EUPHORIA
programming language

tutorialspoint
SIMPLYEASYLEARNING

About Tutorial

This tutorial gives you basic understanding of Euphoria programming language. Euphoria is simple, flexible, easy to learn, and interpreted high-level programming language for DOS, Windows, Linux, FreeBSD, and more. This tutorial describes everything a programmer needs to know such as its environment, data types, syntax and operators, file handling, and controlling the flow of program.

Audience

This tutorial is designed for the aspiring students who are keen to learn and understand Euphoria in detail. This tutorial would be of great help for the IT professionals working as programmers. The enthusiastic readers can access this tutorial as a source of additional reading.

Prerequisites

Before proceeding with this tutorial, you need to have a basic knowledge of working on Windows or Linux. You need to be familiar with any programming language such as C, C++. You need to have sound understanding of operating system, memory allocation and de-allocation, and basics of efficient programming and debugging.

Disclaimer & Copyright

Table of Contents

1. EUPHORIA OVERVIEW

Euphoria stands for **E**nd-**U**ser **P**rogramming with **H**ierarchical **O**bjects for **R**obust **I**nterpreted **A**pplications. Euphoria's first incarnation was created by Robert Craig on an Atari Mega-ST and it was first released in 1993. It is now maintained by Rapid Deployment Software.

It is a free, simple, flexible, easy to learn, and interpreted but extremely fast 32-bit high-level programming language for DOS, Windows, Linux, FreeBSD and more.

Euphoria is being used to develop Windows GUI programs, high-speed DOS games, and Linux/FreeBSD X Windows programs. Euphoria can also be used for CGI (Web-based) programming.

Euphoria Features

Here is the list of major features of Euphoria:

- It is a simple, flexible, powerful language definition that is easy to learn and use.

- It supports dynamic storage allocation which means variables grow or shrink without the programmer having to worry about allocating and freeing the memory. It takes care of garbage collection automatically.

- It is extremely faster than conventional interpreters such as Perl and Python.

- Euphoria programs run under Linux, FreeBSD, 32-bit Windows, and any DOS environment.

- Euphoria programs are not subject to any 640K memory limitations.

- It provides an optimizing Euphoria-To-C translator which you can use to translate your Euphoria program into C and then compile it with a C compiler to get an executable (.exe) file. This can boost your program speed by 2 to 5 times.

- Underlying hardware are completely hidden which means programs are not aware of word-lengths, underlying bit-level representation of values, byte-order etc.

- Euphoria installation comes along with a full-screen source debugger, an execution profiler, and a full-screen multi-file editor.

- It supports run-time error-handling, subscript, and type checking.
- It is an open source language and comes completely free of cost.

Platform Requirements

Euphoria is available on Windows, Linux, FreeBSD, and OSX. Here is the bare minimum version required with the following platforms –

- **WIN32 version:** You need Windows 95 or any later version of Windows. It runs fine on XP and Vista.
- **Linux version:** You need any reasonably up-to-date Linux distribution, that has libc6 or later. For example, Red Hat 5.2 or later works fine.
- **FreeBSD version:** You need any reasonably up-to-date FreeBSD distribution.
- **Mac OS X version:** You need any reasonably up-to-date Intel based Mac.

Euphoria Limitations

Here are some prominent limitations of Euphoria:

- Even though Euphoria is simple, fast, and flexible enough for the programmers; it does not provide call support for many important functionalities. For example, network programming.
- Euphoria was invented in 1993, and still you would not find any book written on this language. There is also not much documentation available for the language.

But these days, the language is getting popular very fast and you can hope to have nice utilities and books available for the language very soon.

Euphoria Licensing

This product is free and open source, and has benefited from the contributions of many people. You have complete royalty-free rights to distribute any Euphoria programs that you develop.

Icon files, such as euphoria.ico and binaries available in euphoria\bin, may be distributed with or without your changes.

You can **shroud** or **bind** your program and distribute the resulting files royalty-free. Some additional 3rd party legal restrictions might apply when you use the Euphoria-To-C translator.

The generous **Open Source License** allows Euphoria to use for both personal and commercial purposes. Unlike many other open source licenses, your changes do not have to be made open source.

2. EUPHORIA ENVIRONMENT

This chapter describes about the installation of Euphoria on various platforms. You can follow the steps to install Euphoria on Linux, FreeBSD, and 32-bit Windows. So you can choose the steps based on your working environment.

Linux, Free BSD Installation

Official website provides **.tar.gz** file to install Euphoria on your Linux or BSD OS. You can download your latest version of Euphoria from its official website Download Euphoria.

Once you have .tar.gz file, here are three simple steps to be performed to install Euphoria on your Linux or Free BSD machine:

Step 1: Installing Files

Untar the downloaded file **euphoria-4.0b2.tar.gz** in a directory where you want to install Euphoria. If you want to install it in /home directory as follows, then:

```
$cp euphoria-4.0b2.tar.gz /home

$cd /home

$gunzip euphoria-4.0b2.tar.gz

$tar -xvf euphoria-4.0b2.tar
```

This creates a directory hierarchy inside **/home/euphoria-4.0b2** directory as follows:

```
$ls -l

-rw-r--r--  1 1001 1001 2485 Aug 17 06:15 Jamfile

-rw-r--r--  1 1001 1001 5172 Aug 20 12:37 Jamrules

-rw-r--r--  1 1001 1001 1185 Aug 13 06:21 License.txt

drwxr-xr-x  2 1001 1001 4096 Aug 31 10:07 bin

drwxr-xr-x  7 1001 1001 4096 Aug 31 10:07 demo

-rw-r--r--  1 1001 1001  366 Mar 18 09:02 file_id.diz
```

```
drwxr-xr-x  4 1001 1001 4096 Aug 31 10:07 include

-rw-r--r--  1 1001 1001 1161 Mar 18 09:02 installu.doc

drwxr-xr-x  4 1001 1001 4096 Aug 31 10:07 source

drwxr-xr-x 19 1001 1001 4096 Sep  7 12:09 tests

drwxr-xr-x  2 1001 1001 4096 Aug 31 10:07 tutorial
```

Note: File name euphoria-4.0b2.tar.gz depends on latest version available. We are using 4.0b2 version of the language for this tutorial.

Step 2: Setting Up the Path

After installing Euphoria, you need to set proper paths so that your shell can find required Euphoria binaries and utilities. Before proceeding, there are following three important environment variables you need to set up:

1. Set PATH environment variable to point /home/euphoria-4.0b2/bin directory.
2. Set EUDIR environment variable to point to /home/euphoria-4.0b2.
3. Set EUINC environment variable to point to /home/euphoria-4.0b2/include.

These variables can be set as follows –

```
$export PATH=$PATH:/home/euphoria-4.0b2/bin

$export EUDIR=/home/euphoria-4.0b2

$export EUINC=/home/euphoria-4.0b2/include
```

Note: The above commands used to set environment variables may differ depending on your Shell. We used *bash* shell for executing these commands to set the variables.

Step 3: Confirming Installation

Confirm if you installed Euphoria successfully or not.

Execute the following command:

```
$eui -version
```

If you get following result, then it means you have installed Euphoria successfully; otherwise you have to go back and check all the steps again.

```
$eui -version

Euphoria Interpreter 4.0.0 beta 2 (r2670) for Linux
```

```
Using System Memory

$
```

That is it, Euphoria Programming Environment is ready on your UNIX machine, and you can start writing complex programs in easy steps.

WIN32 and DOS Installation

Official website provides **.exe** file to install Euphoria on your WIN32 or DOS OS. You can download your latest version of Euphoria from its official website Download Euphoria.

Once you have .exe file, here are three simple steps to follow for installing Euphoria Programming language on your WIN32 or DOS machine:

Step 1: Installing Files

Double click on the downloaded **.exe** setup program to install all the files. We downloaded euphoria-40b2.exe file for installation.

The filename euphoria-40b2.exe depends on latest version available. We use version 4 beta 2 of the language.

By default Euphoria would be installed in *C:\euphoria-40b2* directory but you can also select a desired location.

Step 2: Rebooting the Machine

Re-boot your machine to complete the installation.

Step 3: Confirming Installation

Confirm if you installed Euphoria successfully or not.

Execute the following command:

```
c:\>eui -version
```

If you get following result, then it means you have installed Euphoria successfully; otherwise you have to go back and check all the steps again.

```
c:\>eui -version

Euphoria Interpreter 4.0.0 beta 2 (r2670) for Windows

Using Managed Memory
```

```
c:\>
```

That is it, Euphoria Programming Environment is ready on your WIN32 machine, and you can start writing complex programs in easy steps.

