Introducción al lenguaje de programación C

75.40 - Algoritmos y Programación I

Curso Wachenchauzer

Versión 1: Sebastián Santisi 19/06/2015

Versión 2: Diego Essaya 31/05/2017

Python versus C

Aprendemos un segundo lenguaje en base a las diferencias con el primero

Python	C
Guido van Rossum, 1991	Dennis M. Ritchie, 1972
Alto nivel	Bajo nivel
PCs	Mainframes
Interpretado	Compilado
Tipado dinámico	Tipado estático
Recolector de basura	Memoria administrada en forma manual
Pensado para scripting	Pensado para escribir un sistema operativo
2.x, 3.x, ...	C89, C99, C11, ...

Bajo nivel

Instrucciones de máquina: ¡ceros y unos!

```
 opcode   op1   op2
------------------
10110000 0000 0110 # cargar un 6 en el registro A
01011010 0000 0001 # Incrementar en uno el registro A
11001001 0000 0000 # Finalizar la ejecución
```

Cada modelo de procesador tiene su propio set de instrucciones.

El lenguaje *ensamblador* asigna un código memotécnico a cada instrucción.

```
LDA #6
ADD A, #1
HLT
```

Compilador

Ejemplos: `gcc`, `clang`

Transforma un programa escrito en un lenguaje determinado a instrucciones de máquina:

```
if (x == 0) {
    y += 1;
} else {
    y -= 1;
}
```

→

```
      JNZ A, @E
      ADD B, #1
      JMP @END
@E:   ADD B, #-1
@END: HLT
```

Una vez compilado, el programa solo se puede ejecutar en una plataforma determinada (CPU / sistema operativo).

Intérprete

Ejemplo: `python`

Es un programa compilado para una plataforma.

Carga el código fuente (script) y lo ejecuta línea a línea.

Estereotipos:

- Más fácil de programar
- Más lento
- No optimiza
- Los errores saltan en tiempo de ejecución

Tipado dinámico vs estático

Python (dinámico)

```
a = "hola"
...
a = 5
...
a = a ** 100
```

C (estático)

```
int a;
a = 5;
a = 3.14; // Guarda 3
a = "hola"; // ???
```

Memoria administrada por el entorno

En Python la memoria se administra automáticamente:

```
a = "hola" + "mundo" + "."
```

Objetos creados en memoria:

```
     str("hola")
     str("mundo")
     str(".")
     str("holamundo")
a -> str("holamundo.")
```

Las 4 cadenas intermedias eventualmenteserán destruidas por el *recolector de basura*.

Memoria administrada por el programador

En C, el programador debe encargarse de administrar la memoria.

Generalmente los tipos simples pueden operarse sin manejar memoria salvo para almacenarlos.

```
int a = 4, b = 3, c = 6, d;
d = a + b + c + 5
```

En Python la suma parcial necesita ser administrada, en C sólo ocupan memoria `a`, `b`, `c` y `d`.

La memoria se distingue en **estática** (*stack*) y **dinámica** (*heap*, peticionada al sistema operativo).

Memoria

Es una secuencia numerada **bits** agrupados de a ocho (**bytes**). Cada *celda* tiene una **dirección**:

Dirección	Bits
0	`01010010`
1	`10100001`
2	`01010100`
3	`10100001`
4	`00111010`
5	`11001001`
6	`01010010`
7	`11010110`
8	`00110101`
9	`01101010`
10	`10110111`
...	...

Memoria (cont.)

- La memoria no tiene "tipo".
- Un determinado conjunto de bits puede interpretarse como un número entero con o sin signo, y luego ese número puede ser escrito en notación decimal, octal, hexadecimal, etc.
- O el mismo dato puede ser interpretado como un caracter en alguna codificación (por ejemplo ASCII).
- O puede ser interpretado como un número de punto flotante (signo + mantisa + exponente)
- O cualquier otra interpretación posible.

Dirección	Hex	Bits	8 bits	Hex	8 bits (signed)	32 bits	ASCII
0	0x0	`01010010`	82	0x52	82	1.386.304.673	R
1	0x1	`10100001`	161	0xA1	-95		¡
2	0x2	`01010100`	84	0x54	84		T
3	0x3	`10100001`	161	0xA1	-95		¡
4	0x4	`00111010`	58	0x3A	58	986.272.470	:
5	0x5	`11001001`	201	0xC9	-55		É
6	0x6	`01010010`	82	0x52	82		R
7	0x7	`11010110`	214	0xD6	-42		Ö

Tipos

Declaraciones y definiciones

```
double frecuencia_angular; // declaración
frecuencia_angular = 7.9;  // definición
unsigned int cantidad_asistentes = 50;
unsigned char edad;
long centavos;
float temperatura;
```

- Declarar una variable implica **reservar memoria** para ella.
- Por eso la declaración debe indicar el **tipo**.
- Las variables se declaran una única vez.
- Declarar ≠ definir (asignarle un valor).
- Ojo: Una variable no inicializada contiene *basura* (el valor que estaba en la memoria en el momento de la declaración).

