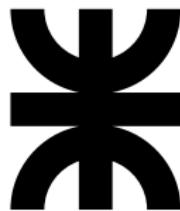


Informática II

Funcionamiento de la PC

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2021 –

Programación en lenguaje C

- ▶ ¿Cuál es el primer programa realizado en “*Informática I*”?

Programación en lenguaje C

- ▶ ¿Cuál es el primer programa realizado en “*Informática I*”?

hola_mundo.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hola mundo.\n");
6     return 0;
7 }
```

Programación en lenguaje C

- ▶ ¿Cuál es el primer programa realizado en “*Informática I*”?
- ▶ ¿Qué otro programa simple recuerda?

Programación en lenguaje C

- ▶ ¿Cuál es el primer programa realizado en “*Informática I*”?
- ▶ ¿Qué otro programa simple recuerda?

suma1.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int entero1, entero2, suma;
6
7     printf("Ingrese el primer entero: ");
8     scanf("%d", &entero1);
9     printf("Ingrese el segundo entero: ");
10    scanf("%d", &entero2);
11
12    suma = entero1 + entero2;
13    printf("La suma es: %d\n", suma);
14    return 0;
15 }
```

Programación en lenguaje C

- ▶ ¿Cuál es el primer programa realizado en “*Informática I*”?
- ▶ ¿Qué otro programa simple recuerda?

suma2.c

```
1 void main(void)
2 {
3     int a = 2, b = 3;
4     int c = a + b;
5 }
```

Programación en lenguaje C

- ▶ ¿Cuál es el primer programa realizado en “*Informática I*”?
- ▶ ¿Qué otro programa simple recuerda?

suma2.c

```
1 void main(void)
2 {
3     int a = 2, b = 3;
4     int c = a + b;
5 }
```

suma3.c

```
1 int main(void)
2 {
3     int a = 2, b = 3;
4     return (a + b);
5 }
```

Programación en lenguaje C

- ▶ ¿Cuál es el primer programa realizado en “*Informática I*”?
- ▶ ¿Qué otro programa simple recuerda?

suma2.c

```
1 void main(void)
2 {
3     int a = 2, b = 3;
4     int c = a + b;
5 }
```

suma3.c

```
1 int main(void)
2 {
3     int a = 2, b = 3;
4     return (a + b);
5 }
```

Observaciones:

1. El programa `hola_mundo.c` utiliza la biblioteca estándar

Programación en lenguaje C

- ▶ ¿Cuál es el primer programa realizado en “*Informática I*”?
- ▶ ¿Qué otro programa simple recuerda?

suma2.c

```
1 void main(void)
2 {
3     int a = 2, b = 3;
4     int c = a + b;
5 }
```

suma3.c

```
1 int main(void)
2 {
3     int a = 2, b = 3;
4     return (a + b);
5 }
```

Observaciones:

1. El programa `hola_mundo.c` utiliza la biblioteca estándar
2. El programa `suma1.c` tiene interacción con el usuario (`stdin`, `stdout`)

Programación en lenguaje C

- ▶ ¿Cuál es el primer programa realizado en “*Informática I*”?
- ▶ ¿Qué otro programa simple recuerda?

suma2.c

```
1 void main(void)
2 {
3     int a = 2, b = 3;
4     int c = a + b;
5 }
```

suma3.c

```
1 int main(void)
2 {
3     int a = 2, b = 3;
4     return (a + b);
5 }
```

Observaciones:

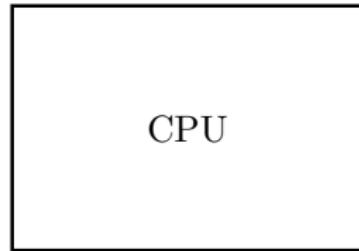
1. El programa `hola_mundo.c` utiliza la biblioteca estándar
2. El programa `suma1.c` tiene interacción con el usuario (`stdin`, `stdout`)
3. Los programas `suma2.c` y `suma3.c` no tienen interacción con el usuario

¿Cómo funciona la PC?

¿Qué función cumple una PC?

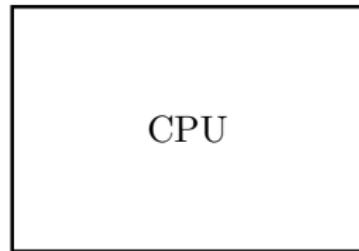
¿Cómo funciona la PC?

Procesa información/datos



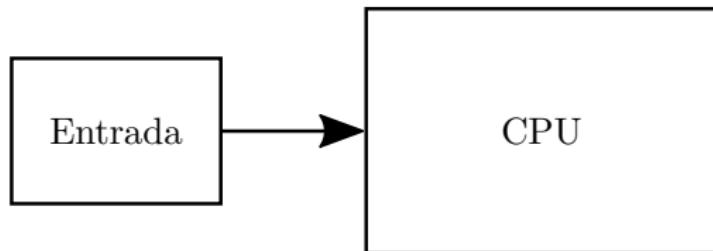
¿Cómo funciona la PC?

¿De donde se obtienen los datos?



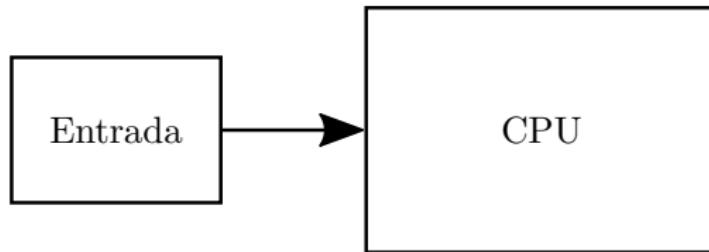
¿Cómo funciona la PC?

¿De donde se obtienen los datos?



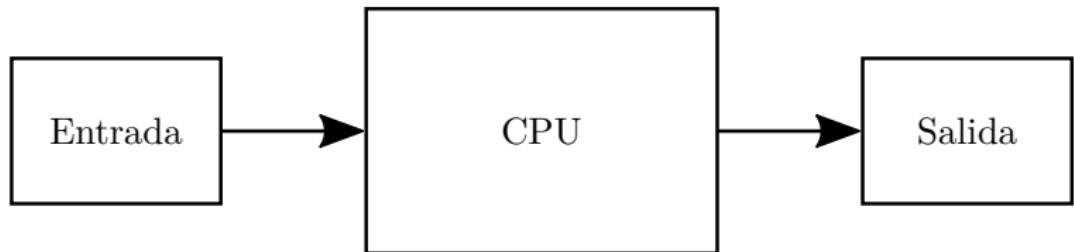
¿Cómo funciona la PC?

¿Cómo pone a disposición el resultado?



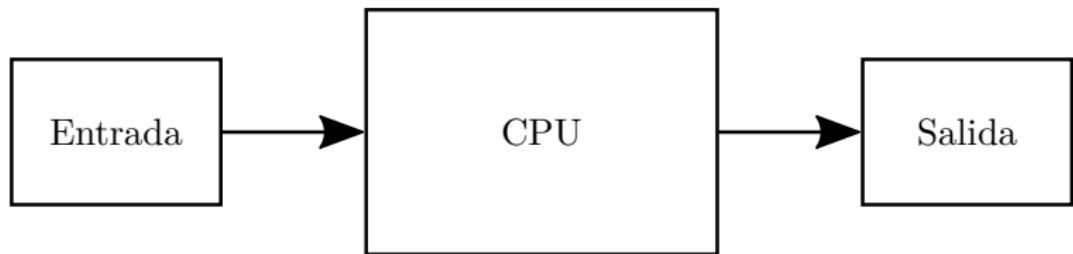
¿Cómo funciona la PC?

¿Cómo pone a disposición el resultado?



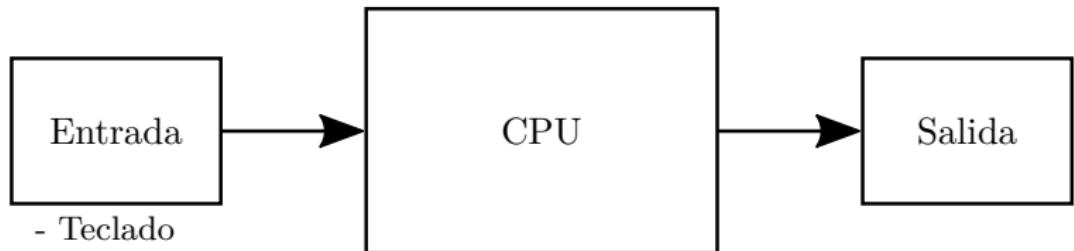
¿Cómo funciona la PC?

¿Qué dispositivos de **entrada** conoce?



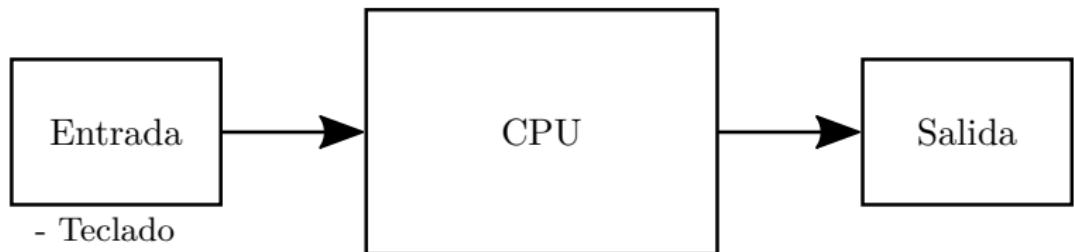
¿Cómo funciona la PC?

¿Qué dispositivos de **entrada** conoce?



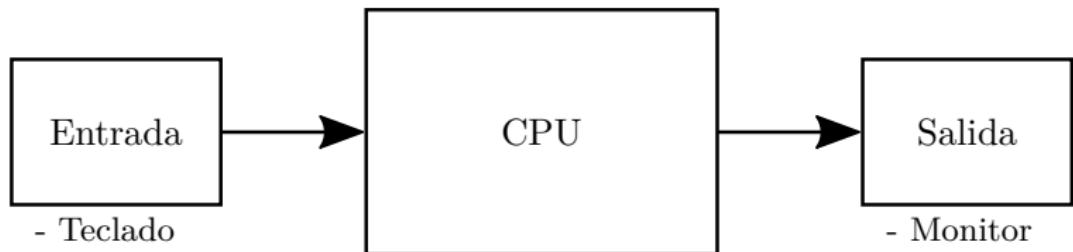
¿Cómo funciona la PC?

¿Qué dispositivos de **salida** conoce?



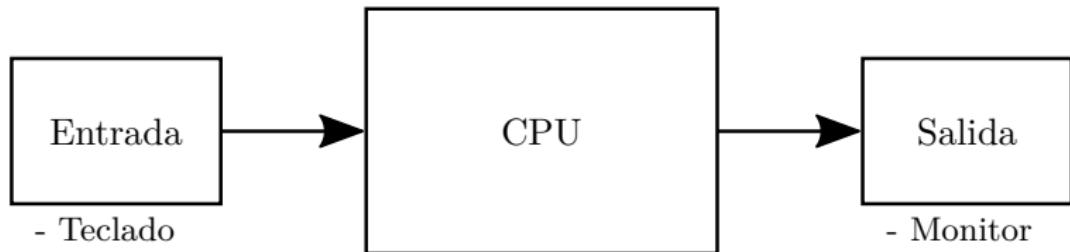
¿Cómo funciona la PC?

¿Qué dispositivos de **salida** conoce?



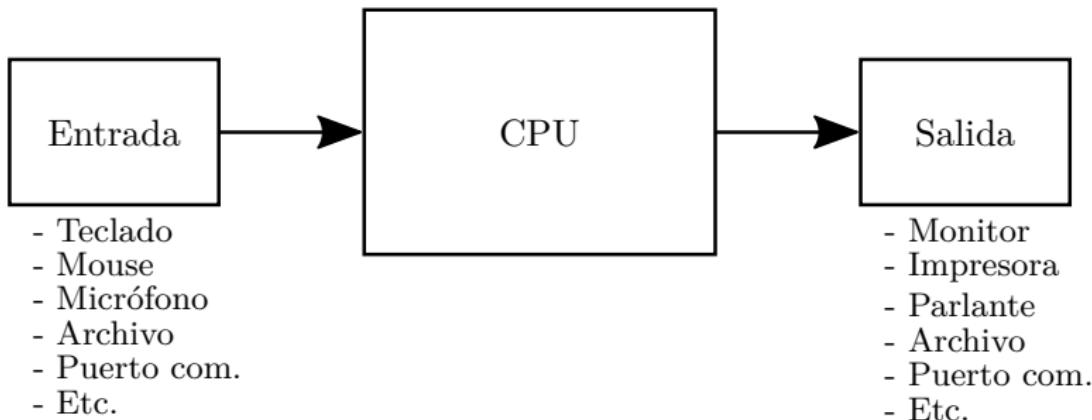
¿Cómo funciona la PC?

Otros dispositivos de entrada/salida



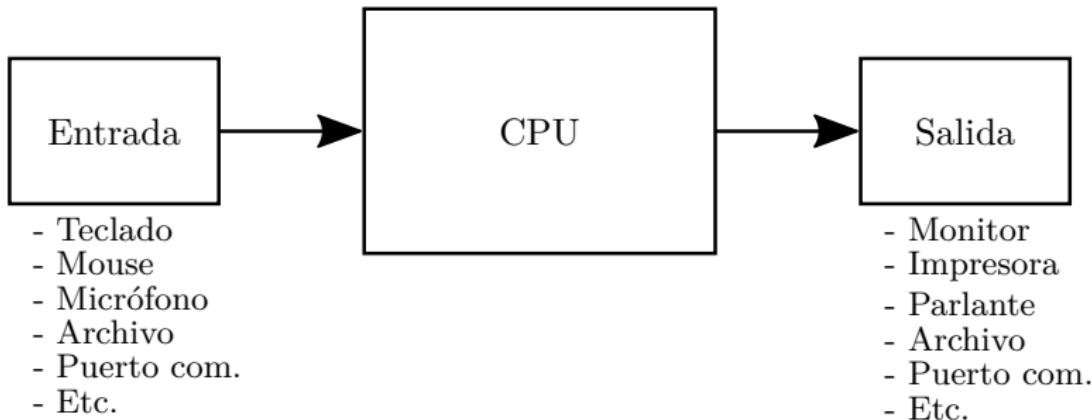
¿Cómo funciona la PC?

Otros dispositivos de entrada/salida



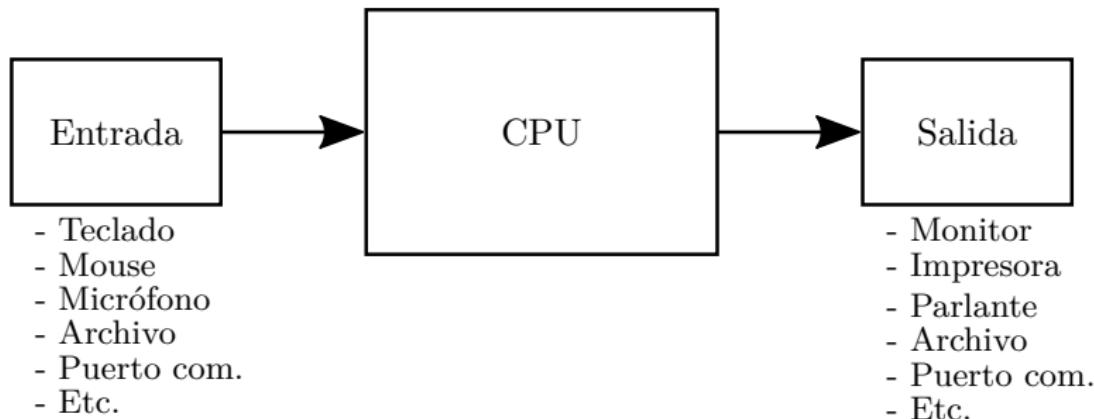
¿Cómo funciona la PC?

Analice el ejemplo de suma dos números (programa en C)



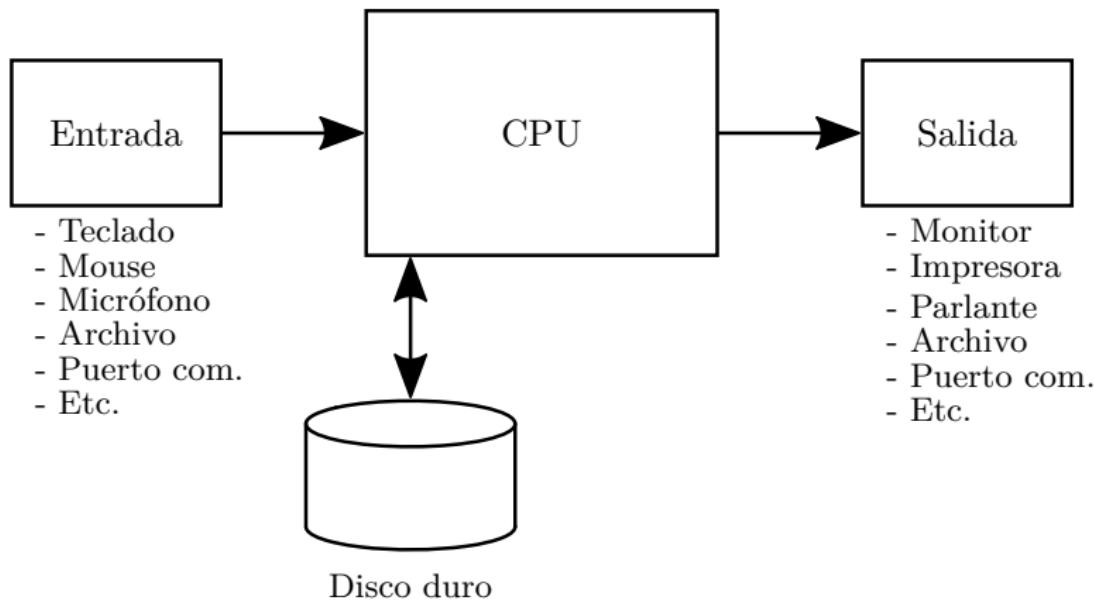
¿Cómo funciona la PC?

¿Dónde se guarda el programa **antes** de su ejecución?



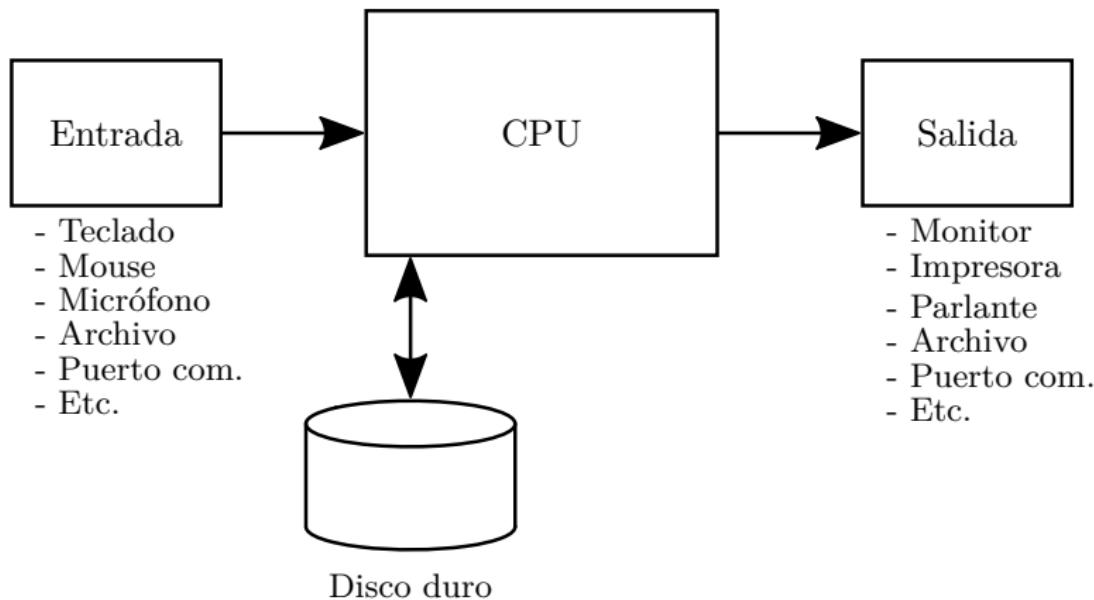
¿Cómo funciona la PC?

¿Dónde se guarda el programa **antes** de su ejecución?



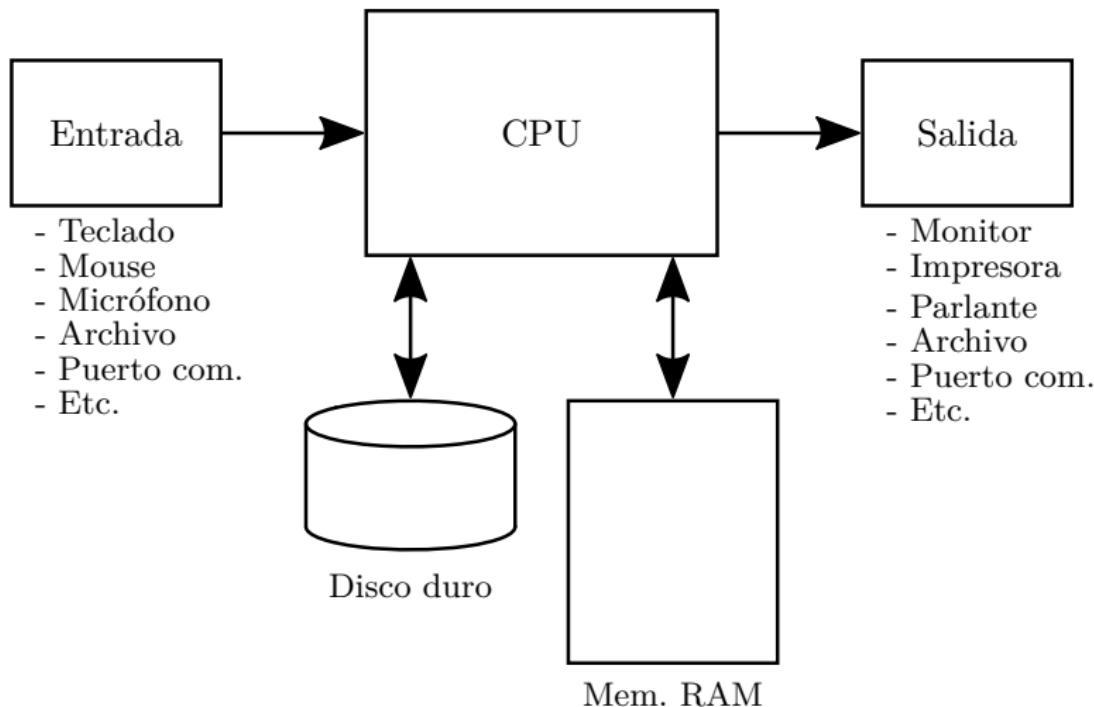
¿Cómo funciona la PC?

¿Dónde se guarda el programa **durante** su ejecución?



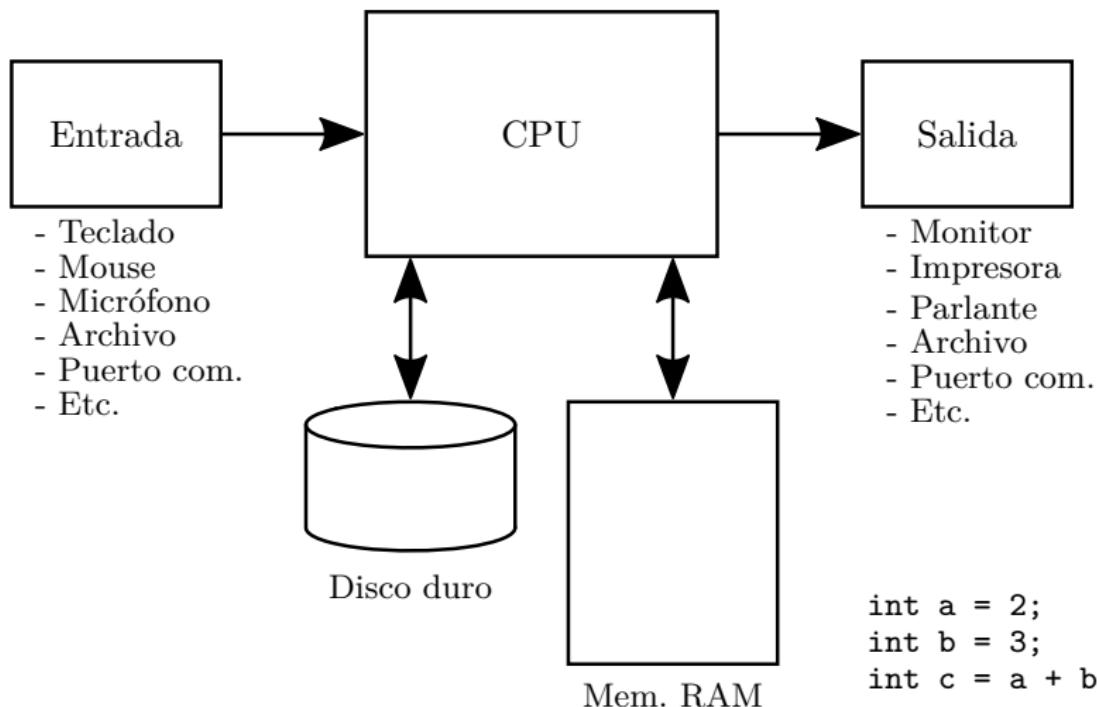
¿Cómo funciona la PC?

¿Dónde se guarda el programa **durante** su ejecución?



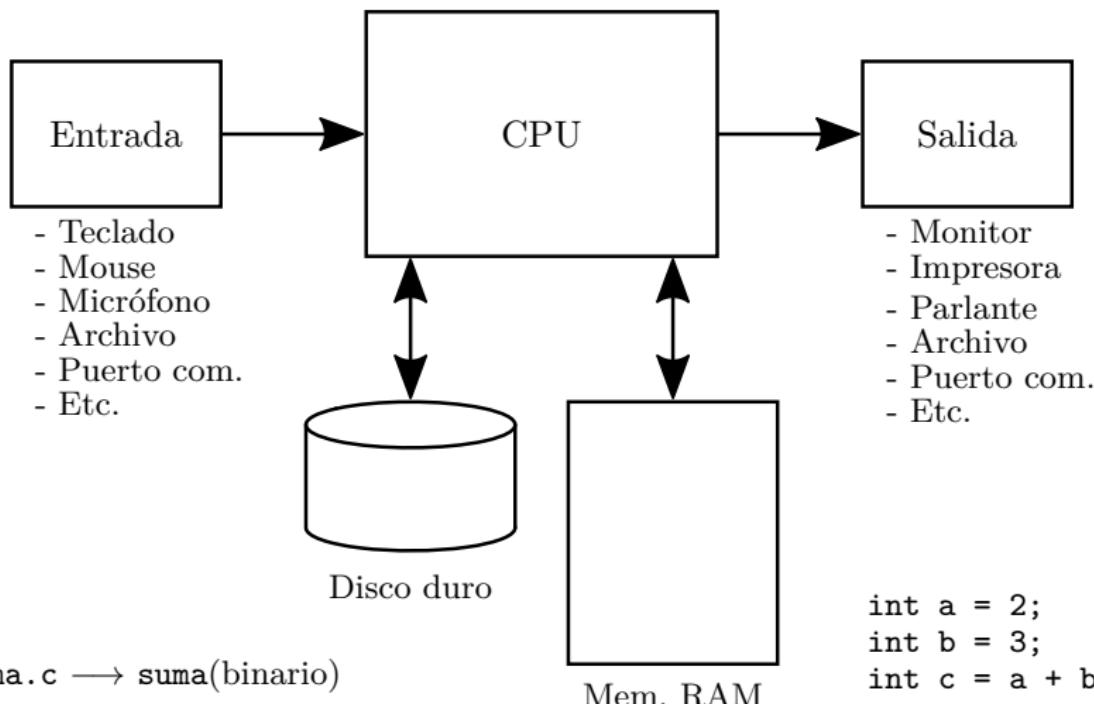
¿Cómo funciona la PC?

Programa que suma dos números



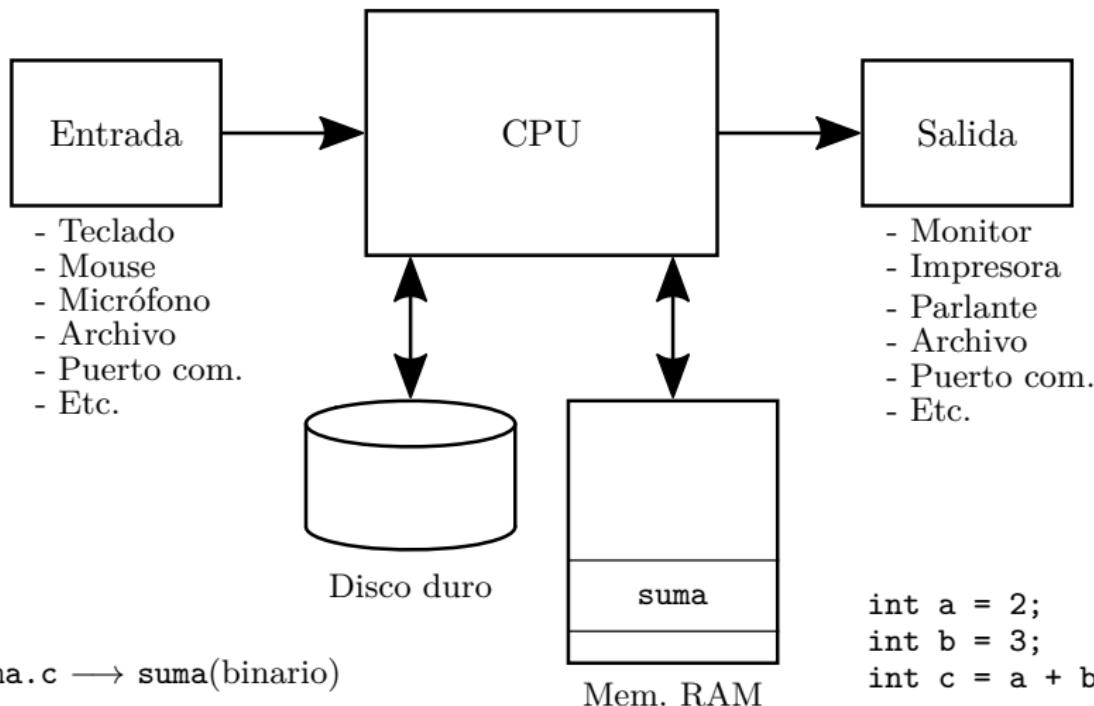
¿Cómo funciona la PC?

Programa que suma dos números



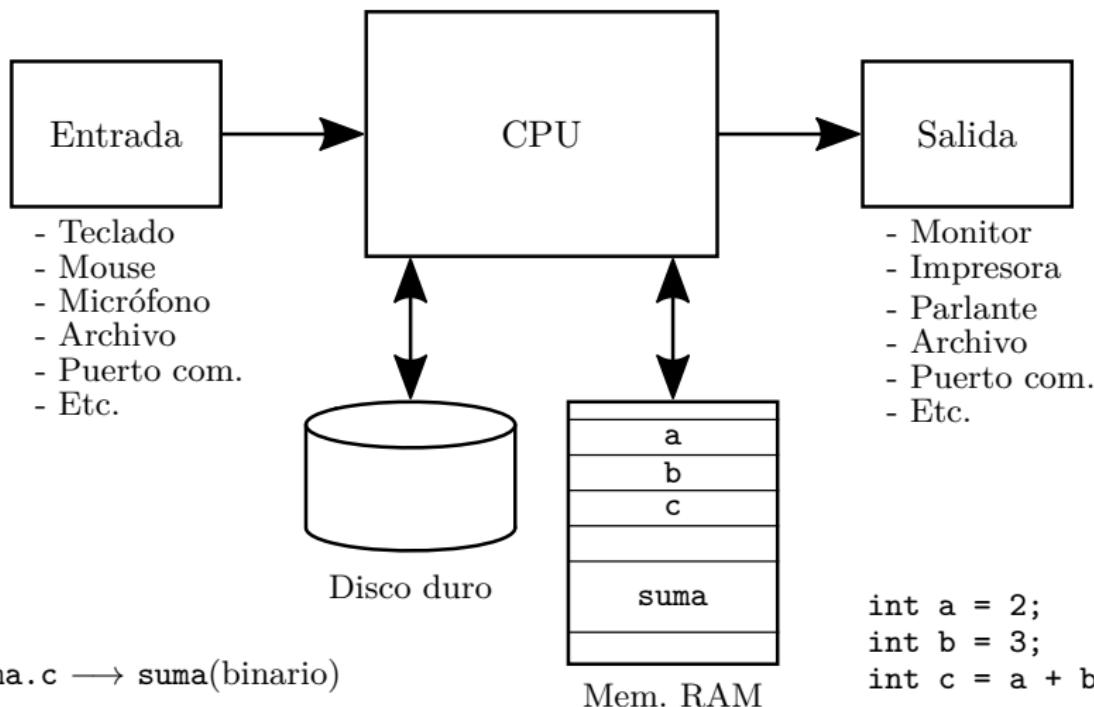
¿Cómo funciona la PC?

Programa que suma dos números



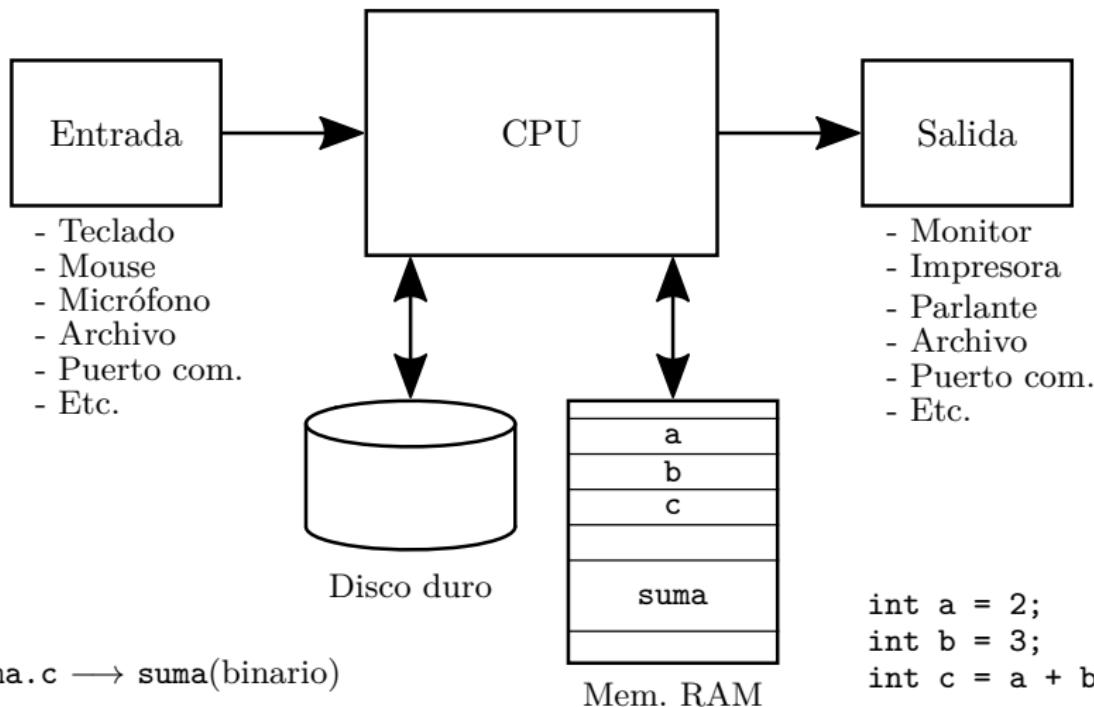
¿Cómo funciona la PC?

Programa que suma dos números



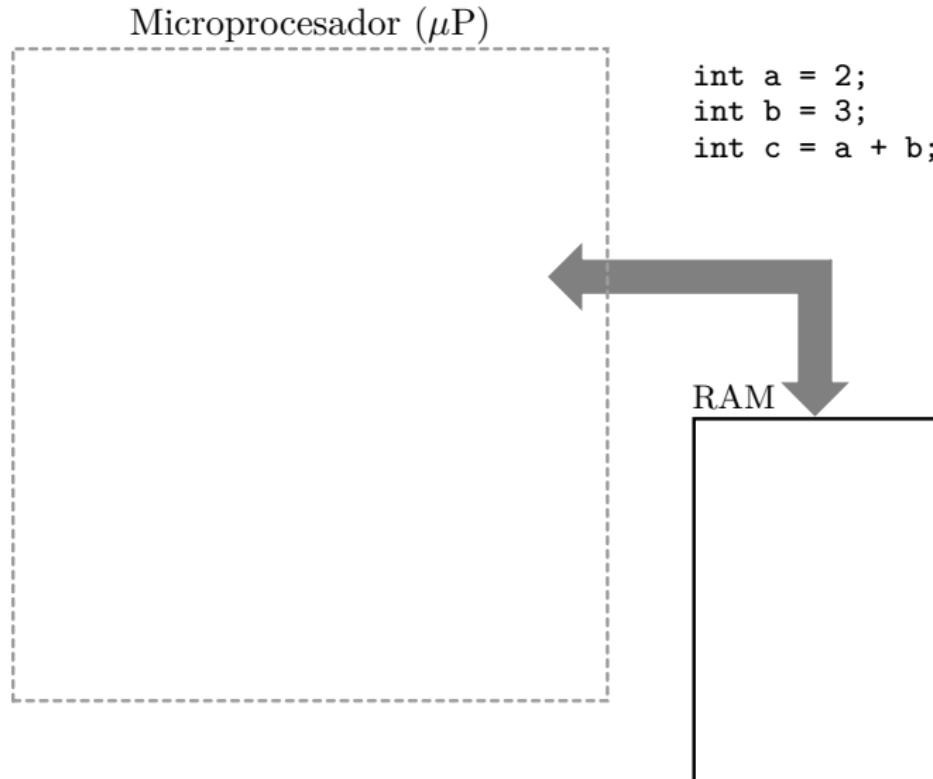
¿Cómo funciona la PC?

