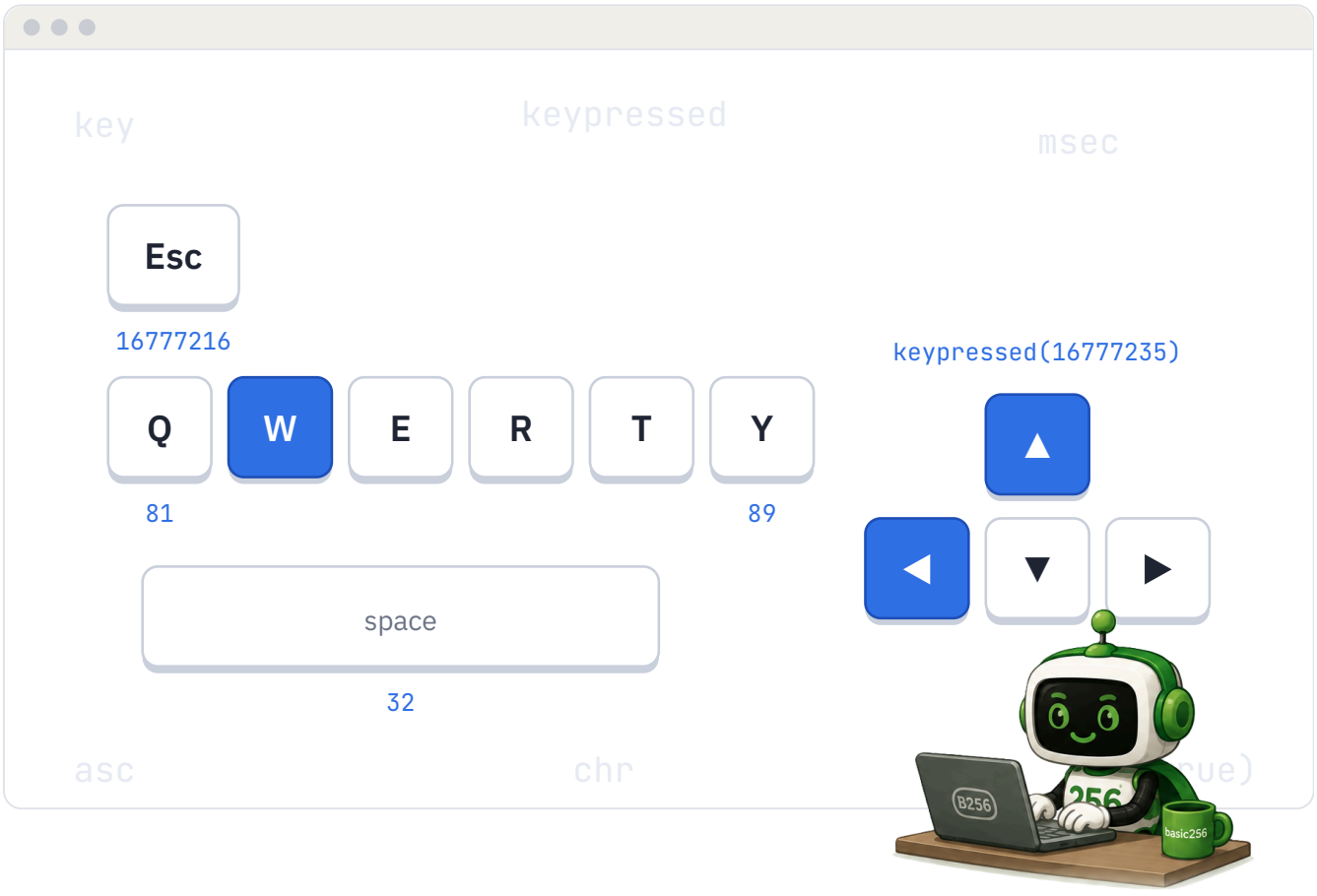
LEARN TO CODE · THE KEYBOARD

Chapter 12

Keyboard Control — Using the Keyboard to Do Things

With **input** your program waits until the user has typed a whole answer and pressed Enter. Games can't wait like that. In this chapter your programs react to single keys the moment they are pressed: letters, arrows and special keys. You learn two ways to read the keyboard — **key**, which gives the last key pressed, and **keypressed**, which tells which keys are held down right now — and you build a game of falling letters.



