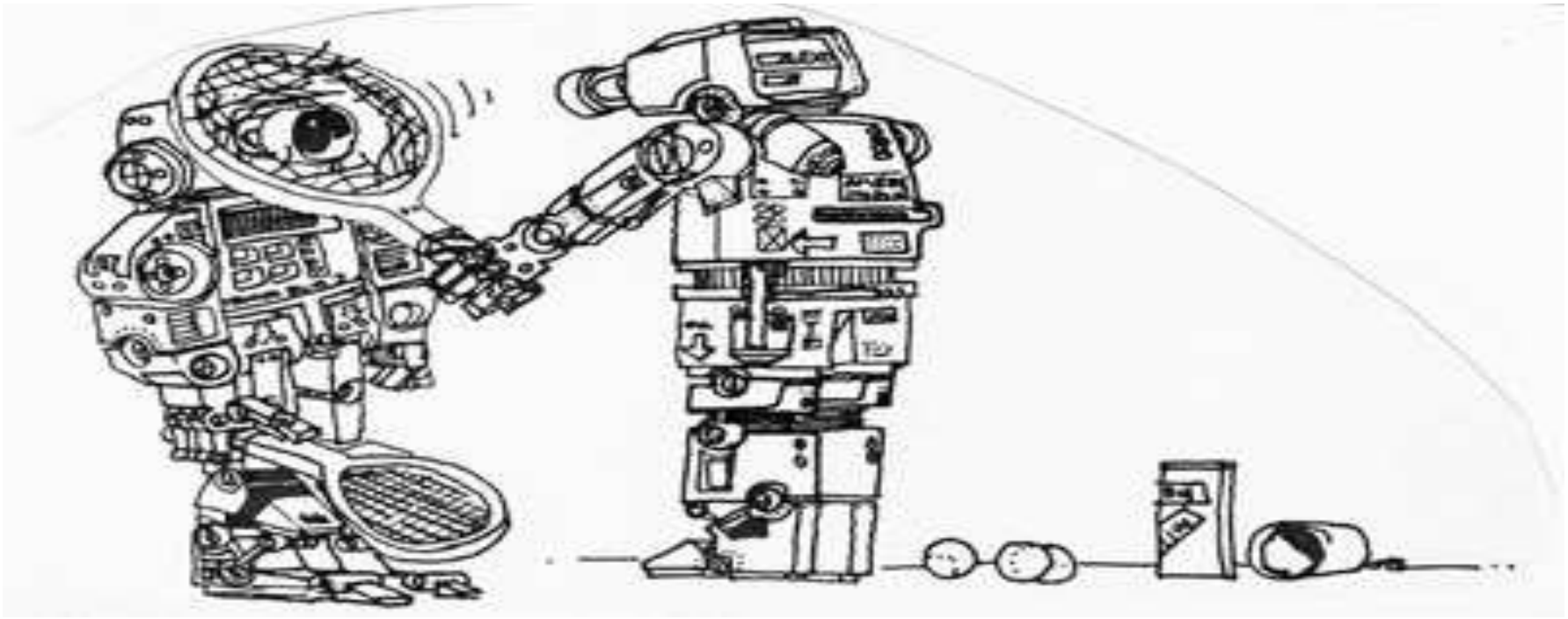


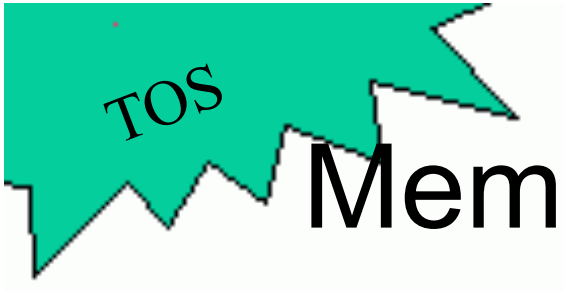
Memory Access

Objectives

- Learn how to read and write from memory using C-pointers
- Implement the very first TOS functions

Memory Access

- Memory often needs to be manipulated manually.
- Two operations:
  - Peek: peek (i.e. read) inside the memory
  - Poke: poke (i.e. write) to the memory
- Warning: uncontrolled poking results in disaster!
- We can use C-pointers to peek and poke:
  - peek:
```
char* ptr = (char*) 0xB8000;
char ch = *ptr;
```
  - poke:
```
char ch = 'A';
char *ptr = (char*) 0xB8000;
*ptr = ch;
```