El tráfico de datos está centrado en el μ P



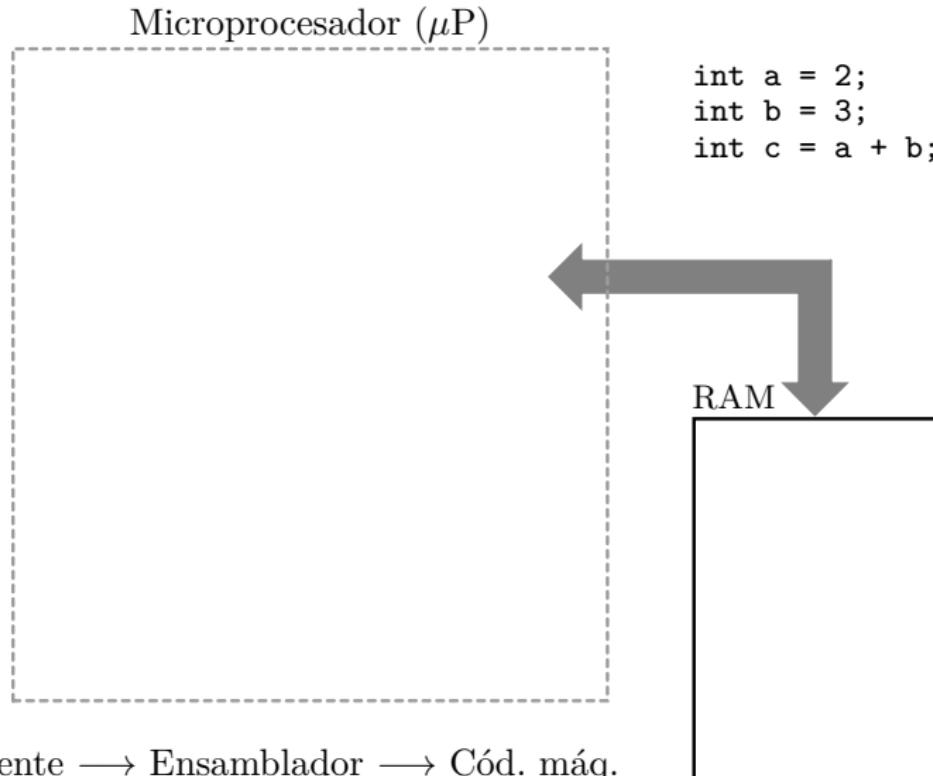
¿Cómo funciona la PC?

¿Qué formato tiene el archivo binario en memoria RAM?



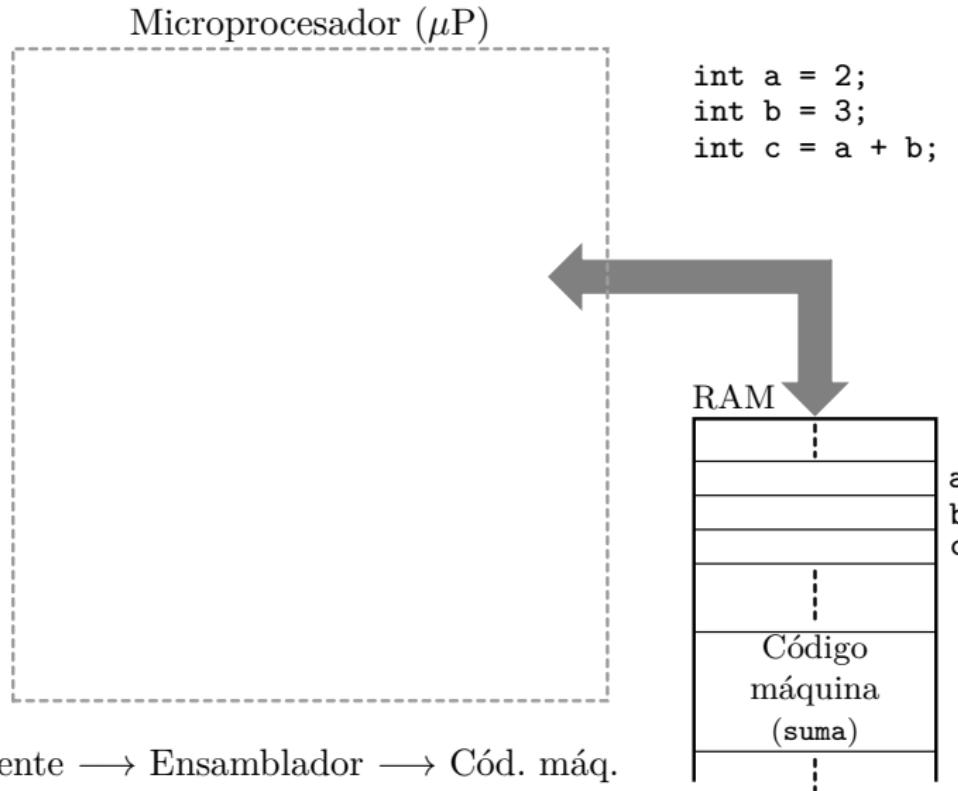
¿Cómo funciona la PC?

¿Qué formato tiene el archivo binario en memoria RAM?



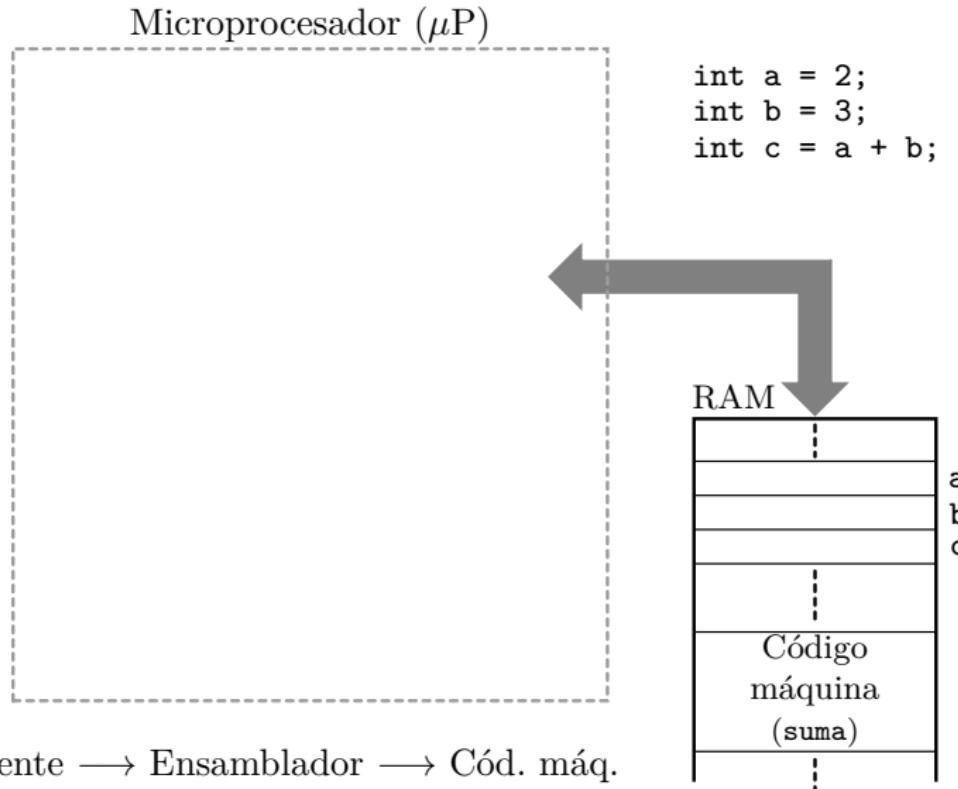
¿Cómo funciona la PC?

¿Qué formato tiene el archivo binario en memoria RAM?



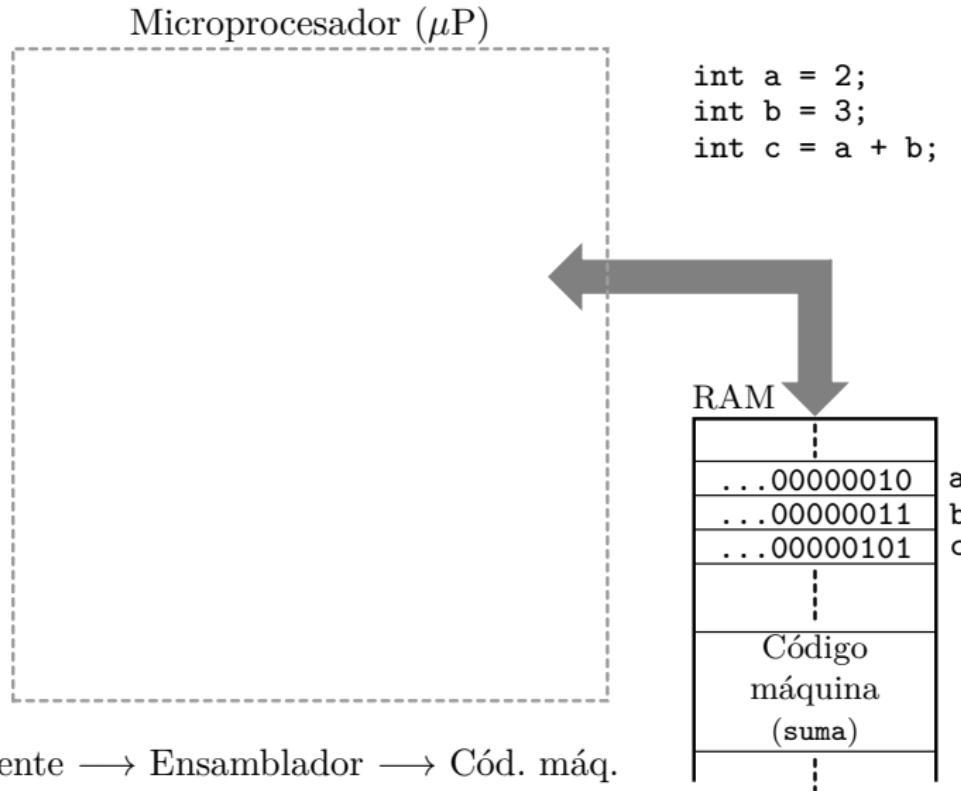
¿Cómo funciona la PC?

¿Qué formato tienen los datos en memoria RAM?



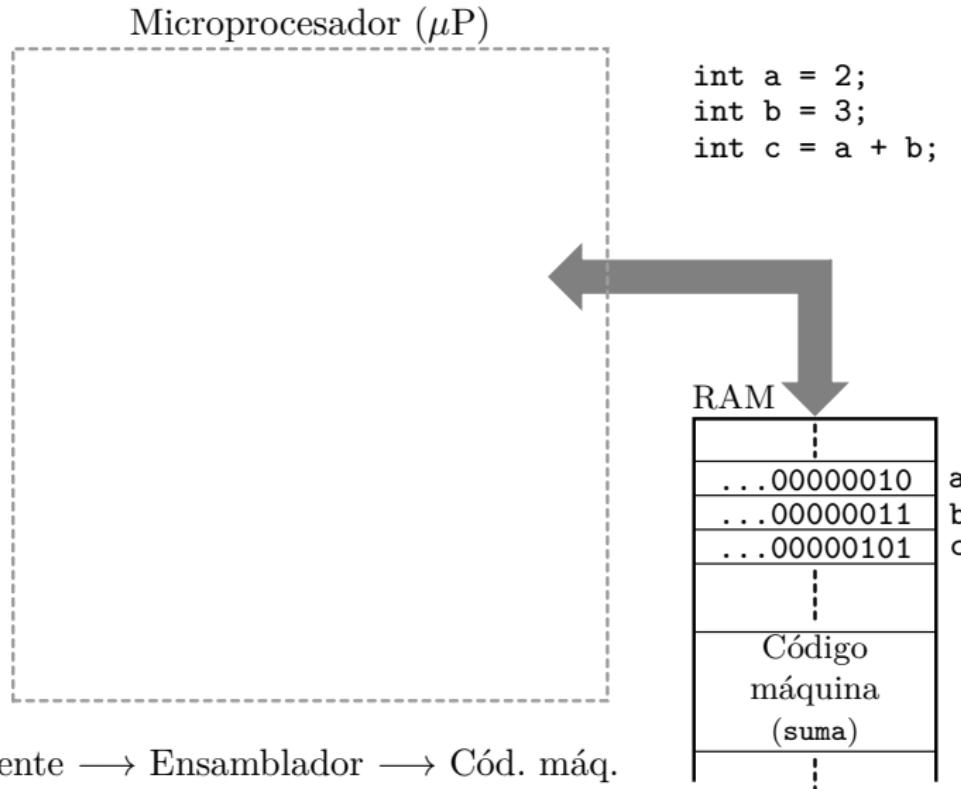
¿Cómo funciona la PC?

¿Qué formato tienen los datos en memoria RAM?



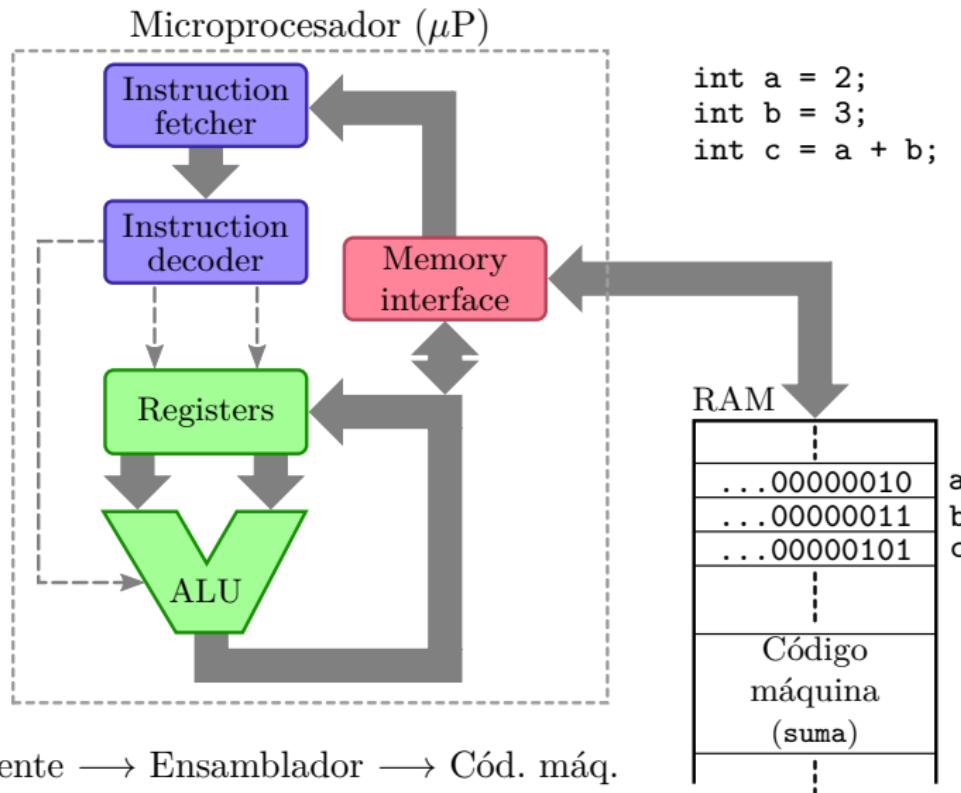
¿Cómo funciona la PC?

¿Cómo decodifica el programa la CPU para procesar los datos?



¿Cómo funciona la PC?

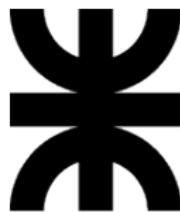
¿Cómo decodifica el programa la CPU para procesar los datos?



Informática II

Programación en C bajo GNU/Linux

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2021 –

Programar en lenguaje C

```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

Programar en lenguaje C

```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

- ▶ ¿Qué representa el texto de arriba? 

Programar en lenguaje C

```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

- ▶ ¿Qué representa el texto de arriba? 
- ▶ ¿Qué hay que hacer para obtener los sig. en la terminal? 

Hola mundo.

Programar en lenguaje C

```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

- ▶ ¿Qué representa el texto de arriba? 
- ▶ ¿Qué hay que hacer para obtener los sig. en la terminal? 

Hola mundo.

- ▶ ¿Qué herramienta utilizan?

IDE – Integrated Development Environment

Entorno de desarrollo integrado

Incluye un conjunto de herramientas informáticas para facilitar el desarrollo de software.

IDE – Integrated Development Environment

Entorno de desarrollo integrado

Incluye un conjunto de herramientas informáticas para facilitar el desarrollo de software. Normalmente consiste en:

IDE – Integrated Development Environment

Entorno de desarrollo integrado

Incluye un conjunto de herramientas informáticas para facilitar el desarrollo de software. Normalmente consiste en:

1. Editor de código fuente

IDE – Integrated Development Environment

Entorno de desarrollo integrado

Incluye un conjunto de herramientas informáticas para facilitar el desarrollo de software. Normalmente consiste en:

1. Editor de código fuente
2. Herramientas de construcción

IDE – Integrated Development Environment

Entorno de desarrollo integrado

Incluye un conjunto de herramientas informáticas para facilitar el desarrollo de software. Normalmente consiste en:

1. Editor de código fuente
2. Herramientas de construcción
3. Depurador (debugger)

IDE – Integrated Development Environment

Entorno de desarrollo integrado

Incluye un conjunto de herramientas informáticas para facilitar el desarrollo de software. Normalmente consiste en:

1. Editor de código fuente
2. Herramientas de construcción
3. Depurador (debugger)

Algunas características de los IDE

IDE – Integrated Development Environment

Entorno de desarrollo integrado

Incluye un conjunto de herramientas informáticas para facilitar el desarrollo de software. Normalmente consiste en:

1. Editor de código fuente
2. Herramientas de construcción
3. Depurador (debugger)

Algunas características de los IDE

- Resaltado de sintaxis

IDE – Integrated Development Environment

Entorno de desarrollo integrado

Incluye un conjunto de herramientas informáticas para facilitar el desarrollo de software. Normalmente consiste en:

1. Editor de código fuente
2. Herramientas de construcción
3. Depurador (debugger)

Algunas características de los IDE

- ▶ Resaltado de sintaxis
- ▶ Indentado automático

IDE – Integrated Development Environment

Entorno de desarrollo integrado

Incluye un conjunto de herramientas informáticas para facilitar el desarrollo de software. Normalmente consiste en:

1. Editor de código fuente
2. Herramientas de construcción
3. Depurador (debugger)

Algunas características de los IDE

- ▶ Resaltado de sintaxis
- ▶ Indentado automático
- ▶ Plegado de código

IDE – Integrated Development Environment

Entorno de desarrollo integrado

Incluye un conjunto de herramientas informáticas para facilitar el desarrollo de software. Normalmente consiste en:

1. Editor de código fuente
2. Herramientas de construcción
3. Depurador (debugger)

Algunas características de los IDE

- ▶ Resaltado de sintaxis
- ▶ Indentado automático
- ▶ Plegado de código
- ▶ Autocompletado

IDE – Integrated Development Environment

Entorno de desarrollo integrado

Incluye un conjunto de herramientas informáticas para facilitar el desarrollo de software. Normalmente consiste en:

1. Editor de código fuente
2. Herramientas de construcción
3. Depurador (debugger)

Algunas características de los IDE

- ▶ Resaltado de sintaxis
- ▶ Indentado automático
- ▶ Plegado de código
- ▶ Autocompletado
- ▶ Administración de proyecto

IDE – Integrated Development Environment

Entorno de desarrollo integrado

Incluye un conjunto de herramientas informáticas para facilitar el desarrollo de software. Normalmente consiste en:

1. Editor de código fuente
2. Herramientas de construcción
3. Depurador (debugger)

Algunas características de los IDE

- ▶ Resaltado de sintaxis
- ▶ Indentado automático
- ▶ Plegado de código
- ▶ Autocompletado
- ▶ Administración de proyecto
- ▶ Terminal embebida

IDE – Integrated Development Environment

Entorno de desarrollo integrado

Incluye un conjunto de herramientas informáticas para facilitar el desarrollo de software. Normalmente consiste en:

1. Editor de código fuente
2. Herramientas de construcción
3. Depurador (debugger)

Algunas características de los IDE

- ▶ Resaltado de sintaxis
- ▶ Indentado automático
- ▶ Plegado de código
- ▶ Autocompletado
- ▶ Administración de proyecto
- ▶ Terminal embebida
- ▶ Etc.

IDE – Integrated Development Environment

Entorno de desarrollo integrado

Incluye un conjunto de herramientas informáticas para facilitar el desarrollo de software. Normalmente consiste en:

1. Editor de código fuente
2. Herramientas de construcción
3. Depurador (debugger)

Algunas características de los IDE

- ▶ Resaltado de sintaxis
- ▶ Indentado automático
- ▶ Plegado de código
- ▶ Autocompletado
- ▶ Administración de proyecto
- ▶ Terminal embebida
- ▶ Etc.

¿Cuáles conocen?

IDE – Integrated Development Environment

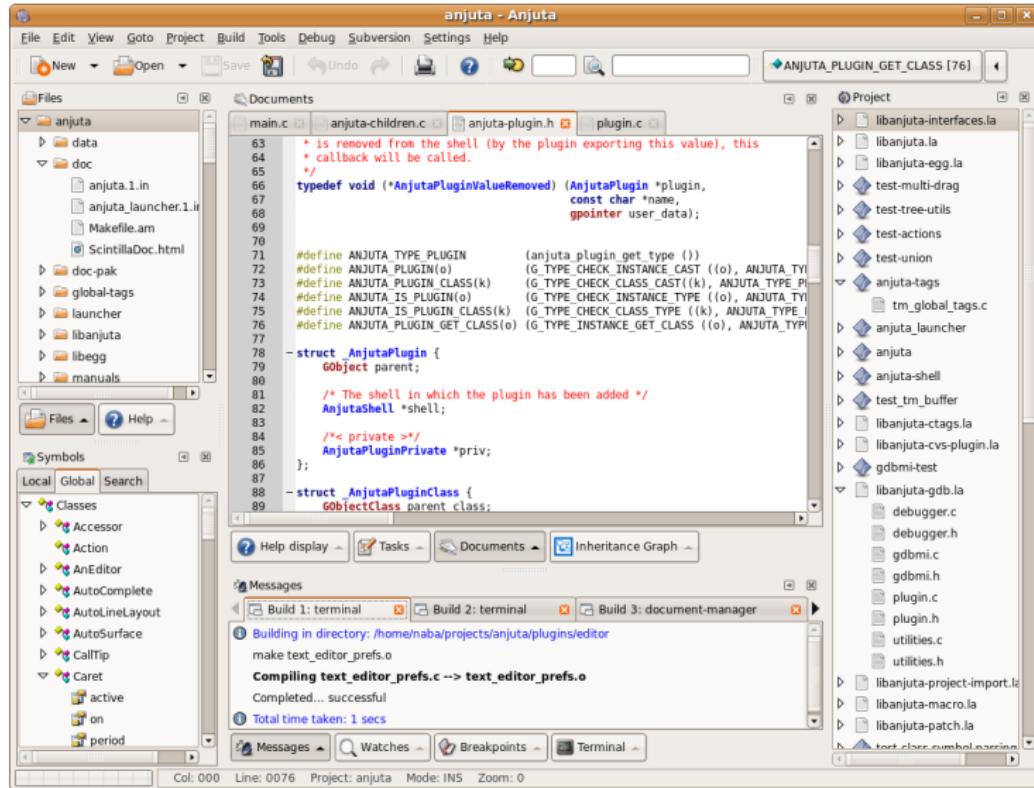
Geany

The screenshot shows the Geany IDE interface with the following details:

- Title Bar:** hello_world.c - /home/user - Geany
- Menu Bar:** File Edit Search View Document Project Build Tools Help
- Toolbar:** Includes icons for file operations like Open, Save, Print, and Build.
- Tab Bar:** Shows multiple tabs: document.c, document.h, editor.c, main.c, libmain.c, and hello_world.c (which is the active tab).
- Code Editor:** Displays the content of the hello_world.c file. The code includes a multi-line comment at the top and a main function that prints "Hello World".
- Status Bar:** Shows the status of the build process:
 - Status: gcc -Wall -o "hello_world" "hello_world.c" (in directory: /home/user)
 - Compiler: hello_world.c: In function 'main':
 - Messages: hello_world.c:27:17: error: 'i' undeclared (first use in this function)
 - Scribble:
 - Terminal: printf("%d\n", i);
 ^
 - Compilation failed.
- Bottom Status:** line: 23 / 32 col: 0 sel: 0 INS TAB mode: LF encoding: UTF-8 filetype: C scope: unknown

IDE – Integrated Development Environment

Anjuta



IDE – Integrated Development Environment

CodeLite

The screenshot shows the CodeLite IDE interface during a debugging session. The main window displays the code file `main.cpp`:

```
#include <iostream>
int main(int argc, char **argv)
{
    std::string str;
    str.append("Hello");
    str.append("\n");
    str.append("World");
    std::cout << str.c_str() << std::endl;
    return 0;
}
```

The cursor is at line 10, where the code is about to execute the opening brace of the `main` function. A red dot indicates a breakpoint has been set at this line. The status bar at the bottom left shows "Done".

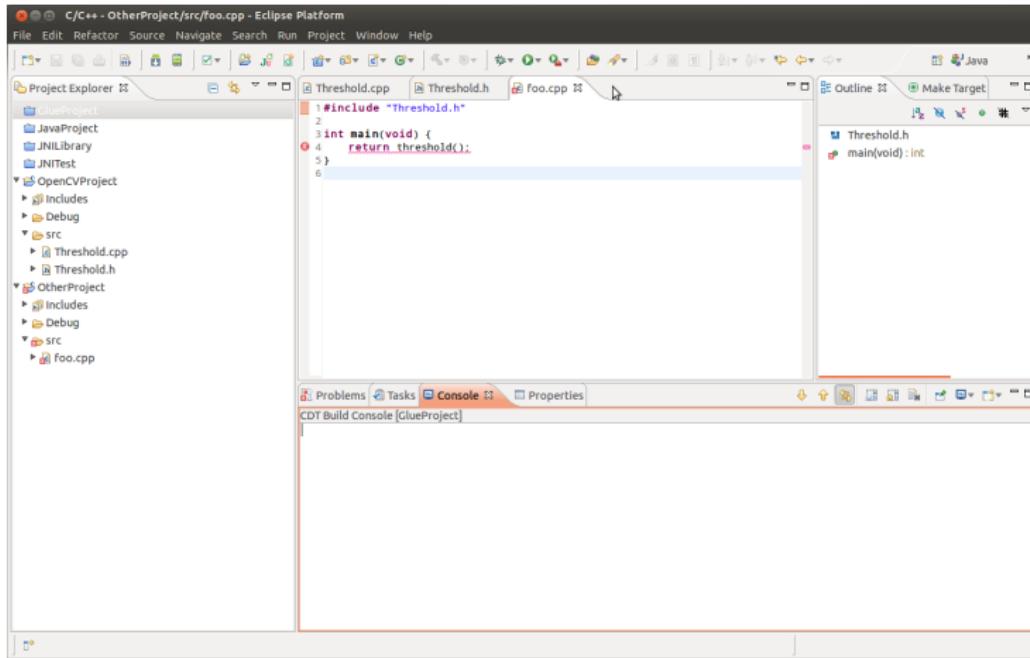
The right side of the interface contains several toolbars and panes:

- Breakpoints**: Shows one breakpoint at line 9 of `main.cpp`.
- Threads**: Shows a single thread with ID 24208, stopped at "step over" in the `main` function.
- Callstack**: Shows the call stack with two entries: `main` at line 0 and `??` at line 1.
- Locals**: Shows the local variables `argc`, `argv`, and `str`. `argc` is 1, `argv` is `0x00007fff5b8a0ab8`, and `str` is `"hello\nworld"`.

The status bar at the bottom right shows "Env: Default, Dbg: Default".

IDE – Integrated Development Environment

Eclipse CDT



IDE – Integrated Development Environment

Qt Creator

```
11 // the setup routine runs once when you press reset:
12 void setup() {
13     // initialize the digital pin as an output.
14     pinMode(led, OUTPUT);
15 }
16
17 // the loop routine runs over and over again forever:
18 void loop() {
19     digitalWrite(led, HIGH);
20     delay(1000);
21     digitalWrite(led, LOW);
22     delay(1000);
23 }
24
25
26
```

```
\FrameworkArduino\WString.o .pioenvs\arduino_uno\FrameworkArduino\abi.o .pioenvs\arduino_uno
\FrameworkArduino\main.o .pioenvs\arduino_uno\FrameworkArduino\new.o
avr-g++ -c .pioenvs\arduino_uno\firmware.elf -Os -mmcu=atmega328p -Wl,--gc-sections .pioenvs\arduino_uno
\src\pemain.o -L.pioenvs\arduino_uno -Wl,--start-group -lm .pioenvs\arduino_uno
/libFrameworkArduinoVariant.a .pioenvs\arduino_uno/libFrameworkArduino.a -Wl,--end-group
avr-objcopy -O ihex -R .eeprom .pioenvs\arduino_uno\firmware.elf .pioenvs\arduino_uno\firmware.hex
"avr-size" --mcu=atmega328p -C -d .pioenvs\arduino_uno\firmware.elf
AVR Memory Usage
=====
Device: atmega328p

Program:   1068 bytes (3.3% Full)
(.text + .data + .bootloader)

Data:        11 bytes (0.5% Full)
(.data + .bss + .noinit)

[SUCCESS] Took 3.76 seconds
11:33:53: The process "C:\Python27\Scripts\platformio.exe" exited normally.
11:33:53: Elapsed time: 00:06.
```

IDE – Integrated Development Environment

gedit

The screenshot shows the gedit integrated development environment. The main window displays a file named "repaso_c.c" containing the following C code:

```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

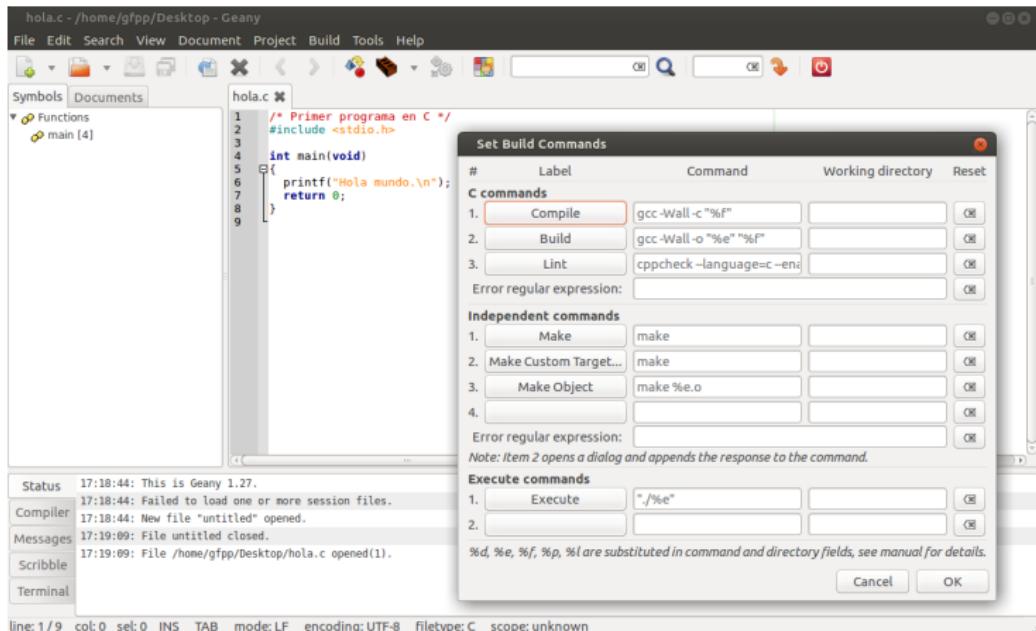
To the left of the code editor is a file browser sidebar showing several other C files and a shell script:

- build_suma.sh
- holo.c
- if_ifelse_error.c
- prog.c
- size_basic_types.c
- smile.c
- suma.c
- suma1.c
- suma2.c
- suma_arg.c

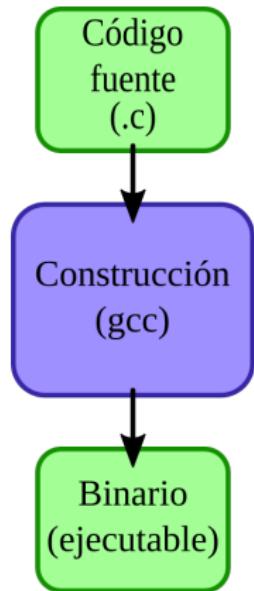
Below the code editor is a terminal window with the prompt "gfpp@gfpp-desktop:~\$".

At the bottom of the interface, there are status indicators: "C", "Tab Width: 8", "Ln 8, Col 3", and "INS".

Ejemplo de configuración del IDE Geany

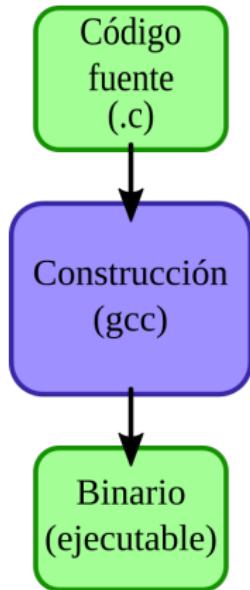


Construcción de un programa



Construcción de un programa

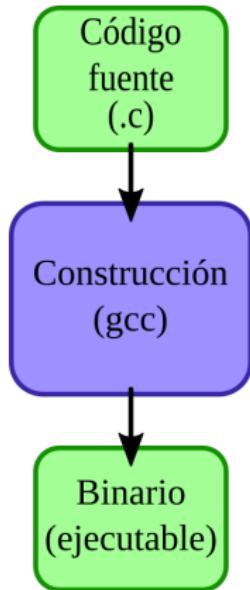
1. Escribir el código fuente



```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

Construcción de un programa

1. Escribir el código fuente

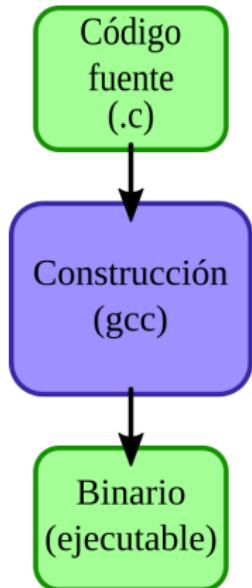


```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

2. Guardar con extensión .c (nombre sin espacios), p.e. hola.c

Construcción de un programa

1. Escribir el código fuente



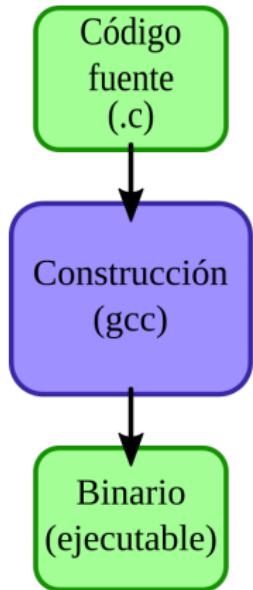
```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

2. Guardar con extensión .c (nombre sin espacios), p.e. hola.c
3. Construir el programa

```
> gcc hola.c
```

Construcción de un programa

1. Escribir el código fuente



```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

2. Guardar con extensión .c (nombre sin espacios), p.e. hola.c
3. Construir el programa

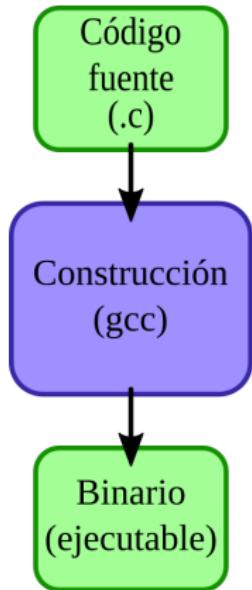
```
> gcc hola.c
```

4. Ejecutar el programa

```
> ./a.out
Hola mundo.
```

Construcción de un programa

1. Escribir el código fuente



```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

2. Guardar con extensión .c (nombre sin espacios), p.e. hola.c

3. Construir el programa

```
> gcc hola.c
```

4. Ejecutar el programa

```
> ./a.out
Hola mundo.
```

En la cátedra se utilizará este procedimiento de compilación

Introducción a gcc

hola.c

```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

Introducción a gcc

hola.c

```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

Compilación

```
> gcc hola.c
```

Introducción a gcc

hola.c

```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

Compilación

```
> gcc hola.c
```

(salida a.out)

Introducción a gcc

hola.c

```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

Compilación

```
> gcc hola.c
```

(salida a.out) Cómo se ejecuta? > ./a.out

Introducción a gcc

hola.c

```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

Compilación

```
> gcc hola.c
```

(salida a.out) Cómo se ejecuta? > ./a.out

Cómo cambiar el nombre a binario?

Introducción a gcc

hola.c

```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

Compilación

```
> gcc hola.c
```

(salida a.out) Cómo se ejecuta? > ./a.out

Cómo cambiar el nombre a binario?

```
> gcc hola.c -o hola
```

Introducción a gcc

hola.c

```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

Compilación

```
> gcc hola.c
```

(salida a.out) Cómo se ejecuta? > ./a.out

Cómo cambiar el nombre a binario?

```
> gcc hola.c -o hola
```

```
> gcc -Wall hola.c -o hola
```

Introducción a gcc

hola.c

```
1 /* Primer programa en C */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     printf("Hola mundo.\n");
7     return 0;
8 }
```

Compilación

```
> gcc hola.c
```

(salida a.out) Cómo se ejecuta? > ./a.out

Cómo cambiar el nombre a binario?

```
> gcc hola.c -o hola
```

```
> gcc -Wall hola.c -o hola
```

(habilita todas las advertencias/warnings)

Introducción a gcc

mal.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Dos y dos son %f\n", 4);
6     return 0;
7 }
```

Introducción a gcc

mal.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Dos y dos son %f\n", 4);
6     return 0;
7 }
```

¿Qué se imprime al ejecutar el programa?

Introducción a gcc

mal.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Dos y dos son %f\n", 4);
6     return 0;
7 }
```

¿Qué se imprime al ejecutar el programa?

