

How to Program Computers

By Stephen C. Baxter

Table of Contents

BEGINNER LEVEL

What Is a Program?

How To Run a Program

Console and DOS32 Programs

Windows and Other GUI Programs

A Program

Statements and Subroutines

Comments

Displaying Words and Numbers

Targeting Parts of the Screen

Variables

Variable Names

Storing Data in Memory
Variable Types
Declaring Variables
Counter Variables in for...do
Assigning Values to Variables
Initializing Variables
Calculations
Accumulations with Variables
The remainder Function
Precedence of Operations
Math Functions
Expressions
Entering Data with Prompt Functions
Sequences of Variables
Sequences Hold Arrays of Variables
Specifying Sequence Elements
Logical Relations in Decision-Making
Relational Operators and Functions
Boolean Expressions
Compound Expressions
Using Logical Statements to Control Program Flow
Repeating Program Operations
The for...do Loop
The while Loop
Exiting Loops
Library Files
Simple Program Example
Debugging
Editors

INTERMEDIATE LEVEL

More on Sequences
Comparing Strings
Searching for Strings
Numbers and Strings
Changing Case
Changing Strings
Combining Strings
More Data Types
Testing for Type
Constants
Writing Subroutines
The type Routine
Passing Parameters
Files and I/O
Control Flow Structures

Scope

ADVANCED LEVEL

Databases

The EDS Database

File Servers

Linking to DLL's

Advantage Database Server

MySQL

Program Planning

Multitasking Programming

Translating Euphoria to C

Web Programming with CGI

The Software Business

EXAMPLES

BlackJack Program

Employees Program

Windows Programs

What Is A Program?

"A program is a sequence of instructions to the computer in a language both you and the computer understand"

-- Microsoft Corporation

"The goal of any program is to perform some useful job, such as word processing, bookkeeping or playing a game."

--Microsoft Corporation

"The process is part science, part art. The science comes from reading books about programming; the art comes from writing your own programs and analyzing the programs of others"

--Microsoft Corporation

Programs may be written in text editors, word processors, or Integrated Development Environments (IDE). Text used to write programs is called "source code" or just "code" for short, and Euphoria code is saved in files with *.ex* (DOS), *.exw* (Windows), or *.exu* (Linux/Unix) file extensions. The characters and punctuation you find in text files are used in program code files, though the meanings may be alien until you become familiar with the language. Except for this and the file extension, these files resemble text files in all other ways, and you should be able to read source code in any text editor. To run a program, the text file that is the source code must be read by language software that can translate or interpret the code you have written or otherwise spot the errors in your writing.

Though your program may be 99% correct and free of typographical errors, a typical language compiler will instantly find the missing parenthesis and point it out. It will not miss any typographical errors. Once the typographical errors are eliminated, the program may still fail due to errors in logic. Yes, the syntax may be perfect, but that does not guarantee that the logic is flawless.

NOTE: Syntax means the rules and form of language.

You will be frustrated at first with the typographical errors that abound, but you will quickly improve with practice. It helps if you are a typist, but it is not required.

Here is an example of the source code for a simple program:

```
~~~~~  
sequence list, sorted_list  
  
function merge_sort(sequence x)  
-- put x into ascending order using a recursive merge sort  
integer n, mid  
sequence merged, a, b  
  
n = length(x)  
if n = 0 or n = 1 then  
    return x -- trivial case  
end if  
  
mid = floor(n/2)  
a = merge_sort(x[1..mid])      -- sort first half of x  
b = merge_sort(x[mid+1..n])   -- sort second half of x  
  
-- merge the two sorted halves into one  
merged = {}  
while length(a) > 0 and length(b) > 0 do  
    if compare(a[1], b[1]) < 0 then  
        merged = append(merged, a[1])  
        a = a[2..length(a)]  
    else  
        merged = append(merged, b[1])  
        b = b[2..length(b)]  
    end if  
end while  
return merged & a & b -- merged data plus leftovers  
end function  
  
procedure print_sorted_list()  
-- generate sorted_list from list  
list = {9, 10, 3, 1, 4, 5, 8, 7, 6, 2}  
sorted_list = merge_sort(list)  
? sorted_list  
end procedure  
  
print_sorted_list() -- this command starts the program
```

The program begins execution at the last line, because it compiles or interprets from the top down. In this case, the last line of code calls the main procedure, **print_sorted_list()**. Execution then proceeds through that procedure line by line until the procedure is finished. The Euphoria language takes advantage of this to create a one-pass translation method that is very quick and efficient. Those items on top are interpreted first followed by the lines that follow in order.

How to Run a Program

NOTE: The console is a text-entry window, often called the DOS window, that has a command line where you run the computer by typing and reading text.

The fact that you are reading this shows that you know how to run a program, but there are also other ways, and a programmer should know about them. Beginners often write programs for the console, so running a program can require invoking the program's name on the command line. For interpreted programs, it also involves invoking the Euphoria interpreter -- see below.

In Euphoria, the source code is offered as a target file to the Euphoria interpreter, *ex.exe* for DOS32 and *exw.exe* for Windows. The interpreter reads, interprets, and executes the code with such speed that it seems nearly instantaneous. We will show why this is a benefit to programmers. On a console command line, the command might resemble the following example:

```
ex myprog  
  
-- or --  
  
ex.exe myprog.ex
```

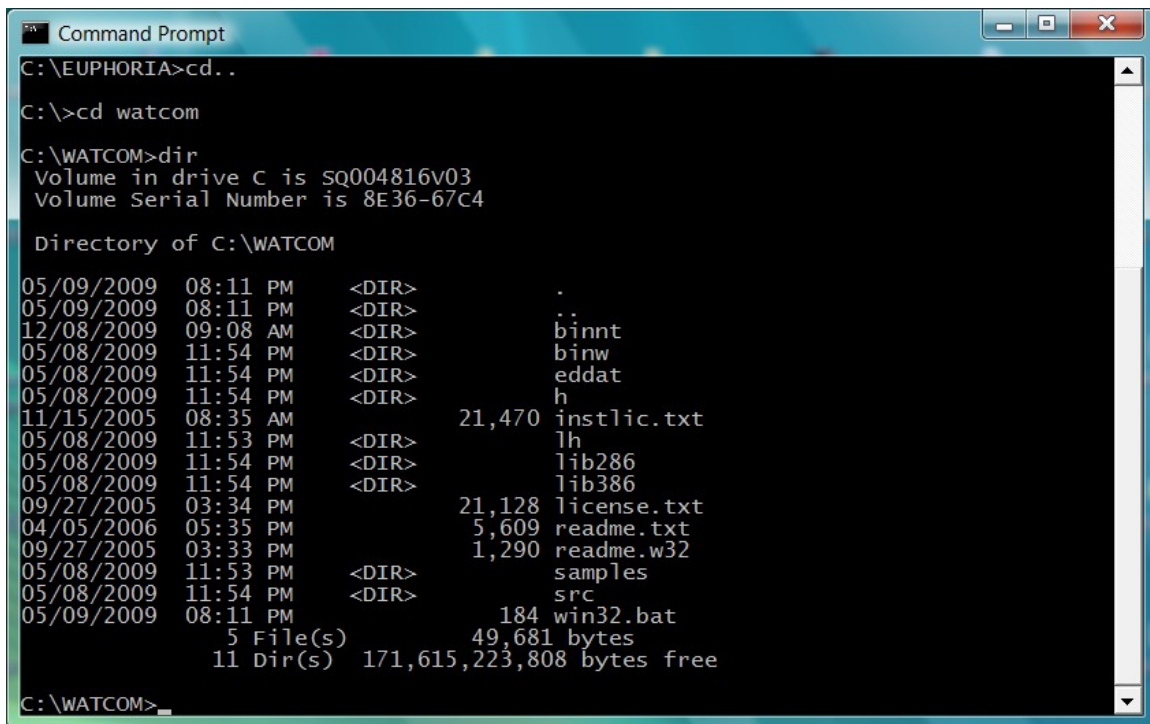
The console command line recognizes both forms. Basically, you are running the *ex.exe* program and offering the name of the source code "myprog" as a run time parameter. If you simply run *ex.exe* by typing *ex* only, the interpreter program will immediately ask for the source code to run with the question, "file name to execute?". If your Euphoria package is installed correctly, you can also click on source files with a mouse and Windows will run them with the appropriate interpreter, and you might even be able to run programs from the command line by typing the full program name alone, such as *myprog.ex*. For a large project such as a professional system installation, you can even use batch files with a *.bat* extension. The batch file may be simple or complex. For instance, it may contain "ex myprog". Such a batch file would be called *myprog.bat*.

Using the Euphoria binder (which is like a compiler), you can convert your source code to a stand-alone *.exe* file, like most programs. Such programs will run simply by invoking them as any other program you have ever used, and the files can be distributed without also distributing the interpreter, *but interpreted programs are the best kind while you are in the development stage*. They make development easier and faster. The ability to run in interpreted mode is one of the advantages that Euphoria has over most other programming languages

With other languages, you must go through several steps to get a program to run just to test it as you write it. With Euphoria, you can run it immediately after making changes.

You can quickly test any and all changes as you make them because it has an interpreter (ex.exe or exw.exe) to work with.

Console or DOS Programming



```
Command Prompt
C:\EUPHORIA>cd .
C:\>cd watcom
C:\WATCOM>dir
Volume in drive C is SQ004816V03
Volume Serial Number is 8E36-67C4

Directory of C:\WATCOM

05/09/2009  08:11 PM    <DIR>          .
05/09/2009  08:11 PM    <DIR>          ..
12/08/2009  09:08 AM    <DIR>          binnt
05/08/2009  11:54 PM    <DIR>          binw
05/08/2009  11:54 PM    <DIR>          eddat
05/08/2009  11:54 PM    <DIR>          h
11/15/2005  08:35 AM             21,470 instlic.txt
05/08/2009  11:53 PM    <DIR>          lh
05/08/2009  11:54 PM    <DIR>          lib286
05/08/2009  11:54 PM    <DIR>          lib386
09/27/2005  03:34 PM             21,128 license.txt
04/05/2006  05:35 PM             5,609 readme.txt
09/27/2005  03:33 PM             1,290 readme.w32
05/08/2009  11:53 PM    <DIR>          samples
05/08/2009  11:54 PM    <DIR>          src
05/09/2009  08:11 PM             184 win32.bat
           5 File(s)          49,681 bytes
          11 Dir(s)  171,615,223,808 bytes free

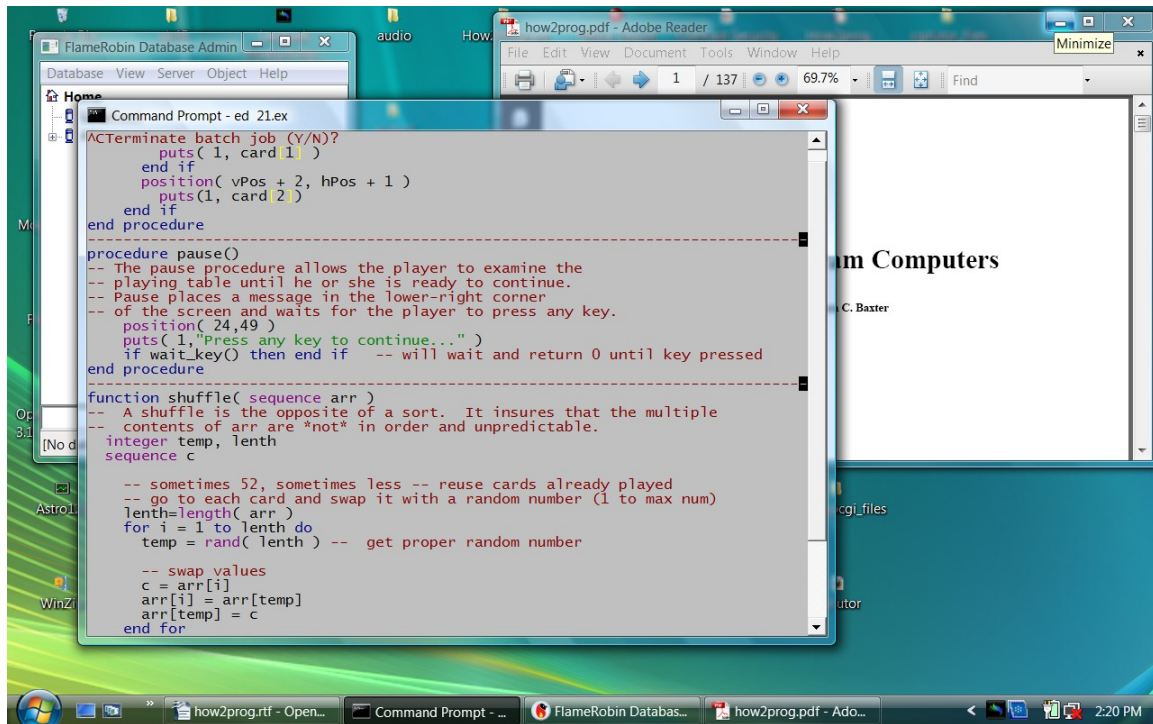
C:\WATCOM>
```

Character-based programming is different from Graphical User Interface (GUI) systems like Windows or Macintosh. Console programs were the norm years ago, and they are still very common in business applications. Typically, the entire screen background is black while the letters and numerals are white.

Beginners do best when they begin writing console programs which are less fancy and which omit many complications. In Windows, a console can be run as one of the windows on the desktop. Windows Vista will not allow full screen console, so a console window is necessary.

MSDOS, which was the most popular operating system before Windows, is a console system, and with Euphoria, you can write DOS32 programs now. Also, Windows allows console programs which can be written with Euphoria. Certain fancy aspects can be added to console programs such as colors and symbols. When you progress to that stage, but console programs allow absolute beginners to start with the pure basics.

Windows and GUI Programs



Windows is the most popular GUI operating system in the world, and more Windows programs are sold than any other. Compared to console or DOS32 programs, Windows and other GUI programs have much extra complexity. There are graphic interface details and many mouse events to deal with.

A Program

Consider the following:

```
puts(1, "Hello, World!")
```

That is a simple one-line program. It is a statement with a print command to print "Hello, World!" to the screen like this:

```
Hello, World!
```

The command **puts()** takes two parameters, the printing destination, 1 means the screen, and the characters to print surrounded by quotes, **Hello, World!**.

Statements and Routines

A program starts as a text file with lines of text. Each line may be a single command to the computer, or a command may span several lines.

Some commands are built-in subroutines or subprograms that end with parentheses. Some subroutines use empty parentheses and some do not.

Some commands are block commands that are meant to spread over two or more lines of text. Block commands have a beginning part and an ending part and allow blank lines in the middle. The **if..then..end if** is a good example. The **if** word marks the beginning, and the **end if** phrase marks the end.

```
if age < 18 then
  puts(1, "Beat it, kid!\n")
else
  puts(1, "What'll ya have?\n")
end if
```

The **for..do** statement is another block statement. It allows a certain number of repetitions based upon the desire of the programmer and the design of the statement. All internal statements contained between **for** and **end for** are repeated a certain number of times. The **for** statement counts each repetition and updates the count in a counting variable.

```
for count = 1 to 10 do
  printf(1, "This is number %d\n", count)
end for
```

The above statement is a block statement bounded by **for...do** at the top and **end for** at the bottom. It basically counts to ten. On each count, the count variable is increased by one. On each count, the count is shown by printing the statement contained within the **for** statement.

```
for count = 1 to 10 do -- 'for' and 'to' and 'do' are the built-
    --in commands
    ...

for count = 1 to 10 -- '1' and '10' are the selected upper and
    --lower bounds
    ...

for count = 1 to 10 do -- count is the loop variable automatically
    --declared
```

The **clear_screen()** procedure is a built-in routine that, well, clears the screen. It is a routine, so it contains parentheses, but it is a procedure because it only performs an action, it does not return a value.

```
clear_screen()
```

The **sqrt()** routine is a function that takes one parameter and returns a value. Its purpose is to perform the *square root* function on number values so the parameter is a number to be evaluated and the return value is a number that is the value of the square root of the parameter number. Since functions must be used within a statement, we put this routine inside a **print()** statement so we can see the result on screen.

```
print(1, sqrt(4))
```

...and the result looks like this

```
2
```

In the previous example, 4 was the parameter placed in the parentheses and 2 was the value returned from the function (which **print()** then printed to screen). 2 is, in fact, the square root of 4. Since, **sqrt()** is a function that returns a value, we used another routine, a procedure, to create a complete statement that could stand alone on one line.

Procedures can stand alone on a line as if they are a simple step in the program, but functions must be used in a statement as if they were a variable. To see the **sqrt()** function in action, we put it in a statement in order to print its result.

Comments

Comments are an important part of programming that explain the program step-by-step and remind the programmer why the program was written. It is a common experience that the programmer will forget in as little as a week why the code was written the way it was. Comments document the process and help the programmer recall the purpose and the process. Comments begin with double hyphen and continue to the end of the line. Any line of code can be disabled, 'commented out', for experimental purposes by placing double hyphens at the beginning of the line. This is a useful debugging technique. The interpreter will ignore comments, but you should not.

```
puts(1, "Hello, World\n") -- this part is a comment
```

Displaying Words and Numbers on Screen

The **print()** statement is a way to show numbers and numeric calculations on screen, but the **?** operator is shorthand for the **print()** statement. Simply use the **print()** statement or its shorthand equivalent followed by the number or numeric expression you wish to print to screen.

```
? (12 + 8) / 4 + 1
-- or --
print(1, (12 + 8) / 4 + 1)
```

The output should appear as follows:

```
3.5
```

Use the **clear_screen()** command to clear the screen.

The **puts()** statement is a way to print characters and strings on screen. This statement is a function that requires two parameters. Multiple parameters are always separated by commas. The first parameter is a number code, 1 represents the screen. The second parameter is the character or string to print. Strings and characters must be surrounded by quotes. Any string may include the newline `'\n'` character at the end within the quotation marks. The `"\n"` character is said to be an imbedded command because it causes an action, moving to a new line, but it is not visible in the final printing.

```
puts(1, "This statement is a procedure.\n")
```

The **print()** (?) statement cannot be used for strings, and the **puts()** statement cannot be used for math, but numbers as characters within quotation marks are allowed.

```
puts(1, "236 Elm Street\n")
```

The **printf()** statement is more complicated but more flexible; it can be used for numbers and strings and exact formatting. Use **printf()** whenever you mix math expressions with string expressions. Use **printf()** when you are printing several lines in columns or tables. The **printf()** statement requires three parameters (all separated by commas). The first is a number that is a code for destination, 1 represents the screen. The second parameter is a formatting string that may contain symbols. The symbols correspond to the items to print. They represent the data type of the item(s) to print. The third parameter is an object that may be an integer, an atom or a sequence. A sequence is used when multiple items are being printed in one statement. The number of symbols in the second parameter must

match the number of elements

```
number = 36
printf(1, "The sum of %f and 4 is %f", {number, number + 4})
printf(1, "The square of %f is %f ", {number, number * number})
```

NOTE: Sequences are surrounded by curly braces {} not parentheses.

The **%f** symbol represents floating-point data with the conventional decimal expression. The symbol is a marker imbedded in a string to show where the data goes and what type it is . If you run the previous three lines of code, the output should look like the following:

```
The sum of 36 and 4 is 40
The square of 36 is 1296
```

The calculations were combined with the formatting of the strings and the printing of the formatted results.

The second parameter, the formatting string, may contain spaces or tabs to place the printed data at precise locations. This is useful for multiple executions that creates a tabular format.

```
printf(1, "Sum of 5.29 and          %f          is          %f\n",
{2.79, 5.29 + 2.79})
```

The output should appear as follows:

```
Sum of 5.29 and          2.79          is          8.08
```

Note that the newline character `\n` was used at the end of the format string to ensure that any following printing would occur on a new line.

The newline character is very important and requires some consideration.

