Sistemas Operativos Avanzados
Práctica 1: C, punteros y mapa de memoria

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

Tema 1

Resumen

El objetivo de esta práctica es estudiar la forma en que los programas utilizan la memoria, a la vez que se profundiza en el conocimiento del lenguaje C y se retoma el contacto con las herramientas de desarrollo utilizadas en el curso pasado.

1. El lenguaje C

Al igual que Pascal[1], C es un lenguaje declarativo. Esto significa que, en un programa escrito en C, todo elemento (variables, funciones, tipos de datos, etc.) debe ser declarado antes de ser utilizado. Como muestra la tabla 1, existe una correspondencia bastante directa entre los elementos básicos de Pascal y C.

El siguiente listado muestra un programa "Hola mundo" en C. La directiva `#include <stdio.h>` hace que el preprocesador inserte el contenido del archivo `stdio.h` en la posición de esa línea antes de que comience la compilación propiamente dicha (la traducción de las instrucciones a código máquina). Con esto se consigue que la funcion `printf` ya esté declarada cuando el compilador llegue a la línea 7. Lo que necesita el compilador para traducir correctamente la llamada a la función, es el tipo de los parámetros que recibe la función, y el tipo de dato que devuelve.

```
1 /*  Programa holamundo.c  */
2
3 #include <stdio.h>    // Contiene la declaración de printf
4
5 int main ()           // Función "principal": devuelve un entero
6 {
7     printf ("Hola mundo.\n");          // '\n': salto de línea
8     return 0;                          // 0: todo bien
9 }
```

El caracter `'\n'` que hay al final de la cadena `"Hola mundo.\n"` provoca un salto de línea. Se trata de una secuencia de escape. La barra invertida (\) sirve para iniciar las secuencias de escape en cualquier cadena de caracteres. El compilador traduce la secuencia `\n` por el carácter de salto de línea LF (*Line Feed*), que tiene el código hexadecimal `0A`, es decir, que se almacena en

[1] Se supone que los alumnos de esta asignatura saben programar en Pascal

Tabla 1: Equivalencia Pascal - C

Pascal	C
Integer	int
Double	double
Char / Byte	char
record	struct
type	typedef
función	función que devuelve algo
procedimiento	función que devuelve void (nada)
uses	#include
write[ln]	printf
read	scanf
:=	=
=	==
<>	!=
begin .. end	{ .. }
{ comentario }	/* comentario */
x>5 and z=3	x>5 && z==3
x and 7	x & 7 (& con 2 operandos)
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 (& con 1 operando)
p^:=0.3;	*p=0.3; (* con 1 operando)

binario como 00001010. El carácter de la barra invertida se escribe en C con dos barras invertidas seguidas: '\\'.

En los programas en C que se ejecutan en sistemas operativos herederos de MS-DOS, la biblioteca de funciones de entrada/salida sustituye el carácter LF por la secuencia CR LF al imprimir por pantalla o al escribir en archivos de texto. El carácter CR (*Carriage Return*) se puede escribir en C como '\r'. Su código hexadecimal es 0D (00001101 en binario). La misma biblioteca de entrada/salida se ocupa de realizar la traducción inversa al leer texto proveninete del teclado o de archivos de texto, contrayendo la secuencia de dos bytes CR LF en un solo byte: LF.

El carácter '%' tiene un comportamiento similar al de la barra invertida, pero sólo en un contexto determinado. A nivel del lenguaje C, el carácter '%' es completamente normal. Sólo las funciones printf() y scanf()[2] interpretan el carácter '%' como una secuencia de escape, y sólo cuando aparece dentro de la cadena de formato.

[2] y sus primas-hermanas fprintf(), fscanf(), sprintf() y sscanf() también

2. Dirección de algunas variables

Teclee el siguiente programa:

```
1  /*
2       Programa experimento_mem.c
3  */
4
5  // Declaración (y definición) de algunas variables GLOBALES:
6
7  int a=34;
8  int b;
9  int c=-5;
10 float d;  // d y D no tienen nada que ver
11 int D;    // (el lenguaje C es sensible a las mayúsculas)
12
13 // Función principal del programa:
14
15 int main (int argc, char * argv[])
16 {
17      // Variables LOCALES:
18      int x=3, y;
19
20      return 0;                                  // 0: todo bien
21 }
```

Compílelo y ejecútelo en modo depuración con las siguientes órdenes:

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

Una vez en modo depuración, añada un punto de ruptura en la línea 20:

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

Arranque el programa con la orden `run` y consulte el valor de todas variables y sus direcciones con la orden `print`. Por ejemplo:

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

Consulte también los valores y las direcciones de los parámetros de la función `main`: `&argc`, `argc`, `&argv`, `argv[0]`, `argv[1]`, `argv[2]` y `argv[3]`.

