King Fahd University of Petroleum & Minerals
Dhahran 31261, Kingdom of Saudi Arabia
Department of Civil Engineering

Aspects of Modular & Structured Programming in FORTRAN

A Presentation to

CE511-Structural Analysis Students

[Dr. Ahmad Sa`ad Al-Gahtani]

By
Mohammed, Aliyu Shazali

04 December 2004

Table of Contents

1. Program Development

- Algorithm Design *
    - Develop the underlying logic of the program
- Program composition *
    - Write the program in a (modular & structured) computer code
- Debugging & testing
    - Ensure program is error-free and reliable
- Documentation (Internal & External)
    - Make the program is easy to use and understand
- Storage & maintenance
    - Save and improve in the light of experience & feedback

2. Algorithm Design

- Sequence of logical steps (recipe) required to perform a specific task within an overall strategy or plan of a program
- Required attributes:
    - Each step must be deterministic
    - Process must always end after a finite number of steps
    - General enough to deal with any contingency
- Algorithm Representation
    - Flowchart (graphical)
    - Pseudocode (code-like statements)
    - Computer code (programs)

3. Program Composition

- Simple information representation (declarations)
- Advanced information representation (data structures)
- Mathematical formulas (assignment & functions)
- Input & Output
- Modular programming (functions & subroutines) **
- Structured programming (sequence, selection & repetition) **

4. Modular Programming

- Programs should be written to emphasize clarity
- Introduce subprograms or modules at the earliest opportunity
- Use top-down-design process to systematically identify program's modules

5. Subprogram

- Subprogram is an independent module containing series of computer instructions that together that form the key to top down software design.
- Advantages of Subprograms
  - You can write and test one module separately from the rest of the program.
  - Debugging is easier because you work with smaller sections of the program.
  - Modules can be used in other programs without rewriting or retesting.
  - Programs are more readable.
  - More than one programmer can work independently on a program
  - Programs are shorter, therefore less subject to error.
  - A module can be used several times by the same program.

6. Types of Subprograms

- Subroutines are used to return multiple values.
- Functions are used to return one value that is equated to the name of the function.
  - Intrinsic functions are built-in functions

7. Types of Function Subprograms

- Statement Functions
  - Single line functions, non-executable statement defined at the top of the program.
  - If computations can be written in one line use Statement Function
- Function subprograms
  - User defined function
  - If computations require more than one line use Function subprogram

8. Rules for Using Subroutine Subprograms

- The subroutine does not represent a value
- A subroutine is referenced with an executable statement of the form:
  CALL subroutine name (argument list)
- The argument list is used for input and output data
- The arguments in the subroutine must match in number, type, and order with the arguments defined in the program

9. Rules for Using Function Subprograms

- Arguments must match in type and order
- The value to be returned is stored in the function name using an assignment statement.
- Control is returned to the main program using the RETURN statement.
- A function may not call itself but can call other functions
- A function must be defined at the beginning or end of the program, or in a separate file.

- A main program and subprograms can be stored in the same file or in separate files
- Variable names can have different meanings in the main program and functions
- Do not expect changes in the arguments as a result of changes made by the function
- Explicitly define the type of the functions

10. Subroutines Differences with Functions:

- Subroutines return multiple values through an output argument list
- The name is not equated to the result.

11. Structured Programming

A set of rules that prescribe good style habits for the programmer to compose any algorithm

- Structure principle:
  The static (coded) structure should correspond in a simple way to the dynamic (actual) structure
- Logical subdivisions or representations:
  - Programs should consists solely of the three fundamental control structures of sequence, selection & repetition
  - Each structure must have only one entrance & one exit
  - Avoid unconditional transfers (GO TOs)
  - Use comments and visual devices to identify the structures
- Software parameterization
  - Data representations
  - Input-output devices
  - Functionality isolation

12. Fundamental Control Structures:

- Sequence: step-by-step, one at a time instruction, structure

```
Instruction 1
Instruction 2
Instruction 3
Instruction 4
```

