Chapter 4
Fortran subprogram

- Basics of Fortran subprogram
- Program design strategy
- Why we need to modularize?
- Modularization of N^{th} prime problem
- Fortran Function
  - Fortran Intrinsic functions
    - Fortran Elemental Intrinsic functions
    - Fortran Inquiry Intrinsic functions
    - Fortran Transformational Intrinsic function
  - Fortran external functions
- Fortran Subroutines

Basics of Fortran subprogram

- What is a subprogram?
- Why we need sub program?
- Is it a must to have it?
- How we can define and implement them?
- How we can use them?
- Others

Basics of Fortran subprogram

- Problem Solving != Debugging
- Problem Solving != writing a code that solves the program
- Consider a program designed to sort data
  - It does successfully for the first set of test data
  - Does it mean it will work for the others?
  - Can a program that sorts list of numbers also sort list of names

Basics of Fortran subprogram

- Hence,
  - Solutions must be generic, reusable, easily modifiable, portable, etc
  - There are usually many possible solutions to a given problem and we should use the best

Basics of Fortran subprogram

- Typical steps involved in problem solving include:
  - define and analyse the problem to be solved
  - clearly investigate and understand the problem
  - select best option in terms of algorithm, solving technique
  - design selected solution
  - implement the designed solution
  - Deploy the system

Program design strategy

- When you attempt to solve complex problem,
  - Think "small" rather than "big"
  - Try to identify each of the small, ordinary tasks necessary to accomplish a big job
  - A "mini-program" can then be written to achieve each of these smaller tasks.
  - Later these mini-programs can be joined to form a single program.
- Using several small programs to accomplish a complex task is often more effective than using one very large program.

Program design strategy

- Two approach
  - Top-down design & Step-wise refinement
    - Appropriate during designing the solution for the problem
  - Bottom-Up design start solving minor problems before thinking the Big
    - Appropriate during implementing the design for the problem

Program design strategy

- Basic philosophy of Top-Down approach are *Decomposition & Abstraction.*
  - **Decomposition** is the process of breaking a problem down into individual manageable sub-problems

    •Each sub-problem then can be decomposed further until it can be solved directly
  - **Abstraction** involves considering sub-problems in isolation to reduce complexity.

Program design strategy

- Basic philosophy of Bottom-up approach is *Integration.*
  - **Integration** is the process of bringing solutions of smaller components and use them to solve larger problem

Program design strategy

- A key element in structured programming is the use of modules

- In Fortran, user-defined modules are implemented as **functions** or **subroutines**

- Functions and subroutines are also called ***subprograms***.

Why we need to modularize?

- The following are some important advantages to using modular programming:

1. You can write and test each module separately from the rest of the program.
2. Debugging is easier because you work with smaller sections of the program.
3. Modules can be used in other programs without rewriting or retesting.
4. Programs are more readable and more easily understood because of the modular structure.

Why we need to modularize?

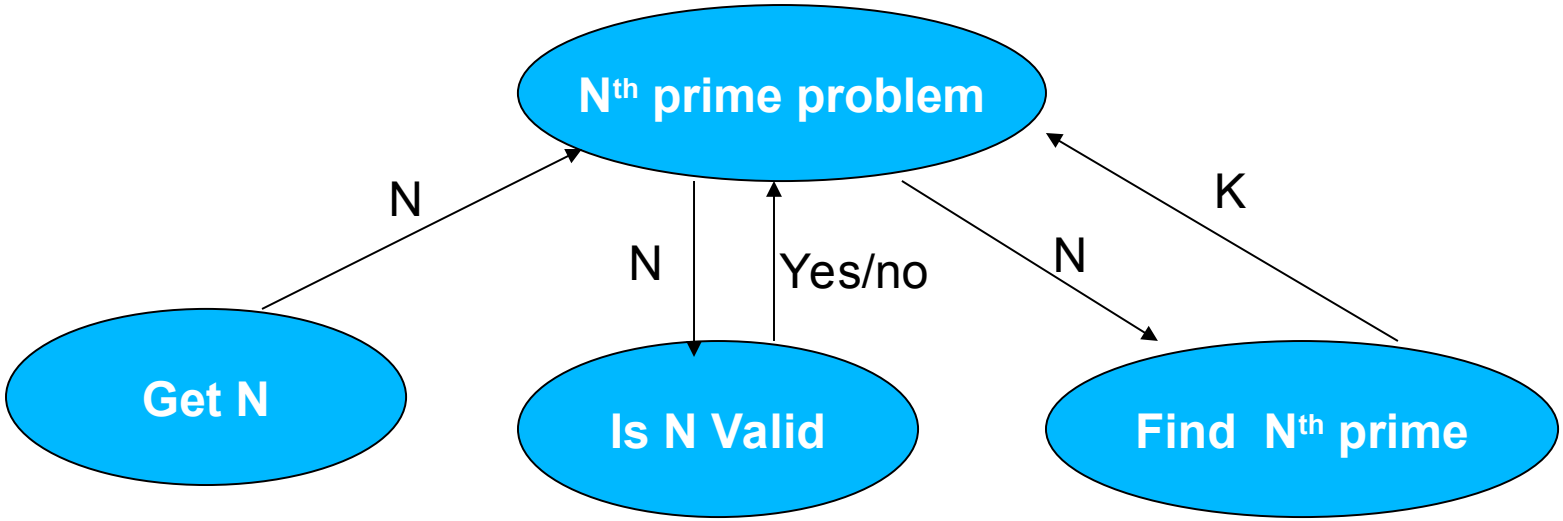
5. Several programmers can work on modules independent of each other.
6. Individual parts of the program become shorter and therefore simpler.
7. A module can be re-used by different program.

Why we need to modularize?

