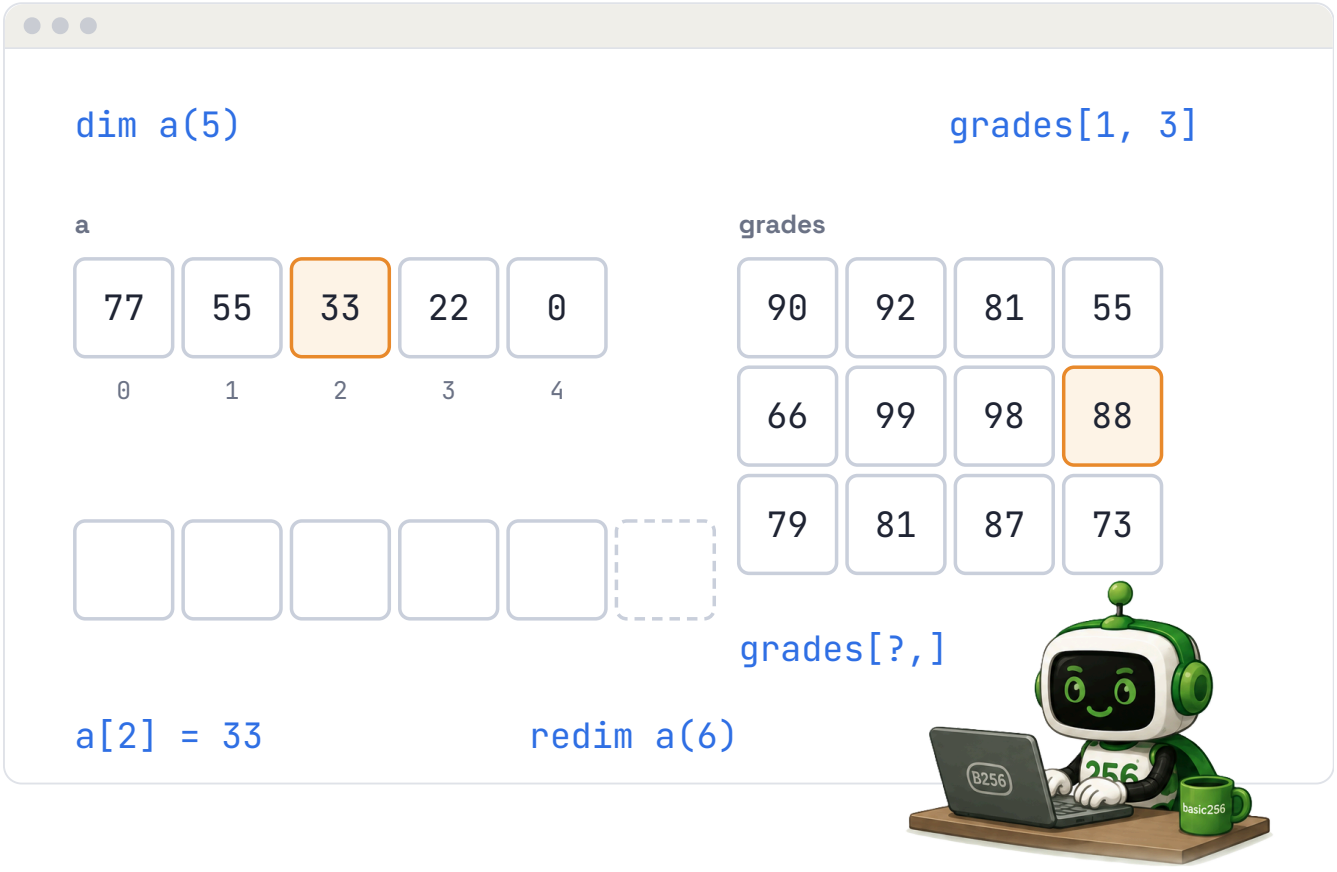
LEARN TO CODE · LISTS AND TABLES

Chapter 15

Arrays — Collections of Information

So far every variable has held one value: one number or one string. But programs often work with whole collections: the names of your friends, the scores of a class, the places of fifty bouncing balls. An **array** is a variable that holds a whole list of values, each with its own number. In this chapter you make arrays, fill them, use them for sounds and shapes, build tables with rows and columns, pass them to subroutines, and make them bigger. At the end you dodge falling space debris.



