Lenguaje de programación C

75.41/95.15 - Algoritmos y Programación II
Cátedra Wachenchauzer

Sebastián Santisi
12/03/2016

Lenguaje C

- Dennis M. Ritchie, 1972
- Mainframes
- Bajo nivel
- Pensado para escribir un sistema operativo
- ANSI-89, ANSI-99

Lenguaje C

- C es un lenguaje:
  - Compilado
  - Procedural
  - Tipado
  - Memoria estática (dinámica administrada por el programador)

¡Hola mundo!

```
#include <stdio.h>

/* Implementa el hola mundo en C */

int main(void) {
    printf(“Hola mundo\n”);
    return 0;
}
```

Tipos en C

- Dos grandes grupos:
  - Enteros: char, short, int, long, etc.
  - Flotantes: float, double, etc.
- Enteros:
  - El tamaño depende de la plataforma y del compilador
  - Admiten modificadores unsigned
- Flotantes:
  - IEEE754
  - Representación en mantisa más exponente
  - float: ~8 dígitos de representación de mantisa en decimal
  - double: ~16 dígitos

Tipos enteros (ej. GCC 32bits)

Tipo	Bits	Bytes	Desde	Hasta
signed char	8	1	-128	127
unsigned char	8	1	0	255
short	16	2	-32.768	32.767
unsigned short	16	2	0	65.535
int	32	4	-2.147.483.648	2.147.483.647
unsigned int	32	4	0	4.294.967.295
long	32	4	-2.147.483.648	2.147.483.647
unsigned long	32	4	0	4.294.967.295

- El tamaño no está fijado por el estándar salvo el de char: sizeof(char) = 1
- Después: sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long), etc.

Ejemplos

- `double frecuencia_angular;`
- `unsigned int cantidad_asistentes;`
- `unsigned char edad;`
- `long centavos;`
- `float temperatura;`
- Lo anterior son ejemplos de “declaraciones” de variables. Declarar una variable implica reservar memoria para ella. Las variables se declaran una única vez. Declarar ≠ definir (asignarle un valor)

Literales

- Los literales, como todo, son tipados en C
- Todos estos representan al 0 con un tipo distinto:

```
0 (int), 0U (unsigned int), 0L (long), '\0' (char), NULL (void*), 0.0 (double), 0.0F (float)
```

- Internamente pueden tener una representación diferente

Funciones

- Tienen tipos, como todo, tanto de las cosas que entran como de las que devuelven

```
int sumar(int a, int b) {
   return a + b;
}

double x;
x = sumar(4.8, 5.8); // x = 9.0
```

¡Atenti Pascaleros!

- El operador de asignación es =
- La comparación en C se hace con ==
- Importante: Compilar SIEMPRE con advertencias (y prestarles atención a las mismas), el compilador ve estas cosas y avisa.

Prototipos

- El compilador de C funciona en una sola pasada
- Para utilizar una función no hace falta que la misma esté definida previamente, pero sí hace falta que al menos esté declarada
- Para declarar las funciones se utilizan los prototipos:

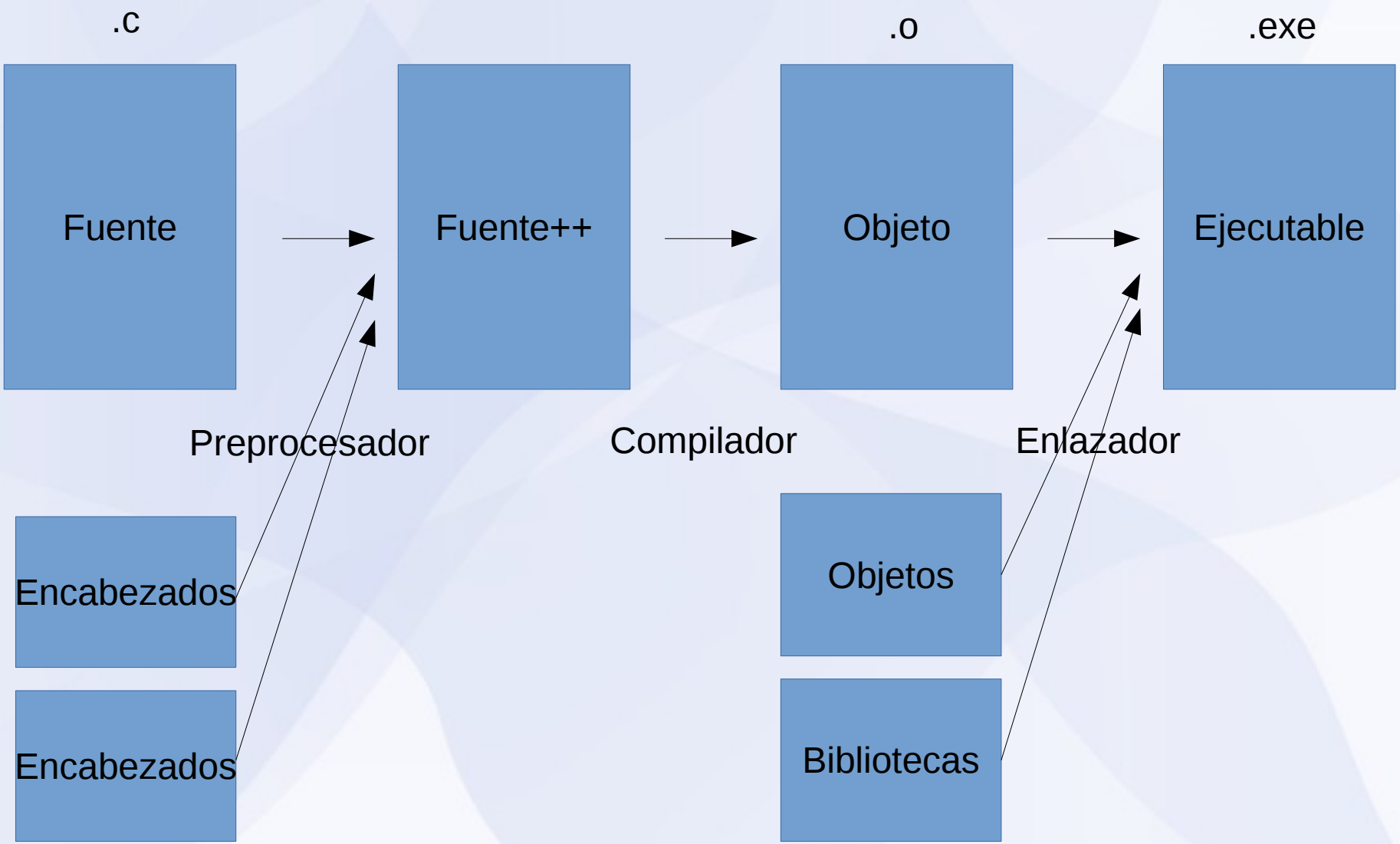
```
void imprimir(int);
```

- Le informa al compilador que imprimir() es una función que recibe un entero y no devuelve nada. El compilador con esto puede ajustar las llamadas a la función aunque no conozca su implementación (ej. “imprimir(3.14)”)