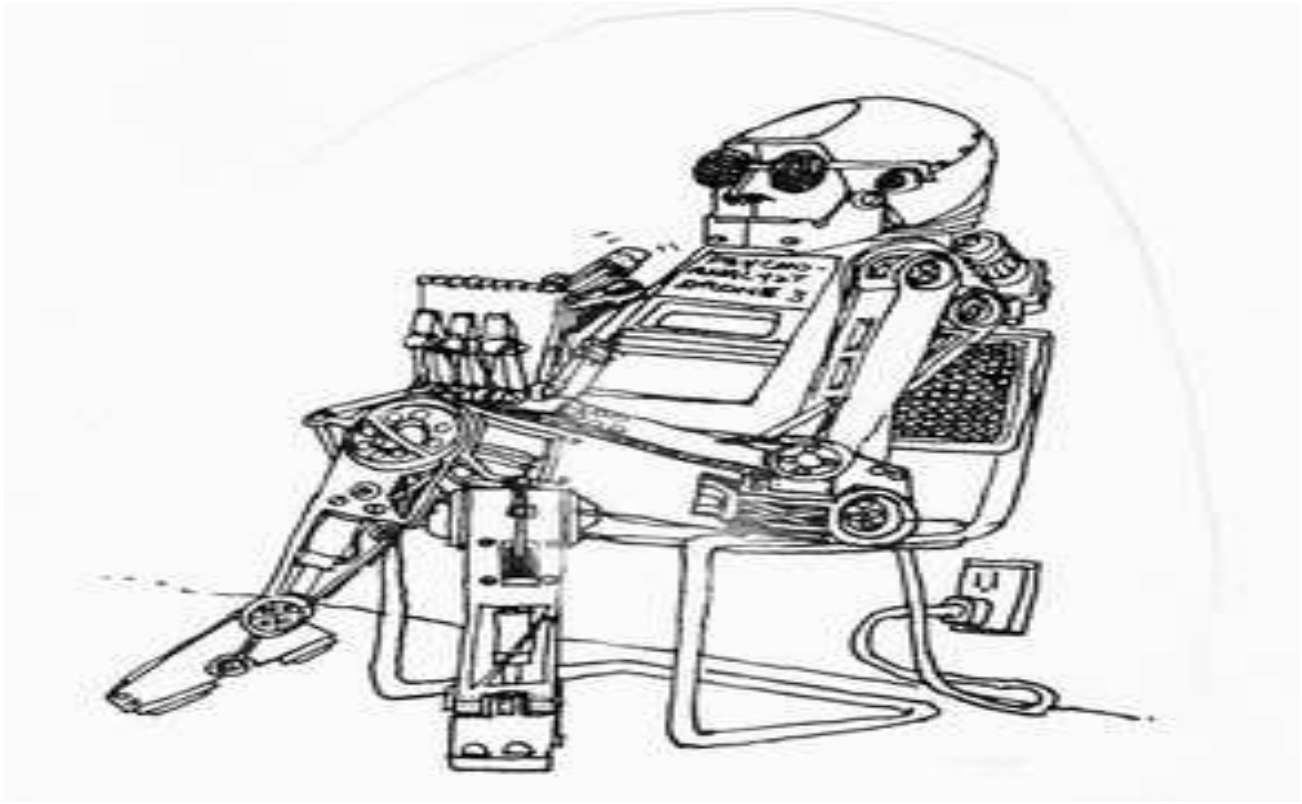


Memory functions in TOS

- Memory functions are implemented in file `tos/kernel/mem.c`
- Peeking and poking is offered for different data types: (see `typedef` in `~/tos/include/kernel.h`)
  - BYTE: 1 byte
  - WORD: 2 bytes
  - LONG: 4 bytes
- Memory address is represented through C-type MEM_ADDR;
- Functions:
  - `void poke_b (MEM_ADDR addr, BYTE value);`
  - `void poke_w (MEM_ADDR addr, WORD value);`
  - `void poke_l (MEM_ADDR addr, LONG value);`
  - `BYTE peek_b (MEM_ADDR addr);`
  - `WORD peek_w (MEM_ADDR addr);`
  - `LONG peek_l (MEM_ADDR addr);`