Compilar y ejecutar

```
> gcc mal.c -o mal
> ./mal
Dos y dos son 0.000000
```

Introducción a gcc

mal.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Dos y dos son %f\n", 4);
6     return 0;
7 }
```

¿Qué se imprime al ejecutar el programa?

Compilar y ejecutar

```
> gcc mal.c -o mal
> ./mal
Dos y dos son 0.000000
```

Error de compilación (-Wall)

```
mal.c:5:10: warning: format '%f'
expects argument of type 'double',
but argument 2 has type 'int'
[-Wformat=]
```

Introducción a gcc

mal.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Dos y dos son %f\n", 4.); // CORREGIDO
6     return 0;
7 }
```

¿Qué se imprime al ejecutar el programa?

Compilar y ejecutar

```
> gcc mal.c -o mal
> ./mal
Dos y dos son 0.000000
```

Error de compilación (-Wall)

```
mal.c:5:10: warning: format '%f'
expects argument of type 'double',
but argument 2 has type 'int'
[-Wformat=]
```

Introducción a gcc

wall1.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int a = 2, b = 3;
6     printf("Dos más tres son %d\n", 5);
7     return 0;
8 }
```

Introducción a gcc

wall1.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int a = 2, b = 3;
6     printf("Dos más tres son %d\n", 5);
7     return 0;
8 }
```

wall2.c

```
1 int main(void)
2 {
3     int x;
4     x == 2 + 2;
5
6     x = 3;
7
8     if(x = 4)
9         x = 6;
10    return x;
11 }
```

Introducción a gcc

- ▶ Si el código fuente presenta errores de sintaxis el compilador (C o C++) reportará los errores y no producirá el archivo binario de salida.

Introducción a gcc

- ▶ Si el código fuente presenta errores de sintaxis el compilador (C o C++) reportará los errores y no producirá el archivo binario de salida.
- ▶ Los compiladores tienen la capacidad de advertirnos sobre situaciones inesperadas en el código fuente.

Introducción a gcc

- ▶ Si el código fuente presenta errores de sintaxis el compilador (C o C++) reportará los errores y no producirá el archivo binario de salida.
- ▶ Los compiladores tienen la capacidad de advertirnos sobre situaciones inesperadas en el código fuente.
- ▶ Se puede modificar este comportamiento utilizando *flags* de compilación.

Introducción a gcc

- ▶ Si el código fuente presenta errores de sintaxis el compilador (C o C++) reportará los errores y no producirá el archivo binario de salida.
- ▶ Los compiladores tiene la capacidad de advertirnos sobre situaciones inesperadas en el código fuente.
- ▶ Se puede modificar este comportamiento utilizando *flags* de compilación.
- ▶ El compilador puede considerarse como una herramienta útil para verificar el código fuente y aprender.

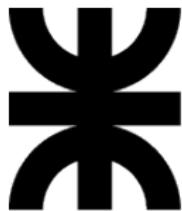
Introducción a gcc

- ▶ Si el código fuente presenta errores de sintaxis el compilador (C o C++) reportará los errores y no producirá el archivo binario de salida.
- ▶ Los compiladores tienen la capacidad de advertirnos sobre situaciones inesperadas en el código fuente.
- ▶ Se puede modificar este comportamiento utilizando *flags* de compilación.
- ▶ El compilador puede considerarse como una herramienta útil para verificar el código fuente y aprender.
- ▶ Los *flags* (banderas) del compilador permiten cambiar o alterar el comportamiento del compilador.

Informática II

Repaso del lenguaje C

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2021 –

Un poco de historia

Ideas provenientes de los lenguajes BCPL¹ (*Martin Richards, 1967*) y del lenguaje B (*Ken Thompson, 1970*).
Lenguajes “sin tipo”.



¹BCPL: Basic Combined Programming Language

Un poco de historia

Ideas provenientes de los lenguajes BCPL¹ (*Martin Richards, 1967*) y del lenguaje B (*Ken Thompson, 1970*).
Lenguajes “sin tipo”.

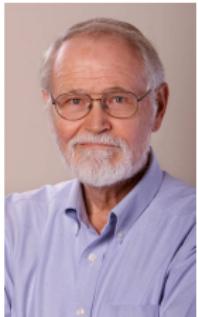


C fue diseñado originalmente en 1972 para el sistema operativo UNIX en el DEC PDP-11 por *Dennis Ritchie* en los laboratorios Bell.

¹BCPL: Basic Combined Programming Language

Un poco de historia

El primer libro de referencia fue *C Programming Language* (1978) de Brian Kernighan y Dennis Ritchie



Un poco de historia

El primer libro de referencia fue *C Programming Language* (1978) de Brian Kernighan y Dennis Ritchie



- ▶ En 1989 aparece el estándar ANSI C (ANSI¹)

¹American National Standards Institute

Un poco de historia

El primer libro de referencia fue *C Programming Language* (1978) de Brian Kernighan y Dennis Ritchie



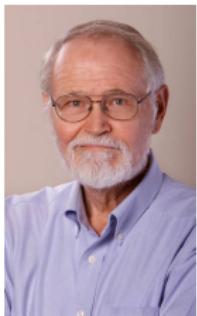
- ▶ En 1989 aparece el estándar ANSI C (ANSI¹)
- ▶ En 1990 aparece el estándar ISO C (ISO²)

¹American National Standards Institute

²International Standards Organization

Un poco de historia

El primer libro de referencia fue *C Programming Language* (1978) de *Brian Kernighan* y *Dennis Ritchie*



- ▶ En 1989 aparece el estándar ANSI C (ANSI¹)
- ▶ En 1990 aparece el estándar ISO C (ISO²)
- ▶ En 1999 aparece el estándar C99

¹American National Standards Institute

²International Standards Organization

Un poco de historia

El primer libro de referencia fue *C Programming Language* (1978) de *Brian Kernighan* y *Dennis Ritchie*



- ▶ En 1989 aparece el estándar ANSI C (ANSI¹)
- ▶ En 1990 aparece el estándar ISO C (ISO²)
- ▶ En 1999 aparece el estándar C99
- ▶ El último estándar publicado de C es el C11 (ISO/IEC 9899:2011) (IEC³)

¹American National Standards Institute

²International Standards Organization

³International Electrotechnical Commission

Algunas características

- ▶ Lenguaje de propósitos generales ampliamente utilizado

Algunas características

- ▶ Lenguaje de propósitos generales ampliamente utilizado
- ▶ Tiene características de lenguajes de **bajo nivel**

Algunas características

- ▶ Lenguaje de propósitos generales ampliamente utilizado
- ▶ Tiene características de lenguajes de **bajo nivel**
- ▶ Lenguaje relativamente pequeño: ofrece sentencias de control sencillas y funciones

Algunas características

- ▶ Lenguaje de propósitos generales ampliamente utilizado
- ▶ Tiene características de lenguajes de **bajo nivel**
- ▶ Lenguaje relativamente pequeño: ofrece sentencias de control sencillas y funciones
- ▶ Permite **programación estructurada** y diseño modular

Algunas características

- ▶ Lenguaje de propósitos generales ampliamente utilizado
- ▶ Tiene características de lenguajes de **bajo nivel**
- ▶ Lenguaje relativamente pequeño: ofrece sentencias de control sencillas y funciones
- ▶ Permite **programación estructurada** y diseño modular
- ▶ Si los programas siguen el estándar ISO el código es portátil entre plataformas y/o arquitecturas

Algunos inconvenientes

- ▶ No es un lenguaje **fuertemente tipado**. (ventaja o desventaja?)

Algunos inconvenientes

- ▶ No es un lenguaje **fuertemente tipado**. (ventaja o desventaja?)
- ▶ Bastante permisivo con la conversión de datos

Algunos inconvenientes

- ▶ No es un lenguaje **fuertemente tipado**. (ventaja o desventaja?)
- ▶ Bastante permisivo con la conversión de datos
- ▶ Su versatilidad permite crear programas difíciles de leer (código ofuscado)⁴

⁴IOCCC: The International Obfuscated C Code Contest

Introducción al lenguaje C

Programas en lenguaje C

Los programas C consisten de módulos o piezas que se denominan funciones. Esto facilita evitar volver a inventar la rueda: *reutilización de software*.

Introducción al lenguaje C

Programas en lenguaje C

Los programas C consisten de módulos o piezas que se denominan funciones. Esto facilita evitar volver a inventar la rueda: *reutilización de software*.

Aprender a programar en “C”

Consta de dos partes:

- ▶ Lenguaje C en sí mismo.
- ▶ Funciones de la biblioteca estándar C.

Introducción al lenguaje C

Programas en lenguaje C

Los programas C consisten de módulos o piezas que se denominan funciones. Esto facilita evitar volver a inventar la rueda: *reutilización de software*.

Aprender a programar en “C”

Consta de dos partes:

- ▶ Lenguaje C en sí mismo.
- ▶ Funciones de la biblioteca estándar C.

Todos los sistemas C consisten, en general, en tres partes:

- ▶ el entorno,
- ▶ el lenguaje, y
- ▶ la biblioteca estándar C.

Lenguaje C – Programas de ejemplo

¿Qué contiene en el sig. código fuente?

```
1  /*
2   * Ejemplo de programa
3   * en lenguaje C
4   */
5 #include <stdio.h>
6
7 int x = 0;
8
9 int main(void)
10 {
11     int x = 2;
12     {
13         int x = 5;
14     }
15     printf("x = %d\n", x);
16     return x;
17 }
```

Lenguaje C – Programas de ejemplo

¿Qué contiene en el sig. código fuente?

```
1  /*
2   * Ejemplo de programa
3   * en lenguaje C
4   */
5 #include <stdio.h>
6
7 int x = 0;
8
9 int main(void)
10 {
11     int x = 2;
12     {
13         int x = 5;
14     }
15     printf("x = %d\n", x);
16     return x;
17 }
```

x = 2

Lenguaje C – Programas de ejemplo

¿Qué contiene en el sig. código fuente?

```
1  /*
2   * Ejemplo de programa
3   * en lenguaje C
4   */
5 #include <stdio.h>
6
7 int x = 0;
8
9 int main(void)
10 {
11     int x = 2;
12     {
13         int x = 5;
14     }
15     printf("x = %d\n", x);
16     return x;
17 }
```

1. Comentarios (¿para qué?)

x = 2

Lenguaje C – Programas de ejemplo

¿Qué contiene en el sig. código fuente?

```
1  /*
2   * Ejemplo de programa
3   * en lenguaje C
4   */
5 #include <stdio.h>
6
7 int x = 0;
8
9 int main(void)
10 {
11     int x = 2;
12     {
13         int x = 5;
14     }
15     printf("x = %d\n", x);
16     return x;
17 }
```

1. Comentarios (¿para qué?)
2. Directiva del preprocesador (#)

x = 2

Lenguaje C – Programas de ejemplo

¿Qué contiene en el sig. código fuente?

```
1  /*
2   * Ejemplo de programa
3   * en lenguaje C
4   */
5 #include <stdio.h>
6
7 int x = 0;
8
9 int main(void)
10 {
11     int x = 2;
12     {
13         int x = 5;
14     }
15     printf("x = %d\n", x);
16     return x;
17 }
```

1. Comentarios (¿para qué?)
2. Directiva del preprocesador (#)
3. Archivos de cabecera/header (.h)

x = 2

Lenguaje C – Programas de ejemplo

¿Qué contiene en el sig. código fuente?

```
1  /*
2   * Ejemplo de programa
3   * en lenguaje C
4   */
5 #include <stdio.h>
6
7 int x = 0;
8
9 int main(void)
10 {
11     int x = 2;
12     {
13         int x = 5;
14     }
15     printf("x = %d\n", x);
16     return x;
17 }
```

1. Comentarios (¿para qué?)
2. Directiva del preprocesador (#)
3. Archivos de cabecera/header (.h)
4. Biblioteca estándar (stdin, stdout)

x = 2

Lenguaje C – Programas de ejemplo

¿Qué contiene en el sig. código fuente?

```
1  /*
2   * Ejemplo de programa
3   * en lenguaje C
4   */
5 #include <stdio.h>
6
7 int x = 0;
8
9 int main(void)
10 {
11     int x = 2;
12     {
13         int x = 5;
14     }
15     printf("x = %d\n", x);
16     return x;
17 }
```

1. Comentarios (¿para qué?)
2. Directiva del preprocesador (#)
3. Archivos de cabecera/header (.h)
4. Biblioteca estándar (stdin, stdout)
5. Función `main` (paréntesis)

x = 2

Lenguaje C – Programas de ejemplo

¿Qué contiene en el sig. código fuente?

```
1  /*
2   * Ejemplo de programa
3   * en lenguaje C
4   */
5 #include <stdio.h>
6
7 int x = 0;
8
9 int main(void)
10 {
11     int x = 2;
12     {
13         int x = 5;
14     }
15     printf("x = %d\n", x);
16     return x;
17 }
```

1. Comentarios (¿para qué?)
2. Directiva del preprocesador (#)
3. Archivos de cabecera/header (.h)
4. Biblioteca estándar (stdin, stdout)
5. Función main (paréntesis)
6. Parámetros y valor de retorno

x = 2

Lenguaje C – Programas de ejemplo

¿Qué contiene en el sig. código fuente?

```
1  /*
2   * Ejemplo de programa
3   * en lenguaje C
4   */
5 #include <stdio.h>
6
7 int x = 0;
8
9 int main(void)
10 {
11     int x = 2;
12     {
13         int x = 5;
14     }
15     printf("x = %d\n", x);
16     return x;
17 }
```

1. Comentarios (¿para qué?)
2. Directiva del preprocesador (#)
3. Archivos de cabecera/header (.h)
4. Biblioteca estándar (stdin, stdout)
5. Función main (paréntesis)
6. Parámetros y valor de retorno
7. Bloque (llaves) - Cuerpo de la función

x = 2

Lenguaje C – Programas de ejemplo

¿Qué contiene en el sig. código fuente?

```
1  /*
2   * Ejemplo de programa
3   * en lenguaje C
4   */
5 #include <stdio.h>
6
7 int x = 0;
8
9 int main(void)
10 {
11     int x = 2;
12     {
13         int x = 5;
14     }
15     printf("x = %d\n", x);
16     return x;
17 }
```

x = 2

1. Comentarios (¿para qué?)
2. Directiva del preprocesador (#)
3. Archivos de cabecera/header (.h)
4. Biblioteca estándar (stdin, stdout)
5. Función `main` (paréntesis)
6. Parámetros y valor de retorno
7. Bloque (llaves) - Cuerpo de la función
8. Parámetros y valor de devolución

Lenguaje C – Programas de ejemplo

¿Qué contiene en el sig. código fuente?

```
1  /*
2   * Ejemplo de programa
3   * en lenguaje C
4   */
5 #include <stdio.h>
6
7 int x = 0;
8
9 int main(void)
10 {
11     int x = 2;
12     {
13         int x = 5;
14     }
15     printf("x = %d\n", x);
16     return x;
17 }
```

x = 2

1. Comentarios (¿para qué?)
2. Directiva del preprocesador (#)
3. Archivos de cabecera/header (.h)
4. Biblioteca estándar (stdin, stdout)
5. Función `main` (paréntesis)
6. Parámetros y valor de retorno
7. Bloque (llaves) - Cuerpo de la función
8. Parámetros y valor de devolución
9. Enunciados (finaliza con ';')

Lenguaje C – Programas de ejemplo

¿Qué contiene en el sig. código fuente?

```
1  /*
2   * Ejemplo de programa
3   * en lenguaje C
4   */
5 #include <stdio.h>
6
7 int x = 0;
8
9 int main(void)
10 {
11     int x = 2;
12     {
13         int x = 5;
14     }
15     printf("x = %d\n", x);
16     return x;
17 }
```

x = 2

1. Comentarios (¿para qué?)
2. Directiva del preprocesador (#)
3. Archivos de cabecera/header (.h)
4. Biblioteca estándar (stdin, stdout)
5. Función `main` (paréntesis)
6. Parámetros y valor de retorno
7. Bloque (llaves) - Cuerpo de la función
8. Parámetros y valor de devolución
9. Enunciados (finaliza con ';')
10. Caracter de escape ('\')

Lenguaje C – Programas de ejemplo

¿Qué contiene en el sig. código fuente?

```
1  /*
2   * Ejemplo de programa
3   * en lenguaje C
4   */
5 #include <stdio.h>
6
7 int x = 0;
8
9 int main(void)
10 {
11     int x = 2;
12     {
13         int x = 5;
14     }
15     printf("x = %d\n", x);
16     return x;
17 }
```

x = 2

1. Comentarios (¿para qué?)
2. Directiva del preprocesador (#)
3. Archivos de cabecera/header (.h)
4. Biblioteca estándar (stdin, stdout)
5. Función `main` (paréntesis)
6. Parámetros y valor de retorno
7. Bloque (llaves) - Cuerpo de la función
8. Parámetros y valor de devolución
9. Enunciados (finaliza con ';')
10. Carácter de escape ('\')
11. Secuencia de escape ('\n')

Lenguaje C – Programas de ejemplo

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     /* Declaración de variables */
6     int entero1, entero2, suma;
7
8     printf("Ingrese el primer entero: ");
9     scanf("%d", &entero1);
10    printf("Ingrese el segundo entero: ");
11    scanf("%d", &entero2);
12
13    /* Asignación de la variable suma */
14    suma = entero1 + entero2;
15    printf("La suma es: %d\n", suma);
16
17    return 0; /* finaliza sin error */
18 }
```

Lenguaje C – Programas de ejemplo

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     /* Declaración de variables */
6     int entero1, entero2, suma;
7
8     printf("Ingrese el primer entero: ");
9     scanf("%d", &entero1);
10    printf("Ingrese el segundo entero: ");
11    scanf("%d", &entero2);
12
13    /* Asignación de la variable suma */
14    suma = entero1 + entero2;
15    printf("La suma es: %d\n", suma);
16
17    return 0; /* finaliza sin error */
18 }
```

```
Ingrese el primer entero: 34
Ingrese el segundo entero: 67
La suma es: 101
```

Lenguaje C – Programas de ejemplo

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     /* Declaración de variables */
6     int entero1, entero2, suma;
7
8     printf("Ingrese el primer entero: ");
9     scanf("%d", &entero1);
10    printf("Ingrese el segundo entero: ");
11    scanf("%d", &entero2);
12
13    /* Asignación de la variable suma */
14    suma = entero1 + entero2;
15    printf("La suma es: %d\n", suma);
16
17    return 0; /* finaliza sin error */
18 }
```

- ▶ Declaración de variables
 - ▶ ¿Dónde se declaran?
 - ▶ ¿Cuáles son los tipos de datos?

Lenguaje C – Programas de ejemplo

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     /* Declaración de variables */
6     int entero1, entero2, suma;
7
8     printf("Ingrese el primer entero: ");
9     scanf("%d", &entero1);
10    printf("Ingrese el segundo entero: ");
11    scanf("%d", &entero2);
12
13    /* Asignación de la variable suma */
14    suma = entero1 + entero2;
15    printf("La suma es: %d\n", suma);
16
17    return 0; /* finaliza sin error */
18 }
```

- ▶ Primer argumento de `printf`: cadena de control de formato
- ▶ `%d`: especificador de conversión
- ▶ `scanf`: cadena de control de formato

Lenguaje C – Programas de ejemplo

```
1 /* Suma de dos números enteros */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     /* Declaración de variables */
7     int entero1, entero2;
8
9     printf("Ingrese el primer entero: ");
10    scanf("%d", &entero1);
11    printf("Ingrese el segundo entero: ");
12    scanf("%d", &entero2);
13
14    /* Imprime el resultado */
15    printf("La suma es: %d\n", entero1 + entero2);
16
17    return 0; /* finaliza sin error */
18 }
```

¿Qué diferencias hay?

Variables

Cada variable tiene un *tipo*, un *nombre*, y un *valor*

Variables

Cada variable tiene un *tipo*, un *nombre*, y un *valor*

Nombre de variables

Un nombre de variable en C es cualquier **identificador** válido.

Un identificador es una serie de caracteres formados de letras, dígitos y subrayados (-) que no se inicie con un dígito.

C es sensible a las minúsculas y mayúsculas.

Variables

Cada variable tiene un *tipo*, un *nombre*, y un *valor*

Nombre de variables

Un nombre de variable en C es cualquier **identificador** válido.

Un identificador es una serie de caracteres formados de letras, dígitos y subrayados (-) que no se inicie con un dígito.

C es sensible a las minúsculas y mayúsculas.

Los **nombres de variables significativos** ayudan a auto-documentar el código.

Operadores

Operadores aritméticos (binarios)

+ - / * = % (módulo, solo con operandos enteros)

Operadores

Operadores aritméticos (binarios)

+ - / * = % (módulo, solo con operandos enteros)

Algunos comentarios:

- ¿Qué pasa con la división de enteros? (p.e. $17/5$, y $17 \% 5$)

Operadores

Operadores aritméticos (binarios)

+ - / * = % (módulo, solo con operandos enteros)

Algunos comentarios:

- ▶ ¿Qué pasa con la división de enteros? (p.e. $17/5$, y $17 \% 5$)
- ▶ Precedencia de operadores
 - ▶ $a * (b + c)$

Operadores

Operadores aritméticos (binarios)

+ - / * = % (módulo, solo con operandos enteros)

Algunos comentarios:

- ▶ ¿Qué pasa con la división de enteros? (p.e. $17/5$, y $17 \% 5$)
- ▶ Precedencia de operadores
 - ▶ $a * (b + c)$
 - ▶ $a1 * b1 + a2 * b2$

Operadores

Operadores aritméticos (binarios)

+ - / * = % (módulo, solo con operandos enteros)

Algunos comentarios:

- ▶ ¿Qué pasa con la división de enteros? (p.e. $17/5$, y $17 \% 5$)
- ▶ Precedencia de operadores
 - ▶ $a * (b + c)$
 - ▶ $a1 * b1 + a2 * b2$
 - ▶ $a0 + a1 * x + a2 * x * x$

Operadores

Operadores aritméticos (binarios)

+ - / * = % (módulo, solo con operandos enteros)

Algunos comentarios:

- ▶ ¿Qué pasa con la división de enteros? (p.e. $17/5$, y $17 \% 5$)
- ▶ Precedencia de operadores
 - ▶ $a * (b + c)$
 - ▶ $a1 * b1 + a2 * b2$
 - ▶ $a0 + a1 * x + a2 * x * x$

Regla de precedencia: primero (), luego *, /, %, y finalmente +, -.

Operadores

Operadores aritméticos (binarios)

+ - / * = % (módulo, solo con operandos enteros)

Algunos comentarios:

- ▶ ¿Qué pasa con la división de enteros? (p.e. $17/5$, y $17 \% 5$)
- ▶ Precedencia de operadores
 - ▶ $a * (b + c)$
 - ▶ $a1 * b1 + a2 * b2$
 - ▶ $a0 + a1 * x + a2 * x * x$

Regla de precedencia: primero (), luego *, /, %, y finalmente +, -.

Operadores de asignación

$+= -= *= /= \%=$

Palabras reservadas (32)

| | | | |
|----------|----------|----------|--------|
| auto | break | case | char |
| const | continue | default | do |
| double | else | enum | extern |
| float | for | goto | if |
| int | long | register | return |
| short | signed | sizeof | static |
| struct | switch | typedef | union |
| unsigned | void | volatile | while |

Palabras reservadas (32)

| | | | |
|---------------|----------|----------|--------|
| auto | break | case | char |
| const | continue | default | do |
| double | else | enum | extern |
| float | for | goto | if |
| int | long | register | return |
| short | signed | sizeof | static |
| struct | switch | typedef | union |
| unsigned | void | volatile | while |

Palabras reservadas (32)

| | | | |
|---------------|---------------|----------|--------|
| auto | break | case | char |
| const | continue | default | do |
| double | else | enum | extern |
| float | for | goto | if |
| int | long | register | return |
| short | signed | sizeof | static |
| struct | switch | typedef | union |
| unsigned | void | volatile | while |

Palabras reservadas (32)

| | | | |
|---------------|----------|----------|--------|
| auto | break | case | char |
| const | continue | default | do |
| double | else | enum | extern |
| float | for | goto | if |
| int | long | register | return |
| short | signed | sizeof | static |
| struct | switch | typedef | union |
| unsigned | void | volatile | while |

Palabras reservadas (32)

| | | | |
|----------|----------|----------|--------|
| auto | break | case | char |
| const | continue | default | do |
| double | else | enum | extern |
| float | for | goto | if |
| int | long | register | return |
| short | signed | sizeof | static |
| struct | switch | typedef | union |
| unsigned | void | volatile | while |

Palabras reservadas (32)

| | | | |
|----------|----------|----------|--------|
| auto | break | case | char |
| const | continue | default | do |
| double | else | enum | extern |
| float | for | goto | if |
| int | long | register | return |
| short | signed | sizeof | static |
| struct | switch | typedef | union |
| unsigned | void | volatile | while |

Palabras reservadas (32)

| | | | |
|----------|----------|----------|--------|
| auto | break | case | char |
| const | continue | default | do |
| double | else | enum | extern |
| float | for | goto | if |
| int | long | register | return |
| short | signed | sizeof | static |
| struct | switch | typedef | union |
| unsigned | void | volatile | while |

Palabras reservadas (32)

| | | | |
|----------|----------|----------|--------|
| auto | break | case | char |
| const | continue | default | do |
| double | else | enum | extern |
| float | for | goto | if |
| int | long | register | return |
| short | signed | sizeof | static |
| struct | switch | typedef | union |
| unsigned | void | volatile | while |

Palabras reservadas (32)

| | | | |
|----------|----------|----------|--------|
| auto | break | case | char |
| const | continue | default | do |
| double | else | enum | extern |
| float | for | goto | if |
| int | long | register | return |
| short | signed | sizeof | static |
| struct | switch | typedef | union |
| unsigned | void | volatile | while |

Programación estructurada

¿Qué es un algoritmo?

Programación estructurada

¿Qué es un algoritmo?

Un procedimiento para resolver un problema en términos de

- ▶ las acciones a ejecutarse, y
- ▶ el orden en el cual estas acciones deben de ejecutarse.

Programación estructurada

¿Qué es un algoritmo?

Un procedimiento para resolver un problema en términos de

- ▶ las acciones a ejecutarse, y
- ▶ el orden en el cual estas acciones deben de ejecutarse.

... ¿seudo-código?

Programación estructurada

¿Qué es un algoritmo?

Un procedimiento para resolver un problema en términos de

- ▶ las acciones a ejecutarse, y
- ▶ el orden en el cual estas acciones deben de ejecutarse.

... ¿seudo-código?

- ▶ Es un lenguaje artificial e informal que auxilia a los programadores a desarrollar los algoritmos.

Programación estructurada

¿Qué es un algoritmo?

Un procedimiento para resolver un problema en términos de

- ▶ las acciones a ejecutarse, y
- ▶ el orden en el cual estas acciones deben de ejecutarse.

... ¿seudo-código?

- ▶ Es un lenguaje artificial e informal que auxilia a los programadores a desarrollar los algoritmos.
- ▶ Ayudan al programador “*a pensar*” un programa antes de intentar escribirlo en un lenguaje de programación como C.

Programación estructurada

¿Qué es un algoritmo?

Un procedimiento para resolver un problema en términos de

- ▶ las acciones a ejecutarse, y
- ▶ el orden en el cual estas acciones deben de ejecutarse.

... ¿seudo-código?

- ▶ Es un lenguaje artificial e informal que auxilia a los programadores a desarrollar los algoritmos.
- ▶ Ayudan al programador “*a pensar*” un programa antes de intentar escribirlo en un lenguaje de programación como C.
- ▶ El seudo-código incluye solo enunciados ejecutables.

Programación estructurada

¿Qué es un algoritmo?

Un procedimiento para resolver un problema en términos de

- ▶ las acciones a ejecutarse, y
- ▶ el orden en el cual estas acciones deben de ejecutarse.

... ¿seudo-código?

- ▶ Es un lenguaje artificial e informal que auxilia a los programadores a desarrollar los algoritmos.
- ▶ Ayudan al programador “*a pensar*” un programa antes de intentar escribirlo en un lenguaje de programación como C.
- ▶ El seudo-código incluye solo enunciados ejecutables.

... ¿y un diagrama de flujo?

Programación estructurada

¿Qué es un algoritmo?

Un procedimiento para resolver un problema en términos de

- ▶ las acciones a ejecutarse, y
- ▶ el orden en el cual estas acciones deben de ejecutarse.

... ¿seudo-código?

- ▶ Es un lenguaje artificial e informal que auxilia a los programadores a desarrollar los algoritmos.
- ▶ Ayudan al programador “*a pensar*” un programa antes de intentar escribirlo en un lenguaje de programación como C.
- ▶ El seudo-código incluye solo enunciados ejecutables.

... ¿y un diagrama de flujo?

Un diagrama de flujo es una representación gráfica de un algoritmo o de una porción de un algoritmo.

Programación estructurada – Estructuras básicas

Teorema del programa estructurado (Böhm-Jacopini)

Programación estructurada – Estructuras básicas

Teorema del programa estructurado (Böhm-Jacopini)

Establece que toda función computable puede ser implementada en un lenguaje de programación que combine solo tres estructuras lógicas. Esas tres formas (también llamadas estructuras de control) son:

Programación estructurada – Estructuras básicas

Teorema del programa estructurado (Böhm-Jacopini)

Establece que toda función computable puede ser implementada en un lenguaje de programación que combine solo tres estructuras lógicas. Esas tres formas (también llamadas estructuras de control) son:

- ▶ *Secuencia*: ejecución de una instrucción tras otra.

Programación estructurada – Estructuras básicas

Teorema del programa estructurado (Böhm-Jacopini)

Establece que toda función computable puede ser implementada en un lenguaje de programación que combine solo tres estructuras lógicas. Esas tres formas (también llamadas estructuras de control) son:

- ▶ *Secuencia*: ejecución de una instrucción tras otra.
- ▶ *Selección*: ejecución de una de dos instrucciones (o conjuntos), según el valor de una variable booleana.

Programación estructurada – Estructuras básicas

Teorema del programa estructurado (Böhm-Jacopini)

Establece que toda función computable puede ser implementada en un lenguaje de programación que combine solo tres estructuras lógicas. Esas tres formas (también llamadas estructuras de control) son:

- ▶ *Secuencia*: ejecución de una instrucción tras otra.
- ▶ *Selección*: ejecución de una de dos instrucciones (o conjuntos), según el valor de una variable booleana.
- ▶ *Iteración*: ejecución de una instrucción (o conjunto) mientras una variable booleana sea verdadera. Se conoce como ciclo o bucle.

En el lenguaje C

- ▶ Secuencia?:

Programación estructurada – Estructuras básicas

Teorema del programa estructurado (Böhm-Jacopini)

Establece que toda función computable puede ser implementada en un lenguaje de programación que combine solo tres estructuras lógicas. Esas tres formas (también llamadas estructuras de control) son:

- ▶ *Secuencia*: ejecución de una instrucción tras otra.
- ▶ *Selección*: ejecución de una de dos instrucciones (o conjuntos), según el valor de una variable booleana.
- ▶ *Iteración*: ejecución de una instrucción (o conjunto) mientras una variable booleana sea verdadera. Se conoce como ciclo o bucle.

En el lenguaje C

- ▶ Secuencia?: Lenguaje secuencial.

Programación estructurada – Estructuras básicas

Teorema del programa estructurado (Böhm-Jacopini)

Establece que toda función computable puede ser implementada en un lenguaje de programación que combine solo tres estructuras lógicas. Esas tres formas (también llamadas estructuras de control) son:

- ▶ *Secuencia*: ejecución de una instrucción tras otra.
- ▶ *Selección*: ejecución de una de dos instrucciones (o conjuntos), según el valor de una variable booleana.
- ▶ *Iteración*: ejecución de una instrucción (o conjunto) mientras una variable booleana sea verdadera. Se conoce como ciclo o bucle.

En el lenguaje C

- ▶ Secuencia?: Lenguaje secuencial.
- ▶ Selección?:

Programación estructurada – Estructuras básicas

Teorema del programa estructurado (Böhm-Jacopini)

Establece que toda función computable puede ser implementada en un lenguaje de programación que combine solo tres estructuras lógicas. Esas tres formas (también llamadas estructuras de control) son:

- ▶ *Secuencia*: ejecución de una instrucción tras otra.
- ▶ *Selección*: ejecución de una de dos instrucciones (o conjuntos), según el valor de una variable booleana.
- ▶ *Iteración*: ejecución de una instrucción (o conjunto) mientras una variable booleana sea verdadera. Se conoce como ciclo o bucle.

En el lenguaje C

- ▶ Secuencia?: Lenguaje secuencial.
- ▶ Selección?: `if`, `if/else`, `switch`.

Programación estructurada – Estructuras básicas

Teorema del programa estructurado (Böhm-Jacopini)

Establece que toda función computable puede ser implementada en un lenguaje de programación que combine solo tres estructuras lógicas. Esas tres formas (también llamadas estructuras de control) son:

- ▶ *Secuencia*: ejecución de una instrucción tras otra.
- ▶ *Selección*: ejecución de una de dos instrucciones (o conjuntos), según el valor de una variable booleana.
- ▶ *Iteración*: ejecución de una instrucción (o conjunto) mientras una variable booleana sea verdadera. Se conoce como ciclo o bucle.

En el lenguaje C

- ▶ Secuencia?: Lenguaje secuencial.
- ▶ Selección?: `if`, `if/else`, `switch`.
- ▶ Iteración/repetición?:

Programación estructurada – Estructuras básicas

Teorema del programa estructurado (Böhm-Jacopini)

Establece que toda función computable puede ser implementada en un lenguaje de programación que combine solo tres estructuras lógicas. Esas tres formas (también llamadas estructuras de control) son:

- ▶ *Secuencia*: ejecución de una instrucción tras otra.
- ▶ *Selección*: ejecución de una de dos instrucciones (o conjuntos), según el valor de una variable booleana.
- ▶ *Iteración*: ejecución de una instrucción (o conjunto) mientras una variable booleana sea verdadera. Se conoce como ciclo o bucle.

En el lenguaje C

- ▶ Secuencia?: Lenguaje secuencial.
- ▶ Selección?: `if`, `if/else`, `switch`.
- ▶ Iteración/repetición?: `while`, `do/while`, `for`.

Programación estructurada – Estructuras básicas

Teorema del programa estructurado (Böhm-Jacopini)

Establece que toda función computable puede ser implementada en un lenguaje de programación que combine solo tres estructuras lógicas. Esas tres formas (también llamadas estructuras de control) son:

- ▶ *Secuencia*: ejecución de una instrucción tras otra.
- ▶ *Selección*: ejecución de una de dos instrucciones (o conjuntos), según el valor de una variable booleana.
- ▶ *Iteración*: ejecución de una instrucción (o conjunto) mientras una variable booleana sea verdadera. Se conoce como ciclo o bucle.

En el lenguaje C

- ▶ Secuencia?: Lenguaje secuencial.
- ▶ Selección?: `if`, `if/else`, `switch`. ¿Qué hacen?
- ▶ Iteración/repetición?: `while`, `do/while`, `for`. ¿Qué hacen?

Programación estructurada – Estructuras básicas

Teorema del programa estructurado (Böhm-Jacopini)

Establece que toda función computable puede ser implementada en un lenguaje de programación que combine solo tres estructuras lógicas. Esas tres formas (también llamadas estructuras de control) son:

- ▶ *Secuencia*: ejecución de una instrucción tras otra.
- ▶ *Selección*: ejecución de una de dos instrucciones (o conjuntos), según el valor de una variable booleana.
- ▶ *Iteración*: ejecución de una instrucción (o conjunto) mientras una variable booleana sea verdadera. Se conoce como ciclo o bucle.

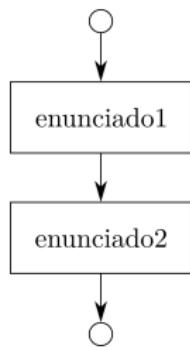
En el lenguaje C

- ▶ Secuencia?: Lenguaje secuencial.
- ▶ Selección?: `if`, `if/else`, `switch`. ¿Qué hacen?
- ▶ Iteración/repetición?: `while`, `do/while`, `for`. ¿Qué hacen?

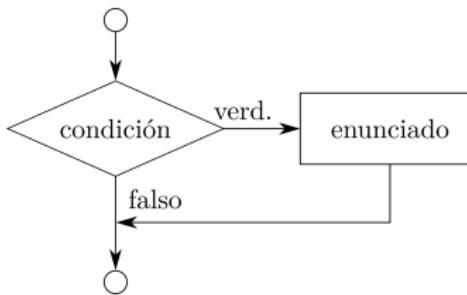
C tiene solo 7 estructuras de control

Programación estructurada – Estructuras básicas

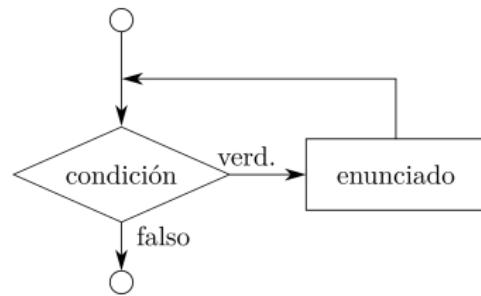
Estructura secuencial



Estructura de selección

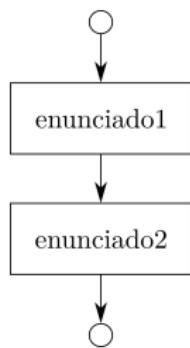


Estructura de repetición

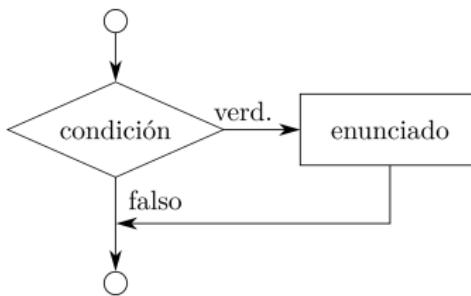


Programación estructurada – Estructuras básicas

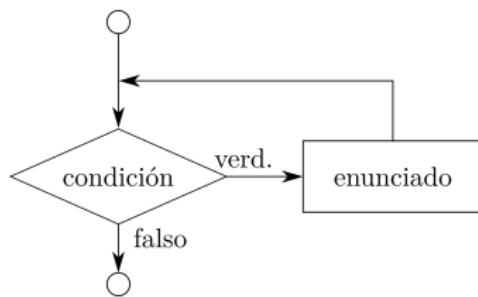
Estructura secuencial



Estructura de selección



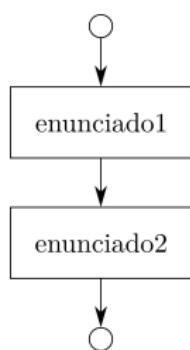
Estructura de repetición



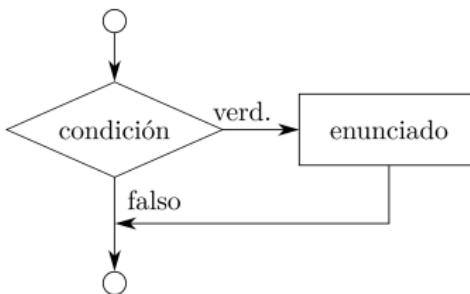
- ▶ Cada estructura tiene una sola entrada y una sola salida.