Conversiones de tipos

- Truncado: Cuando un tipo “más grande” se mete en uno “más chico” (un int en un char, un float en un int, un signado en un unsigned, etc.). Puede haber pérdida de información.
- Promoción: Cuando un tipo “más chico” se mete en uno “más grande” (un short en un int, un int en un double, etc.). No debería haber pérdida, pero puede haberla
- Cuando una expresión se asigna a un tipo se trunca/promueve para ajustarse al tipo de destino
- Cuando se intenta operar entre dos operandos distintos el “más chico” se promueve al tipo del “más grande”
- Se puede convertir explícitamente un tipo (casting) anteponiendo el nombre del tipo destino entre paréntesis (ej. “(float)a / b”)

Proceso de compilación



¡Hola mundo! (ahora sí)

```
#include <stdio.h>

/* Implementa el hola mundo en C */

int main(void) {
    printf(“Hola mundo\n”);
    return 0;
}
```

Desmenuzando el hola mundo

- Las sentencias al preprocesador comienzan con #
- #include trae un archivo de encabezados que contienen “la documentación” de las funciones, tipos, etc. (Pascal: uses)
- stdio.h: standard input/output, encabezados de entrada-salida.
- main(): Punto de entrada
- printf(): Función para imprimir
- return: Al sistema operativo hay que devolverle algo

Generalidades de C

- Los bloques se delimitan con llaves (Pascal: Olvídense del begin, end;, end., etc.)
- Las sentencias se delimitan con punto y coma
- Todas las sentencias deben estar adentro de funciones
- Debe haber uno y sólo un punto de entrada por programa
- Los comentarios se delimitan entre /* … */ o (sólo en ANSI C99) con // … para final de línea
- (Sólo en ANSI C89:) Las declaraciones de variables deben estar al comienzo de la función
- El compilador trabaja de arriba hacia abajo en una sola pasada

Variables, constantes y etiquetas

- Las variables pueden ser constantes:

```
const float pi = 3.14159;
```

- Es una variable como las demás, pero no puede cambiarse su valor a posteriori
- También se pueden crear etiquetas

```
#define PI 3.14159
```

- Directiva del preprocesador
- Hace copy-paste de la expresión donde sea que la encuentre
- No consume memoria, no tiene un tipo asociado, etc.
- Convención de nombres: MAYUSCULAS

Redefiniciones

- Se pueden renombrar tipos.
- La sintaxis es como definir una variable pero antedeciendo la palabra typedef lo que define un tipo:

```
typedef unsigned char edad_t;

...

edad_t edad_juan = 23;
```

Enumerativos

- Es un tipo entero que se usa cuando se quiere representar un conjunto con categorías acotadas:

```
typedef enum {LUNES, MARTES, MIERCOLES,
JUEVES, VIERNES, SABADO, DOMINGO} dia_t;

...

dia_t clase_dia = LUNES;
```

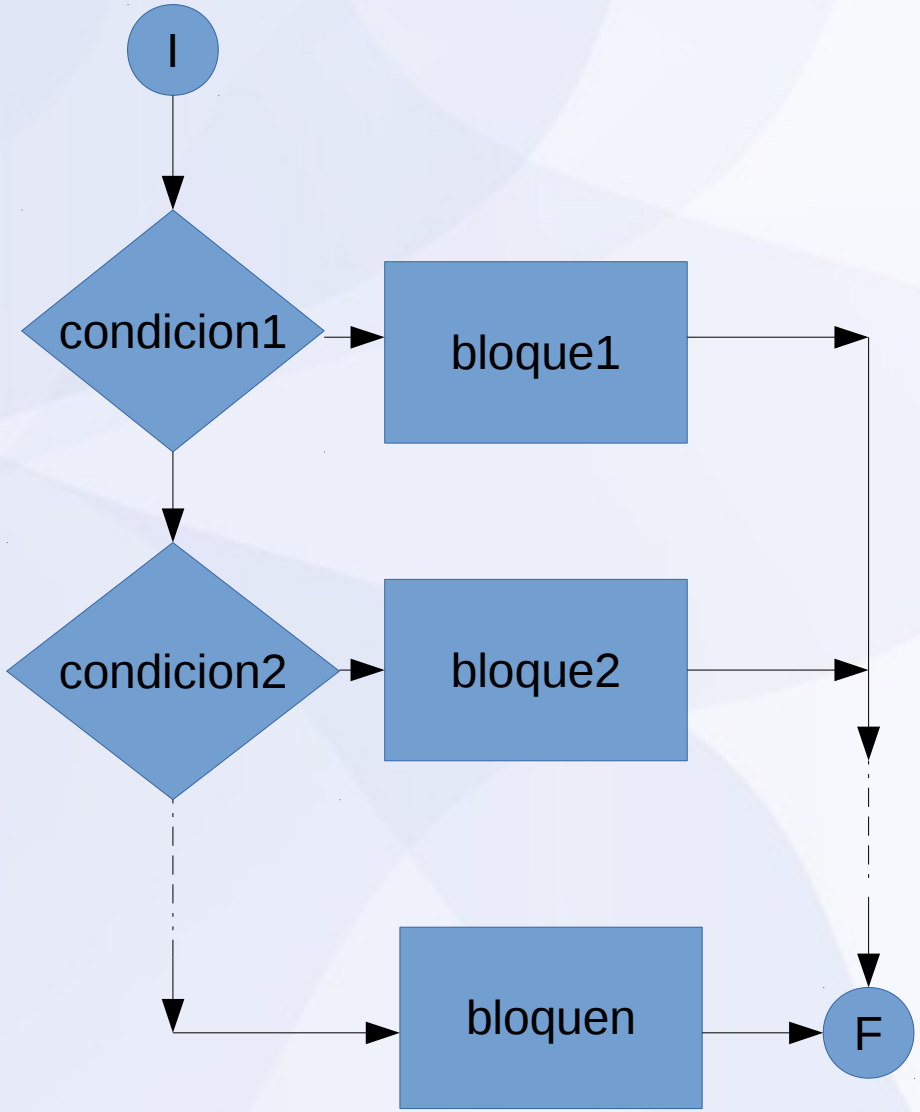
- Internamente, las etiquetas se definen como si se hubieran hecho muchos #define LUNES 0, #define MARTES 1, etc.
- No se puede transformar el contenido de clase_dia en una cadena “LUNES”

Estructuras de control

- if
- while
- for
- do-while
- switch

if

```
if(condicion1) {
    bloque1
}
else if(condicion2) {
    bloque2
}
...
else {
    bloquen
}
```



Bloques

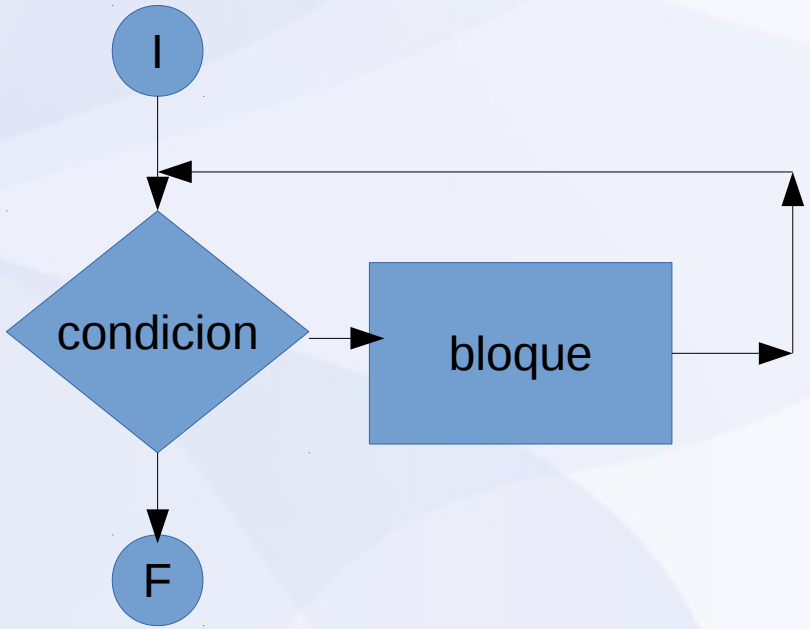
- Se inician y terminan con llaves... a menos que sean una única instrucción
- Es buena práctica indentar los bloques. El lenguaje no obliga a ello (los docentes sí)
- Las condiciones pueden ser cualquier condición booleana... ¡pero en C no existe el tipo bool!

