Updating a Record in a Function

Introduction

Creating the Data Structure

DarkBASIC Pro allows us to create record structures using the TYPE statement. For example, we could create a structure to hold the coordinates of a point in 2D space with the lines

```
TYPE PointType
    x AS INTEGER
    y AS INTEGER
ENDTYPE
```

after which we can create a variable of this type

```
pnt AS PointType
```

and access the fields within that variable with statements such as

```
pnt.x = 12
pnt.y = 20
PRINT pnt.x
```

Operations on the Data Structure

However, if we're taking our programming seriously, statements such as

```
pnt.x = 12
```

are a worry, since direct access to the fields within the record allow us to place any value we wish within it - even invalid ones. For example, if the coordinates stored in a *PointType* variable represent a position on the screen, then neither *x* nor *y* should contain a value of less than zero, but the code does not contain any logic for preventing this.

A hardened programmer would be happier doing the assignment via a function designed to check that the value being assigned is a valid one. For example, let's say that the value assigned to *x* must be in the range 0 to 1000. Then using the function-based approach, we would assign a value to *x* using a function such as

```
FUNCTION SetX(p AS PointType, newx)
    IF newx < 0 OR newx > 1000
        EXITFUNCTION
    ENDIF
    p.x = newx
ENDFUNCTION
```

The function takes two parameters:

p - the *PointType* structure to be changed.

newx - the new value to be assigned to the *x* field within *p*.

The function checks that the value to be assigned to *x* is within the permitted range (0 to 1000) and if it's not, then the function terminates. Only if we have a valid value is it assigned to the *x* field.

Activity Sup038

Create a new project *(recordupdate01.dbpro*) containing the code given.

Add a second function, *SetY()* to the code which allows a value between 0 and 1000 to be assigned to the *y* field.

For slightly more subtle reasons (that we need not go into here), functions are often created which return the value held in each of the fields. For example, we could return the value held in the *x* field using the following function

```
FUNCTION GetX(p AS PointType)
    result = p.x
ENDFUNCTION result
```

Activity Sup039

Add the above code to your project. Also add a function (*GetY()*) to return the *y* value of a *PointType* variable.

The purpose of all this only begins to make sense when you realise that on a large software project many programmers will be involved. One programmer might write the code for the *PointType* structure and its associated functions while a different programmer makes use of variables of this type to solve some other problem. The first programmer does not have to explain to the second how the *PointType* structure was coded - only provide details of the names, parameters and purpose of each function associated with the structure. The point of all this is to hide from anyone using a *PointType* variable the details of how the data type and its operations are constructed. To this end, the coding for *PointType* should be created in a separate file and imported into any program that requires variables of this type using a #INCLUDE statement.

The contents of this file are shown in LISTING-Sup012.

LISTING-Sup012

The PointType Data Structure

```
TYPE PointType
   x AS INTEGER
   y AS INTEGER
ENDTYPE

REM *** PointType operations ***
FUNCTION SetX(p AS PointType, newx)
   IF newx < 0 OR newx > 1000
      EXITFUNCTION
   ENDIF
   p.x = newx
ENDFUNCTION

FUNCTION SetY(p AS PointType, newy)
   IF newy < 0 OR newy > 1000
      EXITFUNCTION
   ENDIF
   p.y = newy
ENDFUNCTION
```

continued on next page

LISTING-Sup012
(continued)

The PointType Data Structure

```
FUNCTION GetX(p AS PointType)
   result = p.x
ENDFUNCTION result

FUNCTION GetY(p AS PointType)
   result = p.y
ENDFUNCTION result
```

Activity Sup040

Type in and save the code given in LISTING-Sup012 (*PointType.dbpro*).

An Application using *PointType*

With the data structure and its operations coded, we can now test the code by creating an application which makes use of a *PointType* variable, assigns values to it, and displays those values on the screen (see LISTING-Sup013).

