EUPHORIA - VARIABLES

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables. Euphoria data types are explained in different chapter.

These memory locations are called variables because their value can be changed during their life time. Next chapter would discuss Euphoria constants whose values can not be changed once assigned.

Variable Declaration

Euphoria variables have to be explicitly declared to reserve memory space. Thus declaration of a variable is mandatory before you assign a value to a variable.

Variable declarations have a type name followed by a list of the variables being declared. For example −

```
integer x, y, z

sequence a, b, x
```

When you declare a variable you name the variable and you define which sort of values may legally be assigned to the variable during execution of your program.

The simple act of declaring a variable does not assign any value to it. If you attempt to read it before assigning any value to it, Euphoria will issue a run-time error as "variable xyz has never been assigned a value".

Assigning Values

The equal sign = is used to assign values to variables. The operand to the left of the = operator is the name of the variable, and the operand to the right of the = operator is the value stored in the variable. For example −

```
#!/home/euphoria/bin/eui

-- Here is the declaration of the variables.
integer counter
integer miles
sequence name

counter = 100           -- An integer assignment
miles   = 1000.0        -- A floating point
name    = "John"        -- A string ( sequence )

printf(1, "Value of counter %d\n", counter )
printf(1, "Value of miles %f\n", miles )
printf(1, "Value of name %s\n", {name} )
```

Here 100, 1000.0 and "John" are the values assigned to *counter*, *miles* and *name* variables, respectively. While running this program, this will produce following result −

```
Value of counter 100
Value of miles 1000.000000
Value of name John
```

To guard against forgetting to initialise a variable, and also because it may make the code clearer

to read, you can combine declaration and assignment −

```
integer n = 5
```

This is equivalent to −

```
integer n
n = 5
```

Identifier Scope

The scope of an identifier is a description of what code can 'access' it. Code in the same scope of an identifier can access that identifier and code not in the same scope cannot access it.

The scope of a variable depends upon where and how it is declared.

- If it is declared within a **for, while, loop** or **switch**, its scope starts at the declaration and ends at the respective **end** statement.
- In an **if** statement, the scope starts at the declaration and ends either at the next **else, elsif** or **end if** statement.
- If a variable is declared within a routine, the scope of the variable starts at the declaration and ends at the routine's end statement. This is knows as a private variable.
- If a variable is declared outside of a routine, its scope starts at the declaration and ends and the end of the file it is declared in. This is known as a module variable.
- The scope of a **constant** that does not have a scope modifier, starts at the declaration and ends and the end of the file it is declared in.
- The scope of a **enum** that does not have a scope modifier, starts at the declaration and ends and the end of the file it is declared in.
- The scope of all **procedures, functions** and **types**, which do not have a scope modifier, starts at the begining of the source file and ends at the end of the source file in which they are declared.

Constants, enums, module variables, procedures, functions and types, which do not have a scope modifier are referred to as local. However, these identifiers can have a scope modifier preceding their declaration, which causes their scope to extend beyond the file they are declared in.

- If the keyword **global** precedes the declaration, the scope of these identifiers extends to the whole application. They can be accessed by code anywhere in the application files.
- If the keyword **public** precedes the declaration, the scope extends to any file that explicitly includes the file in which the identifier is declared, or to any file that includes a file that in turn *public* includes the file containing the *public* declaration.
- If the keyword **export** precedes the declaration, the scope only extends to any file that directly includes the file in which the identifier is declared.

When you **include** a Euphoria file in another file, only the identifiers declared using a scope modifier are accessible to the file doing the include. The other declarations in the included file are invisible to the file doing the include.