Euphoria Interpreters

Depending on the platform you are using, Euphoria has multiple interpreters:

- The main interpreter is **eui**.
- On windows platforms, you have two choices. If you run **eui** then a console window is created. If you run **euiw** then no console is created, making it suitable for GUI applications.
- Euphoria does not care about your choice of file extensions. By convention however; the console-based applications come with **.ex** extension.
- GUI-based applications have **.exw** extension and the include files have **.e** extension.

3. BASIC SYNTAX

The Euphoria language has many similarities to Perl, C, and Java. However, there are some definite differences between the languages. This chapter is designed to quickly get you up to speed on the syntax that is expected in Euphoria.

This tutorial assumes you are working with Linux and all the examples have been written on Linux platform. But it is observed that there is no any prominent difference in program syntax on Linux and WIN32. Hence you can follow the same steps on WIN32.

First Euphoria Program

Let us write a simple Euphoria program in a script. Type the following source code in test.ex file and save it.

```
#!/home/euphoria-4.0b2/bin/eui

puts(1, "Hello, Euphoria!\n")
```

Let us say, Euphoria interpreter is available in */home/euphoria-4.0b2/bin/* directory. Now run this program as follows:

```
$ chmod +x test.ex    # This is to make file executable

$ ./test.ex
```

This produces the following result:

```
Hello, Euphoria!
```

This script used a built-in function **puts()** which takes two arguments. First argument indicates file name or device number, and second argument indicates a string which you want to print. Here 1 indicates STDOUT device.

Euphoria Identifiers

A Euphoria identifier is a name used to identify a variable, function, class, module, or other object. An identifier starts with a letter A to Z or a to z and then followed by letters, digits, or underscores.

Euphoria does not allow punctuation characters such as @, $, and % within identifiers.

Euphoria is a case sensitive programming language. Thus **Manpower** and **manpower** are two different identifiers in Euphoria. For example, the valid identifiers are:

- n
- color26
- ShellSort
- quick_sort
- a_very_long_indentifier

Reserved Words

The following list shows the reserved words in Euphoria. These reserved words may not be used as constant or variable or any other identifier names. Euphoria keywords contain lowercase letters only.

and	exit	override
as	export	procedure
break	fallthru	public
by	for	retry
case	function	return
constant	global	routine
continue	goto	switch
do	if	then
else	ifdef	to
elsedef	include	type

elsif	label	until
elsifdef	loop	while
end	namespace	with
entry	not	without
enum	or	xor

Expressions

Euphoria lets you calculate results by forming expressions. However, in Euphoria you can perform calculations on entire sequences of data with one expression.

You can handle a sequence much as you would handle a single number. It can be copied, passed to a subroutine, or calculated upon as a unit. For example:

```
{1,2,3} + 5
```

This is an expression that adds the sequence {1, 2, 3} and the atom 5 to get the resulting sequence {6, 7, 8}. You would learn sequences in subsequent chapters.

Blocks of Code

One of the first caveats programmers encounter when learning Euphoria is the fact that there are no braces to indicate blocks of code for procedure and function definitions or flow control. Blocks of code are denoted by associated keywords.

The following example shows **if...then...end if** block:

```
if condition then

   code block comes here

end if
```

Multi-Line Statements

Statements in Euphoria typically end with a new line. Euphoria does however, allow to write a single statement in multiple lines. For example:

```
total = item_one +

        item_two +

        item_three
```

Escape Characters

Escape characters may be entered using a back-slash. For example:

The following table is a list of escape or non-printable characters that can be represented with backslash notation.

Backslash notation	Description
\n	Newline
\r	Carriage return
\t	Tab
\\	Backslash
\"	Double quote
\'	Single quote

Comments in Euphoria

Any comments are ignored by the compiler and have no effect on execution speed. It is advisable to use more comments in your program to make it more readable.

There are three forms of comment text:

1. Comments start by two dashes and extend to the end of the current line.

2. The multi-line format comment is kept inside /*...*/, even if that occurs on a different line.

3. You can use a special comment beginning with the two character sequence “#!” only on the first line of the program.

Examples

```
#!/home/euphoria-4.0b2/bin/eui



-- First comment

puts(1, "Hello, Euphoria!\n") -- second comment



/* This is a comment which extends over a number

of text lines and has no impact on the program

*/
```

This produces the following result:

```
Hello, Euphoria!
```

Note: You can use a special comment beginning with “#!”. This informs the Linux shell that your file should be executed by the Euphoria interpreter.

4. EUPHORIA VARIABLES

Variables are nothing but reserved memory locations to store values. This means 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.

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. Variable can be assigned in the following manner:

Variable_Name = Variable_Value

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. This program produces the following result:

```
Value of counter 100

Value of miles 1000.000000

Value of name John
```

To guard against forgetting to initialize 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 the following:

```
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 as identifier 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 beginning 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 **locals**. 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*.

5. EUPHORIA CONSTANTS

Constants are also variables that are assigned an initial value that can never change in the program’s life. Euphoria allows to define constants using **constant** keyword as follows:

```
constant MAX = 100

constant Upper = MAX - 10, Lower = 5

constant name_list = {"Fred", "George", "Larry"}
```

The result of any expression can be assigned to a constant, even one involving calls to previously defined functions, but once the assignment is made, the value of the constant variable is "locked in".

Constants may not be declared inside a subroutine. 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.

Examples

```
#!/home/euphoria-4.0b2/bin/eui



constant MAX = 100

constant Upper = MAX - 10, Lower = 5



printf(1, "Value of MAX %d\n", MAX )

printf(1, "Value of Upper %d\n", Upper )

printf(1, "Value of Lower %d\n", Lower )



MAX = MAX + 1

printf(1, "Value of MAX %d\n", MAX )
```

This produces the following error:

```
./test.ex:10

<0110>:: may not change the value of a constant

MAX = MAX + 1

   ^

Press Enter
```

If you delete last two lines from the example, then it produces the following result:

```
Value of MAX 100

Value of Upper 90

Value of Lower 5
```

The *enums*

An enumerated value is a special type of constant where the first value defaults to the number 1 and each item after that is incremented by 1. Enums can only take numeric values.

Enums may not be declared inside a subroutine. The scope of an **enum** that does not have a scope modifier, starts at the declaration and ends and the end of the file it is declared in.

Examples

```
#!/home/euphoria-4.0b2/bin/eui



enum ONE, TWO, THREE, FOUR



printf(1, "Value of ONE %d\n", ONE )

printf(1, "Value of TWO %d\n", TWO )

printf(1, "Value of THREE %d\n", THREE )

printf(1, "Value of FOUR %d\n", FOUR )
```

This will produce following result –

```
Value of ONE 1

Value of TWO 2

Value of THREE 3

Value of FOUR 4
```

You can change the value of any one item by assigning it a numeric value. Subsequent values are always the previous value plus one, unless they too are assigned a default value.

```
#!/home/euphoria-4.0b2/bin/eui



enum ONE, TWO, THREE, ABC=10, XYZ



printf(1, "Value of ONE %d\n", ONE )

printf(1, "Value of TWO %d\n", TWO )

printf(1, "Value of THREE %d\n", THREE )

printf(1, "Value of ABC %d\n", ABC )

printf(1, "Value of XYZ %d\n", XYZ )
```

This produces the following result:

```
Value of ONE 1

Value of TWO 2

Value of THREE 3

Value of ABC 10

Value of XYZ 11
```

Sequences use integer indices, but with enum you may write code like this:

```
enum X, Y

sequence point = { 0,0 }

point[X] = 3
```

```
point[Y] = 4
```

6. EUPHORIA DATATYPES

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters.

Euphoria has some standard types that are used to define the operations possible on them and the storage method for each of them.

Euphoria has following four standard data types –

- integer
- atom
- sequence
- object

The understanding of atoms and sequences is the key to understanding Euphoria.

Integers

Euphoria integer data types store numeric values. They are declared and defined as follows:

```
integer var1, var2



var1 = 1

var2 = 100
```

The variables declared with type integer must be atoms with **integer** values from -1073741824 to +1073741823 inclusive. You can perform exact calculations on larger integer values, up to about 15 decimal digits, but declare them as atom, rather than integer.

Atoms

All data objects in Euphoria are either atoms or sequences. An atom is a single numeric value. Atoms can have any integer or double-precision floating point value. Euphoria atoms are declared and defined as follows:

```
atom var1, var2, var3


var1 = 1000

var2 = 198.6121324234

var3 = 'E'
```

The atoms can range from approximately -1e300 to +1e300 with 15 decimal digits of accuracy. An individual character is an **atom** which must may be entered using single quotes. For example, all the following statements are legal:

```
-- Following is equivalent to the atom 66 - the ASCII code for B

char = 'B'


