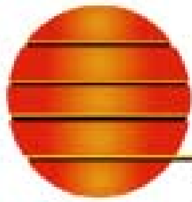
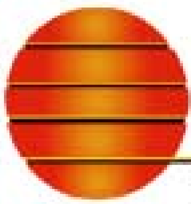


ALBUQUERQUE
High Performance Computing Center

Introduction To Fortran 90/95

Spring, 1999

Richard C. Allen, SNL
Paul M. Alsing, UNM/AHPCC
Andrew C. Pineda, UNM/AHPCC
Brian T. Smith, UNM/AHPCC/CS



The University of New Mexico

Topics Covered Today

- History of Fortran standardization
  - **Why Fortran?**
- Overview of the language
  - **High level view**
  - **basics, procedures, modules, interfaces**

History

- Defined and implemented by Backus and his IBM team in mid 1950s
- Designed to demonstrate that a high level language for scientific computation could be efficient -- generate efficient object code
- Maintained its emphasis on efficiency as well as portability, particularly for numeric computation, throughout its development

History Contd

- Standardized in 1966 by ANSI
- First programming language to be standardized by an official body
- Revised 1978 -- Fortran 77 by ANSI
- Rev.1992 -- Fortran 90 by ANSI and ISO
- Revised 1997 -- Fortran 95 by ISO
- Technical development of Fortran 2K going on now
  - **completion expect in 2002 or so**

References And Textbooks

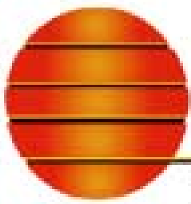
- The Fortran 95 Handbook, Adams, Brainerd, Martin, Smith, Wagener, MIT Press, 1997
- Fortran Top 90, Adams, Brainerd, Martin, Smith, Unicomp, 1994
- Programmer's Guide To Fortran 90, Brainerd, Goldberg, Adams, McGraw-Hill, 1990
- Fortran 90 For Scientists And Engineers, Nyhoff, Leestma, Prentice-Hall, 1997

What Is A Programming Language?

- A tool for instructing machines
- A means of communicating between programmers
- A vehicle for expressing high-level designs
- A notation for algorithms
- A way of expressing relationships between concepts -- a specification language or tool

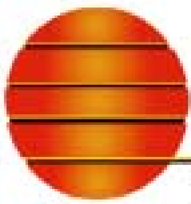
A Programming Language?

- A tool for experimentation
- A means of controlling computerized devices
- A quote from Stroustrup, 1994
  - **NOT a collection of “neat” features**
  - **the workshop necessarily presents a collection of features but the features presented are driven from the applications**



What is Fortran's Niche?

- Efficiency of execution of the object code
  - **Because the standard's definition of the language has the following properties:**
    - **extensive freedom to rearrange code provided the result is mathematically equivalent**
    - **the standard only specifies what the statements of the language mean -- not how they are to be implemented**
    - **no spurious constraints on how the language is to be implemented**



Fortran's Niche Cont'd

- Portability of source code
  - **standardization**
  - **consistency with floating-point hardware standards**
- Major support for scientific (numer) comp
  - **extensive intrinsic library**
  - **extensive numeric precision control**
  - **full support of the complex data type**

Numeric Precision Control

- Multiple floating point precisions
  - **single, double, and possible more**
  - **intrinsic math libraries for all precision provided**
  - **multiple precisions for the complex data type**
- Numeric inquiry and parameterization
  - **machine precision, tiny, huge, …**
  - **consistent with IEEE binary floating-point standard 754**
- Manipulation of numeric parts of floating-point numbers
  - **fraction, exponent, scaling by the base, ...**

Fortran – An Imperative (Procedural) Language

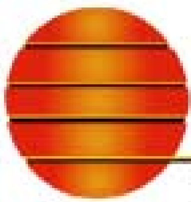
- Other imperative languages
  - **C, Ada, Algols, Pascal, ...**
- Other kinds of languages
  - **functional languages (e.g. Lisp)**
  - **data flow languages (e.g. SISAL, Cantata)**
  - **object-oriented languages (e.g. Smalltalk, C++)**
  - **logic programming languages (e.g. Prolog)**
  - **specification languages (e.g. Ada, Z)**

