Memoria en C/C++

Di Paola Martín

martinp.dipaola <at> gmail.com

Facultad de Ingeniería
Universidad de Buenos Aires

De qué va esto?

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

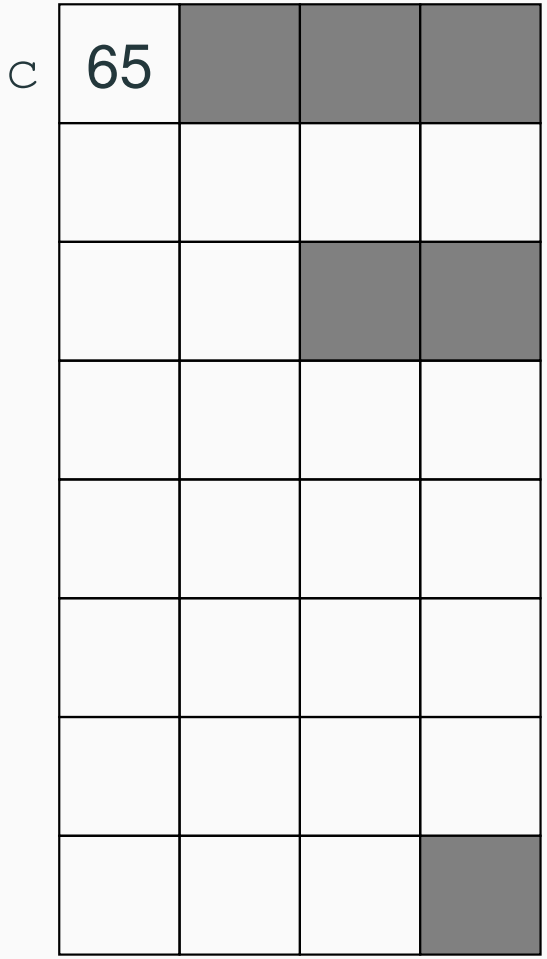
Memoria

Tamaños, Alineación y Padding

Exacta reserva de memoria

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

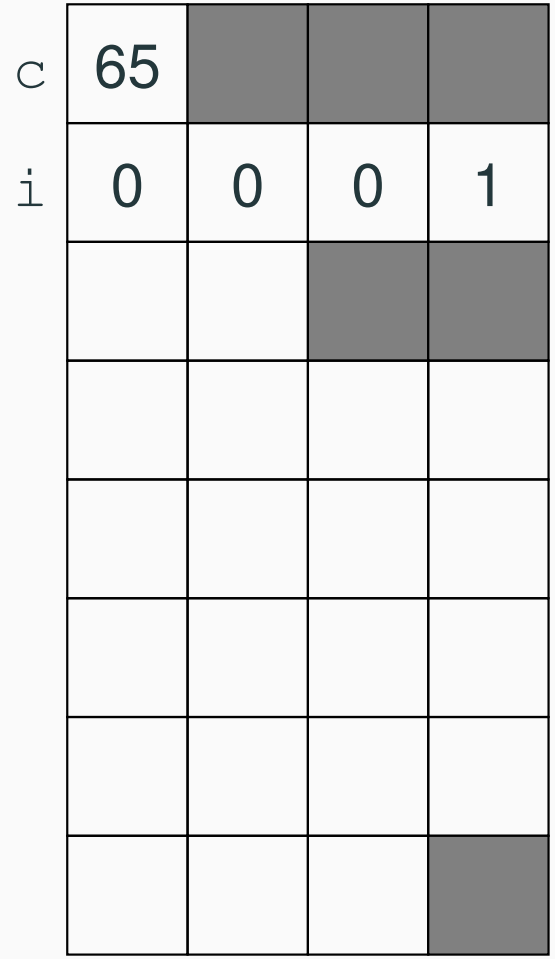
- Todo depende de la arquitectura y del compilador