Anote las direcciones de todas las variables y parámetros. Analice la forma en que están dispuestos en la memoria, y responda a las siguientes preguntas:

1. ¿Están las variables locales en la misma zona de memoria que las variables globales?
2. ¿Y los parámetros `argc` y `argv`? ¿Están cerca de las variables globales y/o cerca de las locales?
3. ¿Qué valor tienen las variables globales a las que el programa no asignó un valor inicial?
4. ¿Y las locales?
5. ¿En qué orden están las variables globales entre sí? ¿Están en el mismo orden que en el código fuente? ¿Tiene algo que ver el orden con el hecho de que unas estén inicializadas y otras no?

3. Dirección de otros elementos

A continuación ampliaremos el programa de la sección anterior. Como va a terminar siendo bastante extenso, dividiremos el programa en varios módulos. Utilizaremos el siguiente `makefile` para compilarlo:

```
all: experimento_mem

experimento_mem: experimento_mem.o mem_dinamica.o
	gcc experimento_mem.o mem_dinamica.o -o experimento_mem -g -Wall

experimento_mem.o: experimento_mem.c mem_dinamica.h
	gcc -c experimento_mem.c -o experimento_mem.o -g -Wall

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

clean:
	rm -f *.o
```

Teclee los archivos de código listados a continuación. No teclee los comentarios si no lo desea. Si prefiere evaluar con `gdb` las direcciones de las variables y funciones, entonces puede ahorrarse teclear las llamadas a `printf`.

```
/*
    mem_dinamica.h
*/

#ifndef MEM_DINAMICA_H      // Si no está definido ...
#define MEM_DINAMICA_H      // ... lo definimos, y además:

typedef int entero;         // Sinónimo del tipo int
```

```
9  void mem_dinamica (void);  // Declaración de una función
10 extern int k;              // Declaración (¡sólo declaración!)
11
12 #endif                     // final del bloque #ifndef - #endif
```

```
1  /*
2      mem_dinamica.c
3  */
4
5  #include <stdio.h>         // Declaración de printf() (y más)
6  #include <stdlib.h>        // Decl. de malloc(), free(), ...
7
8                             // Se incluye el .h de este módulo
9  #include "mem_dinamica.h"  // para asegurar que las
10                            // declaraciones concuerdan con
11                            // las definiciones
12
13 // Variable global "prometida" en
14 // el .h con la declaración extern:
15
16 int k;
17
18 // Función que experimenta con memoria dinámica:
19
20 void mem_dinamica (void)
21 {
22     int * p;    // "p es un puntero a entero", o más bien:
23                 // "*p es un entero" (declaration resembles use)
24
25     char * q, * r, * s;  // Más (y sí: hay que repetir el *)
26
27     printf ("\n\tFunción mem_dinamica(): %p\n", &mem_dinamica);
28
29     printf ("\n\tVariables locales (dir., nombre, valor):\n");
30     printf ("\t\t%p  p  %p\n", &p, p);
31     printf ("\t\t%p  q  %p\n", &q, q);
32     printf ("\t\t%p  r  %p\n", &r, r);
33     printf ("\t\t%p  s  %p\n", &s, s);
34
35     // Pedimos memoria dinámica (malloc=="allocate memory"):
36
37     p = (int*) malloc (7 * sizeof(int));      // 7 enteros
38     q = (char*) malloc (37 * sizeof(char));   // 37 char
39     r = (char*) malloc (5 * sizeof(char));    // 5 char
40     s = (char*) malloc (sizeof(char));        // 1 char
41
42     // (int*) y (char*) son conversiones de tipo necesarias
43     // porque malloc reserva bytes a granel, y devuelve un