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`

Punto flotante (IEEE754): 1 bit de *signo* y dos números enteros: *mantisa* y *exponente* (`m * 10^exp`)

- `float`: 32 bits, ~8 dígitos significativos en representación decimal
- `double`: 64 bits, ~16 dígitos significativos en representación decimal

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

- Se puede usar `sizeof(x)` para saber el tamaño en bytes de un tipo o variable en tiempo de compilación.
- 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.

Literales

```
x = 0;
y = "hola";
```

`x`, `y` son *identificadores*, `0` y `"hola"` son *literales*.

Los literales son tipados en C.

Todos estos representan al número 0 con un tipo distinto:

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

Funciones

También pueden *declararse* y *definirse* por separado.

Se debe indicar el tipo de los argumentos y del valor de retorno.

```
// Declaración:
int sumar(int a, int b);

// Definición:
int sumar(int a, int b) {
    return a + b;
}

...

int x = sumar(4, 5);
```

Conversiones de tipos

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

...

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

Truncado 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 `signed` en un `unsigned`, etc.). Puede haber pérdida de información.

```
char x = 129; // x = -127
int y = 8.5;  // y = 8
```

Promoción de tipos

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.

```
float x = 30; // x = 30.0
```

Cuando se intenta operar entre dos operandos distintos el "más chico" se promueve al tipo del "más grande".

```
int x = 3;
float y = 8.5;
int z = x + y; // ??
```

Casteo

Se puede convertir explícitamente un tipo anteponiendo el nombre del tipo destino entre paréntesis:

```
int a = 5;
int b = 2;
float c = a / b;          // c = 2.0
float d = (float)a / b; // d = 2.5
float e = a / (float)b; // e = 2.5
```

Hello, world

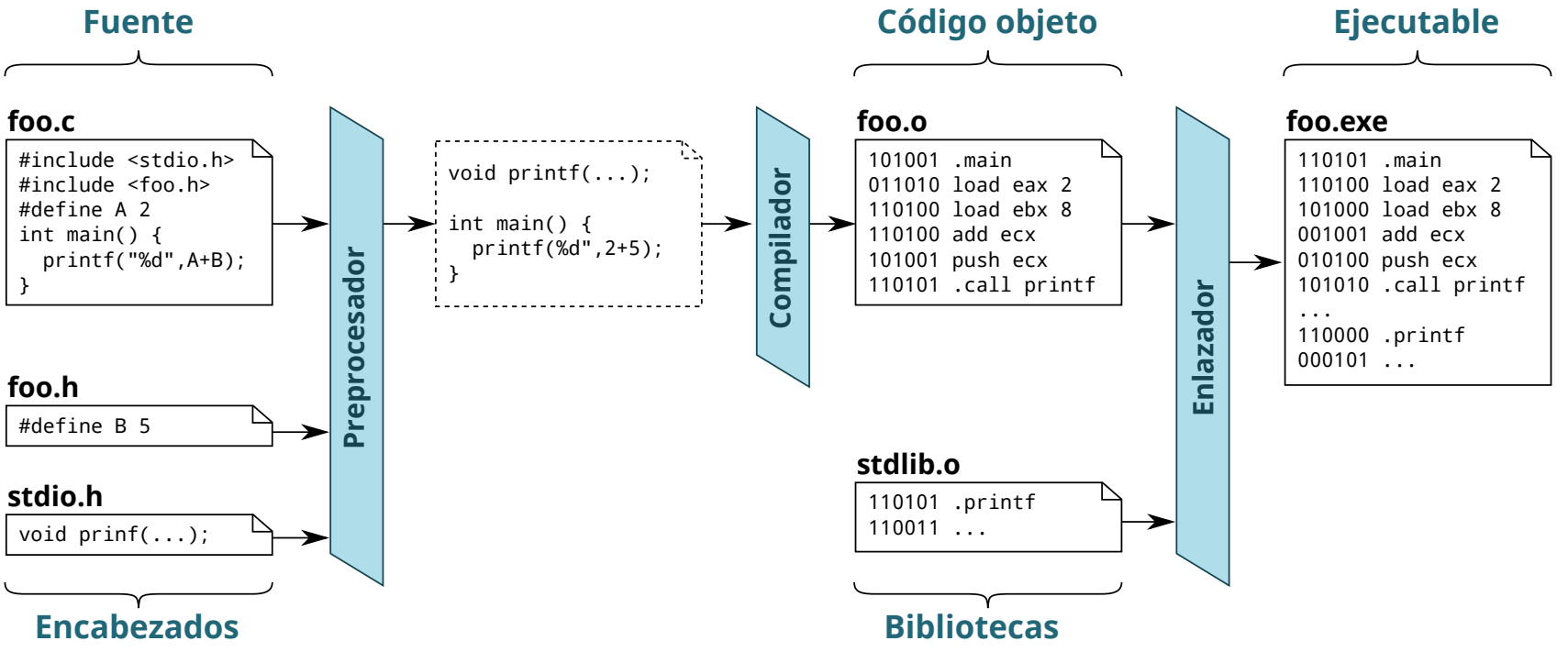
```c
#include <stdio.h>

/* Implementa el hola mundo en C */

int main() {
    printf("Hola mundo\n");
    return 0;
}
```