Exacta reserva de memoria

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

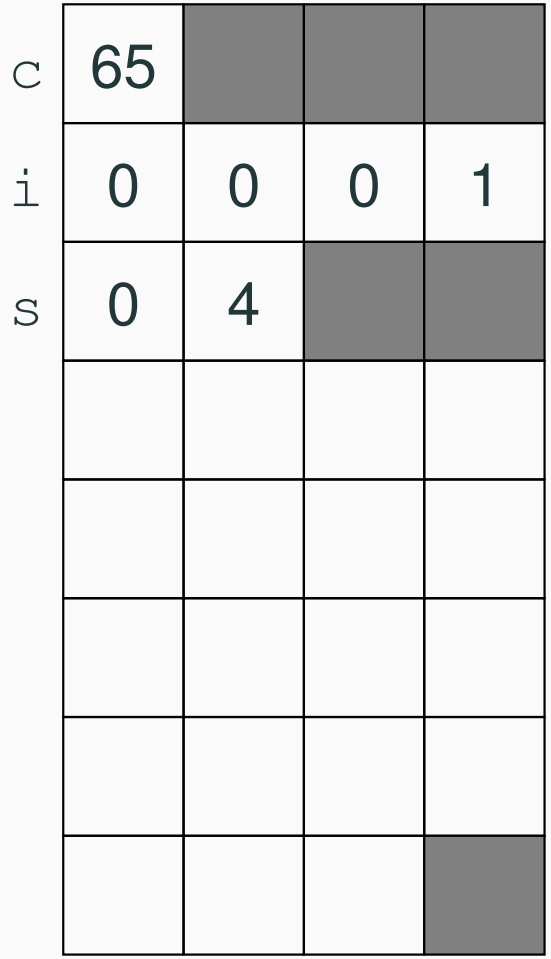
- Todo depende de la arquitectura y del compilador
- Alineación y padding



Exacta reserva de memoria

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

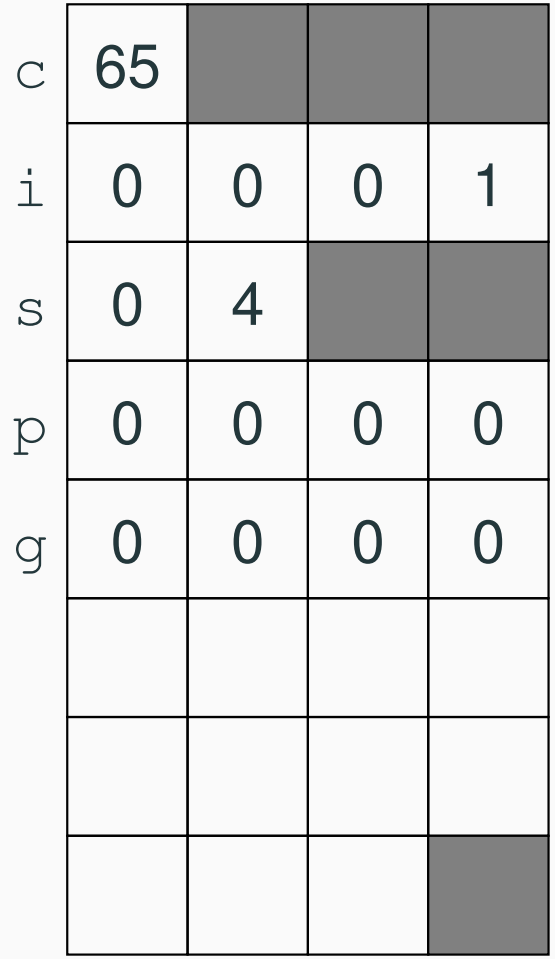
- Todo depende de la arquitectura y del compilador
- Alineación y padding



Exacta reserva de memoria

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

- Todo depende de la arquitectura y del compilador
- Alineación y padding
- Punteros del mismo tamaño



Exacta reserva de memoria

```
  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};
  char a[] = "AB";
```

- Todo depende de la arquitectura y del compilador
- Alineación y padding
- Punteros del mismo tamaño