Booleanos

- En C todos los tipos enteros son booleanos:
  - 0: Falso
  - distinto de cero: Verdadero
- Los operadores booleanos operan sobre los distintos tipos y devuelven un entero según esa lógica
- ==: Igualdad
- !=: Diferencia
- &&: And
- ||: Or
- !: Not
- <, >, <=, >=

while

```
while(condicion) {
    bloque
}
```

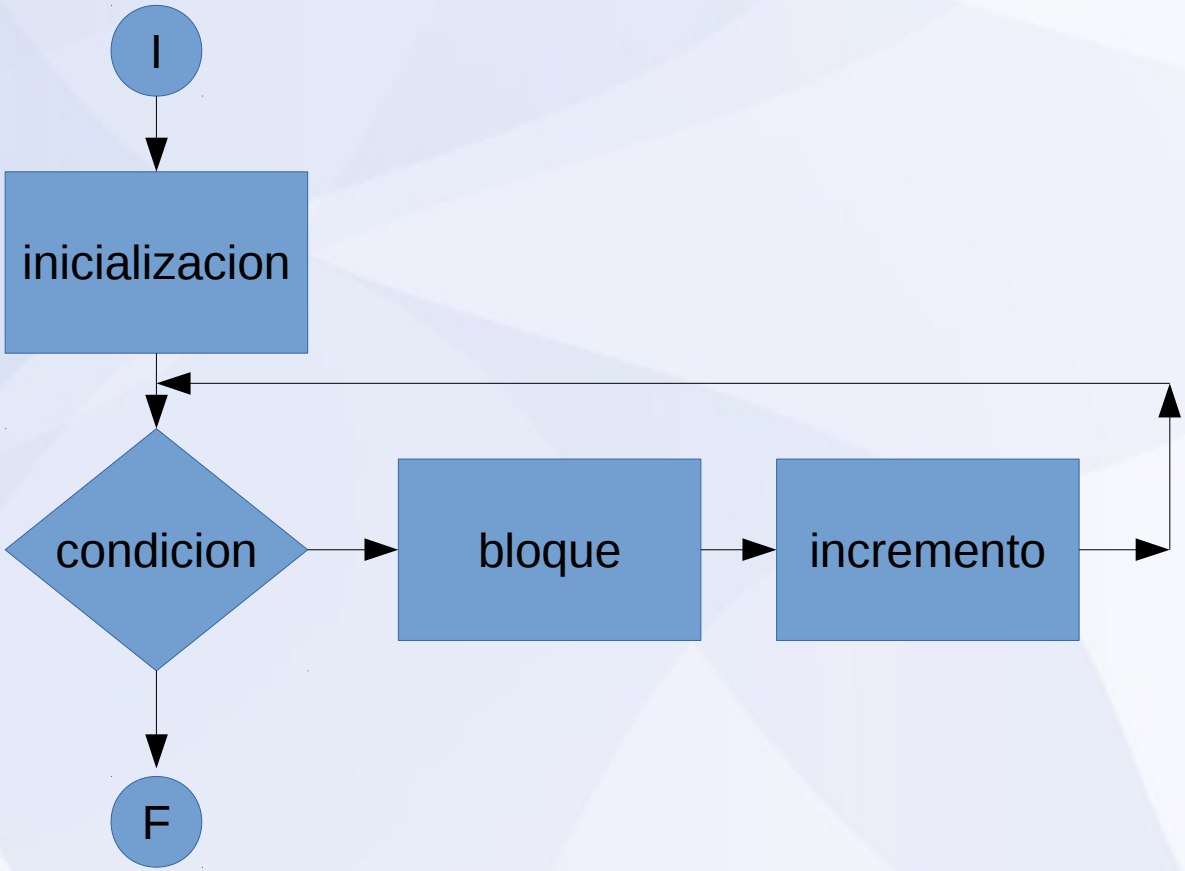


Corte

- En C los bloques siguen su curso habitual... a menos que se topen con una instrucción de corte
- break: Interrumpe el bloque de control en el que está la instrucción.
- continue: Pasa al ciclo siguiente sin seguir ejecutando lo que venga después.
- return: Si bien return no es una instrucción de corte, con return podemos salir de cualquier función en el momento en el que está la instrucción, por lo que si hay un ciclo, actúa como instrucción de corte.

for

```
for(inic; cond; incr) {
    bloque
}
```



for

- La inicialización, la condición de corte y el incremento no tienen por qué tener nada entre sí. No aplican sobre una única variable.
- Más allá de lo anterior: Se espera que lo que uno ponga dentro de estos tres “casilleros” represente semánticamente esas condiciones para el bucle.
- Puede introducirse más de una inicialización o de un incremento separando las sentencias por coma (,).

for (ejemplo)

```
int i;

for(i = 0; i < 10; i++) {
    printf(“%d\n”, i);
}
```

C++, C--

- En C cada operador tiene su contrapartida +op.
  - a = a + b; => a += b;
  - a = a / b; => a /= b;
- Los operadores ++ y -- significan += 1 y -=1 respectivamente.
- Pre: Si el operador está a izquierda se aplica antes de evaluar la expresión. Post: Si el operador está a derecha se aplica después.
  - int a = 5, b;
  - b = ++a;
  - a == 6, b == 6

int a = 5, b;

b = a++;

a = 6, b == 5

printf()

- Sirve para imprimir con formato.
- Recibe una cadena de formato y variables a formatear
- Ejemplo:

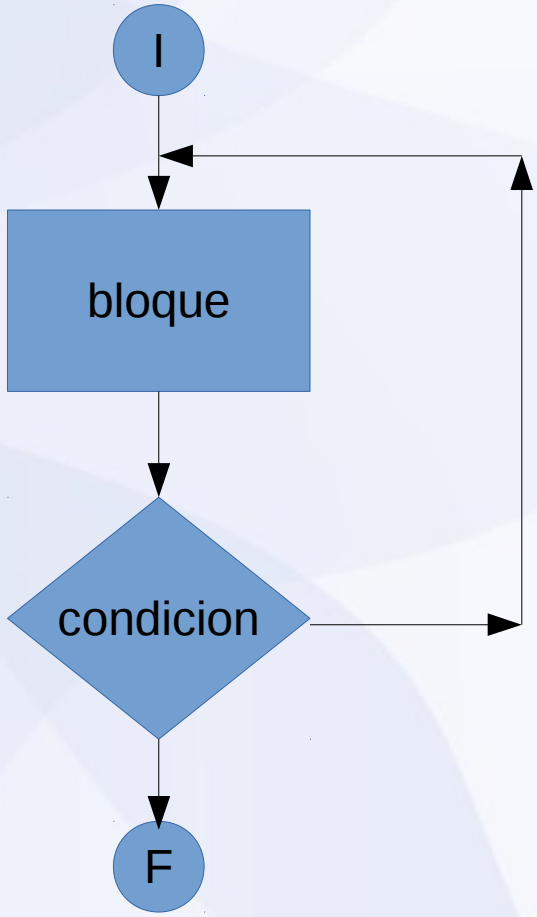
```
printf(“%d %f %c $%.2f”, 64, 3.14, 64, 1.2);
```