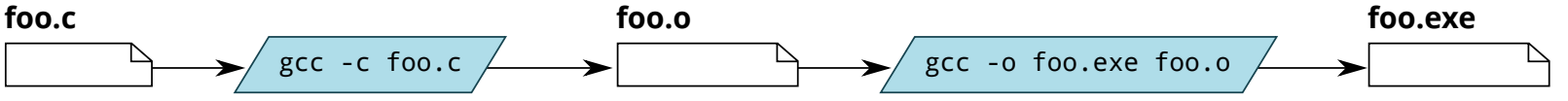
Hola mundo

```c
#include <stdio.h>

int main() {
    printf("Hola mundo\n");
    return 0;
}
```

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

Proceso de compilación





Generalidades de C

```c
int main() {
    printf("Hola mundo\n");
    return 0;
}
```

- Los bloques de código se delimitan con `{ ... }`
- Las sentencias se delimitan con `;`
- Todas las sentencias deben estar adentro de funciones
- Debe haber uno y sólo un punto de entrada por programa (`main()`)

Generalidades de C (cont.)

El compilador trabaja de arriba hacia abajo en una sola pasada.

```c
int main() {
  int x = sumar(5, 4); // ERROR
}

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

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

int main() {
  int x = sumar(5, 4); // OK
}
```

```c
int sumar(int, int);

int main() {
  int x = sumar(5, 4); // OK
}

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

Comentarios

Una línea (`// ...`)

```
int x = 9; // comentario
```

Multilínea (`/* ... */`)

```
/* comentario
   comentario
   comentario */
int x = /* comentario */ 9;
```

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 puede crear **etiquetas**:

```
#define PI 3.14159
```

- `#define` es una 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`

Renombrar tipos

```
typedef <tipo> <nuevo_nombre>;
```

```
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, la variable `clase_dia` es de tipo numérico, y las etiquetas se definen como si se hubiera hecho muchos `#define LUNES 0`, `#define MARTES 1`, etc.

No hay forma de transformar el contenido de `clase_dia` en una cadena `"LUNES"`.

printf()

Python

```
print("Hola {}, tienes {} mensajes".format("Alan", 5))
```

C

```
printf("Hola %s, tienes %d mensajes\n", "Alan", 5);
```

%	interpretado como
%c	`char` (un caracter)
%s	`char *` (cadena)
%d	`int`
%f	`float`
%%	%

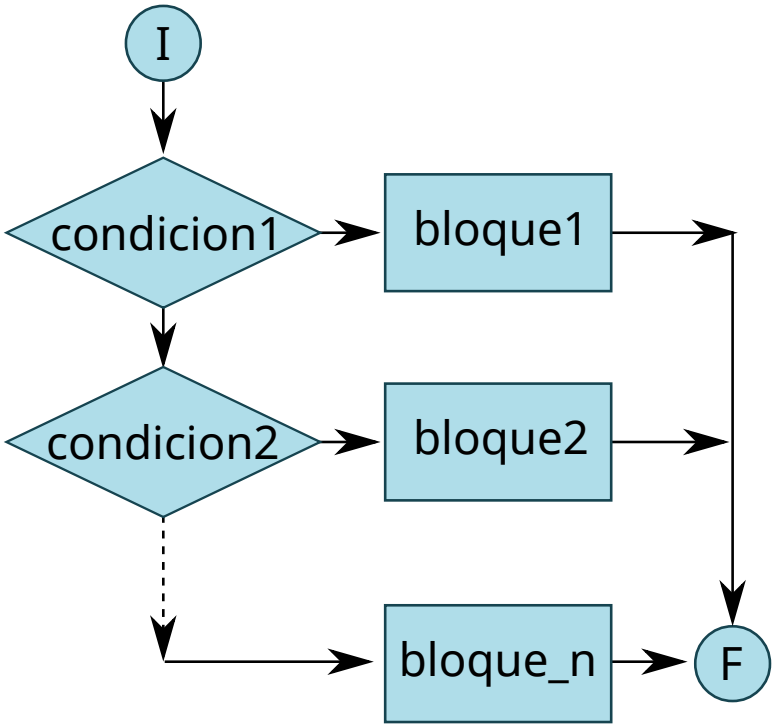
...

Estructuras de control

```
if (...) { ... }
while (...) { ... }
for (...) { ... }
do { ... } while (...)
switch { ... }
```

if

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



Bloques

Se inician y terminan con `{}`.

```
if (llueve) {
  llevar_paraguas();
}
```

Un bloque con una única instrucción puede prescindir de los `{}`.

```
if (llueve)
  llevar_paraguas();
```

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

```
int condicion = 1;
if (condicion) {
  ...
}
```

Los operadores booleanos operan sobre los distintos tipos y devuelven un entero según esa lógica:

- `x == y`: Igualdad
- `x != y`: No igualdad
- `x && y`: And
- `x || y`: Or
- `!x`: Not
- `<`, `>`, `<=`, `>=`

Booleanos (cont.)

