr></table> 10 20 30 40 50 60
Address of	`A(r, c) = base + (c-1)*rows+(r-1)`	`A[r][c] = base + r*columns+c`

- **Try it: what is the address of `A(1,2)` assuming the base (starting) address is 0xC000? Of `A[1][2]`?**



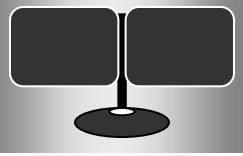
Storage of two dimensional arrays

- **A 2x3 array declared in C but accessed from Fortran**

C	Memory	Fortran
`int A[2][3]`		`integer A(2,3)`
0 1 2 0 10 20 30 1 40 50 60	10 20 30 40 50 60	1 2 3 1 10 30 50 2 20 40 60

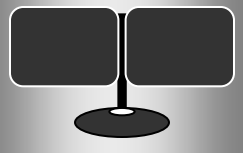
- **A 2x3 array declared in Fortran but accessed from C**

Fortran	Memory	C
`integer A(2,3)`		`int A[2][3]`
1 2 3 1 10 20 30 2 40 50 60	10 40 20 50 30 60	0 1 2 0 10 40 20 1 50 30 60



Arrays of strings

- In Fortran, an array of strings cannot be represented as a two-dimensional array of characters.
- `character*20 array(20)` is correct.
- `character array(20, 20)` is incorrect.



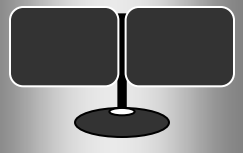
Fortran structure declaration

Fortran:

```
STRUCTURE /POINT/
   REAL X, Y, Z
END STRUCTURE

RECORD /POINT/ CENTER
CENTER.X = 10.1
CENTER.Y = 26.2
CENTER.Z = 101.4
```

C:

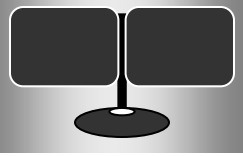
```
struct point {
  float x, y, z;
};

struct point center;
center.x = 10.1;
center.y = 26.2;
center.z = 101.4;
```



Equivalence

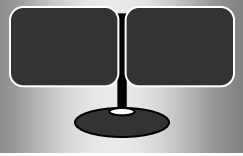
- Same usage as unions in C
- Obsolete in Fortran 77, but in use at Bloomberg
- An example where some integer data is 2 bytes and some is 4 bytes:

```
integer*2 data2(100)
integer*4 data4(50)
equivalence (data4, data2)
```



DATA Statement

- Used to initialize variables and array elements.
- In a `PROGRAM` unit it has the same effect as a set of assignment statements at the beginning of the program.
  - More convenient and efficient since the work is done when the program is loaded.
- In a subprogram (i.e., `SUBROUTINE` or `FUNCTION`), however, a `DATA` statement sets the values only once at the start of the execution, while an assignment statement is executed every time the procedure is called.



DATA Statement: Examples

- **Initializing numeric variables**

```
DOUBLE PRECISION EPOCH
LOGICAL OPENED
DATA EPOCH/195.0D0/, OPENED/.TRUE./
```

OR

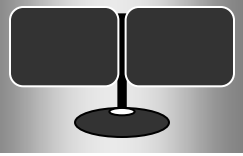
```
DATA EPOCH, OPENED /195.0D0, .TRUE./
```

- **Initializing character variables**

```
CHARACTER*52 LETTER
DATA LETTER(1:26)/'ABCDEFGHIJKLMNOPQRSTUVWXYZ'/,
$     LETTER(27: )/'abcdefghijklmnopqrstuvwxyz'/
```

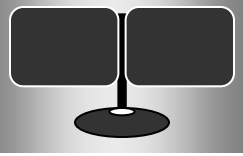
- **Initializing arrays**

```
REAL FLUX(1000)
DATA FLUX /512*0.0, 488*-1.0/
```



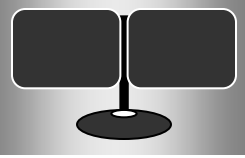
Interoperability between Fortran and C

- Fortran subprograms: functions and subroutines
- Fortran storage rules
- Fortran subprogram call by reference versus C function call by value.
- Fortran common areas



Fortran functions and subroutines

- **Two subprograms in Fortran**
  - A `function` returns a value.
  - A `subroutine` does not.
  - Parameters are passed by reference



Fortran functions and subroutines

- **Define a `function`**
  - `return_type function name(arguments)`
  - **Example**

```
real product(multiplier, multiplicand)
real multiplier, multiplicand
product = multiplier * multiplicand
end
```

The `name` is a variable used to store the return value.

