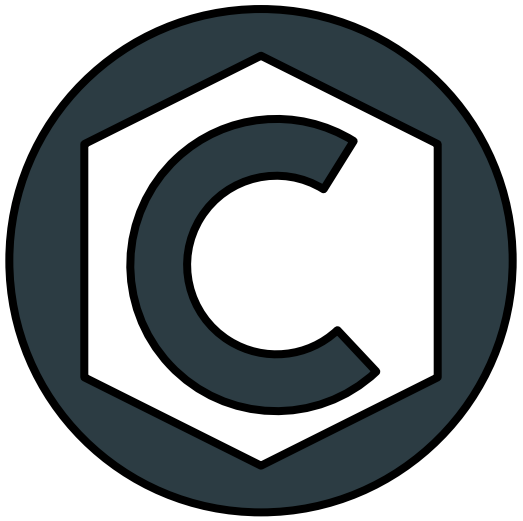


Una introducción informal a C para programadores

Nicolás Calvo Cruz (ncalvocruz@ual.es)

Grupo de Supercomputación y Algoritmos

Universidad de Almería

Noviembre de 2019

Presentación (I)

¿A quién va dirigido este curso?

- Preferentemente a **estudiantes del Grado Ingeniería Informática** que, aunque ya saben programación, **tienen problemas para desenvolverse en C**. Este lenguaje es necesario para asignaturas como **Sistemas Operativos** (2º) y **Multiprocesadores** (3º).

- No obstante, puede serle de interés **a cualquier persona con nociones de programación pero inexperto en C.**

Presentación (II)

Y… asignaturas aparte, ¿por qué aprender C?

- Es un lenguaje que nos da mucho control, lo que permite crear **programas muy eficientes**.
- Es **fácil de abarcar**: tiene un conjunto reducido de palabras reservadas y operadores.
- Permite escribir **todo tipo de software** para casi **todas las plataformas**.
- Es **interoperable**: Podemos asociar código C con Java, Python, Matlab, PHP… De hecho, esos lenguajes están generalmente implementados en C.
- Da pocas comodidades… por lo que "*muchos huyen*" de él aunque sea muy usado. ¡**Quien sabe C, destaca**!

Contenidos

- ¿Qué es C?
- Entorno de trabajo
- ¡Hola Mundo! (Y Compilación)
- Variables y constantes. Ejemplos
- Funciones. Ejemplos
- Punteros (a datos y funciones). Ejemplos
- Memoria y cómo pedirla (y liberarla!)
- Estructuras de control. Ejemplos
- Cómo depurar un programa
- Recibiendo parámetros por consola
- Creando valores aleatorios
- Midiendo tiempos
- Lectura y escritura de archivos de texto
- Aplicación: Creando una librería para operar con matrices

Contenidos

- **¿Qué es C?**
- Entorno de trabajo
- ¡Hola Mundo! (Y Compilación)
- Variables y constantes. Ejemplos
- Funciones. Ejemplos
- Punteros (a datos y funciones). Ejemplos
- Memoria y cómo pedirla (y liberarla!)
- Estructuras de control. Ejemplos
- Cómo depurar un programa
- Recibiendo parámetros por consola
- Creando valores aleatorios
- Midiendo tiempos
- Lectura y escritura de archivos de texto
- Aplicación: Creando una librería para operar con matrices

¿Qué es C? (I)

- **C es un lenguaje de programación** diseñado entre **1969 y 1972** en el seno de los Laboratorios Bell de AT&T.

- Es de **propósito general**, aunque se concibió inicialmente para programar **sistemas operativos**: también se hacen en C desde **drivers** hasta **juegos** y **aplicaciones de usuario**.

- Se trata de un lenguaje de **alto nivel** (de abstracción), pero se suele considerar de **medio**, por las opciones que brinda para controlar el entorno (**p. ej., memoria**)…

¿Qué es C? (II)

- Se trata de un lenguaje:
  - De **sintaxis** casi **idéntica a Java** (entre otros) y sensible a mayúsculas.
  - **Estructurado**: Tiene estructuras de control y tipos de datos estructurados definibles sobre los tipos básicos.
  - **De tipado:**
    - **Estático**: Cada variable tiene que definirse de un tipo antes de poder usarse.
    - **Débil**: Permite la conversión implícita de tipos.
  - Los **argumentos** de sus funciones se pasan **por valor**.

¿Qué es C? (III)

- Se caracteriza por ser:
  - **Compilado**: Para poder ejecutar un programa, debe convertirse en ensamblador y, finalmente, en código máquina para la arquitectura en la que se va a ejecutar.
  - **Compacto**: Tiene un conjunto reducido de definiciones y operaciones. Eso facilita la creación de compiladores… por lo que **C** sirve para casi **cualquier plataforma**.
  - **Estandarizado**: Inicialmente por ANSI y finalmente por ISO… Si se respetan, el **código** es **portable** entre plataformas.
  - Que permite hacer **programas muy eficientes**, siendo muy apreciado en la **computación de altas prestaciones**.

Contenidos

- ¿Qué es C?
- **Entorno de trabajo**
- ¡Hola Mundo! (Y Compilación)
- Variables y constantes. Ejemplos
- Funciones. Ejemplos
- Punteros (a datos y funciones). Ejemplos
- Memoria y cómo pedirla (y liberarla!)
- Estructuras de control. Ejemplos
- Cómo depurar un programa
- Recibiendo parámetros por consola
- Creando valores aleatorios
- Midiendo tiempos
- Lectura y escritura de archivos de texto
- Aplicación: Creando una librería para operar con matrices

Entorno de trabajo (I)

- Podemos programar en C en:
  - Linux
  - Mac
  - Windows

- **Vamos a trabajar en Linux** que, en mi experiencia, es donde **más cómodamente** se puede programar en este lenguaje.

- Necesitamos (como mínimo):
  - Editor de texto
  - Compilador

Entorno de trabajo (II)

- En este **laboratorio ya tenemos** un entorno adecuado:
  - Linux: Ubuntu
  - Editor de texto: Gedit
  - Compilador: GCC

- Si no fuera así, **podríamos construirlo** de esta forma:
  - Instalando Linux (real o virtual): recomiendo Xubuntu
  - Editor de texto: Mousepad (incluido en Xubuntu)
  - Compilador: `sudo apt-get install gcc`
  - Depurador (Opcional): `sudo apt-get install gdb`

Entorno de trabajo (III)

- Si no queremos instalar nada…, podríamos también prepararnos una distribución dentro de un medio USB:
  - Bajamos una **imagen**:

    https://drive.google.com/open?id=0B8Y50FxadxpXb2ZOcWoxMUs0TVU
  - La escribimos en un pendrive de 8 ó 16 GB:
    - Desde Linux:
      - *sudo fdisk -l # Identificamos el pendrive, p.ej., /dev/sdb*
      - *umount /dev/sdb # Lo desmontamos para escribirlo*
      - *sudo dd bs=512 skip=1 if=XubuntuMP.bin of=/dev/sdb # Escribir*
      - *sync # Forzamos que se limpien los buffers!*

El programa con el que hice mi imagen añade una cabecera de 512 bytes que hay que omitir!

Entorno de trabajo (III)

- Si no queremos instalar nada…, podríamos también prepararnos una distribución dentro de un medio USB:
  - Bajamos una **imagen**:

https://drive.google.com/open?id=0B8Y50FxadxpXb2ZOcWoxMUs0TVU

  - La escribimos en un pendrive de 8 ó 16 GB:
    - Desde Windows:

http://www.osforensics.com/tools/write-usb-images.html

Entorno de trabajo (III)

- Si no queremos instalar nada…, podríamos también prepararnos una distribución dentro de un medio USB:
  - Bajamos una **imagen**:

    https://drive.google.com/open?id=0B8Y50FxadxpXb2ZOcWoxMUs0TVU
  - La usamos sabiendo que:

    Nombre de host: *hispania*

    Nombre de usuario: *alumno*

    Contraseña: *alumnoUAL1617*

Entorno de trabajo (IV)

- Finalmente, si alguien tiene especial interés en **trabajar sobre Windows o Mac en el futuro…** aquí vienen unas **orientaciones:**

  - Windows: **Como compilador** recomiendo MinGW (implementación de GCC para Windows)

(Hay también otros como el de la propia Microsoft…)

Y ya, **cualquier editor que nos guste**, como Sublime Text o Notepad++

Entorno de trabajo (IV)

- Finalmente, si alguien tiene especial interés en **trabajar sobre Windows o Mac en el futuro**… aquí vienen unas **orientaciones**:
  - Mac: Se instala el compilador Clang LLVM al instalar el IDE **Xcode** desde la App Store y sus “command-line tools”:

```
xcode-select --install
```

(O se puede poner GCC directamente con “brew” o “macports”)

Y ya, **cualquier editor que nos guste**, como Sublime Text o Gedit

Entorno de trabajo (V)

- Además, aunque no vamos a usarlos, debéis saber que hay muchos IDE's completos y gratuitos para C:
  - Eclipse: https://www.eclipse.org/
  - Code::Blocks: http://www.codeblocks.org
  - NetBeans: https://netbeans.org
- Eso sí, tendremos que enlazarlos con nuestro compilador (y depurador).

Contenidos

- ¿Qué es C?
- Entorno de trabajo
- **¡Hola Mundo! (Y Compilación)**
- Variables y constantes. Ejemplos
- Funciones. Ejemplos
- Punteros (a datos y funciones). Ejemplos
- Memoria y cómo pedirla (y liberarla!)
- Estructuras de control. Ejemplos
- Cómo depurar un programa
- Recibiendo parámetros por consola
- Creando valores aleatorios
- Midiendo tiempos
- Lectura y escritura de archivos de texto
- Aplicación: Creando una librería para operar con matrices

¡Hola Mundo! (Y Compilación) (I)

- Como no podría ser de otra forma… nuestro primer programa C va a ser el típico “Hola Mundo”:

```c
#include <stdio.h>

int main(void){
    printf("¡Hola Mundo!\n");
    return 0;
}
```

archivo1.c

} Así cargamos librerías

stdio.h tiene funciones de entrada/salida.

El archivo va entre “<>” por estar en la ruta del compilador. Con los nuestros usaremos comillas! (Busca en la ruta actual)

***NOTA**: Este código debe ir en un archivo de texto .c.

Los archivos en **C son .c (código) y .h (destinados** por convención **a declaraciones)**

¡Hola Mundo! (Y Compilación) (II)

- Como no podría ser de otra forma… nuestro primer programa C va a ser el típico “Hola Mundo”:

```c
#include <stdio.h>

int main(void){
    printf("¡Hola Mundo!\n");
    return 0;
}
```

archivo1.c

La función “*main*” es la entrada del programa. Aquí empieza la ejecución.

¡Hola Mundo! (Y Compilación) (II)

- Como no podría ser de otra forma… nuestro primer programa C va a ser el típico “Hola Mundo”:

```
1    #include <stdio.h>
2
3    int main(void){
4        printf("¡Hola Mundo!\n");
5        return 0;
6    }
```

archivo1.c

Esta definición no admite argumentos, veremos la otra que sí los admite.

¡Hola Mundo! (Y Compilación) (II)

- Como no podría ser de otra forma… nuestro primer programa C va a ser el típico “Hola Mundo”:

```c
#include <stdio.h>

int main(void){
    printf("¡Hola Mundo!\n");
    return 0;
}
```

archivo1.c

La función “***printf***”, **de stdio** nos permite imprimir por consola (y con formato).

¡Hola Mundo! (Y Compilación) (II)

- Como no podría ser de otra forma… nuestro primer programa C va a ser el típico “Hola Mundo”:

```
1   #include <stdio.h>
2
3   int main(void){
4       printf("¡Hola Mundo!\n");
5       return 0;
6   }
```

Cerramos la función “*main*” devolviendo un valor entero, como requiere la declaración.

archivo1.c

***NOTA**: Por convenio, se espera que un programa que termina bien devuelva “0”.

¡Hola Mundo! (Y Compilación) (III)

- Muy bien… todo se ve sencillo, ¿pero cómo transformamos ese código en un “programa”?
- ¡Pues **compilándolo**!

La forma más simple de compilar el programa es esta:

gcc archivo.c

***NOTA**: El programa resultante, como no hemos indicado un nombre, se llamará “**a.out**” por defecto

¡Hola Mundo! (Y Compilación) (III)

- ¿Y si queremos definir el nombre del programa?
- Pues usamos la opción “-o” (override / sobrescribir):

```
MacBook-Pro-de-Nicolas:Codigo nccruz$ gcc -o programa archivo.c
MacBook-Pro-de-Nicolas:Codigo nccruz$ ./programa
¡Hola Mundo!
```

- (y aquí el orden no importa):

```
MacBook-Pro-de-Nicolas:Codigo nccruz$ gcc archivo.c -o programa
MacBook-Pro-de-Nicolas:Codigo nccruz$ ./programa
¡Hola Mundo!
```

¡Hola Mundo! (Y Compilación) (III)

- ¿**Y si tenemos** un programa más complejo, con **varios archivos de código** (.c y .h)?
- Pues, en general, **tendremos que pasar todos los** archivos **.c** al compilador:

```
gcc -o programa archivo1.c archivo2.c archivo3.c
```

***NOTA**: Puede que sobre decirlo… pero un programa "normal", por muchos archivos de código que tenga, sólo debe contar con un punto de entrada o función "*main*" entre ellos.

¡Hola Mundo! (Y Compilación) (IV)

- Así podemos hacer nuestro código ejecutable y probarlo… ¿Pero **técnicamente qué está ocurriendo**?

- La compilación de un programa C sigue estos pasos:

1. **Preprocesado**: Se eliminan los comentarios y se aplican las directivas *"#include"* insertando los contenidos de los archivos referenciados en el código.
2. **Compilación**: Se pasa el código a lenguaje ensamblador.
3. **Ensamblado**: Se pasa el código ensamblador a lenguaje máquina (pudiendo crear archivos objeto ".o").
4. **Enlazado**: Combina el código objeto de cada unidad de compilación (*.c) e incluye los enlaces a código externo dando lugar a un ejecutable final.

¡Hola Mundo! (Y Compilación) (V)

- Interesante… ¿y sabiendo esto hay algo útil que podamos hacer?

- **En primer lugar**, sabemos que podemos poner **comentarios** sin problemas, no afectarán al código porque se obvian en el pre-procesado:

```
//Esto es un comentario de una sola linea

/* Esto es un comentario de varias
lineas... como en Java todo
*/
```

¡Hola Mundo! (Y Compilación) (V)

- Interesante… ¿y sabiendo esto hay algo útil que podamos hacer?

- **En segundo lugar**, podemos usar también el pre-procesado para **modificar condicionalmente** nuestro código:
  - Vamos a pensar que nuestro programa "¡Hola Mundo!" se vende en 2 versiones: completa y demo. La demo informará de que se trata de un programa de prueba.
  - Podemos crear una u otra sobre el mismo código

¡Hola Mundo! (Y Compilación) (V)

- Interesante… ¿y sabiendo esto hay algo útil que podamos hacer?

- **En segundo lugar**, podemos usar también el pre-procesado para **modificar condicionalmente** nuestro código:

archivo1B.c

```c
#include <stdio.h>

int main(void){
    #ifdef DEMO
        printf("Cómprame\n");
    #else
        printf("¡Hola Mundo!\n");
    #endif
    return 0;
}
```

Este bloque de código sólo se dejará finalmente si GCC recibe un flag "DEMO". Si no, dejará el otro.

¡Hola Mundo! (Y Compilación) (V)

- Interesante… ¿y sabiendo esto hay algo útil que podamos hacer?

- **En segundo lugar**, podemos usar también el pre-procesado para **modificar condicionalmente** nuestro código:

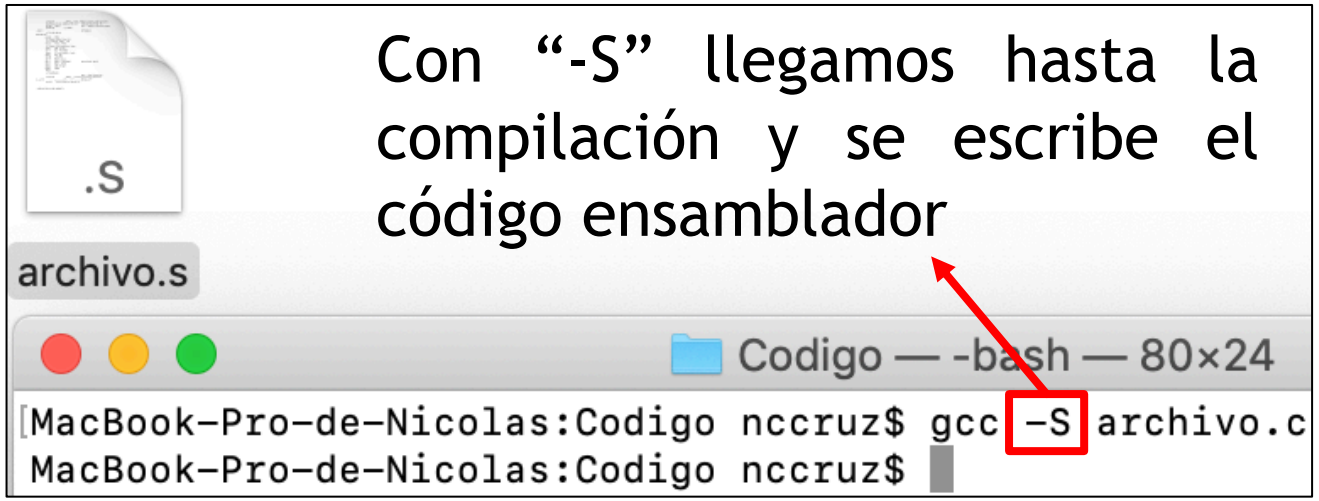
```
MacBook-Pro-de-Nicolas:Codigo nccruz$ gcc -o programa archivo.c
MacBook-Pro-de-Nicolas:Codigo nccruz$ ./programa
¡Hola Mundo!
MacBook-Pro-de-Nicolas:Codigo nccruz$ gcc -o programa archivo.c -DDEMO
MacBook-Pro-de-Nicolas:Codigo nccruz$ ./programa
Cómprame
```

Con “-D” seguido de la etiqueta, la pasamos

¡Hola Mundo! (Y Compilación) (V)

- Interesante… ¿y sabiendo esto hay algo útil que podamos hacer?

- **En tercer lugar**, podemos ver nuestro código en **ensamblador**. Tiene utilidad didáctica y práctica (aunque no vamos a competir con el compilador):



¡Hola Mundo! (Y Compilación) (V)

- Interesante… ¿y sabiendo esto hay algo útil que podamos hacer?

- **En tercer lugar**, podemos ver nuestro código en **ensamblador**. Tiene utilidad didáctica y práctica (aunque no vamos a competir con el compilador):

```
    .section    __TEXT,__text,regular,pure_instructions
    .build_version macos, 10, 14    sdk_version 10, 15
    .globl  _main                   ## -- Begin function main
    .p2align    4, 0x90
_main:                                  ## @main
    .cfi_startproc
## %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    subq    $16, %rsp
```

```
    movl    $0, -4(%rbp)
    leaq    L_.str(%rip), %rdi
    movb    $0, %al
    callq   _printf
    xorl    %ecx, %ecx
    movl    %eax, -8(%rbp)          ## 4-byte Spill
    movl    %ecx, %eax
    addq    $16, %rsp
    popq    %rbp
    retq
    .cfi_endproc
                                    ## -- End function
    .section    __TEXT,__cstring,cstring_literals
L_.str:                                 ## @.str
    .asciz  "\302\241Hola Mundo!\n"
```

¡Hola Mundo! (Y Compilación) (V)

- Interesante… ¿y sabiendo esto hay algo útil que podamos hacer?