```
puts(1, "How old are you?\n")
? 41
```

The output should appear as follows:

```
How old are you?
41
```

If you type and run the **clear_screen()** statement and then type again:

```
puts(1, "How old are you?")
? 41
```

The output should look like the following:

```
How old are you? 41
```


NOTE: Use the symbol "%f" for atom variables and "%s" for sequence variables used as strings with the format string.

Targeting Parts of the Screen

The text screen is made of 25 lines with 80 columns. A single sentence on one line may have 80 characters in it. Each letter in a normal sentence represents a screen column. One normal sentence of less than 80 characters only occupies one row of the screen. With a fresh screen that is blank, DOS will start printing on the first character of the first line unless instructed otherwise. This is the very top left corner. If the line exceeds 80 characters, DOS will wrap (normally) around to the next line where the line can be finished.

If the line is less than 80 characters and it is followed by an embedded **newline command** ("`\n`"), then DOS will ensure that any more printing begins on the line to follow. On the 25th line, DOS will also ensure that the screen scrolls down by one line. A new line is formed and the first line disappears. This is called scrolling.

You can override these automatic features of the operating system and place the cursor on any character position you want ($25 \times 80 = 2000$). The **position()** statement will accept two parameters (numbers) which correspond with row and column. The command, `position(2,34)` places the cursor on the second line at the 34th character, and that is where printing will begin.

If a row exceeds 25 or column exceeds 80 it will cause a fatal error because you have referred to an area off the screen. A fatal error will stop program execution immediately; it will interrupt everything abruptly and unexpectedly.

Variables

Variables are placeholders in memory for storing data. Memory locations have addresses, but you do not need to remember the addresses because you will use variables instead. You just remember the name of the variable, a name you have chosen. Euphoria will associate the variable with a memory address on your behalf.

Data may come from typing at the keyboard, or it may come from opening a file on the disk drive, or it may come from a network communication. Data that is not stored is lost. Variables can hold it temporarily while the computer is on, but to save it longer, you need to save to the disk.

Variable Names

Choose variable names as you like, but there are a few rules. Numbers may be included anywhere in the variable name except the beginning. Punctuation symbols are not allowed except for the underline, '_', character. Choose names that are helpful and suggestive of the contents and purpose. The Euphoria language reserves certain words that may not be used as the name of a variable. Those reserved words are:

Reserved Words

| | | | | |
|---------|----------|-----------|---------|-------|
| and | exit | or | with | elsif |
| end | not | while | else | if |
| include | type | do | global | then |
| to | constant | function | return | xor |
| by | for | procedure | without | |

The names or identifiers are case-sensitive, that is, capitalization *matters*. The variable **hour** and the variable **Hour** are not the same. The identifier **exit** is a reserved word and may not be used for the name of a variable, but **EXIT** is allowed because capitalization makes a big difference.

Older programmers by use the form **first_name**, for example, as a variable to hold the first name of an entrant, but it is more common today to use "Camel Capitalization". An internal capital letter or two eliminates the need for an "_" underline character, for example: **firstName**.

Storing Data in Memory

Memory is divided into storage locations and each location has an address. The address is all that is needed to find the location and any stored contents. Each bit of information has its storage locations and addresses.

Using variables means the programmer does not need to remember any addresses. Euphoria associates the variable to a storage location and address whenever you declare a variable. You can use the variable name instead of memory addresses.

If you want the information 'remembered' after the computer is turned off, you must save it to disk or some other permanent storage by saving the variable contents to disk contents. Like computer memory, disk memory is a series of bytes with an addressable location.

Variable Types

For now, you may think of data as two types, *single* values and *multiple* values. We will call variables designed to hold single (scalar) values *atoms* and variables designed to hold multiple (group) values *sequences*. As you look at the information around you, you can see that there are mathematical calculations and printable characters. To the computer, even the characters are coded as numbers, such as the ASCII code. The ASCII code offers 256 characters coded with the numbers 0 - 255. When the computer encounters numbers in that range and is told they are characters, then the computer will translate them instantly to their ASCII equivalents.

```
E   u   p   h   o   r   i   a   -- alphabet letters
69 117 112 104 111 114 105 97  -- ASCII codes for each letter
```

Since phrases and sentences are strings of characters (including the space character and punctuation), they are stored in sequence variables as strings. Anything that is a string, a list, or a series is stored as a sequence. All other individual values may be stored as an atom.

```
105586.93          -- stored in an atom
"And another thing..." -- stored in a sequence
3                  -- stored as an atom
{3.45, 56, 0.9, 34, 11022} -- stored as a sequence
{4, "four", 5, "five", 6, "six"} -- stored as a sequence
```

A sequence stores a series or list of items. An atom stores only one, usually a number value. Since characters are coded as numbers, and atom can store a single character if it is surrounded by single quotes.

```
atom lilbit

lilbit = 'a'
```

Declaring Variables

Before a variable may be used, it must be declared with a statement situated above the area where the variable will be used. If the variable is to be an atom named **big_num** then it is declared as follows:

```
atom big_num
```

If the variable is to be a sequence named **last_name**, then is declared as follows:

```
sequence last_name
```

You are free to declare several of the same kind of variable on a single line if you will separate each pair by a comma.

```
atom big_num, small_num, calc, diameter, stretch, finalTotal  
sequence last_name, short_list
```

Counter Variables in *for...do* Loops

The variable used as a counting variable in a **for..do** loop is a special variable in the Euphoria language. You do not need to declare the counting variable -- in fact you *must not* declare such variables. If you attempt to use a declared variable as a counting variable in the **for..do** loop, you will generate an error and an error report. If you attempt to use the counter variable outside its loop after the loop is finished, you will generate an error and stop the program. If you need to use the accumulated value of the counter outside the loop, you must assign its value to a declared variable and use the declared variable as such.

```
for i = 1 to 30 do
  for j = 1 to 23 do
    puts(1, '0')
  end for
end for
```

In the above code, **i** and **j** are counting variables.

Assigning Values to Variables

A variable gets its value through an "assignment" statement. There are three parts to an assignment statement: the variable that receives a new value (on the left extreme), an equal sign (=), and the number, string, or calculation whose value the variable takes on (to the right). Here are some valid examples:

```
customer_age = 41
my_name = "Shirley"
result = val_slate + ps_num + val_lower
full_name = first_name & ' ' & last_name
ver = (45 * 678) + (45/56)
```

If you break the rule, if you try to assign a sequential value to an atom variable, you will cause an error halting the program and receive an error message.

```
type_check failure, customer_age is {3, 4,87, 54, 2}
```

Initializing Variables

The first time you assign a value to a new variable it is called *initialization* of the variable. No variable may be used in an expression until it is first initialized. Failure to obey this rule will generate an error and error report. You may initialize number variables to zero and sequence variables to "" or {} (empty sequence) if you like.

```
myVar = 0.0
```

REMEMBER: the initialization of a variable is merely its very first assignment.

Calculations

In Euphoria, addition, subtraction, multiplication, and division are represented by +, -, * (the asterisk), and / (the slash), respectively. Exponential expressions are accomplished by the **power()** function.

| <i>Operation</i> | <i>Result</i> |
|------------------|-------------------------|
| ? 2 + 3 | 5 |
| ? 2 - 3 | -1 |
| ? 2 * 3 | 6 |
| ? 2 / 3 | .666666667 |
| ? power(2, 3) | 8 -- "two to the third" |

Accumulation with Variables

If a variable is initialized to some empty value, it may still accumulate through a series of steps into a variable of increased value. The example follows:

```
atom accum

accum = 0          -- initialization to zero
accum = accum + 3  -- accum now equals 3
accum = accum + 10 -- accum now equals 13

-- let us start again

accum = 1
accum = accum * 10 -- accum equals 10
accum = accum * 10 -- accum equals 100
```

In Euphoria, there is a shorthand version for each of these operations.

```
accum = accum + 1
-- or --
accum += 1      -- special shorthand operator

accum = accum * 20
-- or --
accum *= 20     -- special shorthand operator

-- Also --

accum /= 1
accum -= 1
```

The *remainder()* Function

The **remainder()** function allows a number to be divided by another with only the remainder as a result. This is often called a **modulus** mathematicians.

```
? remainder(7, 2)  -- 1 is the result displayed  
? remainder(6, 3)  -- 0 is the result displayed
```

Precedence of Operations

Euphoria evaluates mathematical expressions from left to right, following the rules of algebraic precedence: exponentiation is performed first, then multiplication and division, then addition and subtraction. The following example program illustrates algebraic precedence:

Parentheses can confirm or override the normal precedence.

```
? 2 * 3 + 2 / 3
```

The output is:

```
6.6666667
```

```
? (2 * 3) + (2 / 3)
```

The output is:

```
6.6666667
```

```
? 2 * (3 + 2) / 3
```

The output is:

```
3.333334
```

Use parentheses whenever you want to be sure what the precedence is. The first expression in parentheses to the left will be evaluated first, then all other parentheses moving to the right, then whatever remains from left to right.

Math Functions

Along with general math operations like addition, subtraction, multiplication, and division, there are advanced math functions like those available on scientific calculators. Generally, they require a number or expression in the parentheses and an assignment to the variable that will hold the final value. These functions are the secret to advanced or highly technical programs giving the computer the power for precise number crunching.

NOTE: All general purpose programming languages have a built-in calculator similar to this.

| | |
|--------------------|--|
| sqrt() | - calculate the square root of an object |
| rand() | - generate random numbers |
| sin() | - calculate sin of an angle |
| arcsin() | - calculate the angle with a given sin |
| cos() | - calculate the cosine of an angle |
| arccos() | - calculate the angle of a given cosine |
| tan() | - calculate the tangent of an angle |
| arctan() | - calculate the arc tangent of a number |
| log() | - calculate the natural logarithm |
| floor() | - round down to the nearest integer |
| remainder() | - calculate the remainder when two are divided |
| power() | - calculate a number raised to a power |
| PI | - precise value of PI |

Expressions

An expression is like a formula. Any combination of numbers, strings, variables, functions, and operators *that can be evaluated* is an expression. As an example, $2 + 2$ is a simple expression that evaluates to 4. Euphoria will evaluate expressions before moving on.

Logical comparison expressions are evaluated to either true (1) or false (0).

```
num = 2 + 4
```

In the above example, Euphoria will evaluate the expression $2 + 4$ before it is assigned. Only the evaluation's result is assigned to the variable on the left. The entire expression, $num = 2 + 4$, can be evaluated as either true (1) or false (0).

```
? num = 6
```

The output is:

```
1
```


Entering Data with the Prompting Functions

*NOTE: To add the prompting functions to the Euphoria language, you must include their library file, **get.e**. using an **include get.e** statement near the top of your program.*

NOTE: A string in programming is a series of printable characters, usually a sentence or phrase.

The **prompt_string()** function displays a prompt (an instruction to the end user), then waits for the user to enter some value. The command is a function that assigns the value inputted to a variable. It is usually used in an assignment statement. The **prompt_string()** function expects a string, and a string is assigned to the waiting variable. You can ask for numbers, but characters of the numerical type are returned when you use the **prompt_string()** function for retrieving numbers. You can print character numerals, but you cannot calculate with them.

```
yourname = prompt_string("What is your name?")
puts(1, yourname & '\n')
```

The **prompt_number()** function displays a prompt string, then waits for the user to enter some value, a numerical value. The **prompt_number()** statement is used to gather mathematical data from the user. Unlike the **prompt_string()**, the **prompt_number()** function requires *two* parameters. The first parameter is the usual prompt string, the second parameter is a sequence. The sequence allows two numbers separated by a comma, or it may be left empty. The first number is a lower limit and the second number is an upper limit. If the number input from the user is outside these limits, it is rejected. If the numbers are 2 and 4, then the error statement displayed when the number 5 is entered is "A number from 2 to 4 was expected -- try again".

```
yourage = prompt_number("What is your age?", {1, 120})
printf(1, %d, yourage)
```

In this above example, only a whole number between 1 and 120, inclusive, is allowed. Any other input is rejected with a message to try again.

The sequence as the second parameter may be an empty ({}) sequence. In this case, no limits are imposed on the input by the statement.

The **prompt_number()** function cannot accept a string value, so an error is reported if a string is entered, "A number was expected -- try again". This enforces input *validation* which is an important duty of many programs; validation eliminates many problems at the source, and that is much better than having to go searching for them after the fact in

an attempt to repair a broken program. The problem that was eliminated by validation was a potential *bug* (any program flaw).

Remember: to include either **prompt_string()** or the **prompt_number()** functions you must include the library they belong to, **get.e**, by placing the statement:

```
include get.e
```

near the top of the file.

```
your_address = prompt_string("What is your address?")
```

In the above line, a street address is expected in the sequence variable **your_address** because such data is stored as a string despite the fact that numbers are included. Numbers such as addresses which are characters signifying something are stored in strings as a rule. Such numerals are not for mathematics and cannot be used in math operations as long as they are characters in a string.

*NOTE: To transform character numerals to mathematical numbers, see the **value()** command.*

Sequences of Variables

If you wanted to keep track of the number of visitors to your website for every day in May, you could use 31 atom variables, named `may1`, `may2`, `may3`, and so on through `may31`.

However, every separate variable needs to be handled individually. If you are trying to prompt for every one of 31 individual variables, you need 31 statements.

```
may1 = prompt_number("How many visitors for May 1?")
-
-
-
-
may31 = prompt_number("How many visitors for May 31?")
```

A similar problem arises when you need to print the thirty-one values.

The solution is to give the entire range a single name with a series of subscripts. Each value is distinguished by its number and each can be handled individually but the entire array can be handled as a group using a single statement.

`May[1] May[2] May[3] May[4] May[5] ...`

The ability to handle related data in a single structure (a sequence or array) is a powerful advantage to computer programming. If one million customers for Ajax Widgets, Inc. are all stored in a single sequence, each with its own unique subscript (1-1000000), then each customer's information can be visited with all others in the same operation with a loop that loops 1000000 times for 1000000 customers. That is a lot of power in a few lines of code.

Sequences Hold Arrays of Variables

Arrays are data types found in every programming language, but in Euphoria they are implemented as sequences. Sequences can be arrays, but they do much more than that. An array is a list, a series, or *sequence* of values. In most languages, all elements of an array must be the same data type, but in Euphoria, each element may be a different data type. In fact, each element may be another sequence.

Arrays offer many advantages. They are an efficient way to use large amounts of memory to control large amounts of data. By using loops, you can handle thousands of variables with a single loop statement thanks to array variables.

Arrays are used when you want to sort a series of numbers or a series of names. They are used whenever you need many variables with a common name (customers, for instance).

If an array named **list** has 100 elements in it, **list[1]** is the first element, and **list[100]** is the last element. Despite their connection to one another, each element may be used as a completely separate variable as long as the correct subscript is identified. In some cases, you will refer to the whole sequence as a sequence variable and then you will not use any index at all.

All you need to declare an array is to declare a sequence variable. Any sequence variable may be used as either a string or an array or perhaps some other data structure, but if you want to, you can size or *dimension* an array. If you wanted an array of 12 elements to symbolize the months of a year, you can dimension the sequence variable **month** like so:

```
sequence month

month = {0,0,0,0,0,0,0,0,0,0,0,0,0}

-- or --

month = {{}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}}

-- or --

month = repeat(0,12)

-- or --

month = repeat({},12)
```

You can rearrange the members of a sequence by swapping their values through

assignment statements. This is how a list is sorted in order, and it is how a deck of 52 cards, as in the game 21.ex, is shuffled. Shuffling is the opposite of sorting.

Specifying Sequence Elements

Since a sequence variable may have a few elements or a few thousand, it is important to measure a sequence from time to time. The **length()** function will tell you the number of elements of any sequence. The notation of subscripts allows mathematical manipulation of subscripts.

```
plot[234+num_points] = 0
pts[4] = plot[length(plot) - 4]
```

You may specify a range within the full range of elements too. The following example specifies the range of elements 5 through 12. This is called *slicing*.

```
oth_seq = grp_a[5..12]
```

The elements 5 through 12 is 8 elements long. The new sequence, **oth_seq** is now 8 elements (subscripts 1 through 8) long. What was **grp_a[5]** is now **oth_seq[1]**; what was **grp_a[12]** is now **oth_seq[8]**. They were only copied, however. The entire sequence, **grp_a**, still exists.

There is a shorthand symbol (\$) for the length of a sequence. It literally means, "the length of this sequence".

```
plot[$ - 1]
```

This expression specifies the next-to-last element in the sequence, **plot**.

Remember: Each element in a sequence is a variable in itself. That variable may be an atom or yet another sequence, or even a sequence of sequences. Some statements only work on sequences; some only on atoms. Sequence statements will accept the sequence as a whole while a single element, if it is an atom, can be passed to an atom-only operation.

Logical Relations Used in Decision-Making

Euphoria can decide whether two numbers or two strings or two sequences are the same or different. On the basis of such a decision, program statements may be executed or repeated. Computers are not really intelligent, they do not actually think the way humans do, but they often seem to because of the ability to observe and make decisions based upon the observed conditions. This is logical decision-making, and your programs can do it.

Relational Operators and Functions

Strings are compared with comparison routines like **equal()** or **compare()**, but numbers and mathematical expressions can be compared with the routines above or with "relational operators".

| Relational Operator | Meaning |
|---------------------|--------------------------|
| <code>=</code> | Equal to |
| <code>!=</code> | Not equal to |
| <code>></code> | Greater than |
| <code><</code> | Less than |
| <code>>=</code> | Greater than or equal to |
| <code><=</code> | Less than or equal to |

Here are some examples of how statements are evaluated:

| Relational Operator | Meaning |
|------------------------|---------|
| <code>7 = 35</code> | False |
| <code>7 != 6</code> | True |
| <code>6 > 1</code> | True |
| <code>4 < 3</code> | False |
| <code>7 <= 7</code> | True |
| <code>8 >= 7</code> | True |

Here are some examples for comparing strings (they use the **equal()** and **compare()** Euphoria functions):

| Relational Operator | Output (meaning) |
|--|------------------|
| <code>not equal("John", "Paul")</code> | 1 (true) |
| <code>compare("John", "Paul")</code> | -1 (less than) |
| <code>equal("Y", "Y")</code> | 1 (true) |
| <code>equal("Y", "y")</code> | 0 (false) |
| <code>compare("miss", "misty")</code> | -1 (less than) |
| <code>compare("mis", "mis")</code> | 0 (equal to) |
| <code>compare("misty", "miss")</code> | 1 (greater than) |

The **equal()** function returns either 1 for true or 0 for false, but the **compare()** function returns 0 for equal, 1 for greater than, and -1 for less than. The answer is 1 when the first parameter is greater than the second. These functions conform to the rules for alphabetical order and are used for sorting strings alphabetically.

Remember that letters of the alphabet equate to numbers in the ASCII system that Euphoria and other languages recognize. When using characters one at a time, if you

surround the character with single quotes, it can be evaluated as a number using the relation operators. Strings may not be compared using the relation operators.

```
'A' > 'a' -- this expression is allowed  
-- whether true or not
```

```
"A" > "a" -- this expression is not allowed  
-- whether true or not
```

```
compare("A", "a") = 1 -- this is the legal equivalent  
-- whether true or not
```

Strings are compared letter by letter until a mismatch occurs. Based upon the alphabetical order of the letters, one may be larger than the other by being found later in the alphabet. The length of the string is also considered when comparing strings alphabetically.

Boolean Expressions

George Boole formulated some of the rules of mathematical logic, so logical assertions have been called "Boolean expressions". This is part of the study of mathematics that seeks to formulate pure logic. Euphoria always evaluates an expression as either true or false. For instance, $4 > 5$ is an expression which is false. Euphoria expresses true and false as 1 and 0. You can see the Boolean evaluation by printing (and thus evaluating) the expression.

```
? 4 > 5 -- result 0 is printed, it is false
? 5 > 4 -- result 1 is printed, it is true
```

Compound Expressions

In Euphoria, a compound Boolean expression is created by connecting two Boolean expressions with a "logical operator". The two most commonly used logical operators are **and** or **or**.

The **and** operator requires both expressions to be true if the compound expression is to be true. When the **or** operator is used, only one of the expressions has to be true for the compound expression to be true.

The following are simple examples:

| Expression | Evaluation |
|--------------------------------------|------------|
| 10 > 5 and 100 < 200 | True |
| 3 < 6 and 7 > 10 | False |
| 8 < 7 or 90 > 80 | True |
| 2 < 1 or 3 > 60 | False |
| 'Y' > 'N' and not equal("yes", "no") | True |
| 'Y' < 'N' or 4 != 4 | False |

The **not** operator reverses the truth or falsity of an expression:

| Expression | Evaluation |
|------------------------|------------|
| not (5 > 10) | True |
| not (8 < 7 or 90 > 80) | False |
| not (3 < 6 and 7 > 10) | True |
| not (0) | True |

The logical operators can be combined to build up very complicated compound expressions. There is virtually no limit to the possible complexity.

NOTE: Euphoria will evaluate a compound relational statement only as far as necessary. The first comparison that makes the entire combination false is the point where comparison stops. This energy-saving feature is called short-circuiting.

Using Logical Statements to Control Program Flow

Which is more important? What decision should be made? Euphoria has a mechanism to decide which part of the program to execute next: The **if...then...else** statement.

This is the syntax of the **if...then...else** statement:

```
if booleanexpression then  
    statements to do something  
else  
    statements to do something else  
end if
```

Repeating Program Operations

Euphoria offers several ways to execute a group of program statements repeatedly. You can repeat them a fixed number of times or until a particular logical condition is met. If you want to execute a block of statements 100 times, you need only type them in once. Euphoria's control structures then control the repetitions.

The ability to repeat program statements has many uses. For example, to enter data for a ten-element array, you must prompt the user ten times. Loop operations make this possible.

The *for...do* Loop

One way to repeat a section of the program is the **for** loop. Use the **for** loop when you know in advance exactly how many repetitions are needed.

Consider the syntax:

```
for count = 1 to 5 do  
    printf(1, "this line is printed 5 times; this is time %d\n", count)  
end for
```

The output printed by the above example follows:

```
this line is printed 5 times; this is time 1  
this line is printed 5 times; this is time 2  
this line is printed 5 times; this is time 3  
this line is printed 5 times; this is time 4  
this line is printed 5 times; this is time 5
```

In the example above, the variable **count** is called the "loop variable" or "loop counter". The two numbers after the equal sign (separated by the keyword **to**) are the start and end values for the loop counter. In this example, 1 is the start value, and 5 is the end value.

The loop counter is a special variable in Euphoria that should not be declared, but which is automatically declared at the start of the loop and automatically destroyed at the end of the loop. The variable will not be recognized outside the loop and should not be replaced by a declared variable used elsewhere.

Before the **for** statement is executed the first time, the loop counter is given the value of the start value (in this case 1). After each execution (repeat) it is increased by 1. This continues until the counter is greater than the end value (in this case 5) at which moment the loop ends and the counter variable is destroyed. The program then continues on the next line following the last line of the loop, "end for" or "end while".

You can also count in increments other than 1, by 2 or by 5, for instance. You can count backwards (count down) by properly constructing the loop to do so, and by introducing the **by** keyword when appropriate. See the following example:

```
for count = 5 to 1 by -1 do -- minus one is the increment,  
                           --decreasing  
    printf(1, "this line is printed 5 times; this is time %d\n", count)  
end for
```

You must be consistent making sure that the start and end values make sense, and making sure that the step value makes sense in the context of the statement.

The *while* Loop

The **while** loop is used when the exact number of repetitions is not known in advance. Use the **while** loop for *indefinite* loops. That is, based upon immediate conditions, a while loop might not execute even once or perhaps four times, or perhaps thousands of times.

Here is the syntax:

```
while <booleanexpression> do  
  <statements to be repeated>  
end while
```

Here is an example of how it works:

```
big = 256.0  
little = 1.0  
while big != little do  
  printf(1, "big = %d    little = %d\n", {big, little})  
  big = big / 2.0  
  little = little * 2.0  
end while
```

The variables **big** and **little** are not equal outside the loop, but the variables are changed in value as the loop repeats. At some point **big** and **little** will be equal, if not, then the loop will never end because the loop is designed to end when they equal. The expression **big != little** following the **while** keyword determines the conditions under which the loop will execute. If **big** and **little** were equal at the start, the loop would not execute even once. While the expression is true, the loop executes, when the expression is false, it is finished.

In the above loop, the exit is taken from the top of the loop where the comparison is made. By placing a constant in the expression place, the "expression" is always true and the loop is endless. To prevent an endless loop, we can use a comparison test at the bottom of the loop containing the **exit** statement. This is how to make a loop that exits from the bottom instead, and it guarantees that the loop will execute at least once.

