Advanced Operating Systems
Practical 1: C, pointers and memory map

UAH, Departamento de Automática, ATC-SOL
http://atc1.aut.uah.es

Theme 1

Abstract

The goal of this practical is to study how do programs use memory. At the same time, some aspects of the C language will be studied in depth, and the usage of development tools will be reviewed.

1 The C language

As well as Pascal[1], C is a declarative language. This means that, in a C program, every element (variable, function, data type, etc.) must be declared prior to its first use. As table 1 shows, a direct correspondence relates the basic elements of Pascal and C.

The next listing shows a "Hello world" program in C. The directive `#include <stdio.h>` makes the preprocessor insert the contents of the file stdio.h in the position of this line before the proper compilation —the actual translation into machine code— begins. As a result, the function printf is already declared when the compiler reaches line 7. What the compiler needs, in order to correctly translate the function call, is the number and type of the parameters that the function expects, and the type of the value it returns.

```
1 /*  Program helloworld.c  */
2
3 #include <stdio.h>    // Contains the declaration of printf
4
5 int main ()           // "main" function: returns an integer
6 {
7     printf ("Hello world.\n");        // '\n': new line
8     return 0;                         // 0: everything OK
9 }
```

The character '\n' at the end of the string "Hello world.\n" makes the next output text —if any— to appear in a new line. It is an escape sequence. The backslash character (\) starts the escape sequences in charater strings. The compiler translates the sequence \n as the LF (Line Feed) character, which has the hexadecimal code 0A, that is, it is stored in binary as 00001010. The ordinary backslash character is written in C with a double backslash: '\\'.

[1] Students of this subject are supposed to know Pascal programming

Table 1: Pascal - C equivalence

Pascal	C
Integer	int
Double	double
Char / Byte	char
record	struct
type	typedef
function	function that returns something
procedure	function that returns void (nothing)
uses	#include
write[ln]	printf
read	scanf
:=	=
=	==
<>	!=
begin .. end	{ .. }
{ comment }	/* comment */
x>5 and z=3	x>5 && z==3
x and 7	x & 7 (& with 2 operands)
inc(x)	x++
inc(x,2)	x+=2
For i:=0 To 100 Step 10 Do	for (i=0; i<=100; i+=10)
Var d:Double	double d;
Var p:^Double	double * p;
p:=^d; / p:=@d;	p=&d (& with 1 operand)
p^:=0.3;	*p=0.3; (* with 1 operand)

In C programs executing on MS-DOS or its successors (Windows...), the input/output function library substitutes the LF character with the sequence CR LF while printing on the screen or writing to text files. The CR (Carriage Return) character can be written in C as '\r'. Its hexadecimal code is 0D (00001101 in binary). The same input/output library carries out the inverse translation while reading text coming from the keyboard or text files, contracting the two-byte sequence CR LF to a single byte: LF.

The character '%' behaves similarly, but only in a specific context. At C language level, the character '%' is just an ordinary character. Only in printf() and scanf()[2] the character '%' acts as an escape sequence starter, and only when it appears inside the format string.

[2] and in their first cousins fprintf(), fscanf(), sprintf() y sscanf() too

2 Address of some variables

Type in the next program:

```
1  /*
2      Program mem_experiment.c
3  */
4
5  // Declaration (and definition) of some GLOBAL variables:
6
7  int a=34;
8  int b;
9  int c=-5;
10 float d;  // d and D are not related at all
11 int D;    // (the C language is case-sensitive)
12
13 // Main function of the program:
14
15 int main (int argc, char * argv[])
16 {
17     // LOCAL variables:
18     int x=3, y;
19
20     return 0;                          // 0: everything OK
21 }
```

Compile and execute it in debug mode with the next commands:

```
user@host:$ gcc mem_experiment.c -g -Wall -o mem_experiment
user@host:$ gdb --args mem_experiment Hola don Pepito
```

Once in debug mode, add a breakpoint in line 20:

```
(gdb) break mem_experiment.c:20
```

Run the program with the command `run` and watch the value of all its variables with the command `print`. For instance:

```
(gdb) print a
$1 = 34
(gdb) print &a
$2 = (int *) 0x804a010
```

Watch also the values and addresses of the parameters of function `main`: `&argc`, `argc`, `&argv`, `argv[0]`, `argv[1]`, `argv[2]` and `argv[3]`.