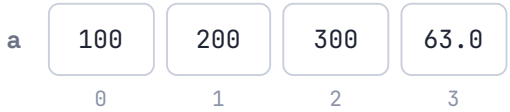
© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 1

A list of numbers

A one-dimensional array is a numbered list of values, kept together under one name.

Think of an array as a row of boxes. The whole row has one name, and each box has a number, its **index**. The boxes are numbered from 0, so an array of four boxes has the indexes 0, 1, 2 and 3. To use one box, write the name of the array with the index in square brackets: `a[2]` is box number 2 of the array *a*.



Program 97 makes an array of four numbers with **dim**. It puts values in the first three boxes, asks the user for the fourth, and then prints all four with a **for** loop. The loop variable *t* goes from 0 to 3, so `a[t]` is each box in turn.

arraynumeric1d.kbs BASIC-256

```
# arraynumeric1d.kbs - a one-dimensional array of numbers

dim a(4)

a[0] = 100
a[1] = 200
a[2] = a[0] + a[1]
inputfloat "Enter a number: ", a[3]

for t = 0 to 3
   print "a[" + t + "] = " + a[t]
next t
```

TEXT OUTPUT

```
Enter a number: 63
a[0] = 100
a[1] = 200
a[2] = 300
a[3] = 63.0
```

Program 97 · One-dimensional Numeric Array

inputfloat (Chapter 6) stores the number it reads straight into box `a[3]`. Because it reads a decimal number, 63 is stored as 63.0.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



NEW CONCEPT

```
dim variable(items)
dim variable(items) fill expression
dim variable(rows, columns)
```

dim makes an array in the computer's memory, with as many elements as you give in the brackets. The size must be a whole number of at least 1. Every element starts at 0, or at the empty string `""` when the name of the array ends in **$**. With **fill**, every element starts with a value of your own instead, for example `dim x(50) fill 10`.

A two-dimensional array has rows and columns, like a table. More about that in Section 4.



NEW CONCEPT

```
variable[index]
variable[row, column]
```

An array element, the name with an index in square brackets, can be used anywhere you would use an ordinary variable: in a calculation, in **print**, on the left of an `=`, or after `+=`.

The first element has index 0 and the last one has index size − 1. An array made with `dim a(4)` has no `a[4]`: using it stops the program with an *index out of bounds* error. Computer people call this a **zero-indexed** array.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 2

Lists of names, lists of balls

Arrays hold strings as easily as numbers, and loops make them really useful.

An array can hold strings too. Program 98 asks how many friends you have and makes an array just that big. Then a loop asks for each name and stores it in the next element. The size of the array is a variable, *n*: the program works for 3 friends or for 30.

listoffriends.kbs BASIC-256

```
# listoffriends.kbs - use an array to store any number of names

print "Make a list of my friends."
inputinteger "How many friends do you have? ", n

dim names(n)
for i = 0 to n - 1
   inputstring "Enter the name of a friend: ", names[i]
next i

# show the names
cls
print "My friends:"
for i = 0 to n - 1
   print "Friend number " + (i + 1) + " is " + names[i]
next i

# pick one at random
x = int(rand * n)
print "The winner is " + names[x] + "!"
end
```

Program 98 · List of My Friends

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

TEXT OUTPUT

```
Make a list of my friends.
How many friends do you have? 3
Enter the name of a friend: Kendra
Enter the name of a friend: Bob
Enter the name of a friend: Susan

My friends:
Friend number 1 is Kendra
Friend number 2 is Bob
Friend number 3 is Susan
The winner is Kendra!
```

The text output before and after **cls**.

After the last name, **cls** clears the text output, so the list starts on a clean window. Then the program picks a winner: `int(rand * n)` (Chapter 7) is a random whole number from 0 to *n* – 1, exactly the indexes of the array. Friend number 1 is in element 0, so the program prints `i + 1` as the friend's number.



NEW CONCEPT

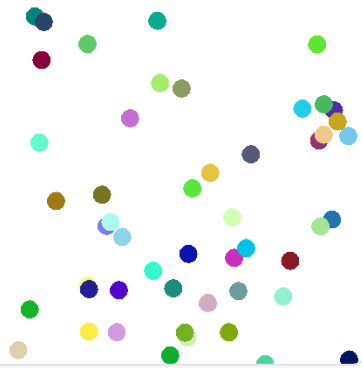
```
cls
```

cls clears the text output, just as **clg** clears the graphics output.