Programación estructurada – Estructuras básicas

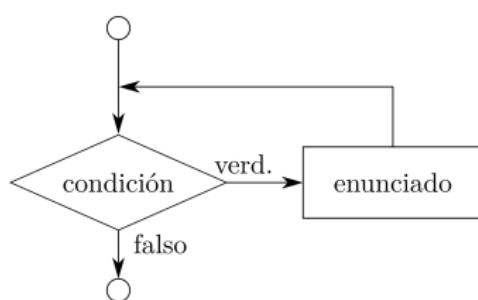
Estructura secuencial



Estructura de selección



Estructura de repetición



- ▶ Cada estructura tiene una sola entrada y una sola salida.
- ▶ Ellas se puede conectar mediante:
 - ▶ *apilamiento*
 - ▶ *anidamiento*.

Programación estructurada – Algunos operadores

[Condicionales] Operadores de igualdad

`== !=`

(menor nivel de precedencia)

[Condicionales] Operadores relacionales

`> < >= <=`

(mayor nivel de precedencia)

Programación estructurada – Algunos operadores

[Condicionales] Operadores de igualdad

`== !=`

(menor nivel de precedencia)

[Condicionales] Operadores relacionales

`> < >= <=`

(mayor nivel de precedencia)

Incremento y decremento

`++ --`

(pre/pos incremento/decremento)

Programación estructurada – Algunos operadores

[Condicionales] Operadores de igualdad

`== !=`

(menor nivel de precedencia)

[Condicionales] Operadores relacionales

`> < >= <=`

(mayor nivel de precedencia)

Incremento y decremento

`++ --`

(pre/pos incremento/decremento)

Operadores lógicos

- ▶ OR lógico: `||`
- ▶ AND lógico: `&&`
- ▶ NOT lógico: `!`

Estructuras de selección – estructura if

Seudo-código

Si calificación es mayor o igual a 60
Imprimir "Aprobó"

Código C

```
if(calificacion >= 60)
    printf("Aprobó \n");
```

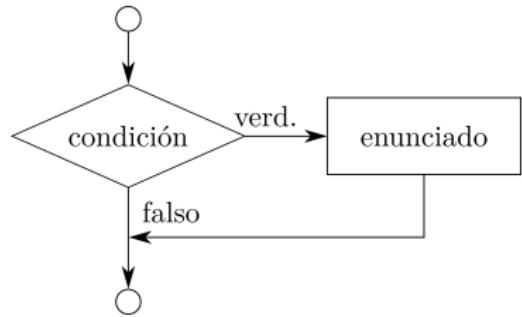
Estructuras de selección – estructura if

Seudo-código

```
Si calificación es mayor o igual a 60
    Imprimir "Aprobó"
```

Código C

```
if(calificacion >= 60)
    printf("Aprobó \n");
```



Estructuras de selección – estructura if/else

Seudo-código

Si calificación es mayor o igual a 60

 Imprimir "Aprobó"

Si no

 Imprimir "No aprobó"

Código C

```
if(calificacion >= 60)
    printf("Aprobó\n");
else
    printf("No aprobó\n");
```

Estructuras de selección – estructura if/else

Seudo-código

Si calificación es mayor o igual a 60

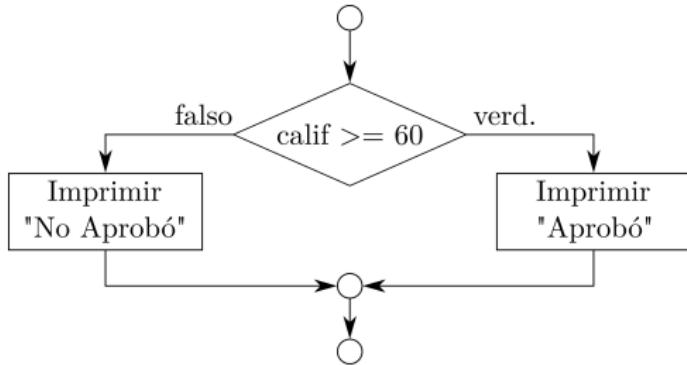
 Imprimir "Aprobó"

Si no

 Imprimir "No aprobó"

Código C

```
if(calificación >= 60)
    printf("Aprobó\n");
else
    printf("No aprobó\n");
```



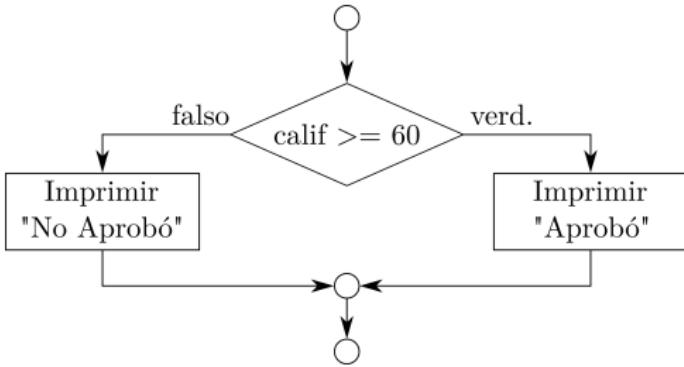
Estructuras de selección – estructura if/else

Seudo-código

```
Si calificación es mayor o igual a 60  
    Imprimir "Aprobó"  
Si no  
    Imprimir "No aprobó"
```

Código C

```
if(calificación >= 60)  
    printf("Aprobó\n");  
else  
    printf("No aprobó\n");
```



C tiene el operador condicional '?:' que está relacionado con la estructura if/else.

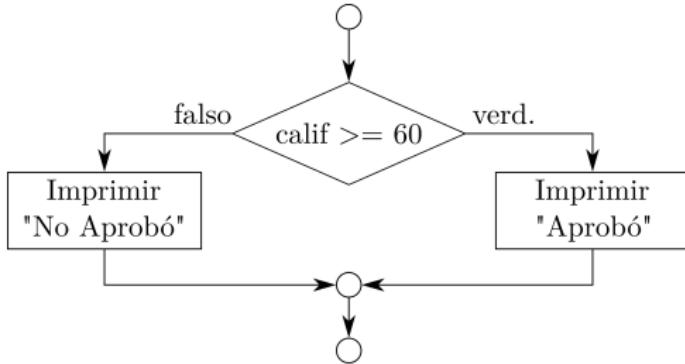
Estructuras de selección – estructura if/else

Seudo-código

```
Si calificación es mayor o igual a 60
    Imprimir "Aprobó"
Si no
    Imprimir "No aprobó"
```

Código C

```
if(calificación >= 60)
    printf("Aprobó\n");
else
    printf("No aprobó\n");
```



C tiene el operador condicional '?:' que está relacionado con la estructura if/else.

```
printf("%s\n", calificación >= 60 ? "Aprobó" : "No aprobó");
```

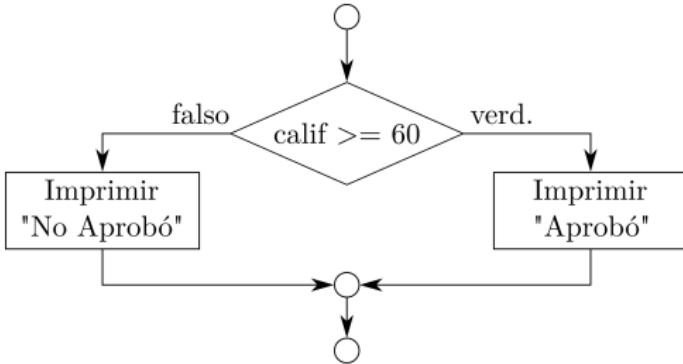
Estructuras de selección – estructura if/else

Seudo-código

```
Si calificación es mayor o igual a 60
    Imprimir "Aprobó"
Si no
    Imprimir "No aprobó"
```

Código C

```
if(calificación >= 60)
    printf("Aprobó\n");
else
    printf("No aprobó\n");
```



C tiene el operador condicional '?:' que está relacionado con la estructura if/else.

```
printf("%s\n", calificación >= 60 ? "Aprobó" : "No aprobó");
```

```
calificación >= 60 ? printf("Aprobó\n") : printf("No aprobó\n");
```

Estructuras de selección – estructura if/else

Enunciados compuestos:

- ▶ para incluir varios enunciados en el cuerpo de un `if` o `if/else`

Estructuras de selección – estructura if/else

Enunciados compuestos:

- ▶ para incluir varios enunciados en el cuerpo de un if o if/else

```
if(calificacion >= 60)
{
    printf("Aprobó\n");
    printf("Felicitaciones!! :) \n");
}
else
{
    printf("No aprobó\n");
    printf("Debes recursar :( \n");
}
```

Estructuras de selección – estructura if/else

Enunciados compuestos:

- ▶ para incluir varios enunciados en el cuerpo de un `if` o `if/else`

```
if(calificacion >= 60)
{
    printf("Aprobó\n");
    printf("Felicitaciones!! :) \n");
}
else
{
    printf("No aprobó\n");
    printf("Debes recursar :( \n");
}
```

Algunas preguntas:

- ▶ ¿Qué sucede si no estuvieran las llaves en el `else`?

Estructuras de selección – estructura if/else

Enunciados compuestos:

- ▶ para incluir varios enunciados en el cuerpo de un if o if/else

```
if(calificacion >= 60)
{
    printf("Aprobó\n");
    printf("Felicitaciones!! :) \n");
}
else
{
    printf("No aprobó\n");
    printf("Debes recursar :( \n");
}
```

Algunas preguntas:

- ▶ ¿Qué sucede si no estuvieran las llaves en el else?
- ▶ ¿Qué sucede si se coloca un punto y coma luego de un if? ... y el if/else?

Estructuras de selección – estructura if/else

Enunciados compuestos:

- ▶ para incluir varios enunciados en el cuerpo de un `if` o `if/else`

```
if(calificacion >= 60)
{
    printf("Aprobó\n");
    printf("Felicitaciones!! :) \n");
}
else
{
    printf("No aprobó\n");
    printf("Debes recursar :( \n");
}
```

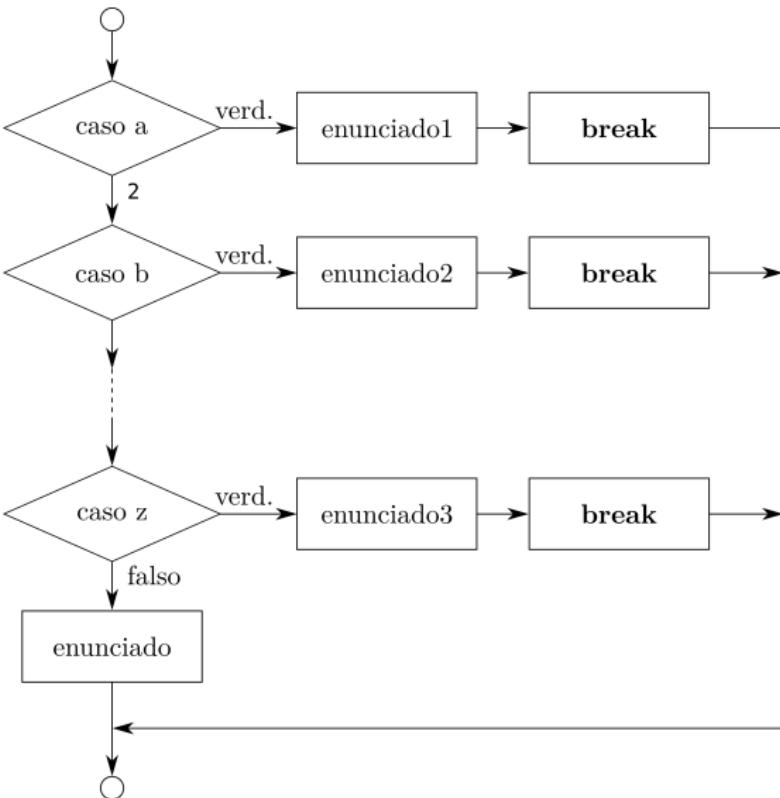
Algunas preguntas:

- ▶ ¿Qué sucede si no estuvieran las llaves en el `else`?
- ▶ ¿Qué sucede si se coloca un punto y coma luego de un `if`? ... y el `if/else`?
- ▶ ¿Cómo sería el diagrama de flujo de estructuras `if/else` anidadas?

Estructuras de selección – estructura switch

Código C

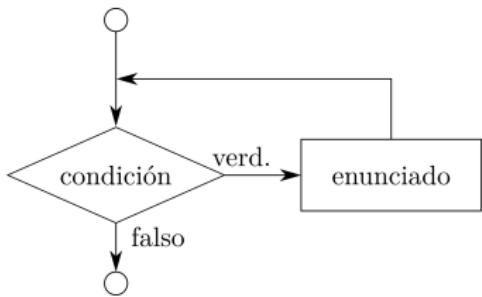
```
switch(caracter)
{
    case 'a':
        enunciado1;
        break;
    case 'b':
        enunciado2;
        break;
    case 'z':
        enunciado3;
        break;
    default:
        enunciado;
}
```



Estructuras de repetición – estructura while

Código C

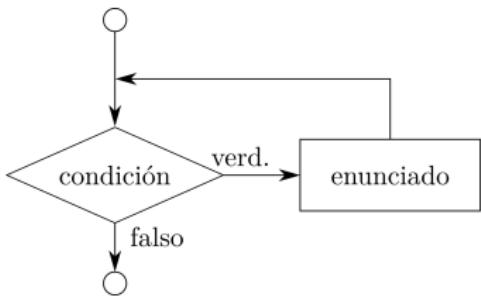
```
producto = 2;  
while(producto <= 1000)  
    producto = 2*producto;
```



Estructuras de repetición – estructura while

Código C

```
producto = 2;  
while(producto <= 1000)  
    producto = 2*producto;
```



- ▶ ¿Qué sucede si se coloca un punto y coma luego de un `while`?

Estructuras de repetición – estructuras do/while

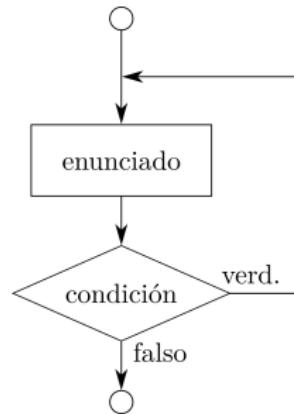
Código C

```
/* Imprime del 1 al 10
   utilizando do/while */
#include <stdio.h>

int main(void)
{
    int num = 1;

    do {
        printf("%d ", num);
    } while(++num <= 10);

    return 0;
}
```



Estructuras de repetición – estructuras do/while

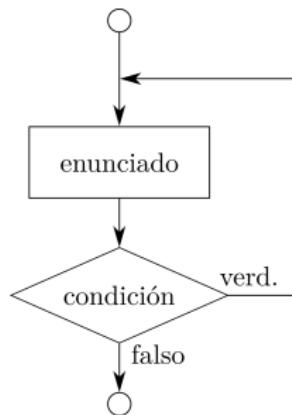
Código C

```
/* Imprime del 1 al 10
   utilizando do/while */
#include <stdio.h>

int main(void)
{
    int num = 1;

    do {
        printf("%d ", num);
    } while(++num <= 10);

    return 0;
}
```



Diferencia entre while y do/while

- En el **while** la condición de continuidad del ciclo se prueba al principio.
- En el **do/while** la condición de continuidad del ciclo se prueba luego de ejecutar el cuerpo.

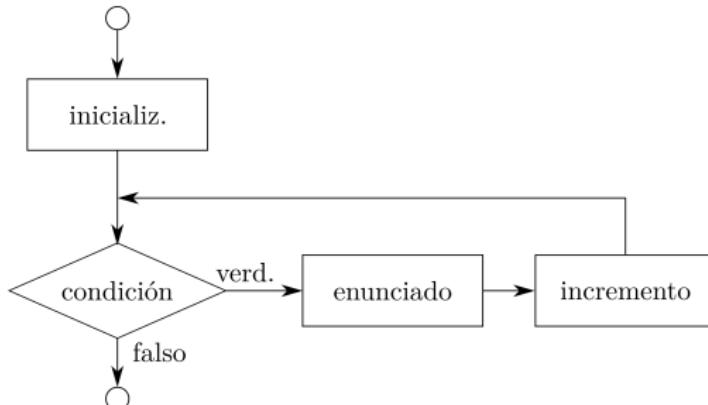
Estructuras de repetición – estructuras for

Código C

```
/* Imprime del 1 al 10 utilizando for */
#include <stdio.h>

int main(void) {
    int num;
    for(num = 1; num <= 10; num++)
        printf("%d ", num);

    return 0;
}
```



Estructuras de repetición – estructuras for

Maneja todas los detalles de la repetición controlada por contador (repetición definida).

Formato general de la estructura for

```
for(expresion1; expresion2; expresion3)
    enunciado;
```

Estructuras de repetición – estructuras for

Maneja todas los detalles de la repetición controlada por contador (repetición definida).

Formato general de la estructura for

```
for(expresion1; expresion2; expresion3)
    enunciado;
```

Equivalente con estructura while

```
expresion1;
while(expresion2) {
    enunciado;
    expresion3;
}
```

Estructuras de repetición – estructuras for

```
1 /* Sumatoria con estructura for */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     int sum = 0, num;
7
8     for(num = 2; num <= 100; num += 2)
9         sum += num;
10
11    printf("La suma es igual a: %d\n", sum);
12
13    return 0;
14 }
```

Estructuras de repetición – estructuras for

```
1 /* Sumatoria con estructura for */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     int sum = 0, num;
7
8     for(num = 2; num <= 100; num += 2)
9         sum += num;
10
11    printf("La suma es igual a: %d\n", sum);
12
13    return 0;
14 }
```

La suma es igual a: 2550

Estructuras de repetición – estructuras for

```
1 /* Sumatoria con estructura for */
2 #include <stdio.h>
3
4 int main(void)
5 {
6     int sum = 0, num;
7
8     for(num = 2; num <= 100; sum += num, num += 2)
9         ;
10
11    printf("La suma es igual a: %d\n", sum);
12
13    return 0;
14 }
```

La suma es igual a: 2550

Funciones – “funcionalización” de programas

Desarrollar y mantener un programa grande es más sencillo si está construido a partir de piezas menores o módulos, cada una más fácil de entender y manejar que el programa original

Funciones – “funcionalización” de programas

Desarrollar y mantener un programa grande es más sencillo si está construido a partir de piezas menores o módulos, cada una más fácil de entender y manejar que el programa original → Modularizar un programa.

Funciones – “funcionalización” de programas

Desarrollar y mantener un programa grande es más sencillo si está construido a partir de piezas menores o módulos, cada una más fácil de entender y manejar que el programa original → Modularizar un programa.

Los programas se escriben combinando:

1. funciones que el programador escribe
2. funciones de la *biblioteca estándar* de C

Funciones – “funcionalización” de programas

Desarrollar y mantener un programa grande es más sencillo si está construido a partir de piezas menores o módulos, cada una más fácil de entender y manejar que el programa original → Modularizar un programa.

Los programas se escriben combinando:

1. funciones que el programador escribe
2. funciones de la *biblioteca estándar* de C

Motivos para la funcionalización de un programa:

- ▶ Enfoque (recursivo) de divide y vencerás
- ▶ Reutilización de software
- ▶ Evitar la repetición de código en un programa

Funciones – “funcionalización” de programas

Desarrollar y mantener un programa grande es más sencillo si está construido a partir de piezas menores o módulos, cada una más fácil de entender y manejar que el programa original → Modularizar un programa.

Los programas se escriben combinando:

1. funciones que el programador escribe
2. funciones de la *biblioteca estándar* de C

Motivos para la funcionalización de un programa:

- ▶ Enfoque (recursivo) de divide y vencerás
- ▶ Reutilización de software
- ▶ Evitar la repetición de código en un programa

Las funciones se invocan mediante *llamadas a funciones*.

Funciones – “funcionalización” de programas

Desarrollar y mantener un programa grande es más sencillo si está construido a partir de piezas menores o módulos, cada una más fácil de entender y manejar que el programa original → Modularizar un programa.

Los programas se escriben combinando:

1. funciones que el programador escribe
2. funciones de la *biblioteca estándar* de C

Motivos para la funcionalización de un programa:

- ▶ Enfoque (recursivo) de divide y vencerás
- ▶ Reutilización de software
- ▶ Evitar la repetición de código en un programa

Las funciones se invocan mediante *llamadas a funciones*.

¿Cuál es el tamaño óptimo de una función?

Funciones – “funcionalización” de programas

Desarrollar y mantener un programa grande es más sencillo si está construido a partir de piezas menores o módulos, cada una más fácil de entender y manejar que el programa original → Modularizar un programa.

Los programas se escriben combinando:

1. funciones que el programador escribe
2. funciones de la *biblioteca estándar* de C

Motivos para la funcionalización de un programa:

- ▶ Enfoque (recursivo) de divide y vencerás
- ▶ Reutilización de software
- ▶ Evitar la repetición de código en un programa

Las funciones se invocan mediante *llamadas a funciones*.

¿Cuál es el tamaño óptimo de una función? ¿Y la cantidad de parámetros?

Funciones – Prototipo y declaración

Prototipo de función

```
tipo_de_valor_de_Regreso nombre_de_la_funcion(lista de parametros);
```

Funciones – Prototipo y declaración

Prototipo de función

```
tipo_de_valor_de_Regreso nombre_de_la_funcion(lista de parametros);
```

Formato de la declaración

```
tipo_de_valor_de_Regreso nombre_de_la_funcion(lista de parametros)
{
    definiciones
    instrucciones
}
```

Funciones – Prototipo y declaración

Prototipo de función

```
tipo_de_valor_de_Regreso nombre_de_la_funcion(lista de parametros);
```

Formato de la declaración

```
tipo_de_valor_de_Regreso nombre_de_la_funcion(lista de parametros)
{
    definiciones
    instrucciones
}
```

- ▶ El tipo de regreso por omisión es **int** (depende del estándar)

Funciones – Prototipo y declaración

Prototipo de función

```
tipo_de_valor_de_Regreso nombre_de_la_funcion(lista de parametros);
```

Formato de la declaración

```
tipo_de_valor_de_Regreso nombre_de_la_funcion(lista de parametros)
{
    definiciones
    instrucciones
}
```

- ▶ El tipo de regreso por omisión es **int** (depende del estándar)
- ▶ El tipo de regreso **void** significa que no regresa nada

Funciones – Prototipo y declaración

Prototipo de función

```
tipo_de_valor_de_Regreso nombre_de_la_funcion(lista de parametros);
```

Formato de la declaración

```
tipo_de_valor_de_Regreso nombre_de_la_funcion(lista de parametros)
{
    definiciones
    instrucciones
}
```

- ▶ El tipo de regreso por omisión es **int** (depende del estándar)
- ▶ El tipo de regreso **void** significa que no regresa nada
- ▶ Las variables declaradas en la definición son locales

Funciones – Parámetros

- ▶ La *lista de parámetros* se refiere al tipo, orden y cantidad de parámetros
- ▶ Parámetros:
 - Formales: parámetros en la declaración de la función
 - Verdaderos: parámetros que envía la función llamadora
- ▶ Los parámetros formales son variables locales a la función

Funciones – Parámetros

- ▶ La *lista de parámetros* se refiere al tipo, orden y cantidad de parámetros
- ▶ Parámetros:
 - Formales: parámetros en la declaración de la función
 - Verdaderos: parámetros que envía la función llamadora
- ▶ Los parámetros formales son variables locales a la función

Parámetro vs. argumento (K&R)

- ▶ Parámetro: declaración dentro de los paréntesis seguidos al nombre de la función
- ▶ Argumento: expresión dentro de los paréntesis en una llamada a función

Funciones – Parámetros

Llamada por valor y referencia

Por valor: Se hace una copia del valor del argumento. Al modificar la copia no se afecta el valor original.

Por referencia: Permite que la función modifique el valor original de la variable.

Funciones – Parámetros

Llamada por valor y referencia

Por valor: Se hace una copia del valor del argumento. Al modificar la copia no se afecta el valor original.

Por referencia: Permite que la función modifique el valor original de la variable.

En C todas las llamadas son llamadas por valor, la llamada por referencia se simular mediante la utilización de operadores de dirección y de indirección.

Funciones – Retorno de una función

- ▶ Si la función no regresa un valor, el control se devuelve a la función llamadora cuando:

Funciones – Retorno de una función

- ▶ Si la función no regresa un valor, el control se devuelve a la función llamadora cuando:
 - ▶ se alcanza la llave derecha que termina la función,

Funciones – Retorno de una función

- ▶ Si la función no regresa un valor, el control se devuelve a la función llamadora cuando:
 - ▶ se alcanza la llave derecha que termina la función,
 - ▶ o al ejecutar el enunciado
`return;`

Funciones – Retorno de una función

- ▶ Si la función no regresa un valor, el control se devuelve a la función llamadora cuando:
 - ▶ se alcanza la llave derecha que termina la función,
 - ▶ o al ejecutar el enunciado
`return;`
- ▶ Si la función regresa un valor, el enunciado
`return expresion;`
devuelve el valor *expresion* a la función llamadora.

Funciones – Pasaje de arreglos a funciones

Arreglos

- ▶ Estructura de datos de elementos relacionados del mismo tipo (vs. `struct`)

Funciones – Pasaje de arreglos a funciones

Arreglos

- ▶ Estructura de datos de elementos relacionados del mismo tipo (vs. `struct`)
- ▶ Posiciones de memoria del mismo nombre y mismo tipo.

Funciones – Pasaje de arreglos a funciones

Arreglos

- ▶ Estructura de datos de elementos relacionados del mismo tipo (vs. `struct`)
- ▶ Posiciones de memoria del mismo nombre y mismo tipo.
- ▶ Los arreglos ocupan espacio en memoria. Tipo y cantidad de elementos.

Funciones – Pasaje de arreglos a funciones

Arreglos

- ▶ Estructura de datos de elementos relacionados del mismo tipo (vs. `struct`)
- ▶ Posiciones de memoria del mismo nombre y mismo tipo.
- ▶ Los arreglos ocupan espacio en memoria. Tipo y cantidad de elementos.
- ▶ Los arreglos (y `struct`) son estructuras de datos estáticas.

Funciones – Pasaje de arreglos a funciones

Arreglos

- ▶ Estructura de datos de elementos relacionados del mismo tipo (vs. `struct`)
- ▶ Posiciones de memoria del mismo nombre y mismo tipo.
- ▶ Los arreglos ocupan espacio en memoria. Tipo y cantidad de elementos.
- ▶ Los arreglos (y `struct`) son estructuras de datos estáticas.
- ▶ `#define` para tamaño de arreglos (constante simbólica) → programas dimensionables.

Funciones – Pasaje de arreglos a funciones

Arreglos

- ▶ Estructura de datos de elementos relacionados del mismo tipo (vs. `struct`)
- ▶ Posiciones de memoria del mismo nombre y mismo tipo.
- ▶ Los arreglos ocupan espacio en memoria. Tipo y cantidad de elementos.
- ▶ Los arreglos (y `struct`) son estructuras de datos estáticas.
- ▶ `#define` para tamaño de arreglos (constante simbólica) → programas dimensionables.
- ▶ Arreglos de tipo `char` → cadenas de caracteres.

Funciones – Pasaje de arreglos a funciones

Arreglos

- ▶ Estructura de datos de elementos relacionados del mismo tipo (vs. `struct`)
- ▶ Posiciones de memoria del mismo nombre y mismo tipo.
- ▶ Los arreglos ocupan espacio en memoria. Tipo y cantidad de elementos.
- ▶ Los arreglos (y `struct`) son estructuras de datos estáticas.
- ▶ `#define` para tamaño de arreglos (constante simbólica) → programas dimensionables.
- ▶ Arreglos de tipo `char` → cadenas de caracteres.
- ▶ ¿Cómo se declara?

Funciones – Pasaje de arreglos a funciones

Arreglos

- ▶ Estructura de datos de elementos relacionados del mismo tipo (vs. `struct`)
- ▶ Posiciones de memoria del mismo nombre y mismo tipo.
- ▶ Los arreglos ocupan espacio en memoria. Tipo y cantidad de elementos.
- ▶ Los arreglos (y `struct`) son estructuras de datos estáticas.
- ▶ `#define` para tamaño de arreglos (constante simbólica) → programas dimensionables.
- ▶ Arreglos de tipo `char` → cadenas de caracteres.
- ▶ ¿Cómo se declara? ¿Cómo se inicializan?

Funciones – Pasaje de arreglos a funciones

Arreglos

- ▶ Estructura de datos de elementos relacionados del mismo tipo (vs. `struct`)
- ▶ Posiciones de memoria del mismo nombre y mismo tipo.
- ▶ Los arreglos ocupan espacio en memoria. Tipo y cantidad de elementos.
- ▶ Los arreglos (y `struct`) son estructuras de datos estáticas.
- ▶ `#define` para tamaño de arreglos (constante simbólica) → programas dimensionables.
- ▶ Arreglos de tipo `char` → cadenas de caracteres.
- ▶ ¿Cómo se declara? ¿Cómo se inicializan?

C pasa los arreglos a las funciones utilizando `llamada por referencia` de forma automática.

Funciones – Pasaje de arreglos a funciones

Función con arreglo

```
double promedio(float datos[], int tam)
{
    . . .
}
```

y con puntero

```
double promedio(float *datos, int tam)
{
    . . .
}
```

Funciones – Pasaje de arreglos a funciones

Función con arreglo

```
double promedio(float datos[], int tam)
{
    . . .
}
```

y con puntero

```
double promedio(float *datos, int tam)
{
    . . .
}
```

En la función se puede acceder a los elementos del arreglo utilizando la notación de arreglo o puntero (aritmética de puntero).

Funciones – Pasaje de arreglos a funciones

Función con arreglo

```
double promedio(float datos[], int tam)
{
    . . .
}
```

y con puntero

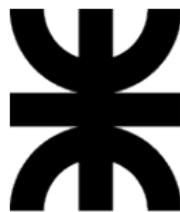
```
double promedio(float *datos, int tam)
{
    . . .
}
```

En la función se puede acceder a los elementos del arreglo utilizando la notación de arreglo o puntero (aritmética de puntero).

Informática II

Repaso de estructuras (**struct**)

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2021 –

Definición de estructuras (**struct**)

- ▶ Son colecciones de datos de variables relacionadas, todas bajo un mismo nombre, que pueden ser de diferentes tipos.

Definición de estructuras (**struct**)

- ▶ Son colecciones de datos de variables relacionadas, todas bajo un mismo nombre, que pueden ser de diferentes tipos.
- ▶ Son **tipos de datos derivados** construidos con objetos de otros tipos.

Definición de estructuras (**struct**)

- ▶ Son colecciones de datos de variables relacionadas, todas bajo un mismo nombre, que pueden ser de diferentes tipos.
- ▶ Son **tipos de datos derivados** construidos con objetos de otros tipos.

Definición:

```
struct horario {  
    int horas;  
    int minutos;  
    int segundos;  
};
```

Definición de estructuras (**struct**)

- ▶ Son colecciones de datos de variables relacionadas, todas bajo un mismo nombre, que pueden ser de diferentes tipos.
- ▶ Son **tipos de datos derivados** construidos con objetos de otros tipos.

Definición:

```
struct horario {  
    int horas;  
    int minutos;  
    int segundos;  
};
```

- ▶ Palabra reservada **struct** declara la estructura

Definición de estructuras (`struct`)

- ▶ Son colecciones de datos de variables relacionadas, todas bajo un mismo nombre, que pueden ser de diferentes tipos.
- ▶ Son **tipos de datos derivados** construidos con objetos de otros tipos.

Definición:

```
struct horario {  
    int horas;  
    int minutos;  
    int segundos;  
};
```

- ▶ Palabra reservada `struct` declara la estructura
- ▶ El identificador `horario` es el *rótulo* de la estructura

Definición de estructuras (`struct`)

- ▶ Son colecciones de datos de variables relacionadas, todas bajo un mismo nombre, que pueden ser de diferentes tipos.
- ▶ Son **tipos de datos derivados** construidos con objetos de otros tipos.

Definición:

```
struct horario {  
    int horas;  
    int minutos;  
    int segundos;  
};
```

- ▶ Palabra reservada `struct` declara la estructura
- ▶ El identificador `horario` es el *rótulo* de la estructura
- ▶ Se declara un nuevo tipo de datos (no reserva espacio en memoria) \Rightarrow `struct horario`

Definición de estructuras (`struct`)

- ▶ Son colecciones de datos de variables relacionadas, todas bajo un mismo nombre, que pueden ser de diferentes tipos.
- ▶ Son **tipos de datos derivados** construidos con objetos de otros tipos.

Definición:

```
struct horario {  
    int horas;  
    int minutos;  
    int segundos;  
};  
  
struct horario mediodia;
```

- ▶ Palabra reservada `struct` declara la estructura
- ▶ El identificador `horario` es el *rótulo* de la estructura
- ▶ Se declara un nuevo tipo de datos (no reserva espacio en memoria) \Rightarrow `struct horario`

Definición de estructuras (`struct`)

- ▶ Son colecciones de datos de variables relacionadas, todas bajo un mismo nombre, que pueden ser de diferentes tipos.
- ▶ Son **tipos de datos derivados** construidos con objetos de otros tipos.

Definición:

```
struct horario {  
    int horas;  
    int minutos;  
    int segundos;  
};  
  
struct horario mediodia;
```

- ▶ Palabra reservada `struct` declara la estructura
- ▶ El identificador `horario` es el *rótulo* de la estructura
- ▶ Se declara un nuevo tipo de datos (no reserva espacio en memoria) \Rightarrow `struct horario`
- ▶ Las variables dentro de las llaves son *miembros* de la estructura

Definición de estructuras (`struct`)

- ▶ Son colecciones de datos de variables relacionadas, todas bajo un mismo nombre, que pueden ser de diferentes tipos.
- ▶ Son **tipos de datos derivados** construidos con objetos de otros tipos.

Definición:

```
struct horario {  
    int horas;  
    int minutos;  
    int segundos;  
};
```

```
struct horario mediodia;
```

- ▶ Palabra reservada `struct` declara la estructura
- ▶ El identificador `horario` es el *rótulo* de la estructura
- ▶ Se declara un nuevo tipo de datos (no reserva espacio en memoria) \Rightarrow `struct horario`
- ▶ Las variables dentro de las llaves son *miembros* de la estructura
- ▶ Miembros de una estructura: variables de tipos básicos o derivados

Definición de estructuras (`struct`)

- ▶ Son colecciones de datos de variables relacionadas, todas bajo un mismo nombre, que pueden ser de diferentes tipos.
- ▶ Son **tipos de datos derivados** construidos con objetos de otros tipos.

Definición:

```
struct horario {  
    int horas;  
    int minutos;  
    int segundos;  
};
```

```
struct horario mediodia;
```

- ▶ Palabra reservada `struct` declara la estructura
- ▶ El identificador `horario` es el **rótulo** de la estructura
- ▶ Se declara un nuevo tipo de datos (no reserva espacio en memoria) \Rightarrow `struct horario`
- ▶ Las variables dentro de las llaves son **miembros** de la estructura
- ▶ Miembros de una estructura: variables de tipos básicos o derivados

Con estructuras y punteros (estructuras auto-referenciadas) se puede generar *estructuras dinámicas de datos* tales como: listas enlazadas, pilas, colas, árboles, etc.

Ejemplos

```
struct horario {  
    int horas;  
    int minutos;  
    int segundos;  
};  
struct horario hinicio, hfin;
```

Ejemplos

```
struct horario {  
    int horas;  
    int minutos;  
    int segundos;  
};  
struct horario hinicio, hfin;
```

```
struct horario {  
    int horas;  
    int minutos;  
    int segundos;  
} hinicio, hfin;
```

Ejemplos

```
struct horario {  
    int horas;  
    int minutos;  
    int segundos;  
};  
struct horario hinicio, hfin;
```

```
struct horario {  
    int horas;  
    int minutos;  
    int segundos;  
} hinicio, hfin;
```

El *rótulo* es opcional, si se omite se pueden definir variables de estructura solo dentro de la declaración.

```
struct {  
    int horas;  
    int minutos;  
    int segundos;  
} hinicio, hfin;
```

Ejemplos

```
struct horario {  
    int horas;  
    int minutos;  
    int segundos;  
};  
struct horario hinicio, hfin;
```

```
struct horario {  
    int horas;  
    int minutos;  
    int segundos;  
} hinicio, hfin;
```

El *rótulo* es opcional, si se omite se pueden definir variables de estructura solo dentro de la declaración.

```
struct {  
    int horas;  
    int minutos;  
    int segundos;  
} hinicio, hfin;
```

¿Qué otros ejemplos de estructuras se les ocurre?

Inicialización y acceso a miembros

```
struct paciente {  
    char apellido[20];  
    char nombre[20];  
    int edad;  
    float peso;  
    float altura;  
};
```

Inicialización y acceso a miembros

```
struct paciente {  
    char apellido[20];  
    char nombre[20];  
    int edad;  
    float peso;  
    float altura;  
};
```

¿Cómo se inicializa una estructura? (inicialización de entero: `int a = 2;`)

Inicialización y acceso a miembros

```
struct paciente {  
    char apellido[20];  
    char nombre[20];  
    int edad;  
    float peso;  
    float altura;  
};
```

¿Cómo se inicializa una estructura? (inicialización de entero: `int a = 2;`)

```
/* Inicialización de la estructura */  
struct paciente jperez = { "Perez", "Juan",  
    48, 88.5, 1.85 };
```

Inicialización y acceso a miembros

```
struct paciente {  
    char apellido[20];  
    char nombre[20];  
    int edad;  
    float peso;  
    float altura;  
};
```

¿Cómo se inicializa una estructura? (inicialización de entero: `int a = 2;`)

```
/* Inicialización de la estructura */  
struct paciente jperez = { "Perez", "Juan",  
    48, 88.5, 1.85 };  
  
/* Asignación de miembros */  
struct paciente cdiaz;  
cdiaz.edad = 39;  
strcpy(cdiaz.apellido, "Díaz");
```

Operaciones válidas

- ▶ Asignar variables de estructuras a otras del mismo tipo.

