

Computer Labs: Introduction to C

2º MIEIC

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

September 17, 2013

C vs. C++

- ▶ C++ is a **super**-set of C
 - ▶ C++ has classes – facilitates OO programming
 - ▶ C++ has references – safer and simpler than C pointers

C and Object Oriented Programming

- ▶ It is possible, and often desirable, to use OO programming in C
- ▶ A “class” may be implemented in a compilation unit, i.e. a file
 - ▶ We can use the keyword `static` to hide some aspects of the “class” implementation from the other code
 - ▶ There are yet some issues related to the visibility/accessibility of the data and functions that we’ll address later
- ▶ For each “class” we can define a header file containing its public interface
 - ▶ The function prototypes of its “public methods”
 - ▶ The data structures defined for the “class” and used in its public “methods”

I/O in C

- ▶ C provides standard streams for I/O:

```
stdin  
stdout  
stderr
```

- ▶ But C does not have the `cin` and `cout` objects nor the `>>` or the `<<` operators

- ▶ C does not support classes

- ▶ Instead you should use the functions:

```
scanf  
printf or fprintf()  
declared in <stdio.h>
```

printf()

```
printf("video_txt:: vt_print_string(%s, %lu, %lu, 0x%X)\n",  
str, row, col, (unsigned)attr);
```

- ▶ The first argument is the format string, which comprises:
 - ▶ Standard characters, which will be printed verbatim
 - ▶ Conversion specifications, which start with a % character
 - ▶ Format characters, such as \n or \t, for newline and tabs.
- ▶ The syntax of the conversion specifications is somewhat complex, but at least must specify the types of the values to be printed:
 - ▶ %c for a character, %x for an unsigned integer in hexadecimal, %d for an integer in decimal, %u for an unsigned integer in decimal, %l for a long in decimal, %lu for an unsigned long in decimal, %s for a string, %p for an address
- ▶ The remaining arguments should:
 - ▶ Match in number that of conversion specifications;
 - ▶ Have types compatible to those of the corresponding conversion specification
 - ▶ The first conversion specification refers to the 2nd argument, and so on

scanf ()

```
scanf("Origin: code = %c, attr = 0x%x, row = %d, col = %d",  
      &ch, &attr, &row, &col);
```

- ▶ The first argument is the format string, which comprises:
 - ▶ Normal characters, which will be printed verbatim – seldom used
 - ▶ Conversion specifications, which start with a % character
 - ▶ White spaces, which match any number, including zero, of white space characters (space, tab, newline, etc.)
- ▶ The syntax of the conversion specifications is similar to that of that used in `printf()`, with minor variations
- ▶ The remaining arguments should:
 - ▶ Match in number that of conversion specifications;
 - ▶ Be addresses of variables (**pointers**) of types compatible to those of the corresponding conversion specification
 - ▶ The first conversion specification refers to the 2nd argument, and so on
- ▶ Returns the number of items successfully matched and assigned (returns immediately if a conversion specification fails)

C Variables and Memory

- ▶ C variables abstract memory, and in particular memory addresses.
- ▶ When we declare a variable, e.g.:

```
int n; /* Signed int variable */
```

what the compiler does is to allocate a region of the process' address space large enough to contain the value of a signed integer variable, usually 4 bytes;

- ▶ Subsequently, while that declaration is in effect (this is usually called the **scope** of the declaration), uses of this variable name translate into accesses to its memory region:

```
n = 2*n; /* Double the value of n */
```

- ▶ However, in C, almost any “real world” program must explicitly use addresses
 - ▶ C++ provides references which are substitutes of C addresses that work in most cases

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: `vt_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 characters and its attributes of VRAM in text mode.

```
static char *video_mem;    /* Address to which VRAM is mapped
static unsigned scr_width; /* Width of screen in columns */
static unsigned scr_lines; /* Height of screen in lines */
```

```
void vt_fill(char ch, char attr) {
    int i;
    char *ptr;
    ptr = video_mem;

    for(i = 0; i < scr_width*scr_lines; i++, ptr++) {
```

- ▶ Variables `video_mem`, etc. are global, but static
- ▶ `ptr++` takes advantage of pointer arithmetic (here just adds one, because in C each character takes only 1 byte)

Strings and Pointers in C: `vt_print_string()`

- ▶ A string is an array of characters terminated by character code `0x00` (zero), also known as *end of string* character.
 - ▶ In C, a string is completely defined by the address of its first character

```
#define HELLO "Hello, World!"
```

```
...
```

```
char *p = HELLO; /* Set p to point to string HELLO */  
for( len = 0; *p != 0; p++, len++);
```

- ▶ The C standard library provides a set of string operations, that are declared in `<string.h>`

```
#include <string.h>
```

```
...
```

```
char *p = HELLO; /* Set p to point to string HELLO */  
len = strlen(p);
```