-- Following is equivalent to the sequence {66}

sentence = "B"
```

Sequences

A sequence is a collection of numeric values which can be accessed through their index. All data objects in Euphoria are either atoms or sequences.

Sequence index starts from 1 unlike other programming languages where array index starts from 0. Euphoria sequences are declared and defined as follows:

```
sequence var1, var2, var3, var4


var1 = {2, 3, 5, 7, 11, 13, 17, 19}

var2 = {1, 2, {3, 3, 3}, 4, {5, {6}}}

var3 = {{"zara", "ali"}, 52389, 97.25}

var4 = {} -- the 0 element sequence
```

A character string is just a **sequence** of characters which may be entered using double quotes. For example, all the following statements are legal:

```
word = 'word'

sentence = "ABCDEFG"
```

Character strings may be manipulated and operated upon just like any other sequences. For example, the above string is entirely equivalent to the sequence:

```
sentence = {65, 66, 67, 68, 69, 70, 71}
```

You will learn more about sequence in Euphoria - Sequences.

Objects

This is a super data type in Euphoria which may take on any value including atoms, sequences, or integers. Euphoria objects are declared and defined as follows:

```
object var1, var2, var3



var1 = {2, 3, 5, 7, 11, 13, 17, 19}

var2 = 100

var3 = 'E'
```

An object may have one of the following values:

- a sequence
- an atom
- an integer
- an integer used as a file number
- a string sequence, or single-character atom

7. EUPHORIA OPERATORS

Euphoria provides a rich set of operators to manipulate variables. We can divide all the Euphoria operators into the following groups:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operators
- Miscellaneous Operators

The Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in Algebra. The following table lists the arithmetic operators. Assume integer variable A holds 10 and variable B holds 20 then:

Operator	Description	Example
+	Addition: Adds values on either side of the operator	A + B gives 30
-	Subtraction: Subtracts right hand operand from left hand operand	A - B gives -10
*	Multiplication: Multiplies values on either side of the operator	A * B gives 200
/	Division: Divides left hand operand by right hand operand	B / A gives 2
+	Unary Plus: This has no impact on the variable value.	+B gives 20
-	Unary Minus: This creates a negative value of the given variable.	-B gives -20

The Relational Operators

There are following relational operators supported by Euphoria language. Assume variable A holds 10 and variable B holds 20 then:

Operator	Description	Example
=	Checks if the value of two operands are equal or not, if yes then condition becomes true.	(A = B) is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

The Logical Operators

The following table lists the logical operators. Assume boolean variables A holds 1 and variable B holds 0 then:

Operator	Description	Example

and	Called Logical AND operator. If both the operands are non zero then then condition becomes true.	(A and B) is false.
or	Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true.	(A or B) is true.
xor	Called Logical XOR Operator. Condition is true if one of them is true, if both operands are true or false then condition becomes false.	(A xor B) is true.
not	Called Logical NOT Operator which negates the result. Using this operator, true becomes false and false becomes true	not(B) is true.

You can also apply these operators to numbers other than 1 or 0. The convention is: zero means *false* and non-zero means *true*.

The Assignment Operators

There are following assignment operators supported by Euphoria language:

Operator	**Description**	**Example**
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B assigns value of A + B into C

+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to
		
		C = C + A

-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to
		
		C = C - A

*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
&=	Concatenation operator	C &= {2} is same as C = {C} & {2}

Note: The equals symbol '=' used in an assignment statement is not an operator, it is just a part of the syntax.

Miscellaneous Operators

There are few other operators supported by Euphoria Language.

The '&' Operator

Any two objects may be concatenated using "&" operator. The result is a sequence with a length equal to the sum of the lengths of the concatenated objects.

For example:

```
#!/home/euphoria-4.0b2/bin/eui

sequence a, b, c

a = {1, 2, 3}

b = {4}

c = {1, 2, 3} & {4}



printf(1, "Value of c[1] %d\n", c[1] )

printf(1, "Value of c[2] %d\n", c[2] )

printf(1, "Value of c[3] %d\n", c[3] )
printf(1, "Value of c[4] %d\n", c[4] )
```

This produces the following result:

```
Value of c[1] 1

Value of c[2] 2

Value of c[3] 3

Value of c[4] 4
```

Precedence of Euphoria Operators

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator.

For example, x = 7 + 3 * 2

Here, x is assigned 13, not 20 because operator * has higher precedence than +.

Hence it first starts with 3*2 and then adds into 7.

Here operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators is evaluated first.

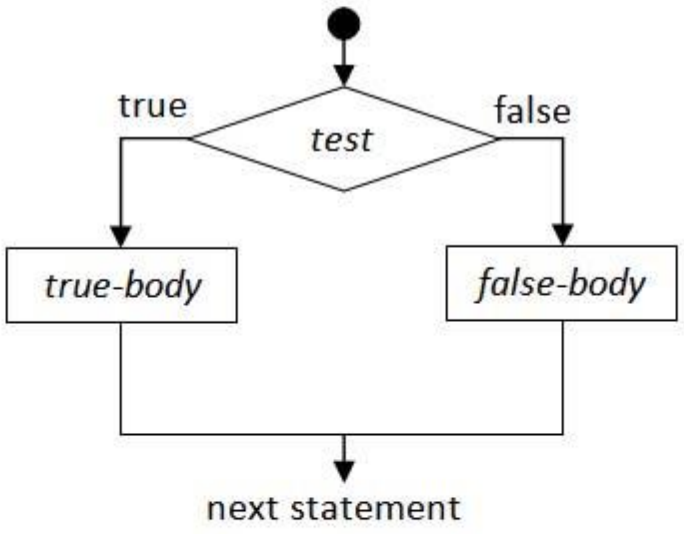
Category	Operator	Associativity
Postfix	function/type calls	
Unary	+ - ! not	Right to left
Multiplicative	* /	Left to right
Additive	+ -	Left to right
Concatenation	&	Left to right

Relational	> >= < <=	Left to right
Equality	= !=	Left to right
Logical AND	and	Left to right
Logical OR	or	Left to right
Logical XOR	xor	Left to right
Comma	,	Left to right

8. EUPHORIA BRANCHING

Branching is the most important aspect of any programming language. While writing your program, you may encounter a situation when you have to take a decision or you have to select one option out of the given many options.

Following diagram shows a simple scenario where a program needs to take one of the two paths based on the given condition.



Euphoria provides following three types of decision making (branching or conditional) statements –

- if statement
- switch statement
- ifdef statement

Let us see the statements in detail:

The *if* Statement

An **if** statement consists of a boolean expression followed by one or more statements.

Syntax

The syntax of *if* statement is:

```
if expression then

   -- Statements will execute if the expression is true

end if
```

If the boolean expression evaluates to true then the block of code inside the if statement is executed. If it evaluates to false, then the first set of code after the end of the if statement is executed.

Example

```
#!/home/euphoria-4.0b2/bin/eui



integer a = 10

integer b = 20



if (a + b) < 40 then

   printf(1, "%s\n", {"This is true if statement!"})

end if



if (a + b) > 40 then

   printf(1, "%s\n", {"This is not true if statement!"})

end if
```

This produces the following result:

```
This is true if statement!
```

The *if...else* Statement

An **if** statement can be followed by an optional **else** statement, which executes when the boolean expression is false.

Syntax

The syntax of *if...else* statement is as follows:

```
if expression then

   -- Statements will execute if the expression is true

else

   -- Statements will execute if the expression is false

end if
```

Example

```
#!/home/euphoria-4.0b2/bin/eui



integer a = 10

integer b = 20



if (a + b) < 40 then

   printf(1, "%s\n", {"This is inside if statement!"})

else

   printf(1, "%s\n", {"This is inside else statement!"})

end if
```

This produces the following result:

```
This is inside if statement!
```

The *if...elsif...else* Statement

An **if** statement can be followed by any number of optional **elsif...else** statement, which is very useful to test various conditions using single if...elsif statement.

Syntax

The syntax of *if...elsif...else* statement is as follows:

```
if expression1 then

   -- Executes when the Boolean expression 1 is true
```

```
elsif expression2 then

   -- Executes when the Boolean expression 2 is true

elsif expression3 then

   -- Executes when the Boolean expression 3 is true

else

   -- Executes when none of the above condition is true.

end if
```

Example

```
#!/home/euphoria-4.0b2/bin/eui



integer a = 10

integer b = 20



if (a + b) = 40 then

   printf(1, "Value of (a + b ) is  %d\n", a + b )

elsif (a + b) = 45 then

    printf(1, "Value of (a + b ) is  %d\n", a + b )

elsif (a + b) = 30 then

    printf(1, "Value of (a + b ) is  %d\n", a + b )

else

    printf(1, "Value of (a + b ) is  %d\n", 0 )

end if
```

This produces the following result:

```
Value of (a + b ) is  30
```

The *if...label...then* Statement

An **if** statement can have a label clause just before the first **then** keyword. Note that an **elsif** clause cannot have a label.

An *if...lable* is used just to name the if block and label names must be double quoted constant strings having single or multiple words. The label keyword is a case sensitive and should be written as **label**.

Syntax

The syntax of label clause is as follows:

```
if expression label "Label Name" then

   -- Executes when the boolean expression  is true

end if
```

Example

```
#!/home/euphoria-4.0b2/bin/eui



integer a = 10

integer b = 20



if (a + b) = 40 label "First IF Block" then

   printf(1, "Value of (a + b ) is  %d\n", a + b )

elsif (a + b) = 45 then

    printf(1, "Value of (a + b ) is  %d\n", a + b )

elsif (a + b) = 30 then

    printf(1, "Value of (a + b ) is  %d\n", a + b )

else

    printf(1, "Value of (a + b ) is  %d\n", 0 )

end if
```

This produces the following result:

```
Value of (a + b ) is  30
```

Nested *if...else* Statement

It is always legal to nest **if...else** statements. This means you can have one if-else statement within another if-else statements.

Syntax

The syntax of nested *if...else* is as follows:

```
if expression1 then

    -- Executes when the boolean expression1  is true

   if expression2 then

       -- Executes when the boolean expression2  is true

   end if

end if
```

Example

```
#!/home/euphoria-4.0b2/bin/eui



integer a = 10

integer b = 20

integer c = 0



if c = 0 then

   printf(1, "Value of c is equal to %d\n", 0 )

   if (a + b) = 30 then

      printf(1, "Value of (a + b ) is  equal to %d\n", 30)

   else

      printf(1, "Value of (a + b ) is equal to  %d\n", a + b )
   end if

else

   printf(1, "Value of c is equal to %d\n", c )

end if
```

This produces the following result:

```
Value of c is equal to 0

Value of (a + b ) is  equal to 30
```

The *switch* Statement

The **switch** statement is used to run a specific set of statements, depending on the value of an expression. It often replaces a set of **if…elsif** statements giving you more control and readability of your program.

Syntax

The syntax of simple switch statement is as follows:

```
switch expression do

   case <val> [, <val-1>....] then

      -- Executes when the expression matches one of the values

   case <val> [, <val-1>....] then

      -- Executes when the expression matches one of the values

   .....................

   case else

      -- Executes when the expression does not matches any case.

end if
```

The <val> in a case must be either an atom, literal string, constant or enum. Multiple values for a single case can be specified by separating the values by commas. By default, control flows to the end of the switch block when the next case is encountered.

Example

```
#!/home/euphoria-4.0b2/bin/eui


atom marks = 'C'


switch marks do
   case 'A' then
      puts(1, "Excellent!\n" )
   case 'B', 'C' then
      puts(1, "Well done!\n" )
   case 'D' then
      puts(1, "You passed!\n" )
   case 'F' then
      puts(1, "Better try again!\n" )
   case else
      puts(1, "Invalid grade!\n" )
end switch
```

This produces the following result:

```
Well done!
```

The *switch...with fallthru* Statement

The **case** statement of a **switch** is executed when it matches with the given expression value and by default it comes out. By default, control flows to the end of the switch block when the next case is encountered.

The default for a particular switch block can be changed so that control passes to the next executable statement whenever a new case is encountered by using **with fallthru** in the switch statement:

Syntax

The syntax of simple *switch...with fallthru* statement is as follows:

```
switch expression with fallthru do

   case <val> [, <val-1>....] then

      -- Executes when the expression matches one of the values

      break -- optional to come out of the switch from this point.

   case <val> [, <val-1>....] then

      -- Executes when the expression matches one of the values

      break -- Optional to come out of the switch from this point.

   .....................

   case else

      -- Executes when the expression does not matches any case.

      break -- Optional to come out of the switch from this point.

end if
```

Example

```
#!/home/euphoria-4.0b2/bin/eui



atom marks = 'C'



switch marks with fallthru do

   case 'A' then

      puts(1, "Excellent!\n" )

   case 'B', 'C' then

      puts(1, "Well done!\n" )

   case 'D' then

      puts(1, "You passed!\n" )
   case 'F' then
      puts(1, "Better try again!\n" )
   case else
      puts(1, "Invalid grade!\n" )
end switch
```

This produces the following result:

```
Well done!
You passed!
Better try again!
Invalid grade!
```

You can use optional **break** statement to come out from a point inside a switch statement as follows:

```
#!/home/euphoria-4.0b2/bin/eui


atom marks = 'C'


switch marks with fallthru do
   case 'A' then
      puts(1, "Excellent!\n" )
      break
   case 'B', 'C' then
      puts(1, "Well done!\n" )
      break
   case 'D' then
      puts(1, "You passed!\n" )
      break
   case 'F' then

      puts(1, "Better try again!\n" )

      break

   case else

      puts(1, "Invalid grade!\n" )

      break

end switch
```

This produces the following result:

```
Well done!
```

The *switch....label* Statement

The **switch** statement can have an optional **label** to name the switch block. This name can be used in nested switch break statements to break out of an enclosing switch rather than just the owning switch.

A switch label is used just to name the block and label names must be double quoted constant strings having single or multiple words. The label keyword is a case sensitive and should be written as **label**.

Syntax

The syntax of simple *switch...label* statement is as follows:

```
switch expression label "Label Name" do

   case <val> [, <val-1>....] then

      -- Executes when the expression matches one of the values

      break "LEBEL NAME"

   case <val> [, <val-1>....] then

      -- Executes when the expression matches one of the values

      break "LEBEL NAME"

   .....................

   case else
```

```
      -- Executes when the expression does not matches any case.

      break "LEBEL NAME"

end if
```

Example

```
#!/home/euphoria-4.0b2/bin/eui



atom marks = 'C'

atom scale = 'L'



switch marks label "MARKS" do

   case 'A' then

      puts(1, "Excellent!\n" )

   case 'B', 'C' then

      puts(1, "Well done!\n" )

      switch scale label "SCALE" do

         case 'U' then

             puts(1, "Upper scale!\n" )

             break "MARKS"

         case 'L' then

             puts(1, "Lower scale!\n" )

             break "MARKS"

         case else

             puts(1, "Invalid scale!\n" )

             break "MARKS"

      end switch

   case 'D' then
```

```
      puts(1, "You passed!\n" )
   case 'F' then
      puts(1, "Better try again!\n" )
   case else
      puts(1, "Invalid grade!\n" )
end switch
```

This produces the following result:

```
Well done!
Lower scale!
```

Note: If you are not using a *with fallthru* statement then you do not need to use a label because switch statement would come out automatically.

The *ifdef* Statement

The **ifdef** statement is executed at parse time not runtime. This allows you to change the way your program operates in a very efficient manner.

Since the ifdef statement works at parse time, runtime values cannot be checked, instead special definitions can be set or unset at parse time as well.

Syntax

The syntax of *ifdef* statement is as follows:

```
ifdef macro then
   -- Statements will execute if the macro is defined.
end if
```

If the boolean expression evaluates to true then the block of code inside the if statement is executed. If not, then the first set of code after the end of the ifdef statement will be executed.

The *ifdef* checks the macros defined by using **with define** keywords. There are plenty of macros defined like WIN32_CONSOLE, WIN32, or LINUX. You can define your own macros as follows:

```
with define    MY_WORD    -- defines
```

You can un-define an already defined word as follows:

```
without define OTHER_WORD -- undefines
```

Example

```
#!/home/euphoria-4.0b2/bin/eui


with define DEBUG


integer a = 10

integer b = 20


ifdef DEBUG then

   puts(1, "Hello, I am a debug message one\n")

end ifdef


if (a + b) < 40 then

   printf(1, "%s\n", {"This is true if statement!"})

end if


if (a + b) > 40 then

   printf(1, "%s\n", {"This is not true if statement!"})

end if
```

This produces the following result:

```
Hello, I am a debug message one

This is true if statement!
```

The *ifdef...elsedef* Statement

You can take one action if given macro is defined otherwise you can take another action in case given macro is not defined.

Syntax

The syntax of *ifdef...elsedef* statement is as follows:

```
ifdef macro then

   -- Statements will execute if the macro is defined.

elsedef

   -- Statements will execute if the macro is not defined.

end if
```

Example

```
#!/home/euphoria-4.0b2/bin/eui



ifdef WIN32 then

   puts(1, "This is windows 32 platform\n")

elsedef

   puts(1, "This is not windows 32 platform\n")

end ifdef
```

When you run this program on Linux machine, it produces the following result:

```
This is not windows 32 platform
```

The *ifdef...elsifdef* Statement

You can check multiple macros using **ifdef...elsifdef** statement.

Syntax

The syntax of *ifdef...elsifdef* statement is as follows:

```
ifdef macro1 then

   -- Statements will execute if the macro1 is defined.

elsifdef macro2 then

   -- Statements will execute if the macro2 is defined.

elsifdef macro3 then

   -- Statements will execute if the macro3 is defined.