```
-- start with an "endless loop" but give it an exit  
num = 0  
while 1 do          -- the test is always true  
  num = num + 1  
  printf(1, "%d\n", num)  
  if num > 10000000 then  
    exit          -- exits here  
  end if  
end while
```

Programmers call this kind of while loop that exits from the bottom (instead of the top) a "do until" loop. A do-until loop always executes at least once. There is another way to do this without using a "forever loop".

```
-- do it at least once, if that did not meet the condition then
-- continue until it does.
num = 0
num = += 1 -- num is increased at least once, but possible more
--times.
while n <= 10000000 do -- when true the loop quits here at the
--comparison logic
  num = += 1
  printf(1, "%d\n", num)
end while -- it may exit from the top, but the
--continuation follows this line
```

The result of this loop is the same; it is a "do until" loop. It executes at least once, but in this case, it exits from the top without any extra exit statements. Like the example before it, it continues *until* a condition is met, not *while* a condition is met.

Be sure that the condition of the loop's test can be met to ensure that the loop will end. Design the comparison test carefully. Be sure the test variables have the value you want before you enter the loop.

Exiting Loops

Any loop in Euphoria may be exited prematurely by using the **exit** statement. An **exit** statement in a **for** loop means the loop will end before the counter's end value is reached. The **exit** is designed to be contained inside an **if..then** statement in order to exit loops conditionally.

```
for i = 45 to 0 step -5 do
  printf(1, "%d", i)
  if i < 5 then
    exit
  end if
end for
```

```
n = 0
while 1 do
  n += 1
  if n > 100 then
    exit
  end if
end while
```

Library Files

Much of the language of Euphoria is optional and available only when extra library files are loaded. To load a library file, you place an "include" statement above the code where the optional features will be used. This mild inconvenience is designed to keep the programs as compact as possible. Libraries not needed are not loaded, and Euphoria is more efficient as a result.

```
include get.e
include misc.e
```

To use the **prompt_string()** and **prompt_number()** routines you will need the **get.e** library file. The **file.e** file is necessary if you want either the **seek()** or **where()** functions. The **bk_color()**, **text_color()**, **get_position()**, **sound()**, and **cursor()** functions are found in the **graphics.e** file. The **lower()** and **upper()** functions may be found in the **wildcard.e** library file.

dll.e

file.e

get.e

graphics.e

image.e

machine.e

misc.e

mouse.e

msgbox.e

sort.e

wildcard.e

*NOTE: You may write your own library files based on some specialty. If you are a real estate agent, you might write code that is real estate specific that other real estate professionals will want to use in their real estate programs. You can store all code related to real estate together in its own library, **real_estate.e**, that is loaded with **include real_estate.e** at the top whenever a program is real estate related.*

Libraries for Windows programming end in *.exw.

Top Level Statements

NOTE: The term "bug" means "troublesome program flaw" ; the term "debug" means to fix problems with the program.

There are special statements that must occur at the top level, above the other code, when they occur. The **include** statement is one these; it calls for the inclusion of library files that are usually needed in larger programs.

The **with** and **without** statements are top level statements that refer to Euphoria debugger and the process of debugging. There are profiling features that offer metrics for debugging and optimization.

The debugger allows *tracing*, single-stepping from statement to statement inspecting variables as you go. This process is turned on with the **with trace** top level statement; it is turned off with the **without trace** statement.

Type-checking is a great safety feature and debugging feature that can be turned on and off with **with type_check** and **without type_check** statements.

Use it to turn on/off the **profile** and **profile time** features.

In the interest of completeness, the Euphoria debugger issues warning statements automatically at the end of program execution, but this feature can be turned off with the **without warning** statement.

Simple Program Example

Consider how to write a simple program and how to describe it beforehand. We will describe a simple program with a single purpose using 5 steps.

1. Prompt the user for the quantity of number to be averaged.
2. If the quantity is zero or negative, print a warning message and do nothing else.
3. If the quantity is positive, pick a variable name for the running total and set it equal to zero. The user has selected the number of entries.
4. Prompt for the numbers one at a time. Tell the user each time which value is being entered (number 1, number 2, number 3, and so on.)
5. When all the numbers have been entered, compute and print the average value of all the numbers entered and stored.

Here is the simple program:

```
include get.e          -- get.e is needed for prompt_number()
atom total, valu, howmany

howmany = prompt_number("How many numbers do you wish to average? ",
                        {1, 100})

total = 0.0           -- initialization of total
if howmany > 0 then   -- extra precaution
  for count = 1 to howmany do -- for counter automatically
                                --declared!,
                                -- good.
    printf(1, "number %d", count) -- each entry counted with count shown
    valu = prompt_number("? ", {}) -- get each entry, floating point
    total += valu                -- same as "total = total + value"
  end for

  -- the last statement is run once, outside the for loop
  printf(1, "The average value is %f\n", total / howmany)
end if

abort(0) -- normal program end
```

Debugging

NOTE: The term "bug" means "troublesome program flaw" ; the term "debug" means to fix problems with the program.

When you finish writing your program, it is highly unlikely that it will run without error. Bugs, errors, are a fact of life for the computer programmer. Some errors are syntax or typographical errors and others are mistaken logic. All you know in the beginning is the program will not run correctly or it will not run at all.

Euphoria is designed to eliminate bugs at the earliest convenience. This is one of the reasons Euphoria is pleasure to program. The extensive type checking built into the environment stops many errors from the start. When errors stop your program, there is a clue on the screen and there are more clues in a log file saved to disk, **ex.err**. Read the text file to see the content of variables just prior to the problem. This is very valuable information. Very large sequences will be shown only partially. In some cases, you can have all the executed statements leading up to the disaster. Remember to inspect **ex.err**.

The crash statement, the statement left on the screen after the program crashes, is often very helpful, but it usually points the finger at the lines or clauses *directly following the true error*. The line with an error may be shown on the screen, or the line following the error may be shown. If you forgot the closing ']' brace, the debugger leaves the message, "Syntax error - expected to see ']' possible, not a variable." Of course, you will quickly learn that you made a typographical error. The variable actually follows the error. If you neglected to supply the **end procedure** statement at the end of a procedure called **CheckAll()** that precedes a procedure called **CheckOne()**, the error message might be:

```
Syntax error - expected to see 'end' not 'procedure'  
  procedure CheckOne()  
    ^
```

The debugger points to the line following the error, the next line that declares the next procedure. The debugger was looking for the **end procedure** statement when it encountered the **procedure CheckOne()** statement instead. So the line with **procedure CheckOne()** is *not* the location of the error, but it is the line that directly follows the error, in this case, an omission.

Notice that the debugger has also named the line where the program stopped. Here is the message when too many parameters (arguments) are assigned to the **find()** statement (line 422):

```
C:\EUPHORIA\DEV\myprog.exw:422  
find takes only 2 arguments  
  if find(trep, subchoice, {1, 2, 3, 4, 5}) then
```

^

A debugger is a valuable tool for programmers that is not always available for every programming environment (product), but in Euphoria a sophisticated debugger is built in. The most valuable debugging tool is the *tracer*. At the top of the program the **with trace** statement must be in effect or the tracer will not work. Tracing begins on the first line that contains the **trace(1)** statement. This ensures that the program proceeds as normal *until the statement is encountered* at which point the program stops at that line and lights up the debug screen. You can step one line at a time by pressing the **ENTER** key.

The debug screen is divided. Above is the code with color coding and highlights and below is a field where variable contents are revealed automatically. If you do not see the contents of a variable of interest to you, you can get what you want by choosing **?** and entering the variable's name. The variable will be added to the lower field and its contents shown.

The debugging steps might lead you inside a loop with hundreds of iterations, but rather than quit or rather than press the key hundreds of times, you may exit the loop immediately with the **DOWN-ARROW** button on your keyboard. As you step through the program lines you learn how the program statements work. You see the variables change normally, but you can catch it very slow motion. You can leave the debug mode and continue the program at full speed by typing 'q'. The program will proceed to the finish unless another **trace(1)** command is encountered. If you type a capital 'Q' instead, the program will proceed to the finish at full speed ignoring any other **trace(1)** statements it finds.

*NOTE: the **trace(2)** statement is the same but with a monochrome screen. The **trace(3)** statement will record all executed statements and save the record to a file called **ctrace.out**.*

While trace is in effect, you can peek at the output screen as it changes by typing the **F1** key; you return to the trace screen by typing the **F2** key. The **DOWN-ARROW** key will also let you single-step procedures and functions without tracing inside them. Type the **!** key quit the trace, program and all. See **ex.err** for a tracing log.

Editors

An editor is a trimmed-down, basic word processor. Some editors for programmers offer color coding of identifiers and other special features programmers appreciate. Some editors are designed for Windows, others are designed for character-based operation, even on Windows. A DOS editor is included with Windows and can be invoked by typing **edit** on the console command line. This editor is important because it has resided on millions of computers and its operation is familiar to millions of people. Windows includes two free Windows editors, Notepad and Wordpad. Wordpad is actually a word processor. Notepad resembles the DOS editor. If you use a word processor, you must choose the option to save as a *.txt file, but do not use the .txt ending, use .ex for DOS and .exw for Windows. Any files other than so-called ASCII files will have processor codes that mess up the translation process. Many programmers are quite happy with Notepad. If you have a choice of font, choose Lucinda Console or some typewriter font, these have equal spacing of characters and are easier to read when writing code.

The Euphoria editor is available free with the installation of Euphoria. The Euphoria editor is a character-based editor like DOS Edit.com but it has special features for programmers, and it is written entirely in the Euphoria language; it is invoked by typing **ed** at the command line in the console window or screen. Though it does not resemble the editors mentioned previously, it is full-featured and easy to learn. It is recommended because the color coding of the code clarifies the code. You can run code from the editor. You can search, cut and paste. You can list the editor's help file, and you can scan the Euphoria documentation that includes a definition of the entire language at your fingertips. It is challenging enough to learn a new programming language, but you must also learn to use an editor if you have not already mastered a compatible one

```
Command Prompt - ed 21.ex
^CTerminate batch job (Y/N)? -
    puts( 1, card[1] )
    end if
    position( vPos + 2, hPos + 1 )
    puts(1, card[2])
    end if
end procedure
-----
procedure pause()
-- The pause procedure allows the player to examine the
-- playing table until he or she is ready to continue.
-- Pause places a message in the lower-right corner
-- of the screen and waits for the player to press any key.
    position( 24,49 )
    puts( 1,"Press any key to continue..." )
    if wait_key() then end if -- will wait and return 0 until key pressed
end procedure
-----
function shuffle( sequence arr )
-- A shuffle is the opposite of a sort. It insures that the multiple
-- contents of arr are *not* in order and unpredictable.
    integer temp, lenth
    sequence c

    -- sometimes 52, sometimes less -- reuse cards already played
    -- go to each card and swap it with a random number (1 to max num)
    lenth=length( arr )
    for i = 1 to lenth do
        temp = rand( lenth ) -- get proper random number

        -- swap values
        c = arr[i]
        arr[i] = arr[temp]
        arr[temp] = c
    end for
```

Ed's menu, type the **ESC** key. The menu is a single line at the top of the screen.

help clone quit save write new ex dos find replace lines
mods ddd CR:_

Press and release the Esc key, then press one of the following keys:

h - Get help text for the editor, or Euphoria. The screen is split so you can view your program and the help text at the same time.

c - "Clone" the current window, i.e. make a new edit window that is initially viewing the same file at the same position as the current window. The sizes of all windows are adjusted to make room for the new window. You might want to use Esc **l** to get more lines on the screen. Each window that you create can be scrolled independently and each has its own menu bar. The changes that you make to a file will initially appear only in the current window. When you press an F-key to select a new window, any changes will appear there as well. You can use Esc **n** to read a new file into any window. **q** - Quit (delete) the current window and leave the editor if there are no more windows. You'll be warned if this is the last window used for editing a modified file. Any remaining windows are

given more space.

- s - Save the file being edited in the current window, then quit the current window as Esc q above.
- w - Save the file but do not quit the window.
- e - Save the file, and then execute it with ex, exw or exu. When the program finishes execution you'll hear a beep. Hit Enter to return to the editor. This operation may not work if you are very low on extended memory. You can't supply any command-line arguments to the program.
- d - Run an operating system command. After the beep, hit Enter to return to the editor. You could also use this command to edit another file and then return, but Esc c is probably more convenient.
- n - Start editing a new file in the current window. Deleted lines/chars and search strings are available for use in the new file. You must type in the path to the new file. Alternatively, you can drag a file name from a Windows file manager window into the MS-DOS window for ed. This will type the full path for you.
- f - Find the next occurrence of a string in the current window. When you type in a new string there is an option to "match case" or not. Press y if you require upper/lower case to match. Keep hitting Enter to find subsequent occurrences. Any other key stops the search. To search from the beginning, press control-Home before Esc f. The default string to search for, if you don't type anything, is shown in double quotes.
- r - Globally replace one string by another. Operates like Esc f command. Keep hitting Enter to continue replacing. Be careful -- there is no way to skip over a possible replacement.
- l - Change the number of lines displayed on the screen. Only certain values are allowed, depending on your video card. Many cards will allow 25, 28, 43 and 50 lines.

In a Linux/FreeBSD text console you're stuck with the number of lines available (usually 25). In a Linux/FreeBSD xterm window, ed will use the number of lines initially available when ed is started up. Changing the size of the window will have no effect after ed is started.

- m - Show the modifications that you've made so far. The current edit buffer is saved as editbuff.tmp, and is compared with the file on disk using the DOS fc command, or the Linux/FreeBSD diff command. Esc m is very useful when you want to quit the editor, but you can't remember what

changes you made, or whether it's ok to save them. It's also useful when you make an editing mistake and you want to see what the original text looked like.

ddd - Move to line number ddd. e.g. Esc 1023 Enter would move to line 1023 in the file.

CR - Esc Carriage-Return, i.e. Esc Enter, will tell you the name of the current file, as well as the line and character position you are on, and whether the file has been modified since the last save. If you press Esc and then change your mind, it is harmless to just hit Enter so you can go back to editing.

My personal favorite editor is David Cuny's free DOS editor `ee` that available at the RDS website. It resembles the DOS editor and operates virtually like clone, so it is easy, easy, easy. But it also has extend programming features like color coded source text. It is fair to call it an IDE rather than an editor. It has many of the special programming features like `ed` that programmers crave.

New programming editors for Windows are being developed all the time, and many of them are customizable so they can be customized to suit Euphoria. Check the Archive at the RDS website periodically to see new product.

More on Sequences

NOTE: In programming, a string is a series of printable characters, usually a sentence or phrase. In Euphoria, strings are handled as sequences. Each character in a string is an individual element that can be specified with a numerical subscript, or sliced with a subscript range. We surround strings with double-quotes to distinguish them from identifiers.

Regardless of type, all variables other than for-loop counters must be declared and then initialized before they can be used in an expression. As previously stated, initialization is the first assignment made to a newly minted variable. It is customary to initialize atoms to **0.0** and sequences are often initialized to empty, either "" or {}.

```
sequence my_series
my_series = {}
```

If you assign an atom value to the sequence, **my_series**, you will cause a fatal error, but you may append an atom to an empty sequence, indeed to *any* sequence. There are two ways to do so.

```
my_series &= 45.67

-- or --

append(my_series, 45.67)
```

In either event, **my_series** is now {**45.67**}. It is not an atom -- it is a sequence of length one whose first element is an atom. You may put any value at any element in a sequence, and you may enlarge or decrease any sequence at will. Actually each element of a sequence is an object data type capable of handling any data type.

NOTE: If you reassign a sequence (or any variable), you will destroy the original contents first.

It is a common mistake to try to assign an atom to a sequence and it is more common to attempt to append values to an uninitialized sequence.

However, each element in a sequence can be singled out with a subscript number, and that element may be treated as an object variable that may assume any value. Therefore, you may assign an atom value to a sequence element when you identify the element with a subscript. Of course, the element must first exist.

In the above example, we went from an empty sequence to a sequence with one element

whose value is 45.67. You can now assign other atom values to this element.

```
my_series[1] = 500000.00099
-- now my_series is {500000.00099}
```

However, the following is an error:

```
my_series[2] = 45.67
```

...because `my_series` only has one element. To create another element, you must append it. To create more elements, *append*, but you may *assign* to elements that already exist. For instance, we might have done this instead:

```
my_series = my_series & 45.67

-- now my_series is {500000.00099, 45.67}
```

If you know that you want a sequence with 12 elements, and it is unlikely that the number of elements will change, you may prefer to dimension the sequence immediately after declaring it.

```
my_series = {0,0,0,0,0,0,0,0,0,0,0,0}
```

Now that `my_series` has been initialized and now that `my_series` has 12 elements established, you may address each of the elements separately and assign to them as you like.

```
my_series[12] = "end"
```

Of course, if you need a sequence of twelve elements with the name of each month in each element, you can initialize the sequence to those values.

```
monthnames =
{"January", "February", "March", "April", "May", "June", "July", "August", "Sep-
tember", "October", "November", "December"}
```

```
-- then....
```

```
puts(1, monthnames[5] & '\n')
```

```
-- prints May
```

If you want a data type to symbolize a chess board you need a matrix. The chessboard is a matrix of 64 empty values. That is, eight rows by eight columns. This can be emulated by a two-dimensional array, which is created in Euphoria with a sequence.

```
sequence ChessBoard
ChessBoard = repeat(repeat(0, 8), 8)
```

```
-- or --
```

```

ChessBoard = {
    {0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0}
}

```

The second square on the third row is **ChessBoard[3][2]**. The fourth square on the last row is **ChessBoard[8][4]**. Note the two subscripts for the two dimensions. You can also create 3 or more dimensions. It is very rare that more than four dimensions are ever needed in any array.

Chess players use a, b, c, d, e, f, g, h to identify the rows (starting on the left side), and 1, 2, 3, 4, 5, 6, 7, 8 columns (starting from the bottom, white's side). If you declare the constants:

```

    constant h = 1, g = 2, f = 3, e = 4, d = 5, c = 6, b = 7, a = 8
-- so --
ChessBoard[h][1] -- upper left hand corner

```

When a sequence holds only a series of numbers from 0 to 255, the sequence qualifies as a string because that is the range of codes for printable characters. Conversely, a sequence used as a string, like a sentence or a phrase, can be printed like a series of numbers instead. Observe...

```

seq1 = "Euphoria"

puts(1, seq1)

-- the result is Euphoria, but...

? seq1

-- the result is {69,117,112,104,111,114,105,87}

```

In the above example, 69 is the ASCII code for capital "E" and 87 is the ASCII code for little "a".

In fact, the sequence type allows sequences of sequences, such as...

```

monthnames =
{"January", "February", "March", "April", "May", "June", "July", "August", "Sep
tember", "October", "November", "December"}

```

This is no longer a string, but a sequence of strings. Another name for this is *nested*

sequence. There are some procedures that must work with strings and will fail if given nested sequences. For instance, **puts()**. You cannot print with **puts()** the sequence **monthnames** because it has a sequence within a sequence (a string within a sequence), but you can print any of the elements because each of the elements alone is a string.