Characteristic of Imperative Language

- Emphasis on executable statements
- Emphasis on data, types, values, operators
- Emphasis on procedures (functions and subroutines) as a means of extension
  - **to perform new operations or actions not intrinsic in the language**

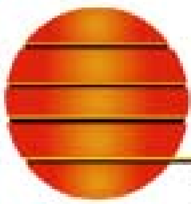
Imperative Language Cont'd

- Emphasis on:
  - **efficiency**
  - **low level**
  - **systems and operations**
  - **imposes no discipline on interfaces between operations, procedures, parts**



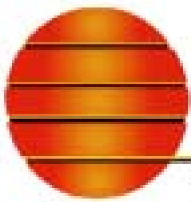
Programming Vs Natural Languages

- Both are a means of communicating, human to machine, human to human
- Both have a structure:
  - **a natural language has a grammar and a set of conventions or rules for expressing thoughts**
  - **a programming language has a STRICT grammar and set of rules**
- Both have statements:
  - **in a procedural programming language, they specify an action or specifies the environment in which the action takes place.**



Comparison Continued

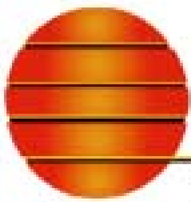
- Both have forms
  - **For a natural language**
    - **books, chapters, paragraphs, sentences**
  - **For a procedural language**
    - **executable programs, program units, constructs, statements**
- Both have rules of well-formed compositions
  - **For a natural language**
    - **interpretation by context, ambiguities**
  - **For a programming language**
    - **strict adherence to rules of grammar and composition**



The University of New Mexico

An Example: DO Loops in Fortran 90

- It is a construct, that is, has some structure
  - **a heading statement**
  - **a block of executable statements**
  - **a terminating statement**
- Alternative forms of heading statement
  - **do**
  - **do while(<exp>)**
  - **do <iteration_clause>**
- Terminating statement (other forms possible)
  - **enddo**

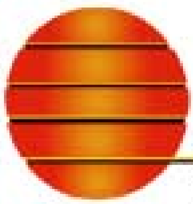


DO Loops Continued

- An iterated DO Loop

```
sum = 0.0
do  j = 1, n
   sum = sum + a(j)*b(j)
enddo
```

- **initial value and final value of do-variable range can be any expression**
- **a stride can be specified as a third item**
  - **it may be negative but must not be zero**
- **any statement can be in the DO-loop body including other DO loops**



DO Loops Continued

- A DO-while loop:

```
continue = .true.
do while ( continue )
   xnew = xold  + step(xold)          ! step is a function
   if( abs(xnew-xold) < epsilon(xold) )  then
      continue = .false.
   else
      xold = xnew
   endif
enddo
```

DO Loops Continued

- The DO forever:

```
do
    ….  ! Before some computation
    ….. ! Check for convergence
    if( converged )  exit
    ….. ! Complete computation for next step
    if( computation_ok )  cycle
    ….. ! Find a better iterate and repeat
    enddo
```

English Vs. Fortran

English	Fortran
Book	Complete program
Chapters	Subprogram
Paragraphs	Parts of a program
Title of chapter	Subprogram header
Intro. paragraph	Specification part
Middle paragraphs	Execution part
Sections	Internal procedures

English Vs. Fortran Cont'd

- Summary paragraph
- Sentences
  - **assertive**
  - **question**
- Words
  - **defined in dictionary**

End statement

Statements

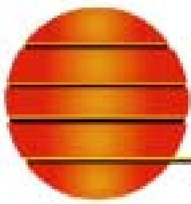
-- spec. statements

-- executable stmts

Tokens

-- identifiers, operators

-- keywords



English Vs. Fortran Cont'd

- Delimiters
  - **periods, commas parenthesis, question marks, blanks**
- Parts of words
  - **letters, apostrophe, digits**
- Side (margin) notes

Delimiters

-- periods, commas, parenthesis, ...