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

Exacta reserva de memoria

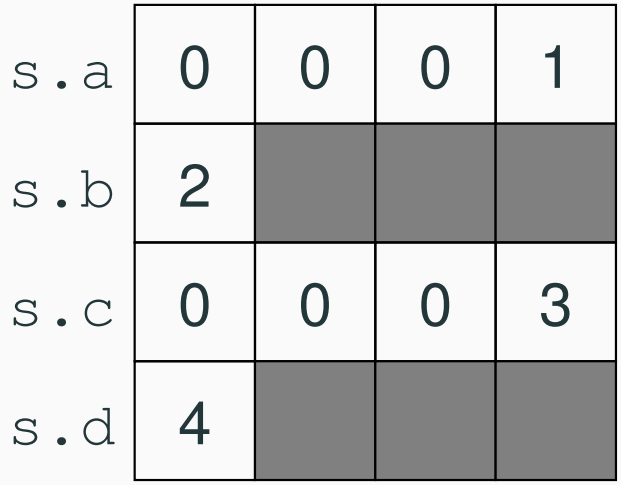
```
   char c = 'A';
2  int i = 1;
3  short int s = 4;
4  char *p = 0;
5  int *g = 0;
6  int b[2] = {1, 2};
7  char a[] = "AB";
```

- 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	

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};
```



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		

Endianess: representación en memoria

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

```
((unsigned char*)&i)
   == {0, 0, 0, 1}
```

0	0	0	1

big

```
((unsigned char*)&i)
   == {1, 0, 0, 0}
```

1	0	0	0

little

```
((unsigned char*)&j)
   == {0, 0, 3, 255}
```

0	0	3	ff

big

```
((unsigned char*)&j)
   == {255, 3, 0, 0}
```

ff	3	0	0

little

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)
```

Memoria

Segmentos de Memoria

Segmentos de memoria

- Code segment: de solo lectura y ejecutable, a donde va el código y las constantes.

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.

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.

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).

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.

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.

Asignación del lifetime y scope

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

void Fa() { }
static void Fb() { }
void Fc();

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

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

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

Asignación del lifetime y scope

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

void Fa() { }
static void Fb() { }
void Fc();

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

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

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

Asignación del lifetime y scope

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

void Fa() { }
static void Fb() { }
void Fc();

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

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

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

Asignación del lifetime y scope

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

void Fa() { }
static void Fb() { }
void Fc();

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

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

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

Asignación del lifetime y scope

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

void Fa() { }
static void Fb() { }
void Fc();

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

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

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

Asignación del lifetime y scope

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

void Fa() { }
static void Fb() { }
void Fc();

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

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

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

Asignación del lifetime y scope

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

void Fa() { }
static void Fb() { }
void Fc();

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

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

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

Asignación del lifetime y scope

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

void Fa() { }
static void Fb() { }
void Fc();

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

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

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

Asignación del lifetime y scope

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

void Fa() { }
static void Fb() { }
void Fc();

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

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

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

Asignación del lifetime y scope

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

void Fa() { }
static void Fb() { }
void Fc();

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

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

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

Asignación del lifetime y scope

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

void Fa() { }
static void Fb() { }
void Fc();

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

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

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

Asignación del lifetime y scope

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

void Fa() { }
static void Fb() { }
void Fc();

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

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

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

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
```

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  }
```

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);
```

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)
```

Punteros a funciones (al code segment)

```
1 int g(char) {}
2
3 int (*p)(char);
4 p = &g;
```

Punteros a funciones (al code segment)

```c
int g(char) {}

int (*p)(char);
p = &g;
```

```c
#include <stdlib.h>
void qsort(void *base,
           size_t nmemb,
           size_t size,

           int (*cmp)(const void *, const void *)
          );

int cmp_personas(const void* a, const void* b) {
    struct Persona *pa = (struct Persona*)a;
    struct Persona *pb = (struct Persona*)b;

    return pa->edad - pb->edad;
}
```

