

Nested Records

Introduction

In the last section we looked at how we could create record structures that could be updated within a function. In this section we will see how complex nested record structures can be created and manipulated. We'll start with a triangle structure and then create a much more complex structure which we'll use to create a slider visual component.

The *TriangleType* Data Structure

DarkBASIC Pro allows us to draw lines or boxes but not much else, so let's assume we'd like to be able to draw triangles as well. Although the most efficient approach might be to create a new statement using C++ and a DLL, this does require us to know an entirely different programming language as well as finding out how to call DirectX functions. An easier approach would simply be to create a record structure to contain all the information we'll need about our triangle and a corresponding set of operations to manipulate and make use of that data.

The Data

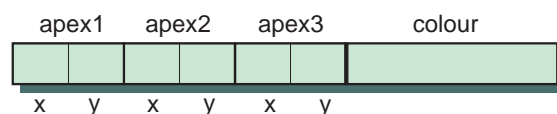
A triangle consists of three points (called **apexes**). Each apex is identified by its position in space - in this case 2D space. If we are to display our triangle, we might also want to record the colour to be used when drawing the triangle, so the record structure we require can be coded as

```
TYPE TriangleType
  apex1 AS PointType
  apex2 AS PointType
  apex3 AS PointType
  colour AS DWORD
ENDTYPE
```

Notice that each apex has been defined as a *PointType* structure, giving us a record within a record. Conceptually our record has the structure shown in FIG-sup016.

FIG-sup016

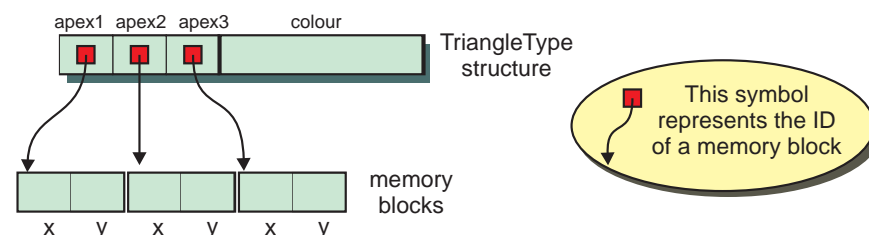
The TriangleType Structure - Concept



However, the reality of our structure is a little different; the fields *apex1*, *apex2* and *apex3* do not actually contain the coordinates of the points on the triangle, but rather the IDs of memory blocks. It is those memory blocks that contain the coordinates (see FIG-sup017).

FIG-sup017

The TriangleType Structure - Reality



Since we require *TriangleType* structures to be updated within functions, this itself must be stored in a memory block!

The Operations

CreateTriangle()

We'll need a *CreateTriangleType()* function to allocate the memory block for a *TriangleType* structure. But, since fields within this structure reference *PointType* memory blocks, these blocks will also need to be allocated. The code for the routine is:

```
FUNCTION CreateTriangleType()  
  REM *** Determine ID for block ***  
  INC memblockID  
  REM *** Allocate block (16 bytes) ***  
  MAKE MEMBLOCK memblockID,16  
  REM *** Remember ID allocated to triangle ***  
  result = memblockID  
  REM *** Create blocks for each Pointtype      ***  
  REM *** and store their IDs within the      ***  
  REM *** first 3 fields of the triangle block ***  
  WRITE MEMBLOCK WORD result,0,CreatePointType()  
  WRITE MEMBLOCK WORD result,4,CreatePointType()  
  WRITE MEMBLOCK WORD result,8,CreatePointType()  
  WRITE MEMBLOCK WORD result,12,RGB(255,255,255)  
  REM *** Return ID allocated to triangle block ***  
ENDFUNCTION result
```

Notice that the function "uses up" four memory block IDs; one for the *TriangleType* block and one for each of the *PointType* blocks. Since we cannot allocate names to areas within a memory block - as we can in a record structure - we have to assign values by specifying the position within the block at which a value is to be stored.

