A Function That Doesn't Return a Value

So, suppose we wanted, for example, to read in a given number of values into an array. We might define a function like so:

```
INTEGER FUNCTION input_elements (                &
&                  element, number_of_elements, &
&                  minimum_number_of_elements,  &
&                  maximum_number_of_elements)
  IMPLICIT NONE
  INTEGER :: number_of_elements
  INTEGER :: minimum_number_of_elements, &
&            maximum_number_of_elements
  REAL,DIMENSION(minimum_number_of_elements:    &
&                maximum_number_of_elements) :: &
&   element
  INTEGER :: numelts, e
  PRINT *, "What are the ", number_of_elements, &
&          " elements of the array?"
  READ *, (element(e),                        &
&          e = minimum_number_of_elements, &
&              maximum_number_of_elements)
  input_elements = 000
END FUNCTION input_elements
```

What on **earth** are we going to return?

The best answer is, we're not going to return anything.

But if we're not returning anything, it's not a function.

So, we have a special construct in case we don't want to return anything: a *subroutine*.

Subroutines

A *subroutine* is exactly like a function, except that:

1. The keyword FUNCTION is replaced by the keyword SUBROUTINE. So, the subroutine has a SUBROUTINE statement and an END SUBROUTINE statement, which are analogous to a function's FUNCTION statement and END FUNCTION statement.
2. A subroutine has no return type and no return variable.

```
SUBROUTINE input_elements (                &
&            element, number_of_elements, &
&            minimum_number_of_elements,  &
&            maximum_number_of_elements)
  IMPLICIT NONE
  INTEGER :: number_of_elements
  INTEGER :: minimum_number_of_elements, &
&            maximum_number_of_elements
  REAL,DIMENSION(minimum_number_of_elements:    &
&                maximum_number_of_elements) :: &
&   element
  INTEGER :: numelts, e
  PRINT *, "What are the ", number_of_elements, &
&          " elements of the array?"
  READ *, (element(e),                        &
&          e = minimum_number_of_elements, &
&              maximum_number_of_elements)
END SUBROUTINE input_elements
```

A subroutine is invoked using a CALL statement:

```
  CALL input_elements(element,numelts, &
&        minnumelements,maxnumelements)
```

Notice that a subroutine **has** to have side effects to be useful.

The general term for functions and subroutines is *procedures*. That is, a procedure is either a function or a subroutine.

Subroutine Example

```
% cat inputarrayprog.f90
  PROGRAM inputarrayprog
    IMPLICIT NONE
    INTEGER,PARAMETER :: first_element  =   1
    INTEGER,PARAMETER :: minnumelements =   1
    INTEGER,PARAMETER :: maxnumelements = 100
    REAL,DIMENSION(first_element:maxnumelements) :: element
    INTEGER :: number_of_elements
    INTEGER :: e
    INTEGER :: input_number_of_elements
    number_of_elements =        &
  &   input_number_of_elements( &
  &     minnumelements, maxnumelements)
    PRINT '(A,I3,A)',                                  &
  &   "The number of elements you plan to input is ", &
  &   number_of_elements, "."
    CALL input_elements(element, number_of_elements, &
  &       minnumelements, maxnumelements)
    PRINT *, "The elements of the array are"
    PRINT *, (element(e), e = 1, number_of_elements)
  END PROGRAM inputarrayprog

  INCLUDE 'inputarrayelts.f90'
% cat inputarrayelts.f90
  SUBROUTINE input_elements (              &
  &            element, number_of_elements, &
  &            minimum_number_of_elements,  &
  &            maximum_number_of_elements)
    IMPLICIT NONE
    INTEGER :: number_of_elements
    INTEGER :: minimum_number_of_elements, &
  &            maximum_number_of_elements
    REAL,DIMENSION(minimum_number_of_elements:    &
  &               maximum_number_of_elements) :: &
  &   element
    INTEGER :: numelts, e
    PRINT *, "What are the ", number_of_elements, &
  &          " elements of the array?"
    READ *, (element(e),                        &
  &          e = minimum_number_of_elements, &
  &              number_of_elements)
  END SUBROUTINE input_elements

  INCLUDE "inputnumelts.f90"
% f90 -o inputarrayprog inputarrayprog.f90
% inputarrayprog
 How many elements would you like
   the array to have
   (between            1  and          100 )?
 7
The number of elements you plan to input is   7.
 What are the            7  elements of the array?
 -5, -3, -1, 0, 2, 4, 6
 The elements of the array are
  -5.000000      -3.000000      -1.000000      0.0000000E+00
   2.000000       4.000000       6.000000
```

Why Do We Like Code Reuse?