Arrays really pay off when a program has many things of the same kind. Program 99 bounces fifty balls, each with its own place, speed and color. Without arrays it would need 250 variables. With arrays it needs five: *x* and *y* for the places, *dx* and *dy* for the speeds, and *colors*. Element *b* of each array belongs to ball number *b*.

The first loop gives every ball a random speed from 2 to 12 pixels and a random color, made with **rgb** (Chapter 2). `fill r` starts every ball at 10, 10, so they all come out of the top left corner. In the animation loop, the inner **for** loop moves, bounces and draws each ball in turn. Like Programs 89 and 90 in Chapter 13, a ball bounces when its center is one radius from an edge.

GRAPHICS OUTPUT



A moment from the animation.

manyballbounce.kbs BASIC-256

```
# manyballbounce.kbs - many balls, each with its own
# place, speed and color
```

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

```
fastgraphics

r = 10          # the radius of every ball
balls = 50      # the number of balls

# the position of the balls: all start in the top left corner
dim x(balls) fill r
dim y(balls) fill r

# the speed and the color of each ball, chosen at random
dim dx(balls)
dim dy(balls)
dim colors(balls)
for b = 0 to balls - 1
   dx[b] = rand * r + 2
   dy[b] = rand * r + 2
   colors[b] = rgb(rand * 256, rand * 256, rand * 256)
next b

while true
   clg white
   for b = 0 to balls - 1
      # move the ball
      x[b] += dx[b]
      y[b] += dy[b]
      # bounce off the sides, the top and the bottom
      if x[b] < r or x[b] > graphwidth - r then dx[b] = -dx[b]
      if y[b] < r or y[b] > graphheight - r then dy[b] = -dy[b]
      # draw it in its own color
      color colors[b]
      circle x[b], y[b], r
   next b
   refresh
   pause .02
end while
```

Program 99 · Bounce Many Balls



TRY IT

Change *balls* to 500 and *r* to 3. How many lines of the program did you have to change? Then start the balls in the middle of the screen with `dim x(balls) fill graphwidth / 2`.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 3

Filling an array from a list

A list in curly braces makes an array and fills it in one go.

You already know lists in curly braces: **sound** played them in Chapter 4, and **poly** and **stamp** drew them in Chapter 9. Give a list to a variable and the variable becomes an array, with the first value in element 0. Program 100 makes two arrays this way.

arrayassign.kbs BASIC-256

```
# arrayassign.kbs - make and fill arrays from lists

numbers = {56, 99, 145}
names = {"Bob", "Jim", "Susan"}

for i = 0 to 2
   print numbers[i] + " " + names[i]
next i
```

TEXT OUTPUT

```
56 Bob
99 Jim
145 Susan
```

Program 100 · Assigning an Array With a List



NEW CONCEPT

```
variable = {value0, value1, …}
variable = {{v00, v01, …}, {v10, v11, …}, …}
```

A list in curly braces turns a variable into an array, sized to fit the list. The values may be numbers or strings. A list of lists makes a two-dimensional array: each inner list is one row. A long list may go on over several lines, as long as each line ends after a comma.

Sound and arrays

sound also plays an array. As with a list, the elements 0, 2, 4 … hold the frequencies and the elements 1, 3, 5 … the durations. Program 101 works out fifty notes with a formula, which would be a very long list to type, and plays them as one quick chirp.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

spacechirp.kbs BASIC-256

```
# spacechirp.kbs - play a spacy sound from an array

# even elements 0, 2, 4 ... hold a frequency
# odd elements 1, 3, 5 ... hold a duration

# 50 notes of 10 milliseconds each: the first is 100 Hz,
# and each note is 80 Hz higher than the one before
dim a(100)
for i = 0 to 98 step 2
   a[i] = i * 40 + 100
   a[i + 1] = 10
next i
sound a
end
```

Program 101 · Space Chirp Sound

The loop goes up in steps of 2, so *i* is always the index of a frequency, and `i + 1` the index of the duration after it. Because *i* grows by 2, each frequency (`i * 40 + 100`) is 80 Hz higher than the one before: from 100 Hz up to 4020 Hz.



