

## Memoria en C/C++

<sup>1</sup>Facultad de Ingeniería  
Universidad de Buenos Aires



## De qué va esto?

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



7542

Memoria en C/C++

Memoria  
Punteros  
Buffer overflowsTamaños, Alineación y Padding  
Segmentos de Memoria

## Exacta reserva de memoria

...sabiendo la arquitectura y la configuración del compilador.

```
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";
```

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	



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

- 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 desperdicia espacio (padding) hay un tradeoff entre velocidad y espacio.
- El tamaño de los punteros no depende de a que apunta; todos los punteros ocupan el mismo tamaño.
- Los strings en C escritos en el programa terminan con el caracter nulo (byte 0). No olvidarselo!!

7542

Memoria en C/C++

Memoria  
Punteros  
Buffer overflowsTamaños, Alineación y Padding  
Segmentos de Memoria

## 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			



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

7542

Memoria en C/C++

## Grupación 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		



- Con el atributo especial de gcc `__attribute__((packed))` el compilador empaqueta los campos sin padding, mas eficiente en memoria pero mas 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!

7542

Memoria en C/C++

## Endianess

```
1 int i = 1;
2
3
4
5
6
7
8 int j = 0xaabbccdd;
```

0	0	0	1	big
1	0	0	0	little
aa	bb	cc	dd	big
dd	cc	bb	aa	little



- El byte mas 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 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 numero en un archivo binario o enviarlo por la red a otra maquina a traves de un socket.
- Siempre hay que especificar el endianess en que se guardan/envian los datos.

7542

Memoria en C/C++

## Endianess

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

- Host to Network

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

- Network to Host

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



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

7542

Memoria en C/C++

## Segmentos de memoria

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



7542

Memoria en C/C++

## Asignación del lifetime y scope

Asignación (casi) exacta del donde.

```
1 int g = 1;
2 static int l = 1;
3 extern char e;
4
5 void Fa() { }
6 static void Fb() { }
7
8 void foo(int arg) {
9     int a = 1;
10    static int b = 1;
11
12    void * p = malloc(4);
13    free(p);
14
15    char *c = "ABC";
16    char ar[] = "ABC";
17 }
```



7542

Memoria en C/C++

## El donde importa!

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



7542

Memoria en C/C++

## Duración y visibilidad (lifetime and scope)

- Duración (lifetime): Desde que momento la variable tiene reservada memoria válida y hasta que es liberada. Determinado por el segmento de memoria que se usa.
- Visibilidad (scope): Cuando una variable se la puede acceder y cuando esta oculta.



7542

Memoria en C/C++

## Asignación del lifetime y scope

Asignación (casi) exacta del donde.

```
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
8 void foo(int arg) {  // Argumentos y retornos son del stack
9     int a = 1;       // Stack segment; scope local (func foo)
10    static int b = 1; // Data segment; scope local (func foo)
11
12    void * p = malloc(4); // p en el Stack; apunta al Heap
13    free(p);           // liberar el bloque explicitamente!!
14
15    char *c = "ABC";   // c en el Stack; apunta al Code Segment
16    char ar[] = "ABC"; // es un array con su todo en el Stack
17 } // fin del scope de foo: las variables locales son liberadas
```



7542

Memoria en C/C++

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

## 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);
```



7542

Memoria en C/C++

- 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.

## Notación

O como leer la bizzarra notación de punteros en C/C++

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



7542

Memoria en C/C++

## 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)
```



7542

Memoria en C/C++

## Punteros a funciones

Apuntando al code segment

```
1 void f() {}
2
3 void (*p)();
4 p = &f;
5
6
7 int g(char) {}
8
9 int (*p)(char);
10 p = &g;
```



7542

Memoria en C/C++

## Notación

O como leer la bizzarra notación de punteros en C/C++

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



7542

Memoria en C/C++

## Notación

O como leer la bizzarra notación de punteros en C/C++

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



7542

Memoria en C/C++

## Simplificando la Notación

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

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



7542

Memoria en C/C++

## Smash the stack

for fun and profit

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



7542

Memoria en C/C++

## Simplificando la Notación

```
1 char *X[10]; // "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
```



7542

Memoria en C/C++

## Simplificando la Notación

Si quiero una variable puntero a una función que retorna nada y recibe un puntero a una función que retorna y recibe un int?

```
1 int (*X) (int); // "X" es un puntero a  
2               // funcion  
3  
4 typedef int (*X) (int); // el tipo "X" es un puntero a  
5                       // funcion que retorna y recibe  
6                       // un int  
7  
8 void (*f) (X); // f es un puntero a funcion cuya  
9               // firma retorna nada y recibe un  
10              // X, o sea, recibe un puntero a  
11              // funcion que retorna y recibe  
12              // un int
```



7542

Memoria en C/C++

## 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 si permiten definir un limite, pero es responsabilidad del programado poner un valor coherente!

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



7542

Memoria en C/C++

## Challenge

se puede hacer que el programa imprima "You win!"?

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



## Referencias I



Bjarne Stroustrup.  
*The C++ Programming Language*.  
Addison Wesley, Third Edition.



man page: gets strcpy



googleen Insecure Programming

