

Memoria en C/C++

Di Paola Martín
martinp.dipaola <at> gmail.com

Facultad de Ingeniería
Universidad de Buenos Aires

1

De qué va esto?

- Memoria
 - Tamaños, Alineación y Padding
 - Segmentos de Memoria
- Punteros
 - Punteros
 - Typedef
- Buffer overflows

2

Memoria

Tamaños, Alineación y Padding

Exacta reserva de memoria

```
1 char c = 'A';  
2 int i = 1;  
3 short int s = 4;  
4 char *p = 0;  
5 int *g = 0;  
6 int b[2] = {1, 2};  
7 char a[] = "AB";
```

- Todo depende de la arquitectura y del compilador
- Alineación y padding
- Punteros del mismo tamaño
- Un cero como "fin de string"

c	65			
i	0	0	0	1
s	0	4		
p	0	0	0	0
g	0	0	0	0
b	0	0	0	1
	0	0	0	2
a	65	66	0	

3

4

- C y C++ son lenguajes de bajo nivel para que el programador pueda tener un control absoluto de dónde y cómo se ejecuta el código.
- El tamaño en bytes de los tipos depende de la arquitectura y del compilador
- El compilador puede guardar las variables en posiciones de memoria múltiplos de 4 (depende de la arquitectura y de los flags de compilación): variables alineadas son accedidas más rápidamente que las desalineadas.
- Como contra, la alineación despendicia espacio (padding) hay un tradeoff entre velocidad y espacio.
- El tamaño de un puntero no depende de a que tipo apunta; todos los punteros ocupan el mismo tamaño (que depende de la arquitectura).
- A los strings en C escritos en el código del programa el compilador les agrega el caracter nulo (byte 0). Tenerlo en cuenta!!

Agrupación de variables

```
1 struct S {  
2     int a;  
3     char b;  
4     int c;  
5     char d;  
6 };  
7  
8 struct S s = {1,2,3,4};
```

s.a	0	0	0	1
s.b	2			
s.c	0	0	0	3
s.d	4			

5

- El padding se hace mas notorio en las estructuras: el acceso a cada atributo es rápido pero hay memoria desperdiciada.

Agrupación de variables

```

1 struct S {
2     int a;
3     char b;
4     int c;
5     char d;
6 } __attribute__((packed));
7
8 struct S s = {1,2,3,4};

```

s.a	0	0	0	1
s.b/s.c	2	0	0	0
s.c/s.d	3	4		

6

- Con el atributo especial de gcc `__attribute__((packed))` el compilador empaqueta los campos sin padding, más eficiente en memoria pero más lento.
- Y es más lento por que para leer el atributo `s.c` hay que hacer 2 lecturas.
- Y cuidado, en algunas arquitecturas la lectura de atributos desalineados hace crashear al programa!

Endianess: representación en memoria

```

1 int i = 1;
2
3
4
5
6
7 int j = 1023;

```

<code>((unsigned char*)&i)</code> == {0, 0, 0, 1}	0 0 0 1	big
<code>((unsigned char*)&i)</code> == {1, 0, 0, 0}	1 0 0 0	little
<code>((unsigned char*)&j)</code> == {0, 0, 3, 255}	0 0 3 ff	big
<code>((unsigned char*)&j)</code> == {255, 3, 0, 0}	ff 3 0 0	little

7

- El byte más significativo se lee/escribe primero (o esta primero en la memoria) en las arquitecturas big endian.
- Por el contrario en las arquitecturas little endian es el byte menos significativo quien esta primero en la memoria.
- El endianess es irrelevante si siempre trabajamos los `ints` como números pero se vuelve relevante en el momento que queremos interpretar un `int` como una tira de bytes (`char*`) o viceversa. Y esto es necesario cuando queremos escribir un número en un archivo binario o enviarlo por la red a otra máquina a través de un socket!
- Siempre hay que especificar el endianess en que se guardan/envían los datos.
- Obviamente lo mencionado aquí para los `ints` aplica para el resto de los objetos en memoria, como los `shorts`

Endianess: representación en memoria

Se puede cambiar el endianess de una variable `short int` y `int` del endianess nativo o "del host" a big endian o "el endianess de la red" y viceversa:

- Host to Network

```
1 htons(short int) htonl(int)
```

- Network to Host

```
1 ntohs(short int) ntohl(int)
```

8

- Para hacer uso de esas funciones hay que hacer `#include <arpa/inet.h>`.

