Computer Labs: C for Lab 5

2º MIEIC

Pedro F. Souto (pfs@fe.up.pt)

November 1, 2019

Contents

C Pointers

- A C pointer is a data type whose value is a memory address.
  - Program variables are stored in memory
  - Other C entities are also memory addresses
- C provides two basic operators to support pointers:

  `&` to obtain the address of a variable. E.g.

```
p = &n; /* Initialize pointer p with
           the address of variable n */
```

  `*` to dereference the pointer, i.e. to read/write the memory positions it refers to.

```
*p = 8; /* Assign the value 8 to memory position
           whose address is
           the value of p (variable n) */
```

- To declare a pointer (variable), use the `*` operator:

```
int *p; /* Variable/pointer p points to integers or
           the value pointed to by p is of type int */
```

- Use of pointers in C is similar to the use of indirect addressing in assembly code, and as prone to errors.

C Pointers and Arrays

- The elements of an array are stored in consecutive memory positions
- In C, the name of an array is the address of the first element of that array:

```
int a[5];
p = a;          /* set p to point to the first element */
p = & (a[0]); /* same as above */
```

- C supports pointer arithmetic – meaningful only when used with arrays. E.g. to iterate through the elements of an array using a pointer:

```
for( i = 0, p = a; i < 5; i++, p++) {
    ...
}
```

or, without using variable `i`:

```
for( p = a; p-a < 5; p++) {
    ...
}
```

IMP: Pointer `p` must be declared to point to variables of the type of the elements of array `a`.

C Pointers and Pointer Arithmetic: `vg_fill()`

- Actually, pointer arithmetic may be used when we want to access a collection of data items of the same type that are layed consecutively in memory. E.g., the pixels in VRAM in graphics mode.

```
static void *video_mem; /* Address to which VRAM is mapped */
static unsigned hres;   /* Frame horizontal resolution */
static unsigned vres;   /* Frame vertical resolution */

void vg_fill(uint32_t color) {
  int i;
  uint32_t *ptr;   /* Assuming 4 bytes per pixel */
  ptr = video_mem;

  for(i = 0; i< hres*vres; i++, ptr++) {
       /* Handle a pixel at a time */
```

- Variables `video_mem`, etc. are global, but static
- `ptr++` takes advantage of pointer arithmetic (here adds 4, because each `uint32_t` takes 4 bytes)

Structs and Pointers: The -> operator

- C structs can be used to define structured types:

```
struct minix_mem_range {
    phys_bytes mr_base; /* Lowest memory address in range */
    phys_bytes mr_limit; /* Highest memory address in range
};
struct minix_mem_range mr, *mrp;
```

- To access a struct's member use the . operator:

```
mr.mr_base = (phys_bytes) vram_base;
```

Using a pointer to a struct:

```
mrp = &mr;
(*mrp).mr_base = (phys_bytes) vram_base;
```

or more readable (better):

```
mrp->mr_base = (phys_bytes) vram_base;
```

Typedef

- C structs are often used with `typedef`, a construct that allows to define new names for a type. For example (from Minix 3.1.8 source code):

```
typedef struct event
{
    ev_func_t ev_func;
    ev_arg_t ev_arg;
    struct event *ev_next;
} event_t;

extern event_t *ev_head;
```

- Basically, this means that instead of writing `struct event`, we can write only `event_t`
- Actually, with `typedef` we need not give a name to the struct (from `liblm.a`):

```
typedef struct {
    phys_bytes phys;        /* physical address */
    void *virtual;          /* virtual address */
    unsigned long size;     /* size of memory region */
} mmap_t;
```

C Unions

- Syntatically, a union data type appears like a struct:

```
union timer_status_field_val {
    uint8_t          byte;       // status, in hexadecimal
    enum timer_init  in_mode;    // initialization mode
    uint8_t          count_mode; // counting mode: 0, 1, ...,
    bool             bcd;        // true, if counting in BCD
};
```

  - Access to the members of a union is via the dot operator

- However, semantically, there is a big difference:

  Union contains space to store **any** of its members, but **not all** of its members simultaneously

  - The name **union** stems from the fact that a variable of this type can take values of **any** of the types of its members

  Struct contains space to store **all** of its members simultaneously

In `timer_print_config()` we are using it to reduce the number of arguments passed

- But need another argument the kind of information passed

Unions with Anonymous Structs

```
typedef struct reg86 {
  union {
    struct { /* 32-bit (double word) access*/
        [...]
    };
    struct { /* 16-bit (word) access */
        [...]
    };
    struct { /* 8-bit (byte) access */
      u8_t intno;  /* Interrupt number (input only) */
      u8_t : 8;    /* unused */
      u16_t : 16;  /* unused */
      [...]        /* unused */
      u8_t al, ah; /* 8-bit general registers */
      u16_t : 16;  /* unused */
      u8_t bl, bh; /* 8-bit general registers */
      u16_t : 16;  /* unused */
      u8_t cl, ch; /* 8-bit general registers */
      u16_t : 16;  /* unused */
      u8_t dl, dh; /* 8-bit general registers */
      [...]        /* unused */
    };
  };
} reg86_t;
```

- Access to the members of a union is via the dot operator