Mathematics

Getting Started

Calculations are what computers do best. Many thousands of complex calculations can be performed In the blink of an eye.

Creative Basic has a rich set of mathematical operations and functions which can be applied to any mathematical or statistical task.

Statements and Expressions

Programs will involve many **Statements** which will look something like:

```
variable = expression
```

The **'variable'** will have been given an appropriate name (using a maximum of 30 characters) and will be defined as one of the data types discussed earlier.

Variables are **'assigned' a** value with the **'='** sign, (For example: count = 5)

When the program is run, the **'expression'** is evaluated, and the result is stored in the receiving variable.

The expression can be any combination of numbers, variables, constants, array elements and function calls.

A simple example is :

```
Def value as int

value = 5 * cos(x)
```

Arithmetic Operators

An expression will usually have several **Arithmetic Operators** separating the values in the expression.

Creative Basic has the following arithmetic operators:

Symbol	*Name*	*Example*	*Result*
+	**Addition**	1.5 + 5.6	7.1
-	**Subtraction**	4.6 - 3.2	1.4
*****	**Multiplication**	1.5 * 2.2	3.3
/	**Division**	4.5 / 2.5	1.8
^	**Exponentiation**	3 ^ 2	9
%	**Modulus (Remainder)**	6 % 4	2

Operator Precedence

In a complicated expression, processing is broadly from left to right, but arithmetic operators have a generally accepted precedence .

The **Precedence** of the operators are (from highest to lowest):

^

/, *

+, -

=

Any terms involving exponentiation (raising to a power), are performed first - followed by multiply and divide terms, and finally any additions and subtractions.

If there is any possibility of this default sequence giving a result which is not what you intend, use parentheses **( )** to clarify what you want to achieve.

Parentheses **( )** take precedence over all other operators.

For example, you might have a complicated expression as in the following example:

```
openconsole
cls

Def value, j as int
Def x as float

x = 0.5
j = 5

x = 4 ^ sqrt(j) + 3.056 / 1.14 * cos(x)

print x

do:until inkey$<>""
closeconsole
end
```

The expression for ‘x’ leaves plenty of scope for not getting the answer you expect. (You can try it out using the console test program above).

As written, the printed result for ‘x’ will be 24.55.

But what if you really meant (note the new parentheses) :

```
x = 4 ^ sqrt(j) + 3.056 / (1.14 * cos(x))
```

Now you will get the printed result for ‘x’ as 25.25.

So always use **Parentheses** in complicated expressions, to specify exactly what you want to achieve.

While parentheses will make your intention clear, there is another approach.

You can use what are known as **Auxiliary** equations.

We could write:

```
aux1 = 4 ^ sqrt(j) + 3.056
aux2 = 1.14 * cos(x)
x = aux1 / aux2
```

In other words, **Auxiliary equations** can be used to split a complex expression into simpler expressions.

Incidentally, this also makes it easier to check the accuracy of intermediate results by placing a **'Stop'** statement just after the equations, and using the Debug display to see the values of each variable .. a very useful facility.

Maths Functions

Creative provides the following maths functions.

All trigonometric functions take their parameters in **radians** and return their results in radians.

Each of these functions takes **one** parameter, a numeric expression or variable, and returns a numeric result.

Function	*Return Value*
SIN (n)	Sine of n
COS (n)	Cosine of n
TAN (n)	Tangent of n
ASIN (n)	ArcSine of n
ACOS (n)	ArcCosine of n
ATAN (n)	ArcTangent of n
SINH (n)	Hyperbolic sine of n
COSH (n)	Hyperbolic cosine of n
TANH (n)	Hyperbolic tangent of n
LOG (n)	Natural logarithm of n
LOG10 (n)	Base 10 logarithm of n
SQRT (n)	Square root of n
EXP (n)	Exponential of n
ABS (n)	Absolute value of n (removes sign)
CEIL (n)	The smallest integer greater than or equal to n
FLOOR (n)	The largest integer that is less than or equal to n
RND (n)	A random number between 0 and n
INT (n)	The whole number portion of n
NOT (n)	Returns the ones compliment of n
SGN (n)	Returns **-1** if negative, **1** if positive and **0** if n is 0

This is a great set of tools - but if you need more - you can just roll your own.

It's easy to do, and is a useful feature of Creative Basic.

We'll do an example - and at the same time, demonstrate how useful the Console Window skeleton program can be.

