Computer Labs: C for Lab 5

2º MIEIC

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

November 10, 2017

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(char ch, char attr) {
  int i;
  unsigned long *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 `unsigned long` takes 4 bytes)

Structs and Pointers: The -> operator

- C structs can be used to define structured types:

  ```
  struct mem_range {
      phys_bytes mr_base; /* Lowest memory address in range */
      phys_bytes mr_limit; /* Highest memory address in range
  };
  struct 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;
```

Contents

C Unions

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

```
union reg_a {
   unsigned char a;     // 8080 A register
   unsigned short ax;   // 8086 AX register
   unsigned long eax;   // 80386 EAX register
} xax;
```

  - 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

Question What are unions good for?

C Union and Type Conversion

```
union reg_a {
    struct {
        unsigned char al, ah, _eax[2]; // access as 8-bit reg
    } b;
    struct {
        unsigned short ax, _eax;  // access as 16-bit registe
    } w;
    struct {
        unsigned long eax;  // access as 32-bit register
    } l;
} ia32_a;
```

- This allows us to initialize the union as a 32-bit register

```
ia32_a.l.eax = 0xD0D0DEAD;
```

- And later access any of the smaller registers available in the IA 32 architecture

```
printf("EAX = 0x%p \t AX = 0x%x \t AH = 0x%x \t AL = 0x%x \n",
       ia32_a.l.eax, ia32_a.w.ax, ia32_a.b.ah, ia32_a.b.al);
```

Contents

C Program Compilation

- A C program source code may be in different files
  - In each lab assignment you've been asked to write a set of functions, usually in a single file
  - In addition, you need to provide code for testing in a different file

  IMP: Following this approach, by the end of the lab assignments, you'll have code for the different devices in several different files, that you can reuse in your final project
- Furthermore, your program may need some code that has already been compiled into:

  User libraries i.e. libraries that some developer generated. E.g. `liblm.a`

  System libraries i.e. libraries that are usually provided together with the operating system. E.g. `libdriver.a` and `libsys.a`
- To simplify the building of your program, use `make` and `Makefiles`

Lab5 Makefile

```
COMPILER_TYPE= gnu

CC=gcc

PROG= lab5
SRCS= lab5.c vbe.c video_gr.c

CFLAGS= -Wall

DPADD+= ${LIBDRIVER} ${LIBSYS}
LDADD+= -llm -ldriver -lsys

LDFLAGS+= -L .

MAN=

BINDIR?= /usr/sbin

.include <bsd.prog.mk>
.include <bsd.gcc.mk>
```