

---

## Data Structures

---

We need two main data structures for the game. The first of these stores details about a single player's piece; the second gives details of a position to which a piece may be moved.

### The *LudoPieceType* Data Structure

For each moveable piece on the board we need to retain the following information:

- The ID number of the 3D model representing that piece
- The piece's colour (coded as 1, 2, 3, or 4)
- The "square" on the board where the piece starts
- The number of "squares" the piece has moved since leaving the home area.

This would give the following structure:

```
TYPE LudoPieceType
  modelID AS INTEGER
  colour AS INTEGER ` 1, 2 3, or 4
  startSquare AS INTEGER
  squaresMoved AS INTEGER
ENDTYPE
```

But in practice, we'll implement it with a memory block and a collection of setter and getter routines:

```
#CONSTANT LudoPieceType INTEGER

FUNCTION CreateLudoPieceType()
  INC memblockID
  result = memblockID
  MAKE MEMBLOCK memblockID,16
ENDFUNCTION result

REM *****
REM ***  Setter functions  ***
REM *****

FUNCTION SetPieceModelID(lp AS LudoPieceType, id)
  WRITE MEMBLOCK WORD lp,0,id
ENDFUNCTION

FUNCTION SetPieceColour(lp AS LudoPieceType, c1)
  WRITE MEMBLOCK WORD lp,4,c1
ENDFUNCTION

FUNCTION SetStartSquare(lp AS LudoPieceType, ss)
  WRITE MEMBLOCK WORD lp,8,ss
ENDFUNCTION

FUNCTION SetSquaresMoved(lp AS LudoPieceType, mv)
  WRITE MEMBLOCK WORD lp,12,mv
ENDFUNCTION

REM *****
REM ***  Getter functions  ***
REM *****
```

```

FUNCTION GetPieceModelID(lp AS LudoPieceType)
    result = MEMBLOCK WORD(lp,0)
ENDFUNCTION result

FUNCTION GetPieceColour(lp AS LudoPieceType)
    result = MEMBLOCK WORD(lp,4)
ENDFUNCTION result

FUNCTION GetStartSquare(lp AS LudoPieceType)
    result = MEMBLOCK WORD(lp,8)
ENDFUNCTION result

FUNCTION GetSquaresMoved(lp AS LudoPieceType)
    result = MEMBLOCK WORD(lp,12)
ENDFUNCTION result

```

In addition, two more functions may prove useful; a function to advance the piece by one "square" (to a maximum of 58) and a function to determine if a piece has moved from the board area onto the stairs area. These routines are coded as:

```

REM *****
REM *** Other functions ***
REM *****

FUNCTION MovePieceOneSquare(lp AS LudoPieceType)
    current = GetSquaresMoved(lp)
    REM *** 58 squares reaches the goal area ***
    IF current >= 58
        EXITFUNCTION
    ENDIF
    ans = current + 1
    SetSquaresMoved(lp, ans)
ENDFUNCTION

FUNCTION IsOnSteps(lp AS LudoPieceType)
    moved = GetSquaresMoved(lp)
    IF moved > 52
        result = moved - 52 `52 squares in board area
    ELSE
        result = 0
    ENDIF
ENDFUNCTION result

```

Notice that *IsOnSteps()* returns either zero (if the piece has not reached the steps) or a number representing the number of stairs climbed.

### Activity Sup059

Create a project named *LudoPieceType.dbpro* and enter the code given above. Include the TYPE definition as a set of comments.

## The *LudoPosition* Data Structure

This structure is used to store the 3D coordinates of a position to which a player's piece may move as well as the ID of any piece at that position. The code for the data and the corresponding operations is:

```

REMSTART
TYPE LudoPositionType
    coords AS Point3DType
    piece AS INTEGER
ENDTYPE
REMEND

```

```

#CONSTANT LudoPositionType INTEGER

FUNCTION CreateLudoPositionType()
  INC memblockID
  result = memblockID
  MAKE MEMBLOCK memblockID,8
  WRITE MEMBLOCK WORD result,0,CreatePoint3DType()
ENDFUNCTION result

REM *****
REM ***   Setter functions   ***
REM *****
FUNCTION SetPieceOn(lbp AS LudoPositionType, id)
  WRITE MEMBLOCK WORD lbp,4,id
ENDFUNCTION

FUNCTION SetPositionCoords(lbp AS LudoPositionType,x#,y#,z#)
  pnt = MEMBLOCK WORD(lbp,0)
  Set3DX(pnt,x#)
  Set3DY(pnt,y#)
  Set3DZ(pnt,z#)
ENDFUNCTION

REM *****
REM ***   Getter functions   ***
REM *****
FUNCTION GetPieceOn(lbp AS LudoPositionType)
  result = MEMBLOCK WORD(lbp,4)
ENDFUNCTION result

FUNCTION GetPositionX(lbp AS LudoPositionType)
  pt = MEMBLOCK WORD(lbp,0)
  result# = Get3DX(pt)
ENDFUNCTION result#

FUNCTION GetPositionY(lbp AS LudoPositionType)
  pt = MEMBLOCK WORD(lbp,0)
  result# = Get3DY(pt)
ENDFUNCTION result#

FUNCTION GetPositionZ(lbp AS LudoPositionType)
  pt = MEMBLOCK WORD(lbp,0)
  result# = Get3DZ(pt)
ENDFUNCTION result#