- **Call a `function`**
  - **Assign the result to a variable**
  - **Example**

```
real result, m, n
…
result = product(m, n)
```

Fortran functions and subroutines

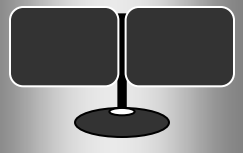
- Define a `subroutine`
  - `Subroutine name(arguments)`
  - Example

```
adder(addend_1, addend_2, sum)
real addend_1, addend_2, sum
sum = addend_1 + addend_2
end
```

- Call a `subroutine`
  - Use the `call` statement
  - Example

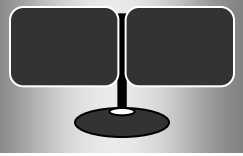
```
real m, n, s
…
call adder(m, n, s)
```

All parameters are called by reference



Fortran functions and subroutines

- **Try it:**
  - **Write a subroutine to output the factors of a number that is passed as an argument. In the main program, read the number from the user.**
    - **Hint: `READ(*,*) I` reads a value from stdin and stores it in `I`.**



Fortran 77 Storage Rules

- **In C, local variables are usually `automatic`**
  - **Activation Record is in the stack**
  - **One instance for each function call**
  - **Recursion is possible**
- **In Fortran 77, local variables are `static`.**
  - **Activation Record is in static memory**
  - **One instance for each subprogram definition**
  - **Recursion is not possible**

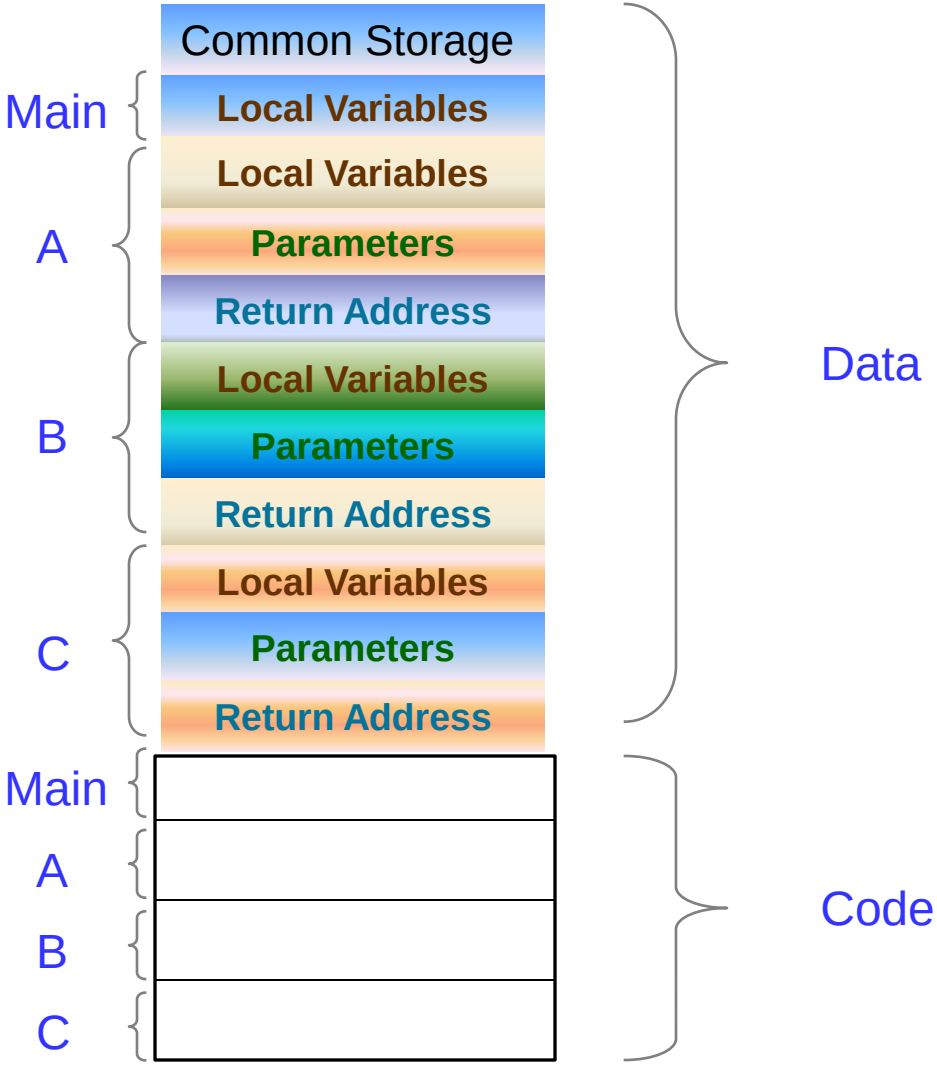
Fortran
Activation Record
(in static memory)