- Selection: program structure flows based on logical condition
  - Single-alternative decision:

```
IF condition THEN
       TRUE block
ENDIF
```

  - Double-alternative decision:

```
IF condition THEN
       TRUE block
ELSE
       FALSE block
ENDIF
```

  - Multiple-alternative decision:

```
IF condition-1 THEN
       block-1
```

```
ELSEIF condition-2 THEN
      block-2
ELSEIF condition-3 THEN
      block-3
ELSE
      block-4
ENDIF
```

- Repetition: provides loops constructs that implements repeated instructions
  - Decision loop:

```
DO
    block-1
    IF  condition EXIT
    block-2
ENDDO
```

  - Count controlled loop:

```
   DO 9 i=istart, ifinish, istep
       block-1
9  CONTINUE
```

13. Good Programming Practices

- Clarity of expression: Never sacrifice clarity of expression for cleverness of expression. Never sacrifice clarity of expression for minor reductions in machine execution time. Avoid confusing programming tricks. Always strive for simplicity and clarity.
- Names: Pick good mnemonic names for all variables. Pick good mnemonic names for all procedures and functions. Use standardized prefixes and suffixes for related variables. Assign names to scalar constants where it will help clarify the readability. Do not make up cryptic or unclear abbreviations for variables and do not use "cute" names that do not have mnemonic value.
- Commenting: Use prologue comments. Comment the declaration of variables. Paragraph all modules with either comments or blank lines. Comment any statement whose intent is not immediately obvious. Include comments from the very beginning of the coding phase. Do not let comments get out of date with the code.
- Indentation: Do intent to highlight the nesting depth of a group of if- statements or do-loops.
- Input validation: Always validate input for legality and for plausibility and echo print the input. If an error is detected, try to capture enough information to identify the exact cause of the error. Your goal should be to write a program that is protected against all improper data.
- Defensive programming: Prevent run-time errors by checking for them before any risk operation. Never assume exact equality of real values. Produce a meaningful error message directly related to the specific error. Never stop if something useful can be done. Check to see that you handled the null cases properly.
- Use of submodules: Protect input parameters. Avoid procedures with side- effects. Make all temporary variables local to the submodule where they are used. Where appropriate use signal flags to return the status of a computation to the calling program.
- Program Independence: Try to develop programs that are independent of: (a) the machine environment (portability) by using standard FORTRAN 77, by avoiding machine dependent constants, by avoiding specific collating sequences, by localizing and identifying unavoidable machine-dependent information. (b) The data set (generality) by

making key data values parameters to procedures not local variables, by considering how to achieve generality from the earliest stages of program design and specification.

- I/O behavior guidelines: Always identify the input you are requesting. Avoid programs that trap the user in an infinite loop demanding correctly structured input, without offering assistance in preparing it. Use a “help” mode to aid users prepares input. Have users supply input in a form most natural to them, not to the program. Use defaults to reduce the amount of input data required for flexible programs.

14. Case Study

Consider an exercise specified to add 2 numbers. Your program must ask the user for 3 numbers, read them as REAL numbers, add them together, and print out the result.

- Program approaches:
  - Simple program
  - Modular program

==============
Simple Program
==============

```
      program add2
c
c     This program takes two numbers and adds them together
c         a - one of the numbers to be added
c         b - the other number in the sum
c         s - the sum of a and b
c
c     Get the numbers from the program user
c     First ask for the numbers
c
      print *, ' This program adds 2 real numbers'
      print *, ' Type them in now separated by a comma or space'
c
c     Now read the numbers that are typed by the user
c     this Fortran read will wait until the numbers are typed
c
      read *, a,b
c
c     Now calculate the sum
c
      s = a + b
c
c     Print out the results with a description
c
      print *, ' The sum of ', a,' and ' , b
      print *, ' is ' , s
      stop
      end
```