```

### Activity Sup060

Create a project named *LudoPositionType.dbpro* and enter the code given above. Include the TYPE definition as a set of comments.

### Activity Sup061

Copy the files *LudoPieceType.dba* and *LudoPositionType.dba* into your *Ludo* folder.

---

## Updating the Ludo Game

---

### New Data

With our basic data structures defined, we can return to the Ludo game and create arrays to represent the home, board, stairs and goal areas of the board. Since we'll use these throughout our code, we'll make the arrays global.

We start by including the new data structures:

```
#INCLUDE "PointType.dba"  
#INCLUDE "SliderType.dba"  
#INCLUDE "Point3DType.dba"  
#INCLUDE "LudoPositionType.dba"  
#INCLUDE "LudoPieceType.dba"
```

Now we can create the required arrays:

```
GLOBAL DIM home(18) AS LudoPositionType  
GLOBAL DIM board(52) AS LudoPositionType  
GLOBAL DIM stairs(4,5) AS LudoPositionType  
GLOBAL DIM goal(18) AS LudoPositionType
```

and finally the array containing details of the players' pieces:

```
GLOBAL DIM pieces(18) AS LudoPieceType
```

The *home*, *goal* and *pieces* arrays are set up in such a way that the ID of a piece and its position in these arrays match. Hence the first red piece, which has a 3D object ID of 3 will have its details stored in element 3 of *home*, *goal* and *pieces*, while the last green piece, whose ID is 18, will be stored in element 18 of the arrays. Of course, this means that elements 1 and 2 in these three arrays will be unused.

## Updated Functions

The *SetUpGame()* function now needs to create the memory block used by the data structures in our new arrays and store the required data in the elements of those arrays. The updated version of the routine is:

```
FUNCTION SetUpGame()  
  LoadImagesUsed()  
  LoadControlFrame()  
  SetUpDataStructures()  
  LoadHomeData()  
  LoadBoardData()  
  LoadStairsData()  
  LoadGoalData()  
  LoadPiecesData()  
  LoadModels()  
ENDFUNCTION
```

The *LoadModels()* function also needs updating since the players' pieces can now be positioned:

```
FUNCTION LoadModels()  
  LOAD OBJECT "ludoboard.x", boardObj  
  LOAD OBJECT "dice.x", diceObj  
  SCALE OBJECT diceObj, 20, 20, 20  
  POSITION OBJECT diceObj, 0, -55, 0  
  REM *** Load player model of each colour ***  
  LOAD OBJECT "redpiece.x", firstRedPiece  
  LOAD OBJECT "bluepiece.x", firstBluePiece  
  LOAD OBJECT "yellowpiece.x", firstYellowPiece  
  LOAD OBJECT "greenpiece.x", firstGreenPiece  
  REM *** Reduce size of models ***  
  SCALE OBJECT firstRedpiece, 20, 20, 20  
  SCALE OBJECT firstBluepiece, 20, 20, 20  
  SCALE OBJECT firstYellowpiece, 20, 20, 20  
  SCALE OBJECT firstGreenpiece, 20, 20, 20  
  REM *** Make three clones of each colour and ***  
  REM *** place each in correct home area ***
```

```

r = 3
FOR c = 1 TO 4
  x# = GetPositionX(home(r))
  y# = GetPositionY(home(r))
  z# = GetPositionZ(home(r))
  POSITION OBJECT r, x#,y#,z#
  FOR p = 1 TO 3
    INC r
    x# = GetPositionX(home(r))
    y# = GetPositionY(home(r))
    z# = GetPositionZ(home(r))
    CLONE OBJECT r, (c-1)*4+3
    POSITION OBJECT r, x#,y#,z#
  NEXT p
  INC r
NEXT c
ENDFUNCTION

```

### Activity Sup062

Update your existing functions to match the latest code given above.

## New Functions

Most of the new functions we require are called by *SetUpGame()* and therefore have already been named.