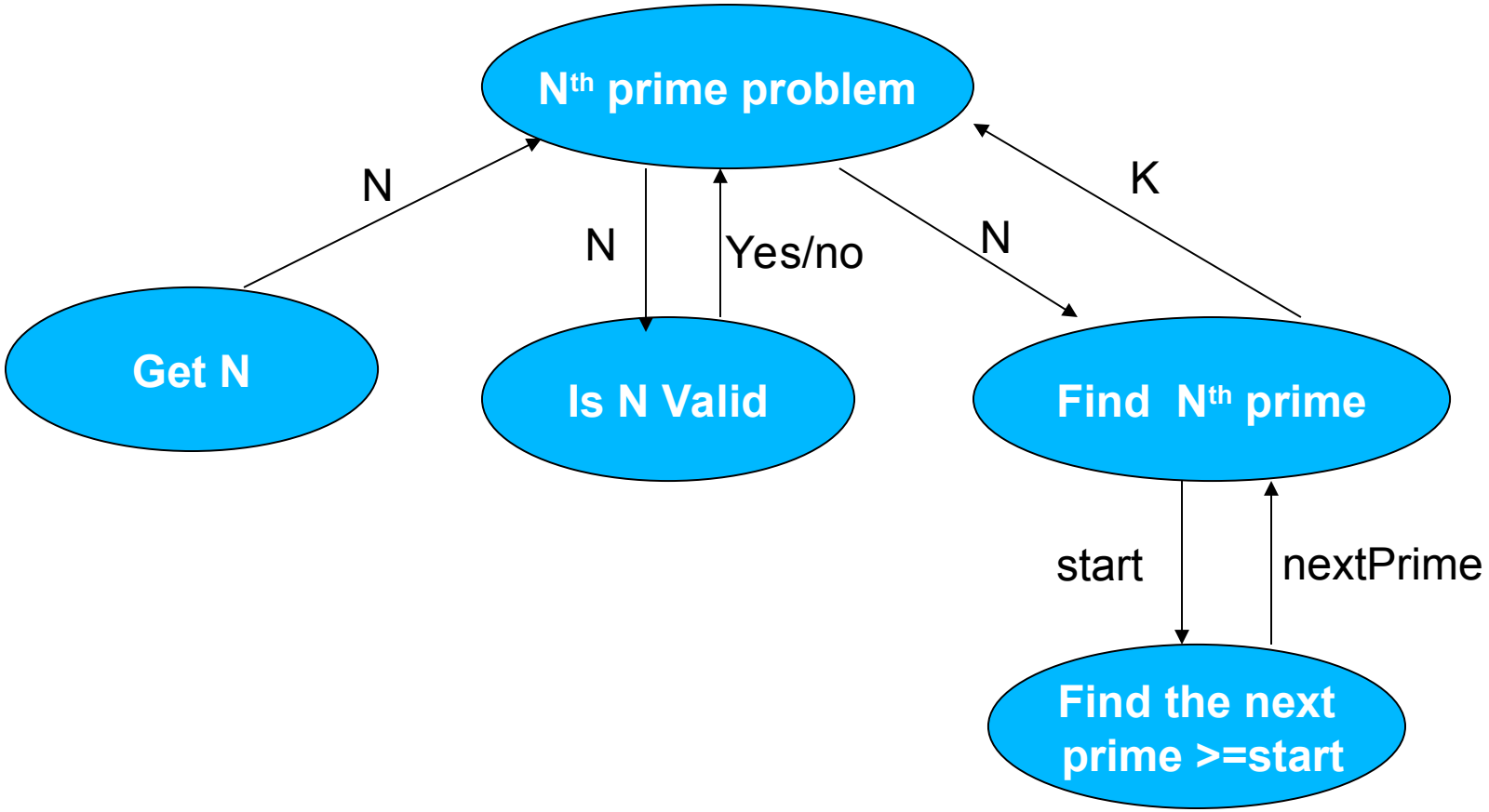
- Consider the example that we have seen on chapter two that computes the N^{th} prime number
- As you can see the program, it is to long, complex to understand easily
- Hence, modularization is the main technique to make it more simple to solve and understandable
- Here is the top down design