44     // puntero genérico (void*), así que lo convertimos
45     // al tipo adecuado antes de hacer la asignación
46
47     if (p==NULL || q==NULL || r==NULL || s==NULL)
48     {
```

```
49          if (p!=NULL) free (p);    // Si alguna reserva falló,
50          if (q!=NULL) free (q);    // deshacer las demás y
51          if (r!=NULL) free (r);    // salir de esta función
52          if (s!=NULL) free (s);
53          return;
54      }
55
56      printf ("\n\tBloques de memoria reservados:\n");
57      printf ("\t\tp (7 enteros):  %p\n", p);
58      printf ("\t\tq (37 enteros): %p\n", q);
59      printf ("\t\tr (5 char): %p\n", r);
60      printf ("\t\ts (1 char): %p\n", s);
61
62      printf ("\n\tAritmética de punteros:\n");
63      printf ("\t\tp = %p \t p+1 = %p\n", p, p+1);
64      printf ("\t\tq = %p \t q+1 = %p\n", q, q+1);
65
66      *r = 'J';      // *r es lo mismo que r[0]
67      r[1] = 'a';    // r[1] es lo mismo que *(r+1)
68      r[2] = 'u';
69      r[3] = '\0';   // Marca de fin de cadena
70
71      printf ("\n\t¡%s, rostro pálido!\n", r);
72
73      printf ("\n\tDualidad caracter/numero del tipo char:\n");
74      printf ("\t\t*r es un char, y vale %d\n", *r);
75      printf ("\t\tpero también vale %c\n", *r);
76
77      free (p);
78      free (q);
79      free (r);
80      free (s);
81  }
```

```
1   /*
2       experimento_mem.c
3   */
4
5   #include <stdio.h>          // Declaración de printf() (y más)
6   #include "mem_dinamica.h"   // Declaraciones de mem_dinamica.c
7
8   // Declaración de otras funciones:
9
10  void hola (void);     // No recibe nada y no devuelve nada
11  void cadenas (void); //  "
12  int tres (void);      // No recibe nada y devuelve un entero
13  int factorial (int); // Recibe un entero y devuelve otro
14
15  // Declaración (Y DEFINICIÓN) de otra función:
16
17  int triple (int x)
18  {
19      printf ("Función triple(): %p\n"  // %p: puntero
```

```
20              "\tValor de x: %d\n"       // %d: int (en decimal)
21              "\tDirección de x: %p",    // %p: otro puntero
22              &triple, x, &x);           // <--- Valores a mostrar
23
24      return 3 * x;
25  }
26
27  // Declaración (y def.) de algunas variables globales:
28
29  int a=34;
30  int b;
31  int c=-5;
32
33  float d;  // d y D no tienen nada que ver
34  int D;    // (el lenguaje C es sensible a las mayúsculas)
35
36  // Función principal del programa:
37
38  int main (int argc, char * argv[])
39  {
40      // Variables locales:
41      int x=3, y;
42
43      hola ();
44
45      printf ("Función printf(): %p\n", &printf);
46      printf ("\nFunción main(): %p\n", &main);
47
48      printf ("\n\tVariables globales (dir., nombre, valor):\n");
49      printf ("\t\t%p  a  %d\n", &a, a);
50      printf ("\t\t%p  b  %d\n", &b, b);
51      printf ("\t\t%p  c  %d\n", &c, c);
52      printf ("\t\t%p  d  %f\n", &d, d);  // %f: float
53      printf ("\t\t%p  D  %d\n", &D, D);
54      printf ("\t\t%p  k  %d\n", &k, k);  // variable del otro .c
55
56      printf ("\n\tVariables locales (dir., nombre, valor):\n");
57      printf ("\t\t%p  x  %d\n", &x, x);
58      printf ("\t\t%p  y  %d\n", &y, y);
59
60      printf ("\n\tCalculando 3! ...\n");
61      x = factorial (tres());
62      printf ("\t3! == %d\n", x);
63
64      mem_dinamica ();    // función del otro .c
65
66      cadenas ();
67
68      return 0;           // 0 == todo ha ido bien
69  }
70