Assignment 2 (Part 1)

- Implement the functions located in `~/tos/kernel/mem.c`:
  - `peek_b(), peek_w(), peek_l()`
  - `poke_b(), poke_w(), poke_l()`
- Test case:
  - `test_mem_1`



Video

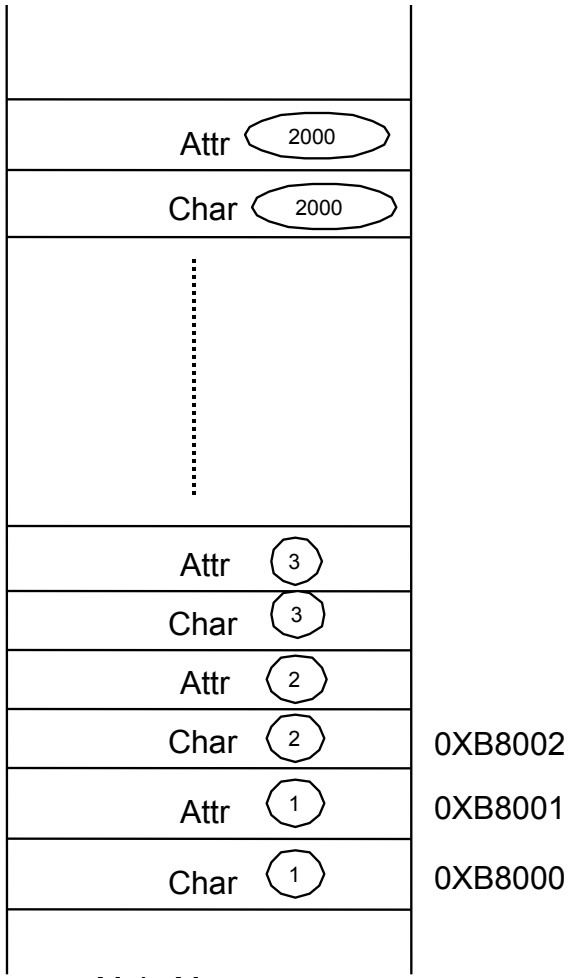
Video

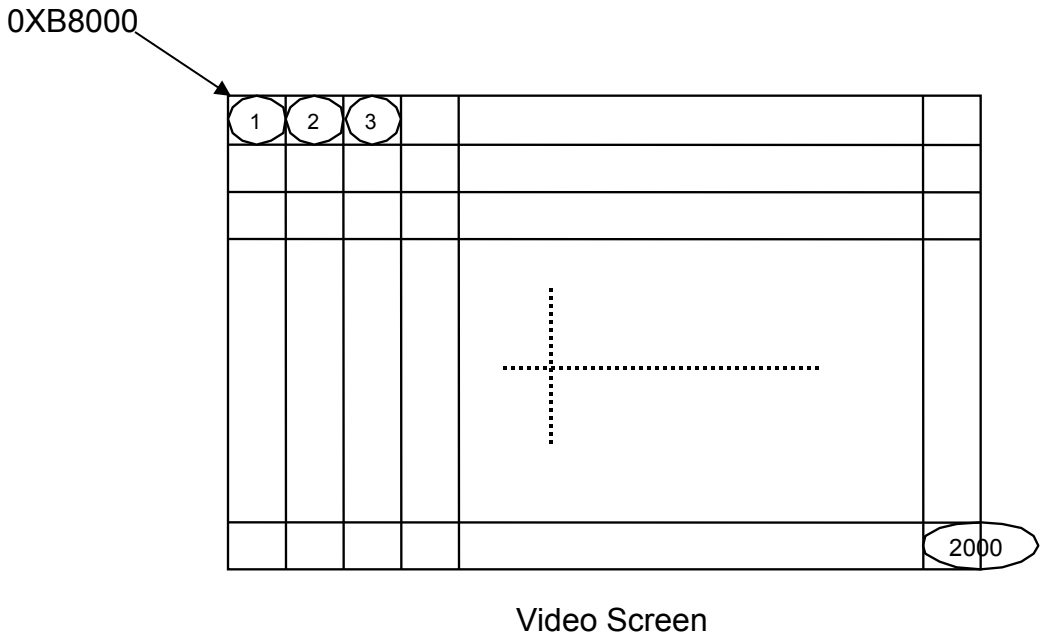
- The video can operate in several different modes:
  - Text mode: initial mode when the PC is turned on.
  - Graphical mode: high resolution mode that is typically activated by a windowing system (not used in TOS).
- Text mode:
  - 25 rows (numbered 0-24)
  - 80 columns (numbered 0-79)
  - top left corner of screen: row 0, column 0
  - lower right corner of screen: row 24, column 79
  - screen can display 2000 (25 X 80) characters