Parts of tokens

-- letters, digits, underscores

Comments

English Vs. Fortran Cont'd

- References
  - see ...
  -
  - [ref]

References

-- CALL statements

-- function references

-- Include statements

-- USE statements

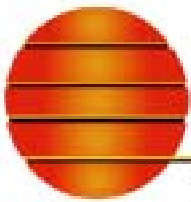
Comparison Summary

- For a natural language, you learn rules for constructing sentences, paragraphs, sections, chapters, books, etc. and memorize meanings of words to construct the other constructs.
- In a programming language, you learn rules for constructing statements, subprograms, and programs, memorize semantics of individual statements, and construct and manipulate data objects with values to create a program.

An Example: hello.f90

- A simple Fortran program

```
program  hello                    ! Header statement

    character(10)  T              ! Specification part
    intrinsic  date_and_time

    call  date_and_time( time = T )      ! Exec. part
    print *, ‘Hello, world: the time is ‘, T(1:2), “:”, &
              T(3:4)

end program  hello                ! End statement
```



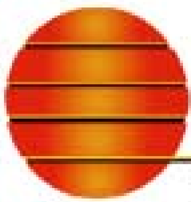
The University of New Mexico

Points To Be Made About Hello.f90

- Program structure
  - header statement (optional for main program
  - specification part (zero or more statements)
  - executable part (zero or more statements)
  - internal procedure part (zero or more statements)
  - end statement
- ALL program units have this structure
  - Program units include:
    - main program, subroutine subprogram, function subprogram, module, …, internal procedure, module procedure

Statements From hello.f90

- Header for a main program
  - **program <name_of_main_program>**
- Specification part
  - **type declaration CHARACTER(<length>) <identifiers>**
    - **defines identifiers to be character strings of a specified length**
  - **intrinsic statement INTRINSIC <names>**
    - **specifies the names are intrinsic procedures**
  - **many other kinds of specification statements**
    - **SAVE, POINTER, ALLOCATABLE, TYPE, DATA, COMMON, INTERFACE blocks, ...**

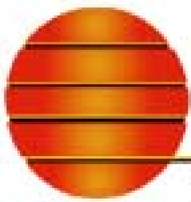


Statements From Hello.f90 Continued

- Executable part
  - **CALL statement -- CALL <name>(<arguments>)**
    - **go to <name> (somewhere else) to execute code and return results, usually as arguments**
  - **PRINT statement -- PRINT <format>, <list_of_output_items>**
    - **print the values of variables listed in the I/O list with specified format**
      - * for <format> means the processor selects a suitable form for printing the listed output items
      - output items can be variables, constants or expressions