1. Bug avoidance: Since we don't have to retype the procedure from scratch every time we use it, we aren't constantly making new and exciting typos.
2. Implementation efficiency: We aren't wasting valuable programming time ($8 - $100s per programmer per hour) on writing commonly used procedures from scratch.
3. Verification: We can test a procedure under every conceivable case, so that we're confident that it works, and then we don't have to worry about whether the procedure has bugs when we use it in a new program.

Why Do We Like User-Defined Procedures?

1. Code Reuse (see above)
2. *Encapsulation*: We can write a procedure that packages an important concept (e.g., the cube root). That way, we don't have to litter our program with cube root calculations. So someone reading our program will be able to tell immediately that, for example, a particular statement has a cube root in it, rather than constantly having to figure out what
   `x ** (1.0 / 3.00)`
   means.
3. *Modular* Programming: If we make a bunch of encapsulations, we can have our main program simply call a bunch of procedures. That way, it's easy for someone reading our code to tell what's going on in the main program, and then to look at individual procedures to see how they work.

Argument Intent

In a procedure, some arguments may be intended to be incoming only; that is, the value of the argument will not be changed as a side effect of the procedure.

On the other hand, some arguments may not have a value when the procedure is called, but instead will be assigned a value as a side effect of the procedure.

Still other arguments may have a value going in, but then that value is changed as a side effect of the procedure.

We refer to this concept as the *intent* of the argument.

If someone is reading your program (including you after leaving it alone for a month), it might help them if you have a way of indicating what the intent of each argument is.

So, Fortran 90 provides an `INTENT` attribute that you can attach to each argument, whose values can be `IN`, `OUT` or `INOUT`:

```
SUBROUTINE input_elements (                &
&            element, number_of_elements, &
&            minimum_number_of_elements,  &
&            maximum_number_of_elements)
  IMPLICIT NONE
  INTEGER,INTENT(IN) :: number_of_elements
  INTEGER,INTENT(IN) :: &
&   minimum_number_of_elements, &
&   maximum_number_of_elements
  REAL, &
&   DIMENSION(minimum_number_of_elements:  &
&              maximum_number_of_elements), &
&   INTENT(OUT) :: element
...
END SUBROUTINE input_elements
```

INTENT Attribute Reduces Bugs

We can reduce some bugs by using the INTENT attribute, since it prevents us from accidentally doing something contrary to the stated intent. It's a way for the compiler to idiotproof our source code.

```
% cat inputarrayeltsintbad.f90
SUBROUTINE input_elements (                &
&            element, number_of_elements, &
&            minimum_number_of_elements,  &
&            maximum_number_of_elements)
  IMPLICIT NONE
  INTEGER,INTENT(IN) :: number_of_elements
  INTEGER,INTENT(IN) :: minimum_number_of_elements, &
&                       maximum_number_of_elements
  REAL,DIMENSION(minimum_number_of_elements:  &
&                maximum_number_of_elements), &
&      INTENT(OUT) :: element
  INTEGER :: numelts, e
  PRINT *, "What are the ", number_of_elements, &
&          " elements of the array?"
  READ *, (element(e),                        &
&          e = minimum_number_of_elements, &
&              number_of_elements)
  number_of_elements = number_of_elements + 1
END SUBROUTINE input_elements

INCLUDE "inputnumeltsint.f90"
% f90 -c inputarrayeltsintbad.f90
f90: Error: inputarrayeltsintbad.f90, line 18:
    There is an assignment to a dummy symbol
      with the explicit INTENT(IN) attribute
        [NUMBER_OF_ELEMENTS]
    number_of_elements = number_of_elements + 1
----^
```