SetApex1()

The usual function for assigning values to the various fields are also required. The *SetApex1()* function sets the values of the first apex of the triangle and has the following code:

```
FUNCTION SetApex1(tr AS TriangleType,x,y)  
  REM *** Get the ID of the first PointType memory block ***  
  temp = MEMBLOCK WORD(tr,0)  
  REM *** Assign the values ***  
  SetX(temp,x)  
  SetY(temp,y)  
ENDFUNCTION
```

The parameters are the structure that is to be updated (*tr*) and the coordinates to be assigned to the point (*x,y*). Note that the assignment makes use of the functions available within the *PointType* structure.

SetColour()

This function assigns a colour value and has the following code:

```
FUNCTION SetColour(tr AS TriangleType, c1)  
  WRITE MEMBLOCK WORD tr,12,c1  
ENDFUNCTION
```

GetApex1X()

Each value stored within the structure must be retrievable, so we'll need a *Get* function for each one. The code for retrieving the *x* coordinate of *apex1* is:

```
FUNCTION GetApex1X(tr AS TriangleType)
    temp = MEMBLOCK WORD(tr,0)
    result = GetX(temp)
ENDFUNCTION result
```

Again, note the use of an existing function defined within *PointType* to retrieve the value required.

GetColour()

Retrieving the colour value requires the following code:

```
FUNCTION GetColour(tr AS TriangleType)
    result = MEMBLOCK WORD(tr,12)
ENDFUNCTION result
```

Activity Sup047

Start a new project named *TriangleType.dbpro* and create code for each of the following functions:

CreateTriangleType()

Creates the memory block required by the *TriangleType* data.

SetApex1(tr AS TriangleType, x,y)

Sets *apex1.x* to *x* and *apex1.y* to *y*.

SetApex2(tr AS TriangleType, x,y)

Sets *apex2.x* to *x* and *apex2.y* to *y*.

SetApex3(tr AS TriangleType, x,y)

Sets *apex3.x* to *x* and *apex3.y* to *y*.

SetColour(tr AS TriangleType, cl)

Sets *tr.colour* to *cl*.

GetApex1X(tr AS TriangleType):INTEGER

Returns the value of *apex1.x*.

GetApex1Y(tr AS TriangleType):INTEGER

Returns the value of *apex1.y*.

GetApex2X(tr AS TriangleType):INTEGER

Returns the value of *apex2.x*.

GetApex2Y(tr AS TriangleType):INTEGER

Returns the value of *apex2.y*.

GetApex3X(tr AS TriangleType):INTEGER

Returns the value of *apex3.x*.

GetApex3Y(tr AS TriangleType):INTEGER

Returns the value of *apex3.y*.

GetColour(tr AS TriangleType, cl)

Returns the value of *tr.colour*.

ShowTriangle()

Not every function within a data structure involves setting and getting and in this case of our triangle, we'll need a function to actually display the triangle on the screen. This is coded as:

```

FUNCTION ShowTriangle(tr AS TriangleType)
  REM *** Set the colour to be used ***
  INK GetColour(tr),0
  REM *** Retrieve coords of each point ***
  x1 = GetApex1X(tr)
  y1 = GetApex1Y(tr)
  x2 = GetApex2X(tr)
  y2 = GetApex2Y(tr)
  x3 = GetApex3X(tr)
  y3 = GetApex3Y(tr)
  REM *** Draw lines between each point ***
  LINE x1,y1,x2,y2
  LINE x2,y2,x3,y3
  LINE x3,y3,x1,y1
ENDFUNCTION

```

Activity Sup048

Add the *ShowTriangle()* function to your code.

Modify *ShowTriangle()* so that it saves and restores the colour setting used before the triangle is drawn.

Finishing the Code

All we need to do now is incorporate the *PointType* code and define a meaning for *TriangleType*. These are done with the lines:

```

#include "PointType.dba"
constant TriangleType INTEGER

```

Activity Sup049

Add the two new lines and check your code is correct by compiling the code.

It will not be possible to run the code since it contains only functions.

Testing the Data Structure Code

With the structure now coded we can create a program to informally test what we've written. This will involve creating a *TriangleType* variable, setting up its apex's coordinates, the colour, and displaying the triangle on the screen. The code for this new program is given in LISTING-Sup015.