- **En cuarto lugar**, podemos **ganar tiempo en la compilación** final de nuestro programa dejando compilados y ensamblados los archivos objeto de las distintas unidades de compilación (archivos *.c):

archivo2_Main.c

```c
#include "archivo2_Lib.h"

int main(void){
    Saludar();
    return 0;
}
```

archivo2_Lib.h

```c
#include <stdio.h>

void Saludar(void);
```

archivo2_Lib.c

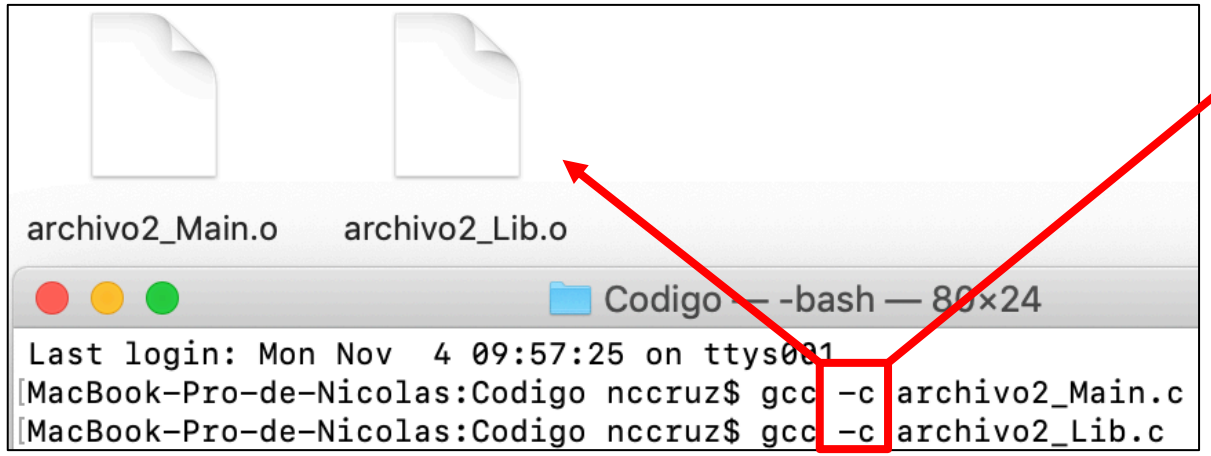
```c
#include "archivo2_Lib.h"

void Saludar(void){
    printf("¡Hola Mundo!\n");
}
```

El archivo está en nuestra ruta, así que usamos comillas en el #include

¡Hola Mundo! (Y Compilación) (V)

- Interesante… ¿y sabiendo esto hay algo útil que podamos hacer?

- **En cuarto lugar**, podemos **ganar tiempo en la compilación** final de nuestro programa dejando compilados y ensamblados los archivos objeto de las distintas unidades de compilación (archivos *.c):



Usamos “-c” para preparar los archivos objeto.

¡Hola Mundo! (Y Compilación) (V)

- Interesante… ¿y sabiendo esto hay algo útil que podamos hacer?

- **En cuarto lugar**, podemos **ganar tiempo en la compilación** final de nuestro programa dejando compilados y ensamblados los archivos objeto de las distintas unidades de compilación (archivos *.c):

```
MacBook-Pro-de-Nicolas:Codigo nccruz$ gcc -o programa archivo2_Main.o archivo2_Lib.o
MacBook-Pro-de-Nicolas:Codigo nccruz$ ./programa
¡Hola Mundo!
```

Ahora pasamos directamente los “.o” para enlazado. Las otras fases están hechas

¡Hola Mundo! (Y Compilación) (V)

- Interesante… ¿y sabiendo esto hay algo útil que podamos hacer?

- **En cuarto lugar**, podemos **ganar tiempo en la compilación** final de nuestro programa dejando compilados y ensamblados los archivos objeto de las distintas unidades de compilación (archivos *.c):

```c
#include "archivo2_Lib.h"

int main(void){
    Saludar();
    printf("Esto es un cambio\n");
    return 0;
}
```

archivo2_Main.c

Ahora sólo tenemos que volver a compilar y ensamblar el archivo “.c” modificado:

```
MacBook-Pro-de-Nicolas:Codigo nccruz$ gcc -c archivo2_Main.c
MacBook-Pro-de-Nicolas:Codigo nccruz$ gcc -o programa archivo2_Main.o archivo2_Lib.o
MacBook-Pro-de-Nicolas:Codigo nccruz$ ./programa
¡Hola Mundo!
Esto es un cambio
```

¡Hola Mundo! (Y Compilación) (V)

- Interesante… ¿y sabiendo esto hay algo útil que podamos hacer?

- **En cuarto lugar**, podemos **ganar tiempo en la compilación** final de nuestro programa dejando compilados y ensamblados los archivos objeto de las distintas unidades de compilación (archivos *.c):

```c
#include "archivo2_Lib.h"

int main(void){
    Saludar();
    printf("Esto es un cambio\n");
    return 0;
}
```

archivo2_Main.c

Los IDE's suelen funcionar así, compilando sólo lo modificado, para acelerar la creación de los ejecutables.

¡Hola Mundo! (Y Compilación) (V)

- Interesante… ¿y sabiendo esto hay algo útil que podamos hacer?

- **En cuarto lugar**, podemos **ganar tiempo en la compilación** final de nuestro programa dejando compilados y ensamblados los archivos objeto de las distintas unidades de compilación (archivos *.c):

```c
#include "archivo2_Lib.h"

int main(void){
    Saludar();
    printf("Esto es un cambio\n");
    return 0;
}
```

archivo2_Main.c

¿Y este *printf*? Si no tenemos cargado “**stdio**”…

- En realidad sí, porque se hace en “archivo2_Lib.h” (“*Propiedad Transitiva*”)
- Aunque **se recomienda** hacer cada “.c” con **inclusiones autocontenidas.**

Contenidos

- ¿Qué es C?
- Entorno de trabajo
- ¡Hola Mundo! (Y Compilación)
- **Variables y constantes. Ejemplos**
- Funciones. Ejemplos
- Punteros (a datos y funciones). Ejemplos
- Memoria y cómo pedirla (y liberarla!)
- Estructuras de control. Ejemplos
- Cómo depurar un programa
- Recibiendo parámetros por consola
- Creando valores aleatorios
- Midiendo tiempos
- Lectura y escritura de archivos de texto
- Aplicación: Creando una librería para operar con matrices

Variables y constantes. Ejemplos (I)

- Sabemos que un lenguaje tipado nos **exige declarar el tipo de datos** de una variable antes de poder usarla.
- La sintaxis para hacerlo en C es prácticamente igual a hacerlo en Java:

```
<Tipo> nombre; // Declaración
<Tipo> nombre = valor; // Declaración & Inicialización
```

- *Las constantes mantienen ese esquema añadiendo la palabra reservada "const":*

~~const <Tipo> nombreVar; // Declaración~~
const <Tipo> nombre = valor; // Declaración & Inicialización

Ahora hay que "saber lo que hay" desde el principio

Se pueden poner varias del mismo tipo seguidas de ","

Variables y constantes. Ejemplos (II)

- Sabemos que un lenguaje tipado nos **exige declarar el tipo de datos** de una variable antes de poder usarla.

```
int valorGlobal = 0;

int main(){
    int valor;
    valor = 6;
    int otroValor = 12;
    //const int yOtro;// Esto
    //yOtro = 24;      // NO
    const int yOtro = 24;
    return 0;
}
```

archivo3.c

Como es lógico, variables y constantes tienen un alcance *(scope) determinado*:

- *valorGlobal* existe para todo el programa (aunque para verlo en otras unidades de compilación (*.c) debemos usar la palabra reservada ***extern***, y para limitarlo ***static***).
- *valor, otroValor* y *yOtro* sólo son visibles en la función *"main"*

***NOTA**: No uses nunca una variable sin inicializar!

Variables y constantes. Ejemplos (III)

- ¿Cómo que ***extern*** y ***static***?

```c
#include <stdio.h>

int valorGlobal = 6;

void Saludar(void);//Presentamos esta funcion
                   //pero su codigo solo se
                   //encontrara enlazando

int main(void){
    Saludar();
    printf("Valor Global: %d\n", valorGlobal);
    return 0;
}
```

archivo4_Main.c

Sobre el ejemplo anterior, así evitamos crear el archivo .h

```c
#include <stdio.h>

void Saludar(void){
    printf("¡Hola Mundo!\n");
    printf("Y mi Valor Global es:%d\n", valorGlobal);
}
```

archivo4_Lib.c

archivo4_Lib.c no se puede compilar por sí mismo, así que no se puede crear el programa

```
MacBook-Pro-de-Nicolas:MisTests nccruz$ gcc -o programa archivo4_Main.c archivo4_Lib.c
archivo4_Lib.c:7:38: error: use of undeclared identifier 'valorGlobal'
        printf("Y mi Valor Global es:%d\n", valorGlobal);
```

Variables y constantes. Ejemplos (III)

- ¿Cómo que ***extern*** y ***static***?

```c
#include <stdio.h>

int valorGlobal = 6;

void Saludar(void);//Presentamos esta funcion
                   //pero su codigo solo se
                   //encontrara enlazando

int main(void){
    Saludar();
    printf("Valor Global: %d\n", valorGlobal);
    return 0;
}
```

archivo4_Main.c

```c
#include <stdio.h>

extern int valorGlobal;

void Saludar(void){
    printf("¡Hola Mundo!\n");
    printf("Y mi Valor Global es: %d\n", valorGlobal);
}
```

archivo4_Lib.c

Ahora, cuando se compila archivo4_Lib.c, GCC sabe que deberá enlazar luego a una variable externa.

```
MacBook-Pro-de-Nicolas:MisTests nccruz$ gcc -o programa archivo4_Main.c archivo4_Lib.c
MacBook-Pro-de-Nicolas:MisTests nccruz$ ./programa
¡Hola Mundo!
Y mi Valor Global es: 6
Valor Global: 6
```

Variables y constantes. Ejemplos (III)

- ¿Cómo que ***extern*** y ***static***?

```c
#include <stdio.h>

int valorGlobal = 6;

void Saludar(void);//Presentamos esta funcion
                   //pero su codigo solo se
                   //encontrara enlazando

int main(void){
    Saludar();
    printf("Valor Global: %d\n", valorGlobal);
    return 0;
}
```

archivo4_Main.c

```c
#include <stdio.h>

static int valorGlobal = 9;

void Saludar(void){
    printf("¡Hola Mundo!\n");
    printf("Y mi Valor Global es: %d\n", valorGlobal);
}
```

archivo4_Lib.c

Ahora, cuando se compila archivo4_Lib.c, GCC sabe que “valorGlobal” se limita a su archivo “.c”.

```
MacBook-Pro-de-Nicolas:MisTests nccruz$ gcc -o programa archivo4_Main.c archivo4_Lib.c
MacBook-Pro-de-Nicolas:MisTests nccruz$ ./programa
¡Hola Mundo!
Y mi Valor Global es: 9
Valor Global: 6
```

Variables y constantes. Ejemplos (IV)

- De hecho, podemos llegar a acotar aún más el alcance de las **variables: ¡dentro de zonas de las funciones!**

```c
#include <stdio.h>

int main(void){
    int valor = 6;
    {
        int valor = 9;
        printf("Valor: %d\n", valor);
    }
    return 0;
}
```

```
Codigo — -bash — 80×24
MacBook-Pro-de-Nicolas:Codigo nccruz$ gcc -o casa archivo5.c
MacBook-Pro-de-Nicolas:Codigo nccruz$ ./casa
Valor: 9
```

archivo5.c

En una zona **delimitada** por "{}", **las variables** definidas ahí quedan **limitadas a esa región.**

Además, **ante igualdad de nombre**, el compilador asume que se hace **referencia a ellas.**

Variables y constantes. Ejemplos (IV)

- De hecho, podemos llegar a acotar aún más el alcance de las **variables: ¡dentro de zonas de las funciones!**

```c
#include <stdio.h>

int main(void){
    int valor = 6;
    {
        int valor = 9;
    }
    printf("Valor: %d\n", valor);
    return 0;
}
```

```
Codigo — -bash — 80×24
MacBook-Pro-de-Nicolas:Codigo nccruz$ gcc -o casa archivo5.c
MacBook-Pro-de-Nicolas:Codigo nccruz$ ./casa
Valor: 6
```

archivo5.c

En una zona **delimitada** por "{}", **las variables** definidas ahí quedan **limitadas a esa región.**

Además, **ante igualdad de nombre**, el compilador asume que se hace **referencia a ellas.**

Variables y constantes. Ejemplos (IV)

- De hecho, podemos llegar a acotar aún más el alcance de las **variables**: **¡dentro de zonas de las funciones!**

```c
#include <stdio.h>

int main(void){
    //int valor = 6;
    {
        int valor = 9;
    }
    printf("Valor: %d\n", valor);
    return 0;
}
```

```
Codigo — -bash — 80×24
MacBook-Pro-de-Nicolas:Codigo nccruz$ gcc -o casa archivo5.c
archivo5.c:8:24: error: use of undeclared identifier 'valor'
        printf("Valor: %d\n", valor);
```

archivo5.c

En una zona **delimitada** por "{}", **las variables** definidas ahí quedan **limitadas a esa región.**

Además, **ante igualdad de nombre**, el compilador asume que se hace **referencia a ellas.**

Variables y constantes. Ejemplos (IV)

- Hay que tener en cuenta que, ante la duda, el compilador se “perderá”, así que dentro del mismo alcance, los **nombres de las variables deben ser únicos (¡aunque sean de distinto tipo!)**.

archivo6.c

```c
#include <stdio.h>

int main(void){
    int valor = 6;
    //int valor = 8;
    float valor = 3.14;
    return 0;
}
```

```
Codigo — -bash — 80×24
MacBook-Pro-de-Nicolas:Codigo nccruz$ gcc -o casa archivo6.c
archivo6.c:6:8: error: redefinition of 'valor' with a different type: 'float' vs
      'int'
        float valor = 3.14;
              ^
archivo6.c:4:6: note: previous definition is here
        int valor = 6;
            ^
1 error generated.
```

Variables y constantes. Ejemplos (V)

- Hay otra forma muy extendida de definir constantes en nuestros programas:

```
#define NOMBRE VALOR
```

- Esta estrategia **difiere de la anterior**:
  - Es una directiva para **preprocesado**: El valor **se inyectará donde corresponda** al crear el ejecutable, y puede perder su nombre en depuración.
  - **No podremos usar la dirección de memoria de la constante** porque será un literal en última instancia.
  - **No** se puede **acotar** su **alcance** o *scope* **ni controlar** su **tipo** con precisión.

Variables y constantes. Ejemplos (V)

- Hay otra forma muy extendida de definir constantes en nuestros programas:

  #define NOMBRE VALOR

```c
#include <stdio.h>

#define NUMERO 5 //Se suele poner
                 //en mayusculas

int main(void){
    printf("Número: %d\n", NUMERO);
    return 0;
}
```

archivo7.c

```
Codigo — -bash — 80×24
MacBook-Pro-de-Nicolas:Codigo nccruz$ gcc -o casa archivo7.c
MacBook-Pro-de-Nicolas:Codigo nccruz$ ./casa
Número: 5
```

Variables y constantes. Ejemplos (VI)

- Muy bien… hay **variables y constantes**, que pueden ser de distintos tipos, y las últimas no pueden modificarse. **¿Pero qué tipos hay?**

- Podemos pensar en estos **tipos simples**:
  - Números enteros & Caracteres
  - Números reales
- Y estos **tipos compuestos**:
  - Arrays
  - Estructuras
  - Uniones
  - Enumeraciones

Variables y constantes. Ejemplos (VII)

Números enteros & Caracteres

- Para **números enteros**, el tipo fundamental es ***int*** (de *Integer*).
- Hay **tipos** derivados **con distinto tamaño** con estos prefijos:
  - *short* (int)
  - *long* (int)
- … y con distinto rango con estos prefijos:
  - *signed*: Por defecto
  - *unsigned*: **Mueve su rango** para empezar en 0 (**Naturales**).

Variables y constantes. Ejemplos (VII)

Esto no es del todo preciso

Números enteros & Caracteres

Nombre	Mínimo	Máximo	Bytes	Formato
(signed) int	-32 767	32 767	2	%i ó %d
(signed) short (int)	-32 767	32 767	2	%hi
(signed) long (int)	−2 147 483 647	2 147 483 647	4	%li
(signed) long long (int)	−9 223 372 036 854 775 807	9 223 372 036 854 775 807	8	%lli
unsigned (int)	0	65 535	2	%u
unsigned short (int)	0	65 535	2	%hu
unsigned long (int)	0	4 294 967 295	4	%lu
unsigned long long (int)	0	18 446 744 073 709 551 615	8	%llu

Variables y constantes. Ejemplos (VII)

Números enteros & Caracteres

- En realidad, **C define sólo el intervalo mínimo** que estos tipos dados deben admitir, pero **puede excederse** según la implementación en cada plataforma.

```c
#include <stdio.h>

int main(void){
    int miEntero = 1000000; // Esto es mas que 32 767
    printf("Mi entero: %d\n", miEntero);
    printf("Bytes de un SIGNED INT: %lu\n", sizeof(int));
    return 0;
}
```

archivo8.c

```
MacBook-Pro-de-Nicolas:Codigo nccruz$ ./casa
Mi entero: 1000000
Bytes de un SIGNED INT: 4
```

Nos da el número de bytes de tipos y variables. Devuelve un valor de tipo "size_t" (definido en <stddef.h>) y equivalente a un unsigned int o unsigned long int según la plataforma (**pero sólo debe usarse para guardar tamaños!**).

Variables y constantes. Ejemplos (VII)

Números enteros & Caracteres

- Además de usar el operador "*sizeof*" para saber los bytes, podemos consultar los tamaños de los tipos enteros de nuestra plataforma con el archivo "**limits.h**":

```c
#include <limits.h>
#include <stdio.h>

int main(void){
    unsigned int enteroMaximo = UINT_MAX;
    printf("El int más grande aquí es: %u\n", enteroMaximo);
    printf("(Y son %lu bytes)\n", sizeof(unsigned int));
    return 0;
}
```

archivo9.c

```
MacBook-Pro-de-Nicolas:Codigo nccruz$ gcc -o casa archivo9.c
MacBook-Pro-de-Nicolas:Codigo nccruz$ ./casa
El int más grande aquí es: 4294967295
(Y son 4 bytes)
```

2^{32} = 4 294 967 296

Variables y constantes. Ejemplos (VII)

Números enteros & Caracteres

- Igualmente, sólo se garantiza que un ***short int*** tiene un tamaño **menor o igual** que un ***int***, que a su vez debe tener un tamaño menor o igual a un ***long int***.
- Esto **puede causar problemas de compatibilidad** de un mismo código entre plataformas.
- A partir del estándar C99, <stdint.h> define tipos enteros con un número fijo de bits:

```
int8_t enteroCon8Bits;
int16_t enteroCon16Bits;
int32_t enteroCon32Bits;
int64_t enteroCon64Bits;
```

Pero no nos alarmemos tampoco… ¡Para este curso podemos quedarnos con los de la próxima tabla!

Variables y constantes. Ejemplos (VII)

Esto es más realista

%x →hex.
%o →octal

Números enteros & Caracteres

Nombre	Mínimo	Máximo	Bytes	Formato
(signed) int	-2 147 483 648	2 147 483 647	4	%i ó %d
(signed) short (int)	-32 768	32 767	2	%hi
(signed) long (int)	-9 223 372 036 854 775 808	9 223 372 036 854 775 807	8	%li
(signed) long long (int)	-9 223 372 036 854 775 808	9 223 372 036 854 775 807	8	%lli
unsigned (int)	0	4 294 967 295	4	%u
unsigned short (int)	0	65 535	2	%hu
unsigned long (int)	0	18 446 744 073 709 551 615	8	%lu
unsigned long long (int)	0	18 446 744 073 709 551 615	8	%llu

Variables y constantes. Ejemplos (VIII)

Números enteros & Caracteres

- Para **caracteres**, el tipo fundamental es ***char*** (de *Character*), aunque realmente, no **son** más que **enteros con semántica** adicional (su valor se asocia implícitamente a un símbolo).

- De hecho, se pueden asignar con el símbolo deseado entre comillas simples, o con el número correspondiente:

```
char unCaracter = 'a';
char otroCaracter = 97;// 'a' es 97
```

Variables y constantes. Ejemplos (VIII)

Números enteros & Caracteres

- El tipo de dato **char** también tiene los derivados con **signed** y **unsigned**:

Nombre	Mínimo	Máximo	Bytes	Formato
(signed) char	-128	127	1	%c
(unsigned) char	0	255	1	%c

- Ah! Aquí un **carácter** es algo muy **distinto a una cadena**!

Variables y constantes. Ejemplos (VIII)

Números enteros & Caracteres

```c
#include <stdio.h>

int main(void){
    char a = 'a';
    printf("El caracter es: %c (o %d)\n", a, a);
    printf("Tiene un tamaño de %lu byte\n", sizeof(a));
    return 0;
}
```

```
Codigo — -bash — 80×24
MacBook-Pro-de-Nicolas:Codigo nccruz$ gcc -o casa archivo10.c
MacBook-Pro-de-Nicolas:Codigo nccruz$ ./casa
El caracter es: a (o 97)
Tiene un tamaño de 1 byte
```

Conversiones implícitas… Es verdad que C es un lenguaje de tipado débil.

archivo10.c

Variables y constantes. Ejemplos (VIII)

Números reales

Varían el modo de representación y pueden ajustarse. Por ejemplo:
float a = 5.6763 con %.3f
muestra: 5.676

- Tenemos:

Nombre	Mínimo representable	Máximo representable	Bytes	Formato
float	1.175494e-38	3.402823e+38	4	%f %F %g %G %e %E %a %A
double	2.225074e-308	1.797693e+308	8	%lf %lF %lg %lG %le %lE %la %lA
long double	3.362103e-4932	1.189731e+4932	16	%Lf %LF %Lg %LG %Le %LE %La %LA

***NOTA**: Cuidado, memoria aparte, manejar valores en coma flotante usa las unidades dedicadas de la CPU, y son mucho más lentas que las de enteros.

- Siguen los estándares IEEE 754 en simple, doble y cuádruple precisión.