- ▶ Array names and string literals are constants not variables. The following is **WRONG**:

```
char a[20];
```

```
a = HELLO; /* This is similar to 2 = 5; */
```

```
HELLO = a; /* Same as above */
```

may use instead:

```
strncpy(a, HELLO, 20); /* Should NOT use strcpy */
```

Structs and Pointers: The `->` operator

- ▶ C structs can be used to define structured types:

```
struct vt_info {
    /* VRAM info */
    unsigned long vram_size; /* size in bytes of VRAM */
    void * vram_base;       /* VRAM physical address */
    /* Text mode resolution */
    unsigned scr_width;     /* # columns of the screen */
    unsigned scr_lines;    /* # lines of the screen */
};
struct vt_info vi, *vip;
```

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

```
vi.scr_width = NO_COLS;
```

Using a pointer to a struct:

```
vip = &vi;
(*vip).scr_width = NO_COLS;
```

or more readable (better):

```
vip->scr_width = NO_COLS;
```

Structs and Typedef

- ▶ To initialize on declaration is simpler:

```
struct vt_info vi = { VRAM_SIZE, VRAM_PHYS,  
                    NO_COLS, NO_LINES };
```

- ▶ C structs are often used with `typedef`, a construct that allows to define new names for a type. For example:

```
typedef struct vt_info vt_info_t;
```

```
vt_info_t vi, *vip;
```

- ▶ Basically, this means that instead of writing

```
struct vt_info, we can write only vt_info_t
```

- ▶ Actually, with `typedef` we need not give a name to the struct:

```
typedef struct {  
    /* VRAM info */  
    unsigned long vram_size; /* size in bytes of VRAM */  
    void * vram_base;       /* VRAM physical address */  
    /* Text mode resolution */  
    unsigned scr_width;     /* # columns of the screen */  
    unsigned scr_lines;    /* # lines of the screen */  
} vt_info_t;
```

Lab Preparation: Again

- ▶ It is a good practice to test your code gradually as you write it

Issue How can you test `vt_fill()` and `vb_blank()` before class, if you do not have Minix 3 installed yet?

Answer This year you cannot. Or better, I provide no support for that.

- ▶ One of the goals of Lab 0 is to help you set up LCOM's development environment
- ▶ Bring your laptop to that lab (preferably, with Linux already installed)

(Changes to the) Code Provided

`lab1.c` No need to change it. It includes:

`main()`

`print_usage()`

`proc_args()` invokes the functions you'll have to develop

And a few functions to parse integer arguments

`video_txt.c` The only file you need to change. It includes:

`vt_init()` which you **must** not change

And stubs, i.e. empty function declarations, for the functions that you have to implement

`vt_txt.h` Header file with the definition of the interface provided by `vt_txt.c`. I.e.,

- ▶ The C function declarations of the functions implemented in `vt_txt.c`

`vt_info.h` Header file with the prototypes of some functions whose implementation is provided in `libvt.a`

- ▶ These functions are called from `vt_init()`
- ▶ Includes also the definition of `vt_info_t()`, the type of the arguments of these functions

C Program Compilation

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

IMP: Following this approach, at the end of the lab assignments you'll have the I/O code for your 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. `libvt.a`

System libraries i.e. libraries that are usually provided together with the system. E.g. `libdriver.a` and `libsys.a`

- ▶ To simplify the building of your programs, we'll use `make` and `Makefiles`

Lab1 Makefile

```
COMPILER_TYPE= gnu

CC=gcc

PROG= lab1
SRCS= lab1.c video_txt.c

CCFLAGS= -Wall

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

LDFLAGS+= -L .

MAN=

BINDIR?= /usr/sbin

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

Configuring Minix and Running Your Program

- ▶ Starting with Lab 1, all the programs you develop in LCOM are privileged user-level programs
- ▶ These programs need to perform actions that non-privileged programs cannot perform, and should not perform.
 - ▶ E.g., in Lab 1, your program must access to VRAM
- ▶ But they should not be allowed to perform actions that they do not need
 - ▶ This is the **Principle of Least Privilege**
- ▶ Minix requires that you specify the privileged actions that a program can execute by means of a file with the name of program in directory `/etc/system.conf.d/`
 - ▶ You must be user `root` for this
 - ▶ But for compiling your program, you should be user `lcom`

/etc/system.conf.d/lab1

```
service lab1
{
    system
        READBIOS
        PRIVCTL
    ;
    ipc
        SYSTEM # to retrieve system information
        rs      # for service initialization
        vm      # to map video RAM
        pm      # for malloc, if needed
    ;
    uid
        0
    ;
};
```

Running Your Program

Disabling VT4

- ▶ Minix is configured with 4 VTs
- ▶ VT 1 is the console: messages written by privileged processes are displayed on VT1
- ▶ Thus, your program will use VT4
 - ▶ You need to type `Alt-F4`, to test the operation of your program
 - ▶ Avoid touching the keyboard when you are using VT4, although we instruct you to disable `getty` on that terminal

Invoking `service`

- ▶ Privileged processes must be started by the resurrection service (RS)
- ▶ In order to start a privileged program in Minix you use the `service` utility

```
# service run /usr/sbin/lab1 -args ["<list of space separated arguments>"]
```

- ▶ It reads and processes the program's configuration file
- ▶ It passes the required information to the RS, which then starts the process with the appropriate permissions