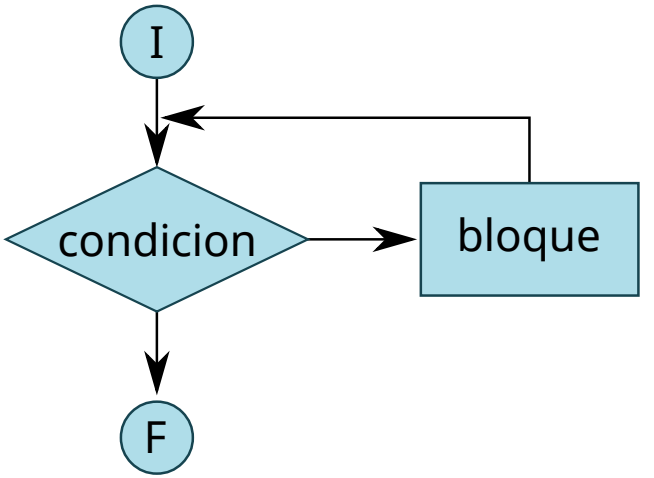
El header `<stdbool.h>` define algunas constantes útiles:

- El tipo de datos `bool`: es un número entero sin signo (típicamente de 8 bits).
- Las constantes `false = 0` y `true = 1`.

```c
#include <stdbool.h>
...
bool condicion = true;
```

while

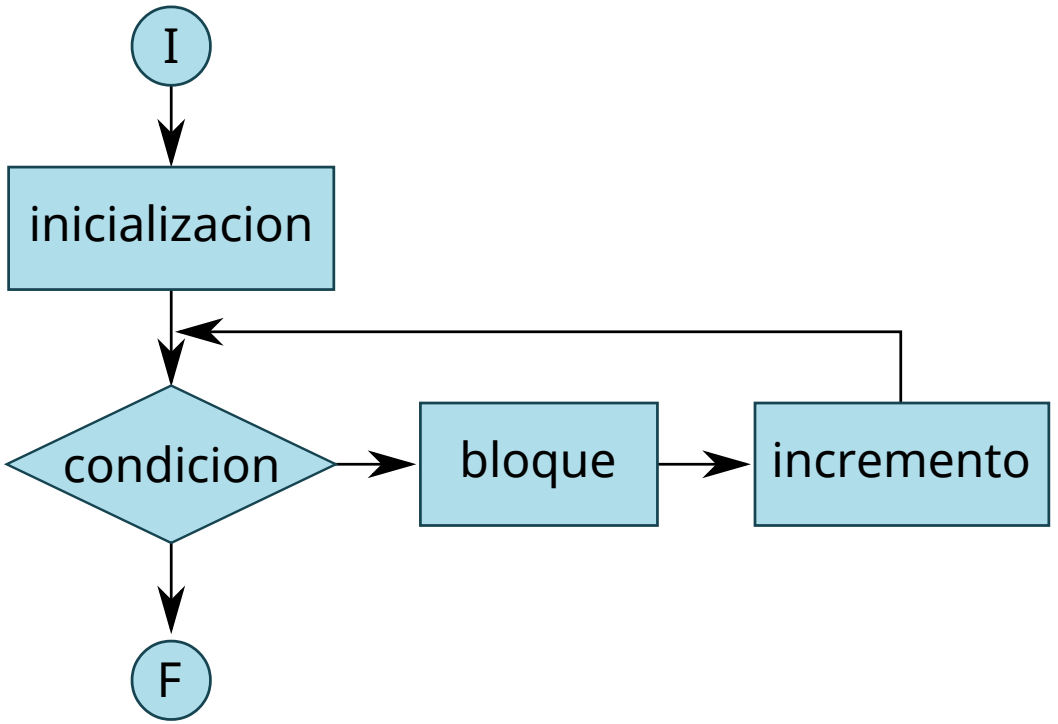
```
while (condicion) {
  bloque
}
```



`break` y `continue` funcionan igual que en Python.

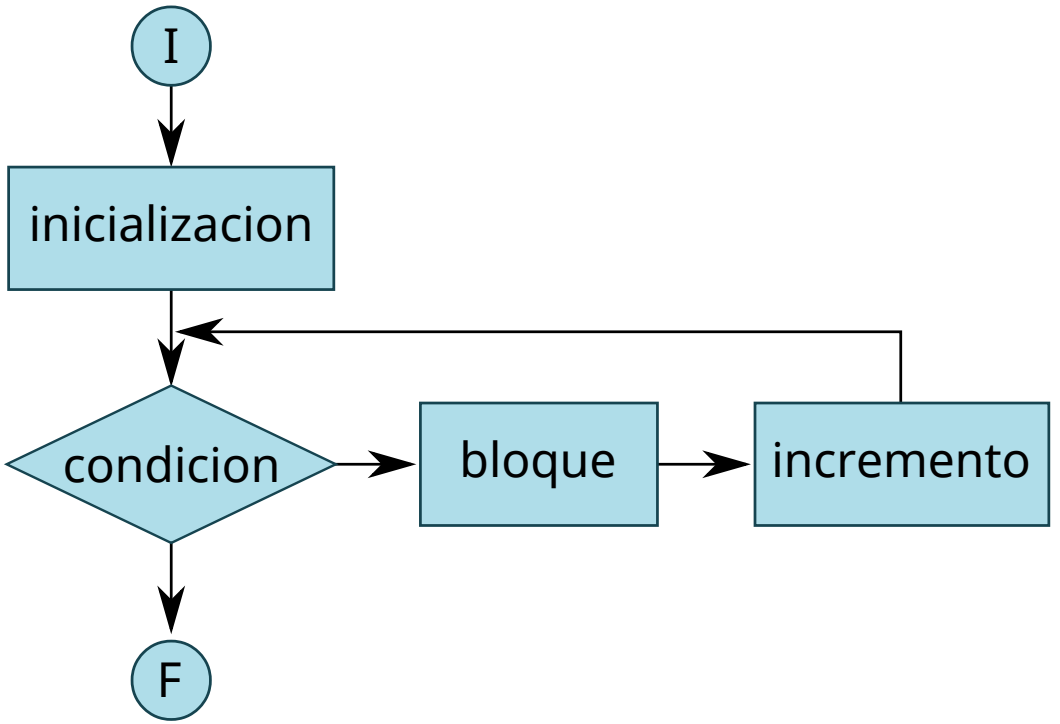
for

```
for (inicializacion; condicion; incremento) {
  bloque
}
```