Variables y constantes. Ejemplos (VIII)

Números reales

Al igual que *limits.h* se centra en enteros, *float.h* tiene definiciones para reales

```c
#include <float.h>
#include <stdio.h>

int main(void){
    float a = 1.0;
    printf("Un float tiene %lu bytes, y abarca [%e, %e]\n", sizeof(a), FLT_MIN, FLT_MAX);
    double b = 2.0;
    printf("Un double tiene %lu bytes, y abarca [%le, %le]\n", sizeof(b), DBL_MIN, DBL_MAX);
    long double c = 3.0;
    printf("Un long double tiene %lu bytes, y abarca [%Le, %Le]\n", sizeof(c), LDBL_MIN, LDBL_MAX);
    return 0;
}
```

```
Codigo — -bash — 80×24
MacBook-Pro-de-Nicolas:Codigo nccruz$ gcc -o casa archivo11.c
MacBook-Pro-de-Nicolas:Codigo nccruz$ ./casa
Un float tiene 4 bytes, y abarca [1.175494e-38, 3.402823e+38]
Un double tiene 8 bytes, y abarca [2.225074e-308, 1.797693e+308]
Un long double tiene 16 bytes, y abarca [3.362103e-4932, 1.189731e+4932]
```

También funciona con una variable particular, no solo tipos

archivo11.c

Variables y constantes. Ejemplos (IX)

- En Java podemos **convertir entre tipos**… ¿y en C? **¡Claro!**

- Hay **2 formas de conversión** de tipos:
  - **Implícita**: Se hace **automáticamente** por el compilador cuando se requiere, como cuando en una expresión se mezclan tipos distintos. Se intenta evitar perder información pasando al tipo más grande, aunque se pueden obviar signos o desbordar variables.

  - **Explícita**: Se solicita directamente por el programador (*casting*), y se hace como en Java:

    ```
    (tipoDeseado) expresión
    ```

Variables y constantes. Ejemplos (IX)

- En Java podemos **convertir entre tipos**… ¿y en C? **¡Claro!**

archivo12.c

```c
#include <stdio.h>

int main(){
    int a = -2;
    signed short int b = 3;
    int c = a + b;
    printf("%d + %hi = %d\n", a, b, c);
    //------------------------
    float valor = a/b;
    printf("%d/%hi = %f :-/\n", a, b, valor);
    valor = (float) (a / b);
    printf("%d/%hi = %f :-/\n", a, b, valor);
    valor = ((float) a) / b;
    printf("%d/%hi = %f :-)\n", a, b, valor);
    valor = (float) a / (float) b;
    printf("%d/%hi = %f :-)\n", a, b, valor);
    return 0;
}
```

Codigo — -ba

```
-2 + 3 = 1
-2/3 = 0.000000 :-/
-2/3 = 0.000000 :-/
-2/3 = -0.666667 :-)
-2/3 = -0.666667 :-)
```

Conversiones implícitas

Conversiones explícitas

Cuidado, porque **se priorizan con paréntesis** y es fácil despistarse….

Variables y constantes. Ejemplos (X)

- ¿Y **esos símbolos** de +, / **qué son**? **Operadores:**

(Menos es más)

Operadores de Aritméticos: Su resultado es un número

Símbolo	Nombre	Sintaxis	Significado	Prioridad
+	Suma	a + b	a + b	**3**
-	Resta	a - b	a - b	3
*	Multiplicación	a * b	a * b	2
/	División	a / b	a / b	2
%	Resto	a % b	a en módulo b	2
-	Opuesto	-a	Invertir signo / Número opuesto	2

¡Cuidado con los negativos! No es 100% el módulo matemático, el estándar pide que: a=(a/b)*b+a%b

Info: https://stackoverflow.com/questions/11720656/modulo-operation-with-negative-numbers

***NOTA**: Usando paréntesis priorizamos de dentro a afuera

Variables y constantes. Ejemplos (X)

Operadores de Aritméticos: Su resultado es un número

```c
#include <stdio.h>

int main(){
    int a = 1, b = 3;
    printf("%d + %d = %d\n", a, b, a + b);
    printf("%d - %d = %d\n", a, b, a - b);
    printf("%d * %d = %d\n", a, b, a * b);
    printf("%d / %d = %d\n", a, b, a / b);
    printf("%d %% %d = %d\n", a, b, a % b);
    printf("-%d = %d; - %d = %d\n", a, -a, b, -b);
    printf("-%d + %d = %d\n", a, b, -a + b);
    printf("-(%d + %d) = %d\n", a, b, -(a + b));
    return 0;
}
```

```
1 + 3 = 4
1 - 3 = -2
1 * 3 = 3
1 / 3 = 0
1 % 3 = 1
-1 = -1; - 3 = -3
-1 + 3 = 2
-(1 + 3) = -4
```

¿Enteros…?
División entera

Sí, para poner "%" hay que indicarlo con un "%"

Priorizando con paréntesis

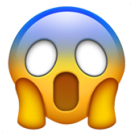
archivo13.c

Variables y constantes. Ejemplos (X)

Operadores Relacionales: Comparaciones ciertas o falsas*

Símbolo	Nombre	Sintaxis	Significado	Prioridad
<	Menor que	a < b	Verdadero si a < b; falso si no	**5**
>	Mayor que	a > b	Verdadero si a > b; falso si no	5
<=	Menor o igual que	a <= b	Verdadero si a <= b; falso si no	5
>=	Mayor o igual que	a >= b	Verdadero si a >= b; falso si no	5
==	Igual que	a == b	Verdadero si a = b; falso si no	6
!=	Distinto a	a != b	Verdadero si a <> b; falso si no	6

*



¿Y eso qué es? No hemos visto **ningún tipo simple** para valores **booleanos**: C históricamente nunca lo ha tenido, se usan números: **0 para FALSO** y **<>0 para CIERTO**. Pero **desde** el estándar **C99** se define el **tipo bool** en **stdbool.h** con *true* = 1 y *false* = 0.

Variables y constantes. Ejemplos (X)

Operadores Relacionales: Su resultado es verdadero o falso

```c
#include <stdio.h>

int main(void){
    int a = 6, b = 9;
    printf("%d < %d: %d\n", a, b, a<b);
    printf("%d > %d: %d\n", a, b, a>b);
    printf("%d <= %d: %d\n", a, b, a<=b);
    printf("%d >= %d: %d\n", a, b, a>=b);
    printf("%d == %d: %d\n", a, b, a==b);
    printf("%d != %d: %d\n", a, b, a!=b);
    return 0;
}
```

archivo14.c

```
6 < 9: 1
6 > 9: 0
6 <= 9: 1
6 >= 9: 0
6 == 9: 0
6 != 9: 1
```

Variables y constantes. Ejemplos (X)

Operadores Lógicos: Su resultado es verdadero o falso

Símbolo	Nombre	Sintaxis	Significado	Prioridad
&&	AND lógico	a && b	Verdadero si a y b son verdaderas	10
||	OR lógico	a || b	Verdadero si a o b son verdaderas	11
!	NOT lógico	!a	Verdadero si a es falso y viceversa	1

```c
#include <stdio.h>

int main(void){
    int a = 0, b = 1;
    printf("%d && %d = %d\n", a, b, a && b);
    printf("%d || %d = %d\n", a, b, a || b);
    printf("!%d = %d ; !%d = %d\n", a, !a, b, !b);
    printf("%d && %d = %d\n", 7, 1, 7 && 1);
    printf("!%d = %d; !%d = %d; !%d = %d\n", 7, !7, -3, !-3, 0, !0);
    return 0;
}
```

archivo15.c

```
0 && 1 = 0
0 || 1 = 1
!0 = 1 ; !1 = 0
7 && 1 = 1
!7 = 0; !-3 = 0; !0 = 1
```

Pues sí, los números se ven como booleanos…

Variables y constantes. Ejemplos (X)

Operadores de Asignación: Modifican una variable

	Símbolo	Nombre	Sintaxis	Significado	Prioridad
Binarios	=	Asignación	a = b	a = b	**0**
	+=	Incremento	a += b	a = a + b	3
	-=	Decremento	a -= b	a = a - b	3
	*=	Producto-Asignación	a *= b	a = a * b	2
	/=	División-Asignación	a /= b	a = a / b	2
	%=	Módulo-Asignación	a %= b	a = a Módulo b	2
Unarios	++	(Pre/post) Incremento	a++ ó ++a	a = a + 1 (antes o después de evaluar el resto)	1
	--	(Pre/post) Decremento	a-- ó --a	a = a – 1 (antes o después de evaluar el resto)	1

Variables y constantes. Ejemplos (X)

Operadores de Asignación: Modificar una variable

archivo16.c

```c
#include <stdio.h>

int main(){
    int a = 1;
    int b = a++;
    int c = ++a;
    printf("A = %d; B = %d; C = %d\n", a, b, c);
    int d;
    d = a+=b;
    printf("A = %d; B = %d; D = %d\n", a, b, d);
    d *= a;
    printf("D = %d\n", d);
    d /= a;
    printf("D = %d\n", d);
    b -= a;
    printf("B = %d\n", b);
    b %= c;
    printf("B = %d\n", b);
    int e = a?16:32;
    printf("Como A=%d, E=%d\n", a, e);
    return 0;
}
```

```
A = 3; B = 1; C = 3
A = 4; B = 1; D = 4
D = 16
D = 4
B = -3
B = 0
Como A=4, E=16
```

Operador Condicional o Ternario

a ? b : c;

- Si *a* es cierto: *b*
- Si *a* es falso: *c*

Variables y constantes. Ejemplos (X)

Operadores a nivel de bit (*bitwise*): Trabajan bit a bit

Símbolo	Nombre	Sintaxis	Prioridad
&	AND bit a bit	a & b	7
|	OR bit a bit	a | b	9
^	XOR bit a bit	a ^ b	8
~	NOT bit a bit	~a	1
>>	Desplazamiento a la derecha de bits	a >> b	4
<<	Desplazamiento a la izquierda de bits	a << b	4

→ Como dividir por 2^b

→ Como multiplicar por 2^b

Variables y constantes. Ejemplos (X)

Operadores a nivel de bit (*bitwise*): Trabajan bit a bit

```c
#include <stdio.h>

int main(){
    printf("240 & 170 = %d\n", 240 & 170);
    printf("8 | 64 = %d\n", 8 | 64);
    printf("36 ^ 255 = %d\n", 36 ^ 255);
    printf("~7 = %d\n", ~7);
    printf("3 << 2 = %d\n", 3 << 2);
    printf("16 >> 2 = %d\n", 16 >> 2);
    return 0;
}
```

archivo17.c

```
240 & 170 = 160
8 | 64 = 72
36 ^ 255 = 219
~7 = -8
3 << 2 = 12
16 >> 2 = 4
```

```
   11110000   (240)
&
   10101010   (170)
-------------------
   10100000   (160)
```

(...)

00010000 (16) =>(2) 00000100 (4)

Variables y constantes. Ejemplos (XI)

- Muy bien… hay **variables y constantes**, que pueden ser de distintos tipos, y las últimas no pueden modificarse. **¿Pero qué tipos hay?**

- Podemos pensar en estos **tipos simples**:
  - Números enteros & Caracteres
  - Números reales
- Y estos **tipos compuestos**:
  - Arrays
  - Estructuras
  - Uniones
  - Enumeraciones

Se definen sobre múltiples elementos de otro tipo

Variables y constantes. Ejemplos (XII)

Arrays

- Son una **colección N-Dimensional** de elementos de un **mismo tipo contiguos en memoria**.
- Se **declaran** de la siguiente forma:

```
tipo nombreArray[D1]…[DN];
```

Tamaño de D_1, definible en [1, T_1] (Natural)
Se **indexa** en [0, T_1-1]

Antes se debía conocer en compilación el tamaño de cada dimensión, **ya se pueden usar variables** (extensión **VLA**)

Otro tipo **simple** o **compuesto**

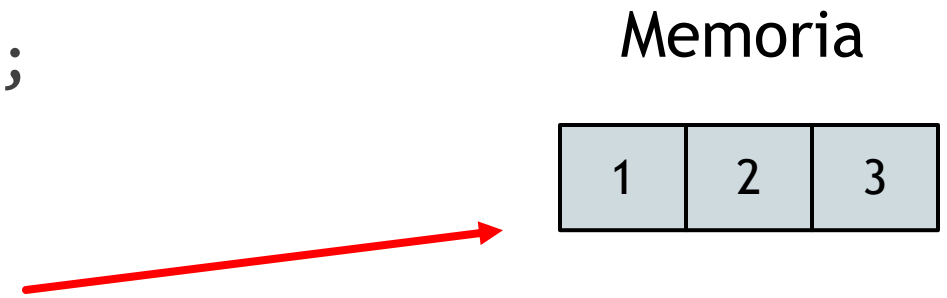
- Se **accede** a sus elementos con los **operadores []**.

Variables y constantes. Ejemplos (XII)

Arrays

- ¿Cómo que contiguos en memoria?

```
tipo array1D[3];
array1D[0] = 1;
array1D[1] = 2;
array1D[2] = 3;
```

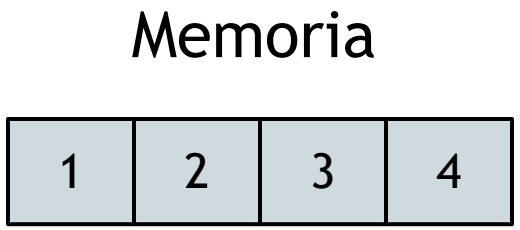


Variables y constantes. Ejemplos (XII)

Arrays

- ¿Cómo que contiguos en memoria?

```
tipo array2D[2][2];
array2D[0][0] = 1;
array2D[0][1] = 2;
array2D[1][0] = 3;
array2D[1][1] = 4;
```



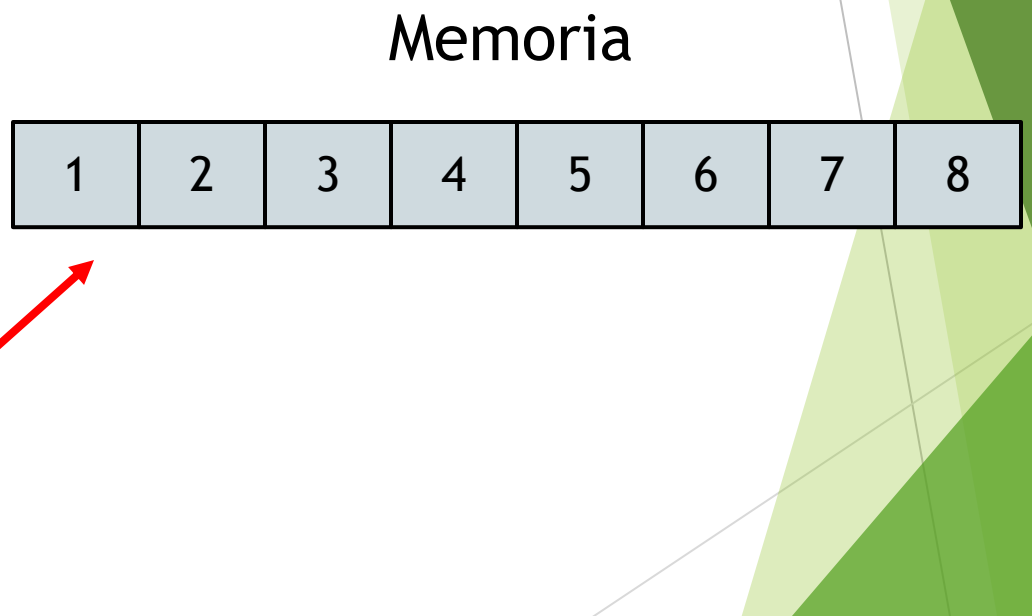
Números de “fila” y “columna” de una hipotética matriz

Variables y constantes. Ejemplos (XII)

Arrays

- ¿Cómo que contiguos en memoria?

```
tipo array3D[2][2][2];
array3D[0][0][0] = 1;
array3D[0][0][1] = 2;
array3D[0][1][0] = 3;
array3D[0][1][1] = 4;
…
array3D[1][1][1] = 8;
```



Variables y constantes. Ejemplos (XII)

Arrays

- También se pueden **declarar e inicializar** a la vez:

`tipo nombreArray[D1]…[DN] = {{a, b},…}` (Los no indicados quedan a 0)

`tipo nombreArray[]…[D] = {{a, b},…};` (El tamaño se ajusta solo, pero **sólo podemos obviar la primera dimensión**)

`tipo nombreArray[D1]…[DN] = {};` (Todo a 0)

`tipo nombreArray[D1]…[DN] = {X};` (Si X<>0, el primer elemento es X y el resto 0)

Variables y constantes. Ejemplos (XII)

Arrays

- **Cuidado**, porque los **arrays no se pueden redimensionar ni asignar** unos a otros.
- Para copiar uno en otro, **lo que en tipos simples se hace con** el operador **"="**, en C debe hacerse **a mano o** con funciones como **memcpy(destino, origen, num_bytes**) de **string.h**.
- **Tampoco** podrán **ser** el **resultado** directo **de una función**.
- ¿Sabías que un "**String**" es un **array de char que termina con el carácter '\0' (0)**?

Variables y constantes. Ejemplos (XII)

Arrays

archivo18.c

```c
#include <stdio.h>

int main(){
    int array1D[3];//No inicializo
    printf("A1D=[%d, %d, %d]\n", array1D[0], array1D[1], array1D[2]);
    printf("El tamaño de A1D es de %lu bytes: 3x%lu\n", sizeof(array1D), sizeof(int));

    int array1D_B[3] = {0};//Inicializo todo a 0
    printf("A1D_B=[%d, %d, %d]\n", array1D_B[0], array1D_B[1], array1D_B[2]);
    int array1D_C[3] = {0, 1};//Inicializo solo los 2 primeros
    printf("A1D_C=[%d, %d, %d]\n", array1D_C[0], array1D_C[1], array1D_C[2]);
    //array1D = array1D_B;//PROHIBIDO

    int array2D[ ][2] = {{1, 2}, {3, 4}};

    printf("A2D=\t[%d, %d,\n\t %d %d]\n", array2D[0][0], array2D[0][1],
                                         array2D[1][0], array2D[1][1]);

    int array3D[2][2][2] = {{{1, 2}, {3, 4}}, {{5, 6}, {7, 8}}};

    printf("A3D[0]=\t[%d, %d,\n\t %d %d]\n", array3D[0][0][0], array3D[0][0][1],
                                         array3D[0][1][0], array3D[0][1][1]);
    printf("A3D[1]=\t[%d, %d,\n\t %d %d]\n", array3D[1][0][0], array3D[1][0][1],
                                         array3D[1][1][0], array3D[1][1][1]);
    return 0;
}
```

```
A1D=[0, -445477992, 32766]
El tamaño de A1D es de 12 bytes: 3x4
A1D_B=[0, 0, 0]
A1D_C=[0, 1, 0]
A2D=    [1, 2,
         3 4]
A3D[0]= [1, 2,
         3 4]
A3D[1]= [5, 6,
         7 8]
```

Podemos obviar la 1ª dim.

Tabulamos con \t

Partimos líneas de código sin ;

Variables y constantes. Ejemplos (XIII)

Estructura

- Tipo que se define como una agrupación de una serie de campos **de otros tipos** (permitiendo **heterogeneidad**).
- Se pueden ver **como "objetos"** Java **sin protección de atributos ni métodos** directamente asociados, y se **definen**:

```
struct Estructura{
   tipoA campo1;
   tipoB campo2;
   …
};
```

No olvides el ';' tras la definición

Variables y constantes. Ejemplos (XIII)

Estructura

- Una vez definida una estructura, podemos **declarar** variables **de ese tipo así**:

```
struct Estructura nombreVariable;
```

- Dada una variable de ese tipo, podremos **acceder a cada campo con** el operador ".":

```
nombreVariable.campo1;
nombreVariable.campo2; …
```

Variables y constantes. Ejemplos (XIII)

Estructura

- Para definir los valores de una variable estructura podemos acceder **manualmente a cada campo o…**:

```
struct Estructura var = {valCampo1, valCampo2…};
```

```
struct Estructura otraVar;
otraVar (struct Estructura) = {.campo1 = valCampo1, .campo2
= valCampo2…}
```

Para C99

- **Asignar y copiar estructuras no es problemático** como con los arrays. Se puede hacer naturalmente con el operador "=". **Comparar no se permite** y debe hacerse manualmente campo a campo.

Variables y constantes. Ejemplos (XIII)

Estructura

- ¿Y una estructura **puede tener tipos como arrays** para sus campos?

- **Sí**, sin problema, y puede tener también hasta otras estructuras, uniones y enumeraciones.

- Igualmente, podremos usar estructuras como tipos con los que **crear arrays** (y uniones).

Variables y constantes. Ejemplos (XIII)

Estructura

- Realmente se hace pesado poner *struct* antes de declarar variables de nuestro tipo… Para darles una entidad más "compacta", podemos usar ***typedef***:

```
typedef struct Estructura{
    tipoA campo1;
    tipoB campo2;
} MiEstructura;
```

Ahora podemos declarar variables de ese tipo usando "*MiEstructura*" como su nombre de tipo.