Statements From Hello.f90 Continued

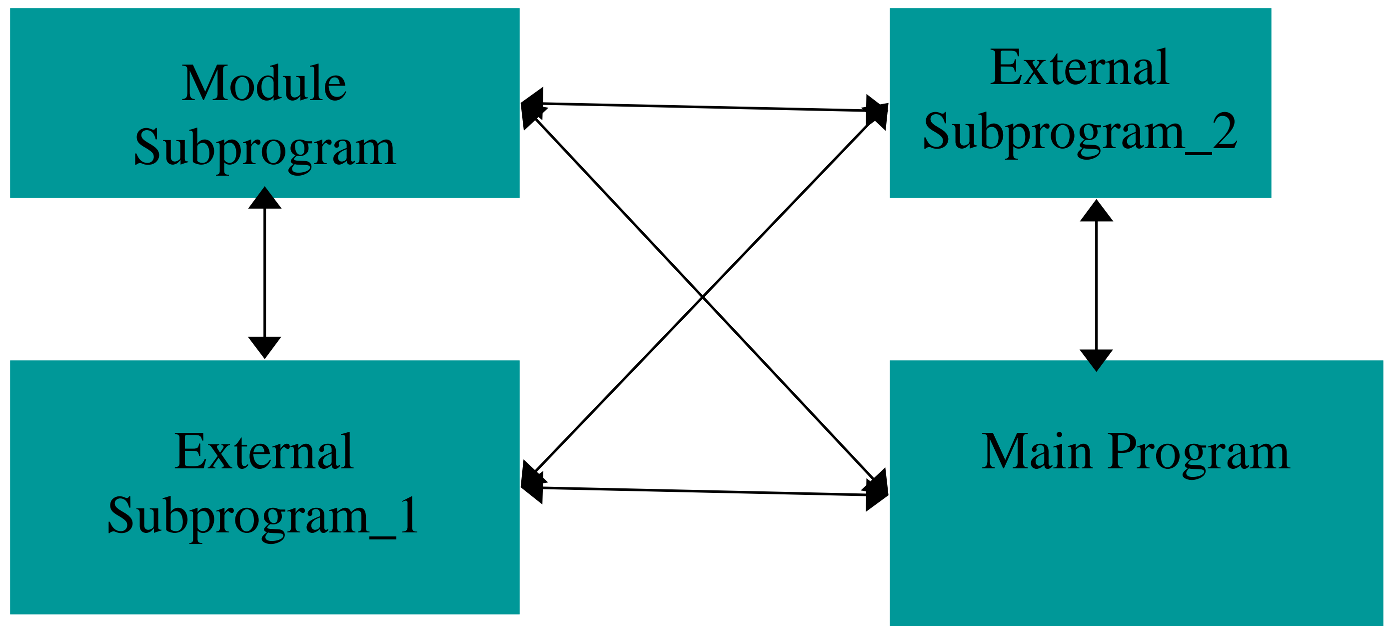
- Executable part continued
  - **other kinds of statements than can be in the executable part**
    - **assignment statements**
    - **control constructs(DO loops, IF construct, CASE construct)**
    - **allocate/deallocate statements**
    - **special array statements (FORALL, WHERE)**
    - **I/O statements (READ, WRITE, OPEN, CLOSE, …)**
    - **pointer-assignment statement**
    - **GOTO, CONTINUE, EXIT, CYCLE, ...**



Statements From Hello.f90 Continued

- Internal procedure part
  - **no internal procedures in Hello.f90**
  - **could be -- begins with a CONTAINS statement**
    - **followed by a subroutine or function program unit**
- END statement
  - **the word END followed optionally by:**
    - **1) the word PROGRAM for the main program**
      - SUBROUTINE for a subroutine
      - FUNCTION for a function
    - **2) the name of the program -- hello in this case**

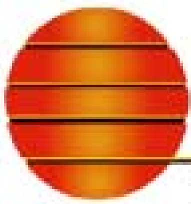
Executable Program – The Whole View



Arrows ↔ indicate association of names

Discussion Of The Whole View

- Program execution starts with the main program
  - **at some point, it calls an external subprogram**
  - **at some point, it calls a procedure in a module**
  - **it makes use of a variable in a module**
- The connection between the main program and procedures or variables is via an association (the lines and arrows)
  - **argument association**
  - **storage association**
  - **USE association**

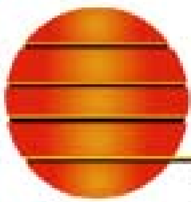


Why Is It Association?