Actual & Formal Argument Names

In Fortran 90, and in most programming languages, if you are using a variable as an actual argument to a procedure, the name you use in the actual argument doesn't have to match the name of the formal argument. For example:

```
% cat argmismatch1.f90
  PROGRAM argmismatch1
    IMPLICIT NONE
    INTEGER :: bobby = 1, betty = 2
    PRINT *, "In PROGRAM argmismatch1:"
    PRINT *, "  bobby = ", bobby
    PRINT *, "  betty = ", betty
    CALL print_both(bobby, betty)
  END PROGRAM argmismatch1

  SUBROUTINE print_both (thing1, thing2)
    IMPLICIT NONE
    INTEGER,INTENT(IN) :: thing1, thing2
    PRINT *, "In SUBROUTINE print_both:"
    PRINT *, "  thing1 = ", thing1
    PRINT *, "  thing2 = ", thing2
  END SUBROUTINE print_both
% f90 -o argmismatch1 argmismatch1.f90
% argmismatch1
 In PROGRAM argmismatch1:
   bobby =            1
   betty =            2
 In SUBROUTINE print_both:
   thing1 =            1
   thing2 =            2
```

Actual & Formal Argument Name Confusion

In fact, because there are really no constraints on the variable names used as actual arguments, nor on the formal argument names, it's perfectly possible to use the same names to mean different things. For example:

```
% cat argmismatch2.f90
  PROGRAM argmismatch2
    IMPLICIT NONE
    INTEGER :: bobby = 1, betty = 2
    PRINT *, "In PROGRAM argmismatch2:"
    PRINT *, "  bobby = ", bobby
    PRINT *, "  betty = ", betty
    CALL print_both(bobby, betty)
  END PROGRAM argmismatch2

  SUBROUTINE print_both (betty, bobby)
    IMPLICIT NONE
    INTEGER,INTENT(IN) :: betty, bobby
    PRINT *, "In SUBROUTINE print_both:"
    PRINT *, "  bobby = ", bobby
    PRINT *, "  betty = ", betty
  END SUBROUTINE print_both
% f90 -o argmismatch2 argmismatch2.f90
% argmismatch2
 In PROGRAM argmismatch2:
   bobby =            1
   betty =            2
 In SUBROUTINE print_both:
   bobby =            2
   betty =            1
```

This is **NOT** a good idea, but people do it anyway. **You should not.**

How Is This Confusion Possible?

In programming languages, we are very concerned with *context*.

The program below has two contexts: the program unit named `argmismatch2` and the subroutine unit named `print_both`.

In general, a variable or argument name is only valid within the context in which it is declared. (There are exceptions to this, but we won't concern ourselves with them now.)

So, in the context of the program unit `argmismatch2`, the variable `bobby` is initialized to 1, and the variable `betty` is initialized to 2.

When we call the subroutine named `print_both`. the first formal argument, named `betty`, corresponds to an actual argument with the value 1, and the second formal argument, named `bobby`, corresponds to an actual argument with the value 2.

```
% cat argmismatch2.f90
  PROGRAM argmismatch2
    IMPLICIT NONE
    INTEGER :: bobby = 1, betty = 2
    PRINT *, "In PROGRAM argmismatch2:"
    PRINT *, "  bobby = ", bobby
    PRINT *, "  betty = ", betty
    CALL print_both(bobby, betty)
  END PROGRAM argmismatch2

  SUBROUTINE print_both (betty, bobby)
    IMPLICIT NONE
    INTEGER,INTENT(IN) :: betty, bobby
    PRINT *, "In SUBROUTINE print_both:"
    PRINT *, "  bobby = ", bobby
    PRINT *, "  betty = ", betty
  END SUBROUTINE print_both
% f90 -o argmismatch2 argmismatch2.f90
% argmismatch2
 In PROGRAM argmismatch2:
   bobby =            1
   betty =            2
 In SUBROUTINE print_both:
   bobby =            2
   betty =            1
```