- Ah, y ***typedef*** nos permite crear alias sobre tipos concretos en general, **no sólo de estructuras**.

Variables y constantes. Ejemplos (XIII)

Estructura

archivo19.c

```c
#include <stdio.h>

typedef struct Estructura{
    int campo1;
    int campo2[3];
} Estructura;

int main(){
    struct interna{//Definicion local
        int campo1;
        Estructura campo2;
    };

    struct interna miA = {0, {1, {2, 3, 4}}};
    struct interna miB = miA;
    miB.campo2.campo2[2] = 66;
    printf("miA: Campo 1: %d; Campo 2: %d, [%d %d %d]\n",
        miA.campo1, miA.campo2.campo1, miA.campo2.campo2[0],
        miA.campo2.campo2[1], miA.campo2.campo2[2]);
    printf("miB: Campo 1: %d; Campo 2: %d, [%d %d %d]\n",
        miB.campo1, miB.campo2.campo1, miB.campo2.campo2[0],
        miB.campo2.campo2[1], miB.campo2.campo2[2]);
    return 0;
}
```

```
MacBook-Pro-de-Nicolas:Codigo nccruz$ ./casa
miA: Campo 1: 0; Campo 2: 1, [2 3 4]
miB: Campo 1: 0; Campo 2: 1, [2 3 66]
```

Variables y constantes. Ejemplos (XIV)

Unión

- Tipo que define una **serie de campos posibles** que se almacenan **en** la **misma** zona de **memoria**, por lo que:
  - **En un** cierto **instante**, **sólo uno de ellos es válido**.
  - El **tamaño** del tipo dato **es igual al del campo** que **mayor** espacio necesite: Ahorramos espacio en ciertos casos.

- Se **definen** así:

```
union MiUnion{
    tipoA campo1;
    tipoB campo2;
    …
};
```

No olvides el ‘;’ tras la definición

Variables y constantes. Ejemplos (XIV)

Unión

- **No se registra qué campo se está** usando con un significado coherente, por lo que eso **debe saberlo el programador**.
- Por lo demás, **se usan como las estructuras**:
  - El operador de **acceso** a los campos es con un “.”.
  - Se pueden asignar con el **operador “=” sin problema**.
  - Su definición se **puede anidar y contener otros tipos**.
  - Hay que declararlas con la palabra reservada ***union*** (a no ser que usemos **typedef**…).
  - Se **inicializan a mano** (o fijando el **campo deseado en C99**)

Variables y constantes. Ejemplos (XIV)

Unión

archivo20.c

```c
#include <stdio.h>

typedef union LaUnion{
    int campo1;
    int campo2[3];
    union{
        int campoA;
        int campoB;
    } campo3;
} MiUnion;

int main(){
    union LaUnion a = {.campo2={1, 2, 3}}; // C99
    MiUnion b;
    b.campo3.campoA = 7; // A mano
    union LaUnion c = b;
    printf("A: %d, %d, %d\n", a.campo2[0], a.campo2[1],
        a.campo2[2]);
    printf("B: %d\n", b.campo3.campoA);
    printf("C: %d\n", c.campo1);
    c = a;
    printf("C: %d, %d, %d\n", c.campo2[0], c.campo2[1],
        c.campo2[2]);
    return 0;
}
```

```
A: 1, 2, 3
B: 7
C: 7
C: 1, 2, 3
```

Leemos otro campo y vemos lo mismo…

Claro, es una unión y hay un bloque de memoria común!

Variables y constantes. Ejemplos (XIV)

Enumeración

- Se trata de un tipo orientado a asignar **nombres asociados a una serie de constantes enteras**. Las variables de este tipo podrán contener cualquiera de esos "enteros con nombre".

- Se **definen** así:

```
enum Enumeracion{
    Nombre1,
    Nombre2
};
```

No olvides el ';' tras la definición

Variables y constantes. Ejemplos (XIV)

Enumeración

- Hay que declararlas con la palabra reservada ***enum*** (a no ser que usemos **typedef**…).
- Las instancias **se asignan sin problemas con '='**
- **Si no ponemos ningún valor, el compilador asocia** el 0 al primer nombre, el 1 al segundo…
- **Si ponemos algunos valores y otros no, los nombres reciben automáticamente el valor previo más 1.**
- **Distintos nombres pueden tener asociado el mismo valor.**
- Los **nombres asociados a enumeraciones** en un mismo ámbito (scope) **no se pueden repetir**, aunque sean enumeraciones diferentes.

Variables y constantes. Ejemplos (XIV)

Enumeración

archivo21.c

```c
#include <stdio.h>

typedef enum BOOL{
    FALSO,
    CIERTO
} bool;

int main(){
    typedef enum Finde{
        Sabado = 6,
        Domingo
    } Finde;
    enum BOOL miVar = CIERTO;
    bool otra = FALSO;
    Finde mio = Domingo;
    Finde otro = mio;
    otro = Sabado;
    printf("Cierto: %d, Falso: %d\n", miVar, otra);
    printf("Finde: Sabado = %d, Domingo = %d\n", otro, mio);
    return 0;
}
```

```
Cierto: 1, Falso: 0
Finde: Sabado = 6, Domingo = 7
```

Contenidos

- ¿Qué es C?
- Entorno de trabajo
- ¡Hola Mundo! (Y Compilación)
- Variables y constantes. Ejemplos
- **Funciones. Ejemplos**
- Punteros (a datos y funciones). Ejemplos
- Memoria y cómo pedirla (y liberarla!)
- Estructuras de control. Ejemplos
- Cómo depurar un programa
- Recibiendo parámetros por consola
- Creando valores aleatorios
- Midiendo tiempos
- Lectura y escritura de archivos de texto
- Aplicación: Creando una librería para operar con matrices

Funciones. Ejemplos (I)

- Las **funciones** son **trozos delimitados de código** y **etiquetados** (con nombre) **que realizan operaciones concretas** y en los **que** se **estructuran** los programas.
- Pueden **recibir parámetros** y **devolver resultados**.
- Tienen asociado un ámbito o ***scope*** **propio e independiente** incluso entre llamadas.
- Se declaran así:

Nombrar los parámetros en la declaración es realmente opcional

```
tipoResultado nombreFuncion(tipoParametro par1,…);
```

Debe ser único en el scope (No hay **overloading** (sobrecarga) como en Java o C++)

Funciones. Ejemplos (II)

- Los **tipos que se pueden usar** al declarar una función son todos los vistos anteriormente (con excepción de los arrays, que se usan de forma distinta).

- **Si no devolverá ningún resultado** directo se debe poner ***void*** (vacío) como el tipo del resultado.

- **Si no recibirá ningún parámetro** directo, se debe poner ***void*** en el lugar de los parámetros.

Funciones. Ejemplos (II)

- Estas son algunas declaraciones válidas:

```
int Sumar(int, int);

void Imprimir(float num);

struct Fecha ObtenerFechaActual(void);

void EscribirHora(void);
```

Funciones. Ejemplos (III)

- Los **parámetros** que recibe una función en C **son siempre una copia** de los que se le dan (**paso por valor**): **los cambios** que se hagan sobre ellos **no afectan** al scope desde el que se llamó (veremos cómo lograrlo…).

- En una función **se pueden crear y usar variables**, pero **su vida está ligada a la** propia **llamada**, así que **se perderán** los valores al salir de la función.

- El **resultado** devuelto por una función **se copiará** (**copia de valor**) para que se pase a quien llamó.

Funciones. Ejemplos (IV)

- La definición de una función se hace así:

```
tipo Sumar(tipo prim, tipo seg){
   … // Operaciones
   return variable_tipo;
}
```

Podemos devolver en cualquier punto de la función, pero eso implicará salir de la misma.

Si nuestra función no devuelve un resultado directo, "***return***" también nos sacará de la misma en cualquier punto.

Funciones. Ejemplos (IV)

- La definición de una función se hace así:

```
tipo Sumar(tipo prim, tipo seg){
   … // Operaciones
   return variable_tipo;
}
```

Podríamos usar también cualquier valor global accesible, por lo que los parámetros no son siempre “todo”.

Podríamos escribir también en cualquier valor global accesible, por lo que el resultado puede ser más de uno y no estar directamente definido.

Funciones. Ejemplos (V)

- Una función se puede declarar y definir a la vez, pero el **compilador requerirá** por lo menos **“conocer”** una función cuando se use:
  - Es igual que el código (definición) esté en otro punto posterior, el compilador debe estar “avisado”.
  - Se suelen **declarar** las funciones **en los archivos “.h”** y ya **implementarlas en los archivos “.c”**.
- Una función puede llamarse a sí misma independientemente: **Recursividad**

Funciones. Ejemplos (V)

- El hecho de que las funciones tengan un nombre global en el programa puede ser problemático, así que podemos **limitar una función a su unidad de compilación como con las variables**:

```
static tipoResultado nombreFuncion(…);
```

- También podemos hacer **variables persistentes** que no se pierden **entre llamadas** con la misma palabra reservada:

```
void Sumar(tipo prim, tipo seg){
   static tipo var = val_inicial;
   …
```

Misma palabra, distinto concepto… y cuidado que esto **¡rompe la modularidad de las funciones!** (Peligro para paralelismo)

Funciones. Ejemplos (VI)

archivo22.c

```c
#include <stdio.h>

void Saludar(int);

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

int main(){
    int miA = 6;
    Saludar(miA);
    Saludar(miA);
    Saludar(miA);
    printf("%d + 3 = %d\n", miA, sumar(miA, 3));
    return 0;
}

void Saludar(int a){
    a++;
    int b = 5;
    static int val = 0;
    printf("Hola %d; A++ = %d; B = %d\n", val, a, b);
    val++;
    b++;
}
```

Si quitamos esto, el compilador se pierde al ver las llamadas a una función "desconocida" en el *main*

La variable *miA* no se está modificando aquí realmente

Este valor persiste entre llamadas.

```
Hola 0; A++ = 7; B = 5
Hola 1; A++ = 7; B = 5
Hola 2; A++ = 7; B = 5
6 + 3 = 9
```

Contenidos

- ¿Qué es C?
- Entorno de trabajo
- ¡Hola Mundo! (Y Compilación)
- Variables y constantes. Ejemplos
- Funciones. Ejemplos
- **Punteros (a datos y funciones). Ejemplos**
- Memoria y cómo pedirla (y liberarla!)
- Estructuras de control. Ejemplos
- Cómo depurar un programa
- Recibiendo parámetros por consola
- Creando valores aleatorios
- Midiendo tiempos
- Lectura y escritura de archivos de texto
- Aplicación: Creando una librería para operar con matrices

Punteros (a datos y funciones) (I)

- El código de los programas y sus datos se cargan en memoria… y **¡C nos permite leer y usar directamente (apuntar) esas direcciones!**

- Para apuntar tenemos los punteros, que son un tipo de dato especial que se puede asociar a:
  - Otro tipo de dato (**Puntero a datos**)
  - Una función (**Puntero a función**)

- **Un puntero es un valor**, como el que puede haber en una variable *int* o *float*, pero **cuyo significado es una dirección de memoria, y** con el que **se puede operar**.

Punteros (a datos y funciones) (I)

- El código de los programas y sus datos se cargan en memoria… y **¡C nos permite leer y usar directamente (apuntar) esas direcciones!**

- Para apuntar tenemos los punteros, que son un tipo de dato especial que se puede asociar a:
  - Otro tipo de dato (**Puntero a datos**)
  - Una función (**Puntero a función**)

- **Un puntero es un valor, como el que puede haber en una variable** *int* o *float*, **pero cuyo significado es una dirección de memoria, y** con el que **se puede operar.**

Punteros (a datos y funciones) (I)

- **Este concepto**, propio de un lenguaje de nivel medio de abstracción como C, **suele resultar muy lioso** para los aprendices.

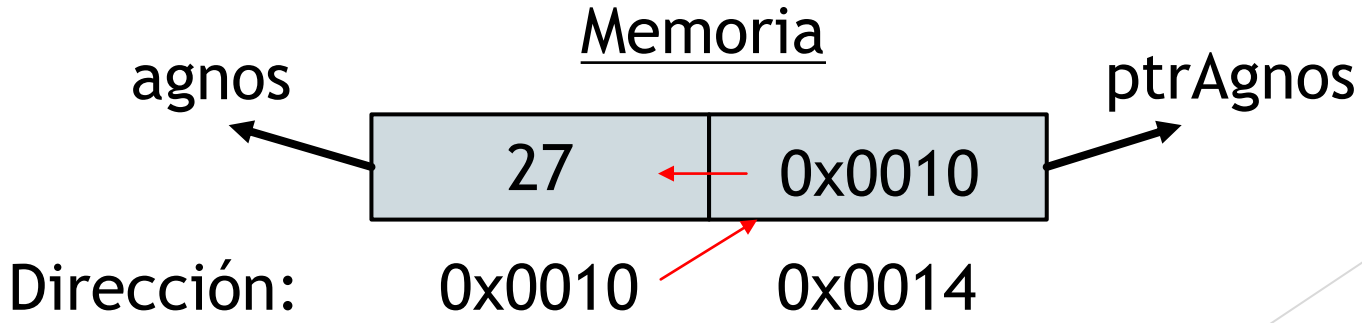
- Pensemos que tenemos una variable de tipo de dato entero, *int agnos*, en el que guardamos una edad (semántica).

- Podemos tener también **una variable** de tipo "puntero a entero" **cuyo contenido es** una dirección de memoria (semántica), **una "flecha",** a la variable anterior, *agnos*.

Punteros (a datos y funciones) (I)

- Vamos a verlo más gráficamente:

```
int agnos = 27;

Puntero_A_Entero ptrAgnos = Dirección_de_agnos;
```



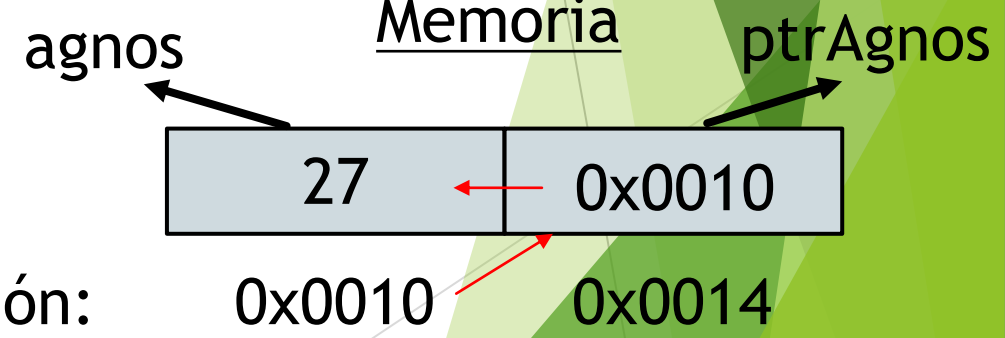
Punteros (a datos y funciones) (II)

- Un puntero se declara con el símbolo '*':

```
tipo * nombreVarPuntero;
```

Por lo que el ejemplo previo se actualizaría así:

```
int agnos = 27;

int *ptrAgnos = Dirección_de_agnos;
```



Punteros (a datos y funciones) (II)

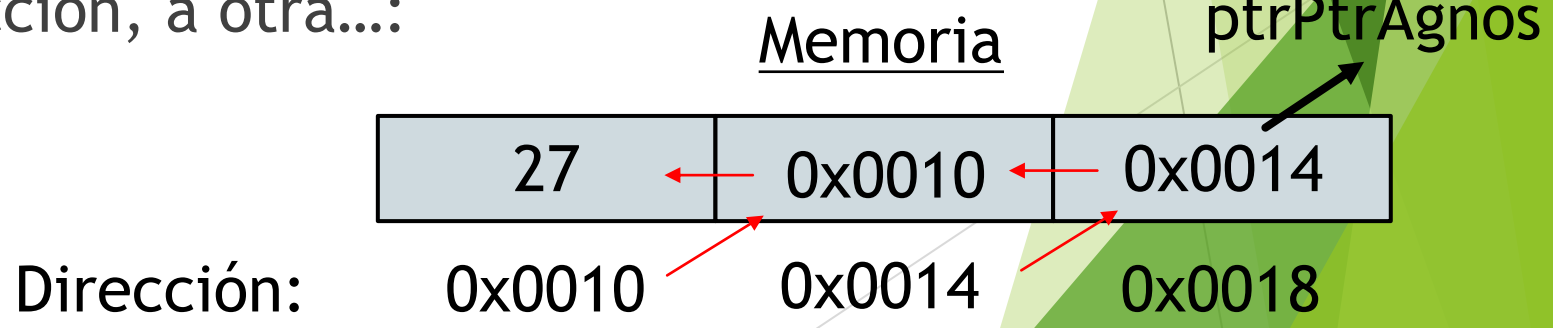
- El símbolo '*' sirve para definir un "puntero a tipo", pero **cuidado**, porque se prioriza su asociación a variables:

```
tipo* ptrA, ptrB, ptrC;
```

Aquí realmente sólo declaramos un puntero, ptrA. Pero, ptrB y ptrC son variables de tipo (no punteros a tipo).

- **Es además anidable**: Podemos definir una variable que es una dirección a otra dirección, a otra…:

```
int** ptrPtrAgnos;
```



Nota: No se recomienda anidar más de 3 '*' porque es excesivamente lioso…

Punteros (a datos y funciones) (II)

- El **puntero es también un tipo**, así que esta hipotética función sería completamente válida:

```
int* funcion(int*, int**);
```

Recibe 1 dirección de memoria en la que sabemos que hay direcciones a enteros, es decir, **puntero a punteros a enteros.**

Devuelve una dirección de memoria en la que sabemos que hay enteros: **puntero a entero.**

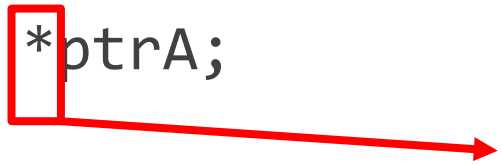
Recibe 1 dirección de memoria en la que sabemos que hay enteros, es decir, **puntero a entero.**

En efecto: `int* != int**`…

Punteros (a datos y funciones) (II)

- Sin embargo, **‘*’ sobre un puntero es** también un **operador para acceder** al dato apuntado en cuestión:

```
tipo *ptrA = Dirección_de_Variable_De_Tipo;
tipo otraVar = *ptrA;
```



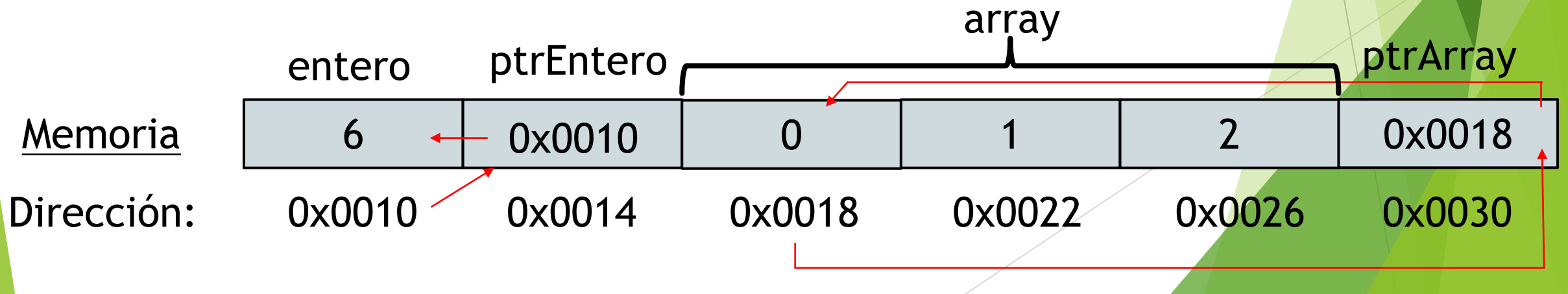
Aquí accedemos a la variable apuntada y copiamos su valor en ‘otraVar’

- Entonces, ¡con **‘*’ declaramos y seguimos punteros**!

Punteros (a datos y funciones) (II)

- Pero: hemos dicho que un puntero a un tipo de dato tiene una dirección de memoria en la que hay datos del tipo al que apunta: **¡así que puede haber uno o muchos más!**

```
int entero = 6;
int array[3] = {0, 1, 2};
int* ptrEntero = Direccion_De_Entero; //A uno
int* ptrArray = Direccion_De_Array; //A varios
```



Punteros (a datos y funciones) (II)

- Los punteros no recogen el número de elementos al que apuntan, **sólo tienen una dirección y la posibilidad de moverse en "saltos" de tantos bytes como cada elemento del tipo al que apuntan.**

- El **programador debe saber** cuánto puede desplazarse sobre un puntero por otros medios. De hecho, **cuando una función recibe un puntero que puede ser a más de uno**, se suele **pasar** un parámetro adicional con la **cantidad**:

```
void funcion(int* punteroAEntero, int cuantos);
```

Punteros (a datos y funciones) (II)

- Muy bien, *puntero a algo* es una dirección de memoria en la que hay datos de tipo a *algo*. Eso se declara con un '*' y, sobre ese nuevo tipo puntero creado, accedemos también con un '*': "*Asterisco con asterisco se va*"

```
int entero = 5;
int* ptrEntero = Direccion_De_Entero; // Creo puntero
int otroEntero = *ptrEntero; // Ahora accedo
```

- **¿Pero y si hay efectivamente más de un elemento? ¿Cómo accedemos** con un simple '*'?