Memoria

Segmentos de Memoria

9

Segmentos de memoria

- Code segment: de solo lectura y ejecutable, a donde va el código y las constantes.
- Data segment: variables creadas al inicio del programa y son válidas hasta que este termina; pueden ser de acceso global o local.
- Stack: variables creadas al inicio de una llamada a una función y destruidas automáticamente cuando esta llamada termina.
- Heap: variables cuya duración esta controlada por el programador (run-time).

10

Duración y visibilidad (lifetime and scope)

- Duración (lifetime): tiempo desde que a la variable se le reserva memoria hasta que esta es liberada. Determinado por el segmento de memoria que se usa.
- Visibilidad (scope): Cuando una variable se la puede acceder y cuando esta oculta.

11

Asignación del lifetime y scope

```
1 int g = 1;
2 static int l = 1;
3 extern char e;

5 void Fa() { }
6 static void Fb() { }
7 void Fc();

void foo(int arg) {
10     int a = 1;
11     static int b = 1;

13     void * p = malloc(4);
14     free(p);

16     char *c = "ABC";
17     char ar[] = "ABC";
}
```

12

Asignación del lifetime y scope

```
1 int g = 1;           // Data segment; scope global
2 static int l = 1;    // Data segment; scope local (este file)
3 extern char e;       // No asigna memoria (es un nombre)
4
5 void Fa() { }         // Code segment; scope global
6 static void Fb() { }  // Code segment; scope local (este file)
7 void Fc();            // No asigna memoria (es un nombre)
8
9 void foo(int arg) {   // Argumentos y retornos son del stack
10     int a = 1;        // Stack segment; scope local (func foo)
11     static int b = 1; // Data segment; scope local (func foo)
12
13     void * p = malloc(4); // p en el Stack; apunta al Heap
14     free(p);            // liberar el bloque explicitamente!!
15
16     char *c = "ABC";    // c en el Stack; apunta al Code Segment
17     char ar[] = "ABC";  // es un array con su todo en el Stack
18 } // fin del scope de foo: las variables locales son liberadas
```

13

El donde importa!

```
1
2 void f() {
3     char *a = "ABC";
4     char b[] = "ABC";
5
6     b[0] = 'X';
7     a[0] = 'X'; // segmentation fault
8 }
```

14

- Como el puntero "a" apunta al Code Segment y este es de solo lectura, tratar de modificarlo termina en un Segmentation Fault

Punteros

Punteros

Punteros

```
1 int *p;    // p es un puntero a int
2            // (p guarda la direccion de un int)
3
4 int i = 1;
5 p = &i;    // &i es la direccion de la variable i
6
7 *p = 2;    // *p dereferencia o accede a la memoria
8            // cuya direccion esta guardada en p
9
10 /* i == 2 */

```

```
1
2 char buf[512];
3 write(&buf[0], 512);
```

15

16

Aritmética de punteros

```
1 int a[10];
2 int *p;
3
4 p = &a[0];
5
6 *p    // a[0]
7 *(p+1) // a[1]
8
9
10 int *p;
11 p+1    // movete sizeof(int) bytes (4)
12
13 char *c;
14 c+2    // movete 2*sizeof(char) bytes (2)
```

17

- La notación de array (indexado) y la aritmética de punteros son esencialmente lo mismo.
- La aritmética de punteros se basa en el tamaño de los objetos a los que se apunta al igual que el indexado de un array.

Punteros a funciones (al code segment)

```
1 int g(char) {}
2
3 int (*p) (char);
4 p = &g;
5
6 #include <stdlib.h>
7 void qsort(void *base,
8             size_t nmemb,
9             size_t size,
10
11             int (*cmp) (const void *, const void *))
12
13 int cmp_personas(const void* a, const void* b) {
14     struct Persona *pa = (struct Persona*)a;
15     struct Persona *pb = (struct Persona*)a;
16
17     return pa->edad < pb->edad;
18 }
```

Punteros

Typedef

Como leer la bizarra notación de punteros en C/C++

```
1 /* Ejemplo 1 */
2 char *a[10];
3     a           // "a"
4     *a          // "a" apunta a
5 char *a         // "a" apunta a char
6 char *a[10];    // "a" apunta a char (10 de esos)
7
8 char *a[10];    // "a" es un array de 10 de esos, o sea
9                // "a" es un array de 10 punteros a char
10
```

