De qué va esto?



- Tamaños, Alineación y Padding
- Segmentos de Memoria



- Punteros
- Typedef
- Buffer overflows





Memoria en C/C++

¹Facultad de Ingeniería

Universidad de Buenos Aires

Tamaños, Alineación y Padding

Exacta reserva de memoria

..sabiendo la arquitectura y la configuración del compilador.

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

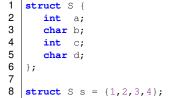
- Todo depende de la arquitectura y del compilador
- Alineación y padding
- Punteros del mismo tamaño
- Un cero como "fin de string"

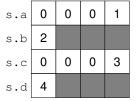
С	65			
i	0	0	0	1
S	0	4		
р	0	0	0	0
g	0	0	0	0
b	0	0	0	1
	0	0	0	2
a	65	66	0	
				C.

Memoria en C/C++

Tamaños, Alineación y Padding

Agrupación de variables





- C y C++ son lenguajes de bajo nivel para que el programador pueda tener un control absoluto de dónde y cómo se ejecuta el
- 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 rapidamente que las desalineadas.
- Como contra, la alineación despedicia espacio (padding) hay un tradeoff entre velocidad y espacio.
- El tamanño 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!!

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



Tamaños, Alineación y Padding Segmentos de Memoria

Agrupación de variables

```
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		

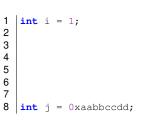
- Con el atributo especial de gcc __attribute__((packed)) el compilador empaqueta los campos sin padding, mas eficiente en memoria pero mas lento.
- Y es más lento por que para leer el atributo s.c hay que hacer 2 lecturas.
- Y cuidado, en algunas arquitecturas la lectura de atributos desalineados hace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



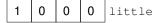
Memoria en C/C++

Tamaños, Alineación y Padding

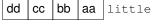
Endianess



0 0 0 1 big







- 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.



Memoria en C/C++

Tamaños, Alineación y Padding

Endianess

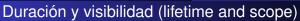
Se puede cambiar el endianess de una variable short int y int del endianess nativo o del host al endianess de la red (big endian) y viceversa:

- 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>.

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).



- 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.



Asignación del lifetime y scope

Asignación (casi) exacta del donde

static void Fb() { }

void foo(int arg) {

static int b = 1;

char *c = "ABC";

char ar[] = "ABC";

void * p = malloc(4);

int a = 1;

free(p);

static int 1 = 1;

extern char e;

void Fa() { }

int q = 1;

5

6

9

10

12

13

15

16

Tamaños, Alineación y Padding Segmentos de Memoria

Tamaños, Alineación y Padding Segmentos de Memoria

Asignación del lifetime y scope

Asignación (casi) exacta del donde

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

Memoria en C/C++

Tamaños, Alineación y Padding Segmentos de Memoria

El donde importa!

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

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

Punteros

```
// p es un puntero a int
  int *p;
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 */
2
   char buf[512];
3 write(&buf[0], 512);
```

Aritmética de punteros

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



7542 Memoria en C/C++

- La notación de array (indexado) y la aritmética de punteros son escencialmente 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.



Memoria en C/C++

Typedef

Punteros a funciones

Apuntando al code segment

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



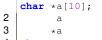
O como leer la bizarra notación de punteros en C/C++

Typedef

Notación

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

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



Notación

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





Notación

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

```
char (*f)(int)[10];
2
          f
3
                        // "f" apunta a
          * f
 4
         (*f) == X
5
6
    char X(int)[10];
7
    char X(int)
                        // X es la firma de una funcion,
8
                        // asi 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
    char (*f)(int)[10]; // f es un array de 10 punteros a funcion,
14
                        // que reciben un int y retornan un chars
15
16
```

Simplificando la Notación

```
char *X[10];
                                     "X" es un array de
                          // 10 punteros a char
2
3
4
                          // el tipo "X" es un array de
   typedef char *X[10];
5
                          // 10 punteros a char
```



Memoria en C/C++

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?

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

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?

```
int (*X)(int);
                                         "X" es un puntero a
                             // funcion
2
3
 4
    typedef int (*X)(int);
                             // el tipo "X" es un puntero a
5
                             // funcion que retorna y recibe
6
                             // un int
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
                           // funcion que retorna y recibe
11
12
                           // un int
```



Smash the stack

for fun and profit

```
#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
```

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++

7542 Memoria en C/C++

Buffer overflows Referencias I Challenge se puede hacer que el programa imprima "You win!"? 1 | #include <stdio.h> // compilar con el flag -fno-stack-protect 3 int main(int argc, char *argv[]) { Bjarne Stroustrup. 4 int cookie = 0; 5 char buf[10]; The C++ Programming Language. 6 Addison Wesley, Third Edition. 7 printf("buf:_%08x_cookie:_%08x\n", &buf, &cookie); 8 gets(buf); man page: gets strcpy 9 googleen Insecure Programming 10 **if** (cookie == 0x41424344) { 11 printf("You_loose!\n"); 12 13 14 return 0;

Memoria en C/C++ 7542 Memoria

15 | } // Insecure Programming