LISTING-Sup013

Using a PointType Variable

```
#INCLUDE "PointType.dba"
REM *** Main section ****
REM *** Declare variable ***
pnt AS PointType
REM *** Assign values ***
SetX(pnt,12)
SetY(pnt,20)
REM *** Display values ***
PRINT GetX(pnt),"  ",GetY(pnt)
WAIT KEY
END
```

Activity Sup041

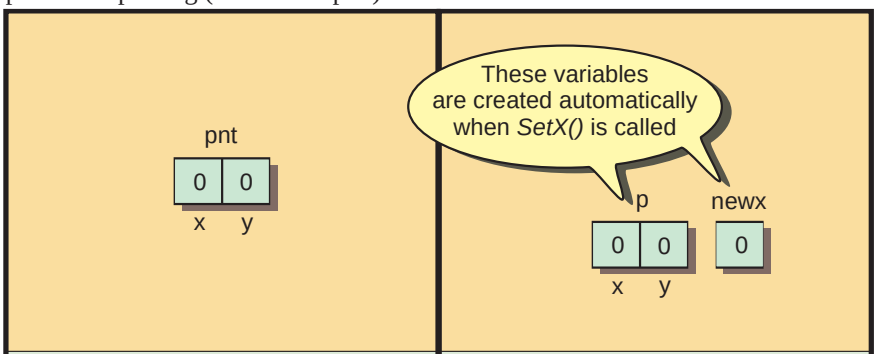
Create a new project (*TestPoint.dbpro*) containing the code given in LISTING-Sup013. Copy the file *PointType.dba* into this project's folder. Run the program and check the results produced.

What went Wrong?

Of course, the problem is that you can't change the contents of a variable passed as a parameter to a DarkBASIC Pro function. That's because the function uses a copy of the original data rather than the origin itself. This is known as **pass by value** parameter passing (see FIG-Sup14).

FIG-Sup14

How Parameter Passing Works

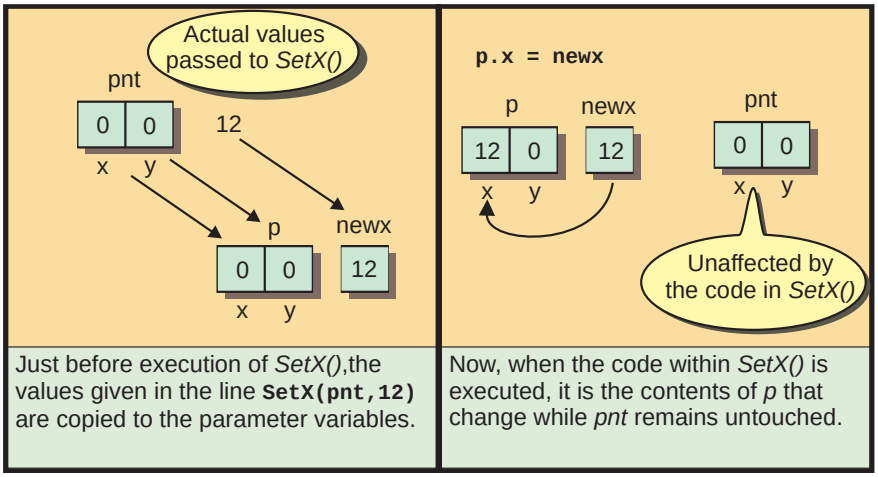


The line **`pnt AS PointType`** creates the variable *pnt* with both its fields, *x* and *y*, containing zeros.

Calling the function *SetX()* with the line **`SetX(pnt,12)`** causes the two variables named in *SetX()*'s parameter list to be created.

FIG-Sup14
(continued)

How Parameter Passing Works



And, because *pnt* is unaffected by executing the code in *SetX()* and *SetY()*, the values displayed on the screen by the line **`PRINT GetX(pnt), "  ",GetY(pnt)`** are zeros.

Using Memory Blocks

DarkBASIC Pro allows us to reserve any number of bytes of memory using the MAKE MEMBLOCK statement. For example, the line

```
MAKE MEMBLOCK 1,20
```

MEMBLOCK statements are covered in Chapter 47 of Volume 2.

would reserve a block of 20 bytes and assign it an ID of 1.

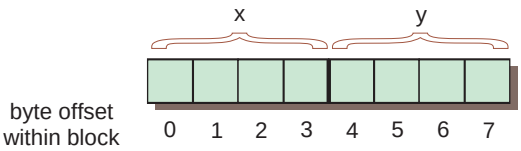
Memory blocks are assigned IDs in much the same way as an image, sprite, or 3D object is given an ID. No two memory blocks can have the same ID.