Como leer la bizarra notación de punteros en C/C++

```
1 /* Ejemplo 2 */
2 char (*c) [10];
3     c           // "c"
4     *c          // "c" apunta a
5     (*c) == X  // llamemos "X" a (*c) temporalmente
6
7 char X[10];
8 char X[10];    // "X" es un char (10 de esos)
9
10 char X[10];    // "X" es un array de 10 char
11 char (*c) [10]; // "c" apunta a un array de 10 char
12
```

Como leer la bizarra notación de punteros en C/C++

```
1 /* Ejemplo 3: modo dios */
2 char (*f) (int) [10];
3     f           // "f"
4     *f          // "f" apunta a
5     (*f) == X
6
7 char X(int) [10];
8 char X(int)     // es la firma de una funcion,
9                // asi que vuelvo un paso para atras
10 char (*f) (int) // entonces esto es un puntero a funcion
11                // cuya firma recibe un int y retorna
12                // un char
13
14 char (*f) (int) [10]; // puntero a funcion, 10 de esos
15 char (*f) (int) [10]; // f es un array de 10 punteros a funcion,
16                // que reciben un int y retornan un chars
17
```

Simplificando la notación

```
1 char *X[10]; // la variable "X" es un array de
2              // 10 punteros a char
3
4 typedef char *X[10]; // el tipo "X" es un array de
5                     // 10 punteros a char
6
7 X my_array; // es una alias, decir "X" es como decir
8 char *my_array[10]; // "array de 10 punteros a char"
```

Simplificando la notación

Si quiero una variable que sea un array de punteros a función que no reciban ni retornen nada?

```
1 void (*X)(); // la variable "X" es un puntero a
2           // funcion

1 typedef void (*X)(); // el tipo "X" es un puntero a
2                   // funcion
3
4 X f[10];          // f es una array de 10 X, entonces
5                   // f es una array de 10 punteros
6                   // a funcion
```

24

Buffer overflows

25

Smash the stack for fun and profit

```
1 // compilar con el flag -fno-stack-protector
2 #include <stdio.h>
3
4 int main(int argc, char *argv[]) {
5     int cookie = 0;
6     char buf[10];
7
8     printf("buf:_%08x_cookie:_%08x\n", buf, &cookie);
9     gets(buf);
10
11     if (cookie == 0x41424344) {
12         printf("You_win!\n");
13     }
14
15     return 0;
16 } // Insecure Programming
```

26

- Es claro que al inicializar `cookie` a cero nunca se va a imprimir "You_win!"... o sí?
- `gets` lee de la entrada estándar hasta encontrar un `'\n'` y lo que lee lo escribe en el buffer `buf`. Pero si el input es más grande que el buffer, `gets` escribe por fuera de este y sobrescribe todo el stack lo que se conoce como Buffer Overflow.
- Para hacer que el programa entre al `if` e imprima "You_win!" se debe forzar a un buffer overflow con un input craftado:
- Debe tener 10 bytes de mínima para ocupar el buffer `buf`.
- Posiblemente deba tener algunos bytes adicionales para ocupar el posible espacio de padding usado para alinear las variables.
- Luego se debe escribir los 4 bytes que sobrescriban `cookie` pero cuidado, dependiendo de la arquitectura y flags del compilador `sizeof(int)` puede no ser 4.
- Suponiendo que sean 4 bytes, hay que escribir el número `0x41424344` byte a byte y el orden dependerá del endianness: "ABCD" en big endian, "DCBA" en little endian.

Buffer overflow

- Funciones inseguras que no ponen un límite en el tamaño del buffer que usan. No usarlas!

```
1 gets(buf);
2 strcpy(dst, src);
```

- Reemplazarlas por funciones que sí permiten definir un límite, pero es responsabilidad del programador poner un valor coherente!

```
1 getline(buf, max_buf_size, stream);
2 strncpy(dst, src, max_dst_size);
```

27

Challenge: hacer que el programa imprima "You win!"




```
1 // compilar con el flag -fno-stack-protector
2 #include <stdio.h>
3
4 int main(int argc, char *argv[]) {
5     int cookie = 0;
6     char buf[10];
7
8     printf("buf:_%08x_cookie:_%08x\n", buf, &cookie);
9     gets(buf);
10
11     if (cookie == 0x41424344) {
12         printf("You_loose!\n");
13     }
14
15     return 0;
16 } // Insecure Programming
```

28

Appendix

Referencias

Referencias I

-  Bjarne Stroustrup.
The C++ Programming Language.
Addison Wesley, Fourth Edition.
-  man page: gets strcpy htons qsort
-  Insecure Programming