In case you've forgotten it from the earlier 'Using Windows' section, here it is again.

```
openconsole


do:until inkey$ <> ""
closeconsole
end
```

(I usually save this simple console skeleton somewhere, so that I can copy and paste it whenever it's required).

Then, if ever you have a little test program to check out - this bit of code is just pasted into the Creative editor - and away you go.

Making your own Functions

As a simple example, we will create a useful function to return a Random Number between any two given values.

The built-in random number function RND(n) takes only one argument and gives a result between 0 and the specified value 'n'.

So our function is just extending this to take two arguments. Here is the test console program and the new function **Rn**(Lower,Upper).

```
openconsole
cls
autodefine "off"

declare Rn(Low as int, High as int)

for i = 1 to 10
   print Rn(10,50)
next i

do:until inkey$<>""
closeconsole
end

sub Rn(Low, High)
def ret:int
' generates an integer random number between Low and High ..
ret = int((high - low + 1) * rnd(1) + low)
return ret
```

Just copy and paste into the Creative editor and run it.

You should see 10 random numbers between 10 and 50.

Notice the statement at the beginning of the test program.

```
declare Rn(Low as int, High as int)
```

This declares the new function **Rn( )**, as a user function, so that it can then be used anywhere in the program, as if it was a Creative built-in function. Very useful.

You can do this for any function you wish to create.

Precision

Creative defaults to a precision of two decimal places for output.

In fact, all Creative maths operations are computed internally using Double precision accuracy (16 significant figures), but only two decimal figures are displayed by default.

It's easy to adjust the number of decimal places by using the **Set Precision** command.

For example:

```
setprecision 5
```

will give 5 decimal places for output values..

A variable of type **Float** can only represent up to **8** significant figures, and a **Double** precision variable up to **15** significant figures.

So there's no point in using the Setprecision statement to go beyond those values.

The **Rn( )** function is also interesting because it uses the in-built Creative function RND(1).

The user guide says that this function generates random numbers from 0 to 'n'. So if you use for example RND(10) , you will get numbers between 0.0 and 9.99999.

Notice that the RND( ) function returns decimal values.

So RND(1) will give random numbers between 0.0 and **0.99999**.

If you need integer values, use INT(RND(n) + 0.5) for numbers between 0 and 'n'.

If you omit the rounding value 0.5, the random numbers will never return the highest value 'n'. Alternatively use INT(RND(n+1)) to make sure the value 'n' will be returned.

While we're on a roll .. you might like this home-grown function **RFlip()**:

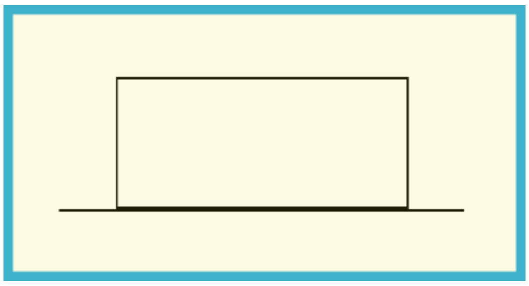
```
openconsole
cls

declare RFlip()

for i = 1 to 10
   print Rn(10,50)
next i

do:until inkey$<>""
closeconsole
end

sub RFlip
' returns a random +1 or -1
def ret: int
if rnd(1) < 0.5
   ret = +1
else
   ret = -1
endif
return ret
```

What on earth is that any good for? .. you might ask.

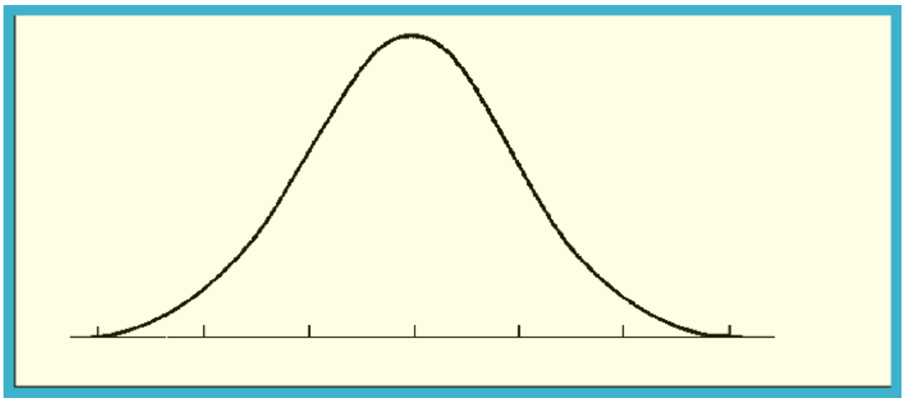
Well, suppose you want to move an object around the screen with random direction and speed, you could use **RFlip( )** as in this snippet ..

```
' set the initial direction of movement and speed (pixels)..
dx = RFlip() * Rn(1,3)
dy = RFlip() * Rn(1,5)
```

The random numbers we've just looked at are drawn from a 'uniform rectangular distribution' in which every number has an equal probability of occurring.