*SetUpDataStructures()* creates the memory blocks required by the new arrays:

```

FUNCTION SetUpDataStructures()
  REM *** Create home area ***
  FOR r = 1 TO 18
    home(r)= CreateLudoPositionType()
  NEXT c
NEXT r
  REM *** Create board area ***
  FOR c = 1 TO 52
    board(c)= CreateLudoPositionType()
  NEXT c
  REM *** Create pieces area ***
  FOR r = 1 TO 18
    pieces(r,c)= CreateLudoPieceType()
  NEXT r
  REM *** Set up stairs area ***
  FOR r = 1 TO 4
    FOR c = 1 TO 5
      stairs(r,c)= CreateLudoPositionType()
    NEXT c
  NEXT r
  REM *** Create goal area ***
  FOR r = 1 TO 18
    goal(r,p)= CreateLudoPositionType()
  NEXT r
ENDFUNCTION

```

*LoadHomeData()* loads the details into the *home* array. These details give the coordinates of where a piece should be placed and the ID of the piece starting at that position:

```

FUNCTION LoadHomeData()
  REM *** Position coordinates for home areas ***
  DATA -8.5,1.03,-8, -8.5,1.03,-6.5, -6.5,1.03,-8,
  ↵-6.5,1.03,-6.5
  DATA -8.5,1.03,8, -8.5,1.03,6.5, -6.5,1.03,8,

```

```

        ↵-6.5,1.03,6.5
DATA 8.5,1.03,8,      8.5,1.03,6.5,      6.5,1.03,8,
        ↵6.5,1.03,6.5
DATA 8.5,1.03,-8,    8.5,1.03,-6.5,    6.5,1.03,-8,
        ↵6.5,1.03,-6.5
pieceID = firstRedPiece
FOR r = 3 To 18
    SetPieceOn(home(r), pieceID)
    INC pieceID
    READ x#,y#,z#
    SetPositionCoords(home(r), x#,y#,z#)
NEXT r
ENDFUNCTION

```

*LoadBoardData()* loads the coordinates of every "square" on the board. Since every position starts off empty, the ID value is set to zero:

```

FUNCTION LoadBoardData()
DATA -1.55,1.15,-9.95, -1.55,1.15,-8.45, -1.55,1.15,-6.95,
    ↵-1.55,1.15,-5.45, -1.55,1.15,-3.95, -1.55,1.15,-2.45
DATA -3.05,1.15,-1.45, -4.55,1.15,-1.45, -6.05,1.15,-1.45,
    ↵-7.55,1.15,-1.45, -9.05,1.15,-1.45, -10.55,1.15,-1.45
DATA -10.55,1.15,0.05
DATA -10.55,1.15,1.55, -9.05,1.15,1.55, -7.55,1.15,1.55,
    ↵-6.05,1.15,1.55, -4.55,1.15,1.55, -3.05,1.15,1.55
DATA -1.55,1.15,2.50, -1.55,1.15,4.00, -1.55,1.15,5.5,
    ↵-1.55,1.15,7, -1.55,1.15,8.5, -1.55,1.15,10
DATA -0.05,1.15,10
DATA 1.45,1.15,10,      1.45,1.15,8.5,      1.45,1.15,7,
    ↵1.45,1.15,5.5,      1.45,1.15,4,      1.45,1.15,2.5
DATA 3,1.15,1.55,      4.5,1.15,1.55,      6,1.15,1.55,
    ↵7.5,1.15,1.55,      9,1.15,1.55,      10.5,1.15,1.55
DATA 10.5,1.15,0.05
DATA 10.5,1.15,-1.45,  9,1.15,-1.45,      7.5,1.15,-1.45,
    ↵6,1.15,-1.45,      4.5,1.15,-1.45,      3,1.15,-1.45
DATA 1.45,1.15,-2.45,  1.45,1.15,-3.95,      1.45,1.15,-5.45,
    ↵1.45,1.15,-6.95,      1.45,1.15,-8.45,      1.45,1.15,-9.95
DATA -0.05,1.15,-9.95
FOR c = 1 TO 52
    READ x#,y#,z#
    SetPositionCoords(board(c), x#,y#,z#)
    SetPieceOn(board(c), 0)
NEXT c
ENDFUNCTION

```

*LoadStairsData()* loads the coordinates of the stairs leading to the goal platform:

```

FUNCTION LoadStairsData()
DATA -0.05,2.35,-8.45, -0.05,3.35,-6.95, -0.05,4.35,-5.45,
    ↵-0.05,5.35,-3.95, -0.05,6.35,-2.45
DATA -9.05,2.35,0.05, -7.55,3.35,0.05, -6.05,4.35,0.05,
    ↵-4.55,5.35,0.05, -3.05,6.35,0.05
DATA -0.05,2.35,8.5, -0.05,3.35,7, -0.05,4.35,5.5,
    ↵-0.05,5.35,4, -0.05,6.35,2.5
DATA 9,2.35,0.0,      7.5,3.35,0.0,      6,4.35,0.0,
    ↵4.5,5.35,0.0,      3,6.35,0.0
FOR r = 1 TO 4
    FOR c = 1 TO 5
        READ x#,y#,z#
        SetPositionCoords(stairs(r,c), x#,y#,z#)
        SetPieceOn(stairs(c), p)
    NEXT c
NEXT r
ENDFUNCTION