Write down the address of every variable or parameter. Analyse their layout in memory, and then answer the next questions:

1. Are local variables in the same memory zone as global variables?
2. What about the parameters `argc` and `argv`? Are they near global variables and/or near local ones?
3. What is the value of the global variables which were not explicitly initialized in the program?
4. What is the value of the uninitialized, local ones?
5. In which order do global variables appear in memory? Are they in the same order as in the source code? Is this related to the fact that some of them are initialized and others are not?

3 Addresses of other elements

In the following we will extend the program of the previous section. We will divide it into several modules, since it will get quite long. We will use the next `makefile` to compile it:

```
all: mem_experiment

mem_experiment: mem_experiment.o dynamic_mem.o
	gcc mem_experiment.o dynamic_mem.o -o mem_experiment -g -Wall

mem_experiment.o: mem_experiment.c dynamic_mem.h
	gcc -c mem_experiment.c -o mem_experiment.o -g -Wall

dynamic_mem.o: dynamic_mem.c dynamic_mem.h
	gcc -c dynamic_mem.c -o dynamic_mem.o -g -Wall

clean:
	rm -f *.o
```

Type in the code files listed below. You do not need to type the comments. If you are going to use `gdb` to watch the addresses of variables and functions, then you can also skip the calls to `printf`.

```
/*
    dynamic_mem.h
*/

#ifndef DYNAMIC_MEM_H        // If it's not defined ...
#define DYNAMIC_MEM_H        // ... we define it, and also:

typedef int integer;         // Synonym of type int
void dynamic_mem (void);     // Declaration of a function
extern int k;                // Declaration (just declaration!)

#endif                       // end of the block #ifndef - #endif
```

```c
/*
     dynamic_mem.c
*/

#include <stdio.h>            // Declaration of printf() (and more)
#include <stdlib.h>           // Decl. of malloc(), free(), ...

                              // The .h of this module is included
#include "dynamic_mem.h"      // in order to ensure that the
                              // declarations match their
                              // corresponding definitions

// Global variable "promised" in
// the .h with the extern declaration:

int k;

// Function that makes some experiments with dynamic memory:

void dynamic_mem (void)
{
    int * p;    // "p is a pointer to integer", or more literally:
                // "*p is an integer" (declaration resembles use)

    char * q, * r, * s;  // More (and yes: we need to repeat the *)

    printf ("\n\tFunction dynamic_mem(): %p\n", &dynamic_mem);

    printf ("\n\tLocal variables (addr., name, value):\n");
    printf ("\t\t%p  p  %p\n", &p, p);
    printf ("\t\t%p  q  %p\n", &q, q);
    printf ("\t\t%p  r  %p\n", &r, r);
    printf ("\t\t%p  s  %p\n", &s, s);

    // Ask for dynamic memory (malloc=="allocate memory"):

    p = (int*) malloc (7 * sizeof(int));      // 7 integers
    q = (char*) malloc (37 * sizeof(char));   // 37 char
    r = (char*) malloc (5 * sizeof(char));    // 5 char
    s = (char*) malloc (sizeof(char));        // 1 char

    // (int*) and (char*) are type conversions, needed
    // cause malloc reserves bytes in bulk, and returns a
    // generic pointer (void*), so we convert it to the
    // right type before making the assignment

    if (p==NULL || q==NULL || r==NULL || s==NULL)
    {
        if (p!=NULL) free (p);    // If any reservation failed,
        if (q!=NULL) free (q);    // undo the others, and
        if (r!=NULL) free (r);    // get out of this function
        if (s!=NULL) free (s);
        return;
    }
```

```

    printf ("\n\tMemory blocks reserved:\n");
    printf ("\t\tp (7 integers):  %p\n", p);
    printf ("\t\tq (37 integers): %p\n", q);
    printf ("\t\tr (5 char): %p\n", r);
    printf ("\t\ts (1 char): %p\n", s);

    printf ("\n\tPointer arithmetics:\n");
    printf ("\t\tp = %p \t p+1 = %p\n", p, p+1);
    printf ("\t\tq = %p \t q+1 = %p\n", q, q+1);

    *r = 'H';      // *r is the same as r[0]
    r[1] = 'o';    // r[1] is the same as *(r+1)
    r[2] = 'w';
    r[3] = '\0';   // Mark the end of the string

    printf ("\n\t%s, paleface!\n", r);

    printf ("\n\tDuality character/number of type char:\n");
    printf ("\t\t*r is a char, and its value is %d\n", *r);
    printf ("\t\tbut this value is also %c\n", *r);

    free (p);
    free (q);
    free (r);
    free (s);
}
```

```
/*
    mem_experiment.c
*/

#include <stdio.h>           // Declaration of printf() (and more)
#include "dynamic_mem.h"     // Decl. of things in dynamic_mem.c

