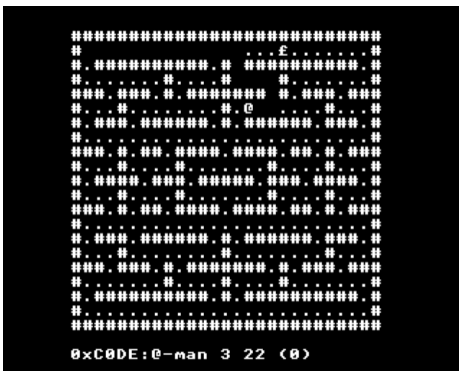
BASIC 10 LINER CONTEST 2020



Title: @-man

Author: 0xC0DE (Twitter @0xC0DE6502)

Category: PUR-120

Platform: Acorn Electron and BBC Micro

Language: BBC BASIC 2

Instructions

Move over Pac-man and make room for @-man! Guide him through the ASCII maze and eat all the dots while avoiding the baddies. In this game you don't chase the money (£), the money chases you! Move left/right with O/P and up/down with Q/A. Every dot scores 1 point. You start with 3 precious lives and the game ends (restarts) when you have none left. You lose a life when one of the baddies catches you. But… you gain a life when you move to the next screen after eating all the dots in the maze. The game will remember your highest score until you exit the program. Every new screen will add one more baddie, up to a maximum of 4, making it more difficult to eat all those delicious dots. Be careful out there!

Playing @-man

Of course, @-man should be played on a real Acorn Electron or BBC Micro ☺. But it can be played in several emulators, for instance: Electrem, BeebEm or even online in jsbeeb.

Simply copy and paste the BASIC program into one of the emulators. Or load the disk image into one of the emulators and press SHIFT+BREAK to execute @-man. Have fun!

Program

```
0MO.6:V.28,6,24,32,2:DIMF(26,20),C 4,G 20:H=0:S=0:F$="906904119011271441311153182131612904311121411314141141313 0":V=0
1!C=&232E20:i=0:v=1:F.y=0TO10:F.x=0TO13:i=i-(v=1):v=v-1-(v=1)*VA.MI.F$,i,1):u=v=0:IFV=0L=3:K=1:H=-(S>H)*S-(S<=H)*H:S=0
2c=1+(1A.i):s=26-x:t=20-y:F(x,y)=c:F(s,y)=c:F(x,t)=c:F(s,t)=c:x=x+u:v=v-u:N.,:CLS:F.y=0TO20:F.x=0TO26:V.C?F(x,y):N.,:P=0
3F(1,1)=0:V.23,1,0;0;0;0;:X=1:Y=1:G!4=&1301:G!8=&119:G!12=&1319:G!16=&707:IFL=0SO.1,-10,150,5:SO.1,-10,99,5:V=0:G.1
4P.TAB(0,22)"0xC0DE:@-man ";L;" ";S;" (";H;")";:D=INKEY-55-INKEY-56:E=(D=0)*(INKEY-66-INKEY-17):X=X-D*(F(X+D,Y)<2)
5V=31:Y=Y-E*(F(X,Y+E)<2):V.V,X,Y,64:s=F(X,Y):SO.1,-10,128,s:S=S+s:P=P+s:IFP=281K=K+1+(K=4):L=L+1:SO.1,-10,200,10:G.1
6F(X,Y)=0:F.i=1TOK:A=G+4*i:g=?A:h=A?1:q=A?2:r=A?3:?G=(F(g,h-1)<2)*((r=255)+(r=0)):IF(D ORE)V.V,X-D,Y-E,C?F(X-D,Y-E)
7G?1=(F(g+1,h)<2)*((q=1)+(q=0)):G?2=(F(g,h+1)<2)*((r=1)+(r=0)):G?3=(F(g-1,h)<2)*((q=255)+(q=0)):IFg=X A.h=Y:G.9
8REP.I=RND(4)-1:U.G?I:V.V,g,h,C?F(g,h):A?2=(1A.I)+2*(I=3):g=g+A?2:A?3=(1A.(I+1))+2*(I=0):h=h+A?3:V.V,g,h,96:?A=g:A?1=h
9IFg=X A.h=Y i=K:N.:L=L-1:SO.0,-10,6,10:V.V,X,Y,C?F(X,Y):F.i=1TOK:A=G+4*i:V.V,?A,A?1,C?F(?A,A?1):N.:G.3EL.N.:G.4
```