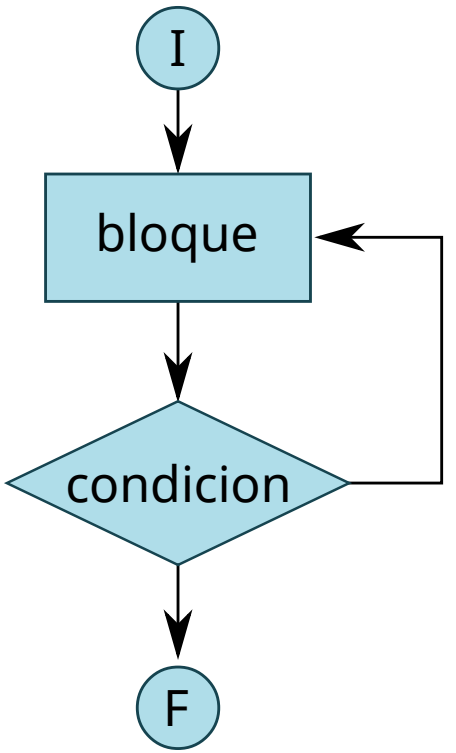
for (ejemplo)

```c
for (int i = 0; i < 10; i++) {
  printf("%d\n", i);
}
```



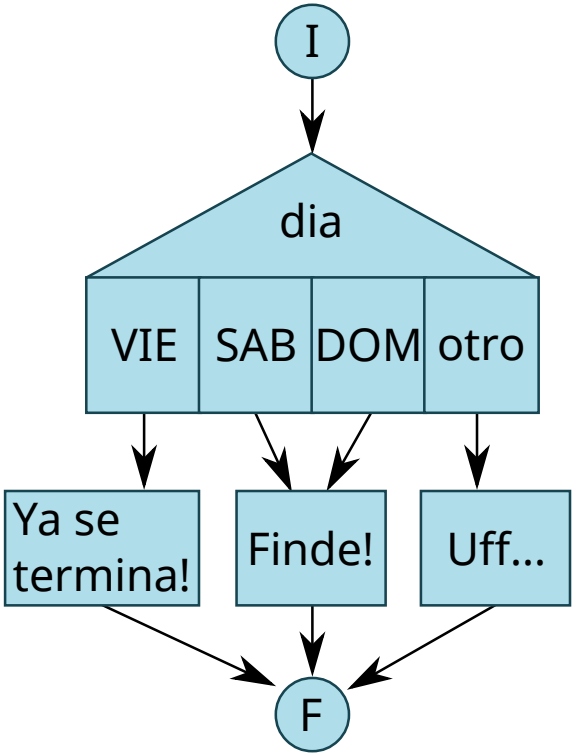
do while

```
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("Uff...\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!)

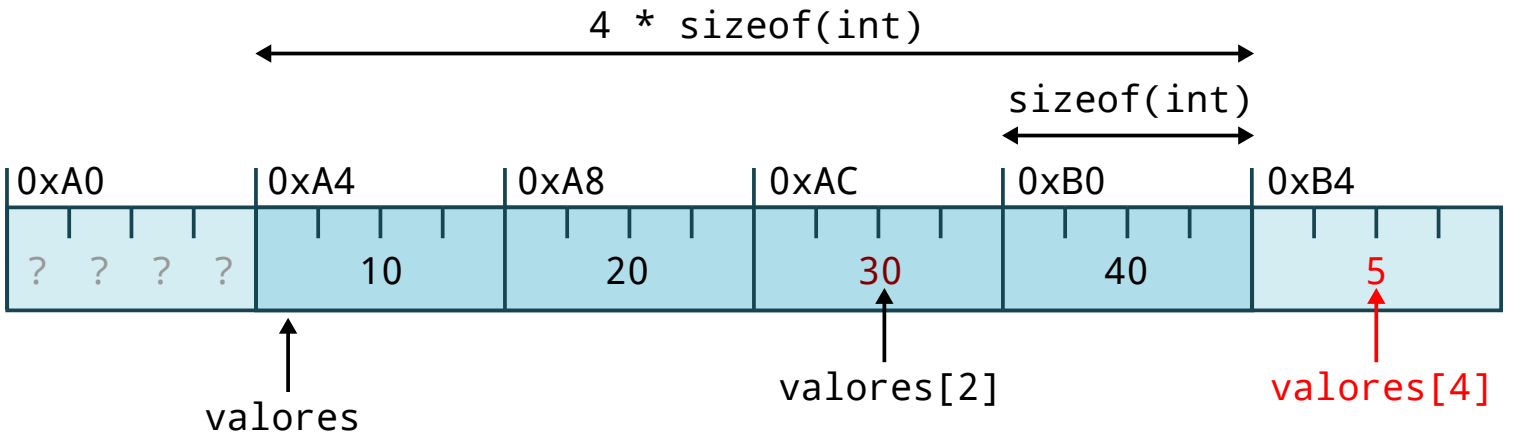
Una vez que se entró por una etiqueta se sigue ejecutando hasta encontrar un `break`.

Si omito el `break` entre dos etiquetas va a seguir ejecutando de una a la siguiente.

Vectores

Vectores