For some applications, you will need another type of random number distribution - the 'Normal distribution' for which numbers are distributed according to the normal probability curve.



So just to complete our exploration of random numbers, here is a test program for a **Rnorm( )** function. The user function declaration is at the top, and the **Rnorm( )** function is at the end.

```
openconsole
cls

declare Rnorm(lower as int,upper as int,mean as int,sdev as int)

def dist[26]:int
def i,j,n:int

lower = 0: upper = 100
mean = (upper - lower) / 2.0
sdev = (upper - lower) / 6.0

for i = 1 to 20000
   n = Rnorm(lower,upper,mean,sdev)/4 + 1
   if n <= 25
      dist[n] = dist[n] + 1
   endif
next i

locate 2,30
color 15,9
print "Normal Distribution"

for i = 1 to 25
   n = dist[i]/100
   for j = 1 to n
      locate   25 - j, i * 3 + 1
      print "X"
   next j
next i

do:until inkey$ <> ""
closeconsole
end

sub Rnorm(lower as int,upper as int,mean as int,sdev as int)
' returns random numbers with normal distribution and given mean
' and standard deviation
def x1,x2,r,nval : double

do
   x1 = 2 * rnd(1) - 1.0
   x2 = 2 * rnd(1) - 1.0
   r = x1 * x1 + x2 * x2
until r < 1.0

nval = x2 * sqrt(-2 * log(r) / r)
nval = mean + nval * sdev

return nval
```

Copy and paste the test program into the Creative editor and run it.

You should see the results plotted as a 'normal curve distribution'.

Trigonometric Functions

The trigonometric functions like sine, tangent, etc. expect the number to be expressed in radians, rather than degrees.

There are (2 * pi) radians in a full circle of 360 degrees.

To work in circular units, you will often need the value for **'pi'.**

You can calculate 'pi' in two ways:

```
pi = 4 * atan(1)
```

or

```
pi =  acos(-1)
```

Then to convert from degree to radians, you can use:

```
d2r = pi / 180
```

To convert from radians to degrees, it's just the inverse:

```
r2d = 180 / pi
```

Logical Operators

Many occasions arise when it is necessary to compare two values.

These are known as **Logical** (or **Boolean**) comparisons.

The main thing to note about such comparisons, is that the result can only be one of two possible values - **TRUE** or **FALSE**.

Some languages have pre-defined constants for these values.
Normally **TRUE** equates to **1**, and **FALSE** equates to **0**.

Creative doesn't have pre-defined values for TRUE and FALSE. In fact you rarely need to type them yourself.

When you carry out logical tests using the logical operators, the result will be a TRUE or FALSE value that other program instructions can use directly.

Creative has a good set of logical comparison operators:

Logical Operator	***Meaning***
>	**Greater than**
<	**Less than**
<>	**Not equal to**
=	**Equal to**
>=	**Greater than or equal to**
<=	**Less than or equal to**

You can use these operators to compare two variables or expressions.

Here is a simple example using the greater than comparison ..

```
openconsole
cls

def a:float
def b:int

a = 2.01
b = 2

if (a > b) then print "a is greater than b"

do:until inkey$<>""
closeconsole
end
```

If you are comparing two numeric values, it doesn’t matter what type they are.

Obviously, trying to compare a **numeric** value with a **string** value such as “Jim”, makes no sense.

You can compare two characters or string values on an ASCII code basis - that is, “a” is a smaller value than “z”.

So “aardvark” is considered a smaller value than “zephyr” in any logical comparisons.

However, watch out for upper and lower case values, because lower case characters have higher ASCII values. (ASCII “A” = 65, but ASCII “a” = 97).

This means that if you are comparing “apples” with “Pears”, the “Pears” value is the smaller value, because of the upper case “P”.