Explanation

```
0MO.6:V.28,6,24,32,2:DIMF(26,20),C 4,G 20:H=0:S=0:F$="906904119011271441311153182131612904311121411314141141313 0":V=0
```

Switch to graphics MODE 6 (40x25) and define a text window (VDU 28) that displays the maze. The two-dimensional array F contains the maze layout of 27x21 characters. A maze location (i.e. value in array F) can be 2 for a wall, 1 for an edible dot, or 0 for a passageway. The printable symbols representing these values ('#' for wall, '.' for dot, and ' ' for passage-way) are stored in C for easy access (see line 1). The initial maze itself is encoded in string F$. High score (H) and current score (S) are set to zero. G contains movement information of the maximum of 4 baddies chasing @-man through the maze. V will be 0 to indicate a completely new game, or 31 to indicate a new level.

```
1!C=&232E20:i=0:v=1:F.y=0TO10:F.x=0TO13:i=i-(v=1):v=v-1-(v=1)*VA.MI.F$,i,1):u=v=0:IFV=0L=3:K=1:H=-(S>H)*S-(S<=H)*H:S=0
```

Start a new game (if V=0) or a new level (if V=31). Initialize C with the 3 symbols used in the maze: &20=' ', &2E='.', &23='#'. The maze is vertically and horizontally symmetrical. That's why we decode only the topleft quadrant of 14x11 characters from F$, which is then copied/mirrored to the other 3 quadrants. The numbers in F$ encode a stretch of wall (value 2), then a stretch of dots (value 1), then wall, and so on until the topleft quadrant is filled. When a new game is started (V=0) then the number of lives (L) is reset to 3, the number of baddies (K) is reset to 1, the current score (S) is reset to 0, and the current highscore (H) is remembered.

```
2c=1+(1A.i):s=26-x:t=20-y:F(x,y)=c:F(s,y)=c:F(x,t)=c:F(s,t)=c:x=x+u:v=v-u:N.,:CLS:F.y=0TO20:F.x=0TO26:V.C?F(x,y):N.,:P=0
```

This is where the 4 quadrants of the maze (F) are filled with value 2 for wall and value 1 for dot by copying and mirroring the values that are written to the topleft quadrant. The text window is cleared (CLS) and the entire 27x21 maze is drawn. Finally, the current number of dots (P) eaten by @-man is set to zero.

```
3F(1,1)=0:V.23,1,0;0;0;0;:X=1:Y=1:G!4=&1301:G!8=&119:G!12=&1319:G!16=&707:IFL=0SO.1,-10,150,5:SO.1,-10,99,5:V=0:G.1
```

Put @-man and the baddies chasing him at their initial positions in the maze. This also happens when you lose a life. @-man starts at location (X,Y)=(1,1) in the maze. F(1,1) is also set to zero to remove the dot that was placed there during maze initialization. The text cursor is switched off for cosmetic reasons. The initial positions of the maximum of 4 baddies are set to (1,19), (25,1), (25,19) and (7,7) respectively. There is a final check to see if no more lives are left (L=0). If that is the case we play a sad sound and start the game all over (V=0) from line 1.

```
4P.TAB(0,22)"0xC0DE:@-man ";L;" ";S;" (";H;")";:D=INKEY-55-INKEY-56:E=(D=0)*(INKEY-66-INKEY-17):X=X-D*(F(X+D,Y)<2)
```