Imprime:

```
64 3.140000 @ $1.20
```

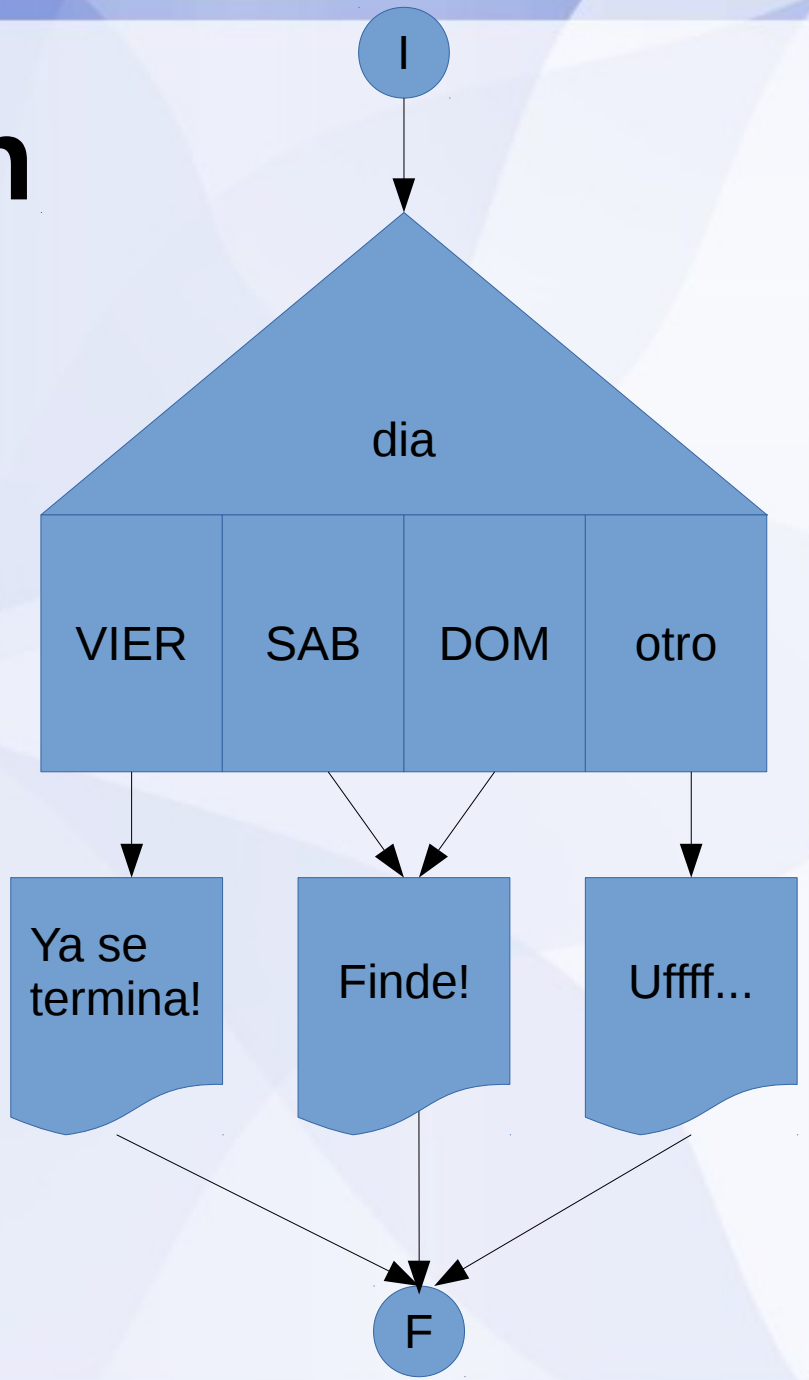
do-while

```
do {
    bloque
} while(condicion);
```



switch

```
dia_t dia;
...
switch(dia) {
   case VIERNES:
      printf(“Ya se termina!\n”);
      break;
   case SABADO:
   case DOMINGO:
      printf(“Finde!\n”);
      break;
   default:
      printf(“Uffff...\n”);
      break;
}
```



switch (cont.)

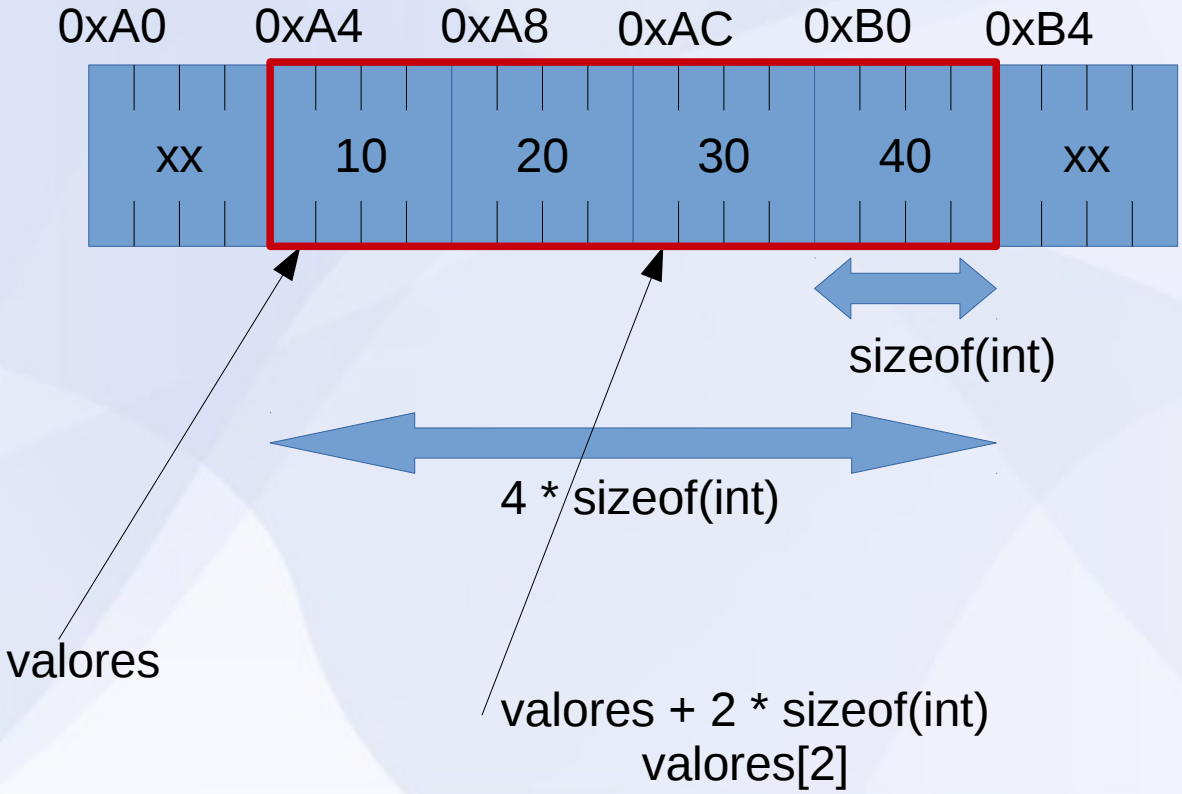
- Funciona solamente sobre variables enteras
- Testea valores exactos (no rangos)
- El break tiene un sentido diferente que en iteraciones (¡el switch no es un iterador!)
- Una vez que se entró por una etiqueta se sigue ejecutando hasta encontrar un break.
- Si omito el break entre dos etiquetas va a seguir ejecutando de una a la siguiente.