EXPLORE

What kind of crazy sounds can you make? Experiment with the formulas for the frequencies and durations. Try `a[i] = 4000 - i * 40` , or `a[i] = rand * 1000 + 200` , or make every note 30 milliseconds long.

Graphics and arrays

Arrays can hold shapes for **stamp** and **poly**: the elements 0, 2, 4 ... are the *x* values of the points, and 1, 3, 5 ... the *y* values. Define a shape once, in an array, and use it as often as you like. Program 102 takes the X from Program 77 in Chapter 11 and stamps it twice: first in gray, a little lower and to the right, then in black on top. The gray X looks like its shadow.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

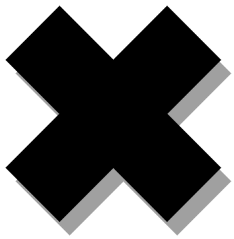
shadowstamp.kbs BASIC-256

```
# shadowstamp.kbs - one shape in an array, stamped twice

xmark = {-1, -2, 0, -1, 1, -2, 2, -1, 1, 0, 2, 1,
   1, 2, 0, 1, -1, 2, -2, 1, -1, 0, -2, -1}

clg white
# the shadow: gray, a little lower and to the right
color grey
stamp 265, 270, 80, xmark
# the X itself
color black
stamp 250, 250, 80, xmark
```

GRAPHICS OUTPUT



Program 102 · Shadow Stamp

Arrays can also make shapes by calculation. Program 103 fills an array of 10 elements, five points, with random places and draws it with **poly**. Where the lines cross, the fill leaves holes, as you can see.

randompoly.kbs BASIC-256

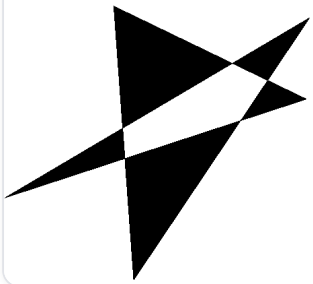
```
# randompoly.kbs - a polygon with 5 random corners

dim shape(10)

for t = 0 to 8 step 2
   shape[t] = rand * graphwidth          # x
   shape[t + 1] = rand * graphheight     # y
next t

clg white
color black
poly shape
```

GRAPHICS OUTPUT



Every run draws a different polygon.

Program 103 · Randomly Create a Polygon

With or without [ ]? Older programs write an empty pair of square brackets after an array that is given to a statement: `sound a[]`, `stamp 250, 250, 80, xmark[]`, `poly shape[]`. BASIC-256 2.3 accepts both forms. The brackets are a reminder that the whole array is meant, not one value.

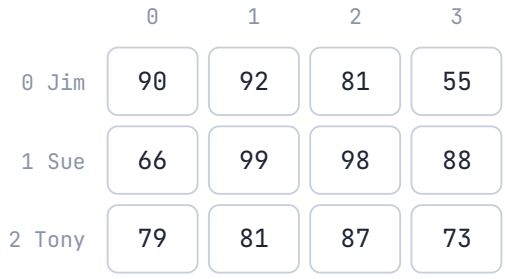
© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 4

Tables: two-dimensional arrays

An array with rows and columns keeps a whole table of values.

The arrays so far were one row of boxes: **one-dimensional** arrays. A **two-dimensional** array is a table of rows and columns, and an element has two indexes: first the row, then the column. In Program 104, each row of *grades* belongs to a student and each column to a test, so `grades[1, 3]` is Sue's fourth test.



The array *grades*: rows 0 to 2, columns 0 to 3. The names are in a one-dimensional array of their own, *students*, in the same order.

grades.kbs BASIC-256

```
# grades.kbs - the average grade of each student
# and of the whole class, in a two-dimensional array

nstudents = 3    # the number of students (rows)
nscores = 4      # the number of scores per student (columns)

students = {"Jim", "Sue", "Tony"}
# one row per student, one column per test
grades = {{90, 92, 81, 55},
          {66, 99, 98, 88},
          {79, 81, 87, 73}}

total = 0
for row = 0 to nstudents - 1
   studenttotal = 0
   for column = 0 to nscores - 1
      studenttotal += grades[row, column]
      total += grades[row, column]
   next column
   print students[row] + "'s average is " + studenttotal / nscores
next row
print "The class average is " + total / (nscores * nstudents)
end
```

Program 104 · Grade Calculator

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

TEXT OUTPUT

```
Jim's average is 79.5
Sue's average is 87.75
Tony's average is 80.0
The class average is 82.4166666667
```

Two loops, one inside the other, visit every element: the outer loop goes through the rows, the inner loop through the columns of that row. The inner loop adds the scores of one student; *total* adds up all twelve scores, for the class average. The table is written as a list of lists, one row to a line, so it looks like the table above.