.......................

elsedef

   -- Statements will execute if the macro is not defined.

end if
```

Example

```
#!/home/euphoria-4.0b2/bin/eui



ifdef WIN32 then

   puts(1, "This is windows 32 platform\n")

elsifdef LINUX then

   puts(1, "This is LINUX platform\n")

elsedef

   puts(1, "This is neither Unix nor Windows\n")

end ifdef
```

When you run this program on Linux machine, it produces the following result:

```
This is LINUX platform
```

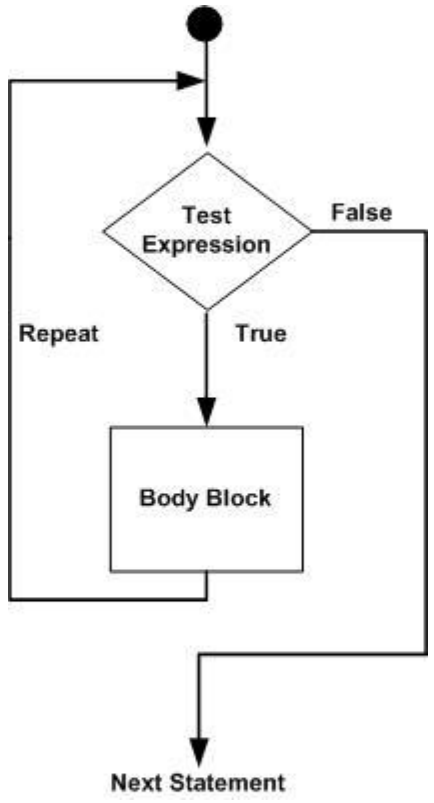
All the above statements have various forms which provide you a flexibility and ease of use based on different situations.

9. EUPHORIA LOOPS

Looping is yet another most important aspect of any programming language. While writing your program, you may encounter a situation when you have to execute same statement many times and sometime may be infinite number of times.

There are several ways to specify for how long the process should go on, and how to stop or otherwise alter it. An iterative block may be informally called a loop, and each execution of code in a loop is called an iteration of the loop.

The following diagram shows a simple logical flow of a loop:



Euphoria provides following three types of loop statements:

- **while** statement
- **loop until** statement
- **for** statement

All the above statements provide you flexibility and ease of use based on different situations. Let us see them in detail one by one:

While Statement

A while loop is a control structure that allows you to repeat a task for a certain number of times.

Syntax

The syntax of a while loop is as follows:

```
while expression do

   -- Statements executed if expression returns true

end while
```

When executing, if the *expression* results in true then the actions inside the loop is executed. This continues as long as the expression result is true.

The key point of the *while* loop is that, the loop might not ever run. When the expression is tested and the result is false, the loop body is skipped and the first statement after the while loop is executed.

Example

```
#!/home/euphoria-4.0b2/bin/eui



integer a = 10



while a < 20 do

   printf(1, "value of a : %d\n", a)

   a = a + 1

end while
```

This produces the following result:

```
value of a : 10

value of a : 11

value of a : 12

value of a : 13
value of a : 14

value of a : 15

value of a : 16

value of a : 17

value of a : 18

value of a : 19
```

The *while....with entry* Statement

It is often the case that the first iteration of a loop is somehow special. Some things have to be done before the loop starts. They are done before the statement starting the loop.

The ***with entry*** statement serves the purpose very well. You need to use this statement with while loop and just add the **entry** keyword at the point you wish the first iteration starts.

Syntax

The syntax of a while loop with entry is as follows:

```
while expression with entry do

   -- Statements executed if expression returns true

entry

   -- Initialisation statements.

end while
```

Before executing the *expression*, it executes initialization statements and then it starts as a normal while loop. Later, these initialization statements become part of the loop body.

Example

```
#!/home/euphoria-4.0b2/bin/eui



integer a = 10
```

```

while a < 20 with entry do

   printf(1, "value of a : %d\n", a)

   a = a + 1

entry

   a = a + 2

end while
```

This produces the following result:

```
value of a : 12

value of a : 15

value of a : 18
```

The *while....label* Statement

A **while** loop can have a **label** clause just before the first **do** keyword. You can keep label clause before or after **enter** clause.

A while loop label is used just to name the loop block and label names must be double quoted constant strings having single or multiple words. The label keyword is a case sensitive and should be written as **label**.

Syntax

The syntax of a while loop with label clause is as follows:

```
while expression label "Label Name" do

   -- Statements executed if expression returns true

end while
```

The labels are very useful when you use nested while loops. You can use **continue** or **exit** loop control statements with label names to control the flow of loops.

Example

```
#!/home/euphoria-4.0b2/bin/eui
```

```
integer a = 10
integer b = 20

while a < 20 label "OUTER" do
   printf(1, "value of a : %d\n", a)
   a = a + 1
   while b < 30 label "INNER" do
      printf(1, "value of b : %d\n", b)
      b = b + 1
      if b > 25 then
         continue "OUTER"  -- go to start of OUTER loop
      end if
   end while
end while
```

This produces the following result:

```
value of a : 10
value of b : 20
value of b : 21
value of b : 22
value of b : 23
value of b : 24
value of b : 25
value of a : 11
value of b : 26
value of a : 12
value of b : 27

value of a : 13

value of b : 28

value of a : 14

value of b : 29

value of a : 15

value of a : 16

value of a : 17

value of a : 18

value of a : 19
```

A **loop...until** loop is similar to a while loop, except that a loop...until loop is guaranteed to execute at least one time.

Syntax

The syntax of a loop...until is as follows:

```
loop do

   -- Statements to be executed.

until expression
```

Notice that the expression appears at the end of the loop, hence the statements in the loop execute once before the expression's value is tested.

If the expression returns true, the flow of control jumps back up to do, and the statements in the loop execute again. This process repeats until the expression is false.

Example

```
#!/home/euphoria-4.0b2/bin/eui



integer a = 10

```

```
loop do
   printf(1, "value of a : %d\n", a)
   a = a + 1
until a < 20
```

This produces the following result:

```
value of a : 10
value of a : 11
value of a : 12
value of a : 13
value of a : 14
value of a : 15
value of a : 16
value of a : 17
value of a : 18
value of a : 19
```

The *loop....with entry* Statement

It is often the case that the first iteration of a loop is somehow special. Some things have to be done before the loop starts. They are done before the statement starting the loop.

The **with entry** statement serves the purpose very well. You need to use this statement with loop...until and just add the **entry** keyword at the point you wish the first iteration starts.

Syntax

The syntax of a loop...until loop with entry is as follows:

```
loop with entry do
   -- Statements to be executed.
entry
```

```
   -- Initialisation statements.

until expression
```

Before executing the *expression*, it executes initialization statements and then it starts as a normal loop. Later, these initialization statements become part of loop body.

Example

```
#!/home/euphoria-4.0b2/bin/eui



integer a = 10



loop with entry do

   printf(1, "value of a : %d\n", a)

   a = a + 1

entry

   a = a + 2

until a > 20
```

This produces the following result:

```
value of a : 12

value of a : 15

value of a : 18
```

The *loop....label* Statement

A **loop...until** loop can have a **label** clause just before the first **do** keyword. You can keep label clause before or after **enter** clause.

This label is used just to name the loop block and label names must be double quoted constant strings having single or multiple words. The label keyword is a case sensitive and should be written as **label**.

Syntax

The syntax of a loop...until with label clause is as follows:

```
loop label "Label Name" do

   -- Statements to be executed.

until expression
```

The labels are very useful when you use nested loops. You can use **continue** or **exit** loop control statements with label names to control the flow of loops.

Example

```
#!/home/euphoria-4.0b2/bin/eui



integer a = 10

integer b = 20



loop label "OUTER" do

   printf(1, "value of a : %d\n", a)

   a = a + 1

   loop label "INNER" do

      printf(1, "value of b : %d\n", b)

      b = b + 1

      if b > 25 then

         continue "OUTER"  -- go to start of OUTER loop

      end if

   until b > 30

until a > 20
```

This produces the following result:

```
value of a : 10
value of b : 20

value of b : 21

value of b : 22

value of b : 23

value of b : 24

value of b : 25

value of a : 11

value of b : 26

value of a : 12

value of b : 27

value of a : 13

value of b : 28

value of a : 14

value of b : 29

value of a : 15

value of a : 16

value of a : 17

value of a : 18

value of a : 19
```

Note: The above example should work as explained, but looks like Euphoria interpreter has some problem and it is working as expected, may be it would be fixed in future versions of Euphoria.

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

A for statement sets up a special loop that has its own loop variable. The loop variable starts with the specified initial value and increments or decrements it to the specified final value.

A for loop is useful when you know the exact number of times a task is required to be repeated.

Syntax

The syntax of a for loop is as follows:

```
for "initial value" to "last value" by "inremental value" do

   -- Statements to be executed.

end for
```

Here, you initialize the value of a variable and then body of the loop is executed. After every iteration, variable value is increased by the given incremental value. The last value of the variable is checked and if it is reached, then loop is terminated.

The initial value, last value, and increment must all be atoms. If no increment is specified then +1 is assumed.

The *for* loop does not support *with entry* statement.

Example

```
#!/home/euphoria-4.0b2/bin/eui



for a = 1 to 6 do

   printf(1, "value of a %d\n", a)

end for
```

This produces the following result:

```
value of a 1

value of a 2

value of a 3

value of a 4

value of a 5

value of a 6
```

The loop variable is declared automatically. It exists until the end of the loop. The variable has no value outside of the loop and is not even declared. If you need its final value, you need to copy it into another variable before leaving the loop.

Here is one more example with incremental value:

```
#!/home/euphoria-4.0b2/bin/eui


for a = 1.0 to 6.0  by 0.5 do

   printf(1, "value of a %f\n", a)

end for
```

This produces the following result:

```
value of a 1.000000

value of a 1.500000

value of a 2.000000

value of a 2.500000

value of a 3.000000

value of a 3.500000

value of a 4.000000

value of a 4.500000

value of a 5.000000

value of a 5.500000

value of a 6.000000
```

10. EUPHORIA FLOW CONTROL

Program execution flow refers to the order in which program statements get executed. By default the statements get executed one after another.

However; many times the order of execution needs to be altered from the default order, to get the task done.

Euphoria has a number of *flow control* statements that you can use to arrange the execution order of statements.

The *exit* Statement

Exiting a loop is done with the keyword **exit**. This causes flow to immediately leave the current loop and recommence with the first statement after the end of the loop.

Syntax

The syntax of an exit statement is as follows:

```
exit [ "Label Name" ] [Number]
```

The **exit** statement terminates the latest and innermost loop until an optional label name or number is specified.

A special form of **exit N** is **exit 0**. This leaves all levels of loop, regardless of the depth. Control continues after the outermost loop block. Likewise, exit -1 exits the second outermost loop, and so on.

Example

```
#!/home/euphoria-4.0b2/bin/eui


integer b


for a = 1 to 16 do

   printf(1, "value of a %d\n", a)

   if a = 10 then
```

```
      b = a
      exit
   end if
end for
printf(1, "value of b %d\n", b)
```

This produces the following result:

```
value of a 1
value of a 2
value of a 3
value of a 4
value of a 5
value of a 6
value of a 7
value of a 8
value of a 9
value of a 10
value of b 10
```

The *break* Statement

The **break** statement works exactly like the **exit** statement, but applies to if statements or switch statements rather than to loop statements of any kind.

Syntax

The syntax of break statement is as follows:

```
break [ "Label Name" ] [Number]
```

The **break** statement terminates the latest and innermost if or switch block until an optional label name or number is specified.

A special form of **break N** is **break 0**. This leaves the outer most if or switch block, regardless of the depth. Control continues after the outermost block. Likewise, break -1 breaks the second outermost if or switch block, and so on.

Example

```
#!/home/euphoria-4.0b2/bin/eui


integer a, b

sequence s = {'E','u', 'p'}


if s[1] = 'E' then

   a = 3

   if s[2] = 'u' then

      b = 1

      if s[3] = 'p' then

         break 0 -- leave topmost if block

      end if

      a = 2

   else

      b = 4

   end if

else

   a = 0

   b = 0

end if

printf(1, "value of a %d\n", a)

printf(1, "value of b %d\n", b)
```

This produces the following result:

```
value of a 3

value of b 1
```

The *continue* Statement

The **continue** statement continues execution of the loop it applies to by going to the next iteration and skipping the rest of an iteration.

Going to the next iteration means testing a condition variable index and checking whether it is still within bounds.

Syntax

The syntax of continue statement is as follows:

```
continue [ "Label Name" ] [Number]
```

The **continue** statement would re-iterate the latest and inner most loop until an optional label name or number is specified.

A special form of **continue N** is **continue 0**. This re-iterate the outer most loop, regardless of the depth. Likewise, continue -1 starts from the second outermost loop, and so on.

Example

```
#!/home/euphoria-4.0b2/bin/eui



for a = 3 to 6 do

   printf(1, "value of a %d\n", a)

   if a = 4 then

      puts(1,"(2)\n")

      continue

   end if

   printf(1, "value of a %d\n", a*a)

end for

This would produce following result:
```

```
value of a 3

value of a 9

value of a 4

(2)

value of a 5

value of a 25

value of a 6

value of a 36
```

The *retry* Statement

The **retry** statement continues execution of the loop it applies to by going to the next iteration and skipping the rest of an iteration.

Syntax

The syntax of retry statement is as follows:

```
retry [ "Label Name" ] [Number]
```

The **retry** statement retries executing the current iteration of the loop it applies to. The statement branches to the first statement of the designated loop neither testing anything nor incrementing the for loop index.

A special form of **retry N** is **retry 0**. This retries executing the outer most loop, regardless of the depth. Likewise, retry -1 retries the second outermost loop, and so on.

Normally, a sub-block which contains a retry statement also contains another flow control keyword like exit, continue, or break. Otherwise, the iteration would be endlessly executed.

Example

```
#!/home/euphoria-4.0b2/bin/eui



integer errors = 0

integer files_to_open = 10
```

```
for i = 1 to length(files_to_open) do
   fh = open(files_to_open[i], "rb")
   if fh = -1 then
      if errors > 5 then
         exit
      else
         errors += 1
         retry
      end if
   end if
   file_handles[i] = fh
end for
```

Since retry does not change the value of i and tries again opening the same file, there has to be a way to break from the loop, which the exit statement provides.

The *goto* Statement

The **goto** statement instructs the computer to resume code execution at a labeled place.

The place to resume execution is called the target of the statement. It is restricted to lie in the current routine, or the current file if outside any routine.

Syntax

The syntax of goto statement is as follows:

```
goto "Label Name"
```

The target of a goto statement can be any accessible **label** statement:

```
label "Label Name"
```

Label names must be double quoted constant strings. Characters that are illegal in Euphoria identifiers may appear in a label name, since it is a regular string.

Example

```
#!/home/euphoria-4.0b2/bin/eui


integer a = 0


label "FIRST"

printf(1, "value of a %d\n", a)

a += 10

if a < 50 then

   goto "FIRST"

end if

printf(1, "Final value of a %d\n", a)
```

This produces the following result:

```
value of a 0

value of a 10

value of a 20

value of a 30

value of a 40

Final value of a 50
```

11. SHORT CIRCUIT EVALUATION

When a condition is tested by **if, elsif, until**, or **while** using **and** or **or** operators, a short-circuit evaluation is used. For example:

```
if a < 0 and b > 0 then

   -- block of code

end if
```

If a < 0 is false, then Euphoria does not bother to test if b is greater than 0. It knows that the overall result is false regardless. Similarly:

```
if a < 0 or b > 0 then

   -- block of code

end if
```

if a < 0 is true, then Euphoria immediately decides that the result true, without testing the value of b, since the result of this test is irrelevant.

In General, whenever you have a condition of the following form:

```
A and B
```

Where A and B can be any two expressions, Euphoria takes a short-cut when A is false and immediately makes the overall result false, without even looking at expression B.

Similarly, whenever you have a condition of the following form:

```
A or B
```

Where A is true, Euphoria skips the evaluation of expression B, and declares the result to be true.

Short-circuit evaluation of *and* and *or* takes place for if, elsif, until, and while conditions only. It is not used in other contexts. For example:

```
x = 1 or {1,2,3,4,5} -- x should be set to {1,1,1,1,1}
```

If short-circuiting were used here, you would set x to 1, and not even look at {1,2,3,4,5}, which would be wrong.

Thus, short-circuiting can be used in if, elsif, until, or while conditions, because you need to only care if the result is true or false, and conditions are required to produce an atom as a result.

12. EUPHORIA SEQUENCES

A sequence is represented by a list of objects in brace brackets { }, separated by commas. A sequence can contain both atoms and other sequences. For example:

```
{2, 3, 5, 7, 11, 13, 17, 19}

{1, 2, {3, 3, 3}, 4, {5, {6}}}

{{"Zara", "Ayan"}, 52389, 97.25}

{} -- the 0-element sequence
```

A single element of a sequence may be selected by giving the element number in square brackets. Element numbers start at 1.

For example, if x contains {5, 7.2, 9, 0.5, 13} then x[2] is 7.2.

Suppose x[2] contains {11,22,33}, Now if you ask for x[2] you get {11,22,33} and if you ask for x[2][3], you get the atom 33.

Example

```
#!/home/euphoria-4.0b2/bin/eui



sequence x

x = {1, 2, 3, 4}



for a = 1 to length(x) do

   printf(1, "value of x[%d] = %d\n", {a, x[a]})

end for
```

Here, length() is the built-in function which returns length of the sequence. The above example produces the following result:

```
value of x[1] = 1

value of x[2] = 2
value of x[3] = 3

value of x[4] = 4
```

Character String

A character string is just a **sequence** of characters. It may be entered in one of the two ways:

(a) Using Double Quotes

```
"ABCDEFG"
```

(b) Using Raw String Notation

```
-- Using back-quotes

`ABCDEFG`



or



-- Using three double-quotes

"""ABCDEFG"""
```

You can try the following example to understand the concept:

```
#!/home/euphoria-4.0b2/bin/eui



sequence x

x = "ABCD"



for a = 1 to length(x) do

   printf(1, "value of x[%d] = %s\n", {a, x[a]})

end for
```

This produces the following result:

```
value of x[1] = A

value of x[2] = B

value of x[3] = C

value of x[4] = D
```

String Arrays

An array of strings can be implemented using Sequences as follows:

```
#!/home/euphoria-4.0b2/bin/eui


sequence x = {"Hello", "World", "Euphoria", "", "Last One"}


for a = 1 to length(x) do

   printf(1, "value of x[%d] = %s\n", {a, x[a]})

end for
```

This produces the following result:

```
value of x[1] = Hello

value of x[2] = World

value of x[3] = Euphoria

value of x[4] =

value of x[5] = Last One
```

Euphoria Structures

A structure can be implemented using Sequences as follows:

```
#!/home/euphoria-4.0b2/bin/eui


sequence employee = {
   {"John","Smith"},

    45000,

    27,

    185.5

   }


printf(1, "First Name = %s, Last Name = %s\n",

                 {employee[1][1],employee[1][2]} )
```

This produces the following result:

```
First Name = John, Last Name = Smith
```

There are various operations which can be performed directly on sequences. Let us see them in detail:

Urinary Operation

When applied to a sequence, a unary operator is actually applied to each element in the sequence to yield a sequence of results of the same length.

```
#!/home/euphoria-4.0b2/bin/eui



sequence x

x = -{1, 2, 3, 4}



for a = 1 to length(x) do

   printf(1, "value of x[%d] = %d\n", {a, x[a]})

end for
```

This produces the following result:

```
value of x[1] = -1
value of x[2] = -2

value of x[3] = -3

value of x[4] = -4
```

Arithmetic Operations

Almost all arithmetic operations can be performed on sequences as follows:

```
#!/home/euphoria-4.0b2/bin/eui



sequence x, y, a, b, c

x = {1, 2, 3}

y = {10, 20, 30}



a = x + y

puts(1, "Value of a = {")

for i = 1 to length(a) do

   printf(1, "%d,", a[i])

end for

puts(1, "}\n")



b = x - y

puts(1, "Value of b = {")

for i = 1 to length(a) do

   printf(1, "%d,", b[i])

end for

puts(1, "}\n")



c = x * 3
```

```
puts(1, "Value of c = {")

for i = 1 to length(c) do

   printf(1, "%d,", c[i])

end for

puts(1, "}\n")
```

This produces the following result:

```
Value of a = {11,22,33,}

Value of b = {-9,-18,-27,}

Value of c = {3,6,9,}
```

Command Line Options

A user can pass command line options to a Euphoria script and it can be accessed as a sequence using **command_line()** function as follows:

```
#!/home/euphoria-4.0b2/bin/eui



sequence x



x = command_line()



printf(1, "Interpeter Name: %s\n", {x[1]} )



printf(1, "Script Name: %s\n", {x[2]} )



printf(1, "First Argument: %s\n", {x[3]})



printf(1, "Second Argument: %s\n", {x[4]})
```

Here **printf()** is Euphoria's built-in function. Now if you run this script as follows:

```
$eui test.ex "one" "two"
```

This produces the following result:

```
Interpeter Name: /home/euphoria-4.0b2/bin/eui

Script Name: test.ex

First Argument: one

Second Argument: two
```

13. EUPHORIA DATE AND TIME

Euphoria has a library routine that returns the date and time to your program.

The *date()* Method

The date() method returns a sequence value composed of eight atom elements. The following example explains it in detail:

```
#!/home/euphoria-4.0b2/bin/eui


integer curr_year, curr_day, curr_day_of_year, curr_hour,

        curr_minute, curr_second

sequence system_date, word_week, word_month, notation,

        curr_day_of_week, curr_month

        word_week = {"Sunday",

                     "Monday",

                     "Tuesday",

                     "Wednesday",

                     "Thursday",

                     "Friday",

                     "Saturday"}

        word_month = {"January", "February",

                      "March", "April", "May",

                      "June", "July", "August",

                      "September", "October",

                      "November", "December"}

-- Get current system date.
```

```
system_date = date()


-- Now take individual elements

curr_year = system_date[1] + 1900

curr_month = word_month[system_date[2]]

curr_day = system_date[3]

curr_hour = system_date[4]

curr_minute = system_date[5]

curr_second = system_date[6]

curr_day_of_week = word_week[system_date[7]]

curr_day_of_year = system_date[8]


if curr_hour >= 12 then

   notation = "p.m."

else

   notation = "a.m."

end if


if curr_hour > 12 then

   curr_hour = curr_hour - 12

end if

if curr_hour = 0 then

   curr_hour = 12

end if


puts(1, "\nHello Euphoria!\n\n")

printf(1, "Today is %s, %s %d, %d.\n",
         {curr_day_of_week, curr_month,

          curr_day, curr_year})


printf(1, "The time is %.2d:%.2d:%.2d %s\n",

         {curr_hour, curr_minute,

          curr_second, notation})


printf(1, "It is %3d days into the current year.\n",

         {curr_day_of_year})
```

This produces the following result on your standard screen:

```
Hello Euphoria!


Today is Friday, January 22, 2010.

The time is 02:54:58 p.m.

It is  22 days into the current year.
```

The *time()* Method

The time() method returns an atom value, representing the number of seconds elapsed since a fixed point in time. The following example explains it in detail:

```
#!/home/euphoria-4.0b2/bin/eui


constant ITERATIONS = 100000000

integer p

atom t0, t1, loop_overhead


t0 = time()
```

```
for i = 1 to ITERATIONS do

    -- time an empty loop

end for


loop_overhead = time() - t0


printf(1, "Loop overhead:%d\n", loop_overhead)


t0 = time()

for i = 1 to ITERATIONS do

    p = power(2, 20)

end for


t1 = (time() - (t0 + loop_overhead))/ITERATIONS


printf(1, "Time (in seconds) for one call to power:%d\n", t1)
```

This produces the following result:

```
Loop overhead:1

Time (in seconds) for one call to power:0
```

Date & Time Related Methods

Euphoria provides a list of methods which helps you in manipulating date and time. These methods are listed in Euphoria Library Routines.

14. EUPHORIA PROCEDURES

A procedure is a group of reusable code which can be called from anywhere in your program. This eliminates the need of writing same code again and again. This helps programmers to write modular code.

Like any other advance programming language, Euphoria also supports all the features necessary to write modular code using procedures.

You must have seen procedures like *printf()* and *length()* in previous chapters. We are using these procedure again and again but they have been written in core Euphoria only once.

Euphoria allows you to write your own procedures as well. This section explains how to write your own procedure in Euphoria.

Procedure Definition

Before you use a procedure, you need to define it. The most common way to define a procedure in Euphoria is by using the **procedure** keyword, followed by a unique procedure name, a list of parameters (that might be empty), and a statement block which ends with **end procedure** statement. The basic syntax is as shown below:

```
procedure procedurename(parameter-list)


  statements

  ..........


end procedure
```

Example

A simple procedure called sayHello that takes no parameters is defined here –

```
procedure  sayHello()

   puts(1, "Hello there")

end procedure
```

Calling a Procedure

To invoke a procedure somewhere later in the script, you simply need to write the name of that procedure as follows:

```
#!/home/euphoria-4.0b2/bin/eui


procedure  sayHello()

   puts(1, "Hello there")

end procedure


-- Call above defined procedure.