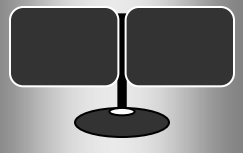
Function value
Local variables
Parameters
Return address

C
Activation Record
(in the stack)

Local variables
Parameters
Dynamic Link
Return address
Return Value

Memory Layout of a Fortran 77 Program





More Storage Rules

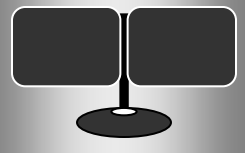
- **Modern Fortran compilers have augmented the Fortran 77 storage rules**
  - **For parameters, temporary variables are created and stored in the stack, if necessary**

```
X = someFunction(Y - 10)
```

    - A temporary variable is automatically created
    - The result of `Y - 10` is stored in the temporary variable
    - The address is passed to the function

```
X = someFunction(Y)
```

    - No temporary variable is necessary
    - The address of `Y` is passed to the function
  - **The function name stores the return value**
    - A temporary variable is created each time the function is called.
  - **The return location is stored in the stack**



More Storage Rules

- **Recursion works!**

The address of `m` is passed to `factorial`

```
…
result = factorial(m)
…
end
```

For each call, a temporary variable is created and used to store the result

```
integer function factorial(n)
integer n
if (n .eq. 0 .or. n .eq. 1) then
   factorial = 1
   return
end if
factorial = n * factorial(n - 1)
end
```

For each call, a temporary variable is created and its address is passed to `factorial`

Storage Rule Limitation

- **Passing constants can cause runtime errors**

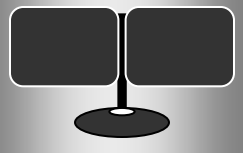
```
…
result = countDown(n)
result = countDown(n - 5)
result = countDown(5)
…
end

integer function countDown(n)
…
n = n - 1
…
end
```

The address of the variable `n` is passed. No problem.

The address of the temporary variable is passed. No problem.

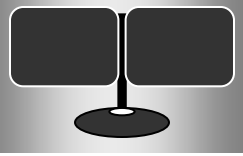
The address of the constant `5` is passed. Runtime error.

```
*** TERMINATING  ./test.sundev1.tsk
*** Received signal 11
POSSIBLE ATTEMPT TO MODIFY CONSTANT
Segmentation Fault
```



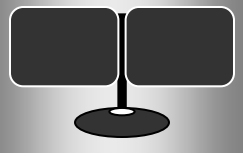
Fortran function or subroutine

- **Two subprograms in Fortran**
  - A `function` returns a value,
  - A `subroutine` does not.
- **When Fortran calls a C function**
  - If the C function returns a value, call it from Fortran as a function.
  - If the C function does not return a value, call it as a subroutine.
- **When C calls a Fortran subprogram**
  - If it is a Fortran function, call it from C as a function that returns a compatible data type.
  - If it is a Fortran subroutine, call it as a function that has a `void` return type.



Fortran Call by reference

- **Fortran passes arguments by reference.**
  - **In C arguments are passed by value.**
- **To call Fortran from C, we have to emulate pass-by-reference.**
  - **Use pointers to memory locations as the arguments.**
- **To call C from Fortran, the C function must be implemented to emulate pass-by-reference.**
  - **Define arguments as pointers**



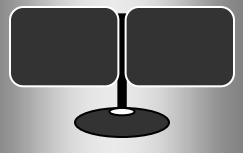
Fortran routine calls

C:	Fortran:

`void foo_(int *a)`
`{`
`  …`
`}`
	`subroutine foo(A)`
	`integer A`
	`…`
	`end`

`double bar_(double *x)`
`{`
`  …`
`  return *x;`
`}`
	`real*8 function bar(x)`
	`real*8 x`
	`…`
	`bar = x`
	`end`

- During compilation, the Fortran compiler converts the subprogram name to lowercase and appends an ‘_’.
- When defining a C function that will be called from Fortran, its name must be lowercase and end with ‘_’.