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

SECTION 1

Getting the last key press

The `key` function tells which key was pressed last.

Every key on the keyboard has a number, its **key code**. The **key** function returns the code of the last key that was pressed. Once your program has read it, **key** returns 0 until another key is pressed. A letter key always gives the code of the capital letter, whether Shift or Caps Lock is on or not: *a* and *A* both give 65.

Program 79 shows the code of every key you press, until you press Q.

readkey.kbs BASIC-256

```
# readkey.kbs - show the code of every key you press

print "Press keys - Q to quit"
do
   k = key
   if k <> 0 then
      # keys that type a character: below 16777216
      if k < 16777216 then print chr(k) + "=";
      print k
   end if
until k = asc("Q")
end
```

Program 79 · Read Keyboard

TEXT OUTPUT

```
Press keys - Q to quit
A=65
Z=90
M=77
16777248
 =32
16777235
16777216
16777220
16777248
Q=81
```

Pressed: A, Z, M, Shift, Space, ↑, Esc, Enter, and Shift + Q.

Keys that type a character have a code below 16,777,216, and for those the program also prints the character with **chr**. Keys that don't type anything — Shift, the arrows, Esc, Enter — have big codes from 16,777,216 up, so only their number is printed. Space types a space, so its line starts with an empty space.

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

NEW CONCEPT

```
key  or  key()
key(true)
```

key returns the code of the last key pressed, and sets it back to 0, so each key press is returned only once. If no key was pressed since the last time, it returns 0. Hold a key down and, after a moment, it repeats, just as it does when you type.



KEY	CODE	
A … Z	65 … 90	the letter keys, with or without Shift or Caps Lock
0 … 9	48 … 57	the digit keys, if they type a digit without Shift
Space	32	
Esc	16777216	
Tab	16777217	
Backspace	16777219	
Enter	16777220	the big Enter key (the one on the number pad is 16777221)
Shift	16777248	
Ctrl	16777249	
← ↑ → ↓	16777234 … 16777237	left 16777234, up 16777235, right 16777236, down 16777237

For the codes of all the other keys, see the list of key codes in the documentation of Qt, the toolkit BASIC-256 is built with: doc.qt.io/qt-6/qt.html#Key-enum.

With **key(true)** you get the Unicode number of the character the key types instead: 97 for *a*, 65 for *A* (with Shift), 13 for Enter, 27 for Esc and 0 for keys that type nothing, like the arrows or Shift.

Read key once, into a variable. `if key = 81 then print key` doesn't work: the first **key** uses up the key press, so the second one returns 0. Write `k = key` and then test and print *k*, as Program 79 does.
Your keyboard layout counts. A key gives the code of what it types. On a French keyboard the key with 7 on it types è without Shift, so **key** returns 200 (the code of È), and 55 only with Shift. The letters and the special keys work the same everywhere.
Click the output. Keys reach your program only when the text or graphics output has the focus. That is so when the program starts, but if you click in the program editor, click on the output again. A few keys, such as F1, are used by BASIC-256 itself and never reach your program.

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



NEW CONCEPT

Unicode is a standard that gives a number to every letter and sign of the world's writing systems: Latin, Greek, Cyrillic, Arabic, Chinese, emoji and many more — over 150,000 characters. The first 128 numbers are the same as in the older ASCII code: 65 is *A*, 97 is *a*, 48 is the digit 0. See www.unicode.org.



NEW CONCEPT

```
asc(string)
```

asc returns the Unicode number of the first character of a string: `asc("Q")` is 81. For an empty string it returns 0.



NEW CONCEPT

```
chr(number)
```

chr does the opposite: it returns a string of one character, the character with that Unicode number. `chr(81)` is "Q".

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

SECTION 2

Timing and moving with keys

How fast can you find a key? And how do you steer something around the screen?

Program 80 shows a random letter and times how long it takes you to press it. It uses the **msec** function, which tells how many milliseconds (thousandths of a second) the program has been running.

keymsec.kbs BASIC-256

```
# keymsec.kbs - keyboard speed drill

# get the code for a random letter from A to Z
c = asc("A") + int(rand * 26)

# show the letter (from its code)
print "Press '" + chr(c) + "'"

time = msec        # the start time
do                 # wait for the right key
   k = key
until k = c
time = msec - time # how long it took (in ms)

print "It took you " + (time / 1000) + " seconds."
```

Program 80 · Keyboard Speed Drill

TEXT OUTPUT

```
Press 'A'
It took you 5.53 seconds.
```

Two wrong keys, S and M, were pressed first: they were ignored.