Operaciones válidas

- ▶ Asignar variables de estructuras a otras del mismo tipo.
- ▶ Tomar la dirección (`&`) de una variable de estructura.

Operaciones válidas

- ▶ Asignar variables de estructuras a otras del mismo tipo.
- ▶ Tomar la dirección (`&`) de una variable de estructura.
- ▶ Utilizar el operador `sizeof` para determinar el tamaño del tipo de datos estructura.

Operaciones válidas

- ▶ Asignar variables de estructuras a otras del mismo tipo.
- ▶ Tomar la dirección (`&`) de una variable de estructura.
- ▶ Utilizar el operador `sizeof` para determinar el tamaño del tipo de datos estructura.
- ▶ Acceder a los miembros de una estructura.

Operaciones válidas

- ▶ Asignar variables de estructuras a otras del mismo tipo.
- ▶ Tomar la dirección (`&`) de una variable de estructura.
- ▶ Utilizar el operador `sizeof` para determinar el tamaño del tipo de datos estructura.
- ▶ Acceder a los miembros de una estructura.

Ejemplos:

```
struct horario {  
    int horas;  
    int minutos;  
    int segundos;  
} hinicio, hfin;
```

Operaciones válidas

- ▶ Asignar variables de estructuras a otras del mismo tipo.
- ▶ Tomar la dirección (`&`) de una variable de estructura.
- ▶ Utilizar el operador `sizeof` para determinar el tamaño del tipo de datos estructura.
- ▶ Acceder a los miembros de una estructura.

Ejemplos:

```
struct horario {  
    int horas;  
    int minutos;  
    int segundos;  
} hinicio, hfin;  
  
hinicio.hora = 13;  
hinicio.minuto = 0;  
hinicio.segundo = 0;
```

Operaciones válidas

- ▶ Asignar variables de estructuras a otras del mismo tipo.
- ▶ Tomar la dirección (`&`) de una variable de estructura.
- ▶ Utilizar el operador `sizeof` para determinar el tamaño del tipo de datos estructura.
- ▶ Acceder a los miembros de una estructura.

Ejemplos:

```
struct horario {  
    int horas;  
    int minutos;  
    int segundos;  
} hinicio, hfin;  
  
hinicio.hora = 13;  
hinicio.minuto = 0;  
hinicio.segundo = 0;  
  
struct hora h = hinicio;
```

Operaciones válidas

- ▶ Asignar variables de estructuras a otras del mismo tipo.
- ▶ Tomar la dirección (`&`) de una variable de estructura.
- ▶ Utilizar el operador `sizeof` para determinar el tamaño del tipo de datos estructura.
- ▶ Acceder a los miembros de una estructura.

Ejemplos:

```
struct horario {  
    int horas;  
    int minutos;  
    int segundos;  
} hinicio, hfin;  
  
hinicio.hora = 13;  
hinicio.minuto = 0;  
hinicio.segundo = 0;  
  
struct hora h = hinicio;  
struct hora *hptr = &hinicio;
```

Operaciones válidas

- ▶ Asignar variables de estructuras a otras del mismo tipo.
- ▶ Tomar la dirección (`&`) de una variable de estructura.
- ▶ Utilizar el operador `sizeof` para determinar el tamaño del tipo de datos estructura.
- ▶ Acceder a los miembros de una estructura.

Ejemplos:

```
struct horario {  
    int horas;  
    int minutos;  
    int segundos;  
} hinicio, hfin;  
  
hinicio.hora = 13;  
hinicio.minuto = 0;  
hinicio.segundo = 0;  
  
struct hora h = hinicio;  
struct hora *hptr = &hinicio;  
size_t hstruct_size = sizeof(struct horario); /* sizeof(hfin) */
```

typedef con estructuras

La palabra reservada **typedef** permite crear alias para tipos de datos definidos anteriormente.

typedef con estructuras

La palabra reservada **typedef** permite crear alias para tipos de datos definidos anteriormente.

Ejemplos:

```
typedef long unsigned int size_t;
```

typedef con estructuras

La palabra reservada **typedef** permite crear alias para tipos de datos definidos anteriormente.

Ejemplos:

```
typedef long unsigned int size_t;
```

typedef en definición de estructuras

```
struct horario {  
    int horas;  
    int minutos;  
    int segundos;  
};  
typedef struct horario horario_t;
```

typedef con estructuras

La palabra reservada **typedef** permite crear alias para tipos de datos definidos anteriormente.

Ejemplos:

```
typedef long unsigned int size_t;
```

typedef en definición de estructuras

```
struct horario {  
    int horas;  
    int minutos;  
    int segundos;  
};  
typedef struct horario horario_t;  
  
horario_t hinicio, hfin;
```

typedef con estructuras

La palabra reservada **typedef** permite crear alias para tipos de datos definidos anteriormente.

Ejemplos:

```
typedef long unsigned int size_t;
```

typedef en definición de estructuras

```
struct horario {  
    int horas;  
    int minutos;  
    int segundos;  
};  
typedef struct horario horario_t;  
  
horario_t hinicio, hfin;
```

```
typedef struct {  
    int horas;  
    int minutos;  
    int segundos;  
} horario_t;
```

typedef con estructuras

La palabra reservada **typedef** permite crear alias para tipos de datos definidos anteriormente.

Ejemplos:

```
typedef long unsigned int size_t;
```

typedef en definición de estructuras

```
struct horario {  
    int horas;  
    int minutos;  
    int segundos;  
};  
typedef struct horario horario_t;  
  
horario_t hinicio, hfin;
```

```
typedef struct {  
    int horas;  
    int minutos;  
    int segundos;  
} horario_t;  
  
horario_t hinicio, hfin;
```

Arreglos y punteros

Arreglo de estructuras

```
struct paciente turno_tarde[10];
```

Arreglos y punteros

Arreglo de estructuras

```
struct paciente turno_tarde[10];
```

```
turno_tarde[2].peso = 78.5;
```

Arreglos y punteros

Arreglo de estructuras

```
struct paciente turno_tarde[10];
```

```
turno_tarde[2].peso = 78.5;
```

Puntero a estructuras

```
struct paciente *jperez;
```

```
(*jperez).peso = 78.5;
```

Arreglos y punteros

Arreglo de estructuras

```
struct paciente turno_tarde[10];
```

```
turno_tarde[2].peso = 78.5;
```

Puntero a estructuras

```
struct paciente *jperez;
```

```
(*jperez).peso = 78.5;  
jperez->altura = 1.76;
```

Arreglos y punteros

Arreglo de estructuras

```
struct paciente turno_tarde[10];
```

```
turno_tarde[2].peso = 78.5;
```

Puntero a estructuras

```
struct paciente *jperez;
```

```
(*jperez).peso = 78.5;  
jperez->altura = 1.76;
```

Operador flecha \rightarrow

Anidamiento de estructuras

```
typedef struct {
    int dia, mes, anio;
} fecha_t;
```

Anidamiento de estructuras

```
typedef struct {
    int dia, mes, anio;
} fecha_t;

typedef {
    char apellido[20];
    char nombre[20];
    int edad;
    float peso;
    float altura;
    fecha_t nacimiento;
} paciente_t;
```

Anidamiento de estructuras

```
typedef struct {
    int dia, mes, anio;
} fecha_t;

typedef {
    char apellido[20];
    char nombre[20];
    int edad;
    float peso;
    float altura;
    fecha_t nacimiento;
} paciente_t;

paciente_t jperez;
```

Anidamiento de estructuras

```
typedef struct {
    int dia, mes, anio;
} fecha_t;

typedef {
    char apellido[20];
    char nombre[20];
    int edad;
    float peso;
    float altura;
    fecha_t nacimiento;
} paciente_t;

paciente_t jperez;
jperez.nacimiento.anio = 1988;
```

Pasaje a funciones

- ▶ Las estructuras se pasan por valor a las funciones.

Pasaje a funciones

- ▶ Las estructuras se pasan por valor a las funciones.
- ▶ Se puede pasar un puntero de estructura a una función simulando llamada por referencia.

Pasaje a funciones

- ▶ Las estructuras se pasan por valor a las funciones.
- ▶ Se puede pasar un puntero de estructura a una función simulando llamada por referencia.
- ▶ Una forma (no ortodoxa) de pasar un arreglo a una función por valor es envolverlo en una estructura.

Informática II

Introducción al sistema operativo GNU/Linux

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2021 –

Sistemas operativos: historia y contexto

- ¿Qué es un sistema operativo?

Sistemas operativos: historia y contexto

- ▶ ¿Qué es un sistema operativo?
 - ▶ Programa o conjunto de programas para **administrar** los recursos de hardware y dar **servicios** a los programas de aplicación (software)

Sistemas operativos: historia y contexto

- ▶ ¿Qué es un sistema operativo?
 - ▶ Programa o conjunto de programas para **administrar** los recursos de hardware y dar **servicios** a los programas de aplicación (software)
 - ▶ Administración: tareas (scheduler), memoria, red, seguridad, disco.

Sistemas operativos: historia y contexto

- ▶ ¿Qué es un sistema operativo?
 - ▶ Programa o conjunto de programas para **administrar** los recursos de hardware y dar **servicios** a los programas de aplicación (software)
 - ▶ Administración: tareas (scheduler), memoria, red, seguridad, disco.
- ▶ ¿Cuales conocen?

Sistemas operativos: historia y contexto

- ▶ ¿Qué es un sistema operativo?
 - ▶ Programa o conjunto de programas para **administrar** los recursos de hardware y dar **servicios** a los programas de aplicación (software)
 - ▶ Administración: tareas (scheduler), memoria, red, seguridad, disco.
- ▶ ¿Cuales conocen?
 - ▶ Windows

Sistemas operativos: historia y contexto

- ▶ ¿Qué es un sistema operativo?
 - ▶ Programa o conjunto de programas para **administrar** los recursos de hardware y dar **servicios** a los programas de aplicación (software)
 - ▶ Administración: tareas (scheduler), memoria, red, seguridad, disco.
- ▶ ¿Cuales conocen?
 - ▶ Windows
 - ▶ Android

Sistemas operativos: historia y contexto

- ▶ ¿Qué es un sistema operativo?
 - ▶ Programa o conjunto de programas para **administrar** los recursos de hardware y dar **servicios** a los programas de aplicación (software)
 - ▶ Administración: tareas (scheduler), memoria, red, seguridad, disco.
- ▶ ¿Cuales conocen?
 - ▶ Windows
 - ▶ Android
 - ▶ Unix

Sistemas operativos: historia y contexto

- ▶ ¿Qué es un sistema operativo?
 - ▶ Programa o conjunto de programas para **administrar** los recursos de hardware y dar **servicios** a los programas de aplicación (software)
 - ▶ Administración: tareas (scheduler), memoria, red, seguridad, disco.
- ▶ ¿Cuales conocen?
 - ▶ Windows
 - ▶ Android
 - ▶ Unix
 - ▶ Mac OS

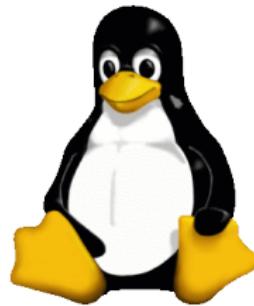
Sistemas operativos: historia y contexto

- ▶ ¿Qué es un sistema operativo?
 - ▶ Programa o conjunto de programas para **administrar** los recursos de hardware y dar **servicios** a los programas de aplicación (software)
 - ▶ Administración: tareas (scheduler), memoria, red, seguridad, disco.
- ▶ ¿Cuales conocen?
 - ▶ Windows
 - ▶ Android
 - ▶ Unix
 - ▶ Mac OS
 - ▶ BSD¹

¹Berkeley Software Distribution, desarrollado desde 1970 (Univ. de California)

Sistemas operativos: historia y contexto

- ▶ ¿Qué es un sistema operativo?
 - ▶ Programa o conjunto de programas para **administrar** los recursos de hardware y dar **servicios** a los programas de aplicación (software)
 - ▶ Administración: tareas (scheduler), memoria, red, seguridad, disco.
- ▶ ¿Cuales conocen?
 - ▶ Windows
 - ▶ Android
 - ▶ Unix
 - ▶ Mac OS
 - ▶ BSD¹
 - ▶ GNU/Linux²



¹Berkeley Software Distribution, desarrollado desde 1970 (Univ. de California)

²GNU: “GNU’s Not Unix” (clon de Unix), bajo licencia GPL

GNU/Linux

Richard Stallman

Richard Matthew Stallman (Manhattan, Nueva York; 16 de marzo de 1953), con frecuencia abreviado como «rms»,¹ es un programador estadounidense y fundador del movimiento del software libre, del sistema operativo GNU y de la Free Software Foundation (Fundación para el Software Libre).

Entre sus logros destacados como programador se incluye la realización del editor de texto GNU Emacs,² el compilador GCC,³ el depurador GDB,⁴ y el lenguaje de construcción GNU Make;⁵ todos bajo la rúbrica del Proyecto GNU. Sin embargo, es principalmente conocido por el establecimiento de un marco de referencia moral, político y legal para el software libre: un modelo de desarrollo y distribución alternativo al software privativo. Es también inventor del concepto de copyleft (aunque no del término): un método legal para licenciar obras contempladas por el derecho de autor, de tal forma que su uso y modificación (así como de sus derivados) permanezcan siempre permitidos.

Su innovador trabajo y activismo en torno al software libre y los derechos digitales le han merecido numerosas distinciones; incluyendo más de una docena de doctorados y profesorados honoríficos, la prestigiosa beca de la Fundación MacArthur, el premio Pioneer de la Electronic Frontier Foundation y varios premios de la ACM. Es miembro del salón de la fama de Internet.

Richard Stallman



Información personal

Nombre de nacimiento Richard Matthew Stallman [🔗](#)

Nacimiento 16 de marzo de 1953 [🔗](#) (68 años)
Manhattan (Estados Unidos)

Residencia Boston [🔗](#)

Richard Stallman inició el proyecto GNU en enero de 1984

Linus Torvalds liberó el código del Kernel de Linux en agosto de 1991 (PC 386)

Linus Torvalds

Linus Benedict Torvalds (Helsinki, Finlandia, 28 de diciembre de 1969¹) es un ingeniero de software finlandés-estadounidense,² conocido por iniciar y mantener el desarrollo del *kernel* (en español, núcleo) Linux, basándose en el sistema operativo Ibm Minix creado por Andrew S. Tanenbaum y en algunas herramientas, varias utilidades y los compiladores desarrollados por el proyecto GNU. Actualmente es responsable de la coordinación del proyecto.

- Índice [ocultar]
- 1 Biografía
 - 2 Creación de Linux
 - 3 Autoría y marca registrada
 - 4 Reconocimiento
 - 5 Véase también
 - 6 Referencias
 - 7 Enlaces externos

Biografía [editar]

Torvalds pertenece a la comunidad sueco-parlante de Finlandia. Sus padres tomaron su nombre de Linus Pauling (estadounidense, Premio Nobel de Química 1954). Comenzó sus andanzas informáticas a los 11 años cuando su abuelo, un matemático

Linus Torvalds



Torvalds en 2014

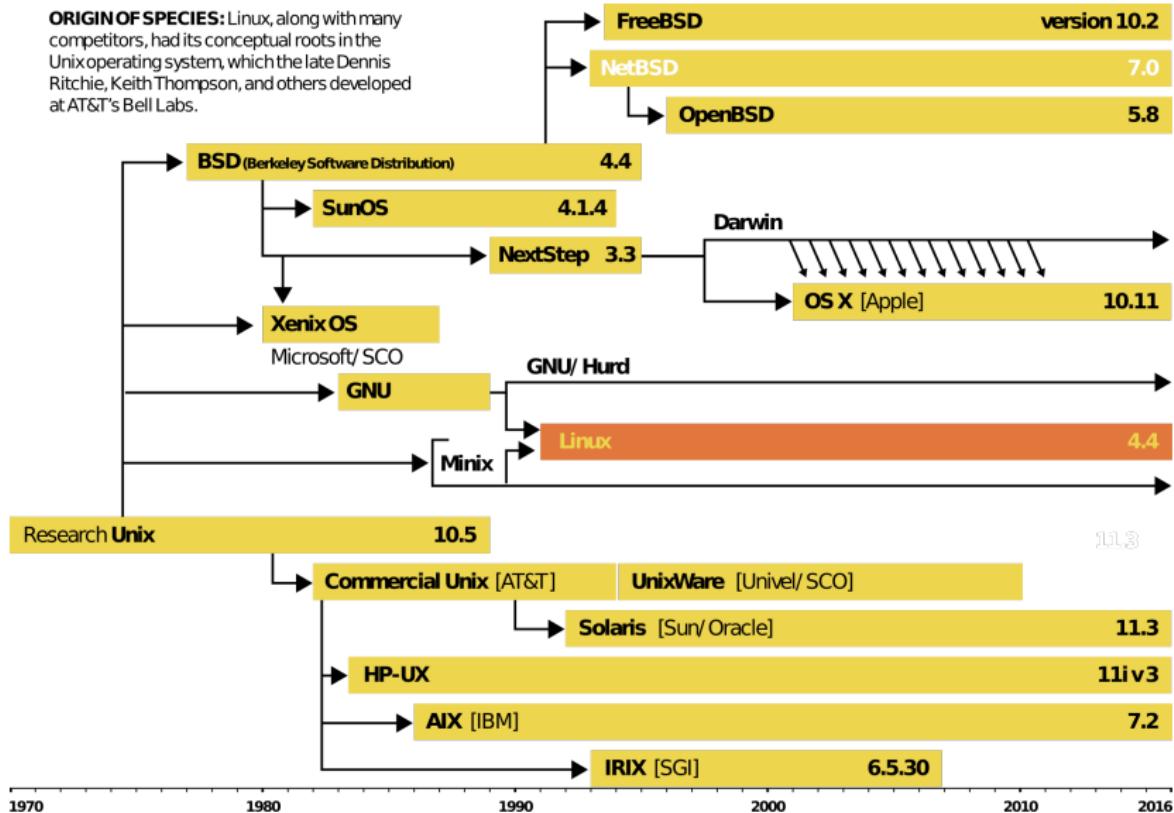
Información personal

Nombre de nacimiento Torvals Benedict Linus

Nacimiento 28 de diciembre de 1969

GNU/Linux

ORIGIN OF SPECIES: Linux, along with many competitors, had its conceptual roots in the Unix operating system, which the late Dennis Ritchie, Keith Thompson, and others developed at AT&T's Bell Labs.



From: torvalds@klaava.Helsinki.FI (Linus Benedict Torvalds)
Newsgroups: comp.os.minix
Subject: What would you like to see most in minix?
Summary: small poll for my new operating system
Message-ID:
Date: 25 Aug 91 20:57:08 GMT
Organization: University of Helsinki

Hello everybody out there using minix -

I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)

Linus (torvalds@kruuna.helsinki.fi)

PS. Yes - it's free of any minix code, and it has a multi-threaded fs. It is NOT protable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks, as that's all I have :-(.

GNU/Linux

- ▶ Linux 1.0 lanzado en 1994

GNU/Linux

- ▶ Linux 1.0 lanzado en 1994
- ▶ Desarrollado a partir de Minix (1987), usado por Linus (BSD no corría en PC, y GNU Hurd no estaba listo)

GNU/Linux

- ▶ Linux 1.0 lanzado en 1994
- ▶ Desarrollado a partir de Minix (1987), usado por Linus (BSD no corría en PC, y GNU Hurd no estaba listo)
- ▶ Sigue el estándar POXIS (Portable Operating System Interface for Unix): estándar para sistemas operativo tipo Unix (IEEE-CS)

GNU/Linux

- ▶ Linux 1.0 lanzado en 1994
- ▶ Desarrollado a partir de Minix (1987), usado por Linus (BSD no corría en PC, y GNU Hurd no estaba listo)
- ▶ Sigue el estándar POXIS (Portable Operating System Interface for Unix): estándar para sistemas operativo tipo Unix (IEEE-CS)
- ▶ Propiedades importantes de Linux
 1. Estabilidad
 2. Seguro
 3. No necesita rebuteo frecuente
 4. Portabilidad & Escalabilidad, etc.

GNU/Linux

- ▶ Linux 1.0 lanzado en 1994
- ▶ Desarrollado a partir de Minix (1987), usado por Linus (BSD no corría en PC, y GNU Hurd no estaba listo)
- ▶ Sigue el estándar POXIS (Portable Operating System Interface for Unix): estándar para sistemas operativo tipo Unix (IEEE-CS)
- ▶ Propiedades importantes de Linux
 1. Estabilidad
 2. Seguro
 3. No necesita rebuteo frecuente
 4. Portabilidad & Escalabilidad, etc.
- ▶ GNU (copyleft): Licenciado bajo GPL (General Public License)

GNU/Linux

- ▶ Linux 1.0 lanzado en 1994
- ▶ Desarrollado a partir de Minix (1987), usado por Linus (BSD no corría en PC, y GNU Hurd no estaba listo)
- ▶ Sigue el estándar POXIS (Portable Operating System Interface for Unix): estándar para sistemas operativo tipo Unix (IEEE-CS)
- ▶ Propiedades importantes de Linux
 1. Estabilidad
 2. Seguro
 3. No necesita rebuteo frecuente
 4. Portabilidad & Escalabilidad, etc.
- ▶ GNU (copyleft): Licenciado bajo GPL (General Public License)
- ▶ Distribuciones (Linux flavors)

Software libre – GNU/Linux

- ▶ RMS: en 1983 inicia oficialmente el proyecto GNU, en 1985 crea la Free Software Foundation, luego redacta la licencia GPL

Software libre – GNU/Linux

- ▶ RMS: en 1983 inicia oficialmente el proyecto GNU, en 1985 crea la Free Software Foundation, luego redacta la licencia GPL
- ▶ Software libre vs. privativo. Software libre vs. gratis

Software libre – GNU/Linux

- ▶ RMS: en 1983 inicia oficialmente el proyecto GNU, en 1985 crea la Free Software Foundation, luego redacta la licencia GPL
- ▶ Software libre vs. privativo. Software libre vs. gratis

Resumen GPL

- ▶ Cualquiera es libre de utilizar el Software Libre para cualquier propósito.

Software libre – GNU/Linux

- ▶ RMS: en 1983 inicia oficialmente el proyecto GNU, en 1985 crea la Free Software Foundation, luego redacta la licencia GPL
- ▶ Software libre vs. privativo. Software libre vs. gratis

Resumen GPL

- ▶ Cualquiera es libre de utilizar el Software Libre para cualquier propósito.
- ▶ Cualquiera es libre de acceder a su código fuente y estudiarlo.

Software libre – GNU/Linux

- ▶ RMS: en 1983 inicia oficialmente el proyecto GNU, en 1985 crea la Free Software Foundation, luego redacta la licencia GPL
- ▶ Software libre vs. privativo. Software libre vs. gratis

Resumen GPL

- ▶ Cualquiera es libre de utilizar el Software Libre para cualquier propósito.
- ▶ Cualquiera es libre de acceder a su código fuente y estudiarlo.
- ▶ Cualquiera es libre de distribuirlo.

Software libre – GNU/Linux

- ▶ RMS: en 1983 inicia oficialmente el proyecto GNU, en 1985 crea la Free Software Foundation, luego redacta la licencia GPL
- ▶ Software libre vs. privativo. Software libre vs. gratis

Resumen GPL

- ▶ Cualquiera es libre de utilizar el Software Libre para cualquier propósito.
- ▶ Cualquiera es libre de acceder a su código fuente y estudiarlo.
- ▶ Cualquiera es libre de distribuirlo.
- ▶ Cualquiera es libre de mejorarla o adaptarla y de distribuir el programa modificado.

Software libre – GNU/Linux

- ▶ RMS: en 1983 inicia oficialmente el proyecto GNU, en 1985 crea la Free Software Foundation, luego redacta la licencia GPL
- ▶ Software libre vs. privativo. Software libre vs. gratis

Resumen GPL

- ▶ Cualquiera es libre de utilizar el Software Libre para cualquier propósito.
- ▶ Cualquiera es libre de acceder a su código fuente y estudiarlo.
- ▶ Cualquiera es libre de distribuirlo.
- ▶ Cualquiera es libre de mejorarla o adaptarla y de distribuir el programa modificado.
- ▶ La única obligación es que si se distribuye, haya que hacerlo bajo la misma licencia GPL.

Software libre – GNU/Linux

- ▶ RMS: en 1983 inicia oficialmente el proyecto GNU, en 1985 crea la Free Software Foundation, luego redacta la licencia GPL
- ▶ Software libre vs. privativo. Software libre vs. gratis

Resumen GPL

- ▶ Cualquiera es libre de utilizar el Software Libre para cualquier propósito.
- ▶ Cualquiera es libre de acceder a su código fuente y estudiarlo.
- ▶ Cualquiera es libre de distribuirlo.
- ▶ Cualquiera es libre de mejorarla o adaptarla y de distribuir el programa modificado.
- ▶ La única obligación es que si se distribuye, haya que hacerlo bajo la misma licencia GPL.
- ▶ FOSS: Free and Open Source Software

Software libre – GNU/Linux

- ▶ RMS: en 1983 inicia oficialmente el proyecto GNU, en 1985 crea la Free Software Foundation, luego redacta la licencia GPL
- ▶ Software libre vs. privativo. Software libre vs. gratis

Resumen GPL

- ▶ Cualquiera es libre de utilizar el Software Libre para cualquier propósito.
- ▶ Cualquiera es libre de acceder a su código fuente y estudiarlo.
- ▶ Cualquiera es libre de distribuirlo.
- ▶ Cualquiera es libre de mejorarla o adaptarla y de distribuir el programa modificado.
- ▶ La única obligación es que si se distribuye, haya que hacerlo bajo la misma licencia GPL.
- ▶ FOSS: Free and Open Source Software
- ▶ OSHW: Open Source Hardware

Distribuciones – versiones/flavors

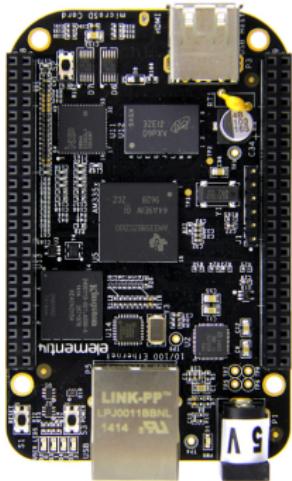
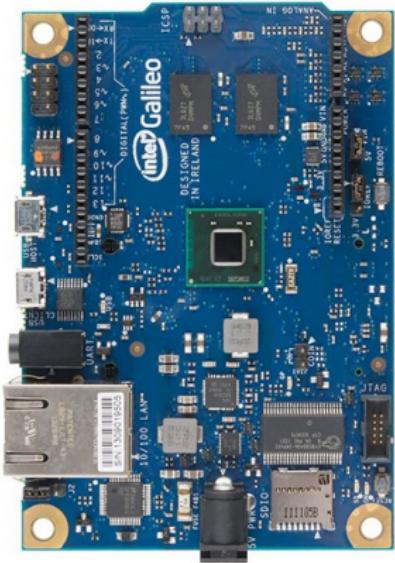


Distribuciones – versiones/flavors

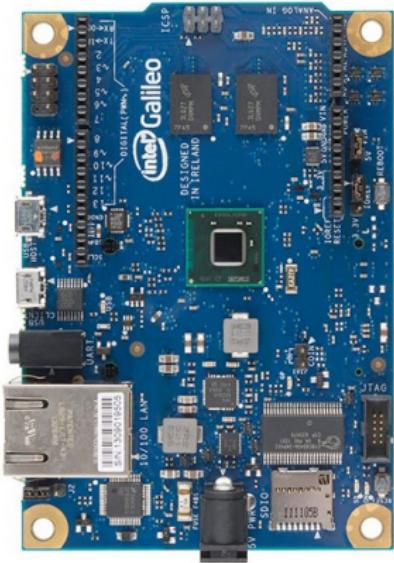


Linux Distribution Timeline

SBC, Single Board Computer



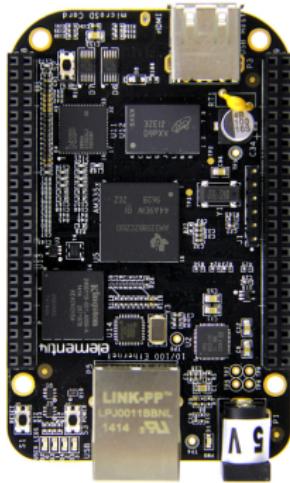
SBC, Single Board Computer



Intel Galileo



RaspberryPi 3



Beaglebone Black

SO GNU/Linux – Componentes

Componentes de un sistema GNU/Linux:

1. El kernel/núcleo (Linux)

SO GNU/Linux – Componentes

Componentes de un sistema GNU/Linux:

1. El kernel/núcleo (Linux)
2. El shell o intérprete de comandos
(`bash`, `ash`, `csh`, etc.)

SO GNU/Linux – Componentes

Componentes de un sistema GNU/Linux:

1. El kernel/núcleo (Linux)
2. El shell o intérprete de comandos
(`bash`, `ash`, `csh`, etc.)
3. Procesos y archivos/sistema de
archivos (File System)

SO GNU/Linux – Componentes

Componentes de un sistema GNU/Linux:

1. El kernel/núcleo (Linux)
2. El shell o intérprete de comandos
(`bash`, `ash`, `csh`, etc.)
3. Procesos y archivos/sistema de
archivos (File System)
4. Entorno de escritorio

SO GNU/Linux – Componentes

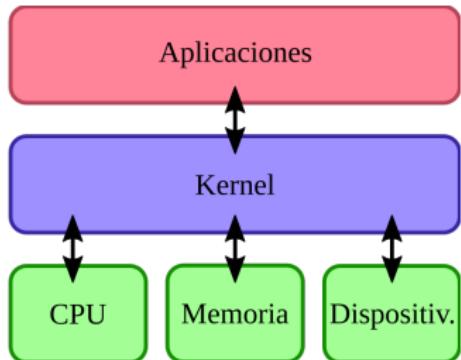
Componentes de un sistema GNU/Linux:

1. El kernel/núcleo (Linux)
2. El shell o intérprete de comandos
(`bash`, `ash`, `csh`, etc.)
3. Procesos y archivos/sistema de
archivos (File System)
4. Entorno de escritorio
5. Aplicaciones

SO GNU/Linux – Componentes

Componentes de un sistema GNU/Linux:

1. El kernel/núcleo (Linux)
2. El shell o intérprete de comandos (bash, ash, csh, etc.)
3. Procesos y archivos/sistema de archivos (File System)
4. Entorno de escritorio
5. Aplicaciones



Kernel de Linux

- ▶ El Kernel se comunica con el hardware

Kernel de Linux

- ▶ El Kernel se comunica con el hardware
- ▶ No tiene ninguna vinculación con el usuario

Kernel de Linux

- ▶ El Kernel se comunica con el hardware
- ▶ No tiene ninguna vinculación con el usuario
- ▶ Coordina las funciones internas y administra los recursos del sistema

Kernel de Linux

- ▶ El Kernel se comunica con el hardware
- ▶ No tiene ninguna vinculación con el usuario
- ▶ Coordina las funciones internas y administra los recursos del sistema

Funciones principales del Kernel

- ▶ Administración de procesos mediante la planificación del tiempo de corrida de cada uno y los privilegios que tienen (administración del microprocesador y memoria)

Kernel de Linux

- ▶ El Kernel se comunica con el hardware
- ▶ No tiene ninguna vinculación con el usuario
- ▶ Coordina las funciones internas y administra los recursos del sistema

Funciones principales del Kernel

- ▶ Administración de procesos mediante la planificación del tiempo de corrida de cada uno y los privilegios que tienen (administración del microprocesador y memoria)
- ▶ Administración el uso de dispositivos de hardware a través de los controladores (drivers) necesarios para su funcionamiento

Kernel de Linux

Está escrito en lenguaje C (`gcc`) con algunas pequeñas secciones de código en ensamblador.

Kernel de Linux

Está escrito en lenguaje C (`gcc`) con algunas pequeñas secciones de código en ensamblador.

Algunas arquitecturas de μ P/ μ C soportadas:

- ▶ x86: IA-32 y x86-64
- ▶ ARM: Freescale i.MX, gumstix, Qualcomm Snapdragon
- ▶ MISP: PlayStation2, Broadcom Wireless
- ▶ PowerPC: PlayStation3, Nintendo
- ▶ SPARC
- ▶ RISC-V
- ▶ etc.

Kernel de Linux – versión

Identificación para la versión del Kernel

1. Antes de la versión 1.0

Kernel de Linux – versión

Identificación para la versión del Kernel

1. Antes de la versión 1.0

- ▶ Primer versión: 0.01, seguido por la 0.02, 0.03, 0.10, 0.11, y 0.12 (primer versión GPL), etc. hasta la versión 1.0

Kernel de Linux – versión

Identificación para la versión del Kernel

1. Antes de la versión 1.0

- ▶ Primer versión: 0.01, seguido por la 0.02, 0.03, 0.10, 0.11, y 0.12 (primer versión GPL), etc. hasta la versión 1.0

2. Luego de la versión 1.0 y antes de las 2.6:

Kernel de Linux – versión

Identificación para la versión del Kernel

1. Antes de la versión 1.0

- ▶ Primer versión: 0.01, seguido por la 0.02, 0.03, 0.10, 0.11, y 0.12 (primer versión GPL), etc. hasta la versión 1.0

2. Luego de la versión 1.0 y antes de las 2.6:

- ▶ La numeración tenía la forma **a.b.c**, donde: **a** era la versión del Kernel, **b** eran las revisiones principales y **c** las revisiones menores.

Kernel de Linux – versión

Identificación para la versión del Kernel

1. Antes de la versión 1.0

- ▶ Primer versión: 0.01, seguido por la 0.02, 0.03, 0.10, 0.11, y 0.12 (primer versión GPL), etc. hasta la versión 1.0

2. Luego de la versión 1.0 y antes de las 2.6:

- ▶ La numeración tenía la forma **a.b.c**, donde: **a** era la versión del Kernel, **b** eran las revisiones principales y **c** las revisiones menores.
- ▶ En algunos casos se agregó un cuarto número (**a.b.c.d**), también la variante RC (release candidate)

Kernel de Linux – versión

Identificación para la versión del Kernel

1. Antes de la versión 1.0

- ▶ Primer versión: 0.01, seguido por la 0.02, 0.03, 0.10, 0.11, y 0.12 (primer versión GPL), etc. hasta la versión 1.0

2. Luego de la versión 1.0 y antes de las 2.6:

- ▶ La numeración tenía la forma **a.b.c**, donde: **a** era la versión del Kernel, **b** eran las revisiones principales y **c** las revisiones menores.
- ▶ En algunos casos se agregó un cuarto número (**a.b.c.d**), también la variante RC (release candidate)

3. A partir de la versión 2.6.0 (2004) comenzó una discusión de un nuevo mecanismo de numeración para períodos de releases más cortos.

Kernel de Linux – versión

Identificación para la versión del Kernel

1. Antes de la versión 1.0

- ▶ Primer versión: 0.01, seguido por la 0.02, 0.03, 0.10, 0.11, y 0.12 (primer versión GPL), etc. hasta la versión 1.0

2. Luego de la versión 1.0 y antes de las 2.6:

- ▶ La numeración tenía la forma **a.b.c**, donde: **a** era la versión del Kernel, **b** eran las revisiones principales y **c** las revisiones menores.
- ▶ En algunos casos se agregó un cuarto número (**a.b.c.d**), también la variante RC (release candidate)

3. A partir de la versión 2.6.0 (2004) comenzó una discusión de un nuevo mecanismo de numeración para períodos de releases más cortos.

- ▶ En 2011 se liberó la versión 3.0 (2.6.39)

Kernel de Linux – versión

Identificación para la versión del Kernel

1. Antes de la versión 1.0

- ▶ Primer versión: 0.01, seguido por la 0.02, 0.03, 0.10, 0.11, y 0.12 (primer versión GPL), etc. hasta la versión 1.0

2. Luego de la versión 1.0 y antes de las 2.6:

- ▶ La numeración tenía la forma **a.b.c**, donde: **a** era la versión del Kernel, **b** eran las revisiones principales y **c** las revisiones menores.
- ▶ En algunos casos se agregó un cuarto número (**a.b.c.d**), también la variante RC (release candidate)

3. A partir de la versión 2.6.0 (2004) comenzó una discusión de un nuevo mecanismo de numeración para períodos de releases más cortos.

- ▶ En 2011 se liberó la versión 3.0 (2.6.39)
- ▶ Sistema de versionado **x.y.z**

Kernel de Linux – versión

Identificación para la versión del Kernel

1. Antes de la versión 1.0

- ▶ Primer versión: 0.01, seguido por la 0.02, 0.03, 0.10, 0.11, y 0.12 (primer versión GPL), etc. hasta la versión 1.0

2. Luego de la versión 1.0 y antes de las 2.6:

- ▶ La numeración tenía la forma **a.b.c**, donde: **a** era la versión del Kernel, **b** eran las revisiones principales y **c** las revisiones menores.
- ▶ En algunos casos se agregó un cuarto número (**a.b.c.d**), también la variante RC (release candidate)

3. A partir de la versión 2.6.0 (2004) comenzó una discusión de un nuevo mecanismo de numeración para períodos de releases más cortos.

