Memoria en C/C++

[1] Facultad de Ingeniería
Universidad de Buenos Aires

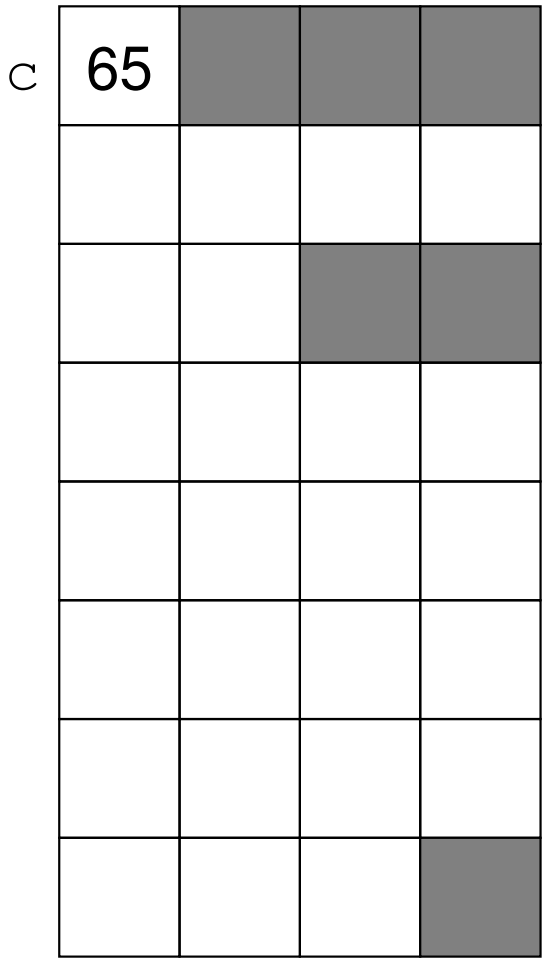
De qué va esto?

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;
int *g = 0;
int b[2] = {1, 2};
char a[] = "AB";
```

- Todo depende de la arquitectura y del compilador

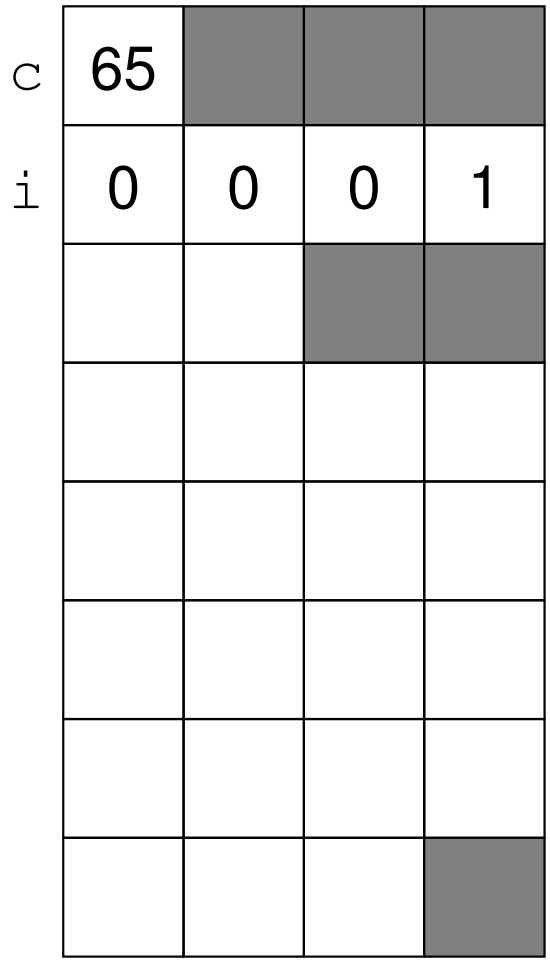


Exacta reserva de memoria

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

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

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

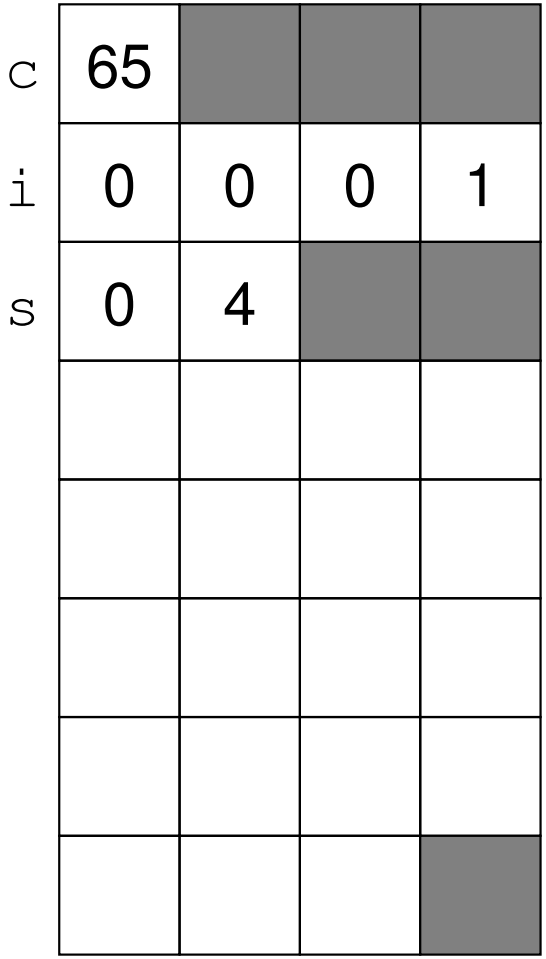


Exacta reserva de memoria

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