NEW CONCEPT

```
dim variable(rows, columns)
variable[row, column]
```

A two-dimensional array is made with **dim** and two sizes, the number of rows and the number of columns, or from a list of lists. Its elements are read and changed with two indexes, row first, both counted from 0. `dim table(3, 4)` makes 3 rows of 4 columns, 12 elements in all, from `table[0, 0]` to `table[2, 3]`.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 5

Array sizes, and arrays in subroutines

Programs can ask how big an array is, and subroutines can work with arrays of any size.

Sometimes a program must work with an array of any size. Put a question mark where the index goes, and you get the size of the array instead: `a[?]` is how many elements *a* has. Program 105 gives two arrays of different sizes to the same subroutine, which prints how many elements each one has and then every element.

size.kbs BASIC-256

```
# size.kbs - the size of an array, and passing
# an array to a subroutine

print "The number array:"
number = {77, 55, 33}
call showarray(number)

print "The random array:"
dim r(5)
for i = 0 to r[?] - 1
   r[i] = int(rand * 10) + 1
next i
call showarray(r)
end

subroutine showarray(a)
   print "It has " + a[?] + " elements."
   for i = 0 to a[?] - 1
      print "element " + i + ": " + a[i]
   next i
end subroutine
```

TEXT OUTPUT

```
The number array:
It has 3 elements.
element 0: 77
element 1: 55
element 2: 33
The random array:
It has 5 elements.
element 0: 6
element 1: 2
element 2: 7
element 3: 10
element 4: 4
```

Program 105 · Get Array Size

The subroutine never needs to be told the size: `a[?]` tells it. Its loop runs to `a[?] - 1`, the last index. The main program uses `r[?]` the same way to fill its random array.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



NEW CONCEPT

```
array[?]
array[?,]
array[,?]
```

`[?]` gives the number of elements of an array. For a two-dimensional array it gives all of them, rows × columns. `[?,]` gives the number of rows and `[,?]` the number of columns.

In Chapter 10 you saw that a subroutine gets a **copy** of what you give it. That is true for arrays too: the subroutine in Program 105 could change `a[0]`, but *number* in the main program would stay as it was. When a subroutine or function must change the array itself, give it the array with **ref**. Program 106 doubles every element of an array in a subroutine.

changearray.kbs BASIC-256

```
# changearray.kbs - a subroutine that changes an array

numbers = {1, 2, 3}
call double(ref(numbers))
for i = 0 to numbers[?] - 1
   print numbers[i]
next i
end

subroutine double(ref(a))
   # double every element of the array
   for i = 0 to a[?] - 1
      a[i] = a[i] * 2
   next i
end subroutine
```

TEXT OUTPUT

```
2
4
6
```

Program 106 · Change an Array in a Subroutine

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



NEW CONCEPT

```
call name(ref(array))
subroutine name(ref(array))
```

ref gives a subroutine or function a **reference** to a variable or an array: not a copy, but the thing itself. Write **ref** both in the **call** and in the first line of the subroutine. Everything the subroutine does to the array then happens to the array of the caller. Without **ref**, the subroutine works on a copy and the caller's array is left as it was.



TRY IT

Take **ref** out of Program 106, in both places, and run it again. What does it print now, and why? Then change Program 104 so that it works out *nstudents* and *nscores* by itself with `grades[?,]` and `grades[,?]`, and add a fourth student.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

SECTION 6

Changing the size of an array

redim makes an array bigger or smaller and keeps what is in it.

An array doesn't have to keep the size it was made with. **redim** gives it a new size and keeps the values that are already in it. Program 107 makes an array of three numbers from a list, makes it one element bigger, and puts a value in the new element.

redim.kbs BASIC-256

```
# redim.kbs - make an array bigger

number = {77, 55, 33}
redim number(4)      # one more element at the end
number[3] = 22

for i = 0 to number[?] - 1
   print i + " " + number[i]
next i
```

TEXT OUTPUT

```
0 77
1 55
2 33
3 22
```

Program 107 · Re-Dimension an Array



NEW CONCEPT

```
redim variable(items)
redim variable(rows, columns)
redim variable(items) fill expression
```

redim gives an array a new size. The values that fit are kept. New elements start at 0 (or `""` for a name ending in **$**), or with the value after **fill**. When the array gets smaller, the elements at the end are cut off and lost.