- ▶ En 2011 se liberó la versión 3.0 (2.6.39)
- ▶ Sistema de versionado **x.y.z**
- ▶ En enero de 2019 se anunció la versión 5.0 (de la 4.20)

The Linux Kernel Archives

[About](#)[Contact us](#)[FAQ](#)[Releases](#)[Signatures](#)[Site news](#)

| Protocol | Location |
|----------|---|
| HTTP | https://www.kernel.org/pub/ |
| GIT | https://git.kernel.org/ |
| RSYNC | rsync://rsync.kernel.org/pub/ |

Latest Release

5.11.10



| | | | | | | | |
|-----------|----------|------------|---------------------------|-------------------------|------------------------------|------------------------------|-----------------------------|
| mainline: | 5.12-rc4 | 2021-03-21 | [tarball] | [patch] | [inc. patch] | [view diff] | [browse] |
| stable: | 5.11.10 | 2021-03-25 | [tarball] | [pgp] | [patch] | [inc. patch] | [view diff] |
| longterm: | 5.10.26 | 2021-03-25 | [tarball] | [pgp] | [patch] | [inc. patch] | [view diff] |
| longterm: | 5.4.108 | 2021-03-24 | [tarball] | [pgp] | [patch] | [inc. patch] | [view diff] |
| longterm: | 4.19.183 | 2021-03-24 | [tarball] | [pgp] | [patch] | [inc. patch] | [view diff] |
| longterm: | 4.14.227 | 2021-03-24 | [tarball] | [pgp] | [patch] | [inc. patch] | [view diff] |
| longterm: | 4.9.263 | 2021-03-24 | [tarball] | [pgp] | [patch] | [inc. patch] | [view diff] |
| longterm: | 4.4.263 | 2021-03-24 | [tarball] | [pgp] | [patch] | [inc. patch] | [view diff] |

La shell de Linux

- ▶ Es la cara visible al usuario, interpreta las órdenes que recibe y las transmite al Kernel mediante *system calls*.
- ▶ Se la conoce también como *interpretes de comandos*

La shell de Linux

- ▶ Es la cara visible al usuario, interpreta las órdenes que recibe y las transmite al Kernel mediante *system calls*.
- ▶ Se la conoce también como *interpretes de comandos*

Algunas shell son:

`sh` : Bourne shell, escrita por Steve Bourne cuando estaba en Bell Labs

La shell de Linux

- ▶ Es la cara visible al usuario, interpreta las órdenes que recibe y las transmite al Kernel mediante *system calls*.
- ▶ Se la conoce también como *interpretes de comandos*

Algunas shell son:

`sh` : Bourne shell, escrita por Steve Bourne cuando estaba en Bell Labs

`csh` : C-shell, escrita por Bill Joy

La shell de Linux

- ▶ Es la cara visible al usuario, interpreta las órdenes que recibe y las transmite al Kernel mediante *system calls*.
- ▶ Se la conoce también como *interpretes de comandos*

Algunas shell son:

`sh` : Bourne shell, escrita por Steve Bourne cuando estaba en Bell Labs

`csh` : C-shell, escrita por Bill Joy

`ash` : Almquist shell, reemplazo de Bourne shell con licencia BSD

La shell de Linux

- ▶ Es la cara visible al usuario, interpreta las órdenes que recibe y las transmite al Kernel mediante *system calls*.
- ▶ Se la conoce también como *interpretes de comandos*

Algunas shell son:

`sh` : Bourne shell, escrita por Steve Bourne cuando estaba en Bell Labs

`csh` : C-shell, escrita por Bill Joy

`ash` : Almquist shell, reemplazo de Bourne shell con licencia BSD

`dash` : Debian Almquist shell, reemplazo de ash en Debian

La shell de Linux

- ▶ Es la cara visible al usuario, interpreta las órdenes que recibe y las transmite al Kernel mediante *system calls*.
- ▶ Se la conoce también como *interpretes de comandos*

Algunas shell son:

`sh` : Bourne shell, escrita por Steve Bourne cuando estaba en Bell Labs

`csh` : C-shell, escrita por Bill Joy

`ash` : Almquist shell, reemplazo de Bourne shell con licencia BSD

`dash` : Debian Almquist shell, reemplazo de ash en Debian

`bash` : Bourne-Again shell, se escribió como parte del proyecto GNU

La shell de Linux

- ▶ Es la cara visible al usuario, interpreta las órdenes que recibe y las transmite al Kernel mediante *system calls*.
- ▶ Se la conoce también como *interpretes de comandos*

Algunas shell son:

sh : Bourne shell, escrita por Steve Bourne cuando estaba en Bell Labs

csh : C-shell, escrita por Bill Joy

ash : Almquist shell, reemplazo de Bourne shell con licencia BSD

dash : Debian Almquist shell, reemplazo de ash en Debian

bash : Bourne-Again shell, se escribió como parte del proyecto GNU

zsh : Z shell, superconjunto de sh, ash, bash, csh, ksh, y tcsh

Entornos gráficos

- ▶ **Display Manager:** LightDM, GDM, KDM, LXDM, etc.
- ▶ **Window Manager:** Compiz, Metacity, Mutter, W9dk, fluxbox, etc.
- ▶ **Desktop Environment:** Gnome (GTK), KDE (Qt), LXDE (Lightweight X11 Desktop Environment), XFCE
- ▶ **Graphical User Interface (GUI)**

Algunas aplicaciones

Oficina: LibreOffice, L^AT_EX, LyX



Algunas aplicaciones

Oficina: LibreOffice, L^AT_EX, LyX

Edición imágenes, diagramas, etc.: Gimp,
Inkscape, Blender



Algunas aplicaciones

Oficina: LibreOffice, L^AT_EX, LyX

Edición imágenes, diagramas, etc.: Gimp,
Inkscape, Blender

Programación: GCC, make, ...



Algunas aplicaciones

Oficina: LibreOffice, L^AT_EX, LyX

Edición imágenes, diagramas, etc.: Gimp,
Inkscape, Blender

Programación: GCC, make, ...

IDE de programación: Gedit, Geany,
Code::Blocks, Eclipse CDT, Qt
Creator, Vim, Emacs



Algunas aplicaciones

Oficina: LibreOffice, L^AT_EX, LyX

Edición imágenes, diagramas, etc.: Gimp, Inkscape, Blender

Programación: GCC, make, ...

IDE de programación: Gedit, Geany, Code::Blocks, Eclipse CDT, Qt Creator, Vim, Emacs

Diseño de circuitos (esquem. y PCB), simulación, etc.: KiCAD, QUCS



Algunas aplicaciones

Oficina: LibreOffice, L^AT_EX, LyX

Edición imágenes, diagramas, etc.: Gimp,
Inkscape, Blender

Programación: GCC, make, ...

IDE de programación: Gedit, Geany,
Code::Blocks, Eclipse CDT, Qt
Creator, Vim, Emacs

Diseño de circuitos (esquem. y PCB), simulación, etc.:
KiCAD, QUCS

Matemática: GNU Octave, Maxima
(wxMaxima), Scilab, R, ...



Algunas aplicaciones

Oficina: LibreOffice, L^AT_EX, LyX

Edición imágenes, diagramas, etc.: Gimp, Inkscape, Blender

Programación: GCC, make, ...

IDE de programación: Gedit, Geany, Code::Blocks, Eclipse CDT, Qt Creator, Vim, Emacs

Diseño de circuitos (esquem. y PCB), simulación, etc.: KiCAD, QUCS

Matemática: GNU Octave, Maxima (wxMaxima), Scilab, R, ...

Otros: Firefox, Iceweasel, Evince, Okular, VLC, Audacity, ...



Informática II

La shell de Linux

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2021 –

Introducción

¿Qué es una shell?

Es un programa que sirve de interfaz entre el usuario y el SO Linux. Permite introducir comandos y que el SO los ejecute.

Introducción

¿Qué es una shell?

Es un programa que sirve de interfaz entre el usuario y el SO Linux. Permite introducir comandos y que el SO los ejecute.

Originalmente Linux no disponía de interfaz gráfica

Introducción

¿Qué es una shell?

Es un programa que sirve de interfaz entre el usuario y el SO Linux. Permite introducir comandos y que el SO los ejecute.

Originalmente Linux no disponía de interfaz gráfica

Programas de shell

- ▶ Se puede programar rápidamente y de forma simple
- ▶ Disponible en la mayoría de las instalaciones del SO Linux
- ▶ Programas de shell: *scripts* (interpretados en tiempo de ejecución)

Introducción

¿Qué es una shell?

Es un programa que sirve de interfaz entre el usuario y el SO Linux. Permite introducir comandos y que el SO los ejecute.

Originalmente Linux no disponía de interfaz gráfica

Programas de shell

- ▶ Se puede programar rápidamente y de forma simple
- ▶ Disponible en la mayoría de las instalaciones del SO Linux
- ▶ Programas de shell: *scripts* (interpretados en tiempo de ejecución)

POSIX.2, IEEE Std 1003.2-1992 indica las especificaciones mínimas de una shell¹:

¹POSIX: Portable Operating System Interface, familia de estándares del IEEE-CS

Introducción

¿Qué es una shell?

Es un programa que sirve de interfaz entre el usuario y el SO Linux. Permite introducir comandos y que el SO los ejecute.

Originalmente Linux no disponía de interfaz gráfica

Programas de shell

- ▶ Se puede programar rápidamente y de forma simple
- ▶ Disponible en la mayoría de las instalaciones del SO Linux
- ▶ Programas de shell: *scripts* (interpretados en tiempo de ejecución)

POSIX.2, IEEE Std 1003.2-1992 indica las especificaciones mínimas de una shell¹:

- ▶ Intérprete de comandos
- ▶ Programas de utilidad

¹POSIX: Portable Operating System Interface, familia de estándares del IEEE-CS

Introducción

Filosofía Unix

- ▶ Todo es un archivo.

Introducción

Filosofía Unix

- ▶ Todo es un archivo.
- ▶ Lo pequeño es bello.

Introducción

Filosofía Unix

- ▶ Todo es un archivo.
- ▶ Lo pequeño es bello.
- ▶ Escribir programas simples que hagan una sola cosa y lo haga bien.
Escribir programas que trabajen en conjunto. Escribir programas que manejen flujos de caractéres, dado que es una interfaz universal.

Introducción

Filosofía Unix

- ▶ Todo es un archivo.
- ▶ Lo pequeño es bello.
- ▶ Escribir programas simples que hagan una sola cosa y lo haga bien.
Escribir programas que trabajen en conjunto. Escribir programas que manejen flujos de caractéres, dado que es una interfaz universal.
- ▶ Guardar los datos y configuración en archivos de texto plano. Los archivos de texto son una interfaz universal, son fáciles de crear, guardar y mover a otro sistema.

Introducción

Filosofía Unix

- ▶ Todo es un archivo.
- ▶ Lo pequeño es bello.
- ▶ Escribir programas simples que hagan una sola cosa y lo haga bien.
Escribir programas que trabajen en conjunto. Escribir programas que manejen flujos de caractéres, dado que es una interfaz universal.
- ▶ Guardar los datos y configuración en archivos de texto plano. Los archivos de texto son una interfaz universal, son fáciles de crear, guardar y mover a otro sistema.
- ▶ Encadenar programas para realizar tareas complejas. Utilizar las tuberías (pipes) y filtros de la shell para encadenar pequeñas utilidades que realizan una tarea a la vez.

Introducción

Filosofía Unix

- ▶ Todo es un archivo.
- ▶ Lo pequeño es bello.
- ▶ Escribir programas simples que hagan una sola cosa y lo haga bien.
Escribir programas que trabajen en conjunto. Escribir programas que manejen flujos de caractéres, dado que es una interfaz universal.
- ▶ Guardar los datos y configuración en archivos de texto plano. Los archivos de texto son una interfaz universal, son fáciles de crear, guardar y mover a otro sistema.
- ▶ Encadenar programas para realizar tareas complejas. Utilizar las tuberías (pipes) y filtros de la shell para encadenar pequeñas utilidades que realizan una tarea a la vez.
- ▶ Elegir portabilidad sobre eficiencia.

Introducción

Filosofía Unix

- ▶ Todo es un archivo.
- ▶ Lo pequeño es bello.
- ▶ Escribir programas simples que hagan una sola cosa y lo haga bien.
Escribir programas que trabajen en conjunto. Escribir programas que manejen flujos de caractéres, dado que es una interfaz universal.
- ▶ Guardar los datos y configuración en archivos de texto plano. Los archivos de texto son una interfaz universal, son fáciles de crear, guardar y mover a otro sistema.
- ▶ Encadenar programas para realizar tareas complejas. Utilizar las tuberías (pipes) y filtros de la shell para encadenar pequeñas utilidades que realizan una tarea a la vez.
- ▶ Elegir portabilidad sobre eficiencia.

Por ejemplo:

```
> ls -la | more
```

Introducción

Filosofía Unix

- ▶ Todo es un archivo.
- ▶ Lo pequeño es bello.
- ▶ Escribir programas simples que hagan una sola cosa y lo haga bien.
Escribir programas que trabajen en conjunto. Escribir programas que manejen flujos de caractéres, dado que es una interfaz universal.
- ▶ Guardar los datos y configuración en archivos de texto plano. Los archivos de texto son una interfaz universal, son fáciles de crear, guardar y mover a otro sistema.
- ▶ Encadenar programas para realizar tareas complejas. Utilizar las tuberías (pipes) y filtros de la shell para encadenar pequeñas utilidades que realizan una tarea a la vez.
- ▶ Elegir portabilidad sobre eficiencia.

Por ejemplo:

```
> ls -la | more
```

KISS: Keep It Small and Simple... o...

Introducción

Filosofía Unix

- ▶ Todo es un archivo.
- ▶ Lo pequeño es bello.
- ▶ Escribir programas simples que hagan una sola cosa y lo haga bien.
Escribir programas que trabajen en conjunto. Escribir programas que manejen flujos de caractéres, dado que es una interfaz universal.
- ▶ Guardar los datos y configuración en archivos de texto plano. Los archivos de texto son una interfaz universal, son fáciles de crear, guardar y mover a otro sistema.
- ▶ Encadenar programas para realizar tareas complejas. Utilizar las tuberías (pipes) y filtros de la shell para encadenar pequeñas utilidades que realizan una tarea a la vez.
- ▶ Elegir portabilidad sobre eficiencia.

Por ejemplo:

```
> ls -la | more
```

KISS: Keep It Small and Simple... o... (Keep It Simple, Stupid! 😊)

Introducción - Ejemplo de script de shell

Programa de la shell

1. Secuencia de comandos ejecutados de forma interactiva
2. Guardar los comandos en un archivo que luego se invoca como un programa

Introducción - Ejemplo de script de shell

Programa de la shell

1. Secuencia de comandos ejecutados de forma interactiva
2. Guardar los comandos en un archivo que luego se invoca como un programa

Archivo hola1.sh

```
1 echo "Hola mundo"
```

Introducción - Ejemplo de script de shell

Programa de la shell

1. Secuencia de comandos ejecutados de forma interactiva
2. Guardar los comandos en un archivo que luego se invoca como un programa

Archivo hola1.sh

```
1 echo "Hola mundo"
```

Ejecutar

```
> bash hola1.sh
```

Introducción - Ejemplo de script de shell

Programa de la shell

1. Secuencia de comandos ejecutados de forma interactiva
2. Guardar los comandos en un archivo que luego se invoca como un programa

Archivo hola1.sh

```
1 echo "Hola mundo"
```

Ejecutar

```
> bash hola1.sh
```

Archivo hola2.sh

```
1 #!/bin/bash
2 echo "Hola mundo"
```

Introducción - Ejemplo de script de shell

Programa de la shell

1. Secuencia de comandos ejecutados de forma interactiva
2. Guardar los comandos en un archivo que luego se invoca como un programa

Archivo hola1.sh

```
1 echo "Hola mundo"
```

Ejecutar

```
> bash hola1.sh
```

Archivo hola2.sh

```
1 #!/bin/bash
2 echo "Hola mundo"
```

- ▶ Puede ejecutarse luego de darle permisos de ejecución (> ./hola2.sh)

Introducción - Ejemplo de script de shell

Archivo files.sh

```
1 #!/bin/bash
2
3 for file in *; do
4     echo -n "Nombre de archivo: "
5     echo ${file}
6 done
7
8 exit 0
```

Introducción - Ejemplo de script de shell

Archivo files.sh

```
1 #!/bin/bash
2
3 for file in *; do
4     echo -n "Nombre de archivo: "
5     echo ${file}
6 done
7
8 exit 0
```

- ▶ Los comentarios comienzan con #
- ▶ Línea especial #!/bin/bash
- ▶ El comando exit devuelve un valor de salida

Usuarios y grupos

Algunos comandos:

- ▶ whoami, groups, id, clear (atajo Ctrl+L en bash)

Usuarios y grupos

Algunos comandos:

- ▶ `whoami`, `groups`, `id`, `clear` (atajo `Ctrl+L` en `bash`)
- ▶ Archivo binarios o ejecutables localizados en el sistema. `which`
Ej. `which whoami`.

Usuarios y grupos

Algunos comandos:

- ▶ `whoami`, `groups`, `id`, `clear` (atajo `Ctrl+L` en `bash`)
- ▶ Archivo binarios o ejecutables localizados en el sistema. `which`
Ej. `which whoami`.
- ▶ ¿Dónde busca la shell los binarios de comandos?

Usuarios y grupos

Algunos comandos:

- ▶ `whoami`, `groups`, `id`, `clear` (atajo `Ctrl+L` en `bash`)
- ▶ Archivo binarios o ejecutables localizados en el sistema. `which`
Ej. `which whoami`.
- ▶ ¿Dónde busca la shell los binarios de comandos?
Ver variable de entorno `PATH`

Usuarios y grupos

Algunos comandos:

- ▶ `whoami`, `groups`, `id`, `clear` (atajo `Ctrl+L` en `bash`)
- ▶ Archivo binarios o ejecutables localizados en el sistema. `which`
Ej. `which whoami`.
- ▶ ¿Dónde busca la shell los binarios de comandos?
Ver variable de entorno `PATH`

Utilizar la función de auto-completar TAB y doble-TAB

Usuarios y grupos

Algunos comandos:

- ▶ `whoami, groups, id, clear` (atajo `Ctrl+L` en `bash`)
- ▶ Archivo binarios o ejecutables localizados en el sistema. `which`
Ej. `which whoami`.
- ▶ ¿Dónde busca la shell los binarios de comandos?
Ver variable de entorno `PATH`

Utilizar la función de auto-completar TAB y doble-TAB

Shell prompt: aparece en la línea de comandos indicando que está a la espera de órdenes

```
> echo $PATH  
> bash --version
```

Usuarios y grupos

Linux es un SO tipo Unix, multiplataforma, multitarea y multiusuario.

Usuarios y grupos

Linux es un SO tipo Unix, multiplataforma, multitarea y multiusuario.

- ▶ Para usar el SO es necesario abrir una sesión de trabajo (identificarse). Cada usuario tiene un *nombre de usuario* único y su correspondiente *número de usuario*, UID (user identifier).

Usuarios y grupos

Linux es un SO tipo Unix, multiplataforma, multitarea y multiusuario.

- ▶ Para usar el SO es necesario abrir una sesión de trabajo (identificarse). Cada usuario tiene un *nombre de usuario* único y su correspondiente *número de usuario*, UID (user identifier).
- ▶ Los usuarios se organizan en grupos, identificados también por un números, GID.

Usuarios y grupos

Linux es un SO tipo Unix, multiplataforma, multitarea y multiusuario.

- ▶ Para usar el SO es necesario abrir una sesión de trabajo (identificarse). Cada usuario tiene un *nombre de usuario* único y su correspondiente *número de usuario*, UID (user identifier).
- ▶ Los usuarios se organizan en grupos, identificados también por un números, GID.
 - ▶ Ej.: para acceder a dispositivos `tty` hay que estar en el grupo `dialout`

Usuarios y grupos

Linux es un SO tipo Unix, multiplataforma, multitarea y multiusuario.

- ▶ Para usar el SO es necesario abrir una sesión de trabajo (identificarse). Cada usuario tiene un *nombre de usuario* único y su correspondiente *número de usuario*, UID (user identifier).
- ▶ Los usuarios se organizan en grupos, identificados también por un números, GID.
 - ▶ Ej.: para acceder a dispositivos `tty` hay que estar en el grupo `dialout`
 - ▶ Algunos grupos: `adm`, `sudo`, `plugdev`, `docker`

Usuarios y grupos

Linux es un SO tipo Unix, multiplataforma, multitarea y multiusuario.

- ▶ Para usar el SO es necesario abrir una sesión de trabajo (identificarse). Cada usuario tiene un *nombre de usuario* único y su correspondiente *número de usuario*, UID (user identifier).
- ▶ Los usuarios se organizan en grupos, identificados también por un números, GID.
 - ▶ Ej.: para acceder a dispositivos `tty` hay que estar en el grupo `dialout`
 - ▶ Algunos grupos: `adm`, `sudo`, `plugdev`, `docker`

Superusuario

- ▶ Tiene privilegios sobre todo el sistema. Nombre de usuario `root`, UID = 0

Usuarios y grupos

Linux es un SO tipo Unix, multiplataforma, multitarea y multiusuario.

- ▶ Para usar el SO es necesario abrir una sesión de trabajo (identificarse). Cada usuario tiene un *nombre de usuario* único y su correspondiente *número de usuario*, UID (user identifier).
- ▶ Los usuarios se organizan en grupos, identificados también por un números, GID.
 - ▶ Ej.: para acceder a dispositivos `tty` hay que estar en el grupo `dialout`
 - ▶ Algunos grupos: `adm`, `sudo`, `plugdev`, `docker`

Superusuario

- ▶ Tiene privilegios sobre todo el sistema. Nombre de usuario `root`, UID = 0
- ▶ Utilizado por el administrador del sistema para tareas administrativas

Usuarios y grupos

Por cada usuario existente se tiene una línea en el archivo del sistema
`/etc/passwd`

```
> cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
gfpp:x:1000:1000:Gonzalo Perez Paina , , ,:/home/gfpp:/bin/bash
```

Usuarios y grupos

Por cada usuario existente se tiene una línea en el archivo del sistema
`/etc/passwd`

```
> cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
gfpp:x:1000:1000:Gonzalo Perez Paina , , ,:/home/gfpp:/bin/bash
```

Contiene la siguiente información:

- ▶ *ID del grupo*: identificador numérico de group (ID) del primero de los grupos a los cuales el usuario es miembro
- ▶ *Directorio home*: el directorio donde se encuentra el usuario luego del login
- ▶ *Shell*: nombre del programa que se ejecuta como interprete de comandos de usuarios

Usuarios y grupos

Por cada usuario existente se tiene una línea en el archivo del sistema
`/etc/group`

```
> cat /etc/group
root:x:0:
adm:x:4:syslog ,gfpp
gfpp:x:1000:
```

Usuarios y grupos

Por cada usuario existente se tiene una línea en el archivo del sistema
`/etc/group`

```
> cat /etc/group
root:x:0:
adm:x:4:syslog,gfpp
gfpp:x:1000:
```

Contiene la siguiente información:

- ▶ *Nombre del grupo*: nombre (único) del grupo
- ▶ *ID del grupo*: ID numérico asociado a ese grupo
- ▶ *Lista de usuarios*: lista separada por coma de nombres de usuarios que son miembros de este grupo

Variables de entorno

- ▶ Variables inicializadas desde el entorno (shell)

Variables de entorno

- ▶ Variables inicializadas desde el entorno (shell)
- ▶ Son de gran utilidad en la programación de script de la shell

Variables de entorno

- ▶ Variables inicializadas desde el entorno (shell)
- ▶ Son de gran utilidad en la programación de script de la shell
- ▶ Escritas en mayúsculas para distinguirlas de otras variables (usuario)

Variables de entorno

- ▶ Variables inicializadas desde el entorno (shell)
- ▶ Son de gran utilidad en la programación de script de la shell
- ▶ Escritas en mayúsculas para distinguirlas de otras variables (usuario)
- ▶ Ej.: HOME, PATH, SHELL, ?

Variables de entorno

- ▶ Variables inicializadas desde el entorno (shell)
- ▶ Son de gran utilidad en la programación de script de la shell
- ▶ Escritas en mayúsculas para distinguirlas de otras variables (usuario)
- ▶ Ej.: HOME, PATH, SHELL, ?
- ▶ Imprimir valor de variable de entorno: `echo $HOME`

Variables de entorno

- ▶ Variables inicializadas desde el entorno (shell)
- ▶ Son de gran utilidad en la programación de script de la shell
- ▶ Escritas en mayúsculas para distinguirlas de otras variables (usuario)
- ▶ Ej.: HOME, PATH, SHELL, ?
- ▶ Imprimir valor de variable de entorno: `echo $HOME`

Algunas variables de entorno:

- ▶ `$HOME`: directorio home del usuario actual

Variables de entorno

- ▶ Variables inicializadas desde el entorno (shell)
- ▶ Son de gran utilidad en la programación de script de la shell
- ▶ Escritas en mayúsculas para distinguirlas de otras variables (usuario)
- ▶ Ej.: HOME, PATH, SHELL, ?
- ▶ Imprimir valor de variable de entorno: `echo $HOME`

Algunas variables de entorno:

- ▶ `$HOME`: directorio home del usuario actual
- ▶ `$PATH`: lista de directorios separados por : para buscar los comandos

Variables de entorno

- ▶ Variables inicializadas desde el entorno (shell)
- ▶ Son de gran utilidad en la programación de script de la shell
- ▶ Escritas en mayúsculas para distinguirlas de otras variables (usuario)
- ▶ Ej.: HOME, PATH, SHELL, ?
- ▶ Imprimir valor de variable de entorno: `echo $HOME`

Algunas variables de entorno:

- ▶ `$HOME`: directorio home del usuario actual
- ▶ `$PATH`: lista de directorios separados por : para buscar los comandos
- ▶ `$PS1`: prompt de comandos

Variables de entorno

- ▶ Variables inicializadas desde el entorno (shell)
- ▶ Son de gran utilidad en la programación de script de la shell
- ▶ Escritas en mayúsculas para distinguirlas de otras variables (usuario)
- ▶ Ej.: HOME, PATH, SHELL, ?
- ▶ Imprimir valor de variable de entorno: `echo $HOME`

Algunas variables de entorno:

- ▶ `$HOME`: directorio home del usuario actual
- ▶ `$PATH`: lista de directorios separados por : para buscar los comandos
- ▶ `$PS1`: prompt de comandos
- ▶ `$PS2`: prompt secundario

Variables de entorno

- ▶ Variables inicializadas desde el entorno (shell)
- ▶ Son de gran utilidad en la programación de script de la shell
- ▶ Escritas en mayúsculas para distinguirlas de otras variables (usuario)
- ▶ Ej.: HOME, PATH, SHELL, ?
- ▶ Imprimir valor de variable de entorno: `echo $HOME`

Algunas variables de entorno:

- ▶ `$HOME`: directorio home del usuario actual
- ▶ `$PATH`: lista de directorios separados por : para buscar los comandos
- ▶ `$PS1`: prompt de comandos
- ▶ `$PS2`: prompt secundario
- ▶ `$0`: nombre del script de shell

Variables de entorno

- ▶ Variables inicializadas desde el entorno (shell)
- ▶ Son de gran utilidad en la programación de script de la shell
- ▶ Escritas en mayúsculas para distinguirlas de otras variables (usuario)
- ▶ Ej.: HOME, PATH, SHELL, ?
- ▶ Imprimir valor de variable de entorno: `echo $HOME`

Algunas variables de entorno:

- ▶ `$HOME`: directorio home del usuario actual
- ▶ `$PATH`: lista de directorios separados por : para buscar los comandos
- ▶ `$PS1`: prompt de comandos
- ▶ `$PS2`: prompt secundario
- ▶ `$0`: nombre del script de shell
- ▶ `$#`: cantidad de parámetros pasados

Más comandos 😊

- ▶ Comandos echo, printenv, grep
 - > printenv | grep PATH

Más comandos 😊

- ▶ Comandos echo, printenv, grep
 - > printenv | grep PATH
- ▶ Comandos cat, head, tail
 - > cat num.txt | head
 - > cat num.txt | tail
 - > cat num.txt | grep 10

Más comandos 😊

- ▶ Comandos echo, printenv, grep
 - > printenv | grep PATH
- ▶ Comandos cat, head, tail
 - > cat num.txt | head
 - > cat num.txt | tail
 - > cat num.txt | grep 10
- ▶ Comandos apropos, man, whatis

Más comandos 😊

- ▶ Comandos echo, printenv, grep
 - > printenv | grep PATH
- ▶ Comandos cat, head, tail
 - > cat num.txt | head
 - > cat num.txt | tail
 - > cat num.txt | grep 10
- ▶ Comandos apropos, man, whatis
- ▶ Opciones de los comandos, letra seguido de '-'. O bien '--' (--help, --version)

Más comandos 😊

- ▶ Comandos echo, printenv, grep
 - > printenv | grep PATH
- ▶ Comandos cat, head, tail
 - > cat num.txt | head
 - > cat num.txt | tail
 - > cat num.txt | grep 10
- ▶ Comandos apropos, man, whatis
- ▶ Opciones de los comandos, letra seguido de '-'. O bien '--' (--help, --version)
- ▶ [Manpages](#): manual en línea (RTFM).
 - > man printenv
 - > man 1 printf
 - > man 3 printf
 - > man man

Páginas de manuales (manpages)

Cuenta con diferentes secciones: > `man man`

Páginas de manuales (manpages)

Cuenta con diferentes secciones: > `man man`

Algunas son:

1. Executable programs or shell commands
2. System calls (functions provided by the kernel)
3. Library calls (functions within program libraries)
4. Special files (usually found in `/dev`)
5. File formats and conventions eg. `/etc/passwd`
6. Games
7. Miscellaneous (including macro packages and conventions)
8. System administration commands (usually only for root)

Páginas de manuales (manpages)

Cuenta con diferentes secciones: > `man man`

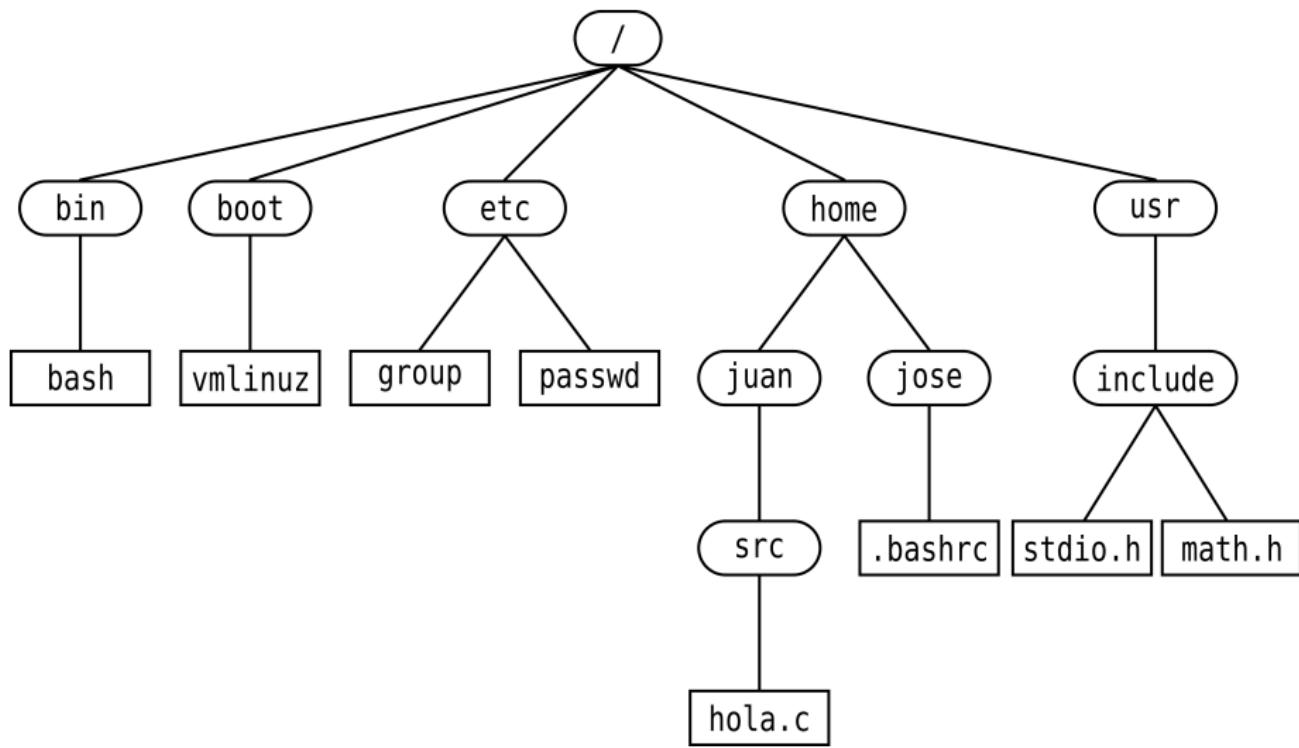
Algunas son:

1. Executable programs or shell commands
2. System calls (functions provided by the kernel)
3. Library calls (functions within program libraries)
4. Special files (usually found in `/dev`)
5. File formats and conventions eg. `/etc/passwd`
6. Games
7. Miscellaneous (including macro packages and conventions)
8. System administration commands (usually only for root)

Ejemplos:

```
> man 1 printf  
> man 3 printf  
> man -a printf  
> man -k '^printf'
```


Sistema de archivos y permisos



Sistema de archivos y permisos

Estructura de archivos en árbol. Archivos “tipo directorio”. Comando `ls`

Sistema de archivos y permisos

Estructura de archivos en árbol. Archivos “tipo directorio”. Comando `ls`

- ▶ Directorios (raíz/root)

Sistema de archivos y permisos

Estructura de archivos en árbol. Archivos “tipo directorio”. Comando `ls`

- ▶ Directorios (raíz/root)
- ▶ Camino/ruta/path

Sistema de archivos y permisos

Estructura de archivos en árbol. Archivos “tipo directorio”. Comando `ls`

- ▶ Directorios (raíz/root)
- ▶ Camino/ruta/path
- ▶ Directorio actual: ‘.’ (punto)

Sistema de archivos y permisos

Estructura de archivos en árbol. Archivos “tipo directorio”. Comando `ls`

- ▶ Directorios (raíz/root)
- ▶ Camino/ruta/path
- ▶ Directorio actual: `..` (punto)
- ▶ Directorio anterior o padre: `..` (doble punto)

Sistema de archivos y permisos

Estructura de archivos en árbol. Archivos “tipo directorio”. Comando `ls`

- ▶ Directorios (raíz/root)
- ▶ Camino/ruta/path
- ▶ Directorio actual: `..` (punto)
- ▶ Directorio anterior o padre: `..` (doble punto)
- ▶ Camino absoluto (comienza en `/`)

Sistema de archivos y permisos

Estructura de archivos en árbol. Archivos “tipo directorio”. Comando `ls`

- ▶ Directorios (raíz/root)
- ▶ Camino/ruta/path
- ▶ Directorio actual: `..` (punto)
- ▶ Directorio anterior o padre: `..` (doble punto)
- ▶ Camino absoluto (comienza en `/`)
- ▶ Camino relativo (comienza en `./` o `../`)

Sistema de archivos y permisos

Estructura de archivos en árbol. Archivos “tipo directorio”. Comando `ls`

- ▶ Directorios (raíz/root)
- ▶ Camino/ruta/path
- ▶ Directorio actual: `..` (punto)
- ▶ Directorio anterior o padre: `..` (doble punto)
- ▶ Camino absoluto (comienza en `/`)
- ▶ Camino relativo (comienza en `./` o `../`)

Ver archivos ocultos: `ls -a`

- ▶ Directorio `/home`. Variable de entorno `HOME` (`ls $HOME`). (`cd`, `pwd`)

Sistema de archivos y permisos

Estructura de archivos en árbol. Archivos “tipo directorio”. Comando `ls`

- ▶ Directorios (raíz/root)
- ▶ Camino/ruta/path
- ▶ Directorio actual: `..` (punto)
- ▶ Directorio anterior o padre: `..` (doble punto)
- ▶ Camino absoluto (comienza en `/`)
- ▶ Camino relativo (comienza en `./` o `../`)

Ver archivos ocultos: `ls -a`

- ▶ Directorio `/home`. Variable de entorno `HOME` (`ls $HOME`). (`cd`, `pwd`)
- ▶ Atributos de archivos (`ls -l /`)

Sistema de archivos y permisos

Estructura de archivos en árbol. Archivos “tipo directorio”. Comando `ls`

- ▶ Directorios (raíz/root)
- ▶ Camino/ruta/path
- ▶ Directorio actual: `..` (punto)
- ▶ Directorio anterior o padre: `..` (doble punto)
- ▶ Camino absoluto (comienza en `/`)
- ▶ Camino relativo (comienza en `./` o `../`)

Ver archivos ocultos: `ls -a`

- ▶ Directorio `/home`. Variable de entorno `HOME` (`ls $HOME`). (`cd`, `pwd`)
- ▶ Atributos de archivos (`ls -l /`)

Comandos

`mkdir`, `rmdir`, `cp`
`mv`, `rm`, `touch`. (`which cd`)

Hacer:

> `cd`
> `touch hola.txt`
> `ls -l hola.txt`

```
-rw-rw-r-- 1 gfpp gfpp 11 mar 29 17:03 hola.txt
```

nombre del archivo

minutos : Fecha y
hora : hora de la
día del mes : última
mes : modificación

Tamaño en bytes

Nombre del grupo

Nombre del propietario

nro. de enlace rígido (hard link)

001 permiso de ejecución : Para
002 permiso de escritura : un usuario
004 permiso de lectura : cualquiera

010 permiso de ejecución : Para usuario
020 permiso de escritura : perteneciente
040 permiso de lectura : al grupo

100 permiso de ejecución : Para usuario
200 permiso de escritura : propietario
400 permiso de lectura :

Tipo de archivo

Sistema de archivos y permisos

Tipos de archivos en Linux

1. Archivos regulares (-)

Sistema de archivos y permisos

Tipos de archivos en Linux

1. Archivos regulares (-)
2. Archivos directorios (d)

Sistema de archivos y permisos

Tipos de archivos en Linux

1. Archivos regulares (-)
2. Archivos directorios (d)
3. Archivos especiales

Sistema de archivos y permisos

Tipos de archivos en Linux

1. Archivos regulares (-)
2. Archivos directorios (d)
3. Archivos especiales
 - ▶ Archivos de bloque (b)

Sistema de archivos y permisos

Tipos de archivos en Linux

1. Archivos regulares (-)
2. Archivos directorios (d)
3. Archivos especiales
 - ▶ Archivos de bloque (b)
 - ▶ Archivos de caracteres (c)

Sistema de archivos y permisos

Tipos de archivos en Linux

1. Archivos regulares (-)
2. Archivos directorios (d)
3. Archivos especiales
 - ▶ Archivos de bloque (b)
 - ▶ Archivos de caracteres (c)
 - ▶ Enlaces simbólicos (l)