Outputting text

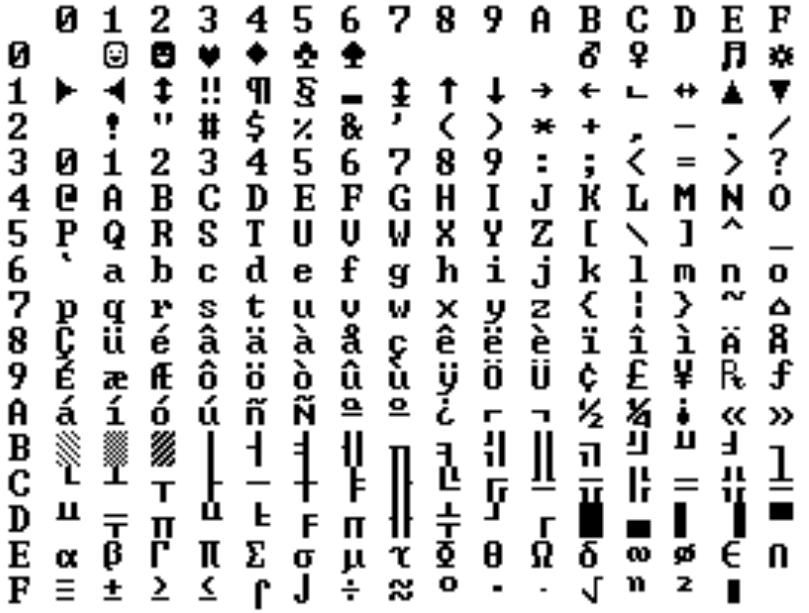
- Video is memory mapped. The area of memory that is provided for the video is called *Video Display Area*.
- The base address of the Video Display Area is 0xB8000.
- In text mode there are 25 X 80 = 2000 visible characters.
- Each visible character is represented by two bytes in the Video Display Area:
  - The character (e.g., ‘A’, ‘Z’, ‘0’. …)
  - The character attributes (e.g. color, intensity)
- The size of the Video Display Area is therefore:
  2 X 2000 = 4000 bytes.
- A character is represented by its ASCII value:
  ‘A’ = 0x41 = 65
  ‘0’ = 0x30 = 48

Layout of Video Display Area





PC Display Characters



- The above table shows which characters will be visible if poked to the video display area.
- 0x41 == 65_{10} == ‘A’
- 0x61 == 97_{10} == ‘a’

Character Attributes

	Background				Foreground			
Attribute:	BL	R	G	B	I	R	G	B
Bit number:	7	6	5	4	3	2	1	0

COLOR	I	R	G	B	HEX	COLOR	I	R	G	B	HEX
Black	0	0	0	0	0	Gray	1	0	0	0	8
Blue	0	0	0	1	1	Light blue	1	0	0	1	9
Gree	0	0	1	0	2	Light green	1	0	1	0	A
Cyan	0	0	1	1	3	Light cyan	1	0	1	1	B
Red	0	1	0	0	4	Light red	1	1	0	0	C
Magenta	0	1	0	1	5	Light magenta	1	1	0	1	D
Brown	0	1	1	0	6	Yellow	1	1	1	0	E
White	0	1	1	1	7	Bright white	1	1	1	1	F

Character Attribute Examples

BACK-GROUND	FORE-GROUND	BACKGROUND				FOREGROUND				
		BL	R	G	B	I	R	G	B	HEX
Black	Blue	0	0	0	0	0	0	0	1	01
Black	Red	0	0	0	1	0	1	0	0	14
Green	Cyan	0	0	1	0	0	0	1	1	23
White	Light magenta	0	1	1	1	1	1	0	1	7D
Green	Gray (blinking)	1	0	1	0	1	0	0	0	A8

In TOS we will just simply use bright white (0x0F) as the only color for all output.