C code calls Fortran functions: Example

- **Calling a C function from Fortran**
- **Code in Fortran:**

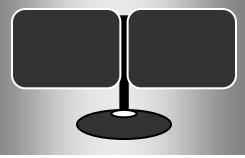
```
external swap
integer i, j
i = 100
j = 101
call swap(i, j)
```

`swap` is converted to `swap_` by the Fortran compiler

Fortran passes parameters as references. The C function must expect pointers.

- **Code in C:**

```
void swap_(int* i, int *j) {
int tmp;
tmp = *i;
*i = *j;
*j = tmp;
}
```



C code calls Fortran functions: Example

- **Calling a Fortran subroutine from C**
- **Code in Fortran:**

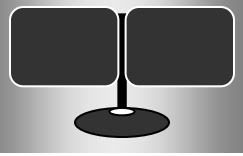
```
SUBROUTINE SWAP(I, J)
INTEGER I, J, TMP
TMP = I
I = J
J = TMP
END
```

`SWAP` is converted to `swap_` by the Fortran compiler

- **Code in C:**

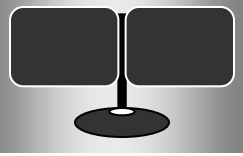
```
int m = 10;
int n = 101;
swap_(&m, &n);
```

Fortran expects parameters as references. The C function must pass pointers.



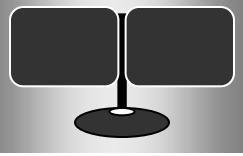
Fortran return code

- Since Fortran passes arguments by reference, the called subprogram can use any argument as either an input or an output.
- At Bloomberg, most Fortran subroutines have `rcode` as their last argument to return an error code.
  - Some exceptions: `parseky4()` use `rcode` as input.



C calling Fortran

- **Try it:**
  - **Write a C program to input an integer from the user and call a Fortran subroutine to output the factors of the number.**
  - **Write a C program that inputs two integers, a base and an exponent, from the user and calls a Fortran function to compute base^{exponent}. Output the result in the C program.**



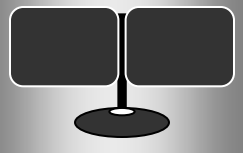
Passing Character Strings as Arguments

- **C programmers must include the size when passing a string to a Fortran subprogram.**
  - **Since the size is an implicit parameter of the Fortran subprogram, it is passed by value instead of by reference.**
- **For example,**

```
int row = 10, startcol = 1, endcol = 5;
const int msgsz = 6;
char msg[msgsz];
snprintf(msg, sizeof(msg), "cool!" );
txtout_(&row, &startcol, &endcol, msg, strlen(msg));
```

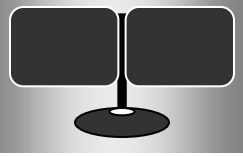
a Fortran subroutine

pass-by-value



Passing Character Strings as Arguments

- **When passing multiple strings to a Fortran subprogram from C, the lengths of the strings are the last arguments.**
  - **If there are `n` string variables passed, then there will be `n` integers representing the lengths of these strings passed as the last arguments.**
  - **The lengths are passed by value and the order is the same as the order of the strings.**
  - **For example, the length of the third string will be the third integer argument.**



Returning character strings from Fortran

- **If a Fortran function returns an n-character string, the first two arguments to the function are actually a pointer to a temporary array and an integer representing the size of the array.**
- **Example:**

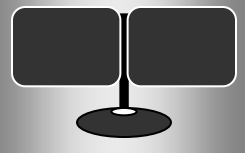
In Fortran:

```
function foo(a)
integer a
character*20 foo
foo = "some message"
end
```