```
puts(1, monthnames) -- Wrong! This generates an error.

puts(1, monthnames[5]) -- Correct. A single element in this sequence
                        --is a string
```

From the example above, the sequence **monthnames** is a nested sequence and the two-dimensional array, **ChessBoard**, is also a nested array. To single out the third element in **monthname** you use **monthnames[3]**. In other words, **monthnames[3]** is "March", but **monthnames[3][3]** is 'r' (the *third* letter in "March").

You must be careful and mindful of your actions when getting started with sequences, but in time they will be easy and will be second nature. For instance, observe the difference:

```
list_of_data = {}
list_of_data &= 89
list_of_data &= 345

--- list_of_data is {89,345}

list_of_data &= {102} -- see the curly braces

--- list_of_data is {89,345,{102}}
--- because {102} is a sequence of one element
--- not an atom. Therefore, list_of_data is now a nested sequence

-- Also, FYI, {{89,345,102}} is a nested sequence as well
```

Up to now, sequences have been used for strings, arrays, and multidimensional arrays, but now let us inspect another use for the sequence data type, records. Record variables are sometimes called structures, structs, or user-defined types. To consider a record variable consider a typical record (as in record-keeping). If you see a telephone directory as a huge table, then each entry (customer) is a single record. For each record, there is name, address, and telephone number. This is true of each record among thousands. What is a record for? If we only wanted to list a thousand names, we could use an array, but if we want more information for each name, then we need an array of record variables. There is very little difference between an array of records and a database.

```
{
  "John",           -- look closely for commas
  "C.",
  "Doe",
  "123 Alpha Drive",
  "345-4567
}
```

So we have a nested sequence where each of the elements is another sequence used as a string. This separation of details allows us to focus on details when necessary. We might go further with this.

```
sequence tel_rec
tel_rec = {}          -- initialization
tel_rec &= {0,0,0,0,0} -- the first record

atom tel_fname, tel_mname, tel_lname, tel_address, tel_number
tel_fname   = 1
tel_mname   = 2
tel_lname   = 3
tel_address = 4
tel_number  = 5

tel_rec[1][tel_fname]   = "Megan"
tel_rec[1][tel_mname]   = "E."
tel_rec[1][tel_lname]   = "Hooper"
tel_rec[1][tel_address] = "434 Elm Lane"
tel_rec[1][tel_number]  = "432-8427"

-- each record has its own serial number (1 here), but each has
-- 5 items of data, first, middle, last, address, and telephone
--number
--
```

Megan, here, is our first customer and our first record. If we have thousands of customers, each has a number, a customer i.d.. We might call it the customer's account number.

```
tel_rec[237937][tel_number] = "443-4837"
```

Customer number 237937 has telephone number "443-4837".

Comparing Strings

Comparing values is a common activity in programming, but comparing strings is more complicated than comparing numbers. String comparison is a central action in any alphabetical sorting routine. You must not use the `>`, `<`, `>=`, `<=`, or `=` operators with strings.

If you want your program to discover if two strings are equal, use the **equal()** function that returns 1 for true and 0 for false. This makes it useful and fluent when combined with the **if** statement.

```
if equal("miss", "Miss") then
  --<someactiontaken>
end if
```

You can also use the **compare()** function for this and for other things. This function returns 0 for equal, -1 for less-than, and 1 for greater-than, *alphabetically* (in fact, **compare()** can also compare numbers in like fashion, it was designed to be multipurpose). If you design your sort routines with **compare()**, they will sort strings as well as numbers.

Comparison starts with the first character and continues until there is a difference in comparison. If a string has added spaces on the end, it will be judged greater than the same string minus the added spaces.

Searching for Strings

NOTE: A string is a series of printable characters, usually a sentence or phrase. We surround strings with quotes to distinguish them from identifiers.

One of the most common string processing tasks is finding a string within a string. In Euphoria, the **match()** function is the method to use. If a smaller string is found within a larger one, **match()** returns the number that is the index into the searched string. If the inner string starts at the third character, then **match()** returns 3.

```
string1 = "A line of text with 37 letters in it."
string2 = "letters"

puts(1, "          1          2          3          4\n")
puts(1, "1234567890123456789012345678901234567890\n")

? match(string1, string2)
```

Output

```
          1          2          3          4
1234567890123456789012345678901234567890
A line of text with 37 letters in it.
24
```

You may use **find()** to search for single characters within a string as long as you specify the single character with single quotes around it. Use **find()** for this because it is much faster than **match()**.

Retrieving Characters from Strings

You can retrieve characters, or strings of characters (called slices), from anywhere in a string using subscript notation. The characters in a string are numbered 1 to the last character in the string. You can find the last character's number by using the **length()** function, and you can refer to the last number within brackets [] by using the **\$** character. To specify the last character:

```
str1[$]
```

If you want to specify the third character (counting from the left):

```
str1[3]
```

If you want to assign the first three characters to another sequence use the following technique:

```
smseq = str1[1..3]
```

If you want to assign instead the last three characters use the following:

```
smseq = str1[$-3..$]
```

If you want to slice off five characters starting with the fourth character then use the following:

```
smseq = str1[4..9]
```

You are not allowed to slice from beyond the true length of the string. If you miscalculate the length, an error will occur. Use the **length()** function to stay within string bounds and avoid errors.

Generating Strings

The **repeat()** function can generate any length string using repeating characters or repeating slices.

```
puts(1, repeat('r', 20))
```

Output

```
rrrrrrrrrrrrrrrrrrrrrr
```

You can also create a string of invisible spaces the same way.

Numbers and Strings

Euphoria does not allow a string to be assigned to a numeric variable, nor does it allow a numeric expression to be assigned to a string variable. For example, both of these statements result in an error message, **Type_check failure,...**

```
TempBuffer = 45
Counter = "45"
```

Instead, use the **value()** function to transform the string to its equivalent number. The string, "45", makes numeric sense, therefore a function like **value()** can transform it. Use **sprintf()** to do the opposite, change a number to its string equivalent.

To use **value()** in your code, include **get.e** at the top...

```
include get.e

sequence s
atom num

s = value("45")

-- s will be a sequence with two elements.  In the first element,
-- there is an error code, and in the second element, the value.
-- The codes are GET_SUCCESS, GET_EOF, and GET_FAIL.  With either
-- failures, the second element will be 0, but with GET_SUCCESS, the
-- second element equals the atom value equivalent
-- s is {GET_SUCCESS, 45}

if s[1] = GET_SUCCESS then
    num = s[2]  -- num is 45
end if
```

The **value()** function can even work with mixtures, like "36P". As long as numerals stand before letters, then **value()** will recognize the numerals and ignore the letters, but only if numerals come first in the string, otherwise letters will transform to 0, no value.

```
s = value("36P")

-- s is {GET_SUCCESS, 36}
```

The BASIC programming language has a **VAL()** function that returns a 0 if no numerical value can be detected and the number otherwise. In Euphoria, you may duplicate this useful function by writing your own function based on **value()**.

```
function val(object parm)
    if sequence(parm) then
```

```
    parm = value(parm)
    return parm[2]
else
    return 0
end if
end function
```

You may use **val()** anywhere in your program following the definition above.

Now, the opposite transformation is also a common task in programming: changing numbers to their string equivalents. The **sprintf()** function duplicates the **printf()** function in detail with the exception that **sprintf()** prints the return value to a string or a text stream instead of the screen and only requires two parameters. Therefore, **sprintf()** uses two parameters instead of three because we are not printing to screen or other device. It is a side-effect of this function that numbers become strings, the function returns a string.

```
sequence str9

str9 = sprintf("%d", 45)

-- str9 is now, "45"
```

Changing Case

Euphoria provides **lower()** and **upper()** functions for changing the case of letters and strings, but you must include the **wildcard.e** file first. These two are functions that return the string in the proper case. If a lower case string is placed as a parameter to the **upper()** function, the function returns the string in all upper case letters. Even if the original string is a mix of upper and lower case characters, a fully capitalized version of the same string is returned. If the parameter is already upper case, it will not generate an error when the same string is returned.

```
include wildcard.e

str4 = "exit"
str4 = upper(str4)

-- str4 now equals "EXIT"

str4 = lower(str4)

-- str4 now equals "exit"

str4 = upper("Exit")

-- str4 now equals "EXIT"
```

Changing Strings

Not only can we analyze and slice strings, but we can change them, too. The secret is the use of the sub-scripting and concatenation features of sequences.

```
sequence Temp

Temp = "In the sun."
printf(1, "%s\n", Temp)

-- Replace the "I" with an "O"
Temp[1] = "O"

-- Replace "sun." with "road":
Temp = Temp[1..6] & "road"
printf(1, "%s\n", Temp)
```

Output

```
In the sun.
On the road
```

Combining Strings

Strings can be combined with the ampersand (&) operator. The string following the ampersand is attached to the string to the left of the ampersand, as shown in the next example.

```
a = "first string"
b = "second string"
c = a & b
puts(1, c & "\n")
```

Output

```
first stringsecond string
```

The process of joining strings this way is called "concatenation", which means linking together.

Note that in the above example there are no intervening spaces. There are a number of ways to supply the space. Such as:

```
b = " second string"    -- leading blank in b
```


More Data Types

In mathematics, a whole number with no fractional part is called an *integer*. We use integers constantly to describe age, attendance, and population to name a few. Most programming languages offer an integer data type in the interest of efficiency, compactness, and performance. Integers use about half the memory space required for floating-point numbers, and their operations are much swifter. This becomes important in programs that need maximum speed and in database applications where large amounts of disk space are devoted to storage. In fact, most professional programmers have a policy, *never use floating-point variables when integer variables will do*. When you follow this advice, you will appear more professional in your output.

The 31-bit technology behind Euphoria's integer variable is responsible for its great efficiency but this limits its capacity to little more than a range of 1073741823 to -1073741824.

Euphoria also offers a very special data type, the *object*, that is smart and flexible. The object can mold itself into any type of variable as needed. An object variable may start with integer values and shift to atom or sequence values. Any object may also be a string. A price is paid in efficiency when using objects. Unless some compelling technique asserts otherwise, it is best to save the use of objects for special situations.

However, in today's world where machines are very fast and memory is cheap, strategies for bullet-proof error-handling exploiting the properties of objects may well be justifiable. In such schemes, all variables are the object data type.

When you declare the parameters of routines to be object type, then the routines may be made so versatile as to accept any manner of data. The object helps provide robustness to programs by avoiding a mishap in the form of unexpected type mismatches.

The object shines as a safe repository for any unknown value. With an unknown value safely contained, it may be polled or tested in order to accommodate the immediate circumstances during program execution.

Both integer and object variables must be declared in much the same way atoms and sequences are.

```
integer my_age, attendees, num_voters
object buffer, inp
```

Even though two data types, atom and sequence, can handle any program, you now have four data types to choose from, atom, sequence, integer, and object. The integer and the object are used to make programs more sophisticated.

Testing for Type

Euphoria has functions to test and ensure the type of any unknown value. Not surprisingly, the test for atom type is the function **atom()** which returns true or false based on the data supplied as a parameter. In like fashion, there are the functions **sequence()**, **integer()**, and **object()**. The inclusion of **object()** is merely in the interest of completeness since **object()** must always return true, all values qualify as object.

Testing the data type prior to use of the variable adds robustness and flexibility to the code. It is central to the programming discipline known as *error handling* that makes programs rugged and reliable.

In common use, an object variable may safely claim an unknown value which is then tested for type by one of the above functions.

```
object x
sequence ln
integer fn

-- a string is returned by gets() until the end of the file
while 1 do
  x = gets(fn)
  if atom(x) then
    exit -- end of file
  end if
  ln &= x -- continue until ln has whole file contents
end while
```

| Test | Result |
|-----------------------|---------------------------------------|
| ==== | ===== |
| integer(45.678) | 0 (false) |
| atom(45.678) | 1 (true) |
| sequence({45.67, 45}) | 1 (true) |
| atom(1) | 1 (true) --an integer is also an atom |

Constants

When you refer in a program to a real number or a literal string, then these are categorized as *constants*, *literal constants*, or *literals*. Constants, like variables, are values, but unlike variables, they are not allowed to change.

In Euphoria, as in most languages, there is a declared data type called **constant** that has real value in sophisticated programming. This kind of constant is used like a variable is used, but it may not change value while the program is running. The value is established when it is first declared.

When critical values that never change, or that rarely change, are assigned once at the start of a program, then any re-writes that may take place later need only be changed *in one conspicuous location*. This is a chief purpose of constants.

The declaration is similar to other types, it must be above the code that uses it, but the declaration includes the assignment statement as well.

```
constant drinking_age = 21,  
        aspect_ratio = 1.77,  
        salutation = "Greetings"
```

Obviously, constants may be any of the data types, the assignment determines this, but the constant does not consume storage memory the way other variables do. Euphoria actually copies the contents to each location where the constant is used.

It is traditional to capitalize constants. We often see a collection of constants at the top of a program to help in adding clarity to the code that follows.

```
constant TRUE = 1,  
        FALSE = 0,  
        SCREEN = 1,  
        LOAN_LIMIT = 30
```

Writing Subroutines

Beginners make good use of trivial, even silly, program examples, but real programs of some size should be subdivided into multiple user-defined routines. If a program performs some task, then it is a task that may be subdivided into smaller tasks. Take the time, therefore, to divide your program into subroutines, both *procedures* and *functions* designed by yourself.

*Note: Subroutines might also be called **subprograms** or **modules**.*

To prepare yourself for this, contemplate the excellent procedures and functions that come built-in to the Euphoria language. Remember that the only real difference between procedures and functions is that functions return values and therefore must be used in an expression like a variable or constant.

Subroutines must be defined and recorded at some point in the code above the actual use. This is after the fashion of variables, because any attempt made to use a variable or a routine before its declaration results in an error, "...has not been declared" as if it did not exist. A typical program seems like a long list of routines followed by a few lines of initial code.

```
procedure toss_em(integer flag, sequence title) -- procedure
--declaration's first line
```

In a larger program, there may be a need for a function called **filter()** that can remove characters from strings as variously specified. The function will accept two string parameters. The first parameter, **text** is the string to be filtered. The second parameter, **filter_string**, is a string containing the list of "acceptable" characters. Inside the function, any characters in **txt** not found in **filter_string** will be removed from the final output. At function's end, **temp** is returned complete with changes, the final product. The example shows both the function declaration/definition followed by the program code that puts it to use. We also add the **val()** function previously explained.

```
function val(object parm)
if sequence(parm) then
  parm = value(parm)
  return parm[2]
else
  return 0
end if
end function

function filter(sequence txt, sequence filter_string)
-- Takes the unwanted characters out of a string by
```

```

-- comparing them with a filter string containing
-- only acceptable characters
sequence temp, ch
integer txt_len

temp = ""
txt_len = length(txt)

for i = 1 to txt_len do -- Isolate each character in
  ch = txt[i]          -- the string

  -- If the character is in the filter string, save it
  if find(ch, filter_string) != 0 then
    temp &= ch
  end if
end for
return temp
end function

-----
sequence a, clean_num

-- Input a line
a = prompt_string("Enter a number with commas.")

-- Look only for valid numeric characters (0123456789.-)
-- in the input string:
clean_num = filter(a, "0123456789.-") -- HERE WE ARE!!

-- Convert the string to a number:
printf(1, The number's value = d%\n", val(clean_num))
abort(0)          -- end normally

```

We wrote the functions **val()** and **filter()** ourselves, but we used them in code as if they belonged permanently to the language.

NOTE: When you declare and define a routine, if the routine accepts parameters, then the data type of each parameter must be declared as well.

--declaration and definition of 'filter()' function

```
function filter(sequence txt, sequence filter_string)
```

```
  edited = filter(form_entry, delimiters)  -- sample function call
```

The *type* Routine

Euphoria offers a special kind of function routine, the type function, that enforces user-defined data types and tests them for conformance. Instead of "function", the definition begins with "type" and ends with "end type", and it gives great latitude to design things your way. For instance, if you write a program that makes use of time and hours, then you might want to create an **hour** data type. Of course, it is not necessary -- you could use integers or atoms quite nicely, but if you make your own **hour** data type, you can enjoy an extra layer of error handling.

NOTE: Type functions, like atom() or sequence(), only return true or false. They must be so designed.

Because **hour** does not exceed 23 nor is it less than zero, hour variables *should not* exceed these bounds. By creating an **hour** data type, Euphoria will automatically check for you that the **hour** variables are legal.

```
type hour(integer x)
  return x >= 0 and x <= 23  -- this defines hour type
end type

-- now you can declare variables as hour types
hour my_time

-- if you try to put 30 hours in this variable
-- Euphoria will stop you with an error message.
```

One of the most useful uses for type is to make a string type of variable. Of course, you learned elsewhere that all strings are sequences, but not all sequences are strings. Because some routines fail when sent non-strings or nested sequences, we can profit from a string data type.

```
type string(sequence st)
  integer ln
  ln = length(st)
  for i = 1 to ln do  -- check each element
    if integer(st[i]) then
      if st[i] < 0 and st[i] > 255 then -- ASCII code range
        return 0  -- if any one fails, the whole seq fails
      end if
    else
      return 0  -- if any one fails, the whole seq fails
    end if
  end for
  return 1  -- If you got here, success
end type

-- now you can declare variables of the string type
```

```
string fname, lname

-- you can also test a variable for string credentials
if string(seq4) then
  seq4[1] = 32
else
  puts(1, "Not a string\n")
end if
```

Euphoria offers the option to turn off type checking in the interest of speed. When this is done, the user-defined type variables are not automatically checked. The checking is needed most during development to look for errors, after the errors are eliminated, then checking can be turned off to get maximum speed performance.

Passing Parameters (Arguments)

Subroutines, both procedures and functions, accept zero or more parameters (input) and return zero or more return values (output). If a subroutine does not return a value, then write it as a procedure. If you want the subroutine to pass back value(s) then write it as a function

NOTE: parameters are also called arguments.

*NOTE: The command **return**, is used to end the subroutine and return to the line following the subroutine call, but in functions, **return** is followed by a value to be returned from the function.*

When the subroutine is defined, each parameter pair (if there are any) is separated by commas. You may elect to have no parameters, in which case the parentheses remain empty. Whatever is decided in the beginning and defined in the definition is the rule that must apply to the use of routine. If a sub accepts three parameters, then three must be passed with each use. If the first is an integer, the second an atom, and the third a sequence, then the parameters must be passed in this order. The values must match the type specified in the definition. Each parameter must have a name (an identifier) like declared variables.

```
procedure print_x(integer dest, sequence patt, object items)

...
  <some statements to execute>
...

end procedure

-- In use...
print_x(3, "%d %d", {34, 56})
```

In the above example, when **print_x()** is invoked, it must accept three values, and the three values must match the data type specified in the declaration line, "procedure print_x(integer dest, sequence patt, object items)". If not, then an error is generated. The parameters **dest**, **patt**, and **items**, become variables for use in the subroutine only. They are variables that disappear once the subroutine is finished.

```
function change_all_three(integer dest, sequence patt, object items)

....
```



```

    <some statements that change the parameters>
    ....
    return {dest, patt, items} -- return ends the routine and returns
                                -- value, copies.
end function

-- In use...
retseq = change_all_three(3, "%d %d", {34, 56})
-- retseq is {56, "5789 575", {5732, 234}}

```

Parameters in Euphoria are passed by value, that is, even if a variable is sent as a parameter, only the value gets through, a copy of the value in the variable. The actual variable is not changed by what happens inside the subroutine. If the value copy is changed, the changed value can be returned if the subroutine is a function. By planning ahead, you can take the returned value and change the original variable if necessary. Functions have a wide latitude and can pass virtually any imaginable data structure.

If the main body of code has an integer called **loc**, a sequence called **key_seq**, and a sequence called **key_list**, they may be passed to the parameters just as the constants were passed above. The fact that the parameters have their own names does not mean that only variables with those names may be passed. In fact, **loc** may be passed through the **dest** parameter because it matches the data type, but the value found in **loc** is copied into the variable called **dest** while it is in the subroutine. Here is an example of passing variables that match the data type properly:

```

retseq = change_all_three(loc, key_seq, key_list)
                        ^   ^       ^
                        |   |       |
                        dest patt  items

```

In fact, by designing subroutines to accept an object, you can pass virtually anything as a parameter. If you pass a sequence, the sequence might be any length, which means you pass as many values as you like. In this example, all that is required is that you pass at least one variable.

Here is another example. It is a function from a program called, 21.ex, that simulates a card game of blackjack (21). The function, **count_hand()** takes three parameters.