The program reads **msec** before and after waiting, and the difference is the time it took. Dividing by 1000 turns milliseconds into seconds. The **do** ... **until** loop reads a key over and over and only stops at the right one: other keys, and the 0 that **key** returns while nothing is pressed, just go round the loop again. You may press the letter with or without Shift.

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



NEW CONCEPT

```
msec  or  msec()
```

msec returns how long the program has been running, in milliseconds: 1000 milliseconds are one second. To time something, store **msec** at the start and subtract it from **msec** at the end.



TRY IT

Count the wrong keys too: add 1 to a counter every time *k* is not 0 and not *c*, and print the count at the end.

Program 81 draws a red ball that you steer with four letter keys: I, J, K and M. On the keyboard they lie in the shape of the arrows: I above, J to the left, K to the right and M below.

Each key moves the ball one radius, 20 pixels. When the ball would go off one edge, it comes back at the opposite edge. The subroutine **drawball** clears the screen and draws the ball again with three lighter circles as a shine, then shows it all with **refresh**.

GRAPHICS OUTPUT



After 4 × K, 4 × I and 1 × J.

keymoveball.kbs — BASIC-256

```
# keymoveball.kbs - move a ball with the keyboard

print "Use i for up, j for left, k for right, m for down,"
print "and q to quit."

fastgraphics

# position of the ball: start in the middle
x = graphwidth / 2
y = graphheight / 2
r = 20      # size of the ball (radius)
```

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

```
# draw the ball for the first time
call drawball(x, y, r)

# loop and wait for the user to press a key
while true
   k = key
   if k = asc("I") then
      y = y - r
      if y < r then y = graphheight - r
      call drawball(x, y, r)
   end if
   if k = asc("J") then
      x = x - r
      if x < r then x = graphwidth - r
      call drawball(x, y, r)
   end if
   if k = asc("K") then
      x = x + r
      if x > graphwidth - r then x = r
      call drawball(x, y, r)
   end if
   if k = asc("M") then
      y = y + r
      if y > graphheight - r then y = r
      call drawball(x, y, r)
   end if
   if k = asc("Q") then exit while
end while
print "All done."
end

subroutine drawball(bx, by, br)
   clg white
   color red
   circle bx, by, br
   # a shine: smaller, lighter circles up and to the left
   color rgb(255, 100, 100)
   circle bx - br / 4, by - br / 4, br / 2
   color rgb(255, 150, 150)
   circle bx - br / 4, by - br / 4, br * 0.3
   color rgb(255, 200, 200)
   circle bx - br / 4, by - br / 4, br * 0.1
   refresh
end subroutine
```

Program 81 · Move Ball

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

TEXT OUTPUT

```
Use i for up, j for left, k for right, m for down,
and q to quit.
All done.
```



TRY IT

Steer the ball with the arrow keys instead of I, J, K and M. Use the codes from the table in Section 1, and keep Q to quit.

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

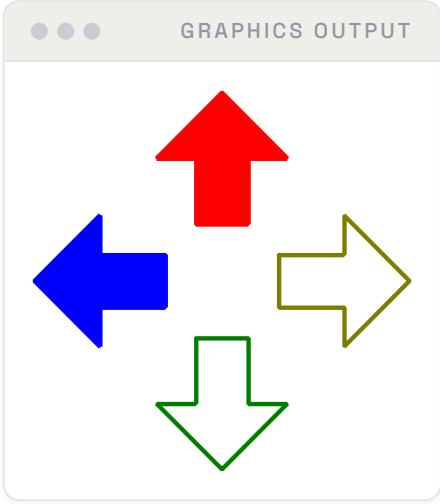
SECTION 3

Getting the keys that are held down

The `keypressed` function tells which keys are being pressed right now.

key returns the last key pressed, even if it has been let go long ago, and only one key at a time. For games you often want to know which keys are being held down *right now* — perhaps several at once. That is what **keypressed** is for. Program 82 shows four arrows on the graphics output, one for each arrow key. While you hold an arrow key, its arrow is filled in.