Punteros

Typedef

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

```
/* Ejemplo 1 */
char *a[10];
      a          // "a"
     *a          // "a" apunta a
char *a          // "a" apunta a char
char *a[10];     // "a" apunta a char (10 de esos)

char *a[10];     // "a" es un array de 10 de esos, o sea
                 // "a" es un array de 10 punteros a char
```

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

```
   /* 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++

```
/* Ejemplo 2 */
char (*c)[10];
       c        // "c"
      *c        // "c" apunta a
     (*c) == X // llamemos "X" a (*c) temporalmente

char X[10];
char X[10];    // "X" es un char (10 de esos)

char X[10];    // "X" es un array de 10 char
char (*c)[10]; // "c" apunta a un array de 10 char
```

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

```
/* Ejemplo 2 */
char (*c)[10];
       c        // "c"
      *c        // "c" apunta a
     (*c) == X // llamemos "X" a (*c) temporalmente

char X[10];
char X[10];    // "X" es un char (10 de esos)

char X[10];    // "X" es un array de 10 char
char (*c)[10]; // "c" apunta a un array de 10 char
```

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

```
   /* 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
```

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

```
   /* 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++

```
/* Ejemplo 3: modo dios */
char (*f)(int)[10];
       f              // "f"
      *f              // "f" apunta a
     (*f) == X

char X(int)[10];
char X(int)           // es la firma de una funcion,
                      // asi que vuelvo un paso para atras
char (*f)(int)        // entonces esto es un puntero a funcion
                      // cuya firma recibe un int y retorna
                      // un char

char (*f)(int)[10]; // puntero a funcion, 10 de esos
char (*f)(int)[10]; // f es un array de 10 punteros a funcion,
                    // que reciben un int y retornan un chars
```

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

```
/* Ejemplo 3: modo dios */
char (*f)(int)[10];
       f              // "f"
      *f              // "f" apunta a
     (*f) == X

char X(int)[10];
char X(int)           // es la firma de una funcion,
                      // asi que vuelvo un paso para atras
char (*f)(int)        // entonces esto es un puntero a funcion
                      // cuya firma recibe un int y retorna
                      // un char

char (*f)(int)[10]; // puntero a funcion, 10 de esos
char (*f)(int)[10]; // f es un array de 10 punteros a funcion,
                    // que reciben un int y retornan un chars
```

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

```
   /* 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];
   char X(int)            // es la firma de una funcion,
                          // asi que vuelvo un paso para atras
   char (*f)(int)         // entonces esto es un puntero a funcion
                          // cuya firma recibe un int y retorna
                          // un char

   char (*f)(int)[10]; // puntero a funcion, 10 de esos
   char (*f)(int)[10]; // f es un array de 10 punteros a funcion,
                       // que reciben un int y retornan un chars
```

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

```
/* Ejemplo 3: modo dios */
char (*f)(int)[10];
       f              // "f"
      *f              // "f" apunta a
     (*f) == X

char X(int)[10];
char X(int)           // es la firma de una funcion,
                      // asi que vuelvo un paso para atras
char (*f)(int)        // entonces esto es un puntero a funcion
                      // cuya firma recibe un int y retorna
                      // un char

char (*f)(int)[10]; // puntero a funcion, 10 de esos
char (*f)(int)[10]; // f es un array de 10 punteros a funcion,
                    // que reciben un int y retornan un chars
```

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

```
   /* 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
```

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

Buffer overflows

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
```

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);
```

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

```c
// compilar con el flag -fno-stack-protector
#include <stdio.h>

int main(int argc, char *argv[]) {
    int cookie = 0;
    char buf[10];

    printf("buf: %08x cookie: %08x\n", buf, &cookie);
    gets(buf);

    if (cookie == 0x41424344) {
        printf("You loose!\n");
    }

    return 0;
} // Insecure Programming
```

Appendix

Referencias

Referencias I


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


man page: gets strcpy htons qsort


Insecure Programming