Programming the Video Display

- Since the video display area is memory mapped, output is created by changing the contents of the memory.
- We can use the poke functions to accomplish this.
- E.g. output a white ‘A’ in the top left corner of the screen:

```
poke_b(0xB8000, ‘A’);
poke_b(0xB8001, 0x0F);
```

or:

```
poke_w(0xB8000, ‘A’ | 0x0F00);
```

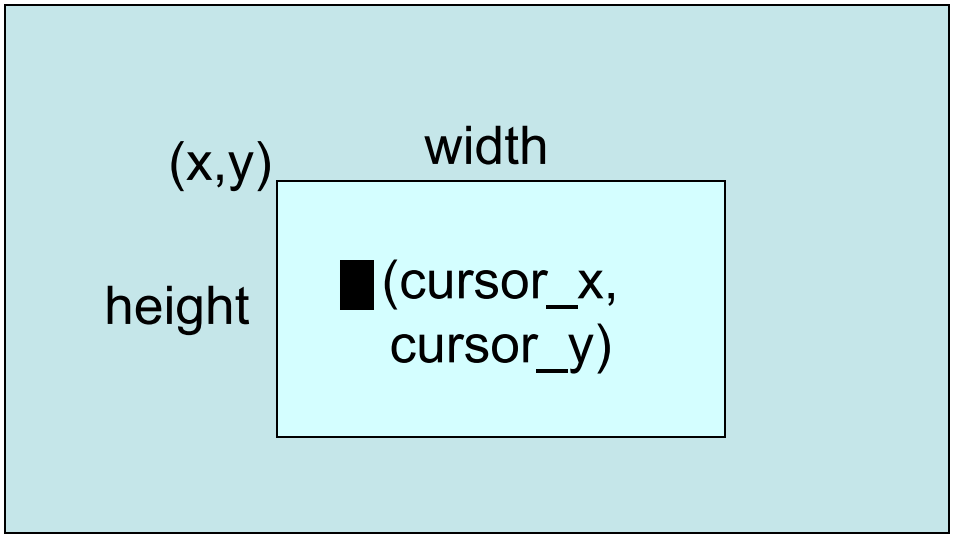
Windowing System

- We will implement a small text-based windowing system in TOS.
- The purpose is to allow each process to generate output in its own window.
- TOS' windowing system is based on the text-mode (I.e., no graphics, no mouse!)
- A window is simply a rectangular region of the video display area.
- We will not get fancy with this:
  - Windows are not allowed to overlap.
  - No need to draw borders around the windows.

Definition of a Window

- **`(x, y)`**: the top left corner of the window.
- **`(width, height)`**: size of the window.
- **`(cursor_x, cursor_y)`**: the location of the cursor relative to the top-left corner of the window.
- **`cursor_char`**: character used as the cursor.

```
// kernel.h
typedef struct {
  int  x, y;
  int  width, height;
  int  cursor_x, cursor_y;
  char cursor_char;
} WINDOW;
```



Video Display Area

Invariants

- The following conditions must be true for all windows:
  - $0 \leq$ `x` $<$ 80
  - $0 \leq$ `y` $<$ 25
  - `x` + `width` $<$ 80
  - `y` + `height` $<$ 25
  - $0 \leq$ `cursor_x` $<$ `width`
  - $0 \leq$ `cursor_y` $<$ `height`
- Furthermore it is assumed that no two windows overlap.

Windowing functions in TOS

- Windowing functions are implemented in file `~/tos/kernel/window.c`
- Functions:
  - `clear_window(WINDOW* wnd)`
    Clear the window. Content of the window is erased and the cursor is placed at the top left corner of the window.
  - `move_cursor(WINDOW* wnd, int x, int y)`
    The position of the cursor is set to be (`x, y`). Note that this position has to be within the boundaries of the window. The position is relative to the top-left corner of the window.
  - `show_cursor(WINDOW* wnd)`
    Shows the cursor of the window by displaying `cursor_char` at the current position of the cursor location.
  - `remove_cursor(WINDOW* wnd)`
    Removes the cursor of the window by displaying a blank character at the current position of the cursor location.