- Most names are local to a program unit in Fortran
  - **For example, in a declaration in the main program**

```
character(10)  T
```

    - **T is a local variable in the main program and is NOT the same as any T in general in any other program unit**
- Global names
  - **external subprogram names, main program names, module names, common block names, and a very few other items**

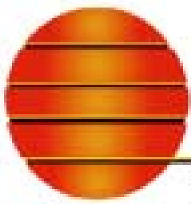


Why Is It Association? Continued

- How does a local variable TIME in a function, say, take the value of a local T in the main program?
  - **Argument association**
    - **while the CALL is being execution, T is the same as TIME**

```
CALL date_and_time( TIME = T )
```

    - **when the CALL completes, the association terminates**
  - **Similarly with storage association, USE association, HOST association, etc. which we will learn about**

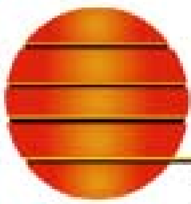


Separate Compilation -- The Whole View

- Fortran permits (and assumes) the concept of separate compilation
  - **external program units can be in separate file and can be compiled separately from parts of the program in other files**
  - **these means that the compiler has to infer what is expected by the called procedure from the call site**
    - **that is, the "interface" to an external program is implicit or guessed from the call site: For example, consider:**

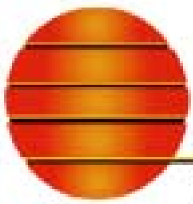
```
call subr( j )
```

      - If j is an integer, the compiler assumes the dummy argument is an integer.



An Example With Functions: simple_ext.f90

```
program  test_function_simple
  implicit  none
  real, parameter :: SMALL = 1.0e-15
  real, parameter :: MODEST = 2.0
  real, parameter :: LARGE = 2.0e15
  integer  i
  real  result
  integer, parameter :: N = 3
  real, dimension(N) :: x
  data  x/SMALL, MODEST, LARGE/
  real, external :: simple ! Necessary because of impl. none
```



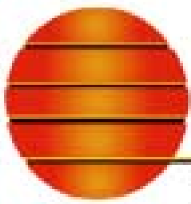
The University of New Mexico

simple.f90 Continued

```
print *, '    x              simple(x)'

do i = 1, N
  result = simple(x(i))
  print *, x(i), result
enddo

end program test_function
```

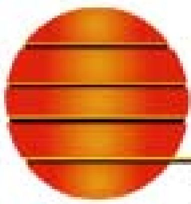
simple.f90 Continued

```
function  simple( x )

   implicit  none
   real  x
   real  simple
   intrinsic  sqrt

   simple = sqrt( 1.0 + X**2 )

end function simple
```



Points About simple.f90

- The previous three slides are in one file
  - **Two program units:**
    - **main program plus a function subprogram**
    - **both are external units**
      - in particular, an external function
  - **The two units may be in separate files**
    - **separate and independent compilation units**
  - **Communication between units:**
    - **argument association only**
      - x(i) in main is associated with dummy x in function, only while the call is being executedo
      - two way association -- both in and out



Points About simple.f90 Continued

- Details about the program
  - **function reference in an expression**
    - **the right hand side of an assignment statement**
    - **this expression is simple here but can be arbitrary expression**
  - **inside the function subprogram, the name simple is both**
    - **the place to define the result as a local variable**
    - **the name of the global function known in other program units**
    - **a result clause can be used to distinguish the two uses**

```
function  simple( x ) result( res )
res = sqrt( 1.0 + x**2 )
```

Points About simple.f90 Continued

- Declarations:
  - **PARAMETER attribute**
    - **defines a named constant -- gives a name to a constant value**
  - **Attributes can be grouped together in a type statement**

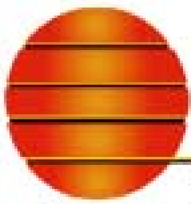
    <type> [, <list_of_attributes> ::] <list_of_identifiers>
  - **Examples:**

```
real, parameter :: modest = 2.0
real, external :: simple
```

  - **The declaration of the name simple is required because implicit none means all identifiers have to be specified**

Interfaces

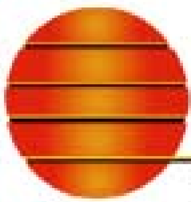
- The declaration of the identifier simple is providing a partial interface to a procedure
  - **An interface consists of:**
    - **the name and number of arguments, and return type**
    - **the type, name, and other attributes of its dummy arguments**
    - **other attributes are:**
      - dimension, pointer, optionality, intent, …
- This interface is partial because:
  - **only types of returned result and argument are known to the calling program unit**