The subroutine **arrow** draws one arrow with **stamp** (Chapter 9): the shape points up, and the angle turns it. With `color darkred, white` (Chapter 2) the arrow is drawn as a dark outline, filled with white; with `color red` it is filled with red. Because the loop draws all four arrows over and over, each one changes as soon as you press or let go of its key.



While ↑ and ← are held down.

keypressarrows.kbs BASIC-256

```
# keypressarrows.kbs - light up the arrows you hold

# key codes
ar_up = 16777235
ar_down = 16777237
ar_left = 16777234
ar_right = 16777236
space = 32

clg white
penwidth 5

print "Hold arrow keys (even more than one)."
print "Press the space bar to end."
while not keypressed(space)
   if keypressed(ar_up) then
      color red
   else
      color darkred, white
   end if
   call arrow(250, 110, 0)
```

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

```
    if keypressed(ar_down) then
       color green
    else
       color darkgreen, white
    end if
    call arrow(250, 390, pi)

    if keypressed(ar_left) then
       color blue
    else
       color darkblue, white
    end if
    call arrow(110, 250, 1.5 * pi)

    if keypressed(ar_right) then
       color yellow
    else
       color darkyellow, white
    end if
    call arrow(390, 250, 0.5 * pi)
end while
print "All done."
end

subroutine arrow(x, y, angle)
    # an arrow pointing up, turned by angle
    stamp x, y, 15, angle, {0, -5, 5, 0, 2, 0, 2, 5,
       -2, 5, -2, 0, -5, 0}
end subroutine
```

Program 82 · Keys Pressed

TEXT OUTPUT

```
Hold arrow keys (even more than one).
Press the space bar to end.
All done.
```

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



NEW CONCEPT

```
keypressed(key code)
keypressed  or  keypressed()
```

keypressed with a key code returns true while that key is held down, and false when it isn't. Without a code it returns how many keys are held down at the moment. Unlike **key**, it doesn't clear anything, so you can ask about as many keys as you like, as often as you like. The key codes are the same as for **key**.



KEY OR KEYPRESSED?

Use **key** when every key press counts once, as when typing an answer or choosing from a menu: no press gets lost, even if your program was busy. Use **keypressed** for movement that goes on while a key is held, or for keys used together, like ↑ and → to move diagonally. (Many keyboards can't report every combination of three or more keys held at once.)



TRY IT

Draw a circle in the middle of the graphics output that turns orange while Shift (16777248) is held down.

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



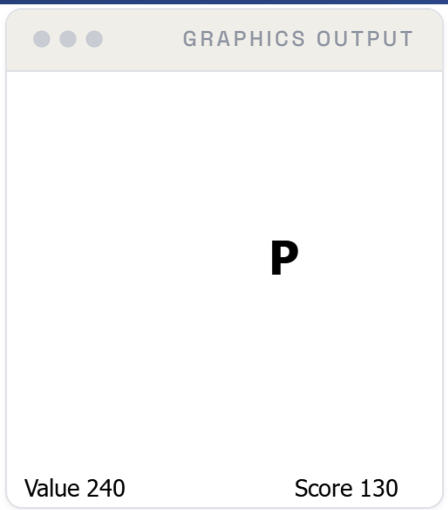
BIG PROGRAM

Falling Letter Game

Random letters fall down the screen, one at a time. Press the right key before the letter reaches the bottom: the sooner you press it, the more points you score. A wrong key costs as many points as the right key would have earned.

The game starts with a title screen and waits for any key. Then ten letters fall, 20 pixels at a time. Below them the program shows how many points the letter is worth now, and your score so far. When the last letter is done, it shows the final score and how many letters you missed.

GRAPHICS OUTPUT



Program 83 waits for the first key in two steps. `k = key` throws away any key pressed before the game started, and `while key = 0` waits for a new one (here **key** may be called in the condition, because the program doesn't need to know which key it was).