Punteros (a datos y funciones) (II)

- Sabiendo que al final el puntero es un tipo de dato consistente en número natural con una dirección de memoria, **podemos operar sobre él para movernos**:

```
int array[3] = {0, 1, 2};
int *ptrArray = Direccion_De_Array;
int primero = *ptrArray;//Asterisco con ast. se va
int segundo = *(ptrArray + 1)
int tercero = *(ptrArray + 2)
```

Usamos '*' como "voy a", pero podemos usar aritmética para indicar desplazamientos antes.

Punteros (a datos y funciones) (II)

- Interesante, pero algo incómodo… ¿no? Por eso **podemos usar también el operador '[ ]'**, que nos permite trabajar igual que con un array:

```
int array[3] = {0, 1, 2};
int *ptrArray = Direccion_De_Array;
int primero = ptrArray[0];
int segundo = ptrArray[1];
int tercero = ptrArray[2];
```

```
int entero = 7;
int* ptrEntero = Direccion_De_Entero;
int otro = ptrEntero[0];
```

Asterisco con [ ] se va

Y es muy legible para punteros multi-dimensionales:

int ***ptr ➡ ptr[][][]

Y no hay ni por qué apuntar a un array para usarlo…

Punteros (a datos y funciones) (III)

- Muy bien, sabemos que un puntero es un tipo que se puede ver como un número, sabemos declararlos incluso anidados, y acceder a ellos (seguirlos)…

- ¿Pero **cómo los inicializamos**? ¿Cómo leemos lo que hasta ahora hemos indicado con "`Direccion_De_...`"?

- Pues **con el operador &** asociado a lo que apuntar:

```
int entero = 17;
int *ptrEntero = &entero;
```

El operador & da 'asterisco' (puntero)

Punteros (a datos y funciones) (III)

- Muy bien, sabemos que un puntero es un tipo que se puede ver como un número, sabemos declararlos incluso anidados, y acceder a ellos (seguirlos)…

- ¿Pero **cómo los inicializamos**? ¿Cómo leemos lo que hasta ahora hemos indicado con "`Direccion_De_...`"?

- Pues **con el operador &** asociado a lo que apuntar:

```
int array[3] = {1, 2, 3};
int *ptrArray = &array[2];
```

El operador & da 'asterisco' (puntero) ¡y **podemos apuntar donde nos convenga!**
Ah, y &array[0] es igual que &array.

Punteros (a datos y funciones) (III)

- Entonces, ¿**y si por ejemplo** queremos asignar un puntero a puntero de enteros? **Pues lo aplicamos igualmente** a lo que hay que apuntar:

```
int entero = 18;
int *ptrEntero = &entero;
int **ptrPtrEntero = &ptrEntero;
```

Esto no es inconsistente con nada dicho: **un puntero es un dato y también podemos apuntar e él.**

- Bien, ¿pero **y si no tenemos aún (o ya) donde apuntar**?

Pues **apuntamos a 0 o NULL** (definido en stdio.h entre otras)

Usar un puntero no inicializado es muy peligroso: ¡el programa puede funcionar **mal sin que nos percatemos**!

Punteros (a datos y funciones) (III)

- Parece obvio, pero los punteros son un tipo en sí, así que:
  - **Podemos crear arrays de punteros.**
  - **Podemos usar punteros como elementos de otros tipos** como estructuras.

- Vale, está claro que son como 'un tipo más'. Pero…**¿y si queremos mostrar un puntero?**
  - Además de poder ver el "número" crudo de dirección, tienen una **secuencia de formato** propia también, **'%p'**.

Punteros (a datos y funciones) (III)

- **¿Y qué tamaño tienen los punteros?** La **respuesta depende de la arquitectura:**
  - En una arquitectura de **32 bits**, para direccionar cualquier celda de la memoria necesitaremos 32 bits, i.e., **4 bytes**.
  - En una arquitectura de **64 bits**, para direccionar cualquier celda de la memoria necesitaremos **64 bits**, i.e., **8 bytes**.

En general…

- De hecho, el estándar **tampoco asegura que un puntero a un tipo A tenga el mismo tamaño que otro a otro tipo B** (aunque **suele ser así**).

Punteros (a datos y funciones) (IV)

- **¿Y para qué tanta complicación?** 

- En primer lugar, usar punteros nos ahorra copiar grandes cantidades de datos, lo que además de **ineficiente** puede directamente **no** ser **factible**:

```
int function(EstructuraMuyGrande arg);
```

Copiamos toda la estructura

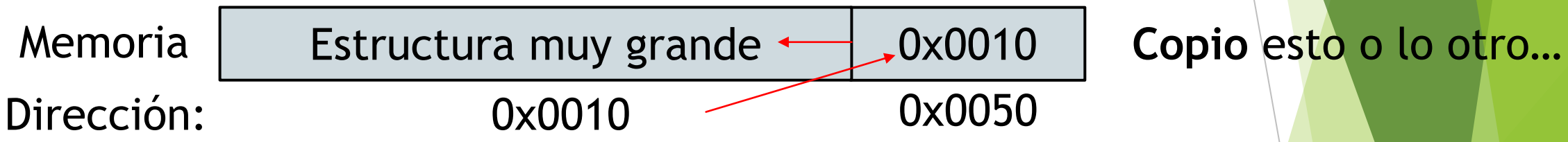
- ¿Merece la pena pasarla por valor (copiarla) para lo que se va a hacer o **nos sirve esto?**

```
int function(EstructuraMuyGrande* arg);
```

Sólo copiamos la dirección de la estructura

Punteros (a datos y funciones) (IV)

- Y no estamos rompiendo la regla especial de las funciones: sus argumentos son una copia…



- AH: Cuando usamos el **operador de acceso '.' desde un puntero, éste se convierte en '->'**. Por ejemplo:

```
int valor = ptrEstructura->campoEntero;
Int valor = (*ptrEstructura).campoEntero;
```

Equivalente

Punteros (a datos y funciones) (IV)

- Y los **arrays** directamente **ni siquiera pueden pasarse ni devolverse de funciones** sólo punteros a ellos:

~~char[]~~ `char* funcion(int param[]);`

Es exactamente igual que:

~~char[]~~ `char* funcion(int* param);`

Existe también el 'puntero a array de…', por ejemplo: **int (*ptr)[]** pero no vamos a entrar en eso.

- Pero un gran poder conlleva una gran responsabilidad: Cuando **pasamos un puntero y hacemos cambios…¡modificamos a lo que apuntamos!**

(La dirección podremos copiarla cuanto queramos, pero el destino sí es siempre el mismo)

Punteros (a datos y funciones) (IV)

- **¿Y para qué tanta complicación?** 

- En segundo lugar, y a consecuencia de la idea anterior, usar punteros no **permite simular el paso por referencia con funciones**, aunque en C todo se pase por valor.

Esta función no nos va a servir para nada:

```
void incrementa(int val){
   val = val + 1;// o val++
}
```

Punteros (a datos y funciones) (IV)

- Pero **esta sí**:

```
void incrementa(int* val){
   *val = *val + 1; // o val[0]++
}

int main(void){
   int valor = 7;
   incrementa(&valor);
}
```

Aquí ya estamos creando el “puntero a entero” que la función nos pide

Punteros (a datos y funciones) (IV)

- También podemos querer modificar un puntero "por referencia":

```
void ponerANull(int** ptr){
   *ptr = 0; // o NULL…
}
int main(){
   int* ptr;
   ponerANull(&ptr);
}
```

Como regla mnemotécnica: **Simular el paso por referencia a un tipo** implica que la **función recibe "un asterisco más"** sobre el tipo deseado.

Aquí ya estamos creando el "puntero a puntero de entero" que la función nos pide

Punteros (a datos y funciones) (IV)

- **¿Y para qué tanta complicación?** 

- En tercer y último lugar, manejando punteros podemos desarrollar **código muy flexible con poco esfuerzo**:

```
int tamA, tamB;
int* conjuntoA = leerConjuntoDeEntero(&tamA);
int* conjuntoB = leerConjuntoDeEntero(&tamB);
int *grande, *pequeno;
```

Estaríamos simulando ya el paso por referencia para escribir los tamaños pertinentes.

Recuerda que el * se asocia a la variable y no al tipo

```
//Apuntar a A y B según sean y ya…:
//Centramos el resto del código en "grande" y
//"pequeño" apunten al que apunten cada uno
```

Punteros (a datos y funciones) (V)

- Y antes de pasar a ejemplos, vamos aprender algunos **detalles interesantes** más de los punteros:
- **Puntero** a "datos" **no se limita a variables**. Como es lógico, también **podemos guardar la dirección de constantes**:

```
const tipo dato = valor;
```

Igual

```
const tipo * ptrDato = &valor;
tipo const * ptrDato = &valor;
```

Nota: Con #define no sería posible porque se reemplaza en pre-procesado

No podremos cambiar el dato al que apuntamos, **pero sí apuntar a otro sitio.**

Lógico. Leyendo de derecha a izquierda: "Puntero a constante *tipo*"

Punteros (a datos y funciones) (V)

- Y antes de pasar a ejemplos, vamos aprender algunos **detalles interesantes** más de los punteros:
- Un "puntero" no tiene por qué ser una variable. También **podemos crear un puntero constante**:

```
tipo dato = valor;
tipo * const ptrDato = &valor;
```

Podremos cambiar el dato al que apuntamos, pero **no dejar de apuntar al mismo sitio.**

Lógico. Leyendo de derecha a izquierda: "Puntero constante a *tipo*"

Punteros (a datos y funciones) (V)

- Y antes de pasar a ejemplos, vamos aprender algunos **detalles interesantes** más de los punteros:
- **Podemos combinar ambas ideas** en un puntero constante que apunta a una constante de cierto tipo:

```
const tipo dato = valor;
```

Igual

```
const tipo * const ptrDato = &valor;
tipo const * const ptrDato = &valor;
```

No Podremos cambiar ni a donde apuntamos ni el valor al que estamos apuntando

Lógico. Leyendo de derecha a izquierda: “Puntero constante a constante de *tipo*”

Punteros (a datos y funciones) (V)

- Y antes de pasar a ejemplos, vamos aprender algunos **detalles interesantes** más de los punteros:
- **Podemos** tener estos aspectos en cuenta también al definir y usar funciones:

```
1.    tipoRes func(const tipo* par);
2.    tipoRes func(tipo * const par);
3.    tipoRes func(const tipo * const par);
```

Nota: El compilador vigilará que no modifiquemos el valor apuntado.

Nota: las declaraciones 2 y 3 tienen poco sentido… El parámetro será de la función y será una copia (paso por copia), así que dará igual que lo cambiemos.

Punteros (a datos y funciones) (V)

- Y antes de pasar a ejemplos, vamos aprender algunos **detalles interesantes** más de los punteros:

- Como tipos que son, **podemos convertir un puntero** a tipo A, a un puntero a tipo B **mediante castings… Aunque cuidado**, **porque deben ser compatibles**: un puntero a tipo X lleva asociada la semántica de, por ejemplo, cuántos bytes ha de desplazarse al moverse.

```
tipoB *ptr_a_B = (tipoB*) ptr_a_A;
```

- Existe el tipo especial ***void****, que significa: "**puntero a cualquier cosa**", y que podemos usarlo **para operar en términos generales**. De hecho, no se puede **"seguir" (de-referenciar) sin convertirlo** a "puntero a algo conocido."

Punteros (a datos y funciones) (VI)

archivo23.c

```c
#include <stdio.h>

void func(int* par){
    printf("func: Recibo esta direccion: %p\n", par);
    int leo = (*par);//Leo el puntero y me copio el valor
    leo = leo + 1;//Incremento mi copia
    (*par) = leo;//Actualizo lo que leo
    //Todo esto equivale a: (*par)++
}

int main(){
    int valor = 5;
    int * ptr_Valor = &valor;
    printf("Valor: %d (estoy en: %p)\n", valor, &valor);
    func(ptr_Valor);//Voy a modificar el valor!
    printf("Valor: %d (sigo en: %p)\n", valor, ptr_Valor);
    printf("Y yo, puntero, tambien tengo direccion: %p\n", &ptr_Valor);
    printf("En este PC x64 un puntero a entero ocupa: %lu bytes\n",
        sizeof(int*));
    printf("\t y uno a double: %lu bytes, y uno a puntero entero: %lu bytes...\n",
        sizeof(double), sizeof(int**));
    return 0;
}
```

```
Valor: 5 (estoy en: 0x7ffee8914b78)
func: Recibo esta direccion: 0x7ffee8914b78
Valor: 6 (sigo en: 0x7ffee8914b78)
Y yo, puntero, tambien tengo direccion: 0x7ffee8914b70
En este PC x64 un puntero a entero ocupa: 8 bytes
        y uno a double: 8 bytes, y uno a puntero entero: 8 bytes...
```

Punteros (a datos y funciones) (VI)

archivo24.c

```c
#include <stdio.h>

typedef struct{
    int campoA;
    int campoB;
} Estructura;

union LaUnion{
    int campoA;
    int campoB;
};

void funcionEst(Estructura* ptr){
    int valCampoA = ptr->campoA;
    int valCampoB = (*ptr).campoB;
    printf("Estructura: (%d, %d)\n", valCampoA, valCampoB);
    ptr->campoA+=10;//Modificamos el origen
}
```

```c
void funcionUnion(union LaUnion* ptr){
    int valor = ptr->campoA;//Es una union... solo hay 1 cosa
    printf("Union: %d\n", valor);
    ptr->campoA+=10;
}

int main(void){
    Estructura miEstructura = {3, 4};
    funcionEst(&miEstructura);
    union LaUnion miUnion;
    miUnion.campoB = 8;
    funcionUnion(&miUnion);
    printf("Queda: Estructura: (%d, %d)\n", miEstructura.campoA,
        miEstructura.campoB);
    printf("\tUnion: %d\n", miUnion.campoA);
    return 0;
}
```

```
Estructura: (3, 4)
Union: 8
Queda: Estructura: (13, 4)
        Union: 18
```

Punteros (a datos y funciones) (VI)

archivo25.c

```c
#include <stdio.h>

int main(void){
    int arrayA[] = {1, 2, 3};
    int* ptr_ini = &(arrayA[0]);//o &array al apuntar al comienzo
    int* ptr_fin = &(arrayA[2]);
    printf("Primer y último valor de A: %d y %d\n", *ptr_ini, *ptr_fin);
    int arrayB [] = {4, 5, 6};
    int* matriz[2];//Array de punteros a entero
    matriz[0] = arrayA;
    matriz[1] = arrayB;
    //Vamos a mostrar los 6 valores de esta matriz:
    printf("Primera fila: [%d %d %d]\n", matriz[0][0], matriz[0][1], matriz[0][2]);
    printf("Segunda fila: [%d %d %d]\n", matriz[1][0], matriz[1][1], matriz[1][2]);
    //Vamos a apuntar a la misma matriz de otra forma:
    int** misma = matriz;// o &(matriz[0])
    printf("Primera fila: [%d %d %d]\n", misma[0][0], misma[0][1], misma[0][2]);
    printf("Segunda fila: [%d %d %d]\n", misma[1][0], misma[1][1], misma[1][2]);
    return 0;
}
```

```
Primer y último valor de A: 1 y 3
Primera fila: [1 2 3]
Segunda fila: [4 5 6]
Primera fila: [1 2 3]
Segunda fila: [4 5 6]
```

Nota: Esta matriz artesanal no tiene por qué estar contigua en memoria

Punteros (a datos y funciones) (VI)

archivo26.c

```c
#include <stdio.h>

const int constante = 9;

int main(){
    const int* ptrAConst = &constante;
    int array3D[2][2][2] = {{{1, 2}, {3, 4}}, {{5, 6}, {7, 8}}};
    printf("Primera rodaja:\n");
    printf("%d %d\n", array3D[0][0][0], array3D[0][0][1]);
    printf("%d %d\n", array3D[0][1][0], array3D[0][1][1]);
    printf("Segunda rodaja:\n");
    printf("%d %d\n", array3D[1][0][0], array3D[1][0][1]);
    printf("%d %d\n", array3D[1][1][0], array3D[1][1][1]);
    printf("O en crudo...:\n");
    int* ptr = &(array3D[0][0][0]);//Vamos a ver que esta todo contiguo:
    printf("%d, %d, %d, %d, ", *ptr, *(ptr+1), *(ptr+2), *(ptr+3));
    printf("%d, %d, %d, %d\n", ptr[4], ptr[5], ptr[6], ptr[7]);
    int (*otro)[2][2] = array3D;// ESTO ES UN PUNTERO A ARRAY
    printf("%d\n", otro[0][1][1]);
    printf("%d\n", (*(otro + 1))[1][1]);
    return 0;
}
```

```
Primera rodaja:
1 2
3 4
Segunda rodaja:
5 6
7 8
O en crudo...:
1, 2, 3, 4, 5, 6, 7, 8
4
8
```

Se pivota en torno a la primera dimensión. El resto deben conocerse para saber "cuánto moverse"

Punteros (a datos y funciones) (VII)

- El código de los programas y sus datos se cargan en memoria… y **¡C nos permite leer y usar directamente (apuntar) esas direcciones!**

- Para apuntar tenemos los punteros, que son un tipo de dato especial que se puede asociar a:
  - Otro tipo de dato (**Puntero a datos**)
  - Una función (**Puntero a función**)

- **Un puntero es un valor**, como el que puede haber en una variable *int* o *float*, pero **cuyo significado es una dirección de memoria, y** con el que **se puede operar**.

Punteros (a datos y funciones) (VII)

- Un **puntero a función** es un tipo de dato que **almacena la dirección de** memoria en la que se encuentra **una función** (en lugar de a datos como anteriormente).

- **Teniendo un puntero a función podemos invocarla** como si la llamada estuviera fijada en el código estático… Pero hay una gran diferencia: **podremos cambiar a dónde apuntamos dentro del propio código**.

- Es una **herramienta muy potente para hacer Ingeniería de Software**, especialmente patrones de diseño como el *Strategy*.

Punteros (a datos y funciones) (VIII)

- A efectos prácticos hay mucha menos complejidad que abarcar y nos basta con saber que:
  - Un puntero a función es **distinto a un puntero a datos**.
  - Es un tipo de datos que **se diferencia según la signatura de función a la que se apunta**, es decir, sus parámetros de entrada y su tipo de resultado. En otras palabras: al igual que int* es distinto a int**, **un puntero a función será distinto a otro si tienen signaturas distintas**.
  - Podemos **pasarlos** y **devolverlos de funciones**, **agruparlos en arrays** y, en general, usarlos como parte de otros tipos como estructuras.
  - **No podemos pensar en aritmética** de punteros cuando pensamos en punteros a funciones: no vamos a apuntar a un trozo en mitad de una función, o saltar de una a otra…

Punteros (a datos y funciones) (IX)

- Para declarar un puntero a función debemos usar esta sintaxis:

```
tipoRes (*nomPtr)(tipoP1, tipoP2…);
```

(Y *void* es una opción válida en los tipos)

- Esto **se lee como**: "*Mi nombre es nomPtr y apunto a una función que espera 2 parámetros: uno tipoP1, otro tipoP2; y que devuelve un valor tipoRes*".

- Se asignan directamente con el nombre de la función:

```
tipRes (*nomPtr)(tipoP1, tipoP2…) = nombre_Func;
```

Nota: Podemos poner '**&**' delante como "pidiendo la dirección", pero es **innecesario**.

Punteros (a datos y funciones) (X)

- Si queremos que una **función reciba un puntero a función como parámetro**:

Nombre opcional en declaración (como cualquier otro parámetro)

```
tipoRes funcion(tipoRes (*nombrePar)(tipoP1, tipoP2..),…);
```

Argumento esperado

Punteros (a datos y funciones) (X)

- Si queremos que una **función devuelva un puntero a función como resultado**:

```
tipRes (*nomFunc(tipoP1)) (tipoP1, tipoP2);
```

- Así creamos una función llamada *nomFunc* que espera:

Punteros (a datos y funciones) (X)

- Si queremos que una **función devuelva un puntero a función como resultado**:

```
tipRes (*nomFunc(tipoP1)) (tipoP1, tipoP2);
```

- Así creamos una función llamada *nomFunc* que espera:
  - Recibir un único parámetro de tipo tipoP1

Punteros (a datos y funciones) (X)

- Si queremos que una **función devuelva un puntero a función como resultado**:

```
tipRes (*nomFunc(tipoP1)) (tipoP1, tipoP2);
```

- Así creamos una función llamada *nomFunc* que espera:
  - Devolver el puntero a una función: tipRes (*)(tipoP1, tipoP2)

Punteros (a datos y funciones) (X)

- Y sí, **es una sintaxis muy liosa…** Pero **con ayuda de *typedef*** para dar un alias al tipo de puntero a función devuelto queda **mucho más legible**:

```
typedef tipoRes (*FuncPTR)(tipoPar1, tipoPar2);

FuncPTR nomFunc(tipoP1);
```

Con el alias creado al tipo de puntero a función que se querría devolver podemos escribirlo donde iría uno de los tipos ya vistos.