Modularization of N^{th} prime problem

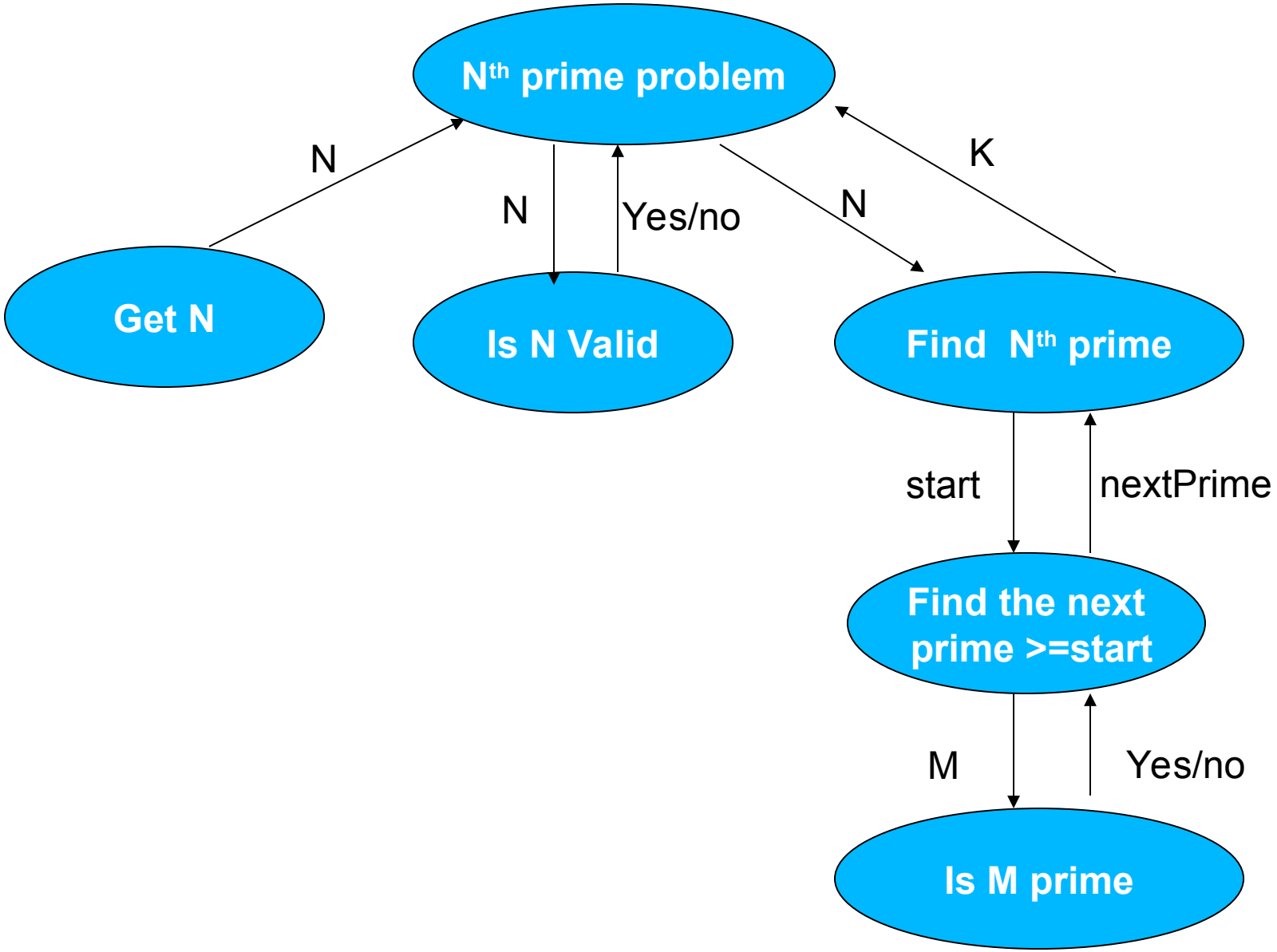
Modularization of Nth prime problem



Modularization of Nth prime problem



Modularization of Nth prime problem



Wait

Better to have idea about array now

Fortran Function

- Function is a mathematical term that maps elements taken from the domain into another element from the range

- Similarly the purpose of a function in Fortran is to take in a number of values or arguments, do some calculations with those arguments and then return a single result

- There are some functions which are written into FORTRAN and can be used without any special effort by the programmer

- They are called intrinsic functions

Fortran Function

- There are over 40 intrinsic functions in FORTRAN and they are mainly concerned with mathematical functions.

- The general way to activate a function is to use the function name in an expression.

- The function name is followed by a list of inputs, also called arguments, enclosed in parenthesis

- The form of a function call is
- ***answer = functionname (argumentl, argument2, . .)***

Fortran Function

Example

- PRINT*, ABS (T)
  - The compiler evaluates the absolute value of T and prints it out

- Y = SIN (X) + 45
  - The compiler calculates the value of sin x, adds 45 then puts the result into the variable Y, where x is in radians.

- M=MAX(a,b,c,d)
  - The compiler puts the maximum value of a, b, c and d into the variable M. If a=2, b=4, c=1 and d=7, then M would have a value of 7.

Fortran Function

Example

- C=SQRT (a* * 2 +b* * 2)
  - The compiler evaluates the expression, a**2+b**2, sends that value to the SQRT function, and places the answer in the variable C

- Fortran also allow programmers to define their own function

- Such functions are called External functions

- Hence, intrinsic and external functions are the two types of functions in fortran

Fortran Intrinsic functions

- Intrinsic functions are function which are built inside Fortran compiler.

- Fortran intrinsic functions can be classified into **3** as
  - ***Elemental Intrinsic function***
  - ***Inquiry intrinsic functions***
  - ***Transformation intrinsic functions***

Fortran Elemental Intrinsic functions

- These are functions specified for scalar arguments but may also be applied to array arguments
- If the argument of elemental function is scalar, then the result is scalar
- If the argument of elemental function is a complex structure of well defined shape, then the result is will also have the same shape (structure)
- If the function has multiple input argument, each argument must have the same shape
- Some of the elemental function ABS, SIN, COS, TAN, EXP, LOG, LOG10, MOD, and SQRT

Fortran Inquiry Intrinsic functions

- Inquiry intrinsic functions are functions whose value depends on the properties of an object being investigated
- An inquiry function doesn't manipulate the elements of the argument
- Rather an inquiry function manipulates the properties of the argument such as dimension, extent, shape and related properties
- It is important to determine the properties of an argument such as size, shape, extent, and legal subscript range of an array

Fortran Inquiry Intrinsic functions

- **Some inquiry intrinsic function**
  - **LBOUND**(Array, DIM) return the lower bound of the array at the specified dimension
  - **LBOUND**(Array) return the lower bounds of the array for each dimension
  - **UBOUND**(Array, DIM) return the upper bound of the array at the specified dimension
  - **UBOUND**(Array) return the upper bounds of the array for each dimension
  - **SHAPE**(Array) return the shape of the array
  - **SIZE**(Array, DIM) return the extent of the array along the given DIM
  - **SIZE**(Array) return the extent of each dimension of the array

Fortran Transformational Intrinsic functions

- Transformational intrinsic function are functions that have one or more array-valued arguments or an array valued result

- Unlike the elemental functions, which operate on element by element basis, these functions operate on array as a whole

- The output of transformational functions will often not have the same shape as the input argument

Fortran Transformational Intrinsic functions

- ***Some of the transformational functions are***
  - DOT_PRODUCT(vector1, vector2) return a scalar value
  - MATMUL(mat1, mat2) returns the product of the two matrices
  - RESHAPE(Array, Shape) reshape the dimension of the array value into the dimension specified as a shape

Some more intrinsic function

Function	Description
abs(x)	*absolute value of x*
acos(x)	*arccosine of x*
asin(x)	*arcsine of x*
atan(x)	*arctangent of x*
cos(x)	*cosine of x*
cosh(x)	*hyperbolic cosine of x*
dble(x)	*converts x to double precision type*
exp(x)	*exponential function of x (base e)*
log(x)	*natural logarithm of x (base e)*
mod(n,m)	*remainder when n is divided by m*
real(x)	*converts x to real (single precision) type*
sign(x,y)	*changes the sign of x to that of y*
sin(x)	*sine of x sinh(x) hyperbolic sine of x*
sqrt(x)	*square root of x tan(x) tangent of x*
tanh(x)	*hyperbolic tangent of x*