By making use of these memory blocks we can solve our problem of updating variables within a function.

First we have to calculate how many bytes a memory block would require if it is to store the details of a *PointType* variable. Since the structure contains two integer values (*x* and *y*) we'll need 8 bytes within the block (4 for each integer) - see FIG-Sup15.

FIG-Sup15

An 8-byte Memory Block for Storing Two Integer Values



To store a value in a specific part of a reserved memory block, we use the WRITE MEMBLOCK statement giving the block ID, the offset position within the block where storage is to begin, and the value to be stored. So, whereas in the traditional record structure we wrote

```
p.x = 12
```

Notice that MEMBLOCK statements use the term WORD rather than INTEGER.

with the memory block, the equivalent statement would be

```
WRITE MEMBLOCK WORD 1,0,12  // block ID 1, offset 0, value 12
```

Activity Sup042

What would be the equivalent of `p.y = 20` when using a memory block with an ID of 1?

To retrieve a value already stored within a memory block we specify the type of value being retrieved, byte, word, float, etc., the block ID and the offset from the start of the block. Hence to retrieve the value of *y* from our block, we would use the statement

```
result = MEMBLOCK WORD(1,4)
```

Redesigning *PointType*

Near the start of this article we stated that any large project will likely involve several programmers, each reponsible for different sections of code. When one of the programmers in a team needs to make changes to his code it should be done in such a way that little or no changes are required in the code produced by the other programmers in the team.

In our scenario we might think of the code for *PointType* being created by one programmer and *TestPoint* by another, so our aim is to rewrite *PointType* using memory block with as little change as possible to the code in the main section.

Handling the Memory Block ID

Every time we want to create a *PointType* variable, we will need to create a new memory block, each with its own unique ID value. To ensure a new ID for each block we'll create a global ID variable which will be incremented each time a new block is created:

```
GLOBAL memblockID = 0
```

The next trick is to redefine *PointType* as an integer. This will be used to contain the ID of the block containing the actual data. Since the #CONSTANT statement performs textual replacement, we can achieve our goal with the line

```
#CONSTANT PointType INTEGER
```

Whenever a programmer requires a *PointType* variable a new memory block must be created and the *memblockID* variable incremented. This is done with the code

```
FUNCTION CreatePointType()
    INC memblockID
    MAKE MEMBLOCK memblockID,8
ENDFUNCTION memblockID
```

Notice that the function returns the ID used when the new block was created.

The other operations in *PointType* would be recoded as follows:

```
FUNCTION SetX(p AS PointType, newx)
    IF newx < 0 OR newx > 1000
        EXITFUNCTION
    ENDIF
    WRITE MEMBLOCK WORD p,0,newx
ENDFUNCTION
```

```
FUNCTION SetY(p AS PointType, newy)
    IF newy < 0 OR newy > 1000
        EXITFUNCTION
    ENDIF
    WRITE MEMBLOCK WORD p,4,newy
ENDFUNCTION

FUNCTION GetX(p AS PointType)
    result = MEMBLOCK WORD(p,0)
ENDFUNCTION result

FUNCTION GetY(p AS PointType)
    result = MEMBLOCK WORD(p,4)
ENDFUNCTION result
```

Notice that although the code for each routine has been changed, the purpose, name and parameters of each function remain unchanged. This will minimise the changes required by any programmer who has made use of our earlier code.

Activity Sup043

Rewrite your *PointType.dbpro* code to reflect the changes made. Do not include the global *memblockID* variable in your code.

Updating the Application Program

As we stated earlier, the aim is to minimise disruption of any other code which happens to make use of *PointType* variables when updating the *PointType* data structure. Here we'll see just what changes are required to the earlier test program.

The first requirement is to add the global variable declaration:

```
GLOBAL memblockID = 0
```

and the second is to add a call to *CreatePointType()*:

```
pnt = CreatePointType()
```

But, other than this, no further changes are needed, so our final code is that shown in LISTING-Sup014.