```

function count_hand ( sequence hand, integer num )
-- This will inspect any hand to determine its score based upon the
-- rules of the game.

integer aces, card_value
tot = 0
aces = 0
card_value = 0

-- to keep score, count the value of the hand
for i = 1 to num do -- the parameter num is used

```

```

-- is it a ten-score card?
if find( hand[i][RANK], "TJQK" ) then -- the parameter hand is used
    card_value = 10

-- ace can be 11 or 1, which will be determined later
elseif find( hand[i][RANK], "A" ) then
    card_value = 11
    aces += 1
else

    -- in all other cases the suit does not matter for face value
    card_value = val( hand[i] )
end if
tot += card_value -- the parameter tot is used
end for

-- score aces favorably as a real player would
while tot > 21 and aces > 0 do
    tot -= 10
    aces -= 1
end while
return tot
end function

-- in the main code below...

```

The following example shows how the values may be passed to the function. The variables being sent have a different name than the parameter names found in the function's declaration. That is normal; they must only agree in type with the previously named parameters. It is their value that will be assigned to the parameters, **hand** and **num**. Inside the function, the corresponding values will be called **hand** and **num**.

```

player_score = count_hand( player_hand, player_cards)
                        ^           ^
                        |           |
                        hand       num

```

The variables **player_hand** and **player_card** can pass their values to function **count_hand** because they are compatible with the parameters **hand** and **num**. Do not name the parameters after the variables to pass. Name them what you will call them inside the function. They will have the same value, but they will be utterly independent.

In this description so far, Euphoria resembles many programming languages, but if you want more freedom and flexibility, then pass only one object or sequence and return an object or sequence (functions offer more flexibility since they return values as well). Within a single object or sequence you can pass virtually any conceivable series of values or data structures, but you must write your subroutine's code to handle the variations that are possible and reject anything outside your chosen range. This is limitless in the possibilities, well beyond the typical programming language.

If you want the rest of your program to reflect the change of values inside your subroutine, pass the values back through a function's return like a relay, and then distribute the values as you see fit.

Files and I/O

Note:

If a program (written in Euphoria or any other language) has a file open for writing, and you are forced to reboot your computer for any reason, you should immediately run scandisk to repair any damage to the file system that may have occurred.

Most general purpose programming languages allow the programmer to read, write, or create disk files of *text* type and of *binary* type, and in most languages, it is a tricky topic. It is intricate because a full file operation is not a single statement but a series of carefully placed statements that cooperate in series to achieve the result. Such operations begin with opening or creating a file and end with closing it. In between, data is retrieved in parts or in whole. Data may be written and saved in parts or in whole.

All file operations in any language are a silent cooperation with the operating system itself. The operating system is like a traffic cop who enforces rules of operation. A programming language is constructed to also enforce the rules of the operating system. For this reason, file operations in Euphoria for Linux differ slightly from operations for Windows or DOS.

Essentially, a file operation begins with opening a file with the `open()` function using the file's name or the file's *name and path*. The `open()` function will then return a number which we call the *file handle*. It is the operating system that has assigned the number. Thereafter, in all subsequent operations on the file, we will refer to the file by the *handle* (a number) *not the name*.

In most operating systems, there is a limit to the number of files that may be open at once, and though in modern computers that limit may be very high, it is a good idea to close the file by closing the file handle at the earliest convenience. If we are extracting information from the file, we store the information in variables and close the file before we analyze or a manipulate the data. We only keep the file open if we expect to immediately store some new information or some new form of the information to the disk.

With the `open()` function, we must also specify the mode of file opening because there are many to choose from, each with its own restrictions.

```
"r" - open text file for reading
"rb" - open binary file for reading
"w" - create text file for writing
"wb" - create binary file for writing
"u" - open text file for update (reading and writing)
"ub" - open binary file for update
"a" - open text file for appending
"ab" - open binary file for appending
```

We are specifying whether or not we will treat a file as a text file or a binary file, and we are further specifying if we intend to **write**, **read**, **append**, or **update** (write and read).

The **write** mode is dangerous but useful. *Anytime you open a file in the **write** mode, the old contents of the file are immediately destroyed.* If you try to open a file in the **write** mode that does not actually exist, then the **open()** function will create the brand new file with the name requested.

The **read** mode is very safe, but we will not be allowed to make any changes to a file in this mode. If you try to open a file (in the read mode) that does not exist, the **open()** function will return -1, the code for error.

The **append** mode is safer than the write mode. It always adds any new material to the end of the file so that old material is saved. Otherwise, append is like write, and if the file asked for does not exist, the **open()** function in **append** mode will create it with the name searched for.

The **update** mode allows either reading or writing, and opening in the **update** mode does not erase the old contents. New material is appended to the end of the file. The **update** mode will not create a new file. If the file asked for does not exist, the number, -1, is returned to signal a file error, so it is good practice to test the file handle to look for -1 and deal with the error quickly and efficiently. Failing to anticipate this error is one of the most common mistakes of beginning programmers. Any file mode may return -1 as the "file handle" if there is an error of some kind. Such errors are ignored at your peril. A -1 means no handle; the attempt failed, usually because the file you expected to find was not there after all.

A file in the **text** mode has a certain structure to the data. A **text** mode file is composed of string data, but the strings are called lines in the file. Each line ends with an end-of-line marker. In Windows and DOS, the end of line marker is a combination of two characters, 13 and 10 (ASCII) ("`\r\n`" or {13,10}). The end of the file is the end-of-file marker, character 26 (ASCII). Those Euphoria statements designated as text file operations assume this structure. Linux is similar, except that the end-of-line marker is the single character, ASCII number 10.

In truth, all files on the disk are binary files. A text file is merely a binary file with a common format that is friendly to text material. There is nothing at all illegal with opening a text file in the **binary** mode, but more care and effort are required to deal with it. The binary mode is more tedious. Opening a non-text file in the text mode is usually disappointing, though.

A file is a long series of bytes. Each byte is a number from 0 to 255. How the bytes are arranged and interpreted is the single difference between a binary file and a text file.

A Microsoft Word file with a *.doc ending is a text file in name only. In truth, it is a

densely formatted binary file with a complicated, proprietary format. That is why it does not end in *.txt*. An informed programmer can write a program to read Word documents, but the programmer will begin with the binary mode. Whenever you need maximum file power, or whenever you need to open a file with no preconceptions, or whenever the file is a complicated or proprietary structure, the binary mode is the way to go.

In the binary mode, you are not much concerned about advancing through the file one line at a time. Instead, you advance one byte at a time. As you access each byte, the file pointer is moved to the next byte. You may deliberately move the pointer yourself to any byte in the file with the `seek()` function. You may report your position at any time with the `tell()` function, but the answer comes in numbers of bytes since the beginning of the file.

In binary operations, besides moving one byte at a time, you may move in multiples of bytes using `get_bytes()`.

The single most important file skill is the ability to write code that persistently watches for the end-of-file marker. Continuing past the end of file marker yields meaningless, undefined information that is unpredictable and unwanted.

The choice of input and output statements you decide to use in your file operations usually depends most on whether you are treating a file as a text file or a binary file. The same input/output statements that write to the screen and read from the keyboard will also write/read a disk file. The only difference is that instead of 1 for screen in the first parameter, you place the number that is the file handle in the first parameter. The `puts()` function will print a string to a file just as it prints a string to the screen if you put the file handle variable, in this case, `fh1`, as the first parameter.

```
-- print ln to screen
puts(1, ln & "\n")

-- print ln to file, fh1
puts(fh1, ln & "\n")           -- DOS will convert "\n" to "\r\n"
                               -- in text files, be sure your string
                               -- ends with newline character.
```

The statements that allow you to retrieve one character at a time from the keyboard and print one character at a time to the screen may be used for file input and output in binary files. Revisit all input and output statements in the Euphoria language and rethink how they are useful for file I/O. In fact, these are the file statements to use.

NOTE: Most operating systems treat devices as files, so most languages can treat devices like files. That is why the form for printing to a file is like the form for printing to the screen which is like the form for printing to a printer. There may be different methods for printing, but usually one method is available that mimics the print-to-screen form.

Printing integers and floating point numbers to files yields a series of bytes for each number. Floating point numbers will consume eight bytes each. Use `atom_to_64bit()`

functions to create IEEE standard variables for storage and shearing with other languages.

Some languages offer another file type called *random access* files. These are binary file methods that handle record variables in a manner similar to many databases. Since Euphoria does not use record variable data type, there are no random access files as such. Random access file methods are difficult to learn and use, and they are quite primitive when compared to a real database. In Euphoria, we are offered the EDS database engine that comes free with the package as a worthy substitute to random access files.

*NOTE: Later you will see how to cobble your own form of random access file with **get_bytes()**.*

The EDS database is a much better choice than any random access file method. It is equally easy to learn, and the performance, efficiency, and flexibility far exceed what is possible from random access files alone. It is highly recommended that you learn all you can about the EDS database even though it is technically an optional (free) add-on. Euphoria has only binary files for random access, but the EDS database is a far superior substitute.

If you need to write programs that manipulate random access files created and manipulated by other programs written in other languages, you can construct routines that add this capability to your Euphoria program using binary techniques with byte compatibility techniques if you are privy to the file's structure. If you are *not* privy to the structure, then no other language will be any help either.

A random access file (not available in Euphoria as standard equipment) is composed of equal sized blocks of bytes. Each block is divided into predetermined and predefined segments. Each segment may be freely interpreted as any native data type, but the match must be compatible. The purpose is to combine differing data types with a predictable geometry that allows the program to pluck a block from anywhere in the middle of the file without searching through the whole file.

Obviously, you can create your own using binary techniques, but I recommend the EDS database instead.

Strictly speaking, binary mode is random access, but the popular use of the term implies a block structure full of aggregate fields with each block the same length and each movement is a block at a time instead of a byte at a time. The format must be consistent and predefined to be useful.

With the EDS database, in sophisticated programs, the files will be more compact overall and rapid access does not depend on uniform consistency of data. The EDS database is also a reasonable substitute for an ISAM database with a built-in index manager. It is very sophisticated indeed.

In the beginning, you will probably use text mode files the most. Most sophisticated programs are empowered with some kind of file access, and text mode access offers sufficient sophistication for most purposes, and the text mode is easier to comprehend because it resembles the most common input and output statements in general programming. One merely adds the file as the target and one watches for the end-of-file marker.

The object variable is essential in text file access because the object is prepared to accept any data from the disk without any preconceptions. It is prepared for any unexpected event. As you read lines from a text file, you are reading in strings, but when you finally reach the end-of-file marker (and you will), then you are reading a single atom. The variable that receives file input, therefore, should be an object variable because an object variable can perform equally as either a string or an atom. After each individual read, the object is tested to see if it is an atom. An atom in a text file is a signal of the end of the file.

```
object inp

[ .... ]

gets(fh1, inp)
if atom(inp) then -- end of file
  exit          -- quit the input loop
end if
```

NOTE: Each string you receive through `gets()` will end with `"\n"`. If you wish to remove the newline character, it is always at the end. In Windows/DOS world the `"\n"` is actually a two-character series, ASCII ten plus ASCII thirteen, `({10, 13})`. In Unix/Linux, it is ASCII ten alone.

```
line = line[1..length(line)-1] -- Unix/Linux
line = line[1..length(line)-2] -- Windows/DOS
```

The **gets()** function reads one line (string) at a time and advances automatically to the next line in order. That is, the first call retrieves the first string. The next call does not retrieve the first string again. Instead, the operating system is keeping track of a file pointer. The **getc()** function that gets a byte each time also advances automatically in the same fashion but only by a byte at a time. The **get_bytes()** command enjoys the same automatic advancement but by a specified number of bytes each time.

NOTE: These commands are inside a loop because we do not know if the file is long or short. If it is long, we must continue loading piece by piece until the end of the file -- hence a loop.

It is rare that you are sure of the file contents. The file may have been created by another program. For this reason, the most common file operation is to load a entire text file one line at a time, eventually into a large sequence, until the end of the file, and then to

analyze the data to find out what the contents are once they are loaded. Lines become strings. The end-of-line markers should be stripped away. You then might edit the strings and save them again to edit the file. When you load them in total and save them in total, you write with the write mode because you want to erase the old contents. **while** loops are essential for such operations.

The file operations just described may be summarized in comments.

```
-- while not the end of the file
-- input the next line into an object
-- test the object to see if it is an atom
-- if yes, then exit the while loop
-- otherwise, append the object to a growing sequence
-- when the end of file is reached, the entire file is now
-- contained in a sequence in memory, and the file boundary
-- was never breached.
```

NOTE: You might plan a section by writing the comments in advance. This clarifies your thoughts, and the comments are used to accompany the code that you will write accordingly.

With binary files, the variable accepting input is an integer because only one byte at a time is input. We periodically test the byte for character 26, end of file. With binary files, you may want to work random access. You may not want to load the entire file, but you may want to change the character at position 127. In that case, you first seek() to 127 and print. The one and only character was changed at the exact byte position. You can also seek() followed by a read.

What is not obvious in these operations is the buffer Euphoria employs silently behind the scenes. A buffer is a holding area for bytes, and file buffers are typically sized in multiples of 1024 to use the disk structure most efficiently. Sizes from 4K to 16K are common. When you access bytes from a binary file one byte at a time, there is not one disk read for each byte. The first read will scoop a thousand or more, if the file is that long, or it will scoop the whole file if it is shorter. Subsequent byte reads are then read from the buffer, though it appears to the programmer that they are read from the disk surface. A similar situation exists for writes. When a file is closed, all pending writes that may be sitting in the buffer are flushed to the disk to ensure that nothing is lost. This happens automatically without your awareness. This happens to ensure that disk reads are kept to a minimum for maximum speed and minimum wear.

There is a flush() command that allows deliberate flushing of the buffer manually. In typical use, if there is a text file in the current directory that you want to "read", then open it with "r":

```
integer fh1
object line
sequence pages
```

```
fh1 = open("sometext.txt", "r")-- "sometext.txt" is filename, "r" is
```

```

--readtext
if fh1 = -1 then          -- -1 only if failed, otherwise
                        --filename
    puts(1, "File not found\n")
    abort(1)
end if

-- if you get this far, fh1 is file number

```

In "r" mode, you can only input the contents of the file. You are not allowed to write to the file. The whole purpose of using text files to begin with is to load them one line at a time. This calls for a text-only command, gets(). We no longer use the file name, only the file number, here it is **fh1**:

```

pages = ""              -- initialize preparing for append
while 1 do              -- we will exit elsewhere
    line = gets(fh1)    -- line is an object; may be sequence or atom
    if atom(line) then  -- when line loads an atom, end of file
        exit           -- stop scooping at end of file
    end if
    pages = append(pages, line) -- pages accumulates the lines
end while -- once you finish this loop, you have the entire file in
--pages
close(fh1)             -- close it quick

-- or --

pages = ""
line = gets(fh1)
while not atom(line) do
    pages = append(pages, line)
    line = gets(fh1)
end while

```

In the above examples, the sequence **pages** contains the contents of the file when the operation is over.

The essential input/output commands that you will mostly likely use for file use (or device use) are:

```

gets(fh) -- function that returns the next string from a file or
-- keyboard

getc(fh) -- function that returns the next byte from the file or
--keyboard

get_bytes(fh, i) -- returns a specified number of bytes
puts(fh, str) -- prints string, str, to file or screen
puts(fh, 'a') -- prints a single byte to the file or device

```

The major alternative with text files is the path. If the file you seek is elsewhere, open with the complete path. Remember, use "\\" instead:

```

fh1 = open("c:\\mydocs\\olddocs\\inventory.txt", "r")

```

If you loaded a text file in order to change or update it, then your changes are made in memory, in pages. To place your changes in the file, you will open in write mode. You already closed it above. Write mode will destroy the current contents of the file, but you aren't worried because you have them currently in memory with your changes. Now:

```
fh1 = open("somettext.txt", "w")
if fh1 = -1 then
  puts(1, "Could not open file\n")
  abort(1)
end if
```

To place lines in the file use the puts() command, the same command used for writing to the screen. Instead of 1 as the first parameter, it is the file handle in fh1.

```
-- DO NOT try to puts() nested sequences
for i = 1 to length(pages) -- This may be longer or shorter than
  --before
  puts(fh1, pages[i]& "\n") -- but you want to save the whole
  --thing
end for
close(fh1) -- close it quick
```

When you open a binary file, you have no preconceptions, and you want every character treated equally, end-of-line, end-of-file, whatever. You will probably load one character at a time:

```
integer fh2, c, char, at
sequence stream, chunk

fh2 = open("somebytes.dat", "ub") -- Update Binary is popular mode

stream = "" -- initialize preparing to append
while 1 do -- exit this loop elsewhere

  -- file pointer advances with each get, here one at a time
  c = getc(fh2) -- get next character
  if c = 26 then -- if character 26, then end of file
    exit -- don't go past end of file
  end if
  stream &= c -- otherwise accumulate bytes in
  --stream
end while
```

The prior example shows how to load the whole file. If you wanted only the character found at location 1043 then:

```
seek(fh2, 1043) -- move file pointer to position 1043
c = getc(fh2) -- get the one byte there
```

When time comes to replace the whole file with new, changed file. Notice that puts() is equally useful for binary and text files:

```

seek(fh2, 1)           -- put file pointer to beginning of file
for i = 1 to length(stream) -- may longer or shorter than before
  puts(fh2, stream[i]) -- putting one character at a time
end for
close(fh2)            -- close it quick

```

If you only want to change the character at position 1043:

```

seek(fh2, 1043)       -- again, put pointer at 1043, to be
                      --sure
puts(fh2, char)       -- put what you got from char
close(fh2)            -- close it quick

```

Occasionally, when you are loading a binary file, you may want to load a fixed chunk of bytes at a time instead of one at a time. That is what so-called random access techniques do. For this, use `get_bytes()`:

```

stream = ""
chunk = ""
while 1 do                -- will exit elsewhere

  -- file pointer moves with each get, here it advances 144 places
  chunk = get_bytes(fh2, 144) -- get next 144 bytes into chunk
  stream &= chunk -- append the way you like, chunk might be empty,ok
  if length(chunk) < 144    -- won't go beyond end of file
    exit                    -- stop gulping bytes past end of file
  end if
end while

```

The last gulp might be less than 144 (or whatever fixture) because `get_bytes()` won't go beyond end if file. Eventually, `get_bytes` will return empty sequences.

To put data in binary files in fixed-number chunks (just like so-called random access), `puts()` is again useful.

```

if length(chunk) > 144 then -- be sure not longer than 144
  chunk = chunk[1..144]
end if
at = find(26, chunk)       -- don't put eof marker accidentally
if at then                 -- or not, it depends on your intention
  chunk = chunk[1..at-1]
end if
puts(fh2, chunk)          -- put it now, wherever you are in the
                          --file

```

=====

REMEMBER: Use a separate variable to load a part at a time and a separate variable to accumulate the loading. If it is a text file, use an object to load each line, so it won't fail when it loads an atom at the end of the file.

AND REMEMBER: You can't open a file for reading "r", and then open it

for writing "w" without first closing it. You may get a new file number every time; you may not. Its up to the operating system.

=====

SUMMARY

=====

```
get()          - gets a Euphoria object, {,,,}
getc()         - gets one byte at a time, after ENTER
                returns -1 after CONTROL + Z
gets()         - gets one string at a time, string has "\n" on end
get_bytes()    - gets one or more bytes at a time.

print          - prints a Euphoria object in object form, {,,,}
puts()         - print bytes or strings
printf()       - formatted printing
sprintf()      - returns string version of sequence.
```

You cannot be interested in files without being oriented to directories. Euphoria has a rich set of commands for dealing with directories. You can move among the directories calling where to go or reporting where you are. You can get from the operating system the current location. You can load all the file names and file statistics in any directory you wish. You can search for multiple file names that fit a wild card search pattern, or you can search for one at a time by name. You can even use walk_dir() to search an exhaustive pattern among directories and their sub-directories using a custom operation on each file that fits the target pattern. See search.ex in c:\euphoria\bin.

dir()

The dir() command is a function that returns either all the files in the named directory (defaults to current directory) or only those files specified. If dir() returns an atom, the search came up empty. If multiple file names and statistics are returned, dir() returns a sequence sophisticated enough to hold the material. The length of the returned sequence is the number of file names that fit the search pattern. Each sub-sequence within each nested sequence will contain separate statistics about each file.

In truth, you will rarely need all that information, but you can use subscript notation to access only the statistics that you need. See library.doc for details.

current_dir()

You usually know where you are, but the program you finally distribute may not once it is installed at will on the user's computer. Using this command, your program can be informed about the current path. It is a function that returns a string naming the current path.

walk_dir()

If you need to run an exhaustive operation on a large number of files that meet your requirements, in sub-directories, too, then this command is your ticket. You can write a procedure to your specifications that does what you need to each file, and walk_dir() will

then empower your code to operate on multiple files. See search.ex in c:\euphoria\bin.

```
system()
```

Use this command to let DOS do the work. Using this command runs a separate version of command.com that lets you use DOS commands in quotes. When finished, command.com is unloaded and your program resumes. You can also use this command to run another program. Your program is suspended in the mean time, but it resumes when the other finishes.