Vectores

- Son un paquete de N variables del mismo tipo
- Ocupan un espacio de N * sizeof(tipo) espacios consecutivos en memoria
- Se numeran del 0 al N-1
- C no chequea el rango de acceso, no hay que leer/escribir fuera del tamaño válido

Vectores (cont.)

```
int valores[4] = {10, 20, 32, 40};
valores[2] = 30;
valores[4] = ERROR!;
```



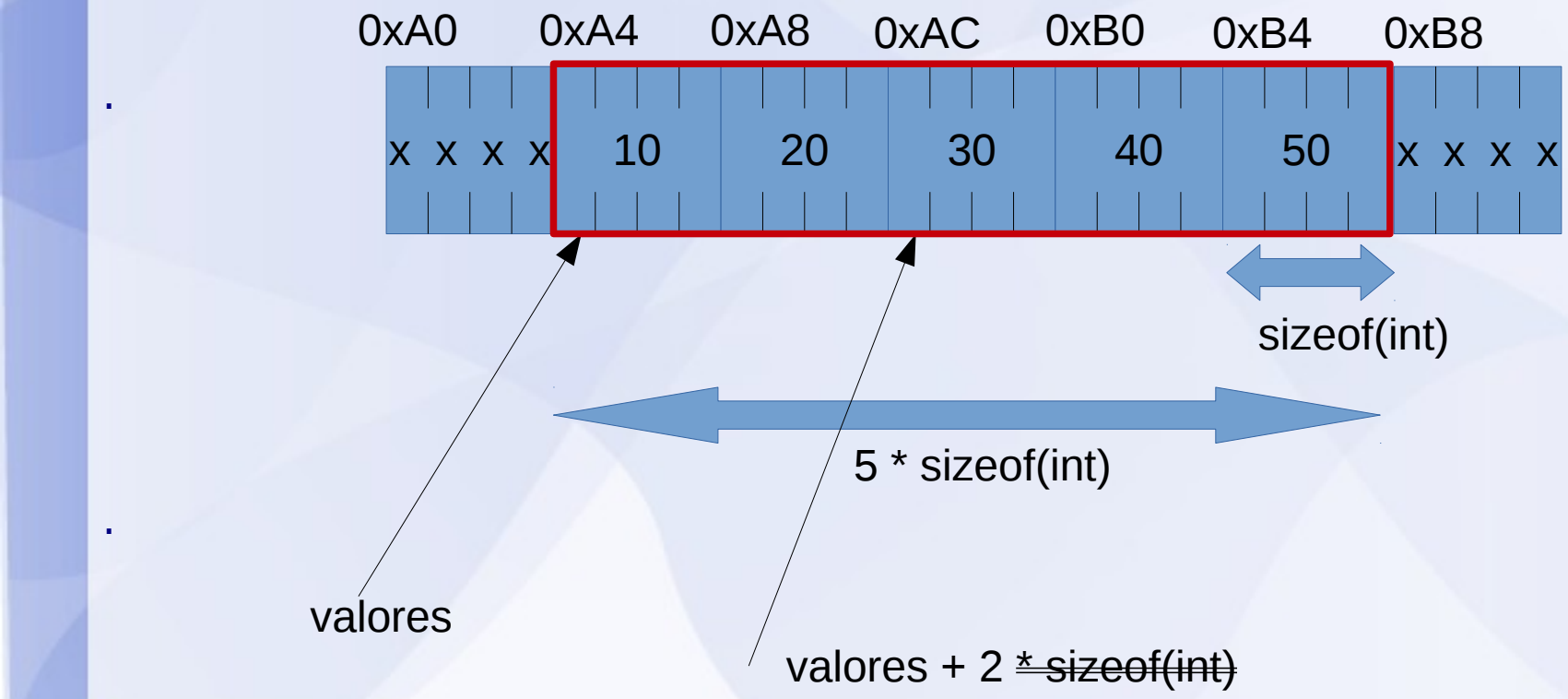
.

Memoria

- La memoria es un continuo de bytes numerados
- La memoria no tiene “tipo”, el tipo es una interpretación que hace el lenguaje sobre los bits de un determinado paquete de bytes
- Internamente, cuando se accede a un dato (cualesquiera) el compilador sabe la posición (dirección) de memoria en la cual “vive” ese dato, sabe el tipo de ese dato y en base a eso interpreta el significado de la memoria

Memoria (cont.)

```
int valores[] = {10, 20, 30, 40, 50};
printf("%u, %u", sizeof(int), sizeof(valores)); // 4, 20  (20 / 4 = 5)
printf("%p", valores); // 0xA4
printf("%p", valores + 2); // 0xAC (0xA4 + 2 * sizeof(int) = 0xAC)
```



Vectores y funciones

```
void imprimir(const int v[], int s) {
   for(int i = 0; i < s; i++)
      printf("%d\n", v[i]);
}
...
int valores[CANT_VAL] = {10, 20, 30, 40, 50};
imprimir(valores, CANT_VAL);
```

- ¿Cómo resuelve el compilador esta llamada?
- ¿Qué pasa si CANT_VAL es grande?

Vectores y funciones (cont.)

- Los vectores en C no se copian por valor en una llamada a función
- Ya habíamos visto que “valores” no es el paquete “10, 20, ..., 50” sino que es 0xA4, la dirección de memoria donde empieza el vector
- Entonces, la función no recibe “el vector”. Recibe la dirección de memoria en la cual comienza el vector.
- Dado que el primer parámetro de la función es sólo una dirección de memoria, no puede conocer cuánto mide el vector a menos que lo reciba como parámetro
- Como lo que recibe es la referencia a la memoria donde “vive” el vector, podría modificar el contenido del mismo

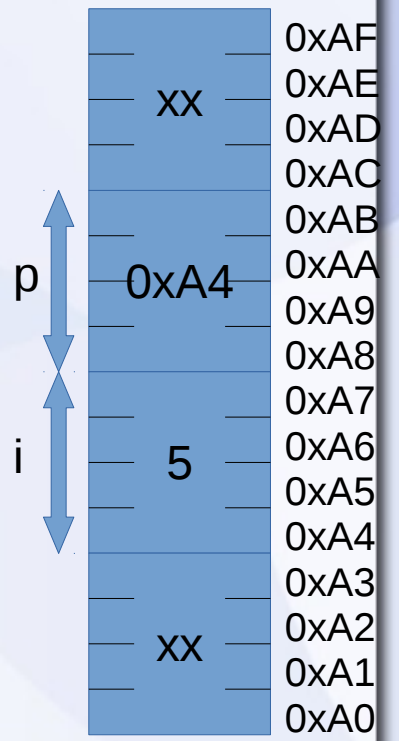
Punteros

- Un puntero es una variable que puede almacenar una dirección de memoria
- Los punteros se declaran según el tipo que van a direccionar no porque la memoria de diferentes tipos sea diferente sino para saber cómo acceder a esa memoria y su dimensión

Punteros (cont.)

```
int i = 5;
int *p;
p = &i;
printf(“%p, %p, %p”, &i, p, p+1); // 0xA4, 0xA4, 0xA8
printf(“%i, %i”, i, *p); // 5, 5
*p = 4;
printf(“%i, %i”, i, *p); // 4, 4
printf(“%p”, &p); // 0xA8
```

.