Windowing functions in TOS

- Functions:
  - `output_char(WINDOW* wnd, unsigned char ch)`
    'ch' is displayed at the current cursor location of the window. The cursor is advanced to the next location.
  - `output_string(WINDOW* wnd, const char* str)`
    'str' is a string that is displayed in the window. The cursor is advanced accordingly.
- Notes:
  - If '\n' is printed, the cursor should advance to the beginning of the next line.
  - The cursor has to stay within the boundaries of the window. If the cursor reaches the right border of the window, it needs to be positions at the beginning of the next line. If the cursor is at the bottom of the window, the contents of the whole window has to scroll up one line (thereby erasing the first line of the window).

wprintf()

- `wprintf()` is TOS ‘es version of the familiar `printf()` function from the C-library.
- ‘w‘ stands for ‘Window‘ to make sure the C-library version is not used by accident.
- `wprintf()` prints its output to a TOS window.
- TOS comes with an implementation of `wprintf()`, located in `~/tos/kernel/window.c`
- `wprintf()` only works once the functions `output_char()`, and `output_string()` work!
- Prototype of `wprintf()`:

```
void wprintf(WINDOW* wnd,
             const char* fmt, ...);
```

Using wprintf()

- The following examples show the usage of `wprintf()`. Apart from the fact that the first parameter designates the window where the output will be made, its usage is identical to `printf()` from the C-standard library.
- `wprintf(wnd, "Hello World!\n");`
  Hello World!
- `wprint(wnd, "Sum of 3 and 4 = %d\n", 3 + 4);`
  Sum of 3 and 4 = 7
- `wprintf(wnd, "20 in hex is 0x%x\n", 20);`
  20 in hex is 0x14
- `wprintf(wnd, "20 in binary is 0b%b\n", 20);`
  20 in binary is 0b10100
- `wprintf(wnd, "A char (%c) and an integer (%d)\n", 'A', 20);`
  A char (A) and an integer (20)

kprintf()

- `~/tos/kernel/window.c` has the following definition of a window that covers the complete screen:

```
static WINDOW kernel_window_def =
                {0, 0, 80, 25, 0, 0, ' '};
WINDOW* kernel_window =
&kernel_window_def;
```

- `kprintf()` (kernel print) prints into this window.
- Prototype of `kprintf()`:

```
void kprintf(const char* fmt, ...);
```

- `kprintf(...)` is a short-form for `wprintf(kernel_window, ...)`

Assignment 2 (Part 2)

- Implement the functions located in `~/tos/kernel/window.c`:
  - `clear_window()`
  - `move_cursor()`
  - `show_cursor()`
  - `remove_cursor()`
  - `output_char()`
  - `output_string()`
- Test cases:
  - `test_window_1`
  - `test_window_2`
  - `test_window_3`
  - `test_window_4`

PacMan (1)

- The purpose of the PacMan application is a simple game as a showcase for some of the TOS API.
- You are encouraged to implement it in order to gain a deeper understanding of the TOS API.
- It is up to you how elaborate you want to implement the game logic.
- A very simple graphic interface is provided for your convenience.
- The implementation can be found in `~/tos/kernel/pacman.c`
- The first stage of PacMan can be implemented when assignment 2 is completed.

PacMan (2)

- After completing assignment 2, compile the TOS kernel via "`make`" (not "`make tests`") and run it inside of Bochs. You should see the PacMan maze.
- In `pacman.c`, create a function called `create_new_ghost()` according to the following pseudo code:

```
void create_new_ghost()
{
    GHOST ghost;
    init_ghost(&ghost);
    while (1) {
       remove ghost at old position (using remove_cursor())
       compute new position of ghost
       show ghost at new position (using show_cursor())
       do a delay
  }
}
```

PacMan (3)

- `init_shell()` (located in `~/tos/kernel/shell.c`) calls `init_pacman()` and already defines a window of proper size and position.
- Make sure you call `create_new_ghost()` from within `init_pacman()`.
- At this stage, all that will happen is that one ghost moves through the maze.
- Make sure that the ghost does not walk through walls (hint: use array `maze[]` for this)
- You can implement any behavior by which the ghost moves through the maze (random, always turn left, etc)
- The delay mentioned on the previous slide can be accomplished by a big for-loop that does nothing. The trick is to find the right number of iterations so that the ghost does not move too fast or too slow. Experiment!