```
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

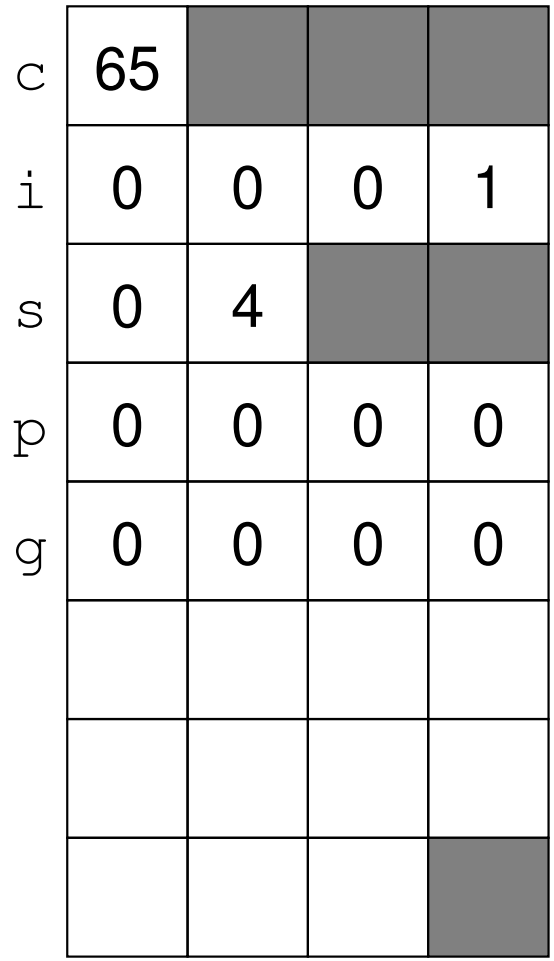


Exacta reserva de memoria

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

```
   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

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