Interfaces Continued

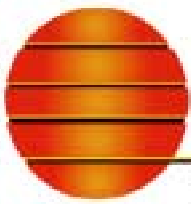
- The complete interface to an external program can be provided by an interface specification block
  - **appears in the calling program in the specification part**
  - **its form is the header, argument specification, and end statement of a subprogram unit**

```
interface
  function simple( x )
    real  x, simple
  end function
end interface
```



Interfaces Continued

- The interface specification makes the interface explicit for external subprogram
  - **otherwise, the interface is implicit**
    - **has to be guessed or assumed by the compiler**
- There are other kinds of procedures
  - **internal and module procedures**
    - **internal procedures are those defined INSIDE a main or procedure program unit**
    - **module procedures are those defined INSIDE a module**
- These procedures also have explicit interfaces because they are visible

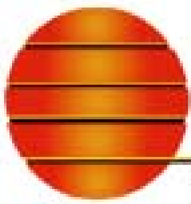


The University of New Mexico

Simple.f90 Using An Internal Function

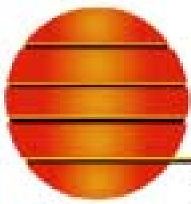
- See file simple_int.f90
  - **The code looks like:**

```
Main Program
      Code for main program
      CONTAINS

            Function simple
                  (all code)

End Main Program
```



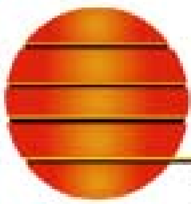
Modules

- Modules are program units than contain:
    - **variables (data)**
    - **constants (fixed data)**
    - **other kinds of data objects**
    - **interfaces**
    - **procedures**
  - **that can be referenced and used by other program units**
- These items are referenced as follows:
  - **a USE statement in any program that wants to use them**
  - **they are known by their name**
  - **a module provides “selective global entities”**



Modules Continued

- “Selective global entities:
  - **known** **only** **to programs that reference the module by a USE statement**
  - **never conflict with local names in program unit that do not reference the module with a USE statement**
  - **In other words, there can be many definitions of the module function simple in the same executable file**
- Permits data abstraction (and data hiding)
  - **can control access to data and access to procedures (using PRIVATE and PUBLIC specifications)**

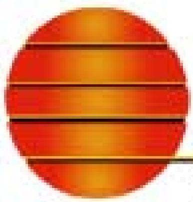


Simple.f90 Using A Module Function

- See file simple_mod.f90

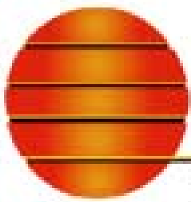
```
Module my_functions
 Module data goes here (if any)
 CONTAINS
      Function simple
           (all code)
 End module my_functions
```

```
Main Program
    USE my_functions
    Code for main program
End main program
```



A Numerically Careful Implementation Of Simple.f90 — Computes $\sqrt{1+x^2}$

- As written, simple.f90 is not very robust
  - **for large x, it may overflow spuriously and either**
    - **stop execution**
    - **continue with an overflow message**
  - **for small x, it may underflow spuriously and either**
    - **continue with no message**
    - **continue with a message**
    - **stop with a diagnostic**
  - **Why is there really no problem?**
    - If x is large in magnitude relative to 1, the answer is x
    - If x is small in magnitude relative to 1, the answer is 1



Careful Implementation Continued

- How do we know what is small or large relative to 1
  - **depends on the precision of the arithmetic used**
    - **suppose 7 digits of precision we used**
      - then, if $|x| > 10^7$, square root of $1 + x^2$ is x within 1 rounding error
      - if $|x| < 10^{-7}$, square root of $1 + x^2$ is 1 within 1 rounding error
    - **TO BE COMPLETED**