```
system_exec()
```

Many programs are designed to return an exit code when they finish. When they run well and finish successfully, zero is returned, but when there is trouble, another number is returned. Your programs in Euphoria can do the same when you use abort() with a parameter other than zero. system_exec() allows you to receive that exit code when the branched program finishes. Your program can then detect if everything went well while it was away. You can also receive exit codes from DOS operations.

This is always a good place to consider getting input from the keyboard. gets() gets a string, but looks for CONTROL + Z to determine string's end. prompt_string() gives prompt, gets string. getc(0) gets byte from keyboard. get_key() gets next byte waiting in buffer or next byte typed - NO WAITING. wait_key() waits for next byte to get byte (compatible with multitasking). getc(0) also waits for keyboard. get_key() is usually used within a loop to poll for responses.

The locking or sharing of files is under the direction of the operating system. Most file openings are in exclusive lock mode. If you try to open a file already open, it will be disallowed in this mode.

Control Flow Structures

Even though it is very easy to write programs that always proceed exactly the same, it is more useful to control and change program flow. While a program is running, the order of program statements can change drastically based upon logic and its response to the immediate situation. The **if** statements in their various forms are very useful in this regard. Some **if** statements can be very involved. They can be expanded or nested.

```
if A<= 50 then
  if B <= 50 then
    puts(1, "A <= 50, B <= 50\n")
  else
    puts(1, "A <= 50, B > 50\n")
  end if
else
  if B <= 50 then
    puts(1, "A > 50, B <= 50\n")
  else
    puts(1, "A > 50, B > 50\n")
  end if
end if
```

Larger block statements can be employed.

```
if X > 0 then
  puts(1, "X is positive\n")
  PosNum += 1
elsif X < 0 then
  puts(1, "X is negative\n")
  NegNum -= 1
else
  puts(1, "X is zero\n")
end if
```

Compound expressions expand what the statement can do.

```
if find(compare(C, "A"), {0,1})
and find(compare(C, "Z"), {0,-1}) then
  puts(1, "Capital letter\n")
elsif find(compare(C, "a"), {0,1})
and find(compare(C, "z"), {0,-1}) then
  puts(1, "Lowercase letter\n")
elsif find(compare(C, "0"), {0,1})
and find(compare(C, "9"), {0,-1}) then
  puts(1, "Number\n")
else
  puts(1, "Not alphanumeric\n")
end if
```

At most, only one block of statements is executed, even if more than one condition is true. For example, if you enter the word "ace" as input to the next example, it prints the message "Input too short" but does not print the message "Can't start with an 'a'".

```
check = gets(0)  -- keyboard enters string
if length(check) > 6 then
  puts(1, "Input too long\n")
elsif length(check) < 6 then
  puts(1, "Input too short\n")
elsif equal(check[1], "a") then
  puts(1, "Can't start with an 'a'\n")
end if
```

Here is another example of nesting.

```
if X > 0 then
  if Y > 0 then
    if Z > 0 then
      puts(1, "All are greater than zero\n")
    else
      puts(1, "Only X and Y are greater than zero\n")
    end if
  end if
end if
elsif X = 0 then
  if Y = 0 then
    if Z = 0 then
      puts(1, "All equal zero\n")
    else
      puts(1, "Only X and Y equal zero\n")
    end if
  end if
end if
else
  puts(1, "X is less than zero\n")
end if

-- or --

TestValue = get(0)  -- get number from keyboard
if find(TestValue, {1,3,5,7,9}) then -- search a list
  puts(1, "Odd\n")
elsif find(TestValue, {2,4,6,8}) then -- search a list
  puts(1, "Even\n")
elsif TestValue < 1 then
  puts(1, "Too low\n")
elsif TestValue > 9 then
  puts(1, "Too high\n")
else
  puts(1, "Not an Integer\n")
end if
```

In many programs, the entire program is supported by a **while-loop** that has a complicated **if..then..else** that branches to one in many routines. When the loop finally exits, the program is over. All or none of the routines may execute based upon the

changing variable values.

```
while 1 do
  retval = get(0)
  if retval = 1 then state_of_business() then
  elsif retval = 2 then pause_for_input() then
  elsif retval = 3 then notify_network(today) then
  elsif retval = 4 then check_clients_file(X, Y) then
  elsif retval = 5 then wrong_choice() then
  else
    exit
  end while
abort(0)
```

Scope

Scope refers to the territory of variables, procedures, and functions. It refers to the lifetime of variables. In Euphoria, there are three levels of scope: *global*, *local* and *private*. Look at the following program.

```
<Program>-----  
  
include get.e  
  
global sequence glo    -- This is a global variable  
atom num              -- local variable  
sequence str, arr     -- local variable  
  
-- private not valid here  
-- local and globals above valid here  
  
procedure get_better() -- get_better() is NOT a global procedure  
integer cnt           -- cnt is private variable inside procedure  
  -- {...}           -- local and globals also valid here  
end procedure         -- cnt ends here  
  
procedure main()      -- main() is NOT a global procedure  
sequence str         -- str is private variable inside procedure  
  -- {...}           -- different variable from local str  
  -- local str is ignored  
end procedure        -- private str ends here  
  
-- private not valid here  
-- local and globals above valid here  
  
main()  
  
<End  
Program>-----
```

Variables declared inside subroutines are private. They last only so long as the subroutine is called. They are created when the sub begins, and they are automatically destroyed when the sub ends. They are not seen or recognized by any code outside the subroutine. A professional programmer will try to make as many of the program's variables private variables wherever possible by declaring them inside subroutines if that is the only place they are needed.

Variables declared at the top of the program outside of any subroutines are local variables. They are seen and recognized everywhere in the program module (from there

to the end of the file), but they are not seen or recognized in any include files that may be included.

Variables declared at the top of the program outside of any subroutines that are declared with the word, "global", are global variables. They are seen and recognized everywhere in the program and in any include files past the point of their declaration.

If you are writing code for an include file, only global variables in the code will be recognized by the programs that include the file. Local and private variables in an include file remain invisible and invalid to any program that includes the file. The include file is included at the top of the program, so any global variables or subroutines in the include file have maximum scope. A wide scope is the only way to share variables and subroutines between include files and programs that include them.

If all the variables in your program were global, there would be no scope issues to think about, but there would be vulnerability and limitation in the program design. It is a bad idea to make all variables global just to avoid scope issues.

*NOTE: Think before you declare anything "global". Chances are, it is better **not** to make it global except for the mentioned exceptions.*

It is always good practice to make a new variable's scope as small and limited as possible. Make as few variables global as possible.

This is not obvious now, but as you design large and complicated programs, the benefits of this approach will be revealed.

Also, in Euphoria, a variable is only seen and recognized in the code that follows its declaration, so if a variable is declared later in the code, it will be invisible and invalid in the code prior to that even if it has a wide scope.

When you try to use a variable beyond its scope, Euphoria interprets it as invalid and undeclared. Its use will trigger an error as if the variable or subroutine did not exist.

Memory management is good practice for any programmer, but Euphoria makes it easy and reliable. However, you do your part to increase the efficiency of your programs when you use private variables as much as possible. The scope of a variable not only specifies its territory but its lifespan as well. Private variables consume memory only so long as they are needed. The results are nearly automatic if you have properly assigned scope to begin with.

Remember: You define scope by where you declare the variable as well as with any descriptors like "global".

Subroutines also have scope, either global or local. This becomes important to libraries

(program code with the *.e or *.ew extension [*eu in Linux]). You can create your own libraries (include files) by starting a new program. Place subroutines (procedures and functions) in your library that you want your library to have, but the only main executable code in the program is test code for testing the routines. Once your testing is finished, erase the code beyond the subroutines and change the name to a new library name. Libraries for DOS have *.e endings in the names and libraries for Windows end in *.exw.

However, none of your subroutines can be shared with another program when your library is included unless some of the subroutines were declared with the "global" keyword. When making a library, be sure to declare all procedures and functions that you want to share with other programs with global:

```
global function do_my_thing(integer self, object parm)
  -- [ *some code here* ]
end function
```

```
GLOBAL!!! function
```

NOTE: The library parts share with each other in the same way different parts of a program share. They are "local" to each other, but they are private to the program that includes them. Those items that are global are shared outside the library as well.

Only the global routines in an include file can be called by the other program that includes the file. This offers vast possibility to enforce controlled and designed scope. The rule is: hide as much code as possible. Let any other code know only what it needs to know about the code and data that are shared.

*NOTE: This tendency to hide one part from all the other parts gives rise to **abstraction**. With extra abstraction, you deal with the big picture without being distracted by extraneous details that have been already handled.*

Also any global variables in the library will be shared, which may or may not be a good thing. Consider carefully which variables need to be global and hide as much as possible from the programs that will later share the library.

You can also take this a step further with *namespaces*. A namespace identifier follows the rules for variable identifiers (names). When you put an include statement at the top of your program, you can add "as such_n_such" to it:

```
include mylib.e as john
```

Later, when you use something from the library, it must begin with the namespace identifier and a colon (:).

```
john:calc_num = 890.23
```

```
- or -
```

```
john:run_swap(4, up_count, biquad)
```

This adds new possibilities for program scope, particularly for large projects. This can also be used to disambiguate variable names and routine names. Two variables can have the same name and be different if one variable is in a library that was loaded into a namespace.

```
total = check + gather + other      -- total local to program only
john:total = fee + options + other  -- total local to library, but
                                   -- shared by namespace without
                                   -- a need to rename the variable
```

Professional programmers working on large projects with teams of programmers will build libraries or objects of code that are useful again and again. Any code that may be useful in many other programs should be in a library, but not knowing what the future holds means that naming conflicts can occur when future code uses the libraries. By using the smallest scope possible whenever possible and by using namespaces, such conflicts can be avoided or kept to a manageable minimum.

The pain and misery of past mistakes cause professionals to be very strict about scope. They learn to think about scope and keep it within the smallest boundaries possible. It is good to start early learning the habit.

Databases

As a student programmer, you are familiar with disk files of various types, but to learn about databases, you must understand the difference between a database and a file. The definition of a database is not very strict and it has evolved over time, but it includes files with a system or architecture that increases the efficiency of storing and retrieving data from a hard disk. Files that are not quite databases are merely *flat files*.

Databases include files, but the files have a special structure and a manager program (engine) that can search files quickly and efficiently. Usually, a special smaller file called an *index* is used. Sometimes all files and indexes belonging to the files are meshed into a single huge file with internal compartments.

An index works like a card catalog in a library. It is usually a smaller file with a look-up list. In simplest terms, an index contains a list that cross references each record with a record number or disk location. It is quicker to search the index for the clue. Once the record number or disk address is known, it is very, very quick to retrieve the information. Indexes have a well-deserved reputation for speeding up data access.

NOTE: It is often not understood how quickly an item on the disk can be retrieved if the address or record number is known. It is virtually instantaneous. So, the object becomes to find the address or record number as quickly as possible which is what an index does.

A single database file is made of many records. Usually each record has a similar structure. It is the similarity of the records that makes swift navigation possible. Usually such a file is quicker and easier to search than a flat file. We refer to this structure as a *table*. A telephone book is structured like a table.

There are several famous database products that enjoy market dominance and that are used in thousands of businesses and institutions. Such champion databases are very expensive and very complicated, and their level of complexity and sophistication is not always needed in many situations. A good database skill is the ability to match an engine with an application. With the very best databases, an extra well-paid employee must be hired just to manage the database. The very best databases are often not required in the small to medium business world, but much of what you read about database technology is aimed at the largest organizations.

Most really useful business programs need some form of database in the structure of the program. Professional programmers can attach their programs to one of the big database engines, but there is a price to pay for this. It is often better to use a less sophisticated engine and enjoy a better economic return and a faster development cycle.

The EDS database engine is supplied free from Rapid Application Software when Euphoria is downloaded or installed. This database is frequently powerful enough for many ordinary applications. It is compact in size and easy to operate. The code can be added to your own program code to make your programs much more powerful at no extra cost.

The EDS database is similar in performance to an ISAM database with an index manager. EDS uses the first column (field) as an index that is sorted and searched with a binary search algorithm. It is difficult to see any difference in this level of performance and that of a top-notch ISAM engine.

If you may need more than 2 billion records per table, then you should consider some database other than the EDS database, but 2 billion records is a lot, and the EDS database can be very small when necessary.

If you need to share the same database files with many operators simultaneously, you may need a more sophisticated database engine. The EDS database cannot lock a single record, but it can lock the entire table.

Other programming languages usually support random access files, but Euphoria does not. The EDS database is a very good substitute for random access files.

ODBC was an initiative by Microsoft to make all database engines accessible to all development languages. It has matured and is now quite excellent. It is based on the fact that most engines are relational types that understand Structured Query Language. Using an ODBC library available from the RDS website, you can connect to a .DLL that allows you to write database applications in Euphoria for every major Windows database engine and most of the minor ones. You should bone up on relational database technology and SQL in order to capitalize on this, but through ODBC, Euphoria is a major player. This is serious business computing power that competes well with all other technologies.

You can work through a programmer's Application Programmer's Interface (API) with most products as well. There are libraries and .DLL's that allow powerful use of popular tools like MySQL and Advantage Database Server (not to mention many others). It links Euphoria to the products as if they were designed for each other. In theory, this is higher performance than ODBC, but in practice there is not so much noticeable difference. MySQL and Advantage Database Server are two such high performance products that are very economical without sacrificing much quality or performance. Most API's are published for C/C++, Delphi, Visual Basic, and the various .NET products, but an experienced programmer can alter these files to Euphoria code, and many have. You may find such tools available for free from the RDS website. Without much effort, I translated an Advantage Database Server include file (*.h) for C/C++ into a Euphoria include file that allows Euphoria to operate the API with equal ease. This was possible due to my familiarity with C and due to the design of Euphoria.

The EDS Database

RDS, the makers of Euphoria, offer the EDS database engine, written in Euphoria code, to replace and surpass the typical random access file mode. It is a true database engine that is speedy, flexible and efficient with all but the most massive corporate files. In small businesses, it is unlikely that a database will grow so large that it overwhelms the capabilities of the EDS database. In any case try it. You can always step up later.

The EDS database is easier to learn and easier to master than random access file methods. To add this capability to our program, we must include database.e include file, which should be installed first in the include directory. Once this file is loaded, the engine is on board, and the commands are available for use. First, the commands must be learned as new additions to the language.

NOTE: database.e should already be installed in the include directory with this installation.

NOTE: With any include command at the top of your code, Euphoria looks first in the current directory. If the file named is not found there, then Euphoria looks into the c:\euphoria\include directory.

A basic database is a table with one or more records, and each record has one or more fields. The records are often also called rows, and the fields are often called columns. A telephone book is a good example of a table. Like a typical database, each record in the phone book is a different customer. Two customers may have the same name but not the same phone number or address. The fields of each record are name, address, and phone number. There may be a million records, but each record has a few fields only.

The distinguishing field is the phone number. No two customers have the same number in simple theory. In practice, we may have different names listed per household, but in our simple example, we will ignore that. For now, accept that each customer has a different phone number.

This qualifies the phone number field as a primary field. The name field is not reliable as a primary field because two customers may have the same name. If we want maximum reliability, we may assign each customer a customer ID number that is guaranteed unique. When we do that, we have added another field that will become our primary field. Another field that is important or distinctive, though not exactly unique, might be called a secondary field. The name field for instance.

With the EDS database, we are free to use a different number of fields per record, but that is not generally done. If we assign the same number of fields and the same structure to each anticipated record, then we need to identify if any field qualifies as a proper primary

field. If not, then we will invent one--perhaps a record ID number. If the records hold customers, then it is a customer ID. If the record holds students, then the number is a student ID.

In EDS, this field, the chosen primary field, will be called the key, the value found in this field in each individual record will be called the key data. The remaining fields will be called record fields and the data they contain are record data.

We use the key for searching, but otherwise it is not different from the record fields. We will be lucky if we have a primary field to choose from without creating our own because the database will then be smaller overall.

Every record has a number that is the simple count of records from beginning to end. This may be called the record number or the physical number. The first record added to the file will always have record number one, and the last record added to the file will always equal the number of records in the file. When we insert a new record, it will be appended to the end of the file with a record number one higher than the previous highest number.

File operations concentrate on the key and the record number. If we always knew what record number we needed in advance, we would not need a key. The key is a shortcut to finding the record number. ONCE THE RECORD NUMBER IS KNOWN, RECORD ACCESS IS NEARLY INSTANTANEOUS. This is true no matter how large the record number because the access method is random access.

EDS allows us great flexibility and freedom. We may use any Euphoria data type as the key, or we may use several, but usually we want the smallest key that can do the job. EDS allows any data type as the remaining record data, but typically we use a sequence with several elements. Each element in the sequence will then be a different field. EDS allows any field to be any length and any data type, but typically we want the same number of fields in each record and every corresponding field should be the same data type from record to record. We may mix data types, but the third field in each record should have the same data type, and the first field in each record should have the same data type, and so on...

Ideally, the key is one of the fields of interest as well and, ideally, it is a fairly small field. The typical database will have one small key and several fields in the record data section. Most of the data is record data. We use the key to get key data, but most of all, we use the key to get the record data.

If we want to simply load the whole database into memory, no key search is required. We simply request the number of records (returned by `db_table_size()`), and using that number, we make a for loop that loads each and every record by number from beginning to end (key and record alike). This is not usually done, however. Most often, we will search for one record in many, and that is what the key is for.

The simplest database that we have described in best called a table. A more sophisticated

database may have more than one table.

Before we use the command to create a database to begin with, we should plan and think about the database design. We should plan what data structures will be used for the key and record data, and we should identify our primary key. Once the structure for a single record is determined and defined, then we are ready to create a table. Before we can add a table, we must create a database to hold it. The database may have only one table, or it may have many, but we will design and add them one at a time. The first time we open a database is when we create it. Thereafter, we will simply open.

We must create a database before we can add the first table. EDS requires a name and a share mode. The second parameter is typically `DB_LOCK_EXCLUSIVE` to indicate the exclusive mode. The first parameter is the name you have chosen in quotes and possibly combined with a path. A number is returned signaling either success or some kind of error. If an error is returned, the database was not created. Once created, it is automatically open. Once open, we may create a table. The first time we select a table is when we create it, but thereafter, we will simply select it. There is no command for opening a table; you select one among those in the opened database. The `db_create_table()` command requires only a table name that we have chosen in quotes. It will return a number that either indicates success or some kind of failure. If it does not return success, then the table was never created. No two tables in the same database may have the same name. Once the table is created, we are free to begin entering data one record at a time, and we may add records periodically over time.

The data in the key must be different for every record. If two records have the same key value, a database error will occur. Do you see why you need to think the design over first? The key functions as an index, and the search is a very fast binary search.

To add the next record, a key search is not required because EDS will merely add it after the last record, so that it now becomes the last record. This process is called an insert. We are also free to delete any record.

What follows may be an extended period of data entry where we add, perhaps, hundreds of records. If we have data already stored in a consistent format in a text file, we may write a simple program that reads the text file, converts the data, and places it in individual records in our new EDS database. A well designed database full of data is usually very valuable to the creator. It may have a lifespan of many years where many more records are eventually added.

If our database is meant to be a log that accompanies one of our programs, then we may begin running the program that opens the database and stores each logging as an individual record.

If we want, we may also add new tables to the same database.

The primary actions in database terminology are insert, delete, seek, update and query. Before we can perform these actions, we must open the database and select the table.

Once we have selected a table, all subsequent database commands assume this table until another table is selected. Only one table may be selected at once, and it is the table last selected that is the target of the EDS commands.

To add a new record, we insert with `db_insert()`. To delete a record, we delete with `db_delete()`. To search for a record number, we seek with `db_find_key()`. To update the record data, we use `db_replace_data()`. We do not update the key data. The only way is to delete the record and create a new one with the same record data and a new key. We often seek in order to retrieve and read the data that is stored.

The `db_find_key()` function expects a key data parameter matching the key's data type. When the key is quickly found, a record number is returned. We then open the record with its record number. The `db_find_key()` is quick, but the record location using a record number is nearly instantaneous.

If a seek fails to find what we want, we may try a find. A find is more time consuming, so finds are not frequently done, but a find is thorough. With find, we open the database and, starting with the first record and continuing until the last, we search every field in record data looking for a match. There is no database command for this however; we have to write the code ourselves with a for loop using `db_record_data()` to search for the item we seek. In this case, we use `db_table_size()` to acquire the the number of records in the table being searched.