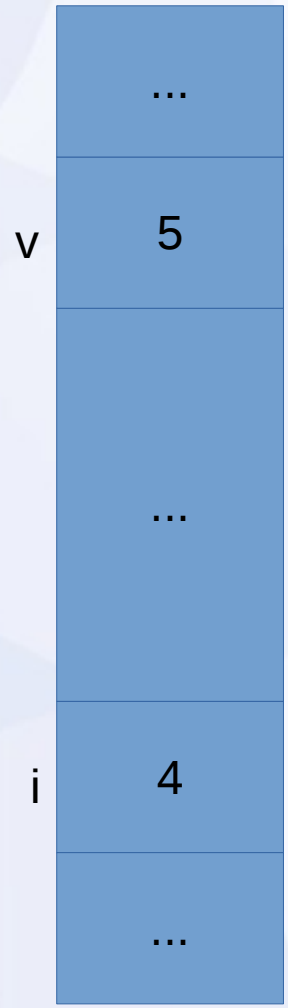


Punteros (cont.)

- tipo *var: Declara una variable de tipo “puntero a tipo”
- &var: Operador de dirección, devuelve la dirección de memoria en la que vive var
- *var: Operador de indirección, devuelve la variable que está en la memoria referenciada por var (si var es de tipo “puntero a X”, *var es de tipo X)
- tipo *y = &x: Se dice que “y apunta a x” (*y: “lo apuntado por y”)
- valores[2]: Esta expresión es equivalente a “*(valores + 2)”
- Para vectores siempre vale: vector == &vector[0] (por eso es válido lo anterior)
- Notar que mediante un puntero se puede acceder a memoria consecutiva de la que se apunta

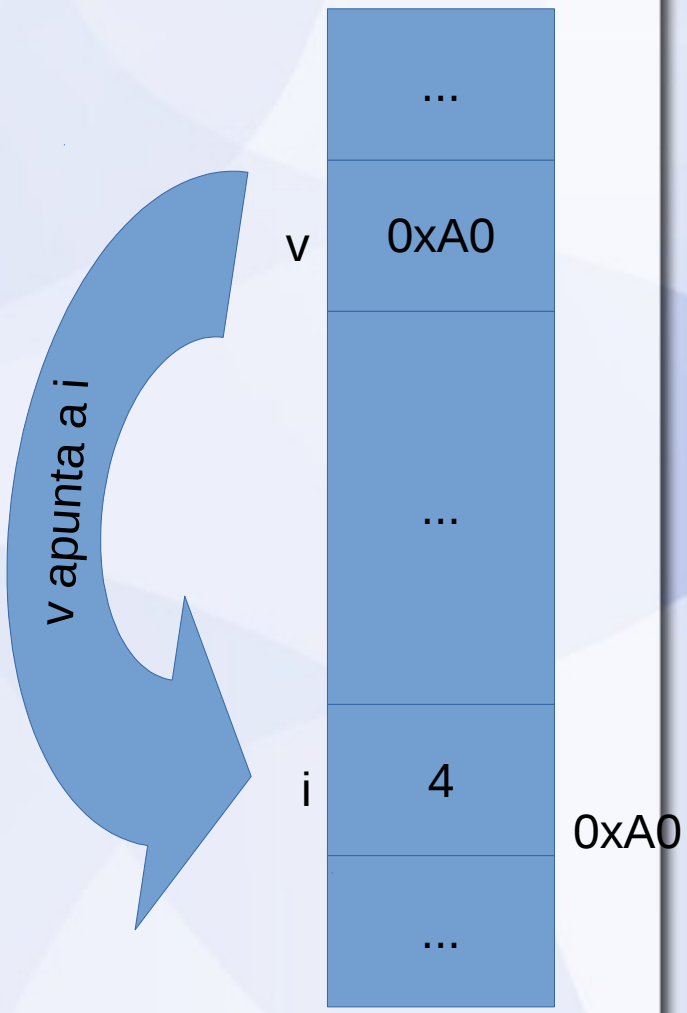
Punteros y funciones

```
void asignar(int v) {
    v = 5;
}

...
int i = 4;
asignar(i);
printf("%d", i); // 4
```



Punteros y funciones (cont.)

```
void asignar(int *v) {
    *v = 5;
}

...
int i = 4;
asignar(&i);
printf("%d", i); // 5
```



Punteros

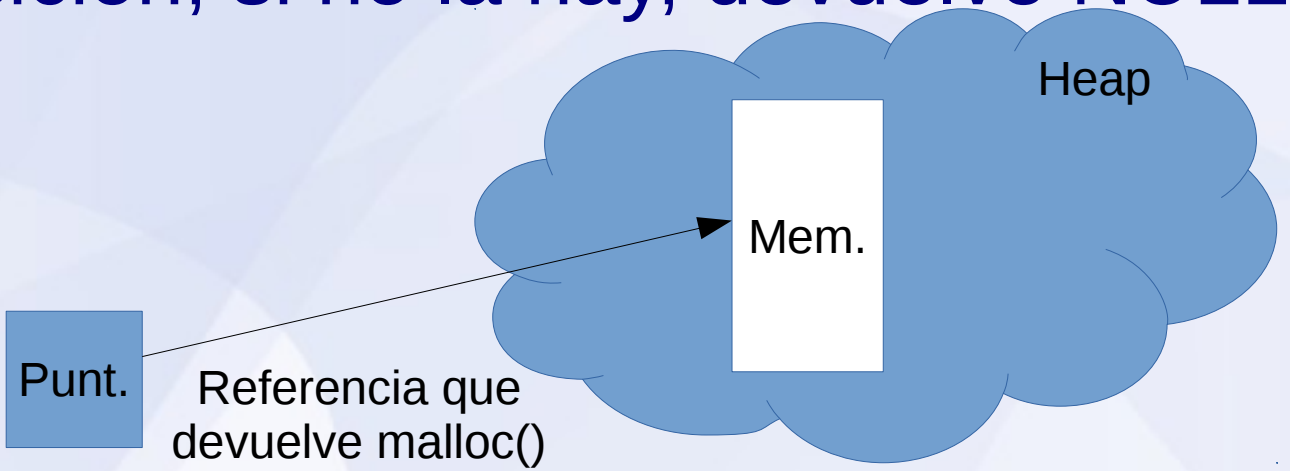
- Los punteros tienen varios usos:
  - Referenciar indirectamente a variables locales
  - Permitir que las funciones de C puedan modificar variables externas
  - Utilizarse para apuntar a memoria gestionada con el sistema operativo (memoria dinámica)

Memoria dinámica

- Memoria estática: Stack, dinámica: Heap
- Stack: Cantidad fija, muy limitada.
- Se gestiona con el sistema operativo.
- Pido la cantidad de memoria que necesito, si el sistema la tiene, me la devuelve.
- Cuando no la uso más, debo liberarla