Punteros (a datos y funciones) (XII)

- **Para usar un puntero a función lo ponemos directamente** como si del nombre de la función se tratara:

```
tipRes (*nomPtr)(tipoP1, tipoP2…) = nombre_Func;
tipoRes resultado = nomPtr(par1, par2…);
```

Nota: Podemos poner '*' delante como "accediendo a la dirección": (*nomPtr) pero es **innecesario**.

Punteros (a datos y funciones) (XIII)

archivo27.c

```c
#include <stdio.h>

void mostrarValores(int a, int b){
    printf("Muestro: %d y %d\n", a, b);
}

typedef void (*MyFunc2INT) (int, int);

int main(){
    int a = 6, b = 9;
    void (*ptrMostrador)(int, int) = mostrarValores;
    MyFunc2INT otroPtrMostrador = &mostrarValores;
    ptrMostrador(a, b);
    (*ptrMostrador)(a, b);
    otroPtrMostrador(a, b);
    (*otroPtrMostrador)(a, b);
    return 0;
}
```

```
Muestro: 6 y 9
Muestro: 6 y 9
Muestro: 6 y 9
Muestro: 6 y 9
```

Equivalente

Equivalente

Punteros (a datos y funciones) (XIV)

archivo28.c

```c
#include <stdio.h>

void mostrarValores(int a, int b){
    printf("Muestro: %d y %d\n", a, b);
}

void mostrarSuma(int a, int b){
    printf("Sumo: %d + %d = %d\n", a, b, a+b);
}

void mostrarResta(int a, int b){
    printf("Resto: %d - %d = %d\n", a, b, a-b);
}

typedef void (*MyFunc2INT) (int, int);

void Aplicar(void (*)(int, int), int, int);//Declaramos

void (*dameLaResta(void))(int, int){
    return mostrarResta;
}
```

```c
int main(){
    int a = 6, b = 9;
    void (*funcSet[3])(int, int);//MyFunc2INT funcSet[3];
    funcSet[0] = mostrarValores;
    funcSet[1] = mostrarSuma;
    funcSet[2] = dameLaResta();//Llamamos
    Aplicar(funcSet[0], a, b);//Mostramos
    (funcSet[1])(a, b);//Sumamos
    (dameLaResta())(a, b);//No usamos funcSet, re-pedimos direccion
    return 0;
}

void Aplicar(void (*func)(int, int), int a, int b){
    func(a, b);
}
```

```
Muestro: 6 y 9
Sumo: 6 + 9 = 15
Resto: 6 - 9 = -3
```

Contenidos

- ¿Qué es C?
- Entorno de trabajo
- ¡Hola Mundo! (Y Compilación)
- Variables y constantes. Ejemplos
- Funciones. Ejemplos
- Punteros (a datos y funciones). Ejemplos
- **Memoria y cómo pedirla (y liberarla!)**
- Estructuras de control. Ejemplos
- Cómo depurar un programa
- Recibiendo parámetros por consola
- Creando valores aleatorios
- Midiendo tiempos
- Lectura y escritura de archivos de texto
- Aplicación: Creando una librería para operar con matrices

Memoria y cómo pedirla (y liberarla) (I)

- Si alguien ha oído **hablar sobre C**, puede que haya escuchado que **había que gestionar la memoria a mano** y que no había un "recolector de basura" como en Java…
- Pero **hasta ahora, no nos hemos preocupado de gestionar memoria**, como mucho de apuntar a zonas: se ha obtenido y liberado automáticamente.
- Nos hemos conformado con crear variables globales o limitadas a funciones… pero **al programar en C podemos pensar en 3 tipos de memoria** (dentro de la principal) con la que lidiar:
  - Estática
  - De pila (*stack*)
  - De montículo (*heap*)

Memoria y cómo pedirla (y liberarla) (II)

- La **Memoria Estática**:
  - Es donde se almacenan las **variables globales y las estáticas** asociadas a funciones. Se **mantiene toda la ejecución** programa.
- La **Memoria de Pila (*Stack*)**:
  - Es una zona de la memoria "*multiuso*" que se reserva para la ejecución del programa y a la que se asigna un **tamaño limitado** pero a la que se puede **acceder rápido**.
  - Se usa para **almacenar las variables locales de las funciones** cuando se llaman ("*push*" en una pila Last In - First Out) que **se eliminan cuando se sale** de ellas, de ahí su corto tiempo de vida. Nunca se debe devolver una dirección (puntero) a este tipo de memoria
  - Se **gestiona automáticamente**, por eso no nos hemos tenido que preocupar hasta ahora.
  - Cada llamada a una función (incluso a la misma) tiene una **visión local** de sus datos… por eso hay que tener cuidado con la recursividad y con la creación de grandes arrays: **Podemos desbordarla pila.**

Memoria y cómo pedirla (y liberarla) (III)

- La **Memoria de Montículo (*Heap*)**:
  - La memoria *Heap* es, a efectos prácticos, "**opuesta**" a la *Stack*:
    - **Apunta al grueso de la memoria** en lugar de a una pequeña región pre-fijada.
    - **La gestión** (pedir y liberar) **ha de hacerse** explícitamente **por el programador**. Así que si se pide y usa memoria *Heap* dentro de una función, **ese espacio seguirá así aunque se salga de la función (y hasta que se libere)**. Es una de las claves para, por ejemplo, poder devolver arrays de funciones y evitar grandes copias a *Stack*.
    - Su uso es algo más lento que el de la memoria Stack y **nos requiere trabajar con punteros**.
  - **Vamos a ver entonces cómo podemos trabajar con este tipo de memoria** que habíamos obviado hasta ahora.

Memoria y cómo pedirla (y liberarla) (IV)

- Las funciones de **gestión de memoria *Heap*** se encuentran declaradas en la librería estándar de C (<**stdlib.h**>).
- Para **pedir un bloque de memoria contiguo en *Heap***:

```
void* malloc(size_t tam);
```

¿Cuántos bytes en total? Ej., *sizeof(int)*10* pide espacio para 10 enteros consecutivos

Nos da el bloque como esté, **sin inicializar**.

Devuelven un "*puntero a cualquier cosa*". Nosotros sabremos si en el bloque guardaremos enteros (int*), doubles (double*)…

Pueden fracasar. Se debe comprobar que el puntero no sea 0!

¿Cuántos elementos y de cuántos bytes cada uno? Ej.,(10, sizeof(int))

```
void* calloc(size_t num, size_t tam);
```

Nos da el bloque ya **inicializado a 0**.

Memoria y cómo pedirla (y liberarla) (IV)

- Para **liberar un bloque pedido en *Heap***:

Puntero al inicio de un bloque de memoria pedido para cualquier tipo.

```
void free(void* puntero_A_Inicio_Bloque);
```

- **Una vez liberado el bloque ya no deberemos volver a acceder a él hasta pedir otro equivalente**. Por eso se suele apuntar a 0 (nulo) un puntero tras liberar el bloque de memoria al que apuntaba.

- Nunca debemos intentar liberar un bloque que no hemos pedido, principalmente el de ninguna variable en *Stack*: **Asocia siempre "*malloc*" y "*free*"**.

- Siempre debemos liberar un bloque de memoria Heap que no vayamos a usar más.

Memoria y cómo pedirla (y liberarla) (IV)

- Para **redimensionar un bloque *en Heap***:

```
void* realloc(void* punteroIni, size_t nuevoTam);
```

- *Realloc* puede o no mover el bloque (p.ej., según quepa), pero **los contenidos que había se preservarán** incluso moviendo, aunque lo nuevo estará sin inicializar.
- Devuelve el puntero al nuevo bloque, que variará o no según si hubo movimiento, o 0 (nulo), si ha fracasado.

Nota: Si damos un puntero nulo, funcionará como un *malloc*. Si damos un puntero válido y un tamaño de 0, funciona como *free*.

Memoria y cómo pedirla (y liberarla) (V)

archivo29.c

```c
#include <stdlib.h>
#include <stdio.h>

int* dameUnArray(int cuantos){
    //int array[cuantos] = {0};
    //Eso no lo podria devolver porque se perdera
    //el control al salir de la funcion
    return malloc(sizeof(int)*cuantos);//Se pasa a int* solo
}

int* dameUnArrayACero(int cuantos){
    //int array[cuantos] = {0};
    //Eso no lo podria devolver porque se perdera
    //el control al salir de la funcion
    return calloc(cuantos, sizeof(int));//Se pasa a int* solo
}
```

```c
int main(){
    int* miArrayRND = dameUnArray(2);
    int* miArrayINI = dameUnArrayACero(2);
    //Nos la jugamos y no comprobamos con un if(...)
    miArrayRND[0] = 8;
    miArrayRND[1] = 9;
    printf("ArrayRND: %d, %d\n", *miArrayRND, *(miArrayRND+1));
    printf("ArrayINI: %d, %d\n", *miArrayINI, *(miArrayINI+1));
    free(miArrayRND);
    free(miArrayINI);
    return 0;
}
```

```
ArrayRND: 8, 9
ArrayINI: 0, 0
```

Memoria y cómo pedirla (y liberarla) (VI)

archivo30.c

```c
#include <stdlib.h>
#include <stdio.h>

int main(){
    int* array = malloc(sizeof(int)*2);
    array[0] = 8;
    array[1] = 8;
    array = realloc(array, sizeof(int)*4);
    printf("[%d, %d, %d, %d]\n", array[0], array[1], array[2], array[3]);
    free(array);
    //Vamos a hacernos una matriz 2x3
    int** matriz = malloc(sizeof(int*)*2);//Vamos a crear un bloque para
                                          //guardar el apuntador a cada fila
    matriz[0] = malloc(sizeof(int)*3);//Reservamos la primera fila guardando su int*
    matriz[1] = malloc(sizeof(int)*3);//Reservamos la primera fila guardando su int*
    matriz[0][0] = 1;//Accedemos primero a la fila [0] y ya vamos a la columna [0]: Izq a Der
    matriz[0][1] = 2;
    matriz[0][2] = 3;
    matriz[1][0] = 4;
    matriz[1][1] = 5;
    matriz[1][2] = 6;
    printf("[%d, %d, %d;\n", matriz[0][0], matriz[0][1], matriz[0][2]);
    printf("%d, %d, %d]\n", matriz[1][0], matriz[1][1], matriz[1][2]);
    //Ahora liberamos, primero cada fila para no "olvidar" su direccion
    free(matriz[0]);
    free(matriz[1]);
    free(matriz);//Finalmente liberamos el bloque que tenia el apuntador a cada inicio de fila
    return 0;// Y NO, ESTA MATRIZ NO ESTA CONTIGUA EN MEMORIA... VEREMOS OTROS METODOS MEJORES
}
```

```
[8, 8, 0, -1610612736]
[1, 2, 3;
4, 5, 6]
```

No inicializado

No podemos saber si esta matriz estará contigua…

Contenidos

- ¿Qué es C?
- Entorno de trabajo
- ¡Hola Mundo! (Y Compilación)
- Variables y constantes. Ejemplos
- Funciones. Ejemplos
- Punteros (a datos y funciones). Ejemplos
- Memoria y cómo pedirla (y liberarla!)
- **Estructuras de control. Ejemplos**
- Cómo depurar un programa
- Recibiendo parámetros por consola
- Creando valores aleatorios
- Midiendo tiempos
- Lectura y escritura de archivos de texto
- Aplicación: Creando una librería para operar con matrices

Estructuras de control. Ejemplos (I)

- **En C**, como en cualquier lenguaje de programación, **hay formas de controlar el flujo de ejecución del programa.**
- Podemos:
  - Seguir una vía u otra según una condición lógica (*if-else*)
  - Ejecutar unas órdenes u otras según un valor (*switch*)
  - Repetir una serie de instrucciones mientras se cumple una condición (*while* y *do-while*)
  - Repetir una serie de instrucciones un número dado de veces (*for*)
  - Saltar a una región de código *(*go-to*)*
- Las **estructuras son combinables** consigo mismas y entre ellas
- La **sintaxis es virtualmente igual a la de Java** (meno go-to, que no se soporta).

Estructuras de control. Ejemplos (II)

- Podemos seguir una vía u otra según una condición (*if-else*):

archivo31.c

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
    int* vec = malloc(sizeof(int)*2);
    if(vec){
        vec[0] = 0;
        vec[1] = 1;
        if(!vec[0]){
            printf("VEC[0] es 0\n");
        }else if(vec[1]){
            printf("VEC[0] es <> 0\n");
        }else{
            printf("Y aqui si que no entramos nunca\n");
        }
    }else{
        printf("No se ha podido reservar memoria\n");
    }
    free(vec);
    return 0;
}
```

```
VEC[0] es 0
```

Anidamos un *if* dentro de *otro*

Contamos con "*else if*" para condicionar también la entrada a las alternativas

Aquí sí comprobamos que la memoria pedida no apunte a 0, como debe hacerse

Estructuras de control. Ejemplos (III)

- Podemos ejecutar unas órdenes u otras según un valor (*switch*):

archivo32.c

```c
#include <stdio.h>

int main(void){
    int val = 7;
    switch(val){
        case 0:
            printf("Entro en 0\n");
            break;
        case 1:
            printf("Entro en 1\n");
            break;
        case 2:
            printf("Entro en 2\n");
            break;
        default:
            printf("Aquí se entra siempre si no hay coincidencia\n");
    }
    return 0;
}
```

```
MacBook-Pro-de-Nicolas:Codigo nccruz$ ./casa
Aquí se entra siempre si no hay coincidencia
```

Cuando se cumple una condición ya no se compara más y se ejecuta el código de todas las que quedan… **a no ser que forcemos la salida** de la estructura con ***break***.

Estructuras de control. Ejemplos (IV)

- Podemos repetir una serie de instrucciones mientras se cumple una condición (*while* y *do-while*):

archivo33.c

```c
#include <stdio.h>

int main(){
    int val = 0;
    while(val<2){
        printf("Hola\n");
        val++;//Modifico o no saldre nunca...
        if(val==3){//Esto sobra, pero nos enseña
            break;//que con un break podriamos salir
            //tanto de un while como de un do-while
            //obviando la condicion
        }
    }
    do{
        //continue;//Si no lo quitamos pasariamos
        //directamente a evaluar y no hariamos nada
        printf("Esto solo lo repito una vez\n");
        val++;//aunque la condicion no se cumpla
        //    Como garantiza un do-while
    }while(val<2);
    return 0;
}
```

```
Hola
Hola
Esto solo lo repito una vez
```

Nota: Si estamos en un bucle anidado, el ***break*** sólo afecta al bucle más próximo al que se inscribe. Relacionado con *break* está **continue**, que acaba la iteración actual donde esté y pasa a re-evaluar la condición para ver si seguir (y también afecta a donde se inscribe sólo).

Estructuras de control. Ejemplos (V)

- Podemos repetir una serie de instrucciones un número dado de veces (*for*):

archivo34.c

```c
#include <stdio.h>

int main(){
    for(int i = 0; i<3; i++){
        printf("El valor de i es: %d\n", i);
    }
    //Y no se suele hacer... pero podemos ser muy creativos:
    for(int i = 0, j = 1; i<6 && j<7; i=i+2, j=j+2){
        printf("I vale: %d y J vale: %d\n", i, j);
    }
    return 0;
}
```

```
El valor de i es: 0
El valor de i es: 1
El valor de i es: 2
I vale: 0 y J vale: 1
I vale: 2 y J vale: 3
I vale: 4 y J vale: 5
```

Nota: *break* y *continue* se usarían como antes

Estructuras de control. Ejemplos (V)

- Podemos saltar a una región de código (go-to):

Imagen: https://3d.xkcd.com/292/

La instrucción *goto* tiene muy mala fama dentro de la programación estructurada por dificultar la lectura, trazabilidad y mantenimiento del código… Históricamente se usaba para hacer "a mano" labores de bucles y condiciones (influencia de programar en ensamblador…).

Estructuras de control. Ejemplos (V)

- Podemos saltar a una región de código *(go-to)*:

Pero aún tiene un uso “noble”:

```
Salimos de un bucle doble de golpe!
```

archivo35.c

```c
#include <stdio.h>

int main(){
    int matriz[2][2] = {{1, 2}, {3, 4}};
    for(int i = 0; i<2; i++){
        for(int j = 0; j<2; j++){
            goto super_salida;
            printf("%d ", matriz[i][j]);
        }
        printf("\n");
    }
    super_salida: printf("Salimos de un bucle doble de golpe!\n");
    return 0;
}
```

Nota: Los estamentos etiquetados se ejecutan cuando se llega a ellos normalmente… Así que, aún quitando el ***goto***, el *print* final se haría tras los bucles.

Contenidos

- ¿Qué es C?
- Entorno de trabajo
- ¡Hola Mundo! (Y Compilación)
- Variables y constantes. Ejemplos
- Funciones. Ejemplos
- Punteros (a datos y funciones). Ejemplos
- Memoria y cómo pedirla (y liberarla!)
- Estructuras de control. Ejemplos
- **Cómo depurar un programa**
- Recibiendo parámetros por consola
- Creando valores aleatorios
- Midiendo tiempos
- Lectura y escritura de archivos de texto
- Aplicación: Creando una librería para operar con matrices

Cómo depurar un programa (I)

- Cualquiera que haya hecho un programa habrá tenido que **enfrentarse** en algún momento **a la tarea de identificar y corregir errores**.
- Ya no sólo es el código propio, sino que se ejecutará en un entorno dinámico con un sistema operativo, asociado a librerías, y con otros programas en ejecución. Además, si usamos paralelismo, es imposible probar todas las situaciones de concurrencia que se podrían dar.
- De hecho, **se tarda años en dar un programa por correcto** con cierta seguridad… (en parte por eso en sectores como la Aviación y la Banca se siguen manteniendo sistemas con décadas de uso a sus espaldas).
- En este enlace por ejemplo se cuenta cómo se identificó un fallo *(bug)* en una popular implementación de una búsqueda binaria que llegó incluso al JDK: https://ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html
- Nosotros mismos vemos que todos nuestros sistemas y programas, muchos comerciales (pagados) piden actualizarse frecuentemente…

Cómo depurar un programa (II)

- Filosofía aparte, **la forma más simple** de ver que un programa se comporta como queremos es haciéndolo **mostrar** resultados **parciales** y comparándolos con lo que esperamos.

- Esto se **complementa** bien con las opciones de **compilación condicionada** que vimos al principio. Pensemos en algo así:

```
#ifdef DEBUG
printf("Resultado = %d en el ciclo %d\n", res, i);
#endif
```

- **Viene además muy bien hacer pequeñas pruebas de concepto con ejemplos autónomos** para ver si algo que creemos que se puede hacer no funciona como pensamos.

Cómo depurar un programa (III)

- Más allá de la estrategia más sencilla, todos sabemos la existencia de herramientas de depuración o "**depuradores**".

- Uno de los **depuradores más conocidos para C es GDB, que es el que vamos a usar**, relacionado con GCC. Los IDE's le añaden una interfaz gráfica, pero **se puede usar en consola** como haremos nosotros.

- Lo primero que hay que saber para **usarlo es que debemos compilar añadiendo el flag -g**, que hace que GCC guarde información del código de partida en el ejecutable:

  `gcc -o nomreProg -g codigo.c` (o equivalente…)

Nota: No mezclarse con opciones de compilación optimizada: -O1, -O2…

Cómo depurar un programa (IV)

- Podemos ver cómo los ejecutables de este programa cambian si se compila sin y con soporte a depuración:

archivo36.c

```c
#include <stdio.h>

int sumaDos(int base){
    int resultado = base + 2;
    return resultado;
}

int main(){
    int valor = 0;
    for(int i = 0; i<3; i++){
        valor = sumaDos(valor);
    }
    printf("Valor: %d\n", valor);
    return 0;
}
```

```
MacBook-Pro-de-Nicolas:Codigo nccruz$ gcc -o casa archivo36.c
MacBook-Pro-de-Nicolas:Codigo nccruz$ ./casa
Valor: 6
MacBook-Pro-de-Nicolas:Codigo nccruz$ ls -l ./casa
-rwxr-xr-x  1 nccruz  staff  8464 12 nov 10:33 ./casa
MacBook-Pro-de-Nicolas:Codigo nccruz$ gcc -o casa archivo36.c -g
MacBook-Pro-de-Nicolas:Codigo nccruz$ ./casa
Valor: 6
MacBook-Pro-de-Nicolas:Codigo nccruz$ ls -l ./casa
-rwxr-xr-x  1 nccruz  staff  8800 12 nov 10:34 ./casa
```

Funcionan igual… pero el que se compila con -g es más grande: porque tiene más información

Cómo depurar un programa (V)

- Bueno, ya que lo tenemos preparado, **vamos a hacer un recorrido con el depurado del archivo36.c** y así vemos las opciones básicas de GDB:

Cargamos el programa desde GDB

```
nicolas@victoriapc:~/Escritorio/DEBUGING$ gdb ./casa
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.3) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
Para las instrucciones de informe de errores, vea:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Leyendo símbolos desde ./casa...hecho.
(gdb)
```

Cómo depurar un programa (V)

- Podemos consultar las secciones de ayuda:

```
(gdb) help
List of classes of commands:

aliases -- Alias de otras órdenes
breakpoints -- Para el programa en ciertos puntos
data -- Examinando datos
files -- Especificando y examinando archivos
internals -- Órdenes de mantenimiento
obscure -- Ocultar características
running -- Corriendo el programa
stack -- Examinar la pila
status -- Preguntas de estado
support -- Facilidades de soporte
tracepoints -- Tracing of program execution without stopping the program
user-defined -- Órdenes definidas por el usuario

Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.
```

Cómo depurar un programa (V)

- Podemos consultar las secciones de ayuda:

```
(gdb) help breakpoints
Para el programa en ciertos puntos.