Print a status line right under the maze, with the name of the game, number of lives (L) left, current score (S) and current highscore (H). Detect keypresses for keys O/P/Q/A and set D and E accordingly. D will be zero if neither O or P are pressed, or if both are pressed at the same time. D will be -1 if only O (left) is pressed. And 1 if only P (right) is pressed. E will be zero if neither Q or A are pressed, or if both are pressed at the same time, or if D<>0 (i.e. O or P was already pressed). Otherwise, E will be -1 if only Q (up) is pressed, or 1 if only A (down) is pressed. Update horizontal position (X) of @-man only if there is no wall in the direction determined by D.

```
5V=31:Y=Y-E*(F(X,Y+E)<2):V.V,X,Y,64:s=F(X,Y):SO.1,-10,128,s:S=S+s:P=P+s:IFP=281K=K+1+(K=4):L=L+1:SO.1,-10,200,10:G.1
```

V is set to 31 to indicate that we started a new level. The value of 31 is not arbitrary because it will be used in subsequent VDU 31 (PRINT TAB) commands. Update vertical position (Y) of @-man only if there is no wall in the direction determined by E. Print @-man ('@') at position (X,Y). Play a short sound if that location still contained a dot in the maze. If a dot was eaten, increment current score (S) and number of dots eaten (P) by 1. Check if the level is finished, i.e. all 281 dots have been eaten. If that is the case, add another baddie (K) up to a maximum of 4 and add an extra life (L). Play a victorious sound and start a new level (V=31) from line 1.

```
6F(X,Y)=0:F.i=1TOK:A=G+4*i:g=?A:h=A?1:q=A?2:r=A?3:?G=(F(g,h-1)<2)*((r=255)+(r=0)):IF(D ORE)V.V,X-D,Y-E,C?F(X-D,Y-E)
```

Mark the current location of @-man as empty (0) in the maze (the dot was eaten, if any). The maze location where @-man was previously located is erased, but only if @-man actually moved (otherwise it would cause unnecessary redrawing/flickering). It is now time to update the position of 1 to 4 baddies. They all move randomly through the maze. They never move back the way they came from and only make a (random) turn when they hit a wall or pass another passageway. The current baddie is located at (g,h) in the maze and wants to move in the direction determined by q and r. The only valid values for (q,r) are: (1,0) if the baddie is moving right, (255,0) if it's moving left, (0,1) if it's moving down, or (0,255) if it's moving up.

```
7G?1=(F(g+1,h)<2)*((q=1)+(q=0)):G?2=(F(g,h+1)<2)*((r=1)+(r=0)):G?3=(F(g-1,h)<2)*((q=255)+(q=0)):IFg=X A.h=Y:G.9
```

A baddie chasing @-man can move in 4 possible directions: left, right, up or down. We calculate a 'chance' for each of these four directions and store them at the start of memory block G. The 'chance' will be zero if that direction is blocked by a wall. The 'chance' for direction up is calculated in line 6. The 'chances' for the other 3 directions are calculated in line 7. Finally, a check is done if the current baddie has caught @-man (i.e. its coordinates match @-man's location). In that case we continue from line 9 (losing a life).

```
8REP.I=RND(4)-1:U.G?I:V.V,g,h,C?F(g,h):A?2=(1A.I)+2*(I=3):g=g+A?2:A?3=(1A.(I+1))+2*(I=0):h=h+A?3:V.V,g,h,96:?A=g:A?1=h
```

We randomly select a non-zero 'chance' and that will be the direction the current baddie moves in. The current location of the baddie is erased by redrawing the maze symbol that was underneath it (a space or a dot). The new direction of movement and the new location of the baddie is stored back into the proper location of memory block G (pointed to here by A). And finally, the symbol for the baddie (£, character code 96) is drawn at the new location.

```
9IFg=X A.h=Y i=K:N.:L=L-1:SO.0,-10,6,10:V.V,X,Y,C?F(X,Y):F.i=1TOK:A=G+4*i:V.V,?A,A?1,C?F(?A,A?1):N.:G.3EL.N.:G.4
```

Again, check if the current baddie has caught poor @-man. If that happens, play a sad sound, decrement the number of lives (L), erase @-man and all 4 possible baddies, before jumping to line 3 (which then resets the location of @-man and the baddies in the maze). If the current baddie didn't catch @-man, we continue updating and checking the next baddie (up to 4 baddies). After that, continue from line 4 (printing the current score, checking for key presses, etc).