External Functions

- The intrinsic functions in FORTRAN are useful but there will be a time when there is no intrinsic function to meet your needs.
- When this occurs you may write your own function subprogram.
- The General form of a function definition

```
Dtype FUNCTION functionName(formalParameterlist)
    IMPLICIT NONE
    [specification part]
    [execution part]
END FUNCTION function-name
```

External Functions

Dtype **FUNCTION** functionName(formalParameterlist)

- This line indicates the function return type, the name of the function and the formal argument list
- The data type is one the valid data type in FORTRAN
- The function name is a valid identifier which must be unique
- The formal argument list are list of variables separated by comma
- Every formal parameter variable must be declared in the specification part of the program

External Functions

IMPLICIT NONE

- This line indicates that the function doesn’t allow implicit way of variable declaration.
- This is optional but recommended

External Functions

[specification part]

- This line is used in the same way as we did for the main program

[execution part]

- This is the set of executable instruction that the function uses to convert the values in the formal argument into a return value
- Function return a value by its function name
- The last statement in any function is RETURN which tells the function to return the current value of the function to the caller

END FUNCTION function-name

- This line indicates the end of the function

External Functions

- The following function takes three real valued arguments and return the average of the result to the caller

```
REAL FUNCTION AVERAGE(X,Y,Z)
    REAL X,Y,Z !declaring formal arguments as local variable
    REAL SUM !declaring local variable
    SUM = X + Y + Z
    AVRAGE = SUM /3.0 !assignment to a function name
    RETURN !this indicate end of the function
END FUNCTION AVERAGE
```

External Functions

- Lets consider the main program that uses the above function

- Of course, the above function can be used by any construct like another function, or procedure

External Functions

```
PROGRAM FUNDEM
      !Declarations for main program
      REAL A,B,C
      REAL AV, AVSQ1, AVSQ2
      REAL AVERAGE !function declaration
      ! Enter the data
      DATA A,B,C/5.0,2.0,3.0/
      !Calculate the average of the numbers
      AV = AVRAGE(A,B,C)
      AVSQ1 = AVERAGE(A,B,C) **2 !The function sent three actual
      argument
      AVSQ2 = AVERAGE(A**2,B**2,C**2) !The function sent three
      actual argument
      PRINT *,'Statistical Analysis'
      PRINT *,'The average of the numbers is:',AV
      PRINT *,‘The square of the average of the numbers: ',AVSQl
      PRINT *,'The average of the squares is: ', AVSQ2
END
```

External Functions

- Note the following in the above program that uses the function called Average
- In the specification part, we have defined the identifier average
- In order to access external function, we have to declare the function name with its data type being the return type
- Then one can call to the external function just like intrinsic functions

External Functions

- While calling a function, the caller must provide the necessary information to the function formal arguments
- The set of values which the caller send must match in data type, number and order or the type
- The set of values that the caller send to a function are called actual arguments
- An Actual Argument can be expression evaluated to a value with the same type as the corresponding formal parameter

External Functions

- There are a few rules for writing external functions

1. Function subprograms and any other subprograms are placed after the END statement of the main program.

2. All variables that are used by the function, including the arguments, must be declared in the function right after the first line.

3. The function name should not be declared within the function specification.

External Functions

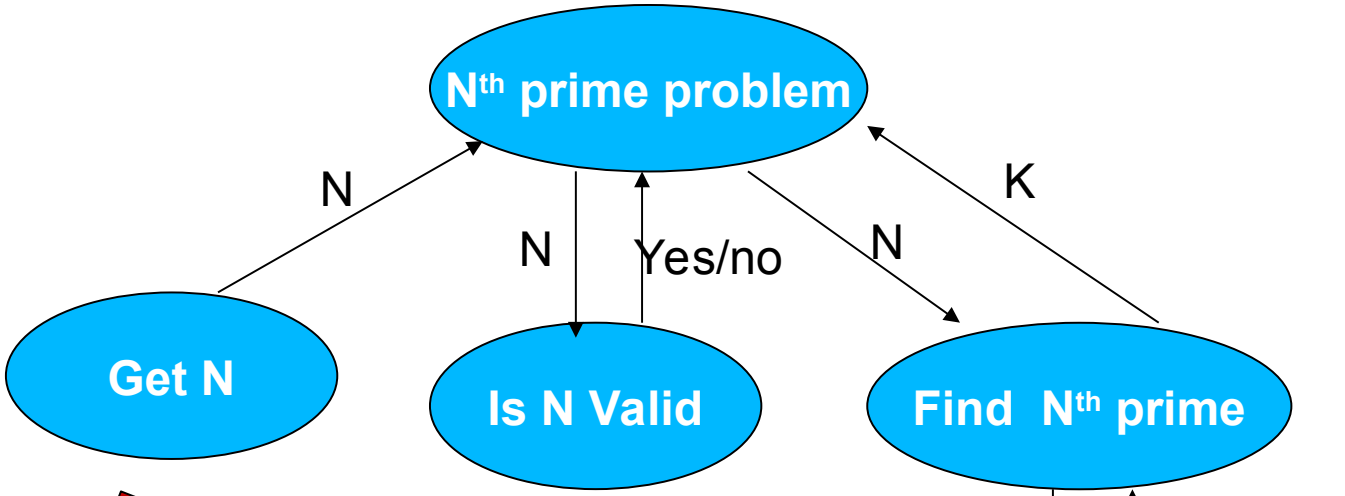
4. You must use the function name in an assignment statement within the function.