Memoria dinámica (malloc)

- void *malloc(size_t n); // #include <stdlib.h>
- Pide n bytes de memoria.
- Si hay memoria suficiente, devuelve la posición, si no la hay, devuelve NULL

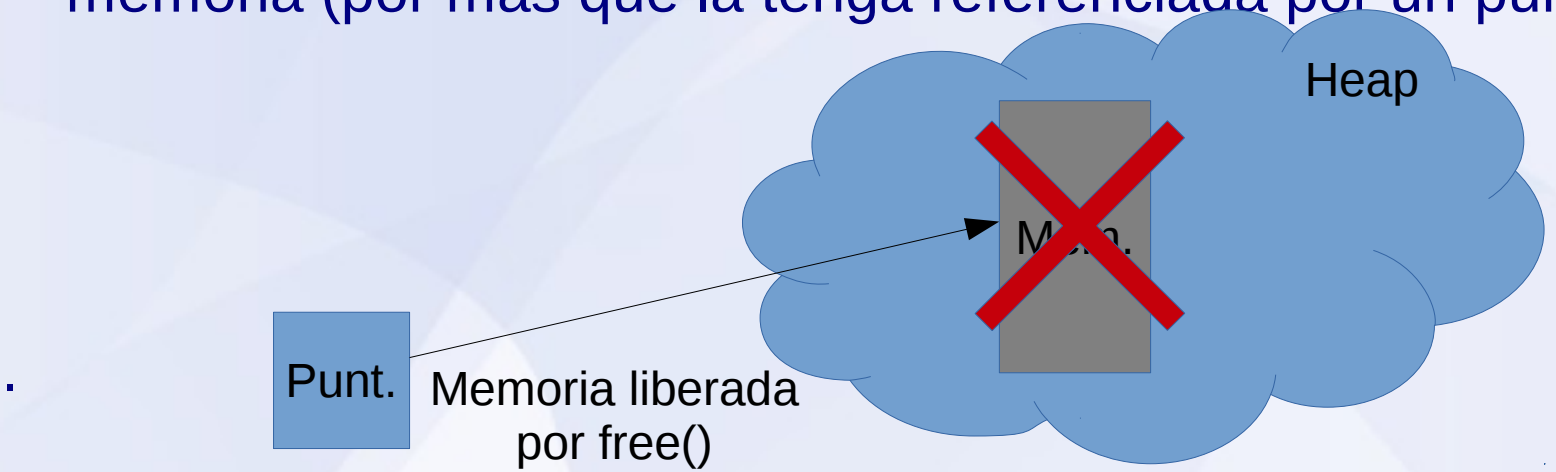


Memoria dinámica (void *)

- En C las variable apuntadoras tiene tipo, sirve para:
  - Acceder al contenido: i = *p
  - Encontrar el elemento siguiente: p + 1
- Cuando se quiere sólo almacenar una dirección de memoria sin importarme el tipo, se pueden usar variables de tipo void *.
- Las funciones que trabajan con memoria genérica utilizan el tipo void * en su interfaz.

Memoria dinámica (free)

- void free(void *);
- Libera la memoria que comienza en la posición dada (esta posición tiene que ser la que devolvió un malloc() -o realloc()- antes).
- Una vez que la memoria apuntada se liberó ya no poseo esa memoria (por más que la tenga referenciada por un puntero).



Memoria dinámica (ejemplo)

```
int *p;

p = malloc(sizeof(int)); // Pido memoria para un entero
if(p == NULL) { ERROR }

*p = 5; // “*p” es un entero

free(p); // Libero la memoria pedida

*p = 5; // ERROR, p no apunta a una dirección válida
```

Memoria dinámica (realloc)

- void *realloc(void *o, size_t n);
- Redimensiona la memoria comprendida en o al tamaño nuevo n.
- n puede ser mayor o menor al tamaño anterior.
- Algoritmo:
  - Intenta hacer un malloc(n):
    - Si falla devuelve NULL (¡¡¡Y no hace NADA más!!!)
    - Si funciona, hace un memcpy() de la memoria vieja y después libera a o. Devuelve la memoria nueva.

Memoria dinámica (realloc cont.)

- Es importante señalar que si no hay memoria no hace nada con la memoria pedida:
- `p = realloc(p, 100); // MAL`
- Siempre debe hacerse:

```
aux = realloc(p, 100);
if(aux == NULL) { free(p); ...}
p = aux;
```

Memoria dinámica (tips)

- free(NULL); <-- No hace nada
- malloc(0); <-- No hace nada
- realloc(NULL, n) <-- Funciona como malloc(n)
- realloc(p, 0) <-- Funciona como free(p)
- Estas cosas pueden ser prácticas para entrar en régimen en un código más complejo.

Memoria dinámica (pérdida)

- Toda la memoria pedida debe ser liberada cuando no haga más falta.
- Si pierdo la referencia a un bloque de memoria pedida no tengo forma de liberarla => “Memory leak” (pérdida de memoria).
- Para detectar pérdidas de memoria utilizamos valgrind.
- Hay que compilar en el GCC con -g (debug).
- valgrind --leak-check=full ./programa:
  - in use at exit: 0 bytes in 0 blocks
  - All heap blocks were freed -- no leaks are possible

Estructuras

- Una estructura permite en C empaquetar variables de distinto tipo en un tipo nuevo
- Cada una de estas variables se llama “miembro”
- Se accede a los miembros usando el operador “.” (punto)

```
struct complejo {
   double x, y;
};
...
struct complejo v1;
v1.x = 5;
v1.y = 3.14;
```

Estructuras (typedef)

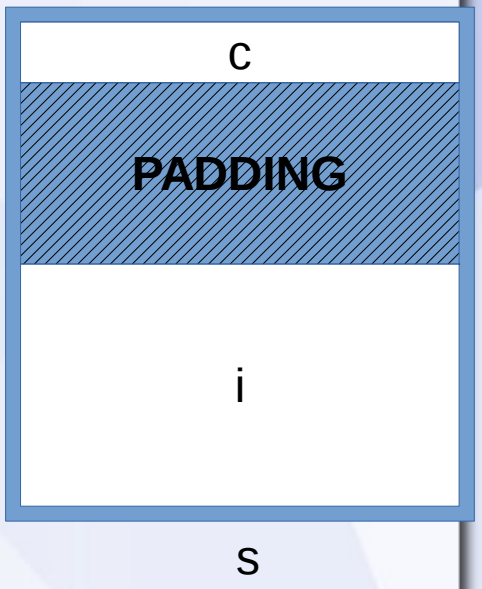
- También se puede hacer:

```
typedef struct {
   double x, y;
} complejo_t;

complejo_t v1;
...
```

Estructuras (acceso)

- Una estructura combina todos los miembros en un bloque de memoria contiguo.
- No necesariamente los miembros ocupen el total de la estructura:
  - Puedo hacer: e2 = e1 porque copia TODO, incluyendo la basura de los espacios vacíos.
  - No puedo hacer: if(e2 == e1) porque compara TODO, incluyendo la basura que puede ser diferente en estructuras idénticas. Sólo puedo comparar miembro a miembro.

```
struct s {
    char c;
    int i;
};
```



Estructuras (punteros)

- El sizeof() de una estructura es >= a la suma del sizeof() de sus miembros.
- A las funciones es conveniente pasarles un puntero a estructuras para no hacer una copia por valor del sizeof() de la estructura.
- Si tengo:

```
struct est *var;

(*var).miembro  ==> var->miembro
```

- Usar siempre el operador -> cuando haga falta acceder al miembro de un puntero a estructura.