List of commands:

awatch -- Establece un punto de interrupción para una expresión
break -- Set breakpoint at specified line or function
break-range -- Establecea un punto de interrupción para un rango de direcciones
catch -- Establecer puntos de captura a eventos de captura
catch assert -- Aserciones Ada no capturadas
catch catch -- Detecta una excepcion
catch exception -- Captura excepciones Ada
catch exec -- Captura llamadas a exec
catch fork -- Captura llamadas a fork
catch load -- Carga un montón de bibliotecas compartidas
catch rethrow -- Capturar una excepción cuando se relance
```

En efecto, hay muchas opciones… y nos vamos a quedar con un uso básico

Cómo depurar un programa (V)

- Sin puntos de parada se nos ejecuta bien directamente:

```
(gdb) run
Starting program: /home/nicolas/Escritorio/DEBUGING/casa
Valor: 6
[Inferior 1 (process 2914) exited normally]
(gdb) q
nicolas@victoriapc:~/Escritorio/DEBUGING$
```

- Vamos a ver el código:

```
(gdb) l
1       #include <stdio.h>
2
3       int sumaDos(int base){
4               int resultado = base + 2;
5               return resultado;
6       }
```

```
(gdb) l 8, 10
8       int main(){
9               int valor = 0;
10              for(int i = 0; i<3; i++){
```

Nota: help o h, run o r, list o l, q o quit…

Cómo depurar un programa (V)

- Vamos a volver a cargar y poner ya un punto de parada y a avanzar línea a línea (n de *next*) mostrando algunas variables (p de *print*):

```
Leyendo símbolos desde ./casa...hecho.
(gdb) b 9
Punto de interrupción 1 at 0x804843a: file archivo36.c, line 9.
(gdb) run
Starting program: /home/nicolas/Escritorio/DEBUGING/casa

Breakpoint 1, main () at archivo36.c:9
9                   int valor = 0;
(gdb) n
10                  for(int i = 0; i<3; i++){
(gdb) print valor
$1 = 0
(gdb) p valor
$2 = 0
```

Nota: Podemos poner breakpoints directamente con el nombre de la función: ***b main*** por ejemplo

```
(gdb) p array
$1 = {1, 2, 3}
(gdb) p array[1]
$2 = 2
```

Y si tuviéramos un array…

Cómo depurar un programa (V)

- Podemos asociar condiciones de activación a los puntos de parada:

```
(gdb) b 9
Punto de interrupción 1 at 0x804843a: file archivo36.c, line 9.
(gdb) condition 1 valor==7
(gdb) r
Starting program: /home/nicolas/Escritorio/DEBUGING/casa
Valor: 6
[Inferior 1 (process 3149) exited normally]
```

Entonces no nos paramos aquí porque el breakpoint 1 ya sólo vale cuando valor=7

- También podemos borrar puntos de parada sabiendo su número:

```
(gdb) delete 1
```

Cómo depurar un programa (V)

- Podemos modificar valores de variables:

```
Breakpoint 1, main () at archivo36.c:9
9               int valor = 0;
(gdb) n
10              for(int i = 0; i<3; i++){
(gdb) n
11                      valor = sumaDos(valor);
(gdb) n
10              for(int i = 0; i<3; i++){
(gdb) p valor
$1 = 2
(gdb) set valor=7
(gdb) p valor
$2 = 7
(gdb) c
Continuando.
Valor: 11
[Inferior 1 (process 3133) exited normally]
```

Y podemos dejar al programa continuar hasta un hipotético siguiente punto de parada… con c (de *continue*)

Cómo depurar un programa (V)

- Podemos entrar en funciones (y dejarlas terminar cuando no queramos seguir viéndolas) con s (de *step*) y *finish*:

```
(gdb) n
10              for(int i = 0; i<3; i++){
(gdb) n
11                      valor = sumaDos(valor);
(gdb) step
sumaDos (base=0) at archivo36.c:4
4               int resultado = base + 2;
(gdb) finish
Correr hasta la salida desde #0  sumaDos (base=0) at archivo36.c:4
0x08048458 in main () at archivo36.c:11
11                      valor = sumaDos(valor);
Value returned is $1 = 2
```

Nota: Sí, cuando nuestro programa ya parece funcionar, lo normal es compilarlo pidiéndo al compilador que intente acelerarlo con -O1 ó -O2… (más es peligroso)

Cómo depurar un programa (VI)

- Imaginemos ahora que **todo parece funcionar bien**… aún así podemos usar **otras herramientas complementarias** que analizan la ejecución de nuestro programa.

- Yo le suelo **dar mucho uso a Valgrind** (**sistemas Linux**), una suite libre que agrupa distintos módulos de análisis de rendimiento y de **uso de memoria** (*heap* principalmente).

- Podemos instalarlo en nuestro Linux (Ubuntu y compatibles) ejecutando:

```
apt-get install valgrind
```

Cómo depurar un programa (VI)

archivo37.c

```c
#include <stdio.h>
#include <malloc.h>

int main(){
    int val;
    printf("Valor sin inicializar: %d\n", val);

    int* vec = malloc(sizeof(int)*2);//VEC[0, 1]
    printf("Valor fuera de rango: %d\n", vec[2]);//Leo sin deber
    vec[2] = 0;//Escribo sin deber... esto puede cerrar todo
    //free(vec);//Y no libero esta memoria... tengo una fuga
    return 0;
}
```

Si no… veremos igualmente que no hay cosas bien, pero el informe será más difícil de interpretar

Todo parece bien obviando el número raro…

```
nicolas@victoriapc:~/Escritorio/DEBUGING$ gcc -o casa -g archivo37.c
nicolas@victoriapc:~/Escritorio/DEBUGING$ ./casa
Valor sin inicializar: 134513851
Valor fuera de rango: 0
```

Cómo depurar un programa (VI)

Pero confirmamos **que no estaba bien…** 😱

```
nicolas@victoriapc:~/Escritorio/DEBUGING$ valgrind ./casa
==3443== Memcheck, a memory error detector
==3443== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==3443== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==3443== Command: ./casa
==3443==
==3443== Use of uninitialised value of size 4
==3443==    at 0x409A21B: _itoa_word (_itoa.c:179)
==3443==    by 0x409D928: vfprintf (vfprintf.c:1660)
==3443==    by 0x40A443E: printf (printf.c:33)
==3443==    by 0x8048469: main (archivo37.c:6)
==3443==
```

Cómo depurar un programa (VI)

Pero confirmamos **que no estaba bien…** 😱

```
==3443== Invalid read of size 4
==3443==    at 0x8048481: main (archivo37.c:9)
==3443==  Address 0x4209030 is 0 bytes after a block of size 8 alloc'd
==3443==    at 0x402917C: malloc (in /usr/lib/valgrind/vgpreload_memche
nux.so)
==3443==    by 0x8048475: main (archivo37.c:8)
==3443==
Valor fuera de rango: 0
==3443== Invalid write of size 4
==3443==    at 0x804849A: main (archivo37.c:10)
==3443==  Address 0x4209030 is 0 bytes after a block of size 8 alloc'd
==3443==    at 0x402917C: malloc (in /usr/lib/valgrind/vgpreload_memche
nux.so)
==3443==    by 0x8048475: main (archivo37.c:8)
```

Cómo depurar un programa (VI)

Pero confirmamos **que no estaba bien…** 

```
==3443== HEAP SUMMARY:
==3443==     in use at exit: 8 bytes in 1 blocks
==3443==   total heap usage: 1 allocs, 0 frees, 8 bytes allocated
==3443==
==3443== LEAK SUMMARY:
==3443==    definitely lost: 8 bytes in 1 blocks
==3443==    indirectly lost: 0 bytes in 0 blocks
==3443==      possibly lost: 0 bytes in 0 blocks
==3443==    still reachable: 0 bytes in 0 blocks
==3443==         suppressed: 0 bytes in 0 blocks
==3443== Rerun with --leak-check=full to see details of leaked memory
==3443==
==3443== For counts of detected and suppressed errors, rerun with: -v
==3443== Use --track-origins=yes to see where uninitialised values come from
==3443== ERROR SUMMARY: 25 errors from 9 contexts (suppressed: 0 from 0)
```

Y esto, tan útil, sin hacer más que ejecutar nuestro programa bajo el paraguas de Valgrind. Tiene muchas más opciones que se pueden explorar (pero no entraremos).

Contenidos

- ¿Qué es C?
- Entorno de trabajo
- ¡Hola Mundo! (Y Compilación)
- Variables y constantes. Ejemplos
- Funciones. Ejemplos
- Punteros (a datos y funciones). Ejemplos
- Memoria y cómo pedirla (y liberarla!)
- Estructuras de control. Ejemplos
- Cómo depurar un programa
- **Recibiendo parámetros por consola**
- Creando valores aleatorios
- Midiendo tiempos
- Lectura y escritura de archivos de texto
- Aplicación: Creando una librería para operar con matrices

Recibiendo parámetros por consola (I)

- Como habréis observado, muchos programas de consola reciben argumentos…
- Eso podemos hacerlo fácilmente usando la otra signatura admitida para la función *"main"*:

~~`int main(void);`~~

Esto nos va a **decir <u>cuántos</u> argumentos recibimos** (**siendo siempre >= 1** porque el nombre del programa se pasa por convención como primer argumento)

```
int main(int argc, char* argv[]);
```

```
int main(int argc, char** argv);
```

Equivalente

Recibiendo parámetros por consola (I)

- Como habréis observado, muchos programas de consola reciben argumentos…
- Eso podemos hacerlo fácilmente usando la otra signatura admitida para la función *"main"*:

~~`int main(void);`~~

Esto se ve como la **matriz de parámetros** que se espera recibir… será texto (Strings), y podemos ver cada fila como un argumento. Es decir, si nuestro programa se llama prog… argv[0] nos llevará a "prog" (array de char's)

```
int main(int argc, char* argv[]);
```

```
int main(int argc, char** argv);
```

Equivalente

Recibiendo parámetros por consola (II)

- Y bueno… **la verdad es que normalmente querremos números** también, no sólo cadenas de texto. Podemos usar para eso las siguientes funciones definidas en <***stdlib.h***>:

  - Pasamos Strings, i.e., arrays de caracteres terminados en '\0' o 0, (el comienzo de cada fila de la hipotética matriz de argumentos)….

```
int atoi (const char * str);
```

Y obtenemos un entero…

```
double atof (const char* str);
```

Y obtenemos un real…

Es lo mínimo… hay otras funciones para convertir Strings a otras cosas como long double… pero **no vamos a entrar más allá**.

Recibiendo parámetros por consola (III)

- Vamos finalmente a un ejemplo:

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv){
    printf("Esto siempre: %s\n", argv[0]);//Voy a la primera fila
    if(argc<3){//Si no controlamos esto, casi siempre se rompera el programa
        printf("Error, se esperaba un entero y un double\n");
        printf("./prog <entero> <double>\n");
    }else{
        int par1 = atoi(argv[1]);
        double par2 = atof(argv[2]);
        printf("Recibo: %d, %.3lf\n", par1, par2);
    }
    return 0;
}
```

archivo38.c

```
MBP-de-Nicolas:Codigo nccruz$ gcc -o casa archivo38.c
MBP-de-Nicolas:Codigo nccruz$ ./casa
Esto siempre: ./casa
Error, se esperaba un entero y un double
./prog <entero> <double>
MBP-de-Nicolas:Codigo nccruz$ ./casa 3 3.141592
Esto siempre: ./casa
Recibo: 3, 3.142
```

Contenidos

- ¿Qué es C?
- Entorno de trabajo
- ¡Hola Mundo! (Y Compilación)
- Variables y constantes. Ejemplos
- Funciones. Ejemplos
- Punteros (a datos y funciones). Ejemplos
- Memoria y cómo pedirla (y liberarla!)
- Estructuras de control. Ejemplos
- Cómo depurar un programa
- Recibiendo parámetros por consola
- **Creando valores aleatorios**
- Midiendo tiempos
- Lectura y escritura de archivos de texto
- Aplicación: Creando una librería para operar con matrices

Creando valores aleatorios (I)

- Otra tarea que podríamos necesitar frecuentemente es la generación de valores aleatorios.

- Para esta tarea podemos usar la función rand(), definida también en <***stdlib.h***>:

```
int rand(void);
```

- Nos va a dar un entero definido entre 0 y RAND_MAX… una constante a la que podemos acceder. Su valor depende de la implementación pero que, según el estándar **debe ser al menos 32767**.

Creando valores aleatorios (II)

- Muy bien… pero ¿y si queremos acotar el rango de los enteros?
- Pues podemos hacer el módulo:

```
int val = (rand() % (MI_MAX+1)); //Ahora el valor estará en [0, MI_MAX]
```

- También podemos sumar un mínimo:

```
int val = (rand() % (MI_MAX-MI_MIN+1)) + MI_MIN; //Ahora el valor estará
                                                 //en [MI_MIN, MI_MAX]
```

- Y multiplicar por -1, dividir…

Creando valores aleatorios (III)

- Perfecto… ¿y si queremos reales?

- Pues podemos crear decimales en rango 0, 1 con RAND_MAX:

```
double val = (double) rand() / RAND_MAX; //Usa bien el casting o tendrás
                                         //un bonito valor 0 / 1
```

- Con un valor decimal en rango 0,1 podemos escalarlo:

```
double val = ((double) rand() / RAND_MAX)*[MI_MAX-MI_MIN] + MI_MIN;
```

[0, 1] *(MAX-MIN) + MIN

Creando valores aleatorios (IV)

- Si usáis cualquiera de esas funciones probablemente os resulte curioso esto: tal cual, cada vez que ejecutemos el programa dará la misma secuencia de números aleatorios…

- Por ejemplo, pedimos 3 valores y vemos: 0.3, 1.9, 2.7.

  Si lo ejecutamos otra vez veremos…: 0.3, 1.9, 2.7. Igual,

  y si lo hacemos otra vez…: 0.3, 1.9, 2.7. Igual.

- **¿Y esto a qué se debe?** 

Creando valores aleatorios (IV)

- **Los números aleatorios en Ciencias de la Computación no *existen*, son** valores **"pseudo-aleatorios"** que se calculan **siguiendo una secuencia** a partir de un estado inicial o "semilla".

- El diseño de nuevas técnicas para obtener aleatorios (p.ej., algoritmo *Mersenne Twister*) es en sí mimo un área de investigación **muy relevante para temas como la Seguridad Informática…** y excede el alcance de este curso.

- Lo importante es que sepamos que **para nuestro programas podremos controlar las semillas de aleatoriedad…** porque si no, el compilador la fijará al crear el ejecutable final, y tendremos siempre los mismos valores aleatorios.

Creando valores aleatorios (IV)

- La función para fijar una semilla de aleatoriedad, también definida en <***stdlib.h***>, es:

```
void srand(unsigned int semilla);
```

- **Una misma semilla dará una misma secuencia** infinita de valores… así que ¿cómo hacer que se cambie cada vez sin tener que leer nuevas?

- ¡Pues haciendo que se defina con el instante de tiempo actual! Para eso se suele usar la función “time” definida en <time.h>:

```
srand(time(0));
```

Nota: Este proceso se suele hacer una única vez en toda la ejecución del programa

Creando valores aleatorios (V)

archivo39.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char** argv){
    srand(time(0));//Si me comentas, veras siempre las mismas salidas
    int valRand = rand();
    printf("Entero aleatorio entre 0 y RAND_MAX: %d\n", valRand);
    valRand = rand() % 6;
    printf("Entero aleatorio entre 0 y 5: %d\n", valRand);
    valRand = (rand()%(10-3+1)) + 3;
    printf("Entero aleatorio entre 3 y 10: %d\n", valRand);
    printf("Y ahora reales...");
    double realRand = ((double) rand() / RAND_MAX);
    printf("Real aleatorio entre 0 y 1: %.3lf\n", realRand);
    realRand = ((double) rand() / RAND_MAX)*30.0;
    printf("Real aleatorio entre 0 y 30: %.3lf\n", realRand);
    realRand = ((double) rand() / RAND_MAX)*(40.-10.)+10.0;
    printf("Real aleatorio entre 10 y 40: %.3lf\n", realRand);
    return 0;
}
```

```
Entero aleatorio entre 0 y RAND_MAX: 1940569440
Entero aleatorio entre 0 y 5: 5
Entero aleatorio entre 3 y 10: 6
Y ahora reales...Real aleatorio entre 0 y 1: 0.953
Real aleatorio entre 0 y 30: 19.878
Real aleatorio entre 10 y 40: 26.658
```

Contenidos

- ¿Qué es C?
- Entorno de trabajo
- ¡Hola Mundo! (Y Compilación)
- Variables y constantes. Ejemplos
- Funciones. Ejemplos
- Punteros (a datos y funciones). Ejemplos
- Memoria y cómo pedirla (y liberarla!)
- Estructuras de control. Ejemplos
- Cómo depurar un programa
- Recibiendo parámetros por consola
- Creando valores aleatorios
- **Midiendo tiempos**
- Lectura y escritura de archivos de texto
- Aplicación: Creando una librería para operar con matrices

Midiendo tiempos (I)

- ¿**Y si queremos saber e indicar cuánto tarda nuestro programa o una parte** del mismo?

- Si es **todo el programa** y estamos en un entorno UNIX podemos usar el **comando *time***:

```
MacBook-Pro-de-Nicolas:Codigo nccruz$ time ./casa
Valor sin inicializar: 254783542
Valor fuera de rango: 0

real    0m0.004s
user    0m0.001s
sys     0m0.002s
```

Esto es lo que nos suele interesar

El tiempo total que pasa en la realidad se indica en el campo "real", y se conoce también como "Wall time", es decir, el tiempo que transcurre sobre un reloj de pare…

Tiempo puro de CPU (usando paralelismo podemos verlo multiplicado aunque el real sea inferior)

Tiempo que el núcleo del sistema interviene en la ejecución

Midiendo tiempos (II)

- Si lo que nos interesa son partes concretas, podemos definir puntos de medición del tiempo real entre tareas. Con esta función podemos **anotar un cierto instante de tiempo en segundos**:

```
#include <time.h>
#include <sys/time.h>
double get_wall_time(){
    struct timeval time;
    if (gettimeofday(&time,NULL)){
        printf("Error de medición de tiempo\n");
        return 0;
    }
    return (double) time.tv_sec + (double) time.tv_usec * 0.000001;
}
```

Fuente: https://stackoverflow.com/questions/17432502/how-can-i-measure-cpu-time-and-wall-clock-time-on-both-linux-windows

(Consultar para opciones multi-plataforma)

Pasamos la información del instante *time* a segundos

Midiendo tiempos (III)

archivo40.c

```c
#include <stdio.h>
#include <time.h>
#include <sys/time.h>
#include <unistd.h>//Para sleep (Linux/UNIX)


//De: https://stackoverflow.com/questions/17432502/
    //how-can-i-measure-cpu-time-and-wall-clock-time-on-both-linux-windows
double get_wall_time(){
    struct timeval time;
    if (gettimeofday(&time,NULL)){
        printf("Error de medición de tiempo\n");
        return 0;
    }
    return (double) time.tv_sec + (double) time.tv_usec * 0.000001;
}
```

```c
int main(void){
    double instanteA = get_wall_time();
    sleep(3);//Simulamos una tarea de 3 segundos
    double instanteB = get_wall_time();
    printf("He tardado: %.3lf segundos\n", instanteB-instanteA);
    return 0;
}
```

```
He tardado: 3.005 segundos
```

Contenidos

- ¿Qué es C?
- Entorno de trabajo
- ¡Hola Mundo! (Y Compilación)
- Variables y constantes. Ejemplos
- Funciones. Ejemplos
- Punteros (a datos y funciones). Ejemplos
- Memoria y cómo pedirla (y liberarla!)
- Estructuras de control. Ejemplos
- Cómo depurar un programa
- Recibiendo parámetros por consola
- Creando valores aleatorios
- Midiendo tiempos
- **Lectura y escritura de archivos de texto**
- Aplicación: Creando una librería para operar con matrices

Lectura/esc. de archivos de texto (I)

- Los archivos se clasifican en **dos tipos base** para Entrada/Salida:
  - **Texto**
  - **Binarios**

- En última instancia todos se traducen en bits… pero **los de texto hacen referencia a caracteres** mientras que los de **los binarios pueden representar diversos tipos de datos** (enteros, reales…).

- En este curso **vamos a ver cómo leer y escribir archivos de texto con C**, que puede ser muy útil para leer archivos de configuración y escribir resultados.

Lectura/esc. de archivos de texto (II)

- Las funciones para lectura escritura de archivos están definidas en <***stdio.h***>. Las que vamos a usar son:

```
FILE* fopen(const char * ruta, const char * modo);
```

```
int fprintf(FILE* archivo, const char* formato, valores);
```

Escribir

```
int fscanf(FILE* archivo, const char* formato, punteros_para_escribir…);
```

Leer

```
int fclose(FILE* archivo);
```

Hay otras como fgetc y fgets para leer y fputc y fputs para escribir, pero no soportan formato.

Lectura/esc. de archivos de texto (III)

```c
#include <stdio.h>

typedef struct{
    char nombre[20];
    int lados;
} Poligono;

int escribirPoligono(FILE* archivo, Poligono plg){
    int num_escrito = fprintf(archivo, "###_Nombre: %s\n", plg.nombre);
    if(num_escrito<=0){
        printf("Error al escribir: %s\n", plg.nombre);
        return 0;
    }else{
        num_escrito = fprintf(archivo, "Lados: %d\n", plg.lados);
        if(num_escrito<=0){
            printf("Error al escribir: %d\n", plg.lados);
            return 0;
        }
        return 1;
    }
}
```

Da el **número de caracteres escritos**…o un valor negativo **y sirve para comprobar errores**.

Da el **número salidas escritas al leer**… y lo usamos **para comprobar errores** (junto con los nombres de los campos inyectados en el formato).