In C:

```
char temp[20];
int a;
foo_(temp, 20, &a)
```

The length of the string is passed by value



Passing Arrays of Character Strings

- **Array and string sizes can be passed explicitly**
- **For example,**

```
program stringTest
   character*(MESSAGE_LENGTH) messages(MESSAGE_COUNT)
   …
   numberOfBlanks = countBlanks(messages,
+          MESSAGE_COUNT, MESSAGE_LENGTH)
end

integer function countBlanks(someStrings, aSize, sSize)
integer sSize, aSize
character*(sSize) someStrings(aSize)
…
end
```

Explicit arguments

Explicit arguments used to define string and array sizes

- **To call from C**

```
char messages[MESSAGE_COUNT][MESSAGE_LENGTH];
int MessageCount = MESSAGE_COUNT;
int MessageLength = MESSAGE_LENGTH;
numberOfBlanks = countblanks_(messages, &MessageCount,
   &MessageLength);
```

Pass-by-reference

Passing Arrays of Character Strings

- **Or implicitly**
- **For example,**

```
program stringTest
    character*(MESSAGE_LENGTH) messages(MESSAGE_COUNT)
    …
    numberOfBlanks = countBlanks(messages)
end

integer function countBlanks(someStrings)
integer sSize, aSize
character*(*) someStrings(*)
…
end
```

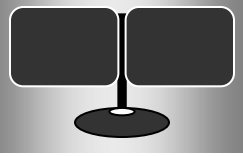
Array size is not defined

Implicit argument used to specify the string size

- **To call from C**

```
char messages[MESSAGE_COUNT][MESSAGE_LENGTH];
int MessageCount = MESSAGE_COUNT;
numberOfBlanks = countblanks_(messages, MESSAGE_LENGTH);
```

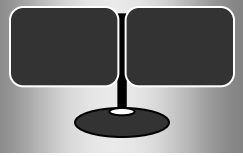
Pass by Value



Passing Arrays of Character Strings

- **For examples of how to pass arrays of reals and strings from one routine to another, see**

  `/bbsrc/training/examples/fortran/passingArraysAndStrings/`



How to write C wrappers for Fortran functions

- For a frequently called Fortran function or subroutine, write a C wrapper. For example,
- Fortran:

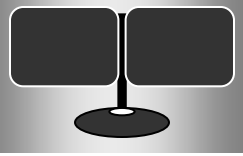
```
SUBROUTINE swap(I, J)
...
END
```

- C wrapper:

```
void swap(int* a, int* b) {
  swap_(a, b);
}
```

- To test (C):

```
int i = 1, j = 2;
swap(&i, &j);
```

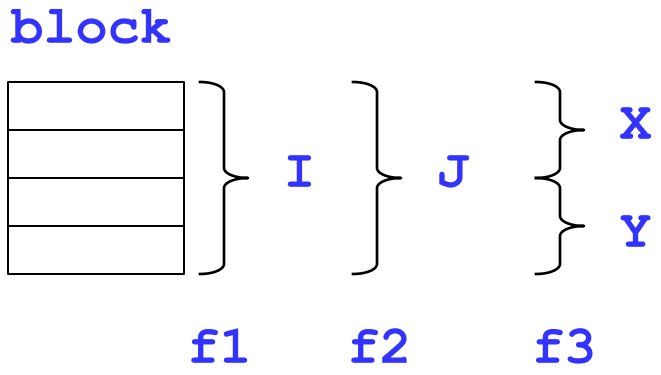


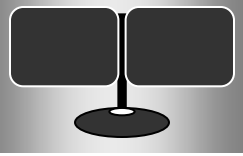
Fortran Common Area

- **Fortran does not have global variables.**
  - **Variable scope is local to its procedure.**
- **Access to local variables of other subroutines/functions can be achieved using *Common Areas*.**
  - **Mapping of a local variable's memory to a storage area that is accessible anywhere in a program.**
  - **Have a global scope, a (dangerous) way to share data between procedures**
- **Common areas can be used with equivalence statements.**

Common Area Example

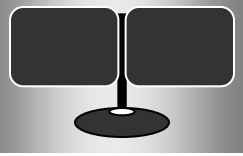
```
subroutine f1
   integer*4 I
   common /block/ I
   I = 5
end
subroutine f2
   integer*4 J
   common /block/ J
   J = 10
end
subroutine f3
   integer*2 X, Y, Z
   common /block/ X, Y
   Z = X + Y
end
```





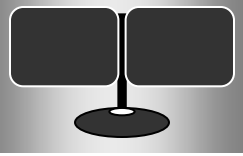
Integrating Common Areas & Structs (1)

- Common areas are mapped to C global variables whose types are a `structs`.
  - Each variable in a common area is a field in the `struct`
- The common area and C global variable reference the same memory.
  - Allows C and Fortran modules to share global data.
- Used extensively in the Bloomberg environment.
  - Bigs (Bloomberg servers)
  - Comdb (Bloomberg databases)
- Can be very dangerous. Data in a common area can be accessed by any subprogram.