LISTING-Sup015

Testing *TriangleType*

```

#include "TriangleType.dba"
GLOBAL memblockID

t1 AS TriangleType

SET DISPLAY MODE 1280,1024,32
t1 = CreateTriangleType()
SetApex1(t1,500,50)
SetApex2(t1,300,300)
SetApex3(t1,700,300)
SetColour(t1,RGB(255,0,0))
ShowTriangle(t1)
WAIT KEY
END

```

Activity Sup050

Create a new project (*TestTriangle.dbpro*) and enter the code given in LISTING-Sup015.

Copy the files *PointType.dba* and *TriangleType.dba* into the project's folder.

Test the program.

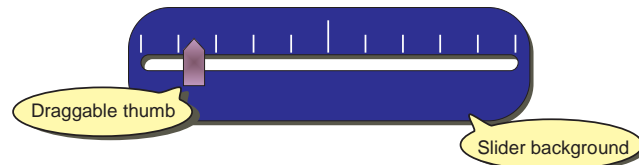
Creating Slider Controls

Introduction

A slider control is similar in concept to a scroll bar, with the user able to drag the thumb across a scaled background (see FIG-Sup018).

FIG-Sup018

The Slider Component



The slider will be constructed from two sprites; one containing the background image and one the thumb image. The figure above may be how we want our slider component to look when it appears on the screen, but we need to store a considerable chunk of data to make this happen.

The *SliderType* Data Structure

The Data

Since the thumb must move independently of the scaled background, each will have to be created as a separate sprite. Since sprites need to be supplied with an image and positioned on the screen, we'll need to store the following details for each sprite:

- its ID
- its image's ID
- its screen coordinates

Since the thumb can be moved only within a limited range along the x-axis, the limits of its movement must be recorded.

A slider might be used to choose a value from 1 to 12 or 0 to 100 (or any other range of values), so the actual range of values must also be stored in our record structure (see FIG-Sup019)

FIG-Sup019

Identifying the Data Required



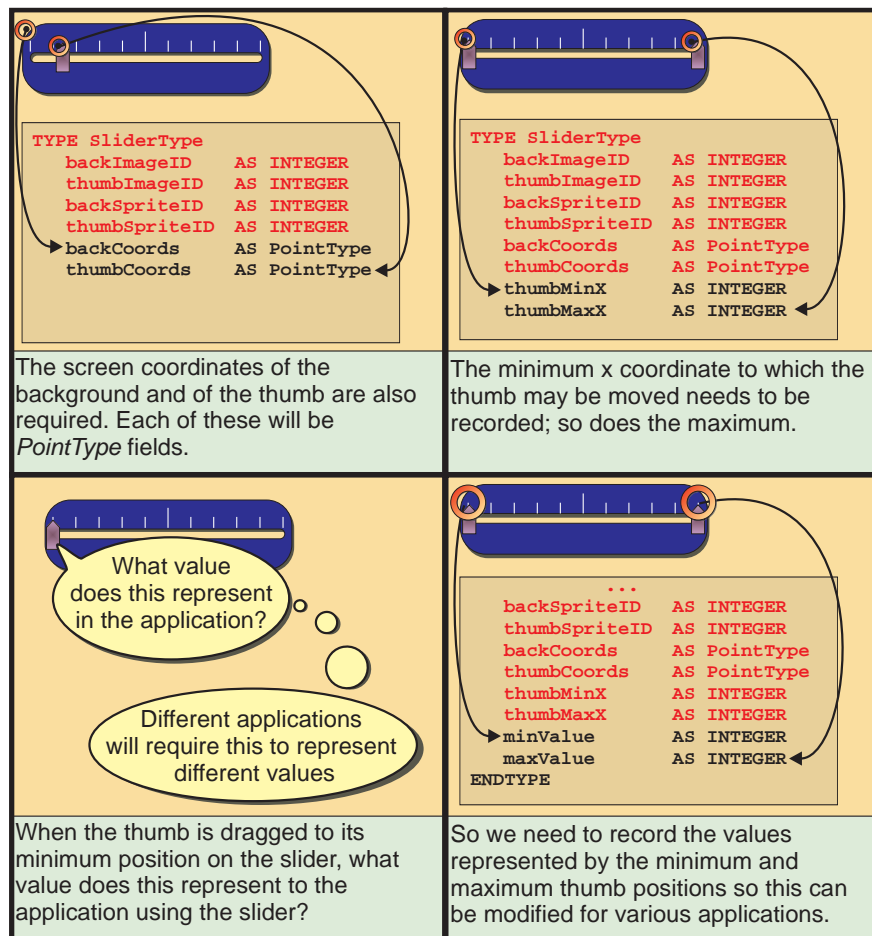
 <pre data-bbox="478 1590 861 1680">TYPE SliderType backImageID AS INTEGER ← thumbImageID AS INTEGER</pre>	 <pre data-bbox="917 1590 1300 1702">TYPE SliderType backImageID AS INTEGER thumbImageID AS INTEGER backSpriteID AS INTEGER ← thumbSpriteID AS INTEGER</pre>
<p>We need to store the IDs of the slider background image and the thumb image.</p>	<p>Since sprites will be used to display the images, we'll also need sprite IDs.</p>