71  // Definición de las funciones restantes:
72
73  void hola (void)
```

```
74 {
75     printf ("¡Hola mundo!\n");
76 }
77
78 int tres (void)
79 {
80     return 3;
81 }
82
83 int factorial (int n)
84 {
85     int f;
86
87     printf ("\tFunción factorial(%d): %p\n", n, &factorial);
88     printf ("\t\tParámetro n (dir., valor): %p  %d\n", &n, n);
89     printf ("\t\tVariable f (dir., valor):  %p  %d\n", &f, f);
90
91     f = n<2 ? 1 : n*factorial(n-1);
92
93     /*
94         Por si no resulta obvio, la línea anterior equivale a:
95
96             if (n<2)
97                 f = 1;
98             else
99                 f = n * factorial (n-1);
100    */
101
102    printf ("\t\tfactorial(%d) devolviendo %d\n", n, f);
103
104    return f;
105 }
106
107 void cadenas (void)
108 {
109    // Estos dos arrays son igual de grandes y
110    // contienen exactamente lo mismo:
111    char a[] = "hola";
112    char b[] = { 'h', 'o', 'l', 'a', '\0' };
113
114    /*
115        En Pascal, las cadenas contienen un entero que indica
116        su longitud. En C, sin embargo, no se almacena el
117        número de caracteres, sino que se señaliza el final de
118        la cadena con un caracter especial: '\0', que es lo
119        mismo que 0 (en binario: 00000000), y que no debe ser
120        confundido con el '0' (en binario: 00110000)
121    */
122
123    // Esto no es un array, sino un puntero:
124    char * c = "hasta luego";
125
126    /*
127        En este momento, el puntero c está apuntando a una
```

```
128             cadena literal (constante, que se puede leer, pero
129             no escribir). Una instrucción como *c='H' provocaría
130             un error en tiempo de ejecución.
131         */
132
133         printf ("\n\tFunción cadenas(): %p\n", &cadenas);
134         printf ("\t\ta: %p  \"%s\"\n", a, a);
135         printf ("\t\tb: %p  \"%s\"\n", b, b);
136         printf ("\t\tc: %p  \"%s\"\n", c, c);
137         printf ("\t\t&c: %p\n", &c);
138
139         printf ("\t\tJugando un poco con c...\n");
140
141         c = a;      // Ahora c apuntará al comienzo del array a
142         *c = 'H';   // Ya sí se puede modificar *c porque ahora c
143                     // apunta a una zona de memoria en la que se
144                     // puede escribir
145
146         /*
147             Nótese que no hemos necesitado escribir &a para
148             obtener la dirección del array a. Esto se debe a una
149             importante excepción en la sintaxis del lenguaje.
150             Normalmente, el nombre de una variable, sin más,
151             arroja el valor de la variable en la expresión en que
152             se usa. Con los arrays, sin embargo, el nombre suelto
153             da la dirección de comienzo del array.
154
155             Muchos compiladores aceptan &a, pero la forma más
156             correcta es simplemente a.
157
158             Lo mismo ocurre con la dirección de las funciones. El
159             nombre suelto, sin los paréntesis, da la dirección de
160             comienzo. A lo largo de todo este programa se ha usado
161             el &, pero no es necesario para obtener la dirección
162             de una función.
163         */
164
165         printf ("\t\ta: %p  \"%s\"\n", a, a);
166         printf ("\t\tb: %p  \"%s\"\n", b, b);
167         printf ("\t\tc: %p  \"%s\"\n", c, c);
168         printf ("\t\t&c: %p\n", &c);
169     }
```