For every letter, `asc("A") + int(rand * 26)` picks a code from 65 to 90, and **chr** turns it into the letter to draw. Again `k = key` first forgets a key pressed too late for the letter before. Then the inner loop moves the letter down. The value is 420 – *y*, so it shrinks as the letter falls. A key press ends the letter at once with **exit for**: the right key adds the value to the score and takes one off *misses*, which started at 10; any other key subtracts the value.

fallinglettergame.kbs — BASIC-256

```
# fallinglettergame.kbs - catch the falling letters

speed = 0.15   # drop speed - lower to make faster
nletters = 10  # letters to play

score = 0
color black

fastgraphics

clg
font "Tahoma", 32, 700
text 40, 150, "Falling Letter Game"
font "Tahoma", 20, 400
text 40, 230, "Press any key to start"
refresh
# forget earlier keys and wait for any key
k = key
```

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

```
while key = 0
   pause speed
end while

misses = nletters   # assume they missed everything
for n = 1 to nletters
   letter = asc("A") + int(rand * 26)
   x = 20 + rand * 420
   k = key   # forget keys pressed for the last letter
   for y = 0 to 400 step 20
      clg
      # show the letter
      font "Tahoma", 40, 700
      text x, y, chr(letter)
      # show the value and the score
      font "Tahoma", 20, 400
      value = 420 - y
      text 20, 460, "Value " + value
      text 330, 460, "Score " + score
      refresh
      k = key
      if k <> 0 then
         if k = letter then
            score += value
            misses -= 1   # didn't miss this one
         else
            score -= value
         end if
         exit for
      end if
      pause speed
   next y
next n

clg
font "Tahoma", 32, 700
text 40, 100, "Falling Letter Game"
text 40, 180, "Game Over"
text 40, 260, "Score: " + score
text 40, 340, "Misses: " + misses
refresh
end
```

Program 83 · Big Program — Falling Letter Game



TRY IT

Make the game harder as it goes: after every letter, multiply *speed* by 0.9. Then play a high **sound** (Chapter 4) for a right key and a low one for a wrong key.

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



PRACTICE

Exercises

WORD SEARCH

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

Find: arrow, asc, capslock, chr, control, escape, key, keypressed, msec, shift, space, unicode



Problems

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

1 Change Program 80, the Keyboard Speed Drill, so that it shows ten letters, one at a time, and waits each time for the user to press that key. When all ten are done, show the total time.

As an extra challenge, count the errors: when the user presses a wrong key, print *Error* and let them try the same letter again until they get it right.

TEXT OUTPUT

```
Press 'A'
Press 'M'
Press 'O'
Error
Press 'U'
Press 'X'
Press 'V'
Press 'K'
Press 'C'
Press 'Z'
Press 'Z'
Time: 15.372 seconds
Errors: 1
```

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

2 Make a “whack-a-mole” game. Show the numbers 1 to 9 in grey and make a random one red. Wait a random time between 0.5 and 1.5 seconds (use **msec**) for the user to press that number. If they do, play a happy sound and show the next one; if they press a wrong key or are too slow, play a sad sound. After 5 misses, show how many they hit.



3 Make a piano out of your keyboard. Give each key of one row — for example A, S, D, F, G, H, J and K — one of the notes C D E F G A B C from the chart of notes and frequencies in Chapter 4. In a loop, read **key**, and play a short **sound** with the right frequency when one of those keys is pressed.

4 Use **keypressed** to move a ball smoothly: start from Program 81, and move the ball a few pixels every time round the loop for as long as an arrow key is held down. Hold two arrow keys to move diagonally.

Chapter 12 · Keyboard Control — Using the Keyboard to Do Things. Modernised layout of the BASIC-256 beginner’s guide, updated for BASIC-256 2.3. Programs are numbered 79–83, continuing from Chapter 11. Corrected for 2.3: the link to the list of key codes now goes to the Qt 6 documentation (the Qt 4.8 page no longer exists); `msec` is a function, not a statement; Unicode has over 150,000 characters now. New: `key(true)`, `keypressed` without a code, the notes on reading `key` once, keyboard layouts, focus and key repeat, and the “key or keypressed?” box; every key code in the table was checked in BASIC-256 2.3. Program 79 shows the character of every key that types one, not only codes 32–127. Program 82 draws its arrows with a subroutine instead of an array (arrays come in a later chapter), uses `end if` throughout with the indentation fixed, and says when it is done. The Big Program now forgets a key pressed too late for the previous letter (it counted as a wrong key for the next one) and uses `-= 1` instead of the untaught `--`. All programs are redrawn for the 500 × 500 graphics output with font weights for 2.3 (700 bold, 400 normal); the ball’s shine moved to the top left. Problem 3 points to the note chart in Chapter 4. Every program was run in BASIC-256 2.3 with real key presses. 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.