Fortran Common Areas and C Struct (2)

- **A C structure declared with an underscore appended can actually reference a Fortran common area, when:**
  - **The name of the C structure variable is all lower case and is the same as the Fortran common area (except the C structure name has the underscore explicitly appended and the Fortran common area name does not).**
  - **The name of the C structure is declared "`extern`" so the linker knows to look for the actual definition elsewhere (i.e., the Fortran common declaration).**



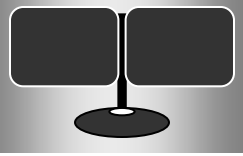
Programming example

- **Fortran:**

```
integer*2 dbbuf(12)
integer*4 funcode
character description*24
real*8 price
common /mydata/ price, description, funcode, dbbuf
```

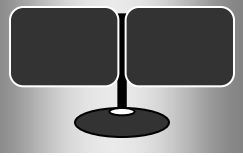
- **C code:**

```
struct DataTag {
   double px;
   char des[24];
   long func;
   short buffer[12];
};
extern struct Datatag mydata_;
```



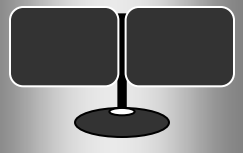
Example explained

- Each of the four variables in the Fortran common area `MyData` are mapped exactly to each of the four members in the C structure `mydata_`.
- For example,
  - `price` is the same as `mydata_.px`.
  - `description` is the same as `mydata_.desc[]`.
- NOTE: Fortran character strings are not NUL-terminated. If `description` is assigned in a Fortran routine, then the 24th byte will not be NUL but will hold character data.



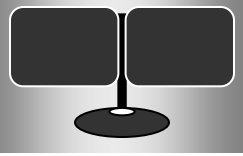
ENTRY statement in Fortran (1)

- `ENTRY` statements are used to specify additional entry points in functions and subroutines.
- `ENTRY` is a non-executable statement which has the same form as a `SUBROUTINE` statement.
- `ENTRY` statements may be used at any point in a procedure but all specification statements relating to its dummy arguments must appear in the appropriate place.



ENTRY statement in Fortran (2)

- The `ENTRY` statement is a deprecated feature in the Fortran standard, but it is in use at Bloomberg.
- The rule to call subroutines/functions from C applies to entry statements as well.



Fortran header files

C:

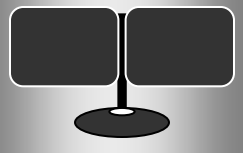
Name: `file.h`

Use: `#include <file.h>`

Fortran:

Name: `file.inc`

Use: `include 'file.inc'`



Fortran References and Resources

- **BP Professional Programmer's Guide to Fortran 77<GO>**
- **BP C from Fortran<GO>**
  - **C/Fortran String Manipulation**
- **BP How to Call a Fortran Routine from C<GO>**

Document History

Date	Comment	Written or updated by
March 2012	Version 8.6	Sean Geoghegan
October 2011	Version 8.5	Sean Geoghegan
October 2011	Version 8.4	Sean Geoghegan
September 2011	Version 8.3	Sean Geoghegan
September 2011	Version 8.2	Tom Zhou
April 2011	Version 8.0, 8.1	Tom Zhou
October 2010	Version 7.4	Updated by Josh Rapps
October 2005	Version 7.0	R&D Training
August 29, 2003	Version 6.0	R&D Training
July 31, 2003	Version 5.0	Updated by Danielle Lahmani
May 23, 2003	Version 4.0	Updated by Danielle Lahmani
May 2, 2003	Version 3.0	Updated by Danielle Lahmani
February 20, 2003	Version 2.0	Updated by Danielle Lahmani
June 2002	Initial version	Danielle Lahmani

Back to Top