5. This is how the compiler knows which value to pass back to the main program.

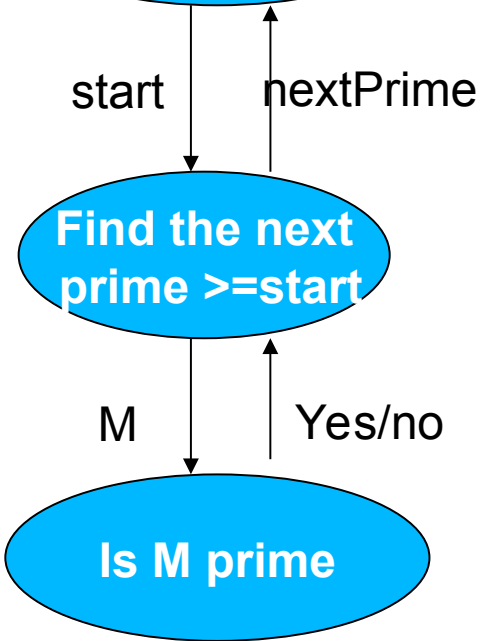
6. A function must finish with RETURN and END statements. ***(Return statement is optional in Fortran 90 and above)***

The Nth prime implementation

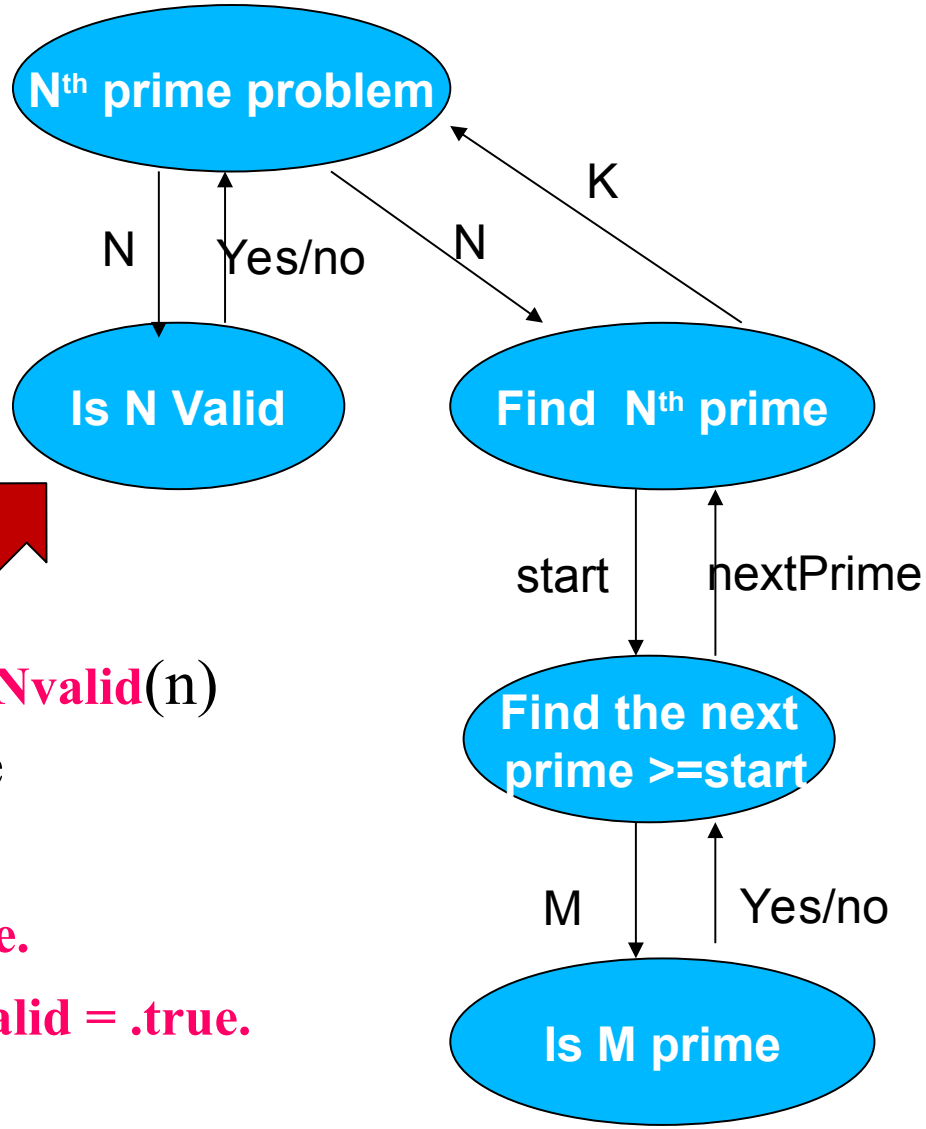
We have already seen the solution without modularization




```
integer function getN()
    implicit none
    integer n
        read *, n
        getn = n
        return
end function getN
```