A two-dimensional array keeps its values in reading order, row after row. If you change the number of columns, the values move to other places: a table of 2 rows of 1, 2, 3 and 4, 5, 6, redimmed to 3 rows of 2 columns, becomes 1, 2 / 3, 4 / 5, 6.

There is more to arrays. BASIC-256 can do a lot more with arrays. **arrayadd** adds a value or a row to the end of an array, so a list can grow one item at a time without **redim**. **sort** puts an array in order. And the **mat** statements and the **dot**, **cross**, **norm** and **unit** operators calculate with whole arrays as matrices and vectors. These come in a later chapter for advanced programmers.

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



BIG PROGRAM

Space Warp

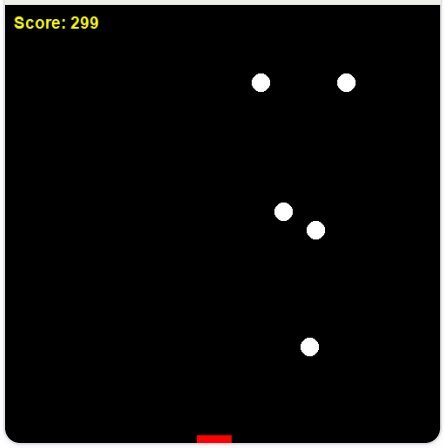
Pieces of space debris fall from the sky, each at its own speed. Move your ship left and right to dodge them. Every round you survive scores a point, but watch out: your ship slowly grows.

The program keeps the debris in three arrays: the *x* and *y* of each piece and its speed. Element *n* of each array belongs to piece number *n*, just like the balls of Program 99.

In every round, each piece falls by its own speed. A piece that falls off the bottom goes back to the top, at a new random place and speed. If a piece reaches the ship and is not further left or right than the ship plus its radius (*hitleft* and *hitright*), the game is over.

Hold J to move left and K to move right; **keypressed** (Chapter 12) keeps the ship moving as long as you hold the key. Q quits.

GRAPHICS OUTPUT



TEXT OUTPUT

```
Space Warp: hold J or K to dodge the falling
space debris. Press Q to quit.
Your score: 1890
Game over.
```

The text output after a game.

TRY IT



Make the game harder as it goes: every 500 points, add one more piece of debris. You will need **redim** for all three arrays, and give the new piece a place and a speed. Remember that *balln* must grow too.

spacewarp.kbs BASIC-256

```
# spacewarp.kbs - dodge the falling space debris

# the debris: one element of each array per piece
balln = 5          # the number of pieces
ballr = 10         # their radius
dim ballx(balln)
dim bally(balln)
dim ballspeed(balln)
```

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

```
# the player: a red bar at the bottom of the screen
playerw = 30      # its width, which grows during the game
playerh = 10
playerx = (graphwidth - playerw) / 2
playery = graphheight - playerh
playerm = 5       # how far it moves each round
growpercent = 0.03   # the chance that it grows each round

keyj = asc("J")
keyk = asc("K")
keyq = asc("Q")

print "Space Warp: hold J or K to dodge the falling"
print "space debris. Press Q to quit."

# every piece starts at the top, at a random place and speed
for n = 0 to balln - 1
   bally[n] = ballr
   ballx[n] = int(rand * (graphwidth - 2 * ballr)) + ballr
   ballspeed[n] = int(rand * 5) + 2
next n

fastgraphics
font "Arial", 14, 700
score = 0
more = true
while more
   clg black
   # move and draw the debris, and check for a hit
   color white
   hitleft = playerx - ballr
   hitright = playerx + playerw + ballr
   for n = 0 to balln - 1
      bally[n] += ballspeed[n]
      if bally[n] > graphheight + ballr then
         # it fell off the bottom: back to the top
         bally[n] = -ballr
         ballx[n] = int(rand * (graphwidth - 2 * ballr)) + ballr
         ballspeed[n] = int(rand * 5) + 2
      end if
      circle ballx[n], bally[n], ballr
      if bally[n] + ballr >= playery then
         if ballx[n] >= hitleft and ballx[n] <= hitright then more = false
      end if
   next n
   # draw the player and the score
   color red
   rect playerx, playery, playerw, playerh
   color yellow
   text 10, 10, "Score: " + score
   refresh
   score += 1
   if rand < growpercent then playerw += 1
   # move the player while J or K is held down
   if keypressed(keyj) then playerx -= playerm
   if keypressed(keyk) then playerx += playerm
   if keypressed(keyq) then more = false
```