FIG-Sup019

(continued)

Identifying the Data
Required



From the above figure we have the final definition for our structure:

Note that the byte offset of each field has been included as a comment

```

TYPE SliderType
  backImageID AS INTEGER           ~ 0
  thumbImageID AS INTEGER          ~ 4
  backSpriteID AS INTEGER          ~ 8
  thumbSpriteID AS INTEGER         ~12
  backCoords AS PointType          ~16
  thumbCoords AS PointType         ~20
  thumbMinX AS INTEGER             ~24
  thumbMaxX AS INTEGER             ~28
  minValue AS INTEGER              ~32
  maxValue AS INTEGER              ~36
ENDTYPE

```

Activity Sup051

Create a new project (*SliderType.dbpro*) and enter the code given above.

Of course, we won't actually use this code to create our data area; rather we will use a memory block just as we did with *PointType*.

Activity Sup052

What size of memory block is required for a *SliderType* data structure?

The Operations

CreateSliderType()

As usual, with a structure stored within a memory block we need a *create* function which will allocate a memory block of the required size and also create any additional memory blocks required by fields within the record:

```
FUNCTION CreateSliderType()  
    REM *** Determine ID for memory block ***  
    INC memblockID  
    REM *** Create the memory block ***  
    MAKE MEMBLOCK memblockID,44  
    REM *** Save a copy of the ID ***  
    result = memblockID  
    REM *** Create memory blocks required by PointType fields ***  
    WRITE MEMBLOCK WORD result,16,CreatePointType()  
    WRITE MEMBLOCK WORD result,20,CreatePointType()  
ENDFUNCTION result
```

The Setter Functions

Next, we need the functions used to set each of the fields within the record structure:

```
REM *** Setter Operations ***  
FUNCTION SetBackImageID(slider AS SliderType, backingID)  
    WRITE MEMBLOCK WORD slider,0, backingID  
ENDFUNCTION  
  
FUNCTION SetThumbImageID(slider AS SliderType, thumbimgID)  
    WRITE MEMBLOCK WORD slider,4,thumbimgID  
ENDFUNCTION  
  
FUNCTION SetBackSpriteID(slider AS SliderType, backsprID)  
    WRITE MEMBLOCK WORD slider,8,backsprID  
ENDFUNCTION  
  
FUNCTION SetThumbSpriteID(slider AS SliderType, thumbsprID)  
    WRITE MEMBLOCK WORD slider,12,thumbsprID  
ENDFUNCTION  
  
FUNCTION SetBackCoords(slider AS SliderType,x,y)  
    temp = MEMBLOCK WORD(slider,16)  
    SetX(temp,x)  
    SetY(temp,y)  
ENDFUNCTION  
  
FUNCTION SetThumbCoords(slider AS SliderType,x,y)  
    temp = MEMBLOCK WORD(slider,20)  
    SetX(temp,x)  
    SetY(temp,y)  
ENDFUNCTION  
  
FUNCTION SetThumbMinX(slider AS SliderType,xmin)  
    WRITE MEMBLOCK WORD slider,24,xmin  
ENDFUNCTION  
  
FUNCTION SetThumbMaxX(slider AS SliderType,xmax)  
    WRITE MEMBLOCK WORD slider,28,xmax
```

```

ENDFUNCTION

FUNCTION SetMinValue(slider AS SliderType,min)
    WRITE MEMBLOCK WORD slider,32,min
ENDFUNCTION

FUNCTION SetMaxValue(slider AS SliderType,max)
    WRITE MEMBLOCK WORD slider,36,max
ENDFUNCTION