LISTING-Sup014

The Updated *TestPointType* Application

```
#INCLUDE "PointType.dba"

REM *** Declare variable holding last memory block ID
GLOBAL memblockID  = 0

REM *** Main section ****
REM *** Declare variable ***
pnt AS PointType

REM *** Create variable ***
pnt = CreatePointType()

REM *** Assign values ***
SetX(pnt,12)
SetY(pnt,20)
REM *** Display values ***
PRINT GetX(pnt),"  ",GetY(pnt)
WAIT KEY
END
```

Activity Sup044

Modify your own *TestPoint.dbpro* to match the code given in LISTING-Sup014.

Run the program and check if the results are correct.

Multiple Variables

There are no restrictions on the number of variables that can be created. For example, we could create and manipulate two *PointType* variables with the lines:

```
#INCLUDE "PointType.dba"
GLOBAL memblockID = 0
p1 AS PointType
p2 AS PointType
p1 = CreatePointType()
p2 = CreatePointType()
SetX(p1,12)
SetY(p1,20)
SetX(p2,50)
SetY(p2,90)
PRINT GetX(p1),"  ",GetY(p1)
PRINT GetX(p2),"  ",GetY(p2)
WAIT KEY
END
```

Activity Sup045

Type in and test the program given above (*TestPoint2.dbpro*).

We could even create an array of *PointType* variables with the lines

```
DIM points(10) AS PointType
FOR c = 1 TO 10
    points(c) = CreatePointType()
NEXT c
```

Activity Sup046

Write a program (*TestPoint3.dbpro*) which sets up an array of 10 *PointType* values and assigns the *x* and *y* fields of each variable a random value between 0 and 1000. The program should then display the contents of each element in the array.

Solutions

Activity Sup038

The program code should be:

```
TYPE PointType
     x AS INTEGER
     y AS INTEGER
ENDTYPE
FUNCTION SetX(p AS PointType, newx)
     IF newx < 0 OR newx > 1000
        EXITFUNCTION
     ENDIF
     p.x = newx
ENDFUNCTION
FUNCTION SetY(p AS PointType, newy)
     IF newy < 0 OR newy > 1000
        EXITFUNCTION
     ENDIF
     p.y = newy
ENDFUNCTION
```

Activity Sup039

The code for *GetY()* is:

```
  FUNCTION GetX(p AS PointType)
     result = p.x
  ENDFUNCTION result
```

Activity Sup040

No solution required.

Activity Sup041

The program displays the values 0 0.

Activity Sup042

The equivalent to **`p.y = 20`** would be

```
WRITE MEMBLOCK 1,4,20
```

Activity Sup043

The code for PointType.dbpro is:

```
#CONSTANT PointType INTEGER

REM *** PointType operations ***
FUNCTION CreatePointType()
  INC memblockID
  MAKE MEMBLOCK memblockID,8
ENDFUNCTION memblockID

FUNCTION SetX(p AS PointType, newx)
   IF newx < 0 OR newx > 1000
      EXITFUNCTION
   ENDIF
   WRITE MEMBLOCK WORD p,0,newx
ENDFUNCTION

FUNCTION SetY(p AS PointType, newy)
   IF newy < 0 OR newy > 1000
      EXITFUNCTION
   ENDIF
   WRITE MEMBLOCK WORD p,4,newy
ENDFUNCTION

FUNCTION GetX(p AS PointType)
   result = MEMBLOCK WORD(p,0)
ENDFUNCTION result

FUNCTION GetY(p AS PointType)
   result = MEMBLOCK WORD(p,4)
ENDFUNCTION result
```

Activity Sup044

No solution required.

Activity Sup045

No solution required.

Activity Sup046

The program code is:

```
#INCLUDE "PointType2.dba"
GLOBAL memblockID  = 0
REM *** Create array ***
DIM points(10) AS PointType
FOR c = 1 TO 10
   points(c) = CreatePointType()
NEXT c
REM *** Seed random number generator ***
RANDOMIZE TIMER()
REM *** Assign values to each point ***
FOR c = 1 TO 10
   SetX(points(c),RND(1000))
   SetY(points(c),RND(1000))
NEXT c
REM *** Display values ***
PRINT "Values stored are:"
FOR c = 1 TO 10
   PRINT c, "  ", GetX(points(c)),"  ",
↳GetY(points(c))
NEXT c
REM *** End program ***
WAIT KEY
END
```