sayHello()
```

This produces the following result:

```
Hello there
```

Procedure Parameters

Till now you have seen procedure without a parameter. But there is a facility to pass different parameters while calling a procedure. These passed parameters can be captured inside the procedure and any manipulation can be done over those parameters.

A procedure can take multiple parameters separated by comma.

Example

Let us do a bit modification in our *sayHello* procedure. This time it takes two parameters:

```
#!/home/euphoria-4.0b2/bin/eui


procedure sayHello(sequence name,atom  age)

   printf(1, "%s is %d years old.", {name, age})

end procedure
```

```
-- Call above defined procedure.

sayHello("zara", 8)
```

This produces the following result:

```
zara is 8 years old.
```

15. EUPHORIA FUNCTIONS

Euphoria functions are just like procedures, but they return a value, and can be used in an expression. This chapter explains how to write your own functions in Euphoria.

Function Definition

Before we use a function we need to define it. The most common way to define a function in Euphoria is by using the **function** keyword, followed by a unique function name, a list of parameters (that might be empty), and a statement block which ends with**end function** statement. The basic syntax is shown below:

```
function functionname(parameter-list)



  statements

  ..........

  return [Euphoria Object]



end function
```

Example

A simple function called sayHello that takes no parameters is defined here:

```
function sayHello()

   puts(1, "Hello there")

   return 1

end function
```

Calling a Function

To invoke a function somewhere later in the script, you would simple need to write the name of that function as follows:

```
#!/home/euphoria-4.0b2/bin/eui


function sayHello()
   puts(1, "Hello there")
   return 1
end function


-- Call above defined function.
sayHello()
```

This produces the following result:

```
Hello there
```

Function Parameters

Till now we have seen function without a parameters. But there is a facility to pass different parameters while calling a function. These passed parameters can be captured inside the function and any manipulation can be done over those parameters.

A function can take multiple parameters separated by comma.

Example

Let us do a bit modification in our *sayHello* function. This time it takes two parameters:

```
#!/home/euphoria-4.0b2/bin/eui


function sayHello(sequence name,atom  age)
   printf(1, "%s is %d years old.", {name, age})
   return 1
end function

```

```
-- Call above defined function.

sayHello("zara", 8)
```

This produces the following result:

```
zara is 8 years old.
```

The *return* Statement

A Euphoria function must have *return* statement before closing statement **end function**. Any Euphoria object can be returned. You can, in effect, have multiple return values, by returning a sequence of objects. For example:

```
return {x_pos, y_pos}
```

If you have nothing to return, then simply return 1 or 0. The return value 1 indicates success and 0 indicates failure.

16. EUPHORIA FILE I/O

Using Euphoria programming language, you can write programs that read and change file data on your floppy drive or hard drive, or create new files as a form of output. You can even access devices on your computer such as the printer and modem.

This chapter described all the basic I/O functions available in Euphoria. For information on more functions, please refer to standard Euphoria documentation.

Displaying on the Screen

The simplest way to produce output is using the *puts()* statement where you can pass any string to be displayed on the screen. There is another method *printf()* which can also be used in case you have to format a string using dynamic values.

These methods convert the expressions you pass them to a string and write the result to standard output as follows:

```
#!/home/euphoria-4.0b2/bin/eui


puts(1, "Euphoria is really a great language, isn't it?" )
```

This produces the following result on your standard screen:

```
Euphoria is really a great language, isn't it?
```

Opening and Closing Files

Euphoria provides basic methods necessary to manipulate files by default. You can do your most of the file manipulation using the following methods:

- open()
- close()
- printf()
- gets()
- getc()

The *open* Method

Before you can read or write a file, you have to open it using Euphoria's built-in *open()*method. This function creates a file descriptor which is utilized to call other supporting methods associated with it.

Syntax

```
integer file_num = open(file_name, access_mode)
```

Above method returns -1 in case there is an error in opening the given file name. Here are the parameters:

- **file_name:** The file_name argument is a string value that contains the name of the file that you want to access.

- **access_mode:** The access_mode determines the mode in which the file has to be opened. For example, read, write append, etc. A complete list of possible values for file opening modes is given in the following table:

Modes	Description
r	Opens a text file for reading only. The file pointer is placed at the beginning of the file.
rb	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file.
w	Opens a text file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
u	Opens a file for both reading and writing. The file pointer is set at the beginning of the file.
ub	Opens a file for both reading and writing in binary format. The file pointer is placed at the beginning of the file.

a	Opens a file for appending. The file pointer is at the end of the file if the file exists (append mode). If the file does not exist, it creates a new file for writing.
ab	Opens a file for appending in binary format. The file pointer is at the end of the file if the file exists (append mode). If the file does not exist, it creates a new file for writing.

Example

The following example creates a new text file in the current directory on your Linux system:

```
#!/home/euphoria-4.0b2/bin/eui



integer file_num

constant ERROR = 2

constant STDOUT = 1



file_num = open("myfile,txt", "w")

if file_num = -1 then

    puts(ERROR, "couldn't open myfile\n")

else

     puts(STDOUT, "File opend successfully\n")

end if
```

If file opens successfully, then it "myfile.txt" is created in your current directory and produces the following result:

```
File opend successfully
```

The *close()* Method

The close() method flushes any unwritten information and closes the file, after which no more reading or writing can be done on the file.

Euphoria automatically closes a file when the reference object of a file is reassigned to another file. It is a good practice to use the close() method to close a file.

Syntax

```
close( file_num );
```

Here the file descriptor received while opening a file is passed as a parameter.

Example

The following example creates a file as above and then closes it before existing the program:

```
#!/home/euphoria-4.0b2/bin/eui


integer file_num

constant ERROR = 2

constant STDOUT = 1


file_num = open("myfile.txt", "w")

if file_num = -1 then

    puts(ERROR, "couldn't open myfile\n")

else

     puts(STDOUT, "File opend successfully\n")

end if


if file_num = -1 then

    puts(ERROR, "No need to close the file\n")

else

     close( file_num )

     puts(STDOUT, "File closed successfully\n")

end if
```

This produces the following result:

```
File opend successfully

File closed successfully
```

Reading and Writing Files

Euphoria provides a set of access methods to make our lives easier while reading or writing a file either in text mode or binary mode. Let us see how to use *printf()* and *gets()*methods to read and write files.

The *printf()* Method

The *printf()* method writes any string to an open file.

Syntax

```
printf(fn, st, x)
```

Here are the parameters:

- **fn:** File descriptor received from open() method.
- **st:** Format string where decimal or atom is formatted using %d and string or sequence is formatted using %s.
- **x:** If x is a sequence, then format specifiers from st are matched with corresponding elements of x. If x is an atom, then normally st contains just one format specifier and it is applied to x. However; if st contains multiple format specifiers, then each one is applied to the same value x.

Example

The following example opens a file and writes the name and age of a person in this file:

```
#!/home/euphoria-4.0b2/bin/eui



integer file_num

constant ERROR = 2

constant STDOUT = 1
```

```
file_num = open("myfile.txt", "w")

if file_num = -1 then

    puts(ERROR, "couldn't open myfile\n")

else

     puts(STDOUT, "File opend successfully\n")

end if


printf(file_num, "My name is %s and age is %d\n", {"Zara", 8})


if file_num = -1 then

    puts(ERROR, "No need to close the file\n")

else

     close( file_num )

     puts(STDOUT, "File closed successfully\n")

end if
```

The above example creates *myfile.txt* file. Is writes given content in that file and finally closes. If you open this file, it would have the following content:

```
My name is Zara and age is 8
```

The *gets()* Method

The *gets()* method reads a string from an open file.

Syntax

```
gets(file_num)
```

Here passed parameter is file description return by the *opend()* method. This method starts reading from the beginning of the file line by line. The characters have values from 0 to 255. The atom -1 is returned on end of file.

Example

Let us take a file *myfile.txt* which is already created.

```
#!/home/euphoria-4.0b2/bin/eui


integer file_num

object line


constant ERROR = 2

constant STDOUT = 1


file_num = open("myfile.txt", "r")

if file_num = -1 then

    puts(ERROR, "couldn't open myfile\n")

else

     puts(STDOUT, "File opend successfully\n")

end if


line = gets(file_num)


printf( STDOUT, "Read content : %s\n", {line})


if file_num = -1 then

    puts(ERROR, "No need to close the file\n")

else

     close( file_num )

     puts(STDOUT, "File closed successfully\n")

end if
```

This produces the following result:

```
File opend successfully

Read content : My name is Zara and age is 8



File closed successfully
```

File & Directory Related Methods

Euphoria provides a list of many methods which helps you in manipulating files. These methods are listed in Euphoria Library Routines.