Strings and Structures

The Basic language has always been good at handling strings (groups of characters).

Creative Basic continues that tradition with an excellent set of string commands.

Here they are:

Function	Result
APPEND$	APPEND$("One plus ","two") returns "One plus two"
ASC	ASC("A") returns 65
CHR$	CHR$(65) returns "A"
DATE$	DATE$("ddd',' MMM dd yyyy") returns "Wed, Aug 31 1994"
HEX$	HEX$(255) returns "FF"
INSTR	Position = INSTR({start, } string1, string2) Returns the position of the string 'string2' in 'string1'. or returns 0 if string2 is not in string1. Optional 'start' variable specifies a starting point in string1 to begin searching.
LCASE$	LCASE$("MainFrame") returns "mainframe"
LEFT$	LEFT$ ("Alfred", 3) returns "Alf" - the first three characters from the left.
LEN	LEN("Zebra") returns 5
LTRIM$	LTRIM$(" 4 spaces") returns "4 spaces" - spaces and tabs removed.
MID$	Result$ = MID$ (string, position {,count}) Extracts 'count' number of characters starting at 'position' from a string. If 'count' is omitted, all of the characters from 'position' to the end of the string are returned. MID$("A song", 3,3) returns "son"
REPLACE$	s = "Good DOGs go to heaven" REPLACE$ s, 6, 3, "dog" results in s = "Good dogs go to heaven"
RIGHT$	RIGHT$("Welcome",4) returns "come"
RTRIM$	RTRIM$("Trailing 4 ") returns "Trailing 4"
SPACE$	SPACE$(5) returns " " - (5 spaces)
STR$	STR$(123) returns "123" STR$ only works to FLOAT precision - 7 significant figures.
STRING$	STRING$(5,"T") returns "TTTTT"
TIME$	TIME$ returns "hh:mm:ss"
UCASE$	UCASE$("Cleopatra") returns "CLEOPATRA"
VAL	VAL("123.67") returns 123.67

Arrays

An **Array** is a collection of data of the same type.

When you **DEF**(ine) an array, memory will be reserved to hold the data, depending on the data type, and the size of the array you define.

Suppose you anticipate having to process 5 integers such as 21**,** 5, 36, 50, 2 .

You will need to define an array such as **DEF A[5]: INT**
The number in square brackets specifies the required number of locations **including zero**.

The 5 allocated memory locations are indexed as : A[0] , A[1], A[2], A[3], and A[4].

This is because array indexing is **zero-based.** That is, the first byte of the zero'th location identifies the start of the array. in memory.

It will allocate **5** sequential storage locations (elements) to the array variable named **A,** each location reserving 4 bytes of memory (since the **Integer** data type requires 4 bytes of storage).

Each of our 5 integers can now be stored, one in each location, so that:

```
A[0] =21
A[1] = 5
A[2] = 36
A[3] = 50
A[4] = 2
```

How do we load the numbers ? Well, we could write separate assignments for each location as above. That works, but it becomes a chore when a large array is involved.

Alternatively, we can write more concisely:

```
A[0] = 21, 5, 36, 50, 2
```

Also, if later we were to write:

```
A[3] = 66, 105
```

We would overwrite the last two locations, so the array content would then be:

```
A[0] =21
A[1] = 5
A[2] = 36
A[3] = 66
A[4] = 105
```

An instruction **PRINT A[2]** , would display the value **36**

An instruction **A[3] = A[3] + 1** , would increment array location A[3] by one to 67.

Arrays are at their most useful in looping operations.

Here's an example using a loop to print the values stored in the array:

```
openconsole
cls

def a[5]:int
' load the array ..
A[0] = 21, 5, 36, 50, 2

for i = 0 to 4
   print "A[", i, "] = ", a[i]
next i

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

Notice that the FOR loop runs from **0 to 4**, to access the zero'th element of the array **A[0]**, and run through to the **A[4]** element which holds the **fifth** number.

Now you're probably thinking that you sort of understand what an array is, but are feeling confused about the zero-based indexing.

You'd probably prefer to place your first value in A[1], and the fifth value in A[5].

So would I, so here's how to do it.

Define one more location for the array than before - **DEF A[6]:INT**

Now we can ignore the zero'th location A[0]. It's still there, so we are wasting a small amount of space.

Now we can load the values into the array using:

```
A[1] = 21, 5, 36, 50, 2
```

The five numbers are now stored in array locations A[1] to A[5] as follows:

```
A[1] =21
A[2] = 5
A[3] = 36
A[4] = 50
A[5] = 2
```

I think that's much easier to follow.

One other thing to mention, is that we don't have to 'waste' the zero'th location - it can be used to hold the number of array elements in use.

Suppose we allocate a larger array having **10** locations, by defining an array size A[**11**]. (Remember the extra one to allow for the zero'th location)

Maybe we've only processed **5** items of information, so the array is nowhere near full.

If, as we process each item, we update a count in **A[0]**, we can use this to tell us exactly how many array locations have been used. So in our example A[0] = 5

Now we can re-do our program to print the contents of the array using this count.

```
openconsole
cls

def A[11]:int
' load the array with only 5 values ..
A[1] = 21, 5, 36, 50, 2
' keep count ..
A[0] = 5

for i = 1 to A[0]
   print "A[", i, "] = ", a[i]
next i

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

The five values in array A[ ] are now located in locations A[1] to A[5] , and the awkward A[0] location is pressed into service as a count, telling us how many array locations we've used.