```
for i = 1 to db_table_size()
  if find(our_key, db_record_data(i)) then
    buffer = db_record_data(i)
    exit
  end if
end for
```

In ordinary operation, the create commands are used only to build the table the first time. Such commands may be in a separate utility program from the main program. Once databases and tables are created, it becomes a matter of opening and selecting. You do not open a table, but you select it. You may open more than one database, in which case, you must select the one you need as you need it. Once the database is selected, you then select a table.

Most operations on a working database will be `db_find_key()`. Occasionally you will delete and insert. The function `db_find_key()` returns a record number that may then be used for other critical operations. It is the primary concern of `db_find_key()` to return a record number.

To view the contents of a record, `db_record_data()` and `db_record_key()` require a record number. `db_delete()` requires a record number. `db_replace_data()` requires a record number. A record number is not needed for `db_insert` because the new number is chosen by the EDS engine.

Once a working database is in place, a typical work session will follow these steps:

```
-- We must find a record based on some key.
rec_num = db_find_key(our_key)    -- rec_num contains the record
                                   --number if
                                   -- found
seq = db_record_data(rec_num)    -- seq contains data from record
                                   --fields

-- We then inspect seq to see what we found in the record
-- We don't need to load key data because we have that already
-- If we make changes to the data we found, to move the changed
-- data back to the record we:

db_replace_data(rec_num, seq)    -- replace record with changed data
```

File Servers

The *file server* database is an economical type that is somewhat more sophisticated than the EDS database, but not as sophisticated as a client/server database. In a business system with many workstations, the database files that are shared by all workstations are stored on a single computer called a *server*. The files themselves are not locked, but portions of the file, records for instance, can be locked. This is important so that no one is changing the data in the same record as you at the same time as you. This keeps the data secure and accurate.

The file server database can serve ten and up to fifty users at a time, but usually, the system should be re-indexed at least once a day. It will cause more network traffic than its more sophisticated cousin, the client/server database.

Foxpro and the various Xbase versions like dBase, Clipper, Visual Objects, Harbour, and Flagship are file server types. The Access database is also this type of database.

Linking to DLL's

The term **DLL** means *dynamically linked library*, and such libraries are the stuff that Windows is made of. Throughout any Windows computer are many of them. They resemble the SO files found on Linux and Unix systems. DLL's contain code that is to be shared by more than one program, code that is reused. They are very similar to executable files, but they can never run alone, they are always running in support of an executable file specifically designed for them.

Being able to program with DLL files is an important skill to learn when writing Windows programs, but there are many other advantages to DLL's. There are DLL's that are widely available, some of them for free, that can be used by programmers to avoid writing difficult code. When one uses DLL's, one is borrowing the programming work of others.

One cannot use a DLL if the author does not allow it, but many DLL's are so designed that they offer many services and functions to those programmers who want them. But there is a skill to learn first.

Most DLL's for Windows were written in the programming languages C or C++. One does not need to know the C language, but it is a big help to know a little.

C and C++ have more data types than Euphoria, but Euphoria has tools that allow Euphoria programmers to merge with code written in C or C++. Euphoria programs can use C data and C pointers and transform them to Euphoria variables. C and C++ are so prevalent that most other programming languages have similar ways of conforming. It is this conforming that makes DLL's so practical.

Powerful code tools are available in .DLL form, and they are stored as procedures and functions available for linking. Euphoria offers linking commands that allow Euphoria programmers to call the procedures and functions in their Euphoria programs. It is easier if you are familiar with the C data type, especially those that are different from Euphoria types. There is a way to handle every kind, even pointers.

Yes, Euphoria allows pointers though one of the strengths of Euphoria is that you rarely need them. Pointers are variables (usually 32-bit long integers) that hold the address of data in memory. Going directly to memory allows swift algorithms, but it is a dangerous kind of programming that can cause very, very difficult bugs, that is why we avoid them. The C/C++ language makes extensive use of pointers, especially to pass parameters by reference. The .DLL that you use, will surely pass a pointer to Euphoria. Since the address is a number that may exceed the capacity of a Euphoria integer, Euphoria must store pointers in an atom variable. Euphoria then references the pointers by **peek()** or **poke()** statements.

The C language, like many languages has 4-byte, 32-bit integers, signed and unsigned, called long and ulong. The C language uses IEEE 64-bit floating point types called double (for double precision). C has special way of dealing with strings. C strings end with null to indicate the end of the string. The C language offers a 32-bit floating point variable called a float. In Windows, even the C integer is 32-bit, 4-bytes. Pointers, which are addresses are long integers, so pointers are 32-bit 4-byte values.

int_to_bytes() is a Euphoria function that can convert integers and atoms to 4-byte integers compatible with C.

bytes_to_int() is a Euphoria function that can convert 4-byte C integers, int, long, ulong to Euphoria atoms.

atom_to_float64() is a Euphoria function that can convert an atom to a C-type IEEE 64-bit (8-byte) floating point value.

float64_to_atom() is a Euphoria function that can convert a C-type 64-bit float (see above) to a Euphoria atom.

In C, strings are actually pointers to strings in memory. In Euphoria, **allocate_string()** is a function that will return the pointer to a string in memory, this pointer may be needed to communicate with C-based code. The function **allocate()** will return a pointer to any portion of memory that is reserved for such. Pointers must be freed after use; do this with the **free()** Euphoria function.

Rudimentary C tutorials are widely available, and learning a little of the C language, especially the data types, is advisable if you want to advance to advanced programming topics. The good news is that most of the parameters passed back and forth will be 4-byte segments.

If you know to what purpose the four bytes are reserved, you may extract them in Euphoria form with the various peeking commands (for peeking into memory). If you know the four bytes in memory are an unsigned long C variable, you can **peek4u()** into a Euphoria atom. If the four bytes is a signed C long, then peek with **peek4s()**. Or you can always peek a specified number of bytes with **peek()**.

Advantage Database Server

This product is not affiliated with the makers of Euphoria, and I have no affiliation with them, but I mention them here honorably because programmers should know about good products. This is a database server that performs like the very best though it might not be ideal for the very largest corporations. It is perfect for any small to medium size business. It is priced much lower than its famous competition, and no extra employees need be hired in order to administer it.

ADS, as it is affectionately called, is an outgrowth of the Xbase world that began with dBASE and evolved to Clipper, FoxPro, Recital, Harbour and others. These products are all dialects of the same Xbase language and structure. Older *.dbf databases can be easily updated to Advantage Database Server. Legacy Xbase skills can be employed and exploited in newer modern development.

This server is an SQL relational server, but it also offers the navigational features of a great ISAM engine which will be familiar to anyone who has developed applications and databases in dBASE, Clipper or FoxPro. The engine is available free as a .DLL called a Local Server (AdsLoc32.dll). This allows the development of file sharing databases that may be later upgraded to client/server databases with little change. With the local server .DLL, you can develop applications that have .dbf databases or that share databases with other Xbase products. AdsLoc32.dll is free to own and free to distribute. The Local Server is not adequate for mass client/server duty, but it allows programmers to become familiar with the company and its technology. It is sufficient for file sharing database systems. It shares with other Xbase products. If you elect to go fully to client/server technology, you must purchase the server from its makers. You can design your modest systems so that the later shift to client/server requires no change in code and no recompile. The strategy is outlined in the API.

MySQL

This is a popular and powerful database engine that can be used free of charge in some situations. It is economical and reliable regardless which situation you elect for yourself. By offering the product free, many programmers have used it, are familiar with it, and this kind of sponsorship has enhanced its reputation with corporations. It compares with ADS for quality and suitability, and it is widely employed.

Firebird Server

The Firebird Server (or Interbase 6.0) is the best choice for most situations today where the highest quality database performance and reliability is needed. Despite the fact that it is free of charge and easy to download from the web, it is a match for the very best products available, even Oracle and MS SQL Server. In systems with 200 workstations or less, there is no database engine faster. In any system, few database engines can match Firebird for power, features, completeness and reliability. So what is the catch? There doesn't seem to be one. It is a small download or installation, and it is easy to configure for immediate use. It configures in a variety of ways in order to be flexible, and it is even available as an embedded system. It is based on technology that is tested and proven in many real installations for more than twenty years.

It is probably best to use ODBC with Firebird (drivers are widely available), and ODBC Euphoria code is available for download at the RDS web site. The code must be altered for Firebird, and the DNS connection must be made like all ODBC applications, but this is a good way to learn ODBC technology which should be learned in any case.

The options available for use by Firebird is so complete and powerful, that it qualifies fully as an enterprise database server for all demanding situations. In advanced corporate software architecture, it can fully contain the business rules within the database engine with stored procedures, triggers (before and after), exceptions, checks and events. It has very sophisticated transaction abilities. There is none better, and it is free of charge. Why use any other?

In many large and medium-sized businesses today, you will find either Oracle or MS SQL Server as the database server in a client-server system with programs written in a variety of languages that connect to the server by ODBC. With Firebird replacing the others, you will still have cutting-edge technology that can be integrated with your Euphoria programs.

ADS offers uniqueness and value for the money. MySQL is a reliable product that is currently popular and inexpensive, but Firebird is the best choice for most situations. Whatever you choose, knowledge of relational databases and SQL are essential.

Program Planning

Structured procedural programming languages demand a top-down approach, but the planning is critical. Divide the whole job into smaller jobs, and the smaller jobs into still-smaller jobs. Subroutines, functions and procedures, should appear above the code where they are invoked.

An old technique that some programmers use for planning is the flow chart which is a schematic of the logic and flow using symbols. Other programmers use pseudo-code, which is a simplified form of programming language that is easy to translate into a real programming language. Some gifted programmers can merely plan a program in their heads. One method you might use is similar to pseudo code. Merely write your comments in advance line-by-line, then write the code under the comment that describes the programming step.

Start by writing comment lines describing the program line-by-line. This is a kind of pseudo code. The description should be carefully thought out and ordered. After describing every line, follow the comment with the code to do the action described. Here is a small example where a file retrieval process is described. What follows is a series of comments using '--' to describe the code to write.

```
-- while not the end of the file
-- input the next line into an object
-- test the object to see if it is an atom
-- if yes, then exit the while loop
-- otherwise, append the object to a growing sequence
-- when the end of file is reached, the entire file is now
-- contained in a sequence in memory, and the file boundary
-- was never breached.
```

Now, lets add the program lines to the comment lines.

```
object inp
sequence ln

ln = ""
-- while not the end of the file
while 1 do

    -- input the next line into an object
    inp = gets(fh1)

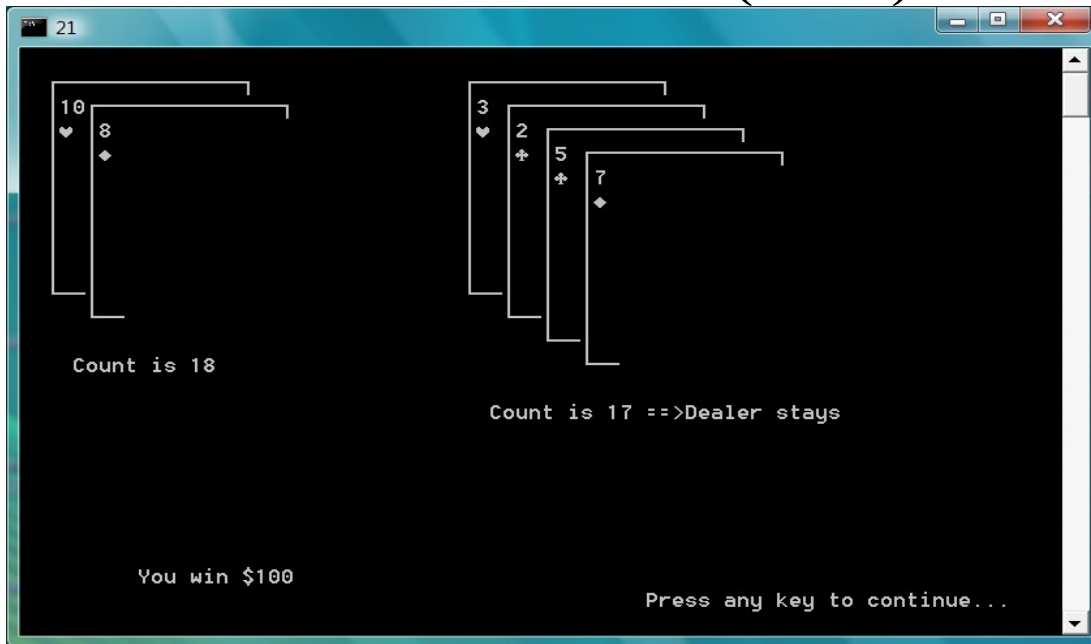
    -- test the object to see if it is an atom
    if atom(inp) -- end of file, if yes

        -- if yes, then exit the while loop
```

```
    exit --
end if

-- otherwise, append the object to a growing sequence
ln &= inp
end while
-- when the end of file is reached, the entire file is now
-- contained in a sequence in memory, and the file boundary
-- was never breached
```

BlackJack Game (21.ex)



BlackJack or Twenty-One is a card game for gambling that is popular at lavish casinos. Consider rules of the game as an algorithm. An algorithm is a step-by-step method or process. Writing a program or algorithm is like describing a card game.

To play BlackJack, you start with a standard deck of 52 playing cards and shuffle them. Among the players, there is a dealer, and one or more players. Then each player, including the dealer, gets a two-card hand initially. The dealer takes one card face down so that only the dealer knows what is beneath. The other players have both cards revealed.

Each hand is scored by scoring the cards; "Ten" and royalty get 10 points each, other cards are scored by their face value except Aces which may be scored as 11 or 1 (optionally whichever is more advantageous). To win you come as close as possible to 21 without going over. When 21 is exceeded, the player is "busted". The player with the highest score without going over 21 wins the "hand". If you receive twenty-one with only two cards, you shout "BlackJack" and win.

After the first two cards, you may ask for more by taking your turn. If you are less than twenty-one, you can ask for more cards in order to try for twenty-one, however, if you go over twenty-one, you lose.

To ask for another card, you say "hit me". To stay with what you currently have, you "stay".

This program has elements that seem graphic because they use graphic-like characters from the ASCII code, but they are not true computer graphics. Luckily, the four suits of cards have their symbols in the ASCII code.

The game program here, 21.ex, is a DOS program that allows a single challenger, you, play the dealer, the computer, at a true game of BlackJack.

Here is the main code that ties it all together.

```
integer game_over, round_over, busted

player_hand = repeat( 0,11 )      -- initialize with
{0,0,0,0,0,0,0,0,0,0,0}
dealer_hand = repeat( 0,11 )      -- initialize with
{0,0,0,0,0,0,0,0,0,0,0}
busted = FALSE
game_over = FALSE
cur_worth = 250
clear_screen()
cursor( NO_CURSOR )
init_deck()
next_card = 1
full_deck = shuffle( full_deck )
first_round = TRUE
round_over = FALSE
while 1 do                          -- 'do until' loops at least once
  game_over = get_bet()
  first_round = FALSE

  -- play til game over
  if not game_over then
    round_over = start_game()

    -- play this round til round over
    if not round_over then
      busted = player_play()
      if not busted then
        dealer_play()
      end if
    end if
    winner()
  end if
  if game_over then                  -- exit *while* loop only if game over.
    exit
  end if
end while

-- exiting the above *while* loop ends the program
cursor( UNDERLINE_CURSOR )
```

```
-----
-- Here's the plan...
-----
```

```
-- Create a brand new deck of cards
-- Shuffle the deck
-- start the game
-- take bets
-- the player plays
-- the dealer plays
-- find a winner
-- end or start over (a while loop runs the program however long it
--takes)
```

```
-----
-- Again, but with more details
-----
```

```
-- Create a brand new deck of cards
-- Declare a sequence var. called full_deck
-- The 52 cards come from 4 suits times 13 ranks
-- Therefore, we need two nested for..loops, one for suits,
--one for ranks
-- we need a counter to count 52, call it card

-- Shuffle the deck
-- a for..loop for 52 reps
-- at each iteration, generate a random number 1-52
-- swap the current element with element number *random*
-- after 52, all cards are shuffled and unpredictable

-- Start the game
-- Draw two cards each for two players
-- Draw the screen graphics that look like a playing surface
-- Use another routine for drawing each card
-- This is text-based programming, so there are no graphics,
-- only graphic-like characters in the ASCII character collection
-- These represent card boundaries and card suits, a happy
--coincidence

-- take bets
-- explain the betting and prompt a bet
-- if no choice is made, use a default bet amount

-- the player plays
-- Rules differ for players and dealers
-- prompt the player for Hit or Stay
-- use another routine, hit_or_stay()
-- draw a new card and count the score
-- you will need to move a pointer with move_pointer()
-- Print updated statistics as you move next to the dealer's play

-- if over 21 then stop player play

-- the dealer plays
```



```

-- turn over the hidden card
-- determine automatically if more cards are needed
-- get more cards as needed
-- draw cards to screen and keep score, display_card()
-- if over 21 then busted
-- Print updated statistics

-- find a winner
-- count both hands
-- win, lose, or draw?
-- score accordingly and redistribute bets

-- end or start over
-- call it yes_or_no()
-- As in the beginning, prompt the user
-- play or quit?

```

The program is divided into 17 subroutines plus the small amount of main code that weaves the threads together.

bsort() -- this is a simple sort.

card_convert() -- this is a scoring routine that is converts a card to a score for comparison; it is used in conjunction with the sort.

count_hand() -- this is a scoring routine that scores the hand for comparison; this is how the winner is identified.

dealer_play() -- this handles a single play; the rules for the dealer are slightly different from for other players.

display_card() -- software logic chooses the cards, but this routine draws them to the screen.

get_bet() -- this prompts the player to bet and records and calculates the bet.

hit_or_stay() -- this prompts the player to declare, hit or stay.

init_deck() -- a data structure that symbolizes a deck of cards is constructed.

move_pointer() -- the game proceeds one step at a time and the pointer keeps track; it also detects when the deck is exhausted (to shuffle the used cards).

noise_maker() -- various beeps and sound signals

Pause() -- pauses the program and invites the user to press any key to proceed.

player_play() -- this handles single play for players.

shuffle() -- this is the opposite of a sort. Using a random number generator cards are swapped in pairs at random. After 52 swaps, they are thoroughly shuffled.

start_game() -- this is called at the start of each game. It prepares the cards and the table.

val() -- this transforms string numerals to numbers.

winner() -- this determines if there is a winner or a draw and it identifies the winner.

yes_no() -- this prompts the user to decide and enter yes or no.

Employees.ex

Variables take data from the external universe (input) and process it for useful information (output). The data is stored in the computer's memory and variable names help keep track of the data.

Windows Programs

The Windows operating system from Microsoft became popular in the 1990's and is now the most popular operating system in the world. The operating system of a computer is a collection of programs and utilities that manage everything on the computer. The operating system handles the disk actions that save data, and it sets the rules that programs must abide by. If your program breaks the rules of the operating system, it will be stopped.

Windows is a Graphical User Interface (GUI) type of operating system that requires huge amounts of memory and which relies a great deal on the point-and-click technology of a desktop mouse. Windows is a multitasking operating system that can run many programs simultaneously, each in its own window. Windows is flexible, so you can still run older MSDOS programs, and you can run console programs that are not GUI programs.

The most important thing about Windows from Microsoft is that it is the most widely used operating system, so programs for Windows are the most profitable.

There are a number of ways to write Windows programs, but the most popular way in Euphoria is the easy way, using the Win32lib.ew library with the Judith Evans Integrated Development Environment. This is what we discuss in this section. This allows you to virtually draw the program you want on screen with drawing tools. The Win32lib.ew library handles much of the complexity for you, but you must still adjust your point of view to programs that are event driven. Up to now, most of your programs were procedural proceeding from the top down. Event driven programs may suddenly branch in any direction by the unexpected click of a mouse on the screen.

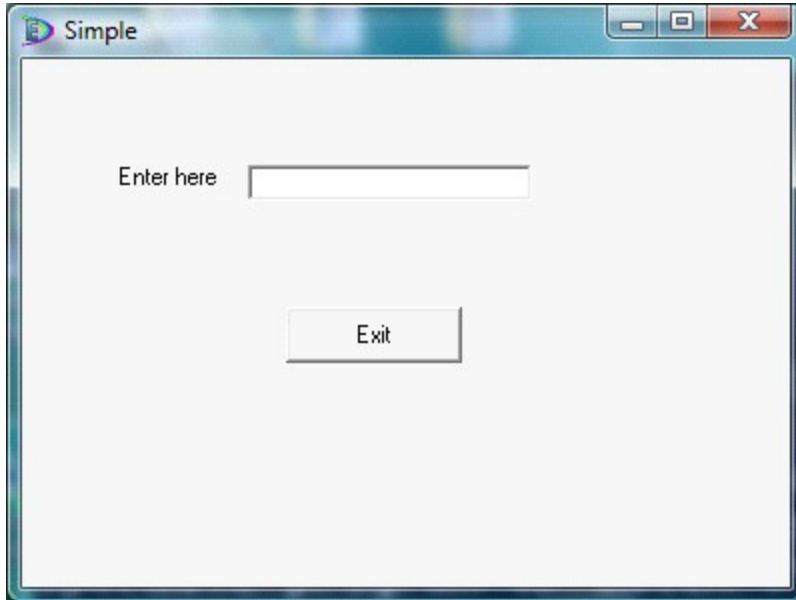
Stop and think about what you see on screen when you are in Microsoft Windows. You see windows pop up in many sizes and many locations. Windows may have buttons, list boxes, text-entry boxes, scroll bars, icons, and other pictures. Each of these things is a control, and one control often belongs to another control. A window is a special control, often called a form, that may be a host to other controls (each a child control). A button on a Window is a control that belongs to another control, the window.

Your Windows programs will need to describe each window (size, location, features, title) and each control found on each window. Windows sends and receives messages constantly. If the user clicks a button, Windows received the click event message related to a button (size, location, features, caption) that is found on a certain window (name, size, location, features).

Even though Windows allows many operations at once, there is only one focus at any

given time. The focus is on the last button clicked or the last selection by the user. If there are several windows on the screen, then the window with focus will move to the front unless there is a modal window open. A modal window is a window, like a message window, that dominates the focus until a decision is made and the modal window is clicked. A warning message box is this kind of window.

The cursor is a pointer that is linked to the mouse, and it allows aiming and targeting with great accuracy. It shows where the mouse is pointing.



The example program, *simple.exw*, has enough code to launch a Windows program with this appearance. It does not do much, as example programs frequently do not, but it shows the basic requirements of the most primitive Windows program written with the Win32lib.ew library that many Euphoria programmers prefer to use.

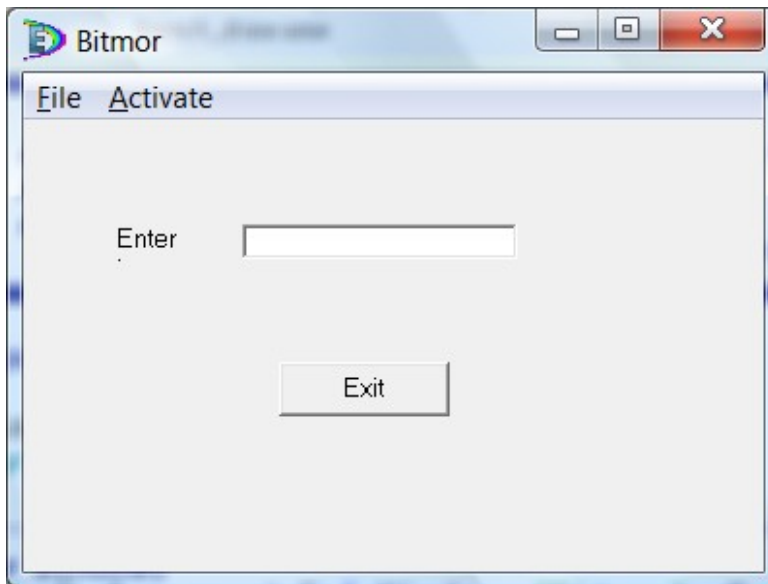
```
include Win32lib.ew
without warning