```
int valores[4] = {10, 20, 32, 40};
valores[2] = 30;
valores[4] = 5; // !!!!
```



Vectores (cont.)

Son un paquete de `N` variables del mismo tipo.

Ocupan un espacio de `N * sizeof(tipo)` espacios consecutivos en memoria.

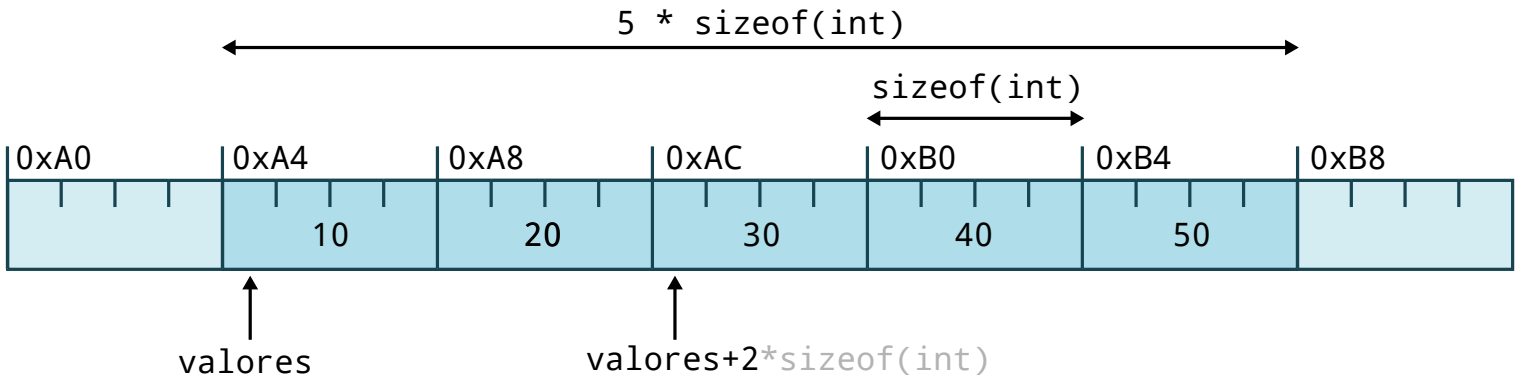
Se numeran del `0` al `N - 1`.

C no chequea el rango de acceso, es responsabilidad del programador no leer/escribir fuera del rango válido.

Vectores (cont.)

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

Internamente, `valores` es un número entero, que corresponde a la dirección de memoria del primer valor del vector.



Vectores y funciones

```
void imprimir_vector(int v[], int n) {
  for(int i = 0; i < n; i++) {
    printf("%d\n", v[i]);
  }
}
...
#define CANT_VAL 5
int valores[CANT_VAL] = {10, 20, 30, 40, 50};
imprimir_vector(valores, CANT_VAL);
```

¿Cómo resuelve el compilador esta llamada?

¿Por qué es necesario pasar la cantidad de elementos?

Vectores y funciones (cont.)

Al llamar a una función en C, todos sus argumentos se pasan **por valor**.

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.

Estructuras

```
struct punto {
  double x;
  double y;
};
...
struct punto p;
p.x = 5;
p.y = 3.14;
```

Permite definir un tipo de datos nuevo, agrupando un conjunto de variables de distinto tipo (**miembros**).

Se accede a los miembros usando el operador `.` (punto).

Estructuras no son clases

¡C no es un lenguaje orientado a objetos!, ¡no soporta objetos ni nada parecido!

Una estructura es sólo un agrupamiento en memoria de variables. No puede asociarse funciones (métodos) a estructuras.

Sirve para agrupar y ordenar información, y sobre eso pueden diseñarse tipos abstractos.

Punteros

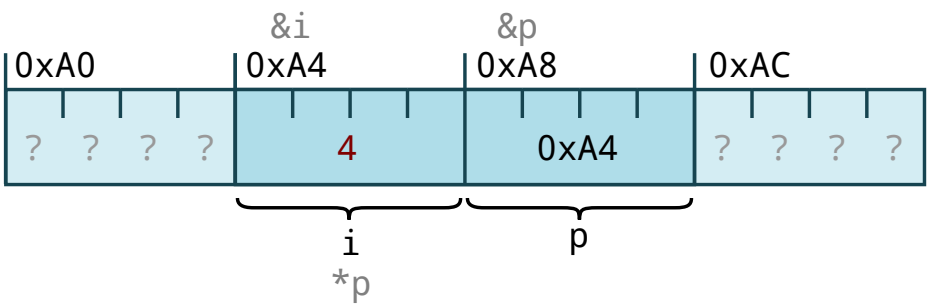
Punteros

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 interpretar el contenido de esa memoria y su dimensión).

Punteros (cont.)