A much more satisfactory arrangement I think.

The above example described an array of integers, but an array can be defined to store any of the available variable types including strings.

We might for instance have an array of strings:

```
def A[6]:string

A[1] = "Alan"
A[2] = "Janet"
 .. etc ..
```

Multi-Dimensional Arrays

Now we move on into even murkier waters .. arrays with more than one dimension.

Creative Basic provides for arrays having up to three dimensions.

So far, we've only looked at a single column array - one dimension.

A 2-dimensional array is like a spreadsheet or table - having so many rows and so many columns - for example **def A[10,20]:int**

3 dimensions are harder to visualise - maybe something like a book of 2-dimensional tables with so many pages.

For example, we could get to a particular data item in a 3-dimensional array on Row 5, Column 3, Page 4 by using a statement like:

```
def A[10,5,10]:int
def value:int
   ..
value = A[5,3,4]
```

Internally arrays are stored in column major order.

That means for a 2-dimensional array, the first column loads from element 0,0 and the second column from element 0,1

Here's an example which stores Name and Age for two people - name in the first column and age in the second.

```
openconsole
cls
' two dimensional array

def a[3,3]:string

a[1,1] = "Fred","Alyson"
a[1,2] = str$(23),str$(25)

for i = 1 to 2
   print a[i,1],a[i,2]
next i

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

It works, but it doesn't look or feel very efficient. I don't like having to convert numbers to strings for example, just to satisfy the one type of data requirement.

Multi-dimensional arrays are fine if loaded as part of a maths calculation routine, but are a strain on the brain when initialising values in the correct order.

There are better ways.

There are in fact no such things as multi-dimensional arrays in computer memory.
All data is laid out flat from the start of an array, with the memory locations corresponding to the 1, 2 or 3 dimensions calculated by the computer.

Therefore, it is easier conceptually and in practice, to set up several one dimensional arrays to hold data 'records' which can then be of mixed types.

Associated Arrays

A simple example might be a dataset to hold Names, Address, Postcode, and Telephone number. If we insist on keeping the data items separate for the sake of the example, that would require 4 dimensions in the previous method, which was not possible.

(Obviously all the strings could be combined into one or two long records).

Here's an example of how this works:

```
openconsole
cls
' two dimensional array
def size:int
def tabs:string

size = 3
' dynamic array sizing
def name[size],address[size],postcode[size],phone[size]:string
' initialise arrays
name[1] = "Fred","Alyson"
address[1] = "2 Smith Square, Bexley"
address[2] = "43 Berkley Road, Tipton"
postcode[1] = "CH3 5TG","TY2 H23"
phone[1] = "0121 478574","0155 687235"

tabs = string$(6,str$(9))

for i = 1 to 2
   print
name[i],tabs,address[i],tabs,postcode[i],tabs,phone[i]
next i

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

This example also shows that arrays can be **dynamically sized**. The required size is determined at run time, and the arrays are then **DEF**(ined)

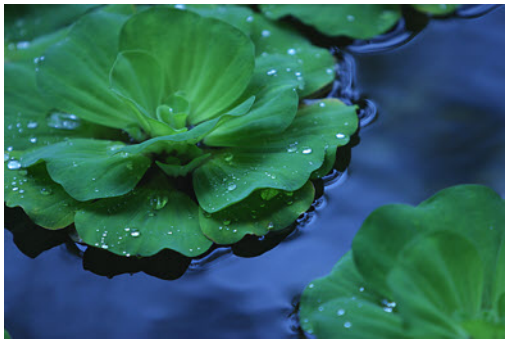
Data is initialised starting at the first array element (rather than the zero'th) as before.

The array Name[ ] could be searched for a required name. Once the index was obtained, it would apply to all the other associated arrays, and the record could be displayed.

This seems an improvement over the multi-dimensioned array method, and can be used easily with mixed data types.

However, there's an even better way.

User Data Types



Creative Basic provides a really flexible way to deal with data records - the **User-defined data type**.

This advanced data type can take the place of multi-dimensioned arrays, and becomes really useful when graphics and games programs are involved.

This example will use the previous example of Name, Address, Postcode, and Telephone number data.

```
openconsole
cls

def tabs:string

type NAPP
   def name :string
   def address :string
   def postcode :string
   def phone :string
endtype

def rec[5] : NAPP

rec[1].name = "Fred"
rec[1].address = "2 Smith Square, Bexley"
rec[1].postcode = "CH3 5TG"
rec[1].phone = "0121 478574"

rec[2].name = "Alyson"
rec[2].address = "43 Berkley Road, Tipton"
rec[2].postcode = "TY2 H23"
rec[2].phone = "0155 687235"

tabs = string$(5,str$(9))

for i = 1 to 2
' note the closing comma to ensure printing continues on the same line
   print rec[i].name,tabs,rec[i].address,tabs,
   print rec[i].postcode,tabs,rec[i].phone
next i

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

We have defined an array **rec[5],** where each array entry holds a complete record for each person.

Each data item is accessed using the dot **'.'** operator

The elements of each record would normally be loaded dynamically as the user enters them, but could be loaded from a disk file.

This is a much neater method, and becomes even more powerful when you consider that any of the User Type variables can themselves be nested UDT's.