-----
-- Window Window1
constant Window1 = createEx( Window, "Simple", 0, Default, Default, 400, 300, 0)
constant EditText3 = createEx( EditText, "", Window1, 112, 52, 144, 20, 0, 0 )
constant PushButton4 = createEx( PushButton, "Exit", Window1, 132, 124, 88, 28, 0, 0 )
constant LText2 = createEx( LText, "Enter here", Window1, 48, 52, 60, 20, 0, 0 )
-----

-----
procedure PushButton4_onClick (integer self, integer event, sequence params)--params is ()
  closeWindow(Window1)
end procedure
setHandler( PushButton4, w32HClick, routine_id("PushButton4_onClick"))

WinMain( Window1,Normal )
```

Most of the code is devoted to creating the visible controls you see when the program runs. The window has two controls, an edit box and a button. The edit box has editing capabilities already supplied by the Windows operating system and all those wonderful programmers at Microsoft. The button has added action. It will end the program when pushed. This added function is added by procedure **PushButton4_onClick()** and the **setHandler()** statement. You could have programmed the button to launch bells and whistles if you wanted to, but since the caption says "Exit", it is a good idea to deliver "exit-ation", if that is a word. When the button is pushed, a Windows event occurs. Nothing else will happen unless you write code that responds to the event, an event handler, if you will. So **setHandler** sets up an event handler. A procedure is needed to complete the set. **PushButton4_onClick()** is connected to the handler by being specified in the third parameter of **setHandler()**. The procedure takes advantage of the fact that when you close with **closeWindow()** the program's main window, you end the program, those are the rules.



The next program, *bitmor.exw*, is a slight expansion on the previous one. It adds some drop-down menu controls.

```
include Win32lib.ew
without warning

-----
-- Window Window1
constant Window1 = createEx( Window, "Bitmor", 0, Default, Default, 400, 300, 0, 0 )
constant mnuFile = createEx( Menu, "&File", Window1, 0, 1, 0, 0, 0, 0 )
constant mnuFileNew = createEx( MenuItem, "&New", mnuFile, 0, 2, 0, 0, 0, 0 )
constant mnuFileOpen = createEx( MenuItem, "&Open", mnuFile, 0, 3, 0, 0, 0, 0 )
constant mnuFileSave = createEx( MenuItem, "&Save", mnuFile, 0, 4, 0, 0, 0, 0 )
constant mnuFileSep1 = createEx( MenuItem, "-", mnuFile, 0, 5, 0, 0, 0, 0 )
constant mnuFileExit = createEx( MenuItem, "E&xit", mnuFile, 0, 6, 0, 0, 0, 0 )
constant mnuActivate = createEx( Menu, "&Activate", Window1, 0, 7, 0, 0, 0, 0 )
constant mnuActivateNow = createEx( MenuItem, "&Now", mnuActivate, 0, 8, 0, 0, 0, 0 )
```

```

constant mnuActivateLater = createEx( MenuItem, "&Later", mnuActivate, 0, 9, 0, 0, 0, 0 )
constant EditText3 = createEx( EditText, "", Window1, 112, 52, 144, 20, 0, 0 )
constant PushButton4 = createEx( PushButton, "Exit", Window1, 132, 124, 88, 28, 0, 0 )
constant LText2 = createEx( LText, "Enter here", Window1, 48, 52, 60, 20, 0, 0 )
-----
procedure PushButton4_onClick (integer self, integer event, sequence params)--params is ()
  closeWindow(Window1)
end procedure
setHandler( PushButton4, w32HClick, routine_id("PushButton4_onClick"))
-----
procedure mnuFileNew_onClick (integer self, integer event, sequence params)--params is ()
integer ans
  ans = message_box("You clicked New on File menu", "New clicked", MB_OK)
end procedure
setHandler( mnuFileNew, w32HClick, routine_id("mnuFileNew_onClick"))

WinMain( Window1,Normal )

```

Most of this code was written by another program, the Win32lib Interactive Development Environment. Using this IDE with a mouse, you can draw the controls; you can drag and drop. Then the IDE writes the code needed to create the program that works with those controls. You may then add functions and other code manually as you like. This will probably include more windows and controls and functions that respond to mouse-activated events with those controls. Some programmers refer to such an IDE as a "screen painter".

```

constant Window1 = createEx( Window, "Bitmor", 0, Default, Default, 400, 300, 0, 0 )

```

The **createEx()** function is widely used to create a control like a window; it returns the control's id number which is stored as a constant since it will not change once created (it may be willfully destroyed, but that is another story). The name of the Window (control) is the name of the constant, **Window1**. So a control resembles a variable in some ways. Window1 does not belong to any other controls, but it does belong to the Bitmor program, so Bitmor is a parameter in its proper place. Other parameters set the size and location (a default in this case).

```

constant mnuFile = createEx( Menu, "&File", Window1, 0, 1, 0, 0, 0, 0 )

```

mnuFile is another control created with **createEx()** but it is not a window; it is a child control that belongs to Window1; it is a child of Window1. That is shown by having Window1 as the third parameter. We have now created a menu control which will need some children, menu items, which are themselves controls.

```

constant mnuFileNew = createEx( MenuItem, "&New", mnuFile, 0, 2, 0, 0, 0, 0 )

```

mnuFileNew is another control created with **createEx()** but it is not a window or a menu; it is a menu item. It is a child of mnuFile which is a child of Window1. There are control types as there are data types. The first parameter of createEx() calls for the control type, Window, Menu or MenuItem for instance.

When the program launches, it is given the focus. It will keep the focus until the operator clicks some other control on the desktop. When the window Window1 has the focus, you can click on its child controls. You can enter text in the text box or click the exit button or select from the menu. Whatever you select by clicking takes the focus. If you later click on another program, Window1 and all of its children will lose the focus. The cursor is no longer blinking invitingly from the text box; there is no longer an outline around the button that once had focus. Can you see how important it is for Windows to keep track of the focus when multiple programs are running simultaneously?

Just as it was your job to name variables you created; it is your job to name controls; you may choose any name you like within reason, but be smart and sensible. Window1 will do for the main window, but windowMain is another. The name windowInput is probably better. You should expect to have more than one window in a program.

There is a mnuFile that is a holder for menu items like mnuFileNew, mnuFileOpen, mnuFileSave, mnuFileExit. The names were chosen to identify these as menu item controls that belong to the mnuFile control. It is the property of menus that there must be a menu before there are menu items. The menu mnuFile is merely a holder for the menu items, but it is a control in its own right, and it must be created before its children are created. Think ahead of the structure and function of the controls so you can name them well. Changing the name later means changing too much code.

Notice that we are using the so-called Win32lib method which will not work unless you have included the Win32lib.ew library at the top of your program. This is the largest library yet available for Euphoria, and it is constantly maintained by various expert programmers. There is a fabulous IDE available for free that makes Windows programming like painting on a surface. If you have ever used Microsoft's Visual Basic you know how fun that can be.

NOTE: If you would like to see an example of raw Windows development without Win32lib, look at the Windows examples in the DEMO subdirectory of the Euphoria installation.

Are there any advantages to raw, bare-bones programming in Windows?. Yes. There is less code bloat, and the programs load much faster. Those C/C++ programmers who are used to programming for Windows may prefer the raw method, but it looks very daunting to beginners because so many lines are required for simple operations even though there is no extra bloat behind the scenes. You will need a guide to programming for Windows if you hope to program this way. Such a guide is a good idea anyway. A guide will outline the system DLL's and what services they contain for your use. You will need to master programming with DLL's to do that. A guide will offer real examples to demonstrate difficult topics.

Remember, Windows is also a collection of data entry code *that is already written for you for you to use*. Try to remember that when you think how complicated Windows programming is. You are re-using much valuable code that belongs to the system itself.

That is complicated code that you do not have to write yourself.

You are free to add your own style to your Windows programs, but that is widely frowned upon in professional circles. Microsoft has published guidelines and has demonstrated the guidelines in its own software that governs proportions and appearances. Programs that use odd or oversized buttons with florescent colors look unprofessional. The same is true of other extreme departures from normal appearance. Very large fonts and oddball structure are like an automobile manufacturer putting the brake pedal in some new and innovative location. It may be fun and exciting, but it probably spells trouble for someone.

Depending on the type of program, however, you may improve the appearance with skin packages from third party suppliers. These make for sleek or futuristic looking controls and formats. If you are different just to be different, you will leave the impression with purchasers and fellow programmers that you are playing with kiddie code. Of course that is just advice for those who hope to enter the profession. Play around as you like, of course; you are bound to learn from experimentation.

Look at the window to `Bitmor.exw`. Do you see "Enter here"? That is a caption that belongs to a label control, called an `LText` control. What does a `LText` control do? It offers a place to write captions, and its caption and location can change later in the program. Most of the functional details of a Windows program are controls with properties to be set or to be changed as needed.

To find out what properties each control has, you can check the `Win32lib.ew` documentation, but much of it is intuitive. A `LText` control has little value other than a place to show text, called a caption property. There is font and font size to consider. There is its location relative to the window's dimensions. There is color (usually the same color as the background). There is the size, length, width and height of the control. These details are properties to controls. Most properties you must set yourself, usually when you create the control, but you can change properties later. Some properties come with a default if you choose to let Windows choose for you.

Programming Multitasking

Multitasking is the ability to perform two or more actions at once. The multitasking actions are actually sharing time with the processor invisibly. It is a kind of round-robin that allows one action to advance a step or two then it stops briefly so another action can advance a step or two, and so on. Because the the switching is very fast, the results are fluid like truly simultaneous operation should be. Operating systems employ preemptive multitasking that dictates system-wide from moment to moment how resources are shared. But programs usually use cooperative multitasking because it is better suited to planned algorithms. Multitasking does not speed up any of the processes. It has to slow each action a little in order to share the resources with other tasks or processes.

So why multitask? Multitasking is not usually recommended, but it solves some problems very neatly. In computer games, the appearance of simultaneous action is a necessity. When database servers multitask (they usually do) they allow sharing, but short tasks still finish early while long tasks finish later as is expected. This serves to keep the sharing invisible. Those requesting brief service need not wait for the long tasks to finish first. So, to reiterate, multitasking does not speed up the results, but it makes the sharing process itself invisible and normal.

There are times when the computer program waits for a response from the user. During this waiting, other tasks could be completing behind the scenes. This calls for **time sharing** multitasking where one task sits almost at idle waiting for a response. The program loses none of its response time, essentially, but it uses its resources more efficiently. When the user is roused to respond, you may redistribute the resources to match the request.

In a game with several players, actors or sprites all moving independently and unexpectedly, it is important to establish the appearance of natural simultaneous action without jitters or periodic stalls. This calls for **real time** multitasking.

Multitasking in Euphoria is quite easy to do with more than one strategy available. Sometimes the simplest strategy is perfectly effective. The tutorial on multitasking by RDS found in the DEMO directory of Euphoria installation is brief and adequate when the time comes that you think you need multitasking.

Translating Euphoria to C

RDS supplies an Euphoria-to-C translator for Windows and another one for DOS in the original installation. The translator will emit various files with C code that is compatible with popular compilers. Several free compilers of high quality are available that will compile the translated files into binary executable files. These files perform essentially like programs written in C. This is significant since C programs have a reputation for efficiency, speed and power, but it is much harder to program in C. The speed increase is insignificant in many programs, but it may increase the speed by 2 to 5 times in other situations. Experimentation will tell. Programs written in C often have pointer-related bugs, but Euphoria translations are probably free of such things. These binary executables are the last word in hiding the original source code when that becomes a very serious consideration; the Euphoria binder will shroud the code, but there is room for doubt in minds of some developers.

Even without a C compilation, Euphoria can bind the source code into a *.exe file that is superficially indistinguishable from any other *.exe file. Furthermore, bound (non-translated) executables execute with impressive speed, more than thirty times faster than Python and Perl programs and considerably faster than Java programs. It is merely academic that a bound executable is not a true binary executable. Translation to C is rarely compelling simply on the basis of speed of execution, but it remains an impressive and valuable option available to all Euphoria programmers. It lends great weight to the argument, "Why bother programming in C/C++?".

You must acquire and install one of the suggested free C packages in order to exploit the translator. I suggest either the Boland 5.5 package or the Open Source Watcom package. These two products are unsurpassed in quality yet they are available free of charge. On Linux and Unix, the C packages come with operating system. This requirement is no small effort, but it is necessary for anyone who wants to translate to C on a regular basis.

Invoking the translator is like invoking the interpreter, but the translator emits C code and finishes with the message to launch a batch file by typing "emake". Shortly (if everything is properly set up) there is a binary executable to execute; extraneous files are deleted at the end of the process. The windows executables are rather large, but they can be compacted with a professional compactor. See the RDS literature.

Modern computers are so fast that an extra boost in speed is not a high priority, but in the minds of many programmers, programs written in C offer the highest performance of any high-level language. It is gratifying to know that if the program you are writing needs the utmost in speed, you have that option in the end.

Web Programming with CGI

The basic skill needed for any kind of world wide web programming is the language HTML. HTML is not a general purpose programming language like Euphoria or C, but it is a page layout language for browsers. The commands and syntax of HTML merely instruct a browser to print a page with a certain format with font selection, heading and centering or left/right justification. It allows you to make links to other pages or other sites. When you arrive at a web site, your browser loads a file with HTML code and creates the page on your computer as instructed by HTML. From there, a two-way communication takes place. If you click on a link to a new page, a new file of HTML code is loaded. Fortunately, HTML is easy to learn and millions have learned it. If you do not yet know HTML, you must learn it. There are many free tutorials on line.

For years, the main technology for advancing simple web programming to powerful and professional web programming was Common Gateway Interface (CGI). This is a fancy name for a simple concept. CGI is a method to attach a web page to an executable program so they can communicate back and forth in a secure way. This method allows visitors to your site to run one of your programs without putting yourself at risk of malicious tampering.

To the user, it simply seems that the web page is powerful and interactive. CGI usually begins with an input text box that allows the user/visitor to input information, like a search box for instance. Then there is a button to press to launch the process. The web page with CGI capability will launch a program and send it input. The program will parse the information into its basic components and respond by sending HTML code with its output so that the output is also a web page.

HTML becomes the GUI form, and your specially prepared programs add programming power to your web page. They interact and cooperate so that the website acts like a powerful program. Often CGI programs query databases and return the results in a web page. Since CGI was introduced, other technologies like ASP and PHP were introduced, but they are simply variations on the same thing.

The first CGI programs, were written in Perl, and that has become a tradition, but CGI works with virtually any programming language. Many famous and huge web sites operated for years relying essentially on CGI and Perl, so it is a proven technology. Euphoria can do what Perl does, but it executes over thirty times faster, so Euphoria is perfect for web programming. Even binary executables like compiled C programs can do CGI with obvious advantages.

NOTE: Lately PHP is a very popular form of CGI, but Euphoria code executes 60 times faster than PHP even without a translation to C. Extra speed means extra bandwidth which means extra traffic capacity without any annoying delays.

So a CGI program can do what you expect programs to do, but it contains parsing code for parsing the inputs string sent by the HTML page and its output must contain enough HTML to create a web page in response. The gateway is the standard input controlled by the operating system. Most service providers use either Linux or USB Unix for the operating system, and there are free versions of Euphoria for Linux and PSB Unix. The code you write should require little if no change to swap operating systems.

It is fortunate that every skill you now learn as you learn Euphoria is a skill that will contribute to your ability to become a professional web programmer thanks to CGI. For years, advanced web programming *inferred* CGI though there are now more technologies at your disposal, but CGI is not obsolete and no matter what advanced method you use, you will still need to understand HTML.

The RDS website uses CGI with Euphoria and Robert Craig has offered much of the code for your use. You will find an excellent parsing routine that you may use, and other contributors have offered their CGI libraries for your use.

Some website providers offer a database engine like MySQL or ADS, so you can use your database technology for this. However, the EDS database is quite adequate for most of the database requirements for most websites.

Consider a complicated site like an online bookseller. You need to let the visitor search a database of book titles. You need to let the visitor register, add his or her name to a customer database. You must take orders and charge fees. A web page in HTML is not powerful enough. You need CGI for those extra features and functions.

The Software Business

Probably most professional programmers work for corporations who write programs in teams. The language preference by corporate programming teams has tended to go in waves of popularity. Currently C#, Java, VB.NET, C/C++ and Delphi are very common. Recital may be used in some cases. COBOL programs are still in operation that need constant maintenance by COBOL programmers. Not so long ago Visual Basic 6.0 was very popular, but not so much any more. Most such programmers should be proficient with relational database technology. Corporations prefer five years of recent experience. It is difficult to get back into the programming business if you have been out of the main for a year or two because the technology changes so fast.

There are small consulting firms that build systems of programs for small businesses that may use a variety of technologies including FoxPro, Access, Visual Basic, Harbour or any of the languages previously mentioned. They may be hired to customize a doctor's office or small retail operation. They usually also offer web programming these days. VBA allows programmers who know Visual Basic to program for customizing the programs that come with Microsoft Office, Excel, Access, Word, etc. Linux may grow among small business, and products like OpenOffice may replace MS Office in such cases.

There will always be opportunity for freelance programmers, but being in business for yourself is always a risky business. Programmers with excellent resumes, excellent education and excellent experience, may charge more than \$100.00/hour, but they do not take as long to finish the job and they have a proven record.

As long as you have the skill and the time to develop a valuable program, you may sell your program on an entrepreneurial basis. You may create a web site devoted to selling your program. Any professional or avocational experience in your life may combine with your programming skill to create a unique product of value with a potential for sales. As a self-taught programmer, other programmers will look upon you as an amateur, but as an entrepreneur that does not matter.

Many experienced professionals know more than one programming language. Euphoria has attracted many programmers who once programmed in another language. Many C/C++ programmers have switched gladly to Euphoria. Visual Basic programmers are amazed that Euphoria is even easier to learn with greater capacity and performance plus portability. The portability of Euphoria is not common and should not be underestimated.

The shareware business model may appeal to you. According to this model, you distribute your program to others to use on a conditional basis so they can try before they buy. You may alter or cripple your circulated program, and supply a perfect copy to

those who "register" and buy, or you may not. You may merely trust that many people will pay you what they owe you. Be sure to have a registration database with the names and addresses of everyone who registers your product. It may come as surprise to you that many programmers have found this method of selling software satisfactory. If you go the shareware route, you are saving money on promotion, so you might consider lowering the price of purchase. Carefully consider the asking price in any case. You will be expected to supply support after the sale. But the world wide web and commerce have made retail so easy that shareware is not as attractive to sellers as it once was. You could have a web store instead.

There are only a few professional software companies known to me who are currently employing Euphoria programmers. If you search with Monster.com or something similar for a job for a Euphoria programmer, you probably will not see many offers, or any offers at all. I did not choose Euphoria because it is popular. I chose it because I thought it was the greatest value in the application development business at this time, and because that it will probably remain so for some time to come.

Any profession or avocation is enhanced if the worker has extra skills like computer programming. If you are a systems analyst, you will do your job better if you are fluent in some kind of "scripting" language that lets you automate your work. If you are a lawyer or chemist, you could benefit from the customized programming you can do for yourself. If you do any kind of office or clerical business, your ability to program is a big asset. If you want to start your own business, writing your own customized software is the single most valuable skill of all. A well designed practical program is like a reliable employee who works for free and lasts forever.