Estructuras (anidación)

- Se pueden anidar y referenciar otras estructuras.
- Se puede incluir dentro de una estructura una referencia a sí misma (estructura recursiva).

```
struct direccion {
   char calle[MAX_CALLE];
   int numero;
};
struct persona {
   char nombre[MAX_NOMBRE];
   struct direccion casa;  // Anidada, consume memoria dentro de persona.
   struct persona *padre, *madre; // Referenciada, si es NULL consume sólo memoria.
};
```

Estructuras (punteros cont.)

- Si quiero acceder a la calle del bisabuelo en línea paterna de una persona:

```
struct persona *p;
(*(*(*(*p)->padre)->padre)->padre).calle;
p->padre->padre->padre.calle;
```

- **¡¡¡Usar el operador “flechita”!!!**
- (Similar a *(*(*(v+i)+j)+k) versus v[i][j][k])

Cadenas

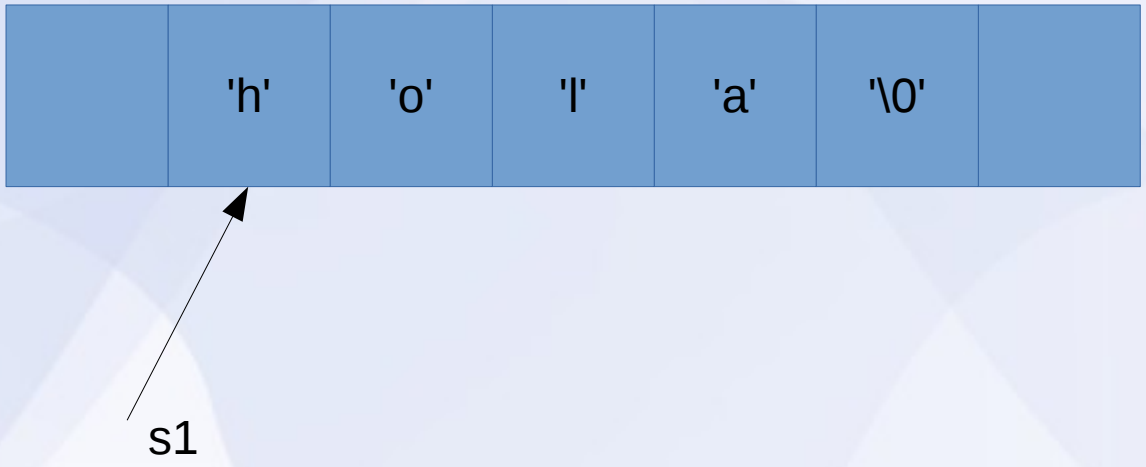
- En C existe un soporte rudimentario para cadenas de caracteres
- En memoria las cadenas se representan como una sucesión (vector) de variables de tipo char
- Para prescindir de informar la longitud de una cadena, se utiliza como centinela el carácter '\0' al final de la misma
- Los literales de cadena se declaran entre comillas dobles (i.e.: “hola”)
- El tamaño del soporte no tiene por qué ser igual al tamaño de la cadena (sí tiene que ser al menos uno más)

Cadenas (cont.)

```
char s[] = “hola”;
...
printf(“%s”, s); // Paso un puntero a donde comienza, conoce el final por el '\0'
...
printf(“%u”, sizeof(s)); // 5, el vector es {'h', 'o', 'l', 'a', '\0'}
...
[[ Requiere  #include <string.h> ]]
printf(“%u”, strlen(s)); // 4, es la longitud útil de la cadena.
...
void imprimir(char s[]) {
   for(int i = 0; s[i] != '\0'; i++)
      putchar(s[i]);
}
```

Cadenas (cont.)

```
char s1[] = “hola”, s2[] = “chau”;
if(s1 == s2) // ¡ERROR! estoy comparando los punteros
if(strcmp(s1, s2) == 0) // Función que compara
carácter a carácter, <string.h>
```



.

Cadenas (cont.)

```
char s2[] = “hola”, s1[10];
s1 = s2; // ¡ERROR! esto ni siquiera compila
strcpy(s1, s2); // Función que copia carácter a carácter, <string.h>
```

- Importante: ¡Debo garantizar que s2 entra en s1!
- Existe también strncpy(s1, s2, n) pero también tengo que garantizar que entra porque de no hacerlo no copiará hasta el '\0' y s1 **no será** una cadena.

Cadenas (strdup())

- Para crear una copia de una cadena suele implementarse la función strdup() (perteneciente al estándar POSIX):

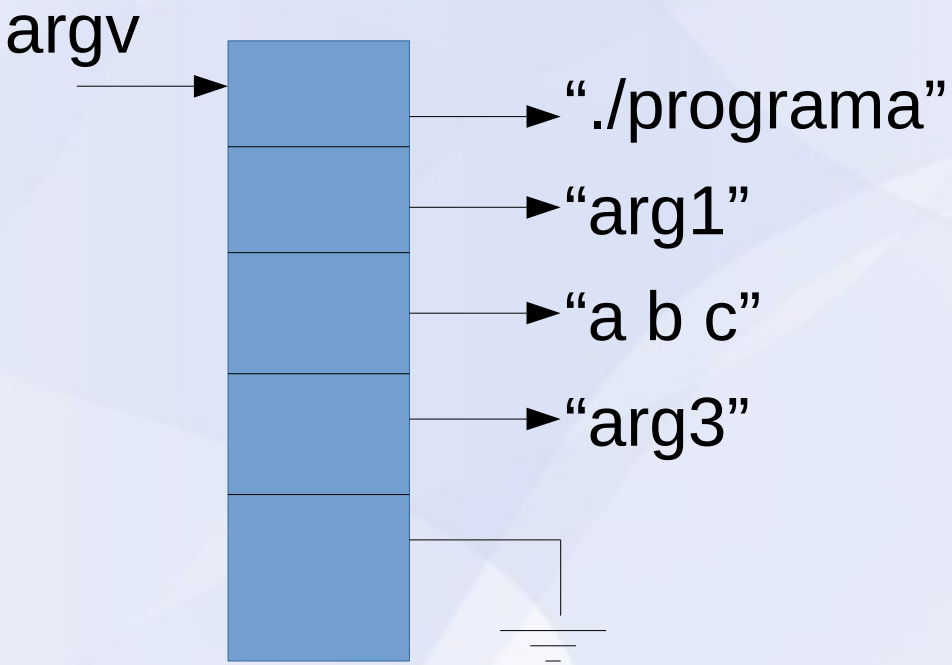
```
char *strdup(const char *old) {
   char *new;
   if((new = malloc(strlen(old) + 1)) == NULL)
      return NULL;
   strcpy(new, old);
   return new;
}
```

CLA

- Los programas en C pueden recibir argumentos desde la línea de comandos.
- Hay dos firmas admitidas para la función main():
  - `int main(void);`
  - `int main(int argc, char *argv[]);`
- argc es la cuenta de argumentos (incluyendo el nombre del programa) y argv un arreglo de punteros a cadenas.

CLA (cont)

- $ ./programa arg1 “a b c” arg3



argc = 4