Sistema de archivos y permisos

Tipos de archivos en Linux

1. Archivos regulares (-)
2. Archivos directorios (d)
3. Archivos especiales
 - ▶ Archivos de bloque (b)
 - ▶ Archivos de caracteres (c)
 - ▶ Enlaces simbólicos (l)

Probar:

```
> ls -l | grep ^-
> ls -l | grep ^d
> cd /dev
> ls -l | grep ^b
> ls -l | grep ^c
> ls -l | grep ^l
```

Sistema de archivos y permisos

Alterar permisos de archivos – comando **chmod**

► > **chmod u-r hola.txt.**

Sistema de archivos y permisos

Alterar permisos de archivos – comando `chmod`

- ▶ > `chmod u-r hola.txt.`
- ▶ > `cat hola.txt`

Sistema de archivos y permisos

Alterar permisos de archivos – comando `chmod`

- ▶ > `chmod u-r hola.txt.`
- ▶ > `cat hola.txt`
- ▶ > `chmod -x a.out`

Sistema de archivos y permisos

Alterar permisos de archivos – comando `chmod`

- ▶ > `chmod u-r hola.txt.`
- ▶ > `cat hola.txt`
- ▶ > `chmod -x a.out`
- ▶ > `chmod o+x a.out`

Sistema de archivos y permisos

Alterar permisos de archivos – comando `chmod`

- ▶ > `chmod u-r hola.txt.`
- ▶ > `cat hola.txt`
- ▶ > `chmod -x a.out`
- ▶ > `chmod o+x a.out`

Notación numérica de los permisos

| | | | | | | | | | | |
|---|---|---|--|---|---|---|--|---|---|---|
| r | w | x | | - | w | x | | r | - | x |
| 4 | 2 | 1 | | 0 | 2 | 1 | | 4 | 0 | 1 |

Equivale a un permiso 735
 $4+2+1, 0+2+1, 4+0+1 = 7, 3, 5$

Más comandos 😊

- ▶ `uptime`, `cal`
- ▶ `uname` (`-n`, `-v`, `-r`, `-m`, `-a`)
- ▶ `lshw`, `lsusb`, `lspci` (`-tv`)
- ▶ `df -h`
- ▶ `dmesg`
- ▶ `find`

(Consultar las páginas de manuales para ver la función de cada comando)

Más comandos 😊

- ▶ `uptime`, `cal`
- ▶ `uname` (`-n`, `-v`, `-r`, `-m`, `-a`)
- ▶ `lshw`, `lsusb`, `lspci` (`-tv`)
- ▶ `df -h`
- ▶ `dmesg`
- ▶ `find`

(Consultar las páginas de manuales para ver la función de cada comando)

Ejemplos:

```
> find / -name 'uname -r'  
> find / -name 'uname -r' 2> /dev/null
```


Redirección de entrada, salida y error

Redirección de salida (`stdout`)

► `ls -l > lsoutput.txt`

Redirecciona la salida del comando `ls` al archivo `lsoutput.txt`

Redirección de entrada, salida y error

Redirección de salida (`stdout`)

- ▶ `ls -l > lsoutput.txt`
Redirecciona la salida del comando `ls` al archivo `lsoutput.txt`
- ▶ `ps >> lsoutput.txt`
Agrega la salida del comando `ps` al archivo

Redirección de entrada, salida y error

Redirección de salida (`stdout`)

- ▶ `ls -l > lsoutput.txt`
Redirecciona la salida del comando `ls` al archivo `lsoutput.txt`
- ▶ `ps >> lsoutput.txt`
Agrega la salida del comando `ps` al archivo
- ▶ Otro ejemplo:
`cat lsoutput.txt > /dev/null`

Redirección de entrada, salida y error

Redirección de salida (`stdout`)

- ▶ `ls -l > lsoutput.txt`
Redirecciona la salida del comando `ls` al archivo `lsoutput.txt`
- ▶ `ps >> lsoutput.txt`
Agrega la salida del comando `ps` al archivo
- ▶ Otro ejemplo:
`cat lsoutput.txt > /dev/null`

Redirección de salida y error (`stdout` y `stderr`)

- ▶ `ls -ld /tmp /tnt`

Redirección de entrada, salida y error

Redirección de salida (`stdout`)

- ▶ `ls -l > lsoutput.txt`
Redirecciona la salida del comando `ls` al archivo `lsoutput.txt`
- ▶ `ps >> lsoutput.txt`
Agrega la salida del comando `ps` al archivo
- ▶ Otro ejemplo:
`cat lsoutput.txt > /dev/null`

Redirección de salida y error (`stdout` y `stderr`)

- ▶ `ls -ld /tmp /tnt`
- ▶ `ls -ld /tmp /tnt >/dev/null`

Redirección de entrada, salida y error

Redirección de salida (`stdout`)

- ▶ `ls -l > lsoutput.txt`
Redirecciona la salida del comando `ls` al archivo `lsoutput.txt`
- ▶ `ps >> lsoutput.txt`
Agrega la salida del comando `ps` al archivo
- ▶ Otro ejemplo:
`cat lsoutput.txt > /dev/null`

Redirección de salida y error (`stdout` y `stderr`)

- ▶ `ls -ld /tmp /tnt`
- ▶ `ls -ld /tmp /tnt >/dev/null`
- ▶ `ls -ld /tmp /tnt 2>/dev/null`

Redirección de entrada, salida y error

Redirección de salida (`stdout`)

- ▶ `ls -l > lsoutput.txt`
Redirecciona la salida del comando `ls` al archivo `lsoutput.txt`
- ▶ `ps >> lsoutput.txt`
Agrega la salida del comando `ps` al archivo
- ▶ Otro ejemplo:
`cat lsoutput.txt > /dev/null`

Redirección de salida y error (`stdout` y `stderr`)

- ▶ `ls -ld /tmp /tnt`
- ▶ `ls -ld /tmp /tnt >/dev/null`
- ▶ `ls -ld /tmp /tnt 2>/dev/null`

Descriptor de archivo 0: entrada estándar, 1: salida estándar, 2: salida de error estándar

Redirección de entrada, salida y error

Redirección de salida (`stdout`)

- ▶ `ls -l > lsoutput.txt`
Redirecciona la salida del comando `ls` al archivo `lsoutput.txt`
- ▶ `ps >> lsoutput.txt`
Agrega la salida del comando `ps` al archivo
- ▶ Otro ejemplo:
`cat lsoutput.txt > /dev/null`

Redirección de salida y error (`stdout` y `stderr`)

- ▶ `ls -ld /tmp /tnt`
- ▶ `ls -ld /tmp /tnt >/dev/null`
- ▶ `ls -ld /tmp /tnt 2>/dev/null`

Descriptor de archivo 0: entrada estándar, 1: salida estándar, 2: salida de error estándar

Redirección de entrada (`stdin`) [`ls -lR /usr/include > lsoutput.txt`]

- ▶ `more < lsoutput.txt`

Redirección de entrada, salida y error

Redirección de salida (`stdout`)

- ▶ `ls -l > lsoutput.txt`
Redirecciona la salida del comando `ls` al archivo `lsoutput.txt`
- ▶ `ps >> lsoutput.txt`
Agrega la salida del comando `ps` al archivo
- ▶ Otro ejemplo:
`cat lsoutput.txt > /dev/null`

Redirección de salida y error (`stdout` y `stderr`)

- ▶ `ls -ld /tmp /tnt`
- ▶ `ls -ld /tmp /tnt >/dev/null`
- ▶ `ls -ld /tmp /tnt 2>/dev/null`

Descriptor de archivo 0: entrada estándar, 1: salida estándar, 2: salida de error estándar

Redirección de entrada (`stdin`) [`ls -lR /usr/include > lsoutput.txt`]

- ▶ `more < lsoutput.txt`

Ver ejemplo de código fuente con `stdout` y `stderr` (`sqrt.c`)

Otro uso de la redirección de entrada salida

Ver forma de participar de aceptaelreto.com

Otro uso de la redirección de entrada salida

Ver forma de participar de aceptaelreto.com

```
./a.out < entrada > salida  
diff salida salida.ok
```

Otro uso de la redirección de entrada salida

Ver forma de participar de aceptaelreto.com

```
./a.out < entrada > salida  
diff salida salida.ok
```

Opciones de **diff**:

- ▶ **-s**: informa si los archivos son iguales
 - ▶ **-q**: informa si los archivos son diferentes
 - ▶ **-c**: formato de salida de contexto
 - ▶ **-u**: formato de salida unificado
 - ▶ **-i**: ignora las diferencias entre mayúsculas y minúsculas
- (**man diff**)

Pipes

Se puede conectar procesos utilizando el operador pipe, |.

Ejemplo: utilizar `sort` para ordenar de forma inversa la salida de `ls`

```
ls -1 | sort -r
```

(Ver el manual de `ls` y `sort`)

Pipes

Se puede conectar procesos utilizando el operador pipe, |.

Ejemplo: utilizar `sort` para ordenar de forma inversa la salida de `ls`

```
ls -1 | sort -r
```

(Ver el manual de `ls` y `sort`)

Si no se utiliza pipes, se necesitan varios pasos

1. `ls -1 > ls.txt`
2. `sort -r ls.txt > lsrev.txt`

Pipes

Se puede conectar procesos utilizando el operador pipe, |.

Ejemplo: utilizar `sort` para ordenar de forma inversa la salida de `ls`

```
ls -1 | sort -r
```

(Ver el manual de `ls` y `sort`)

Si no se utiliza pipes, se necesitan varios pasos

1. `ls -1 > ls.txt`
2. `sort -r ls.txt > lsrev.txt`

Conectando los procesos mediante pipes

```
► ls -1 | sort -r > lsrev.txt
```

Pipes

Se puede conectar procesos utilizando el operador pipe, |.

Ejemplo: utilizar `sort` para ordenar de forma inversa la salida de `ls`

```
ls -1 | sort -r
```

(Ver el manual de `ls` y `sort`)

Si no se utiliza pipes, se necesitan varios pasos

1. `ls -1 > ls.txt`
2. `sort -r ls.txt > lsrev.txt`

Conectando los procesos mediante pipes

► `ls -1 | sort -r > lsrev.txt`

Otros ejemplos

- `ps aux | sort`
- `ps aux | sort | more`

Procesos

Un proceso es una instancia de un programa en ejecución

Procesos

Un proceso es una instancia de un programa en ejecución

Cuando se ejecuta un programa (`./a.out`, `ls`, `gcc`, etc.), el kernel:

- ▶ carga el código del programa en memoria virtual,

Procesos

Un proceso es una instancia de un programa en ejecución

Cuando se ejecuta un programa (`./a.out`, `ls`, `gcc`, etc.), el kernel:

- ▶ carga el código del programa en memoria virtual,
- ▶ reserva espacio para las variables del programa, e

Procesos

Un proceso es una instancia de un programa en ejecución

Cuando se ejecuta un programa (`./a.out`, `ls`, `gcc`, etc.), el kernel:

- ▶ carga el código del programa en memoria virtual,
- ▶ reserva espacio para las variables del programa, e
- ▶ inicializa estructuras de datos propias del kernel para guardar información del proceso (PID, estado, IDs de usuario y grupo, etc.).

Procesos

Un proceso es una instancia de un programa en ejecución

Cuando se ejecuta un programa (`./a.out`, `ls`, `gcc`, etc.), el kernel:

- ▶ carga el código del programa en memoria virtual,
- ▶ reserva espacio para las variables del programa, e
- ▶ inicializa estructuras de datos propias del kernel para guardar información del proceso (PID, estado, IDs de usuario y grupo, etc.).

El estándar UNIX (IEEE Std 1003.1-2004) define un proceso como:

“an address space with one or more threads executing within that address space, and the required system resources for those threads”

Procesos

Un proceso se divide lógicamente en las siguientes partes, conocidas como *segmentos*:

Procesos

Un proceso se divide lógicamente en las siguientes partes, conocidas como *segmentos*:

text : instrucciones de un programa

Procesos

Un proceso se divide lógicamente en las siguientes partes, conocidas como *segmentos*:

text : instrucciones de un programa

data : variables estáticas utilizadas por un programa

Procesos

Un proceso se divide lógicamente en las siguientes partes, conocidas como *segmentos*:

text : instrucciones de un programa

data : variables estáticas utilizadas por un programa

heap : área desde la cual un programa puede asignar memoria adicional de forma dinámica

Procesos

Un proceso se divide lógicamente en las siguientes partes, conocidas como *segmentos*:

text : instrucciones de un programa

data : variables estáticas utilizadas por un programa

heap : área desde la cual un programa puede asignar memoria adicional de forma dinámica

stack : porción de memoria que crece o se encoje a medida que las funciones se llaman y retornan; utilizada para almacenar las variables locales e información de enlace de las funciones

Procesos

Un proceso se divide lógicamente en las siguientes partes, conocidas como *segmentos*:

text : instrucciones de un programa

data : variables estáticas utilizadas por un programa

heap : área desde la cual un programa puede asignar memoria adicional de forma dinámica

stack : porción de memoria que crece o se encoje a medida que las funciones se llaman y retornan; utilizada para almacenar las variables locales e información de enlace de las funciones

- ▶ Cada proceso tiene asociado un identificador único, PID (entero positivo entre 2 y 32768)
- ▶ Cada proceso tiene asociado un identificador del proceso padre, PPID
- ▶ El nro. de proceso 1 está reservado para el proceso `init/systemd`

Procesos

Un proceso se divide lógicamente en las siguientes partes, conocidas como *segmentos*:

text : instrucciones de un programa

data : variables estáticas utilizadas por un programa

heap : área desde la cual un programa puede asignar memoria adicional de forma dinámica

stack : porción de memoria que crece o se encoje a medida que las funciones se llaman y retornan; utilizada para almacenar las variables locales e información de enlace de las funciones

- ▶ Cada proceso tiene asociado un identificador único, PID (entero positivo entre 2 y 32768)
- ▶ Cada proceso tiene asociado un identificador del proceso padre, PPID
- ▶ El nro. de proceso 1 está reservado para el proceso `init/systemd`

Ver comandos: `ps`, `top`, `htop`, `pstree`, `ps aux` (`man ps`)

Procesos

Atributos de los procesos

- ▶ PID: Valor numérico que identifica al proceso
- ▶ TTY: Terminal asociada al proceso
- ▶ STAT: Estado del proceso
- ▶ TIME: Tiempo de CPU consumido por el proceso
- ▶ COMMAND: Comandos y argumentos utilizados

Procesos

Atributos de los procesos

- ▶ PID: Valor numérico que identifica al proceso
- ▶ TTY: Terminal asociada al proceso
- ▶ STAT: Estado del proceso
- ▶ TIME: Tiempo de CPU consumido por el proceso
- ▶ COMMAND: Comandos y argumentos utilizados

Algunos estados posibles:

- ▶ S: Sleeping. Esperando un evento (señal o entrada)
- ▶ R: Running. En la cola para ser ejecutado
- ▶ T: Stopped, por la shell o debugger
- ▶ N: Tarea de baja prioridad
- ▶ Z: Proceso zombie
- ▶ <: Tarea de alta prioridad

Procesos

Ejecución de procesos

Primer plano: Ejecución normal, la shell no admite otro comando hasta que haya terminado el proceso en ejecución

Procesos

Ejecución de procesos

Primer plano: Ejecución normal, la shell no admite otro comando hasta que haya terminado el proceso en ejecución

Segundo plano: Permite la ejecución de otros comandos mientras se ejecuta el proceso (&)

Procesos

Ejecución de procesos

Primer plano: Ejecución normal, la shell no admite otro comando hasta que haya terminado el proceso en ejecución

Segundo plano: Permite la ejecución de otros comandos mientras se ejecuta el proceso (&)

Combinación de teclas:

- ▶ **Ctrl+C:** interrumpe la ejecución del proceso
- ▶ **Ctrl+Z:** suspende la ejecución del proceso

Procesos

Ejecución de procesos

Primer plano: Ejecución normal, la shell no admite otro comando hasta que haya terminado el proceso en ejecución

Segundo plano: Permite la ejecución de otros comandos mientras se ejecuta el proceso (&)

Combinación de teclas:

- ▶ **Ctrl+C:** interrumpe la ejecución del proceso
- ▶ **Ctrl+Z:** suspende la ejecución del proceso

Probar lanzar procesos en segundo plano (gedit, gnome-calculator, evince, etc.). Ver comando **jobs**.

Procesos

Ejecución de procesos

Primer plano: Ejecución normal, la shell no admite otro comando hasta que haya terminado el proceso en ejecución

Segundo plano: Permite la ejecución de otros comandos mientras se ejecuta el proceso (**&**)

Combinación de teclas:

- ▶ **Ctrl+C:** interrumpe la ejecución del proceso
- ▶ **Ctrl+Z:** suspende la ejecución del proceso

Probar lanzar procesos en segundo plano (gedit, gnome-calculator, evince, etc.). Ver comando **jobs**.

Probar: Ejecutar el comando **yes** y detener con **Ctrl+C**. Luego ejecutar con redirección (**yes >/dev/null**) y:

- ▶ Suspender, **Ctrl+Z**
- ▶ Enviar a 2do plano, **bg**; y a 1er plano, **fg**
- ▶ Detener, **Ctrl+C**

Procesos demonio – daemon

Un demonio (daemon) es un proceso de propósitos especiales creado para tareas de administración del sistema. Se distingue por las siguientes características:

- ▶ Arranca cuando se inicia el sistema (buteo) y permanece en existencia hasta que se apaga
- ▶ Corre en segundo plano (background) y no tiene terminal de control desde la cual se puede leer entrada o sobre la cual mostrar la salida

Procesos demonio – daemon

Un demonio (daemon) es un proceso de propósitos especiales creado para tareas de administración del sistema. Se distingue por las siguientes características:

- ▶ Arranca cuando se inicia el sistema (buteo) y permanece en existencia hasta que se apaga
- ▶ Corre en segundo plano (background) y no tiene terminal de control desde la cual se puede leer entrada o sobre la cual mostrar la salida

Algunos ejemplos de procesos “*demonios*” son:

- ▶ **syslogd**: guarda logs (registro) del sistema
- ▶ **httpd**: servidor de páginas web
- ▶ **sshd**: servidor de SSH (Secure Shell)
- ▶ **cupsd**: sistema de impresión de UNIX
- ▶ Etc.

Señales

Es un mecanismo de comunicación entre procesos, IPC (Interprocess Communication)

Señales

Es un mecanismo de comunicación entre procesos, IPC (Interprocess Communication)

- ▶ Es un evento generado por un sistema UNIX o Linux

Señales

Es un mecanismo de comunicación entre procesos, IPC (Interprocess Communication)

- ▶ Es un evento generado por un sistema UNIX o Linux
- ▶ Se describen a menudo como *interrupciones de software*

Señales

Es un mecanismo de comunicación entre procesos, IPC (Interprocess Communication)

- ▶ Es un evento generado por un sistema UNIX o Linux
- ▶ Se describen a menudo como *interrupciones de software*
- ▶ El arribo de una señal le indica al proceso que ha ocurrido un evento o condición excepcional

Señales

Es un mecanismo de comunicación entre procesos, IPC (Interprocess Communication)

- ▶ Es un evento generado por un sistema UNIX o Linux
- ▶ Se describen a menudo como *interrupciones de software*
- ▶ El arribo de una señal le indica al proceso que ha ocurrido un evento o condición excepcional
- ▶ Hay varios tipos de señales, cada una de las cuales identifican diferentes eventos o condiciones

Señales

Es un mecanismo de comunicación entre procesos, IPC (Interprocess Communication)

- ▶ Es un evento generado por un sistema UNIX o Linux
- ▶ Se describen a menudo como *interrupciones de software*
- ▶ El arribo de una señal le indica al proceso que ha ocurrido un evento o condición excepcional
- ▶ Hay varios tipos de señales, cada una de las cuales identifican diferentes eventos o condiciones
- ▶ Ejemplo de señal: *caracter de interrupción* Ctrl-C

Señales

Es un mecanismo de comunicación entre procesos, IPC (Interprocess Communication)

- ▶ Es un evento generado por un sistema UNIX o Linux
- ▶ Se describen a menudo como *interrupciones de software*
- ▶ El arribo de una señal le indica al proceso que ha ocurrido un evento o condición excepcional
- ▶ Hay varios tipos de señales, cada una de las cuales identifican diferentes eventos o condiciones
- ▶ Ejemplo de señal: *caracter de interrupción* Ctrl-C

Ver comando `kill` (`man 7 signal`)

Señales

Es un mecanismo de comunicación entre procesos, IPC (Interprocess Communication)

- ▶ Es un evento generado por un sistema UNIX o Linux
- ▶ Se describen a menudo como *interrupciones de software*
- ▶ El arribo de una señal le indica al proceso que ha ocurrido un evento o condición excepcional
- ▶ Hay varios tipos de señales, cada una de las cuales identifican diferentes eventos o condiciones
- ▶ Ejemplo de señal: *caracter de interrupción* Ctrl-C

Ver comando **kill** (`man 7 signal`)

Terminar/matar un proceso:

1. Identificar el proceso utilizando el comando **ps**
2. Enviarle una señal con el comando **kill**

Señales en lenguaje C

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main(void)
5 {
6     printf("Iniciando bucle...\n");
7     while(1)
8     {
9         printf("Corriendo.\n");
10        sleep(1);
11    }
12    printf("Terminando bucle...\n");
13
14    return 0;
15 }
```

Señales en lenguaje C

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4
5 unsigned int flag = 1;
6 void handler(int );
7
8 int main(void) {
9
10    printf("Iniciando bucle...\n");
11    while(flag)
12    {
13        printf("Corriendo.\n");
14        sleep(1);
15    }
16    printf("Terminando bucle...\n");
17
18    return 0;
19 }
20
21 void handler(int sig)
22 { flag = 0; }
```

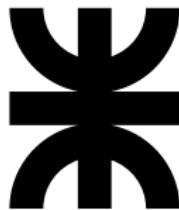
Señales en lenguaje C

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <signal.h>
4
5 unsigned int flag = 1;
6 void handler(int );
7
8 int main(void) {
9     signal(SIGINT, handler);
10    printf("Iniciando bucle...\n");
11    while(flag)
12    {
13        printf("Corriendo.\n");
14        sleep(1);
15    }
16    printf("Terminando bucle...\n");
17
18    return 0;
19 }
20
21 void handler(int sig) // manejador de la señal
22 { flag = 0; }
```

Informática II

Relación entre punteros y arreglos

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2021 –

Repaso sobre arreglos

Definición de arreglo

- ▶ Estructura de dato que consiste en elementos relacionados del mismo tipo
- ▶ Colección de datos bajo un mismo nombre

Repaso sobre arreglos

Definición de arreglo

- ▶ Estructura de dato que consiste en elementos relacionados del mismo tipo
- ▶ Colección de datos bajo un mismo nombre

Desde un punto de vista de bajo nivel:

- ▶ Es un grupo de posiciones de memoria consecutivas relacionadas entre sí
- ▶ Todas del mismo nombre y del mismo tipo

Repaso sobre arreglos

Definición de arreglo

- ▶ Estructura de dato que consiste en elementos relacionados del mismo tipo
- ▶ Colección de datos bajo un mismo nombre

Desde un punto de vista de bajo nivel:

- ▶ Es un grupo de posiciones de memoria consecutivas relacionadas entre sí
- ▶ Todas del mismo nombre y del mismo tipo

Por ejemplo:

```
int a = 10; /* Definición de un entero */  
int v[3]; /* Definición de un arreglo de enteros */
```

Repaso sobre arreglos

Definición de arreglo

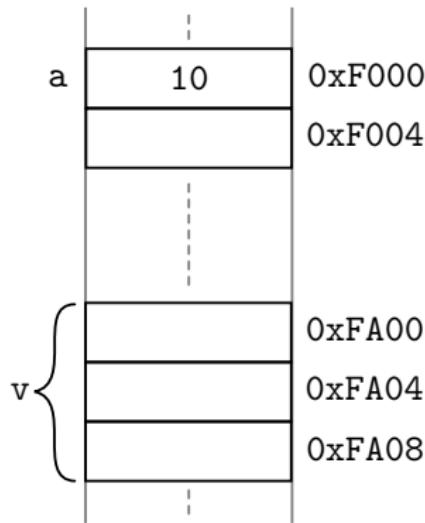
- ▶ Estructura de dato que consiste en elementos relacionados del mismo tipo
- ▶ Colección de datos bajo un mismo nombre

Desde un punto de vista de bajo nivel:

- ▶ Es un grupo de posiciones de memoria consecutivas relacionadas entre sí
- ▶ Todas del mismo nombre y del mismo tipo

Por ejemplo:

```
int a = 10; /* Definición de un entero */  
int v[3]; /* Definición de un arreglo de enteros */
```



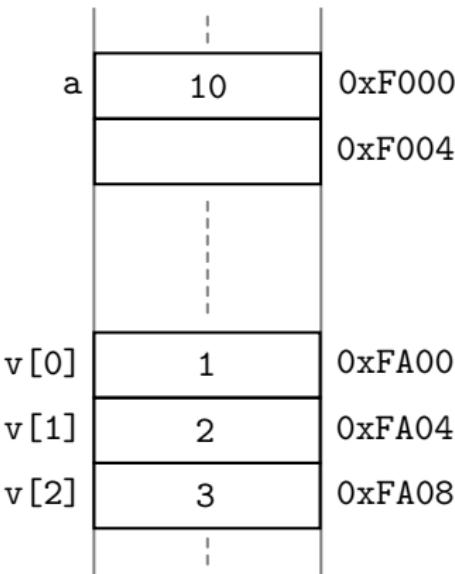
Repaso sobre arreglos – ejemplo

- ▶ ¿Cómo se inicializa un arreglo?
- ▶ ¿Cómo acceder a un elemento particular?

Repaso sobre arreglos – ejemplo

- ▶ ¿Cómo se inicializa un arreglo?
- ▶ ¿Cómo acceder a un elemento particular?

```
1 #include <stdio.h>
2
3 #define TAM 3
4
5 int main(void)
6 {
7     int a = 10;
8     int v[TAM] = {1, 2, 3}; /* arreglo */
9     int i; /* subíndice */
10
11    for(i = 0; i < TAM; i++)
12        printf("%d\n", v[i]);
13
14    return 0;
15 }
```



Punteros

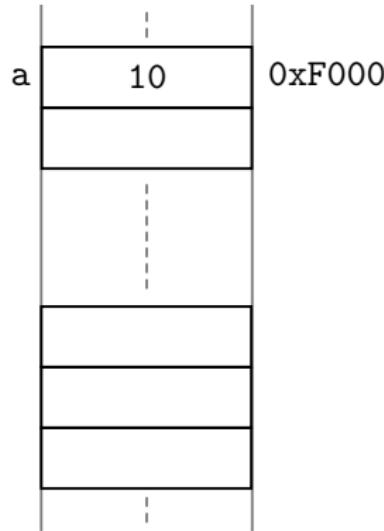
Definición

Son variables cuyos valores representan posiciones de memoria

Punteros

Definición

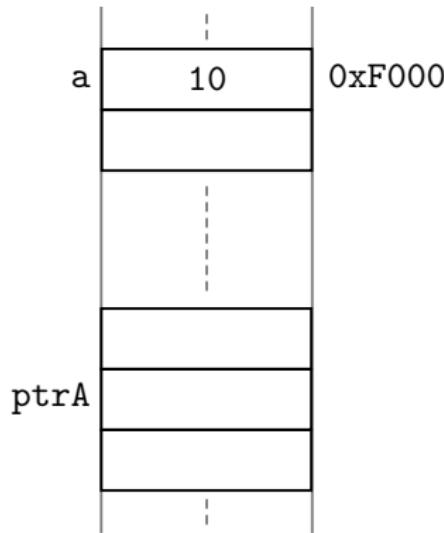
Son variables cuyos valores representan posiciones de memoria



Punteros

Definición

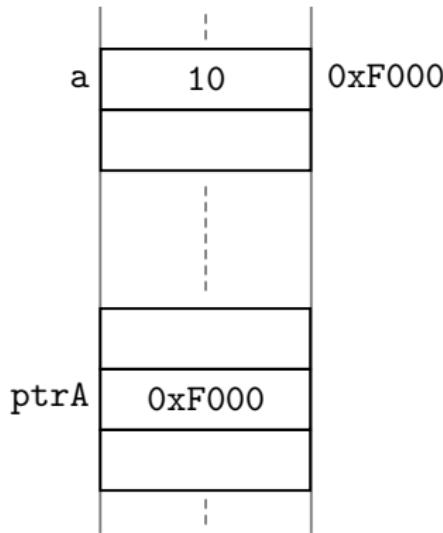
Son variables cuyos valores representan posiciones de memoria



Punteros

Definición

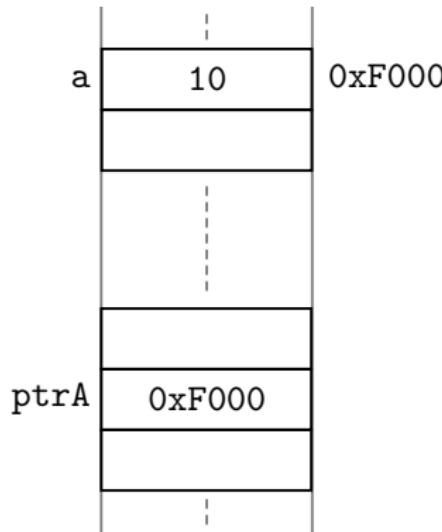
Son variables cuyos valores representan posiciones de memoria



Punteros

Definición

Son variables cuyos valores representan posiciones de memoria



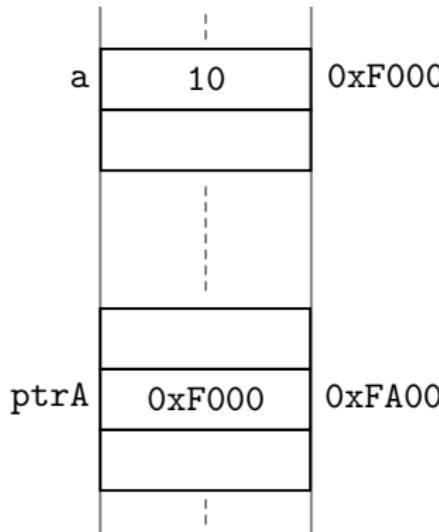
Por ejemplo:

```
int a = 10; /* Definición de un entero */  
int *ptrA = &a; /* Definición de un puntero */
```

Punteros

Definición

Son variables cuyos valores representan posiciones de memoria



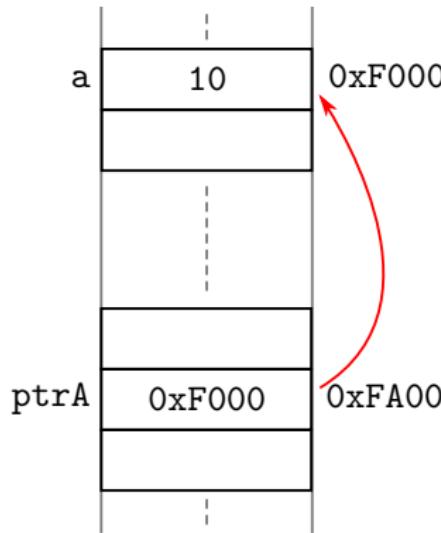
Por ejemplo:

```
int a = 10; /* Definición de un entero */  
int *ptrA = &a; /* Definición de un puntero */
```

Punteros

Definición

Son variables cuyos valores representan posiciones de memoria



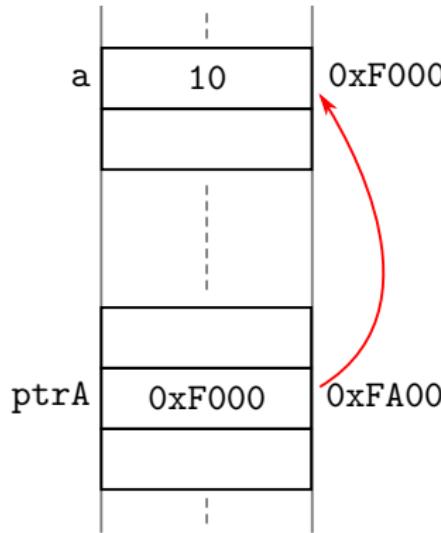
Por ejemplo:

```
int a = 10; /* Definición de un entero */  
int *ptrA = &a; /* Definición de un puntero */
```

Punteros

Definición

Son variables cuyos valores representan posiciones de memoria



Por ejemplo:

```
int a = 10; /* Definición de un entero */  
int *ptrA = &a; /* Definición de un puntero */
```

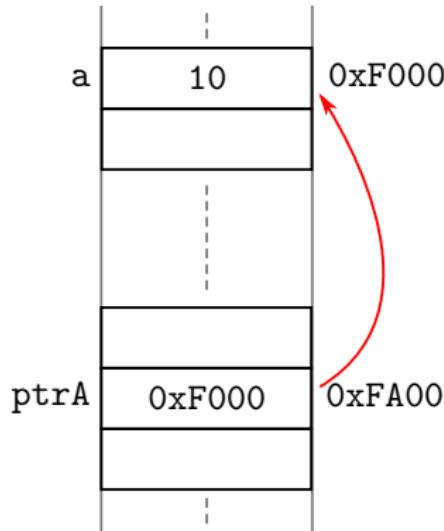
O sea:

- ▶ Una variable hace referencia a un valor de *forma directa*

Punteros

Definición

Son variables cuyos valores representan posiciones de memoria



Por ejemplo:

```
int a = 10; /* Definición de un entero */  
int *ptrA = &a; /* Definición de un puntero */
```

O sea:

- ▶ Una variable hace referencia a un valor de *forma directa*
- ▶ Un puntero hace referencia a un valor de *forma indirecta*

Punteros – ejemplo

```
1 #include <stdio.h>
2
3 int main(void) {
4     int a; /* variable tipo int */
5     int *ptrA; /* puntero a variable tipo int */
6
7
8
9
10
11
12
13
14
15     return 0;
16 }
```

Punteros – ejemplo

```
1 #include <stdio.h>
2
3 int main(void) {
4     int a; /* variable tipo int */
5     int *ptrA; /* puntero a variable tipo int */
6
7     a = 10;
8     ptrA = &a; /* ptrA toma la dirección de a */
9
10
11
12
13
14
15     return 0;
16 }
```

Punteros – ejemplo

```
1 #include <stdio.h>
2
3 int main(void) {
4     int a; /* variable tipo int */
5     int *ptrA; /* puntero a variable tipo int */
6
7     a = 10;
8     ptrA = &a; /* ptrA toma la dirección de a */
9
10    printf("La dirección de a es %p\n", &a);
11    printf("El valor de ptrA es %p\n", ptrA);
12
13
14
15    return 0;
16 }
```

Punteros – ejemplo

```
1 #include <stdio.h>
2
3 int main(void) {
4     int a; /* variable tipo int */
5     int *ptrA; /* puntero a variable tipo int */
6
7     a = 10;
8     ptrA = &a; /* ptrA toma la dirección de a */
9
10    printf("La dirección de a es %p\n", &a);
11    printf("El valor de ptrA es %p\n", ptrA);
12
13
14
15    return 0;
16 }
```

```
La dirección de a es 0x7ffe4271501c
El valor de &a es 0x7ffe4271501c
```

Punteros – ejemplo

```
1 #include <stdio.h>
2
3 int main(void) {
4     int a; /* variable tipo int */
5     int *ptrA; /* puntero a variable tipo int */
6
7     a = 10;
8     ptrA = &a; /* ptrA toma la dirección de a */
9
10    printf("La dirección de a es %p\n", &a);
11    printf("El valor de ptrA es %p\n", ptrA);
12
13    printf("\nEl valor de a es %d\n", a);
14    printf("El valor de *ptrA es %d\n", *ptrA);
15    return 0;
16 }
```

```
La dirección de a es 0x7ffe4271501c
El valor de &a es 0x7ffe4271501c
```

Punteros – ejemplo

```
1 #include <stdio.h>
2
3 int main(void) {
4     int a; /* variable tipo int */
5     int *ptrA; /* puntero a variable tipo int */
6
7     a = 10;
8     ptrA = &a; /* ptrA toma la dirección de a */
9
10    printf("La dirección de a es %p\n", &a);
11    printf("El valor de ptrA es %p\n", ptrA);
12
13    printf("\nEl valor de a es %d\n", a);
14    printf("El valor de *ptrA es %d\n", *ptrA);
15    return 0;
16 }
```

```
La dirección de a es 0x7ffe4271501c
El valor de &a es 0x7ffe4271501c

El valor de a es 10
El valor de *ptrA es 10
```

Más sobre punteros

Operadores

- ▶ &: operador de dirección u operador de referencia
- ▶ *: operador de indirección u operador de desreferencia

Más sobre punteros

Operadores

- ▶ &: operador de dirección u operador de referencia
- ▶ *: operador de indirección u operador de desreferencia

(ambos son operadores unarios)

Más sobre punteros

Operadores

- ▶ &: operador de dirección u operador de referencia
- ▶ *: operador de indirección u operador de desreferencia

(ambos son operadores unarios)

Otras consideraciones:

- ▶ Los punteros puede apuntar a objetos de cualquier tipo

Más sobre punteros

Operadores

- ▶ &: operador de dirección u operador de referencia
- ▶ *: operador de indirección u operador de desreferencia

(ambos son operadores unarios)

Otras consideraciones:

- ▶ Los punteros puede apuntar a objetos de cualquier tipo
- ▶ Se deben inicializar en la declaración o mediante una asignación

Más sobre punteros

Operadores

- ▶ &: operador de dirección u operador de referencia
- ▶ *: operador de indirección u operador de desreferencia

(ambos son operadores unarios)

Otras consideraciones:

- ▶ Los punteros puede apuntar a objetos de cualquier tipo
- ▶ Se deben inicializar en la declaración o mediante una asignación
- ▶ Se puede inicializar a NULL o a una dirección válida

Más sobre punteros

Operadores

- ▶ `&`: operador de dirección u operador de referencia
- ▶ `*`: operador de indirección u operador de desreferencia

(ambos son operadores unarios)

Otras consideraciones:

- ▶ Los punteros puede apuntar a objetos de cualquier tipo
- ▶ Se deben inicializar en la declaración o mediante una asignación
- ▶ Se puede inicializar a `NULL` o a una dirección válida
- ▶ Un puntero a `NULL` no apunta a ningún lado

Más sobre punteros

Operadores

- ▶ `&`: operador de dirección u operador de referencia
- ▶ `*`: operador de indirección u operador de desreferencia

(ambos son operadores unarios)

Otras consideraciones:

- ▶ Los punteros puede apuntar a objetos de cualquier tipo
- ▶ Se deben inicializar en la declaración o mediante una asignación
- ▶ Se puede inicializar a `NULL` o a una dirección válida
- ▶ Un puntero a `NULL` no apunta a ningún lado

Los *punteros* son una de las características más poderosas del lenguaje C.

Más sobre punteros

Operadores

- ▶ &: operador de dirección u operador de referencia
- ▶ *: operador de indirección u operador de desreferencia

(ambos son operadores unarios)

Otras consideraciones:

- ▶ Los punteros puede apuntar a objetos de cualquier tipo
- ▶ Se deben inicializar en la declaración o mediante una asignación
- ▶ Se puede inicializar a NULL o a una dirección válida
- ▶ Un puntero a NULL no apunta a ningún lado

Los *punteros* son una de las características más poderosas del lenguaje C.

Sirven para:

- ▶ Simular llamadas de funciones por referencias
- ▶ Crear y manipular estructuras de datos dinámicas

El operador `sizeof`

Es un operador unario que sirve para determinar el tamaño de un arreglo u otro tipo de dato. Devuelve el tamaño en bytes (cantidad de bytes).

El operador `sizeof`

Es un operador unario que sirve para determinar el tamaño de un arreglo u otro tipo de dato. Devuelve el tamaño en bytes (cantidad de bytes).

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int a = 10;
6
7
8     printf("Tamaño de a: %lu\n", sizeof a);
9
10
11
12     return 0;
13 }
```

El operador `sizeof`

Es un operador unario que sirve para determinar el tamaño de un arreglo u otro tipo de dato. Devuelve el tamaño en bytes (cantidad de bytes).