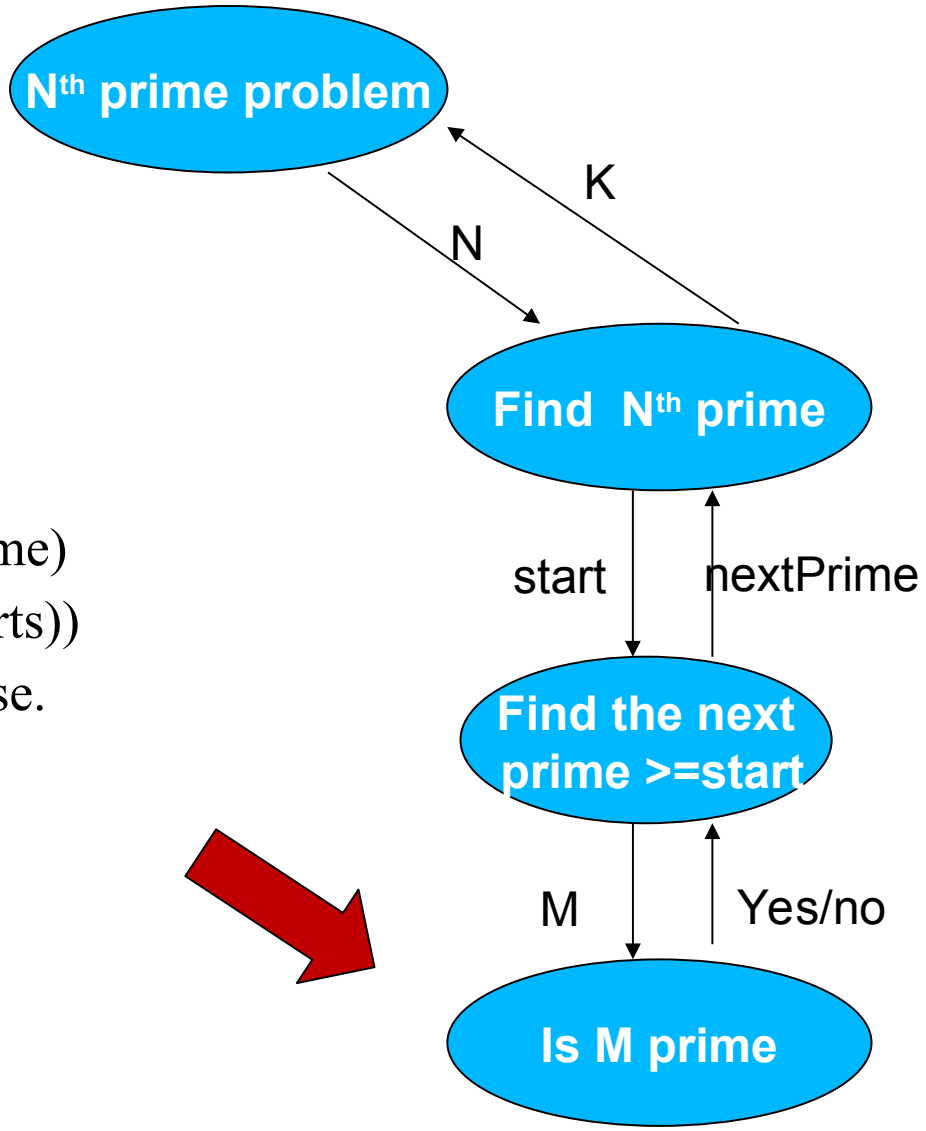
The Nth prime implementation




```
logical function isNvalid(n)
        implicit none
        integer n
        isnvalid = .false.
        if(n > 1) isnvalid = .true.
        return
end function isNValid
```

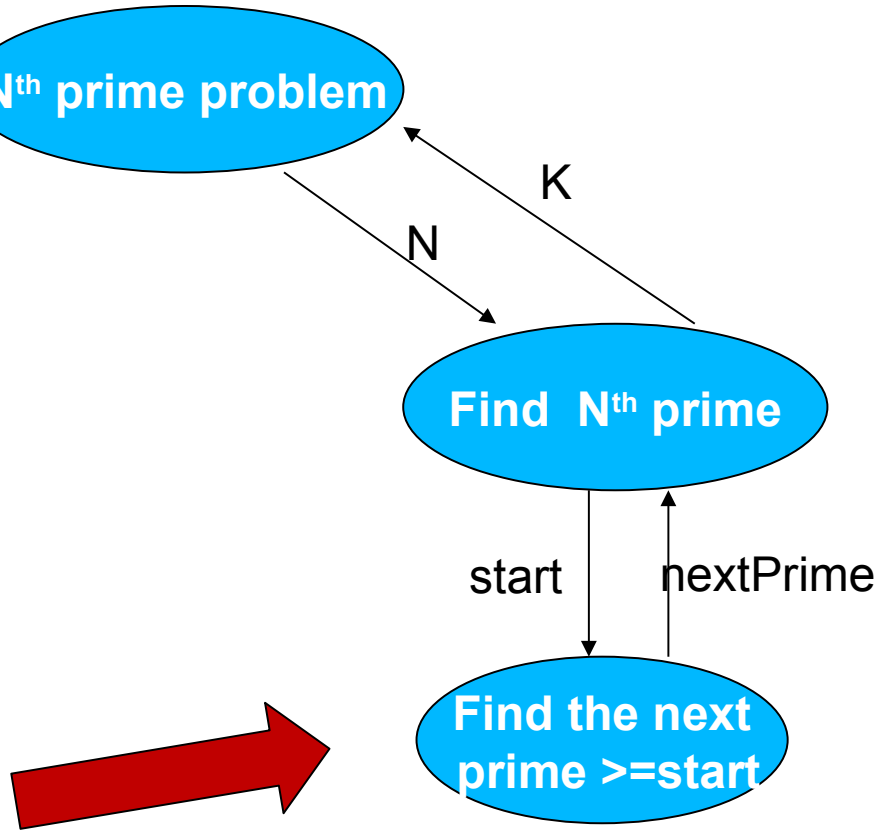
The Nth prime implementation

```
logical function isMprime(m)
        implicit none
        integer m, starts
        logical isPrime
        if(m == 2)
                    isMprime = .true.
                    return
        end if
        isprime = .true.; starts = 2
        do while(starts < m .and. isPrime)
            mod = m - (starts * (m / starts))
            if(mod == 0) isPrime = .false.
            starts = starts + 1
        end do
        isMprime = isPrime
        return
end function isMPrime
```