Bitwise Operators

There are three other operators which are available for use:

Bitwise /Logical Operator	*Meaning*
&	**Logical AND**
|	**Logical OR**
||	**Exclusive OR (XOR)**

The logical **AND** operator (**&**) is used when you require two expressions to be TRUE in order for a compound test to succeed. For example:

```
openconsole
cls

def a,b,c :int
a = 2 : b = 7 : c = 5

if (a < c) & (b = 7) then print "success"

do:until inkey$ <> ""
closeconsole
end
```

In this example, variable ‘a’ is less than ‘c’ AND ‘b’ does equal 7 - so both expressions are TRUE, and the word ‘success’ is printed.

The logical **OR** operator (**|**)is frequently used to combine several style parameters for Windows and controls.

For example, when you create a Window, there are a number of ‘flags’ you can specify to set how the window is arranged.

Here’s an example:

```
WINDOW win1,0,0,350,350,@MINBOX|@MAXBOX|@SIZE,0,"Example",main
```

The @SIZE parameter specifies the window is sizeable.
@MINBOX will display the window minimising icon at the top right of the window, and @MAXBOX displays the maximising icon.

Each of these flags is an integer constant, the values being carefully chosen so that they can be combined to bring each feature into operation.

So @SIZE = 0x40000 (in Hexadecimal), @MINBOX = 0x20000, and @MAXBOX = 0x10000 (You can look them up in a list of Window’s constants).

If you combine these using the logical OR (|) operator, as in the example, you will get an overall flag value of 0x70000.

You could just use this ‘magic number’ instead of writing `@MINBOX|@MAXBOX|@SIZE.`

It would give the same window arrangement, but I prefer to use each flag separately because it’s easier to understand what’s happening.

Now I have to say, the logical operator **XOR** (||) is not often used. I can't recall ever having used it - so I've had to look around for any useful applications.

You can express the outcome of an exclusive or (**XOR**) logical operation like this:

- if a is true and b is true then (a **XOR** b) is false
- if a is false and b is false then (a **XOR** b) is false
- if a is false and b is true then (a **XOR** b) is true
- if a is true and b is false then (a **XOR** b) is true

Incidentally, you can set a variable to zero by **XOR**-ing it with itself.

So, (a **XOR** a) = 0 That's very interesting, but it seems easier to just say a = 0!

One possible use is monitoring the status of a modem, or some other equipment, and storing the status of all lines in the bits of a variable 'status' as '0' and '1''.

Then, in a timing loop, you can read the 'currentstatus', and XOR it with the initial status to see if anything has changed.

If any bit has changed, the result of the XOR test will be non-zero. If nothing has changed the result will be zero, since (according to the above truth table), if both sets of bits are the same, all bits in the result will be '0'.

The bitwise **XOR** operation is probably more useful than the logical XOR operation.

When operating at the bit level, the XOR truth table is:

```
1 Xor 1 = 0
1 Xor 0 = 1
0 Xor 1 = 1
0 Xor 0 = 0
```

Use can be made of this in a simple encryption method.

The method takes all 8 bits of each text character and XOR's them with the 8 bits from a chosen **keyword** character.

For example, using a 'key' character bit pattern of ' 01010101'

(11111111 **XOR** 01010101) produces the encoded bit pattern 10101010

Once you have the encrypted result, if you **XOR** again using the same key, you recreate the original value.

10101010 (Old Result) **XOR** 01010101 (Key) gives the original result 11111111.

We need a test program to try this out ..

So here's the test program:

```
openconsole
cls
autodefine "off"

def key$,code$,text$:string

' declare user function encrypt ..
declare encrypt(x$ as string,k$ as string)

' choose an encryption key ..
key$ = "shazam"
text$ = "Mary had a little lamb"

print "Original text:   ",text$

' encrypt the text ..
code$ = encrypt(text$,key$)

print:print "Coded Text:"
print code$:print

' now convert the code$ back again ..
print "Decoded text :   ",encrypt(code$,key$)

do:until inkey$ <> ""
closeconsole
end

sub encrypt(x$ as string,k$ as string)
def textlen,keylen,kchar: int
def a$,b$,ret:string

' encrypts (or decrypts) x$ by XOR'ing with k$ ..
keylen = len(k$)
textlen = len(x$)

for i = 1 to textlen
    a$ = mid$(x$,i,1)                          :' take each x$ character in turn
    kchar = (i - 1) % keylen + 1               :' calculate the next key character
    b$ = mid$(key$,kchar,1)                    :' get the next character from the key
' convert the characters to ASCII, XOR them, and convert to a character again
    if a$ = b$
        ret = ret + a$
    else
        ret = ret + chr$(asc(a$) || asc(b$))
    endif
    next i

return ret
```

I don't think it would slow down a cryptologist for long, but it's interesting nevertheless.