```

The final set of coordinates are for the goal platform. These are set up using *LoadGoalData()*:

```

FUNCTION LoadGoalData ()
  DATA -0.05,6.75,-0.4,  -1.4.,6.75,-1.5,  -0.05,6.75,-1.5,
    ↳1.35,6.75,-1.5,
  DATA -0.6,6.75,-0.05,  -2,6.75,1.1,  -2,6.75,-0.05,
    ↳-2,6.75,-1.1,
  DATA -0.05,6.75,0.4,  1.35,6.75,1.5,  -0.05,6.75,1.5,
    ↳-1.4,6.75,1.5
  DATA 0.6,6.75,-0.05,  2,6.75,-1.1,  2,6.75,-0.05,
    ↳2,6.75,1.1
  FOR r = 3 TO 18
    READ x#,y#,z#
    SetPositionCoords (goal (r) ,x#,y#,z#)
    SetPieceOn (goal (r) ,0)
  NEXT r
ENDFUNCTION

```

*LoadPiecesData()* is the last of the new functions and loads details of each moveable piece:

```

FUNCTION LoadPiecesData ()
  FOR r = 3 To 18
    SetPieceModelID (pieces (r) ,r)
    SetPieceColour (pieces (r) , (r-3)/4 + 1)
    SetStartSquare (pieces (r) , (GetPieceColour (pieces (r) )-1)
      ↳*13+1)
    SetSquaresMoved (pieces (r) ,0)
  NEXT r
ENDFUNCTION

```

### Activity Sup063

Add the new functions to your code and check that it runs correctly.

## Moving a Piece

Although we won't be moving a piece under player control until the next section, it would nevertheless be useful to check that the coordinates stored in the various arrays have been entered correctly, so we'll add a temporary function to move a single piece all the way round the board and into the goal area.

The routine, *TestMove()*, takes one parameter. This represents the ID of the piece to be moved as held in the *pieces* array.

```

FUNCTION TestMove (p)
  MovePieceOneSquare (pieces (p) )
  square = (GetSquaresMoved (pieces (p) )+
    ↳GetStartSquare (pieces (p) )-2) mod 52 + 1
  px = IsOnSteps (pieces (p) )
  IF NOT px
    POSITION OBJECT id,GetPositionX (board (square) ) ,
      ↳GetPositionY (board (square) ) ,GetPositionZ (board (square) )
  ELSE
    IF px < 6
      POSITION OBJECT p,GetPositionX (stairs (p) ) ,
        ↳GetPositionY (stairs (p) ) ,GetPositionZ (stairs (r,p) )
    ELSE
      POSITION OBJECT id,GetPositionX (goal (p) ) ,
        ↳GetPositionY (goal (p) ) ,GetPositionZ (goal (p) )

```

```

ENDIF
ENDIF
WAIT 10
ENDFUNCTION

```

There are a few lines of code to note from this routine. The square to which a piece should be moved is calculated by the line:

```

square = (GetSquaresMoved(pieces(p)) +
↳GetStartSquare(pieces(p)) - 2) mod 52 + 1

```

Although red pieces move from squares 1 to 52 before climbing the stairs, other colours have to move past square 52 back round to square 1. For example, blue pieces move from square 14 on through square 52 and back round to square 13 before climbing their set of stairs. So the rather complex calculation ensures that a piece can move forward from square 52 to square 1.

For the first 52 moves the 3D coordinates at which a piece should be placed are stored in the *board* array, but if a piece has moved between 53 and 57 squares then its position is obtained from the *stairs* array and the coordinates for the 58<sup>th</sup> move come from the *goal* array. It is for this reason that the code contains 3 alternative POSITION OBJECT statements.

Finally, to test movement of the piece we need to change the main logic of the game to:

```

REM *****
REM ***   Main Game Logic   ***
REM *****
SetUpScreen ()
SetUpGame ()
DO
  HandleSlider ()
  TestMove (1, 1)
LOOP
END

```

#### Activity Sup064

Add the *TestMove()* function and update the main logic of your program.

Does the piece move correctly around the board?

Change the call to *TestMove()* so that a yellow piece is moved.

Add a second call to *Test Move()* so that both a yellow and blue piece are moved.