The Nth prime implementation

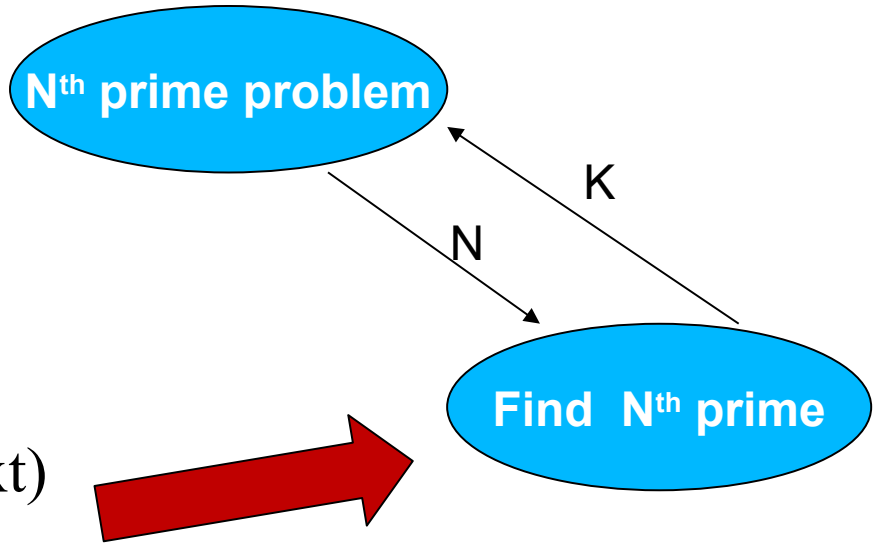
```
integer function nextPrime(start)
        implicit none
        integer m, start
        logical isMprime, isPrime
        isprime = .false.;
        do while(.not. isPrime)
            isPrime = isMprime(start)
            start = start + 1
        end do
        nextPrime = start – 1
        return
end function nextPrime
```



The Nth prime implementation

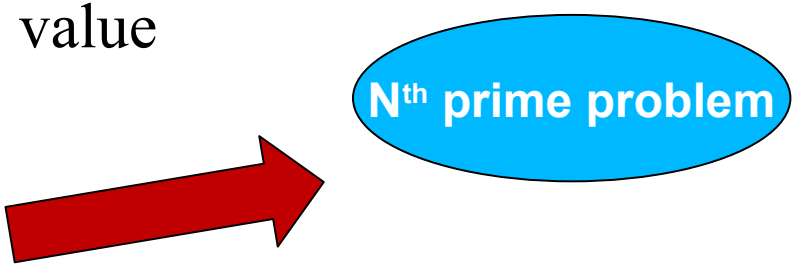
```
integer function findNthPrime(n)
        implicit none
        integer n, m, nextPrime, next
        m = 0; next =2
        do while(m < n)

                next = nextPrime(next)
                m = m +1
        end do
        findNthPrime = next
        return
End function findNthPrime
```



The Nth prime implementation

```
program NthPrime
        IMPLICIT NONE
        integer n, getN, findNthPrime, value
        logical isNValid, isValid
        n = getN()
        isValid = isNvalid(n)
        if(.not. isValid)
            print *, “The input value of n is not valid”
            stop
        end if
        value = findNthPrime(n)
        print  *, “The “, n, “th prime number is “, value
        stop
End program
```



Fortran Subroutines

- A Function receives some input via its formal arguments from outside world and computes and returns one value, the function value, with the function name

- In some cases, you do not want to return any value or you may want to return more than one values

- Thus, Fortran's subroutines may be what you need

- Functions and subroutines are referred to as subprograms.

Fortran Subroutines

- A subroutine is a self-contained unit that
  - receives some "input" from the outside world via its formal arguments,
  - does some computations, and then
  - returns the results, if any, with its formal arguments.

Fortran Subroutines

- The syntax of a Fortran subroutine is:

```
SUBROUTINE subroutine-name (formalArgumentList)
    IMPLICIT NONE
    [specification part]
    [execution part]
END SUBROUTINE subroutine-name
```

Fortran Subroutines

- The first line of a subroutine starts with the keyword SUBROUTINE, followed by that subroutine's name.
- Unlike functions, the name of a subroutine is not a special name to which you can save a result
- Subroutine's name is simply a name for identification purpose and you cannot use it in any statement except the CALL statement.
- Following subroutine-name, there is a pair of parenthesis in which a number of arguments arg1, arg2, ..., arg_n are listed separated with commas
- These arguments are referred to as formal arguments

Fortran Subroutines

- If there is no argument in the parenthesis, the parenthesis is optional
  - In this case the subprogram is said to be totally self-contained
  - it generates and/or prints information and it neither receives nor gives information to the calling program.

- The formal argument list, enclosed in parentheses after the subroutine name, contains only the variable names (separated by commas) and cannot be expressions and constants

Fortran Subroutines

- All variables stated are local and should be (normally) declared explicitly within the subroutine, particularly when coding subroutines "externally"

Fortran Subroutines

Example

- The following is a subroutine definition called Factorial.
  SUBROUTINE Factorial(n, Answer)
  - It has two formal arguments n and Answer.

- The following is a subroutine definition called TestSomething
  SUBROUTINE TestSomething(a, b, c, Error)
  - It takes four formal arguments a, b, c, and Error.

Fortran Subroutines

- Between SUBROUTINE and END SUBROUTINE, there are
  - IMPLICIT NONE,
  - specification part, and
  - execution part
- These are exactly identical to that of a PROGRAM
- Any statements that can be used in a PROGRAM can also be used in a SUBROUTINE.
- A subroutine last execution statement can have optional RETURN (Don’t use STOP)

Designing Subroutines

- A subroutine must end with END SUBROUTINE, followed by its name.

- When the control of execution reaches END SUBROUTINE, the values stored in some formal arguments are passed back to their corresponding actual arguments

Designing Subroutines

- Example

Subroutine menu

*print *, "A: to sort in ascending order"*

*print *, "Z: to sort in descending order"*

*print *, "E: to enter list of numbers"*

*print *, "V: to display the list of numbers"*

End subroutine menu

Fortran Subroutines

- One can call a subroutine from
  - The main program or
  - Any other subprogram)
- The syntax to call a subroutine is

  CALL subroutinename !if there is no need to give argument

  OR

  CALL subroutinename() !if there is no need to give argument

  OR

  CALL subroutinename(formalArgumentList)

Fortran Subroutines

- The actual arguments and the formal arguments must match in
  - number,
  - order, and
  - type
- Switching the order of the arguments, neglecting to declare the data type of any variables used as arguments, or omitting arguments can lead to errors ***(except when the arguments are optional)***

Fortran Subroutines - Exercise

- Consider a problem that request the user to enter three number then the system requests to what to do with the numbers by showing a list of menu options
- The menu options are add the number, find the product of the number, and square each number and return their squares
- Try to do the following based on this