Confusinger and Confusinger

This problem can get arbitrarily bad:

```
% cat argmismatch3.f90
  PROGRAM argmismatch3
    IMPLICIT NONE
    INTEGER :: bobby = 1, betty = 2
    PRINT *, "In PROGRAM argmismatch2:"
    PRINT *, "  bobby = ", bobby
    PRINT *, "  betty = ", betty
    CALL print_both(bobby, betty)
  END PROGRAM argmismatch3

  SUBROUTINE print_both (betty, bobby)
    IMPLICIT NONE
    INTEGER,INTENT(IN) :: betty, bobby
    REAL :: tweed = 1.5, wool = -0.5
    PRINT *, "In SUBROUTINE print_both:"
    PRINT *, "  bobby = ", bobby
    PRINT *, "  betty = ", betty
    CALL print_others(tweed, wool)
  END SUBROUTINE print_both

  SUBROUTINE print_others (bobby, betty)
    IMPLICIT NONE
    REAL,INTENT(IN) :: betty, bobby
    PRINT *, "In SUBROUTINE print_others:"
    PRINT *, "  bobby = ", bobby
    PRINT *, "  betty = ", betty
  END SUBROUTINE print_others
% f90 -o argmismatch3 argmismatch3.f90
% argmismatch3
 In PROGRAM argmismatch2:
   bobby =            1
   betty =            2
 In SUBROUTINE print_both:
   bobby =            2
   betty =            1
 In SUBROUTINE print_others:
   bobby =    1.500000
   betty =  -0.5000000
```

Scope of Variables & Named Constants

The *scope* of a variable, argument or named constant is the set of units (e.g., program unit, function units) in which it can be accessed.

For example, in the function `input_number_of_elements`, the scope of the variable `numelts` is the function only; the variable `numelts` cannot be accessed by the program `userarray`.

This scope is referred to as *local*.

```
INTEGER FUNCTION input_number_of_elements (    &
&                  minimum_number_of_elements, &
&                  maximum_number_of_elements)
  IMPLICIT NONE
  INTEGER :: minimum_number_of_elements, &
&            maximum_number_of_elements
  INTEGER :: numelts
  DO
    PRINT *, "How many elements would you like"
    PRINT *, "  the array to have"
    PRINT *, "  (between ",                        &
&            minimum_number_of_elements, " and ", &
&            maximum_number_of_elements, ")?"
    READ *, numelts
    IF (numelts <      &
&       minimum_number_of_elements) THEN
      PRINT *, "Too few, idiot!"
    ELSE IF (numelts > &
&            maximum_number_of_elements) THEN
      PRINT *, "Too many, fool!"
    ELSE
      EXIT
    END IF
  END DO
  input_number_of_elements = numelts
END FUNCTION input_number_of_elements
```

Scope Example

```
% cat aggregates.f90
  PROGRAM aggregates
    IMPLICIT NONE
    INTEGER :: inval
    INTEGER,EXTERNAL :: sumthru, factorial
    PRINT *, "Enter a positive integer:"
    READ *, inval
    PRINT *, "The sum from 1 to ", inval * 3, &
  &          " is ", sumthru(inval * 3),   "."
    PRINT *, "The factorial of  ", inval * 2,  &
  &          " is ", factorial(inval * 2), "."
  END PROGRAM aggregates

  INCLUDE "sumthru.f90"
  INCLUDE "fact.f90"
% cat sumthru.f90
  INTEGER FUNCTION sumthru (n)
    IMPLICIT NONE
    INTEGER,INTENT(IN) :: n
    INTEGER :: i, aggregate = 0
    DO i = 1, n
      aggregate = aggregate + i
    END DO
    sumthru = aggregate
  END FUNCTION sumthru
% cat fact.f90
  INTEGER FUNCTION factorial (n)
    IMPLICIT NONE
    INTEGER,INTENT(IN) :: n
    INTEGER :: i, aggregate = 1
    DO i = 1, n
      aggregate = aggregate * i
    END DO
    factorial = aggregate
  END FUNCTION factorial
% f90 -o aggregates aggregates.f90
% aggregates
 Enter a positive integer:
 5
 The sum from 1 to           15  is          120 .
 The factorial of            10  is      3628800 .
```

Exercise: Rewriting Statistics Program

Rewrite the statistics program that we worked on in class by writing a function to input the number of elements, a subroutine to input the elements of the array, a function to calculate the mean, a function to calculate the standard deviation, and a subroutine to output the mean and standard deviation.

The body of the program must not have any numeric or logical literal constants; all constants must be declared using appropriate symbolic names.

Don’t worry about comments.