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

```
    # keep the player on the screen
    if playerx < 0 then playerx = 0
    if playerx > graphwidth - playerw then playerx = graphwidth - playerw
    pause .02
end while

print "Your score: " + score
print "Game over."
end
```

Program 108 · Big Program — Space Warp Game

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout



PRACTICE

Exercises

WORD SEARCH

b o r d d t d n o b b y u
x e d n i l d o j k p r w
g y s d m e g i m c w o a
a c i v e h c t p o f m g
b t z a n o m c n l n e s
a n e y s r i e y u s m n
r e o w i e d l r m w m i
r m f k o p e l l n o k m
a e t i n e r o i t r m n
y l s r o u h c s a g e j
r e c c l l i f t b k n n
u k l c m i o m k l c n y
i l m g d f e r g e v i w

Find: array, collection, column, dimension, element, fill, index, list, memory, redim, ref, row, size, table



Problems

Try these — they only use what this chapter and the earlier ones covered.

1 Ask the user how many numbers they want to add together. Make an array of that size, ask for each number with **inputfloat** and store it in the array. When all the numbers are in, loop through the array and print the total.

2 Add to Problem 1: after the total, print the average.

3 Add to Problem 1: print the smallest and the largest number. To find the smallest, copy the first element into a variable. Then compare every other element with it: whenever an element is smaller, copy that one into the variable instead.

4 Turn your answers to Problems 2 and 3 into three functions, *average*, *smallest* and *largest*, that each take an array and return a number. Use `[?]` so that they work with an array of any size. Do they need **ref**?

© 2019 James M. Reneau · CC BY-NC-SA 3.0 US · adapted layout

5 Ask for a list of numbers, as in Problem 1. Then print a table of every number multiplied by every other number. Hint: you need a loop inside another loop, and the columns don't have to line up.

TEXT OUTPUT

```
How many numbers? 5
Number 1: 4
Number 2: 7
Number 3: 9
Number 4: 12
Number 5: 45
16 28 36 48 180
28 49 63 84 315
36 63 81 108 405
48 84 108 144 540
180 315 405 540 2025
```

6 Change Program 104 so that it also prints the average of each test: the average of each **column** instead of each row.

Chapter 15 · Arrays — Collections of Information. Modernised layout of the BASIC-256 beginner's guide, updated for BASIC-256 2.3. Programs are numbered 97–108, continuing from Chapter 14; this chapter includes the array sections that the original printed at the start of the Mathematics chapter. Corrected for 2.3: `dim` and `redim` give every new element 0 (or "" for a $ name) instead of leaving it empty; an array can be passed to a subroutine without `ref()` (the subroutine gets a copy) and `ref()` is needed only to change it, so the new Program 106 shows that; the `[]` after an array given to `sound`, `stamp` or `poly` may be left out; the Space Chirp rises 80 Hz a note, not 40; the list of friends started at line 15 and the text pointed to the wrong program numbers; the X comes from Chapter 11; `sound` lists from Chapter 4. The many-ball program no longer draws a green square that `clg` erased at once, bounces with the radius and uses `+=`; the grade calculator writes its table as a list of lists; the random polygon and the shadow stamp are drawn for the 500 × 500 graphics output. New: `cls` explained, pictures of arrays and tables, a note on what comes later (`arrayadd`, `sort`, `mat` and the vector operators), and Problem 6. The Big Program uses `keypressed`, `clg`, a score on the screen and a smooth 50 rounds a second, and checks a hit with the ship's width plus the radius. Every program was run in BASIC-256 2.3. Sample outputs of programs that ask for input were made with copies that print the typed answers (shown in orange); the Space Warp picture and score come from a copy steered by the program itself.
Original text © 2019 James M. Reneau, licensed under CC BY-NC-SA 3.0 US. This adaptation is shared under the same licence.
Mascot: BitBot.