```

The *Getter* Functions

The corresponding *getter* functions return the values of each field:

```

REM *** Getter Operations ***
FUNCTION GetBackImageID(slider AS SliderType)
    result = MEMBLOCK WORD(slider,0)
ENDFUNCTION result

FUNCTION GetThumbImageID(slider AS SliderType)
    result = MEMBLOCK WORD(slider,4)
ENDFUNCTION result

FUNCTION GetBackSpriteID(slider AS SliderType)
    result = MEMBLOCK WORD(slider,8)
ENDFUNCTION result

FUNCTION GetThumbSpriteID(slider AS SliderType)
    result = MEMBLOCK WORD(slider,12)
ENDFUNCTION result

FUNCTION GetBackCoordsX(slider AS SliderType)
    temp = MEMBLOCK WORD(slider,16)
    result = GetX(temp)
ENDFUNCTION result

FUNCTION GetBackCoordsY(slider AS SliderType)
    temp = MEMBLOCK WORD(slider,16)
    result = GetY(temp)
ENDFUNCTION result

FUNCTION GetThumbCoordsX(slider AS SliderType)
    temp = MEMBLOCK WORD(slider,20)
    result = GetX(temp)
ENDFUNCTION result

FUNCTION GetThumbCoordsY(slider AS SliderType)
    temp = MEMBLOCK WORD(slider,20)
    result = GetY(temp)
ENDFUNCTION result

FUNCTION GetThumbMinX(slider AS SliderType)
    result = MEMBLOCK WORD(slider,24)
ENDFUNCTION result

FUNCTION GetThumbMaxX(slider AS SliderType)
    result = MEMBLOCK WORD(slider,28)
ENDFUNCTION result

FUNCTION GetMinValue(slider AS SliderType)
    result = MEMBLOCK WORD(slider,32)
ENDFUNCTION result

FUNCTION GetMaxValue(slider AS SliderType)
    result = MEMBLOCK WORD(slider,36)
ENDFUNCTION result

```

ShowSlider()

The *ShowSlider()* routine displays the slider on the screen and is coded as:

```
FUNCTION ShowSlider(slider AS SliderType)
  REM *** Get back data ***
  backimageID = GetBackImageID(slider)
  backspriteID = GetBackSpriteID(slider)
  backx = GetBackCoordsX(slider)
  backy = GetBackCoordsY(slider)
  REM *** Get thumb data ***
  thumbImageID = GetThumbImageID(slider)
  thumbspriteID = GetThumbSpriteID(slider)
  thumbx = GetThumbCoordsX(slider)
  thumby = GetThumbCoordsY(slider)
  REM *** Display complete slider ***
  SPRITE backspriteID,backx,backy,backimageID
  SPRITE thumbspriteID,thumbx,thumby,thumbimageID
ENDFUNCTION
```

SpriteOverThumb()

The *SpriteOverThumb()* routine returns the distance between the thumb sprite's origin and the mouse tip sprite as measured along the x-axis. If the mouse is not over the thumb or the left mouse button is not being pressed, zero is returned. The routine is coded as:

```
FUNCTION SpriteOverThumb(slider AS SliderType, spriteID)
  REM *** Get thumb's ID ***
  thumbspriteID = GetThumbSpriteID(slider)
  REM *** IF mouse over thumb and left mouse button pressed ***
  IF SPRITE HIT(thumbspriteID, spriteID) AND (MOUSECLICK()=1)
    REM *** Set result to x offset between thumb and mouse***
    result = MOUSEX() - SPRITE X(thumbspriteID)
  ELSE
    REM *** Set result to zero ***
    result = 0
  ENDIF
ENDFUNCTION result
```

DragThumb()