Una vez tecleado el código, puede homogeneizar el estilo (sangrado etc.) con el programa `astyle` (*artistic style*) para hacerlo más legible. No obstante, si lo ha tecleado con cuidado, no será necesario.

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

Compile el programa con `make` y ejecútelo:

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

Si ha tecleado las llamadas a `printf`, obtendrá un resultado similar a este fragmento:

```
¡Hola mundo!
Función printf(): 0x804839c

Función main(): 0x80484b8

        Variables globales (dir., nombre, valor):
                0x804a020  a  34
                0x804a038  b  0
                0x804a024  c  -5
                0x804a034  d  0.000000
```

Si no ha tecleado las llamadas a `printf`, depure el programa paso a paso con `gdb` y obtenga los datos correspondientes con la orden `print`. Recuerde que, en `gdb`, la orden para avanzar un paso es `step`, y la orden para continuar la ejecución hasta haber salido de la función actual es `finish`.

Observe el mapa de memoria de la figura 1. Realice un mapa completo con todos los elementos del programa: funciones, variables locales, variables globales, parámetros de las funciones etc.

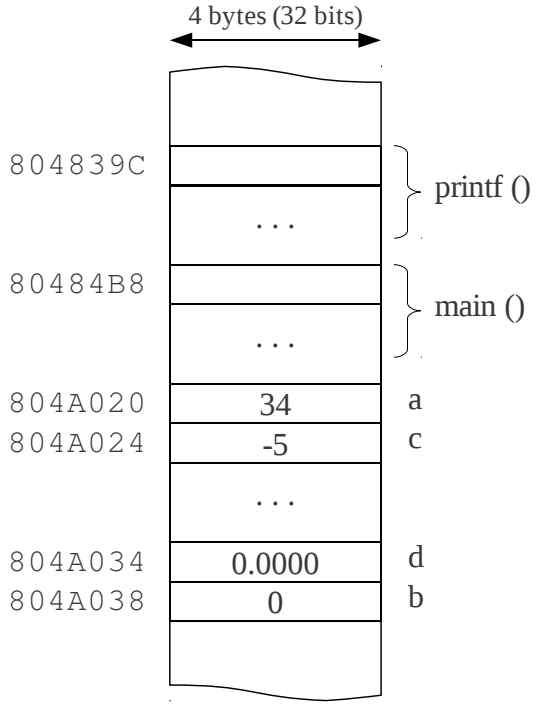

Figura 1: Mapa de memoria con algunos elementos del programa

Responda a las siguientes cuestiones:

6. ¿En qué direcciones de memoria se encuentran las funciones? ¿Hay algún otro elemento del programa en esa zona de la memoria?

7. ¿En qué dirección está `k` (variable global perteneciente a `mem_dinamica.c`)? ¿Está junto a las variables globales de `experimento_mem.c` o está en una zona separada?

8. Observe la dirección de los parámetros pasados a las funciones. ¿Guardan alguna relación con las direcciones de otros elementos del programa?

9. Observe las direcciones de las variables locales de las diferentes funciones. Hay algún caso en que dos variables de diferentes funciones ocupan la misma posición en la memoria. ¿Cómo se explica? ¿Puede esto ocasionar algún problema?

10. La función `factorial` es recursiva (en ocasiones se invoca a sí misma). Observe las direcciones y los valores del parámetro `n` y la variable local `f` en las sucesivas invocaciones. ¿Son siempre las mismas direcciones? ¿Qué sentido tiene esto?

11. ¿Cambia la dirección de la propia función `factorial` en las sucesivas invocaciones anidadas? ¿Puede esto ocasionar algún problema?

12. Observe la función `cadenas`. ¿Se encuentra la cadena literal `"hasta luego"` en la misma región de memoria que los arrays `a` y `b`? (Por cierto, reconsidere su respuesta a la pregunta 6) ¿Qué sentido tiene esto?

13. Observe la función `mem_dinamica`. ¿En qué direcciones se ubican los bloques de memoria dinámica reservados por `malloc`? ¿Se encuentran cerca de otros elementos del programa o en una región de memoria separada?

14. Observe las direcciones que resultan al evaluar `p+1` y `q+1` en relación a `p` y `q` respectivamente. ¿Cuál es la diferencia, en bytes, en un caso y en el otro? ¿Cómo incide el tipo del puntero en la operación de suma *puntero+entero*?

15. Observe la ambivalencia del tipo `char` (equivalente a los tipos `Char` y `Byte` de `Pascal`). ¿Qué valor tiene `*r` cuando se muestra con `%c`? ¿Y cuando se muestra con `%d`? ¿Cuál es la correspondencia entre números y caracteres?

Sustituya la línea final (`return 0;`) de la función `main` por un bucle infinito:

```
1  int main (int argc, char * argv[])
2  {
3      // [...]
4
5      for (;;) {}      // Lo mismo que: while (1) {}
6  }
```

Abra dos terminales y ejecute una instancia del programa en cada una. De esta forma ejecutará simultáneamente (o para ser precisos: concurrentemente) dos procesos iguales. Observe los resultados y responda a las siguientes preguntas:

16. ¿Coinciden las direcciones de memoria de las variables de un proceso con las del otro? Si coinciden, o si coincidieran... ¿cómo podrían estar en direcciones iguales, al mismo tiemo, y seguir siendo variables diferentes e independientes entre sí?

Puede detener la ejecución del programa con la secuencia de teclas `Control C`.