// Declaration of other functions:

void hello (void);     // Doesn't receive or return anything
void strings (void);   //  "
int three (void);      // Receives nothing; returns an integer
int factorial (int);   // Receives an integer; returns another one

// Declaration (AND DEFINITION) of other function:

int triple (int x)
{
    printf ("Function triple(): %p\n" // %p: pointer
            "\tValue of x: %d\n"      // %d: int (in decimal)
            "\tAddress of x: %p",     // %p: another pointer
            &triple, x, &x);          // <--- Values to show

    return 3 * x;
}
```

```c

// Declaration (and def.) of some global variables:

int a=34;
int b;
int c=-5;

float d;  // d and D are not related at all
int D;    // (the C language is case-sensitive)

// Main function of the program:

int main (int argc, char * argv[])
{
    // Local variables:
    int x=3, y;

    hello ();

    printf ("Function printf(): %p\n", &printf);
    printf ("\nFunction main(): %p\n", &main);

    printf ("\n\tGlobal variables (addr., name, value):\n");
    printf ("\t\t%p  a  %d\n", &a, a);
    printf ("\t\t%p  b  %d\n", &b, b);
    printf ("\t\t%p  c  %d\n", &c, c);
    printf ("\t\t%p  d  %f\n", &d, d);  // %f: float
    printf ("\t\t%p  D  %d\n", &D, D);
    printf ("\t\t%p  k  %d\n", &k, k);  // variable of other .c

    printf ("\n\tLocal variables (addr., name, value):\n");
    printf ("\t\t%p  x  %d\n", &x, x);
    printf ("\t\t%p  y  %d\n", &y, y);

    printf ("\n\tComputing 3! ...\n");
    x = factorial (three());
    printf ("\t3! == %d\n", x);

    dynamic_mem ();     // function of the other .c

    strings ();

    return 0;           // 0 == everything went OK
}

// Definition of the remaining functions:

void hello (void)
{
    printf ("Hello world!\n");
}

int three (void)
{
```

```
    return 3;
}

int factorial (int n)
{
    int f;

    printf ("\tFunction factorial(%d): %p\n", n, &factorial);
    printf ("\t\tParameter n (addr., value): %p  %d\n", &n, n);
    printf ("\t\tVariable f (addr., value):  %p  %d\n", &f, f);

    f = n<2 ? 1 : n*factorial(n-1);

    /*
        In case it was not obvious, the prev. line is the same as:

            if (n<2)
                f = 1;
            else
                f = n * factorial (n-1);
    */

    printf ("\t\tfactorial(%d) returning %d\n", n, f);

    return f;
}

void strings (void)
{
    // These two arrays have the same size
    // and contain exactly the same:
    char a[] = "hello";
    char b[] = { 'h', 'e', 'l', 'l', 'o', '\0' };

    /*
        In Pascal, strings contain an integer that indicates
        their length. In C, the number of characters is not
        stored. Instead, the end of the string is marked with
        a special character: '\0', which is the same as 0
        (in binary: 00000000), and should not be mistaken for
        '0' (in binary: 00110000).
    */

    // This variable is not an array, but a pointer:
    char * c = "see you later";

    /*
        Right now, c is pointing to a literal string (constant,
        can be read but not written). An instruction like *c='H'
        would cause a runtime error.
    */

    printf ("\n\tFunction strings(): %p\n", &strings);
    printf ("\t\ta: %p  \"%s\"\n", a, a);
```

```
    printf ("\t\tb: %p  \"%s\"\n", b, b);
    printf ("\t\tc: %p  \"%s\"\n", c, c);
    printf ("\t\t&c: %p\n", &c);

    printf ("\t\tPlaying a bit with c...\n");

    c = a;      // Now c will point to the beginning of the
    *c = 'H';   // array a. Then we can modify *c cause c
                // now points to a memory zone that can be
                // written

    /*
        Note that we didn't need to write &a to obtain the
        address of the array a. This responds to an important
        exception in the syntax of the language.
        Usually, the name of a variable, unaccompanied,
        represents the value of the variable in the
        expression where it is used. With arrays, instead,
        the name itself represents the starting address of
        the array.

        Many compilers accept &a, but the most correct form
        is simply a.

        The same goes for functions' addresses. The
        unacompanied name ---without the parentheses---
        represents the starting address. In this program we
        used & for functions, but it is not necessary for
        obtaining the address of a function.
    */

    printf ("\t\ta: %p  \"%s\"\n", a, a);
    printf ("\t\tb: %p  \"%s\"\n", b, b);
    printf ("\t\tc: %p  \"%s\"\n", c, c);
    printf ("\t\t&c: %p\n", &c);
}
```