The *DragThumb()* function moves the thumb as it is dragged by the mouse. The function requires as a parameter the x offset value calculated by the *SpriteOverThumb()* function. The function's code is:

```
FUNCTION DragThumb(slider AS SliderType, xoffset)
  mouseCoordX = MOUSEX()
  SetThumbCoords(slider, mouseCoordX-xoffset, GetThumbCoordsY(slider))
  IF GetThumbCoordsX(slider) < GetThumbMinX(slider)
    SetThumbCoords(slider, GetThumbMinX(slider),
  GetThumbCoordsY(slider))
  ELSE IF GetThumbCoordsX(slider) > GetThumbMaxX(slider)
    SetThumbCoords(slider, GetThumbMaxX(slider),
  GetThumbCoordsY(slider))
  ENDIF
  ENDIF
  SPRITE
  GetThumbSpriteID(slider), GetThumbCoordsX(slider), GetThumbCoordsY(sli
  der), GetThumbImageID(slider)
ENDFUNCTION
```

GetSliderValue()

The *GetSliderValue()* function returns the value represented by the thumb's position. This value will lie between *minValue* and *maxValue*. The code for the routine is:

```
FUNCTION GetSliderValue(slider AS SliderType)
    REM *** Calculate thumb's maximum movement ***
    length = GetThumbMaxX(slider) - GetThumbMinX(slider)
    REM *** Calculate thumb's actual movement ***
    distance# = GetThumbCoordsX(slider) - GetThumbMinX(slider)
    REM *** Calculate actual movement as a percentage of ***
    REM *** possible movement ***
    fraction# = distance#/length
    REM *** Calculate value range size ***
    valuegap = GetMaxValue(slider)- GetMinValue(slider)
    REM *** Calculate thumb's value ***
    result = GetMinValue(slider)+ valuegap * fraction#
ENDFUNCTION result
```

Activity Sup053

Add the code for the various functions given above to your *SliderType.dbpro* project.

Comment out the TYPE definition since it was only required in order to give us a simple-to-follow structure for our memory block.

Compile the code to make sure there are no mistakes.

A Slider in Operation

Now we need to create a new project to see a slider operate. In this first program we will simply check out that the thumb can be moved using the mouse and that it cannot be dragged outside the area defined.

First we need to include the *SliderType* and *PointType* files as well as define the *memblockID* global variable:

```
#INCLUDE "SliderType.dba"
#INCLUDE "PointType.dba"
GLOBAL memblockID
```

The program itself will need a *SliderType* variable which we'll make global:

```
GLOBAL slider AS SliderType
```

Now comes the main logic:

```
REM *** Main logic ***
SetUpScreen()
LoadImagesUsed()
SetUpSlider()
DO
    REM *** Move mouse pointer sprite ***
    SPRITE 3,MOUSEX(),MOUSEY(),3
    REM *** Calculate its offset from the thumb ***
    xoffset = MouseOverThumb(slider,3)
    REM *** WHILE mouse dragging thumb DO ***
    WHILE MouseOverThumb(slider,3)
```

```

        REM *** Move thumb to mouse position ***
        DragThumb(slider,xoffset)
        REM *** Move mouse pointer sprite ***
        SPRITE 3,MOUSEX(),MOUSEY(),3
    ENDWHILE
LOOP
END

```

Now we need the code for each function.

SetUpScreen()

```

FUNCTION SetUpScreen()
    REM *** Set Up Screen ***
    SET DISPLAY MODE 1280,1024,32
    COLOR BACKDROP RGB(255,255,255)
    SET IMAGE COLORKEY 255,255,255
ENDFUNCTION

```

LoadImagesUsed()

```

FUNCTION LoadImagesUsed()
    LOAD IMAGE "slider.bmp",1,1
    LOAD IMAGE "thumb.bmp",2,1
    LOAD IMAGE "white.bmp",3,1
ENDFUNCTION

```

SetUpSlider()

```

FUNCTION SetUpSlider()
    slider = CreateSliderType()
    SetBackImageID(slider,1)
    SetThumbImageID(slider,2)
    SetBackSpriteID(slider,1)
    SetThumbSpriteID(slider,2)
    SetBackCoords(slider,20,50)
    SetThumbCoords(slider,20,70)
    SetThumbMinX(slider,20)
    SetThumbMaxX(slider,200)
    SetMinValue(slider,0)
    SetMaxValue(slider,255)
    ShowSlider(slider)
ENDFUNCTION

```

Activity Sup054

Create the test project (*TestSlider.dbpro*) and enter the code given above.

Copy the *PointType.dba* and *SliderType.dba* files into the project folder.

Copy the images used into the project folder.

Test the program and make sure that the thumb can be dragged within the specified limits.