==============
Modular Program-1
==============

```
      program add2
c
c   use subroutine input to get the values for a and b
c
      call input(a,b)
c
c    find the sum of and b
c
      call add(a,b,s)
c
c     use the subroutine output to send the results to the screen
c
      call output(a,b,s)
c
    stop
      end
c
c-------------------------------
      subroutine add (a,b,s)
c-------------------------------
c
c   Add two numbers and store the sum in "s"
      s = a + b
      return
      end
c
c-------------------------------
      subroutine input (a,b)
c-------------------------------

      print *, ' This program adds 2 real numbers'
      print *, ' Type them in now separated by a comma or space'
c
c   Now read the numbers that are typed by the user
c   this Fortran read will wait until the numbers are typed
c
      read *, a,b
c
      return
      end
c
c-------------------------------
      subroutine output (a,b,s)
c-------------------------------
c
c   Print out the results with a description
c
      print *,  ' The sum of ', a,' and ' , b
      print *, ' is ' , s
c
      return
      end
```

===============
Modular Program-2
==============

```
      program add2
c
c   use parameter to declare constants for array sizing
      parameter (nsp=24)
c
c  declare array dimension
      dimension a(nsp), b(nsp), s(nsp)
c
c   use subroutine input to get the values for a and b
      call input(a,b,nsp,nsd)
c
c    find the sum of a and b
      call add(a,b,s,nsd)
c
c     use the subroutine output to send the results to the file
      call output(a,b,s,nsd)
c
      stop
      end
c
c-------------------------------
      subroutine add(a,b,s,nsd)
c-------------------------------
c  declare variable array dimension
      dimension a(nsd), b(nsd), s(nsd)
c
c  Loop over to add two arrays and store the sum in "s"
      do 10 i=1,nsd
      s(i) = a(i) + b(i)
   10 continue

      return
      end
c
c-------------------------------
      subroutine input(a,b,nsp,nsd)
c-------------------------------
c  declare array dimension
      dimension a(nsp), b(nsp)
c
      print *, ' '
      print *, ' This program adds 2 arrays of real numbers'
      print *, ' saved in a file named add2.inp'
c
c  Declare open the input file
      open (unit=1, file='add2.inp', access='sequential',
     .action='read', status='old', form='formatted')
c
c  Now read the arrays
c  first the count of students
      read(1, *) nsd
      print *, ' '
      print *, ' Student count, nsd=',nsd
c
c then a count controlled loop over the two arrays
      do 10 i=1,nsd
      read(1,*) a(i),b(i)
   10 continue
c
      return
      end
c
c-------------------------------
```

```
      subroutine output (a,b,s,nsd)
c--------------------------------
c  declare array dimension
      dimension a(nsd), b(nsd), s(nsd)
c
      print *, ' '
      print *, ' program results are printed'
      print *, ' to an output file named add2.out'
c
c  Declare open the output file
      open (unit=2, file='add2.out', access='sequential',
     .action='write', status='replace', form='formatted')
c
c   Print out the results with a description
      write(2,15) nsd
   15 format(/ 'Student count, nsd='i5)

      write(2,25) 'stID','a(i)','b(i)','s(i)'
   25 format(/a10, 3a15)
      do 10 i=1,nsd

      write(2,35) i,a(i),b(i),s(i)
   35 format(i10,3f15.3)

   10 continue
c
      return
      end
```

15. add2.inp File Sample

```
5
1.       2.
3.       4.
5.       6.
7.       8.
9.       10.
```

16. add2.out File Sample

```
Student count, nsd=    5

      stID           a(i)           b(i)           s(i)
         1          1.000          2.000          3.000
         2          3.000          4.000          7.000
         3          5.000          6.000         11.000
         4          7.000          8.000         15.000
         5          9.000         10.000         19.000
```

17. Readings

1. http://cac.psu.edu/~jhm/jhm.html
2. http://users.otenet.gr/~kchas/index.htm
3. http://www.star.le.ac.uk/~cgp/prof77.html