Once you have typed the code, you can homogenize the style —i.e., indentation— with the program `astyle` (*artistic style*) in order to make the code more readable. Though, if you typed it carefully, this will not be necessary.

```
user@host:$ astyle --style=ansi *.c *.h
```

Compile the program with `make` and execute it:

```
user@host:$ make
user@host:$ ./mem_experiment
```

If you typed the calls to `printf`, the result will be similar to the next excerpt:

```
Hello world!
Function printf(): 0x804839c

Function main(): 0x80484b8

        Global variables (addr., name, value):
                0x804a020  a  34
                0x804a038  b  0
                0x804a024  c  -5
                0x804a034  d  0.000000
```

If you did not type the calls to `printf`, debug the program step by step with `gdb` and collect the corresponding data using the `print` command. Remember that, in `gdb`, the command for taking one step is `step`, and the command for running —executing many steps in a row— until exiting the current function is `finish`.

See the memory map displayed in figure 1. Make a full map with all the elements of the program: functions, local variables, global variables, function parameters etc.

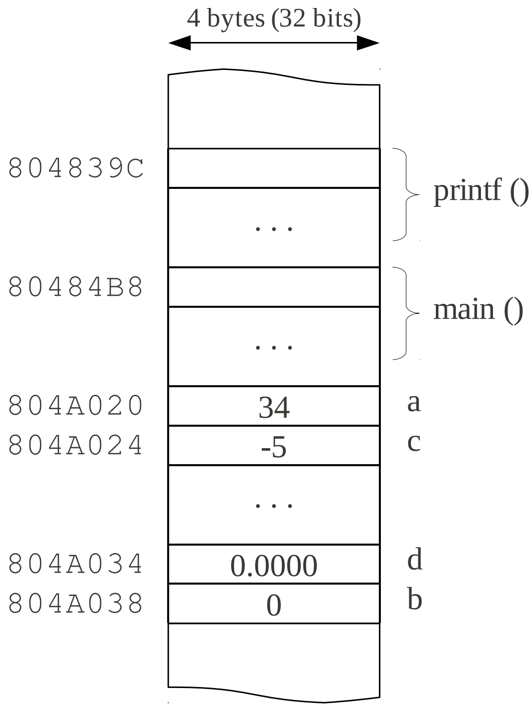

Figure 1: Memory map with some elements of the program

Answer the next questions:

6. Which are the memory addresses of the functions? Is there any other program element in that memory zone?

7. What is the address of `k` (global variable that belongs to `dynamic_mem.c`)? Is it near the global variables of `mem_experiment.c` or is it in a different zone?
8. See the addresses of the parameters passed to the functions. Are they related to the addresses of other elements of the program?
9. Compare the addresses of local variables of different functions. There is a case where two different variables occupy the same memory location. How can this be explained? Is it an issue?
10. The function `factorial` is recursive —sometimes it calls itself—. See the addresses and values of the parameter `n` and the local variable `f` in the successive invocations. Are they always the same addresses? What is the meaning of this?
11. Does the address of `factorial` itself change in the successive nested calls? Is it an issue?
12. See the function `strings`. Is the literal string `"see you later"` in the same memory zone as the arrays `a` and `b`? (By the way, reconsider your answer to question 6) What is the meaning of this?
13. See the function `dynamic_mem`. Which are the memory addresses of the dynamic memory blocks allocated by `malloc`? Are they near other elements of the program, or in a separated zone of memory?
14. Compare the addresses obtained evaluating `p+1` and `q+1` with those of `p` and `q` respectively. What is the difference, in bytes, in every case? How does the type of pointer affect the sum operation *pointer+integer*?
15. Analyse the ambivalence of the type `char` —equivalent to the types `Char` and `Byte` of `Pascal`—. What is the value of `*r` when it is printed with `%c`? And when it is printed with `%d`? What is the correspondence between numbers and characters?

Substitute the final line (`return 0;`) of the function `main` with an infinite loop:

```
int main (int argc, char * argv[])
{
    // [...]

    for (;;) {}      // The same as: while (1) {}
}
```

Open two console terminals and execute one instance of the program in every terminal. This will execute simultaneously —being precise: concurrently— two equal processes. Analyse the results and answer the next questions:

16. Do the memory addresses of the variables of the two processes match? If they do/did match... how could they be in equal addresses, at the same time, and still be different and independent variables?

You can stop the program with the key sequence `Control C`.