  Top Down design

  - Identify weather to use sub routine or function for each component
  - Implement the program according to the modularization result

Argument Passing In Fortran

- Generally in programming language there are four types of passing argument between calling function and a subprogram
- The standard names for these methods are:

1. Call by value

   –Value of the actual parameter is copied into the formal argument variables

2. call by reference (or "address")

   –Address of the actual parameter is copied into the value of the formal argument variables

3. call by name (or "expression")

   –Address of the actual parameter is shared with the formal argument variables

4. call by result (or "value-result")

   –Value of the formal argument will be copied into the actual argument when the program unit called complete its operation

Argument Passing In Fortran

- In some languages (e.g., C++, Ada, Pascal), the programmer has the choice of determining the passing scheme for each argument.
- In other languages (e.g., Fortran, C), the user has no choice.
- The passing scheme is predetermined, but sometimes may be different for arrays than for scalars
- In Fortran, the specific passing methods are predetermined by the implementation, and no choice is possible by the user.
- In Fortran, the actual arguments are always changed in the calling segment, if the formal arguments were changed in the subprogram segment.

Argument Passing In Fortran

- But the methods used to produce the change in value may vary from implementation to implementation.

- Most implementations of FORTRAN-IV and some implementations of Fortran-90/95/2003 use
  - call by result (value-result) for simple (scalar) variables

  and
  - call by reference (address) for arrays

- Most implementations of FORTRAN-77 and other implementations of Fortran-90/95/2993 use
  - call by reference for *all* arguments (scalars or arrays).

Argument Passing In Fortran

- In **call by result (value-result)**, the subprogram *copies* **values** from the actual arguments in the calling program

- But the subprogram works with values in *independent memory locations* and the argument values may be changed independently of the variables originally associated with them in the calling program

- When the subprogram is finished (i.e. when the RETURN or END statement is reached), the associated variables in the calling program are *changed* to correspond to the values the arguments received in the subprogram.

Argument Passing In Fortran

- In **call by reference (address)**, no new memory locations are allocated.
- Instead, a link is set up so that whenever a subprogram variable is referenced, the corresponding variable in the calling program is accessed and changed.
- This scheme may result in one memory locations having two different "names" [one used by the main program and one by the subprogram], a phenomenon often termed "aliasing."

Argument Passing In Fortran

- Note that in both of these calling conventions, the variables can be changed upon completion of the subprogram.
- This is not true of the call-by-value arguments (used in C, C++, or Pascal).
- In **call by value,** the subprogram sets up new independent memory locations, copies the values from the actual arguments (as in call-by-value-result), but does **not** change any variables in the calling program upon its completion.

Argument Passing In Fortran

- To show that problems may occur if a programmer is unclear as to which argument passing scheme is used, and thus, to what is happening exactly between the main program and the subroutine,
- let us examine the following example, written in a *pseudo-language*.

```
function icrazy(i,j,k):integer
    i=j+k
    j=j+k
    icrazy=i+j+k
    k=j
    return
end
```

Argument Passing In Fortran

- Suppose the main program code included the following:

```
n = 5
ivalue = icrazy(n,n,n)
ivalue = ivalue + n
write (ivalue)
```

- Then, depending on the type of "calling" method used, one gets three different answers!!
- **call by value-result** (FORTRAN-IV scalars)
  = 35
- **call by value** (C, C++ [without the &], Pascal [without the **var**)
  = 30
- **call by address** (Fortran arrays, FORTRAN-77 scalars, Pascal **var** arguments and C++ [WITH the &])
  = 80

Argument Passing In Fortran

- In the *call by value* case, the n in the calling code remains unchanged, whereas using *call by value-result* it changes.
- In the *call by address* case, the one memory location identified by n in the main program has three additional names in the function, i, j, and k.
- Therefore, whenever any one of them changes, the other three change values as well!
- Since there is no uniformity as to how Fortran-90/95/2003 compilers actually pass scalar arguments, one should either test the argument passing scheme via some short program or avoid passing the same actual argument to different dummy arguments.

Argument "Intent"

- Fortran-90/95/2003 has introduced the ability to designate the "intent" of subprogram arguments.

- Explicitly specifying the intent of arguments very helpful in FORTRAN programming
  - Firstly it aids in documentation and readability of the code
  - in some cases, it prevent unintentional reassignment of argument values.
- The three options for the "intent" of an argument are:
  - **in** -- used for data that only comes *into* a subprogram;
  - **out** -- used for data that only goes *out* from a subprogram;
  - **in out** -- used for data flowing in *either direction*.

Argument "Intent"

- One can add the information about intent as an attribute when declaring an argument within the subprogram.

For example,

```
REAL, INTENT(IN) :: X, Y
```

- If X and Y have been declared as in this example, any attempt to change the value of X or Y within the subprogram, would result in an error message.

- Such a warning would occur since any change in an argument within a subprogram also affects the corresponding actual argument in the calling segment.

Argument "Intent"

- An argument with intent OUT cannot have any input value.

  REAL, INTENT(OUT) :: Z

- It receives a value within the subprogram and conveys the new value back to the calling program.
- The actual argument that corresponds to the format argument with intent(out) declaration must be a variable
- Look at the following example which has both intent specification

Argument "Intent"

```
real function mySqrt(input, isValid)
    real, intent(in) :: input
    logical, intent(out) :: isValid
     isValid = .false.
    if(input >= 0) isValid = .true.
    mySqrt = sqrt(input)
end function foo
```

SAVE Statement

- Normally when a subprogram is completed, the values of all variables are left undefined.
- If one wishes to re-use a function or subroutine several times and also wishes the previous local variables to retain old values, this can be done by means of invoking the SAVE specification statement.
- The SAVE statement comes before any DATA or executable statement and lists those variables whose values are to be saved.

SAVE Statement

- If no variables are listed, *all* variable values are saved.

example,

- if one wishes to save the values of variables A, B, and C, the following statement is used:
  - SAVE A, B, C
- if one wishes to save the values of all the variables the following statement is used:
  - SAVE