```c
int i = 5; // i es un número entero
int *p; // p es un puntero a un entero
p = &i; // p ahora contiene la dirección de memoria de i
        // p "apunta a" i

printf("%p, %p, %p",  &i,   p,    &p);
                    // 0xA4, 0xA4, 0xA8
printf("%d, %d", i, *p);
                // 5,  5

*p = 4;
printf("%d, %d", i, *p);
                // 4,  4
```



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 el valor que está en la dirección de memoria `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`").

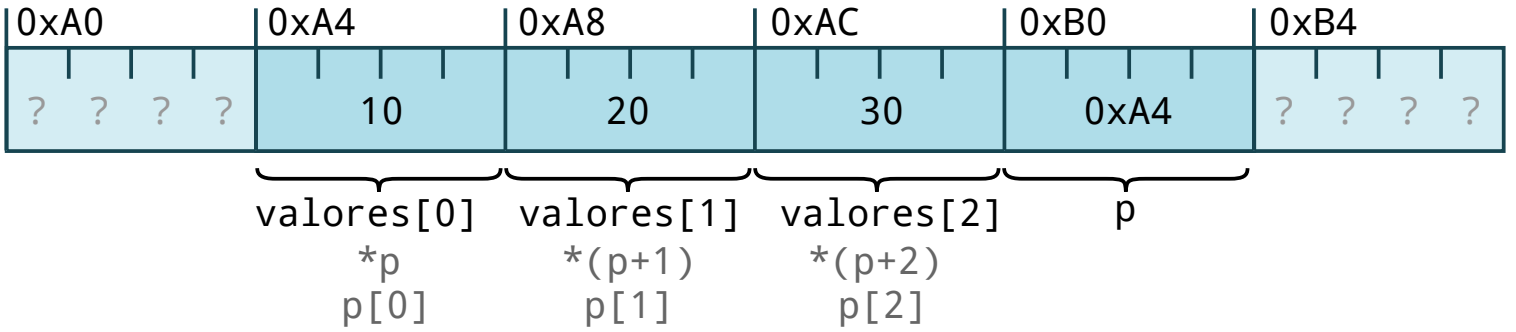
Aritmética de punteros

```c
int valores[] = {10, 20, 30}
int *p = valores;

printf("%p, %p, %p, %p", valores,  p,   p+1,  p+2);
                        //  0xA4,  0xA4, 0xA8, 0xAC
printf("%d", *(p+2), p[2]);
            // 30,    30
```

Mediante un puntero se puede acceder a memoria consecutiva de la que se apunta.

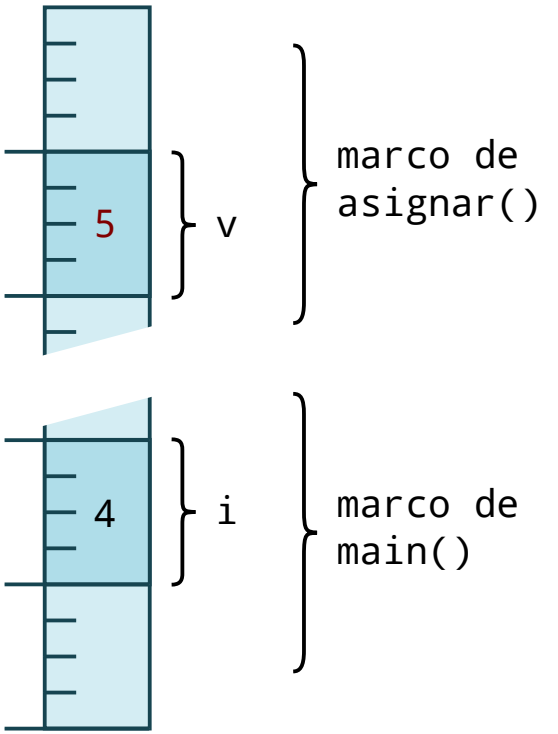
El resultado de `p + x` es la dirección `p + x * sizeof(*p)`.

`p[x]` es equivalente a `*(p + x)`.



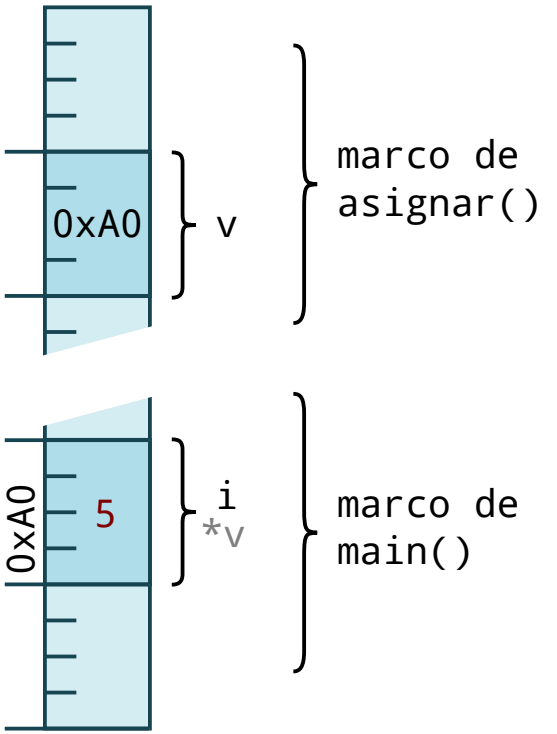
Punteros y funciones

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

int main() {
  int i = 4;
  asignar(i);
  printf("%d", i); // 4
  return 0;
}
```



Punteros y funciones (cont.)

