Mixed Type Numeric Expressions

In the CPU calculations must be performed between objects of the same type, so if an expression mixes type some objects must change type.

Default types have an implied ordering:

1. INTEGER -- lowest
2. REAL
3. DOUBLE PRECISION
4. COMPLEX -- highest
The result of an expression is always of the highest type. For example:

* INTEGER * REAL gives a REAL  (3 * 2.0 = 6.0)
* REAL * INTEGER gives a REAL  (3.0 * 2 = 6.0)
* DOUBLE PRECISION * REAL gives DOUBLE PRECISION
* COMPLEX * <any type> gives COMPLEX
* DOUBLE PRECISION * REAL * INTEGER gives DOUBLE PRECISION

The actual operator is unimportant.
Mixed Type Assignment

Problems often occur with mixed-type arithmetic. The rules for type conversion are given below.

- **INTEGER = REAL**
  
  the RHS is evaluated, truncated (all of the decimal places lopped off) and assigned to the LHS.

- **REAL = INTEGER**
  
  the RHS is promoted to be REAL and stored (approximately) in the LHS.

Example: program mixedassign.f90
Intrinsic Procedures

Fortran 90 has over 100 built-in or intrinsic procedures to perform common tasks efficiently. They fall into a number of classes:

- **Elemental**
  - Mathematical (SQRT, SIN, LOG, etc.)
  - Numeric (ABS, CEILING, SUM, etc.)
  - Character (INDEX, SCAN, TRIM, etc.)
  - Bit (IAND, IOR, ISHFT, etc.)

- **Inquiry** (ALLOCATED, SIZE, etc.)

- **Transformational** (REAL, TRANSPOSE, etc.)

- **Miscellaneous** or non-elemental subroutines (SYSTEM_CLOCK and DATE_AND_TIME)
Fortran 90 has extremely powerful, flexible and easy-to-use capabilities for output formatting.

* The default formatting may be sufficient on your computer for now, but sometimes roundoff error causes “ugly” looking real values.

* It’s not a malfunction of the computer’s hardware, but a fact of life of finite precision arithmetic on computers.

* Replace the asterisk denoting the default format with a custom format specification.

* Example: add_2_reals.f90
Edit Descriptors

The three most frequently used edit descriptors are:

* `f` (floating point) for printing of reals
  
  syntax: \( f_{w.d} \)
  
  \( w \) = total number of positions
  
  \( d \) = number of places after the decimal point

* the decimal point occupies a position, as does a minus sign

* `a` (alphanumeric) for character strings

* `i` (integer) for integer (can use \( i_{w.d} \) format, where the \( d \) will pad in front of the value with zeroes

Also the new line (/) and tab (t) edit descriptors.

Example: `format_examples.f90`
Subroutines and Functions
There are two types of procedures:

- **SUBROUTINE**: a parameterized named sequence of code which performs a specific task and can be invoked from other program units.
  - invoked with the `CALL` statement

- **FUNCTION**: same as a subroutine but returns a result in the function name.
  - invoked by placing the function name (and its associated arguments in an expression)
  - use when just one return value is needed.

**Example**: `sort3.f90` and `sort_3.f90`
This simple example illustrates one of the important uses of subroutines: To exhibit the overall structure of a program and put the details in another place.

Internal subroutines and functions are designated by the `contains` statement.

The `implicit none` in the host program applies to the internal subroutines. Also used in modules.

Can we go even further with this example?
Subroutines with Arguments

- We can pass values to a subroutine by placing them in parentheses after the name of the subroutine in the call statement.
- In the call to swap, n1 and n2 are called arguments.
- Although it may appear to be handy, internal procedures may not be nested.
- To make subroutine swap available to other program units, we would need to place it within a module.
Functions

* Just like a subroutine, but intended to return one value (or an array of values). Invoked just like an intrinsic function.

* The result of a function should be placed in a result variable using the `result` keyword at the end of the function statement.

* If the result keyword and variable are omitted, the function name is used as the return variable and must be declared in the function.

* Example: `series.f90`
Argument Association

- The variables \( a \) and \( b \) in subroutine swap are placeholders for the two numbers to be swapped. These are **dummy arguments** and must be declared in the subroutine. The variables \( n1 \) and \( n2 \) in the first call to swap are the **actual arguments**.

- If the value of a dummy argument changes, then so does the value of the actual argument (**pass-by-reference**).

- An actual argument that is a constant or an expression more complicated than a variable can only pass a value to the corresponding dummy argument. This is called **pass-by-value**.
It is bad programming practice to modify arguments in function calls.

In general, the number of actual and dummy arguments **must be the same**.

Also, the data type (and kind parameter) of each actual argument must match that of the corresponding dummy argument.

Keyword arguments and optional arguments: best explained by an example *(series2.f90)*