```c
int leerPoligono(FILE* archivo, Poligono* out_plg){
    int check = fscanf(archivo, "###_Nombre: %s\n", out_plg->nombre);
    if(!check){
        printf("Error al leer el nombre del polígono\n");
        return 0;
    }else{
        check = fscanf(archivo, "Lados: %d\n", &(out_plg->lados));
        if(!check){
            printf("Error al leer el número de lados del polígono\n");
        return 0;
        }
        return 1;
    }
}
```

archivo41.c

Lectura/esc. de archivos de texto (III)

Hay además modos mixtos, apertura binara…

Modo:

r: Abrir para lectura
w: Abrir para escritura (sobreescribir)
a: Abrir para escritura (o añadir al final si existe)

archivo41.c

```c
int main(void){
    Poligono cuadrado = {"Cuadrado", 4};
    Poligono pentagono = cuadrado;
    sprintf(pentagono.nombre, "Pentágono");
    pentagono.lados = 5;
    //Escribimos:
    FILE* salida = fopen("archivo41_Poligonos.txt", "w");
    if(salida){
        if(escribirPoligono(salida, cuadrado)){
            escribirPoligono(salida, pentagono);
        }
    }
    fclose(salida);
```

Función hermana de printf y fprintf, y muy útil para escribir Strings sin tener que hacerlo carácter a carácter.

Jugamos con el corto-circuito de AND de C para evitar anidar condiciones al comparar.

```c
    //Leemos:
    FILE* entrada = fopen("archivo41_Poligonos.txt", "r");
    if(entrada){
        Poligono copias_leidas[2];
        int check = leerPoligono(entrada, &(copias_leidas[0]));
        if(check){
            printf("Leído: Nombre: %s; Lados: %d\n", copias_leidas[0].nombre,
                copias_leidas[0].lados);
        }
        check = check && leerPoligono(entrada, &(copias_leidas[1]));
        if(check){
            printf("Leído: Nombre: %s; Lados: %d\n", copias_leidas[1].nombre,
                copias_leidas[1].lados);
        }
    }
    fclose(entrada);
    return 0;
}
```

Cerramos siempre al terminar!

Lectura/esc. de archivos de texto (IV)

- Internet está muy bien para resolver dudas… ¿pero sabías que podemos tener un **manual dentro de la consola**?

man nombre_funcion

Si no está instalado podemos poner los siguientes paquetes:

```
sudo apt-get install manpages-dev
sudo apt-get install manpages-posix-dev
```

Codigo — less ◂ man printf — 80×24

```
PRINTF(1)                 BSD General Commands Manual                 PRINTF(1)

NAME
     printf -- formatted output

SYNOPSIS
     printf format [arguments ...]

DESCRIPTION
     The printf utility formats and prints its arguments, after the first,
     under control of the format.  The format is a character string which con-
     tains three types of objects: plain characters, which are simply copied
```

Contenidos

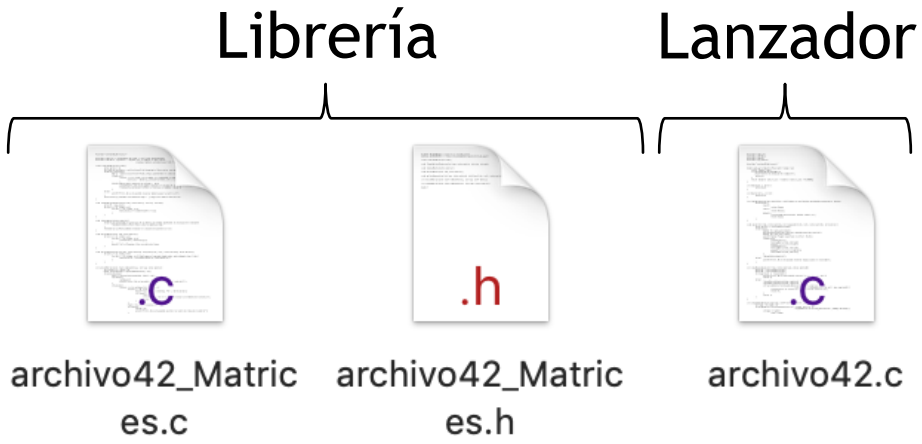
- ¿Qué es C?
- Entorno de trabajo
- ¡Hola Mundo! (Y Compilación)
- Variables y constantes. Ejemplos
- Funciones. Ejemplos
- Punteros (a datos y funciones). Ejemplos
- Memoria y cómo pedirla (y liberarla!)
- Estructuras de control. Ejemplos
- Cómo depurar un programa
- Recibiendo parámetros por consola
- Creando valores aleatorios
- Midiendo tiempos
- Lectura y escritura de archivos de texto
- **Aplicación: Creando una librería para operar con matrices**

Una librería para operar con matrices (I)

- Vamos a terminar el curso haciendo una librería sencilla para hacer algunas operaciones con matrices cuadradas de enteros.

- El modo 1 va a crear 2 matrices cuadradas del tamaño dado, va a guardarlas en disco, sumarlas (0) o restarlas (1) y mostrar los resultados.

- El modo 2 va a hacer lo mismo pero sin generar nuevos datos, sino leyendo los últimos que se guardaron.



Una librería para operar con matrices (II)

- Estas son las funciones que nos “comprometemos” a ofrecer en el archivo de declaración “.h”:

```
#ifndef A42_MATRICES // Esto es un include guard
#define A42_MATRICES // https://en.wikipedia.org/wiki/Include_guard

int** reservarMatriz(int tam);

void llenarMatrizAleatoria(int tam, int** matriz, int min, int max);

void liberarMatriz(int** matriz);

void mostrarMatriz(int tam, int** matriz);

void aplicarOperaction(int tam, int** matrizA, int(*func)(int, int),
                                                int** matrizB, int** matrizC);

int volcarMatriz(const char* nombreArchivo, int tam, int** matriz);

int cargarMatriz(const char* nombreArchivo, int* tam, int*** matriz);

#endif
```

archivo42_Matrices.h

Una librería para operar con matrices (III)

- Y pasamos a implementarlas…:

archivo42_Matrices.c

```c
#include "archivo42_Matrices.h"

#include <stdio.h> // Intentamos que cada .c sea lo mas autocontenido
#include <stdlib.h> //posible... dejando los .h tambien lo mas ligeros
                    //aunque tambien autonomos para todo tipo que definan/usen

int** reservarMatriz(int tam){
    int** matriz = 0;
    int* matriz_container = malloc(sizeof(int)*tam*tam);//Gran vector contiguo
    if(matriz_container){
        matriz = malloc(sizeof(int*)*tam);//Aqui guardaremos el comienzo de cada fila
        if(matriz){
            for(int i = 0; i<tam; i++){//Vamos a ir apuntando a cada fila
                matriz[i] = &(matriz_container[i*tam]);//El inicio de la fila i es i*columnas
            }
        }else{//Ahora pediir memoria ha fallado... pero:
            printf("Error. No se ha podido reservar la carcasa 2D de la matriz.\n");
            free(matriz_container);//Esto si funciono, y debemos limpiarlo
        }
    }else{
        printf("Error. No se ha podido reservar memoria para la matriz.\n");
    }
    return matriz;//Tenemos esta memoria heap... y luego otro debera liberarla eh
}
```

Una librería para operar con matrices (III)

- Y pasamos a implementarlas…:

archivo42_Matrices.c

```c
void llenarMatrizAleatoria(int tam, int** matriz, int min, int max){
    int diff = max-min+1;
    for(int i = 0; i<tam; i++){
        for(int j = 0; j<tam; j++){
            matriz[i][j] = (rand()%(diff)) + min;
        }
    }
}

void liberarMatriz(int** matriz){
    if(matriz){//Liberando el principio de la matriz ya estamos apuntando al principio del container
        free(matriz[0]);//Pero claro, solo si apunta a algo
    }
    free(matriz);//Ahora podemos eliminar el conjunto de punteros a fila
}

void mostrarMatriz(int tam, int** matriz){
    for(int i = 0; i<tam; i++){
        for(int j = 0; j<tam; j++){
            printf("%d ", matriz[i][j]);
        }
        printf("\n");//Terminar fila con salto de linea
    }
}
```

Una librería para operar con matrices (III)

- Y pasamos a implementarlas…: archivo42_Matrices.c

```
void aplicarOperaction(int tam, int** matrizA, int(*func)(int, int), int** matrizB, int** matrizC){
    for(int i = 0; i<tam; i++){
        for(int j = 0; j<tam; j++){//Aplicamos el operador dado sobre cada elemento (por filas)
            matrizC[i][j] = func(matrizA[i][j], matrizB[i][j]);
        }
    }
}
```

```
int volcarMatriz(const char* nombreArchivo, int tam, int** matriz){
    int salida = 1, check = 0;
    FILE* archivoSalida = fopen(nombreArchivo, "w");
    if(archivoSalida){
        check = fprintf(archivoSalida, "%d\n", tam);
        if(check<=0){
            salida = 0;
            printf("Error. No se ha podido escribir el tamaño de la matriz\n");
        }
```

Una librería para operar con matrices (III)

- Y pasamos a implementarlas…: archivo42_Matrices.c

```
        if(salida){
            for(int i = 0; salida==1 && i<tam; i++){
                for(int j = 0; j<tam; j++){
                    check = fprintf(archivoSalida, "%d ", matriz[i][j]);
                    if(check<=0){
                        salida = 0;
                        printf("Error. No se ha podido escribir un elemento de la matriz");
                        break;
                    }
                }
                if(check){
                    check = fprintf(archivoSalida, "\n");
                    if(check<=0){
                        salida = 0;
                        printf("Error. No se ha podido escribir un salto de linea de la matriz");
                    }
                }
            }
        }
        fclose(archivoSalida);
    }else{
        salida = 0;
        printf("Error. No se ha podido abrir el archivo para escribir\n");
    }
    return salida;
```

Una librería para operar con matrices (III)

- Y pasamos a implementarlas…: archivo42_Matrices.c

```
int cargarMatriz(const char* nombreArchivo, int* tam, int*** matriz){
    int salida = 1, check = 0;
    FILE* archivoEntrada = fopen(nombreArchivo, "r");
    if(archivoEntrada){
        int buffer = 0;
        check = fscanf(archivoEntrada, "%d\n", &buffer);
        if(check==1){
            *tam = buffer;//Modificamos el valor en la funcion que llamo (paso por referencia)
        }else{
            salida = 0;
            printf("Error. No se ha encontrado el tamaño de la matriz\n");
        }
        if(salida){
            (*matriz) = reservarMatriz(buffer);//Ey, vamos a escribir tambien "fuera" (referencia)
            if(*matriz){
                int miTam = buffer;
                for(int i = 0; salida==1 && i<miTam; i++){
                    for(int j = 0; j<miTam; j++){
                        check = fscanf(archivoEntrada, "%d ", &buffer);
                        if(check==1){
                            (*matriz)[i][j] = buffer;//Escribiendo fuera... (referencia)
                        }else{
```

Una librería para operar con matrices (III)

- Y pasamos a implementarlas…:

archivo42_Matrices.c

```
                            salida = 0;
                            liberarMatriz(*matriz);
                            printf("Error. No se ha podido leer el elemento de la matriz\n");
                            break;
                        }
                    }
                }
            }
        }
        fclose(archivoEntrada);
    }else{
        salida = 0;
        printf("Error. No se ha podido abrir el archivo para leer\n");
    }
    if(!salida){
        *matriz = 0;//Somos amables y la dejamos apuntada a NULL por seguridad
    }
    return salida;
}
```

Una librería para operar con matrices (IV)

- Finalmente, vamos a crear el programa deseado con nuestra librería:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>

#include "archivo42_Matrices.h"

double get_wall_time(){//Para medir tiempo real
    struct timeval time;
    if (gettimeofday(&time,NULL)){
        printf("Error de medición de tiempo\n");
        return 0;
    }
    return (double) time.tv_sec + (double) time.tv_usec * 0.000001;
}

int Sumar(int a, int b){
    return a+b;
}

int Restar(int a, int b){
    return a-b;
}
```

archivo42.c

Una librería para operar con matrices (IV)

- Finalmente, vamos a crear el programa deseado con nuestra librería:

```
int (*elegirFuncion(int tipo))(int, int){//Esto es una funcion que devuelve punteros a funcion
    switch(tipo){
        case 0:
            return Sumar;
        case 1:
            return Restar;
        default:
            printf("Modo desconocido. Usando 'Sumar'\n");
            return Sumar;
    }
}
```

archivo42.c

Una librería para operar con matrices (IV)

- Finalmente, vamos a crear el programa deseado con nuestra librería:

```
void operar(int tam, int** matrizA, int (*operador)(int, int), int** matrizB, int mostrar){
    int** matrizC = reservarMatriz(tam);
    if(matrizC){
        double TA = get_wall_time();
        aplicarOperaction(tam, matrizA, operador,matrizB, matrizC);
        double TB = get_wall_time();
        printf("Hecho! Tiempo invertido: %.3lf\n", TB-TA);
        if(mostrar){
            printf("A:\n");
            mostrarMatriz(tam, matrizA);
            printf("--------------\n");
            printf("B:\n");
            mostrarMatriz(tam, matrizB);
            printf("--------------\nC:\n");
            mostrarMatriz(tam, matrizC);
        }
        liberarMatriz(matrizC);
    }else{
        printf("Error. No se ha podido reservar espacio para el resultado");
    }
}
```

archivo42.c

Una librería para operar con matrices (IV)

- Finalmente, vamos a crear el programa deseado con nuestra librería:

```
int crearNuevosDatos(int tam, int*** matrizA, int*** matrizB){
    *matrizA = reservarMatriz(tam);
    *matrizB = reservarMatriz(tam);
    if(!(*matrizA) || !(*matrizB)){
        printf("Error. No se han podido reservar las matrices A y B\n");
        return 0;
    }else{
        llenarMatrizAleatoria(tam, *matrizA, 0, 10);
        llenarMatrizAleatoria(tam, *matrizB, 0, 10);
        if(!volcarMatriz("archivo42_MatrizA.txt", tam, *matrizA) ||
                        !volcarMatriz("archivo42_MatrizB.txt", tam, *matrizB)){
            printf("Error al volcar las nuevas matrices a disco.\n");
            return 0;
        }
        return 1;
    }
}
```

archivo42.c

Una librería para operar con matrices (IV)

- Finalmente, vamos a crear el programa deseado con nuestra librería:

```
int cargarDatosPrevios(int* tam, int*** matrizA, int*** matrizB){
    int tamA = 0, tamB = 0;
    if(cargarMatriz("archivo42_MatrizA.txt", &tamA, matrizA) &&
                        cargarMatriz("archivo42_MatrizB.txt", &tamB, matrizB)){
        if(tamA == tamB){
            *tam = tamA;
            return 1;
        }else{
            printf("Error. Las matrices leídas no son compatibles.\n");
            return 0;
        }
    }else{
        printf("Error. No se han podido cargar las matrices desde archivo.\n");
        return 0;
    }
}
```

archivo42.c

Una librería para operar con matrices (IV)

- Finalmente, vamos a crear el programa deseado con nuestra librería:

```
int main(int argc, char* argv[]){
    if(argc!=3 && argc!=5){
        printf("Ejecuta el modo 1 para hacer un cálculo y guardar los datos.\n");
        printf("Ejecuta el modo 2 para hacer una operación con datos guardados.\n");
        printf("[1]: ./prog semilla tam operacion[0/1] mostrar\n");
        printf("[2]: ./prog operacion[0/1] mostrar\n");
    }else{
```

archivo42.c

Una librería para operar con matrices (IV)

- Finalmente, vamos a crear el programa deseado con nuestra librería:

```
    }else{
        int** matrizA = 0;
        int** matrizB = 0;
        int (*operador)(int, int) = elegirFuncion(atoi(argv[argc-2]));
        int mostrar = atoi(argv[argc-1]);
        int tam = 0, ok = 1;
        if(argc==5){
            srand(atoi(argv[1]));
            tam = atoi(argv[2]);
            ok = crearNuevosDatos(tam, &matrizA, &matrizB);
        }else{
            ok = cargarDatosPrevios(&tam, &matrizA, &matrizB);
        }
        if(ok){
            operar(tam, matrizA, operador, matrizB, mostrar);
        }
        liberarMatriz(matrizA);
        liberarMatriz(matrizB);
    }
    return 0;
}
```

archivo42.c

Una librería para operar con matrices (I)

- Y a probar...

```
MacBook-Pro-de-Nicolas:Codigo nccruz$ gcc -o casa archivo42.c archivo42_Matrices
.c
MacBook-Pro-de-Nicolas:Codigo nccruz$ ./casa
Ejecuta el modo 1 para hacer un cálculo y guardar los datos.
Ejecuta el modo 2 para hacer una operación con datos guardados.
[1]: ./prog semilla tam operacion[0/1] mostrar
[2]: ./prog operacion[0/1] mostrar
```

```
MacBook-Pro-de-Nicolas:Codigo nccruz$ ./casa 1234 3 0 1
Hecho! Tiempo invertido: 0.000
A:
9 5 3
8 8 0
1 10 5
--------------
B:
10 1 10
7 1 6
10 10 1
--------------
C:
19 6 13
15 9 6
11 20 6
```

```
MacBook-Pro-de-Nicolas:Codigo nccruz$ ./casa 0 1
Hecho! Tiempo invertido: 0.000
A:
9 5 3
8 8 0
1 10 5
--------------
B:
10 1 10
7 1 6
10 10 1
--------------
C:
19 6 13
15 9 6
11 20 6
```

Enlaces de interés

- Este es el **final del curso… ¡Espero que os sea de utilidad** y os **agradezco** vuestro interés!

- Por si queréis enlaces adicionales:

https://www.tutorialspoint.com/cprogramming/index.htm (Inglés)

https://www.w3schools.in/c-tutorial/ (Inglés)

http://www.it.uc3m.es/pbasanta/asng/course_notes/c_programming_part_es.html (Español)

- Y si queréis dar el salto a C++:

http://c.conclase.net

- Abarca C y C++, en Español, y con muy buenas explicaciones

https://www.haiku-os.org/development/learning_to_program_with_haiku/

- Empieza con C, llega a C++ y ya se centra en programación para Haiku (Inglés)

¡Muchas gracias!

Especial agradecimiento a:

- Asociación UNIA de la Universidad de Almería por invitarme a hacer este curso y a los alumnos que me recomendaron.
- Grupo de Investigación Supercomputación – Algoritmos (SAL) de la Universidad de Almería, del que formo parte, por darme acceso a todas sus instalaciones.
- A Juana López Redondo y José Domingo Álvarez Hervás por darle importancia a este curso y dejarme todo el tiempo que he necesitado para terminarlo.
- A StackOverflow: cuántas dudas se resuelven siempre consultando sus preguntas.

Si ves algún error no dudes en avisar: ncalvocruz@ual.es