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

int main() {
  int i = 4;
  asignar(&i);
  printf("%d", i); // 5
  return 0;
}
```



Punteros (con esta terminamos)

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)

En esta introducción nos interesa únicamente que se entienda el mecanismo de pasaje de vectores a funciones.

Cadenas

Cadenas

En C existe un soporte rudimentario para cadenas de caracteres.

En memoria las cadenas se representan como un vector de valores de tipo `char`. Cada caracter se codifica usando el estándar ASCII.

Para prescindir de informar la longitud de una cadena, se utiliza como centinela el **caracter nulo** `'\0'` al final de la misma.

Los literales de cadena se declaran entre comillas dobles:

```
char s[] = "hola"; // {'h', 'o', 'l', 'a', '\0'}
```

Un literal de tipo `char` se declara entre comillas simples:

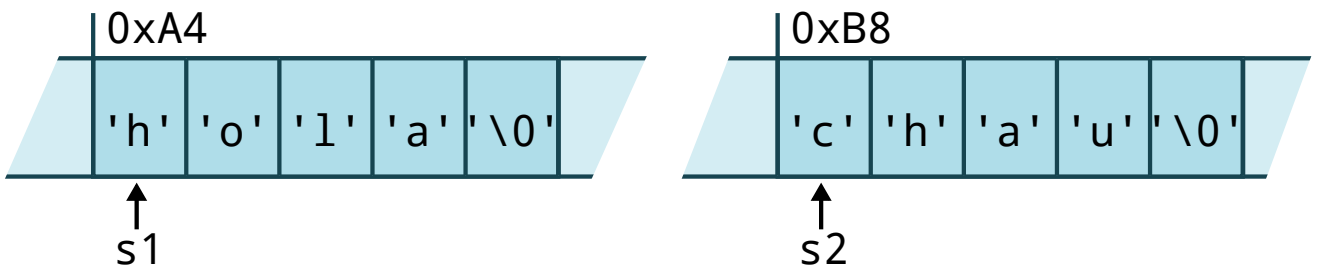
```
char c = 'H'; // c contiene el valor ASCII del caracter H
s[0] = c; // {'H', 'o', 'l', 'a', '\0'}
```

Cadenas (cont.)

```c
char s[] = "hola";

// printf recibe el puntero al inicio de la cadena
// Recorre e imprime hasta que encuentra un '\0'
printf("%s", s);

printf("%u", sizeof(s)); // 5

// Requiere #include <string.h>
printf("%u", strlen(s)); // 4

void imprimir(char s[]) {
  for(int i = 0; s[i] != '\0'; i++) {
    putchar(s[i]);
  }
}
```

Cadenas (cont.)

```c
char s1[] = "hola", s2[] = "chau";
if (s1 == s2) {
  // ¡ERROR! estoy comparando los punteros
}
if (strcmp(s1, s2) == 0) {
  // Compara caracter a caracter, <string.h>
}
```



Interacción

Interacción (en C y en Python)

Todos los procesos (programas) tienen 3 flujos (archivos) abiertos por omisión

- `stdin`: Asociado al teclado
- `stdout`: Asociado a la consola (monitor)
- `stderr`: Asociado a la consola (monitor)

Este comportamiento puede modificarse.

Interacción en Python

Alto nivel

```
print("hola") # Imprime en stdout
s = input()   # Lee de stdin
```

Bajo nivel

```
import sys
sys.stdout.write("hola\n")
sys.stderr.write("Error\n")
s = sys.stdin.readline()
```

Lectura en C

`getchar()` permite leer de a un caracter de `stdin`:

```
int c = getchar();
```

`fgets()` ofrece un comportamiento similar a `input()`:

```
char s[30];
fgets(s, 30, stdin);
// Ojo: la cadena incluye el `\n`
```

`atoi()` permite convertir una cadena a un entero (como `int()` en Python):

```
int n = atoi(s);
```

Modularización

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)`).

Modularización

C permite compilar en objetos (`.o`) distintos fuentes (`.c`) por separado para luego enlazarlos en un único ejecutable.

Para esto, cada módulo debe conocer las funciones, tipos, etiquetas, etc. que ofrecen los módulos que utilizan.

Los archivos de **encabezados** (`.h`) contienen (principalmente) los **prototipos**, `typedef`s y `#define`s de los módulos existentes.

Por ejemplo, `stdio.h` contiene el prototipo de la función `printf()`, pero no su definición.

Para usar una función definida en otro módulo, se debe incluir (`#include`) el encabezado (`.h`) correspondiente, y luego enlazar el ejecutable con el archivo objeto (`.o`) que contenga la definición de esa función.

Fin