```
   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

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

```
   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"

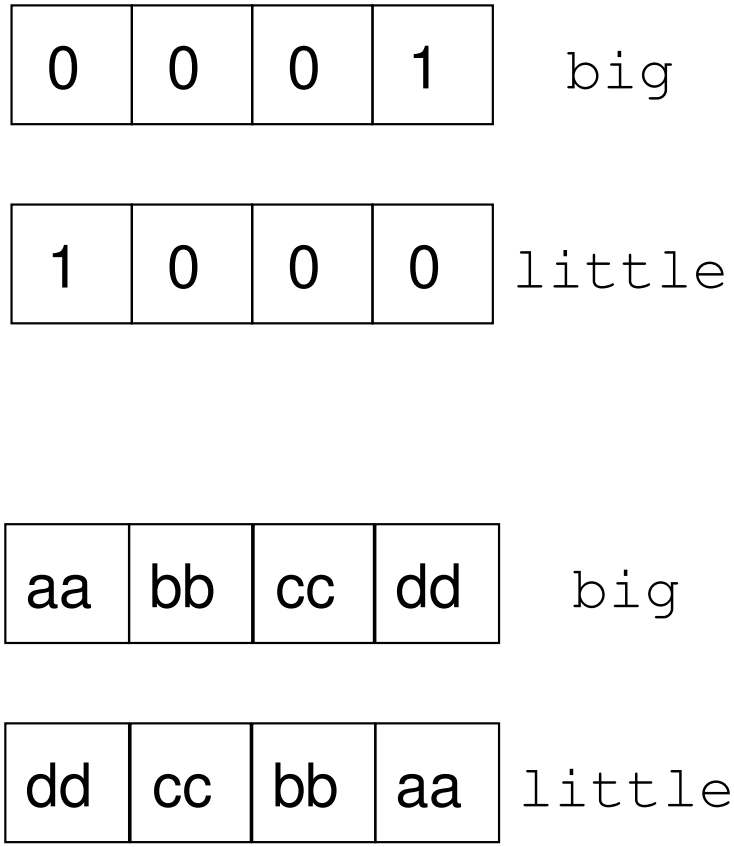
c	65			
i	0	0	0	1
s	0	4		
p	0	0	0	0
g	0	0	0	0
b	0	0	0	1
	0	0	0	2
a	65	66	0	

Agrupación de variables

```
1 struct S {
2     int a;      // 4 bytes
3     char b;     // 1 byte
4 }
5
6 struct S s;         // (5 bytes + padding) = 8 bytes
```

Endianess

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



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 de 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 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.

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 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.

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.

Asignación del lifetime y scope

Asignación (casi) exacta del donde.

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

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

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

Asignación (casi) exacta del donde.

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

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

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

Asignación (casi) exacta del donde.

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

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

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

Asignación (casi) exacta del donde.

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

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

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

Asignación (casi) exacta del donde.

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

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

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

Asignación (casi) exacta del donde.

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

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

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

Asignación (casi) exacta del donde.

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

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

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

Asignación (casi) exacta del donde.

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

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

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

Asignación (casi) exacta del donde.

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

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

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

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

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

Asignación del lifetime y scope

Asignación (casi) exacta del donde.

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

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

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

Asignación (casi) exacta del donde.

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

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

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

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
```

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

```
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, 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

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;
```

Notación

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

```
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
```

Notación

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

```
   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
```

Notación

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

```
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
```

Notación

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

```
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
```

Notación

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

```
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
```

Notación

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

```
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
```

Notación

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

```
char (*f)(int)[10];
       f              // "f"
      *f              // "f" apunta a
     (*f) == X

char X(int)[10];
char X(int)           // X 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
```

Notación

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

```
char (*f)(int)[10];
2         f                 // "f"
3        *f                 // "f" apunta a
4        (*f) == X
5

char X(int)[10];
char X(int)                 // X 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
```

Notación

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

```
char (*f)(int)[10];
      f              // "f"
     *f              // "f" apunta a
     (*f) == X

char X(int)[10];
char X(int)          // X 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
```

Notación

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

```
char (*f)(int)[10];
       f              // "f"
      *f              // "f" apunta a
     (*f) == X
char X(int)[10];
char X(int)           // X 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
```

Notación

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

```
char (*f)(int)[10];
       f              // "f"
      *f              // "f" apunta a
     (*f) == X

char X(int)[10];
char X(int)           // X 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

```

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
```

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
```

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
```

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
```

Buffer overflow

- Funciones inseguras que no ponen un limite 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);
```

Challenge

compilar con el flag -fno-stack-protect

```
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 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