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int a = 10;
6     int v[] = {1, 2, 3, 4, 5};
7
8     printf("Tamaño de a: %lu\n", sizeof a);
9     printf("Tamaño del vector: %lu\n", sizeof v);
10
11
12     return 0;
13 }
```

El operador `sizeof`

Es un operador unario que sirve para determinar el tamaño de un arreglo u otro tipo de dato. Devuelve el tamaño en bytes (cantidad de bytes).

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int a = 10;
6     int v[] = {1, 2, 3, 4, 5};
7
8     printf("Tamaño de a: %lu\n", sizeof a);
9     printf("Tamaño del vector: %lu\n", sizeof v);
10    printf("Tamaño del tipo 'double': %lu\n", sizeof(double));
11
12    return 0;
13 }
```

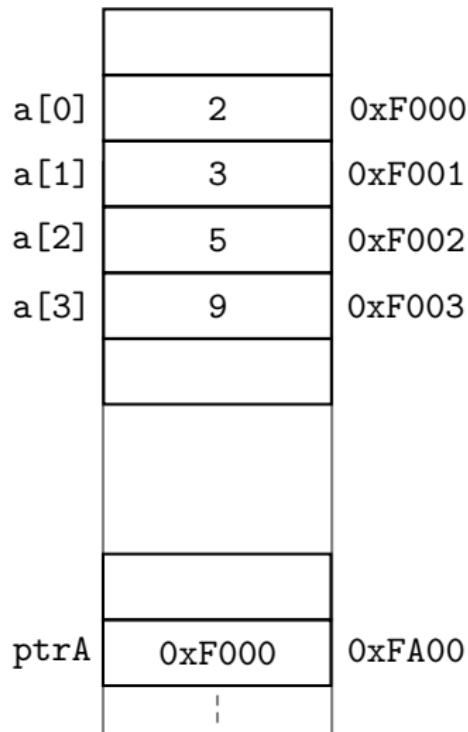
El operador `sizeof`

Es un operador unario que sirve para determinar el tamaño de un arreglo u otro tipo de dato. Devuelve el tamaño en bytes (cantidad de bytes).

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int a = 10;
6     int v[] = {1, 2, 3, 4, 5};
7
8     printf("Tamaño de a: %lu\n", sizeof a);
9     printf("Tamaño del vector: %lu\n", sizeof v);
10    printf("Tamaño del tipo 'double': %lu\n", sizeof(double));
11
12    return 0;
13 }
```

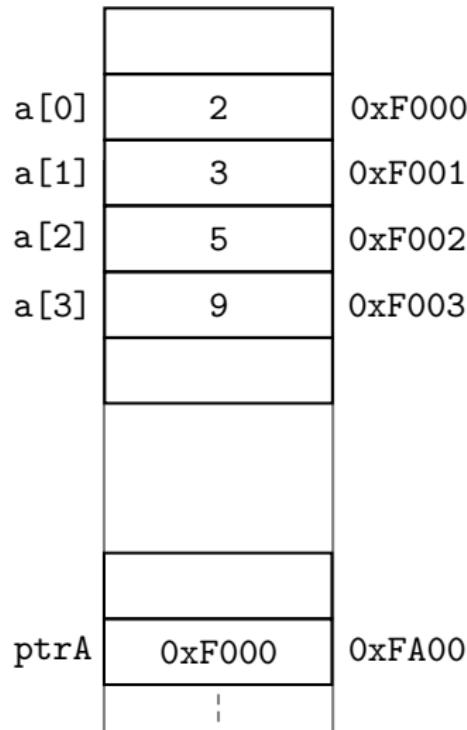
```
Tamaño de a: 4
Tamaño del vector: 20
Tamaño del tipo 'double': 8
```

Aritmética sobre punteros



Aritmética sobre punteros

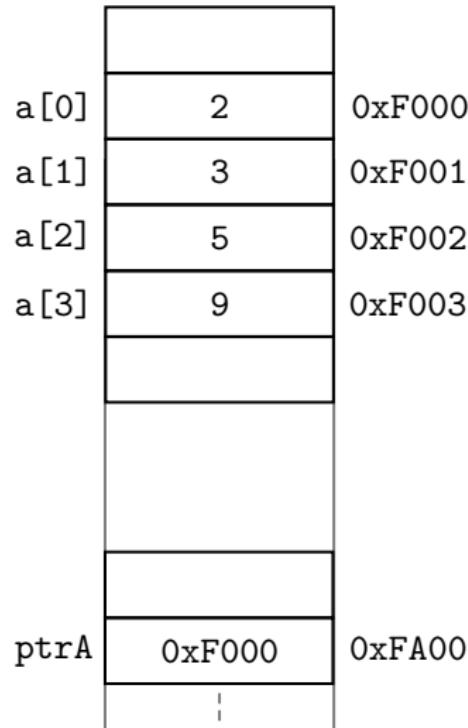
```
1 #include <stdio.h>
2 #define TAM 4
3
4 int main(void)
5 {
6     char a[TAM] = {2, 3, 5, 9};
7     char *ptrA;
8
9     ptrA = &a[0];
10
11
12     printf("%d\n", *ptrA);
13
14     return 0;
15 }
```



Aritmética sobre punteros

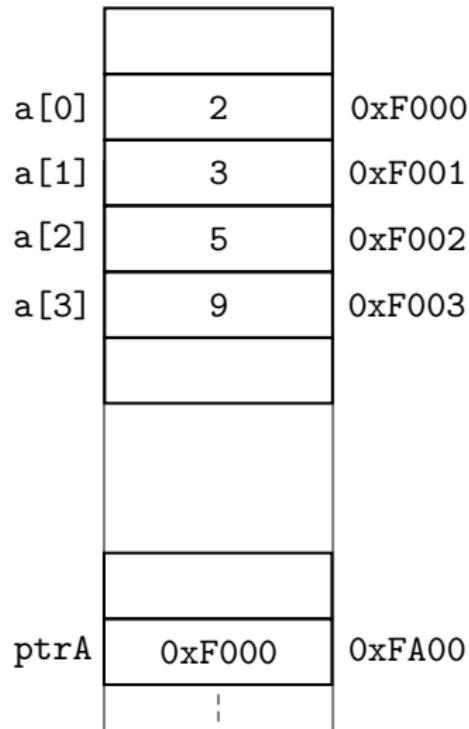
```
1 #include <stdio.h>
2 #define TAM 4
3
4 int main(void)
5 {
6     char a[TAM] = {2, 3, 5, 9};
7     char *ptrA;
8
9     ptrA = &a[0];
10
11
12     printf("%d\n", *ptrA);
13
14     return 0;
15 }
```

2



Aritmética sobre punteros

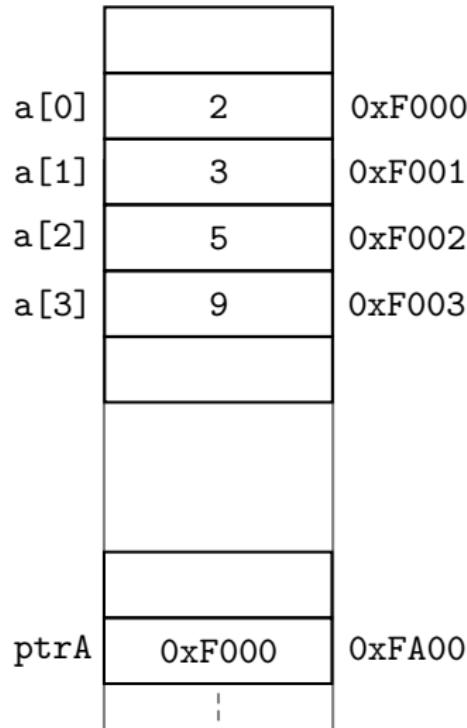
```
1 #include <stdio.h>
2 #define TAM 4
3
4 int main(void)
5 {
6     char a[TAM] = {2, 3, 5, 9};
7     char *ptrA;
8
9     ptrA = &a[0];
10    ptrA = ptrA + 1; /* ptrA += 1 */
11
12    printf("%d\n", *ptrA);
13
14    return 0;
15 }
```



Aritmética sobre punteros

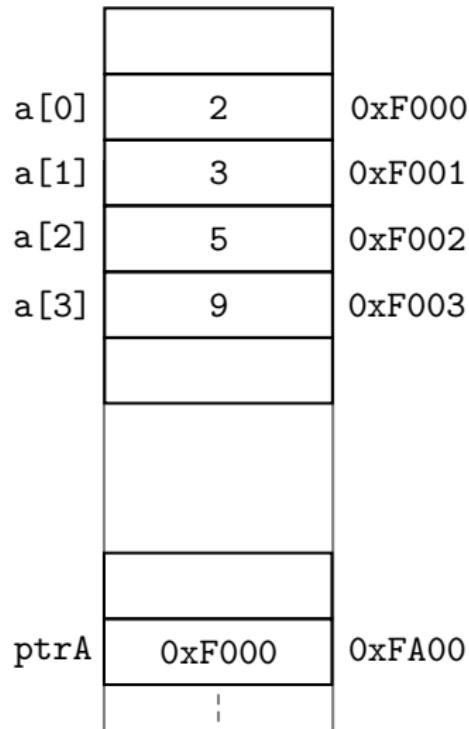
```
1 #include <stdio.h>
2 #define TAM 4
3
4 int main(void)
5 {
6     char a[TAM] = {2, 3, 5, 9};
7     char *ptrA;
8
9     ptrA = &a[0];
10    ptrA = ptrA + 1; /* ptrA += 1 */
11
12    printf("%d\n", *ptrA);
13
14    return 0;
15 }
```

3



Aritmética sobre punteros

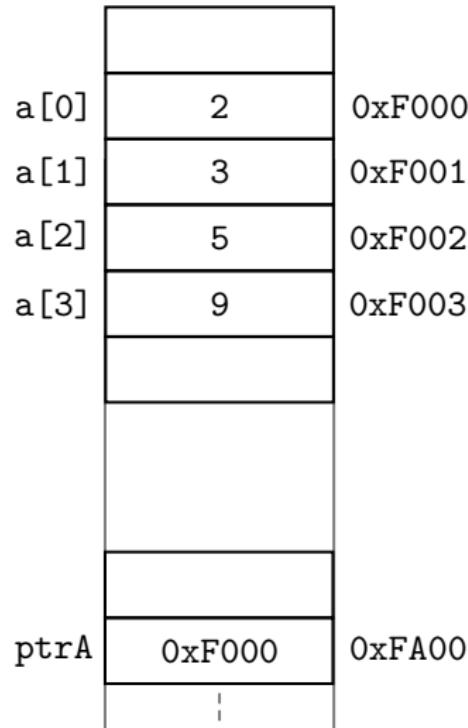
```
1 #include <stdio.h>
2 #define TAM 4
3
4 int main(void)
5 {
6     char a[TAM] = {2, 3, 5, 9};
7     char *ptrA;
8
9     ptrA = &a[0];
10
11
12     printf("%d\n", *(ptrA + 2));
13
14     return 0;
15 }
```



Aritmética sobre punteros

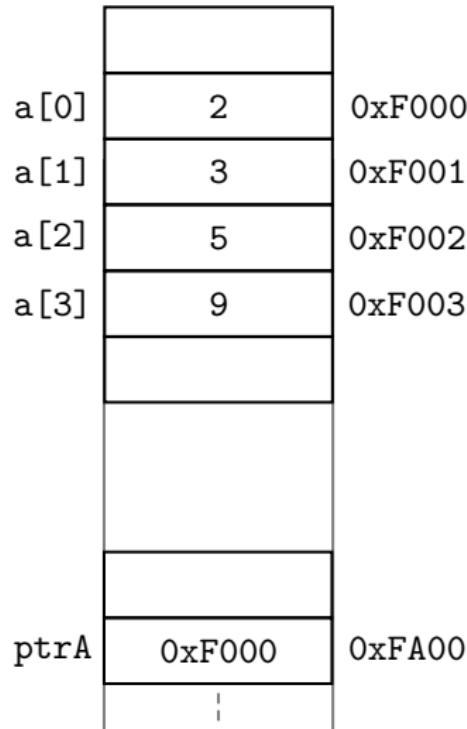
```
1 #include <stdio.h>
2 #define TAM 4
3
4 int main(void)
5 {
6     char a[TAM] = {2, 3, 5, 9};
7     char *ptrA;
8
9     ptrA = &a[0];
10
11
12     printf("%d\n", *(ptrA + 2));
13
14     return 0;
15 }
```

5



Aritmética sobre punteros

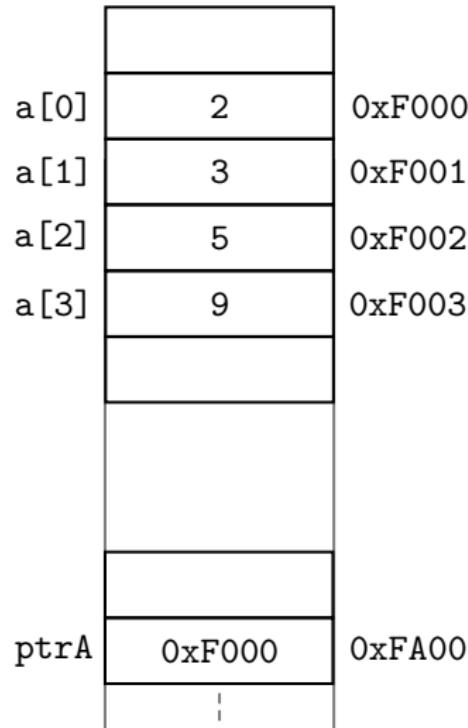
```
1 #include <stdio.h>
2 #define TAM 4
3
4 int main(void)
5 {
6     char a[TAM] = {2, 3, 5, 9};
7     char *ptrA;
8
9     ptrA = &a[0];
10
11
12     printf("%d\n", *ptrA + 2);
13
14     return 0;
15 }
```



Aritmética sobre punteros

```
1 #include <stdio.h>
2 #define TAM 4
3
4 int main(void)
5 {
6     char a[TAM] = {2, 3, 5, 9};
7     char *ptrA;
8
9     ptrA = &a[0];
10
11
12     printf("%d\n", *ptrA + 2);
13
14     return 0;
15 }
```

4



Aritmética sobre punteros

```
1 #include <stdio.h>
2 #define TAM 8
3
4 int main(void)
5 {
6     int i; /* índice */
7     char v[TAM] = {0};
8     char *ptrV = &v[0];
9
10    for(i = 0; i < TAM; i++)
11        printf("La dirección del elemento %d es %p\n", i, (ptrV + i));
12
13    return 0;
14 }
```

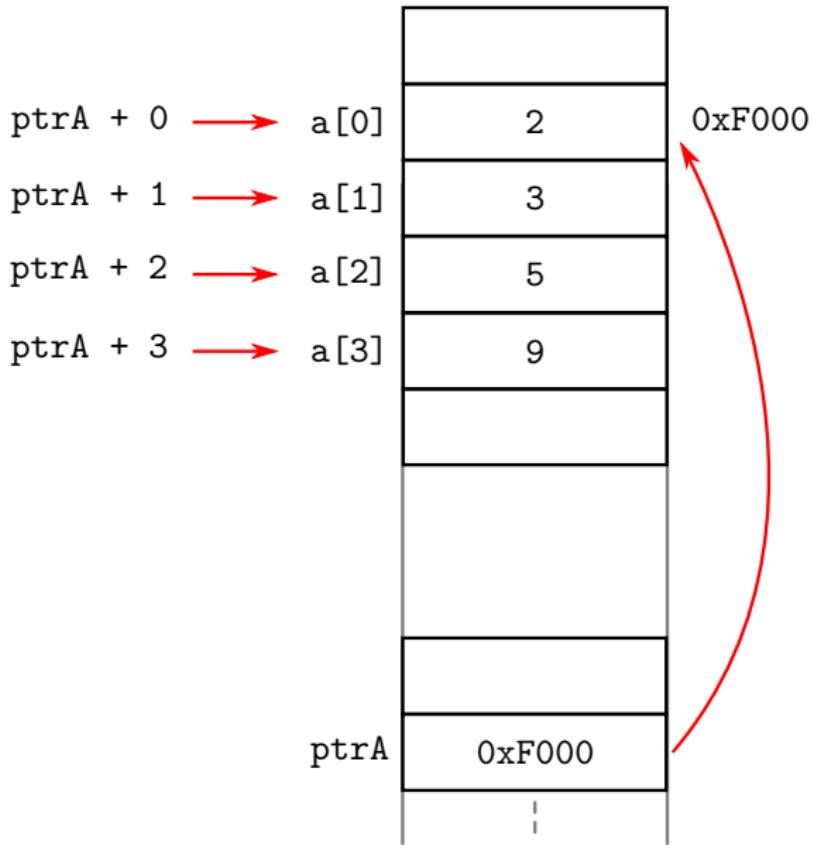
```
La dirección del elemento 0 es 0x7ffccbf01aa0
La dirección del elemento 1 es 0x7ffccbf01aa1
La dirección del elemento 2 es 0x7ffccbf01aa2
La dirección del elemento 3 es 0x7ffccbf01aa3
La dirección del elemento 4 es 0x7ffccbf01aa4
La dirección del elemento 5 es 0x7ffccbf01aa5
La dirección del elemento 6 es 0x7ffccbf01aa6
La dirección del elemento 7 es 0x7ffccbf01aa7
```

Aritmética sobre punteros

```
1 #include <stdio.h>
2 #define TAM 8
3
4 int main(void)
5 {
6     int i; /* índice */
7     int v[TAM] = {0};
8     int *ptrV = &v[0];
9
10    for(i = 0; i < TAM; i++)
11        printf("La dirección del elemento %d es %p\n", i, (ptrV + i));
12
13    return 0;
14 }
```

```
La dirección del elemento 0 es 0x7ffe227a2710
La dirección del elemento 1 es 0x7ffe227a2714
La dirección del elemento 2 es 0x7ffe227a2718
La dirección del elemento 3 es 0x7ffe227a271c
La dirección del elemento 4 es 0x7ffe227a2720
La dirección del elemento 5 es 0x7ffe227a2724
La dirección del elemento 6 es 0x7ffe227a2728
La dirección del elemento 7 es 0x7ffe227a272c
```

Aritmética sobre punteros



Relación entre punteros y arreglos

- En el lenguaje C los arreglos y punteros están íntimamente relacionados

Relación entre punteros y arreglos

- ▶ En el lenguaje C los arreglos y punteros están íntimamente relacionados
- ▶ En algunos casos se pueden utilizar de manera indistinta

Relación entre punteros y arreglos

- ▶ En el lenguaje C los arreglos y punteros están íntimamente relacionados
- ▶ En algunos casos se pueden utilizar de manera indistinta
- ▶ Un nombre de arreglo puede interpretarse como un puntero constante

Relación entre punteros y arreglos

- ▶ En el lenguaje C los arreglos y punteros están íntimamente relacionados
- ▶ En algunos casos se pueden utilizar de manera indistinta
- ▶ Un nombre de arreglo puede interpretarse como un puntero constante

Por ejemplo:

```
int a[5] = {1, 2, 3, 4, 5};  
int *ptrA = &a[0];
```

Relación entre punteros y arreglos

- ▶ En el lenguaje C los arreglos y punteros están íntimamente relacionados
- ▶ En algunos casos se pueden utilizar de manera indistinta
- ▶ Un nombre de arreglo puede interpretarse como un puntero constante

Por ejemplo:

```
int a[5] = {1, 2, 3, 4, 5};  
int *ptrA = &a[0];
```

Es equivalente a:

```
int a[5] = {1, 2, 3, 4, 5};  
int *ptrA = a;
```

Relación entre punteros y arreglos

- ▶ En el lenguaje C los arreglos y punteros están íntimamente relacionados
- ▶ En algunos casos se pueden utilizar de manera indistinta
- ▶ Un nombre de arreglo puede interpretarse como un puntero constante
- ▶ Se puede utilizar los punteros para operaciones con subíndices de arreglos

Por ejemplo:

```
int a[5] = {1, 2, 3, 4, 5};  
  
int *ptrA = &a[0];
```

Es equivalente a:

```
int a[5] = {1, 2, 3, 4, 5};  
  
int *ptrA = a;
```

Relación entre punteros y arreglos

- ▶ En el lenguaje C los arreglos y punteros están íntimamente relacionados
- ▶ En algunos casos se pueden utilizar de manera indistinta
- ▶ Un nombre de arreglo puede interpretarse como un puntero constante
- ▶ Se puede utilizar los punteros para operaciones con subíndices de arreglos

Por ejemplo:

```
int a[5] = {1, 2, 3, 4, 5};  
int *ptrA = &a[0];
```

```
printf("%d", a[3]);  
printf("%d", *(ptrA + 3));
```

Es equivalente a:

```
int a[5] = {1, 2, 3, 4, 5};  
int *ptrA = a;
```

Relación entre punteros y arreglos

- ▶ En el lenguaje C los arreglos y punteros están íntimamente relacionados
- ▶ En algunos casos se pueden utilizar de manera indistinta
- ▶ Un nombre de arreglo puede interpretarse como un puntero constante
- ▶ Se puede utilizar los punteros para operaciones con subíndices de arreglos

Por ejemplo:

```
int a[5] = {1, 2, 3, 4, 5};  
int *ptrA = &a[0];
```

```
printf("%d", a[3]);  
printf("%d", *(ptrA + 3));
```

Es equivalente a:

```
int a[5] = {1, 2, 3, 4, 5};  
int *ptrA = a;
```

```
printf("%d", ptrA[3]);  
printf("%d", *(a + 3));
```

Relación entre punteros y arreglos

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int a[] = {1, 2, 3, 4, 5};
6     int *ptrA = a;
7
8     printf("a[3] : %d\n", a[3]);
9     printf(*(ptrA + 3) : %d\n", *(ptrA + 3));
10    printf("ptrA[3] : %d\n", ptrA[3]);
11    printf(*(a + 3) : %d\n", *(a + 3));
12
13    return 0;
14 }
```

```
a [3]      : 4
*(ptrA+3)  : 4
ptrA [3]   : 4
*(ptrA+3)  : 4
```

Relación entre punteros y arreglos

`a[i]`: Notación subíndice de arreglos

Relación entre punteros y arreglos

`a[i]`: Notación subíndice de arreglos

`*(ptrA+i)`: Notación puntero/desplazamiento

Relación entre punteros y arreglos

`a[i]`: Notación subíndice de arreglos

`*(ptrA+i)`: Notación puntero/desplazamiento

`ptrA[i]`: Notación puntero/subíndice

Relación entre punteros y arreglos

`a[i]`: Notación subíndice de arreglos

`*(ptrA+i)`: Notación puntero/desplazamiento

`ptrA[i]`: Notación puntero/subíndice

`*(a+i)`: Notación puntero/desplazamiento donde el puntero es el nombre del arreglo

Ejemplo: tamaño de un arreglo/puntero

```
1 #include <stdio.h>
2 #define TAM 20
3
4 void imprimirTamanio(float v[])
5 {
6     printf("Tamaño del arreglo: %lu\n", sizeof(v));
7 }
8
9 int main(void)
10 {
11
12
13
14
15
16     return 0;
17 }
```

Ejemplo: tamaño de un arreglo/puntero

```
1 #include <stdio.h>
2 #define TAM 20
3
4 void imprimirTamanio(float v[])
5 {
6     printf("Tamaño del arreglo: %lu\n", sizeof(v));
7 }
8
9 int main(void)
10 {
11     float arreglo[TAM]; /* crea arreglo */
12
13     printf("Tamaño del arreglo: %lu\n", sizeof(arreglo));
14
15
16     return 0;
17 }
```

Ejemplo: tamaño de un arreglo/puntero

```
1 #include <stdio.h>
2 #define TAM 20
3
4 void imprimirTamanio(float v[])
5 {
6     printf("Tamaño del arreglo: %lu\n", sizeof(v));
7 }
8
9 int main(void)
10 {
11     float arreglo[TAM]; /* crea arreglo */
12
13     printf("Tamaño del arreglo: %lu\n", sizeof(arreglo));
14
15
16     return 0;
17 }
```

Tamaño del arreglo: 80

Ejemplo: tamaño de un arreglo/puntero

```
1 #include <stdio.h>
2 #define TAM 20
3
4 void imprimirTamanio(float v[])
5 {
6     printf("Tamaño del arreglo: %lu\n", sizeof(v));
7 }
8
9 int main(void)
10 {
11     float arreglo[TAM]; /* crea arreglo */
12
13     printf("Tamaño del arreglo: %lu\n", sizeof(arreglo));
14     imprimirTamanio(arreglo);
15
16     return 0;
17 }
```

Tamaño del arreglo: 80

Ejemplo: tamaño de un arreglo/puntero

```
1 #include <stdio.h>
2 #define TAM 20
3
4 void imprimirTamanio(float v[])
5 {
6     printf("Tamaño del arreglo: %lu\n", sizeof(v));
7 }
8
9 int main(void)
10 {
11     float arreglo[TAM]; /* crea arreglo */
12
13     printf("Tamaño del arreglo: %lu\n", sizeof(arreglo));
14     imprimirTamanio(arreglo);
15
16     return 0;
17 }
```

```
Tamaño del arreglo: 80
Tamaño del arreglo: 8
```

Ejemplo: tamaño de un arreglo/puntero

```
1 #include <stdio.h>
2 #define TAM 20
3
4 void imprimirTamanio(float v[TAM])
5 {
6     printf("Tamaño del arreglo: %lu\n", sizeof(v));
7 }
8
9 int main(void)
10 {
11     float arreglo[TAM]; /* crea arreglo */
12
13     printf("Tamaño del arreglo: %lu\n", sizeof(arreglo));
14     imprimirTamanio(arreglo);
15
16     return 0;
17 }
```

```
Tamaño del arreglo: 80
Tamaño del arreglo: 8
```

Ejemplo: función para imprimir un arreglo

```
1 #include <stdio.h>
2 #define TAM 20
3
4 void imprimirArreglo(int v[], int n)
5 {
6     int i; /* subíndice */
7
8     for(i = 0; i < n; i++)
9         printf("%d ", v[i]);
10    printf("\n");
11 }
12
13 int main(void)
14 {
15     int v[TAM] = {1, 2, 3, 4, 5};
16
17     imprimirArreglo(v, 10);
18     return 0;
19 }
```

1 2 3 4 5 0 0 0 0 0

Ejemplo: función para imprimir un arreglo

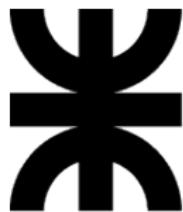
```
1 #include <stdio.h>
2 #define TAM 20
3
4 void imprimirArreglo(int v[], int n)
5 {
6     int i; /* subíndice */
7
8     for(i = 0; i < n; i++)
9         printf("%d ", v[i]);
10    printf("- Tamaño: %d\n", sizeof(v));
11 }
12
13 int main(void)
14 {
15     int v[TAM] = {1, 2, 3, 4, 5};
16
17     imprimirArreglo(v, 10);
18     return 0;
19 }
```

```
1 2 3 4 5 0 0 0 0 - Tamaño: 8
```


Informática II

Caracteres y cadenas

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2021 –

Cadenas

- ▶ ¿Existe el tipo de datos *string* (cadena) en C?

Cadenas

- ▶ ¿Existe el tipo de datos *string* (cadena) en C?
- ▶ ¿Cómo se almacenan/tratan las cadenas en C?

Cadenas

- ▶ ¿Existe el tipo de datos *string* (cadena) en C?
- ▶ ¿Cómo se almacenan/tratan las cadenas en C?

```
char cadena1[] = {67, 97, 100, 49, 0};  
printf("%s", cadera1); // ¿Qué se imprime?
```

Cadenas

- ▶ ¿Existe el tipo de datos *string* (cadena) en C?
- ▶ ¿Cómo se almacenan/tratan las cadenas en C?

```
char cadena1[] = {67, 97, 100, 49, 0};  
printf("%s", cadera1); // Imprime "Cad1"
```

Cadenas

- ▶ ¿Existe el tipo de datos *string* (cadena) en C?
- ▶ ¿Cómo se almacenan/tratan las cadenas en C?

En lenguaje C las *cadenas* son arreglos de caracteres (ASCII)

Cadenas

- ▶ ¿Existe el tipo de datos *string* (cadena) en C?
- ▶ ¿Cómo se almacenan/tratan las cadenas en C?

En lenguaje C las *cadenas* son arreglos de caracteres (ASCII)

Ejemplos:

```
char cadena1[] = {'C', 'a', 'd', '1', '\0'};
```

Cadenas

- ▶ ¿Existe el tipo de datos *string* (cadena) en C?
- ▶ ¿Cómo se almacenan/tratan las cadenas en C?

En lenguaje C las *cadenas* son arreglos de caracteres (ASCII)

Ejemplos:

```
char cadena1[] = {'C', 'a', 'd', '1', '\0'};  
char cadena2[5] = {'C', 'a', 'd', '2', '\0'};
```

Cadenas

- ▶ ¿Existe el tipo de datos *string* (cadena) en C?
- ▶ ¿Cómo se almacenan/tratan las cadenas en C?

En lenguaje C las *cadenas* son arreglos de caracteres (ASCII)

Ejemplos:

```
char cadena1[] = {'C', 'a', 'd', '1', '\0'};  
char cadena2[5] = {'C', 'a', 'd', '2', '\0'};  
char cadena3[4] = {'C', 'a', 'd', '3', '\0'};
```

Cadenas

- ▶ ¿Existe el tipo de datos *string* (cadena) en C?
- ▶ ¿Cómo se almacenan/tratan las cadenas en C?

En lenguaje C las *cadenas* son arreglos de caracteres (ASCII)

Ejemplos:

```
char cadena1[] = {'C', 'a', 'd', '1', '\0'};  
char cadena2[5] = {'C', 'a', 'd', '2', '\0'};  
char cadena3[4] = {'C', 'a', 'd', '3', '\0'}; // ERROR
```

Cadenas

- ▶ ¿Existe el tipo de datos *string* (cadena) en C?
- ▶ ¿Cómo se almacenan/tratan las cadenas en C?

En lenguaje C las *cadenas* son arreglos de caracteres (ASCII)

Ejemplos:

```
char cadena1[] = {'C', 'a', 'd', '1', '\0'};  
char cadena2[5] = {'C', 'a', 'd', '2', '\0'};  
char cadena3[4] = {'C', 'a', 'd', '3', '\0'}; // ERROR  
char cadena4[] = {67, 97, 100, 52, 0}; // ASCII
```

Cadenas

- ▶ ¿Existe el tipo de datos *string* (cadena) en C?
- ▶ ¿Cómo se almacenan/tratan las cadenas en C?

En lenguaje C las *cadenas* son arreglos de caracteres (ASCII)

Ejemplos:

```
char cadena1[] = {'C', 'a', 'd', '1', '\0'};  
char cadena2[5] = {'C', 'a', 'd', '2', '\0'};  
char cadena3[4] = {'C', 'a', 'd', '3', '\0'}; // ERROR  
char cadena4[] = {67, 97, 100, 52, 0}; // ASCII  
char cadena5[] = "Cad5";
```

Cadenas

- ▶ ¿Existe el tipo de datos *string* (cadena) en C?
- ▶ ¿Cómo se almacenan/tratan las cadenas en C?

En lenguaje C las *cadenas* son arreglos de caracteres (ASCII)

Ejemplos:

```
char cadena1[] = {'C', 'a', 'd', '1', '\0'};  
char cadena2[5] = {'C', 'a', 'd', '2', '\0'};  
char cadena3[4] = {'C', 'a', 'd', '3', '\0'}; // ERROR  
char cadena4[] = {67, 97, 100, 52, 0}; // ASCII  
char cadena5[] = "Cad5";  
char *cadena6 = "Cad6";
```

Cadenas

- ▶ ¿Existe el tipo de datos *string* (cadena) en C?
- ▶ ¿Cómo se almacenan/tratan las cadenas en C?

En lenguaje C las *cadenas* son arreglos de caracteres (ASCII)

Ejemplos:

```
char cadena1[] = {'C', 'a', 'd', '1', '\0'};  
char cadena2[5] = {'C', 'a', 'd', '2', '\0'};  
char cadena3[4] = {'C', 'a', 'd', '3', '\0'}; // ERROR  
char cadena4[] = {67, 97, 100, 52, 0}; // ASCII  
char cadena5[] = "Cad5";  
char *cadena6 = "Cad6";
```

Imprimir

```
printf("La cadena es: %s\n", cadena6);
```

Cadenas

- ▶ ¿Existe el tipo de datos *string* (cadena) en C?
- ▶ ¿Cómo se almacenan/tratan las cadenas en C?

En lenguaje C las *cadenas* son arreglos de caracteres (ASCII)

Ejemplos:

```
char cadena1[] = {'C', 'a', 'd', '1', '\0'};  
char cadena2[5] = {'C', 'a', 'd', '2', '\0'};  
char cadena3[4] = {'C', 'a', 'd', '3', '\0'}; // ERROR  
char cadena4[] = {67, 97, 100, 52, 0}; // ASCII  
char cadena5[] = "Cad5";  
char *cadena6 = "Cad6";
```

Imprimir

```
printf("La cadena es: %s\n", cadena6);
```

- ▶ ASCII: American Standard Code for Information Interchange ([Tabla](#))
- ▶ Constante de carácter: 'a', '\0', '\n', etc.

Arreglo de caracteres

Archivo cadenas.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     char cadena1[] = {'C', 'a', 'd', '1', '\0'};
6     char cadena2[5] = {'C', 'a', 'd', '2', '\0'};
7 /* char cadena3[4] = {'C', 'a', 'd', '3', '\0'};*/
8     char cadena4[] = {67, 97, 100, 52, 0};
9     char cadena5[] = "Cad5";
10    char *cadena6 = "Cad6";
11
12    printf("Cadena 1: %s\n", cadena1);
13    printf("Cadena 2: %s\n", cadena2);
14 /* printf("Cadena 3: %s\n", cadena3);*/
15    printf("Cadena 4: %s\n", cadena4);
16    printf("Cadena 5: %s\n", cadena5);
17    printf("Cadena 6: %s\n", cadena6);
18
19    return 0;
20 }
```

Cadenas en lenguaje C

Serie de caracteres tratados como una única unidad

Cadenas en lenguaje C

Serie de caracteres tratados como una única unidad

- En C, una cadena es un arreglo de caracteres terminado con el carácter nulo '\0'

Cadenas en lenguaje C

Serie de caracteres tratados como una única unidad

- ▶ En C, una cadena es un arreglo de caracteres terminado con el carácter nulo '\0'
- ▶ Puede incluir letras, dígitos y caracteres especiales como +, -, *, /, \$, etc.

Cadenas en lenguaje C

Serie de caracteres tratados como una única unidad

- ▶ En C, una cadena es un arreglo de caracteres terminado con el carácter nulo '\0'
- ▶ Puede incluir letras, dígitos y caracteres especiales como +, -, *, /, \$, etc.
- ▶ En C, las *literales* o *constantes* de cadenas se escriben en comillas dobles.

Cadenas en lenguaje C

Serie de caracteres tratados como una única unidad

- ▶ En C, una cadena es un arreglo de caracteres terminado con el carácter nulo '\0'
- ▶ Puede incluir letras, dígitos y caracteres especiales como +, -, *, /, \$, etc.
- ▶ En C, las *literales* o *constantes* de cadenas se escriben en comillas dobles.
- ▶ Se tiene acceso mediante el puntero al primer carácter.

Cadenas en lenguaje C

Serie de caracteres tratados como una única unidad

- ▶ En C, una cadena es un arreglo de caracteres terminado con el carácter nulo '\0'
- ▶ Puede incluir letras, dígitos y caracteres especiales como +, -, *, /, \$, etc.
- ▶ En C, las *literales* o *constantes* de cadenas se escriben en comillas dobles.
- ▶ Se tiene acceso mediante el puntero al primer carácter.

```
char color[] = "verde";
char *ptrColor = "verde";
```

- ▶ Un arreglo de caracteres debe tener el tamaño adecuado para contener la cadena más el carácter de terminación.

Cadenas en lenguaje C

Serie de caracteres tratados como una única unidad

- ▶ En C, una cadena es un arreglo de caracteres terminado con el carácter nulo '\0'
- ▶ Puede incluir letras, dígitos y caracteres especiales como +, -, *, /, \$, etc.
- ▶ En C, las *literales* o *constantes* de cadenas se escriben en comillas dobles.
- ▶ Se tiene acceso mediante el puntero al primer carácter.

```
char color[] = "verde";
char *ptrColor = "verde";
```

- ▶ Un arreglo de caracteres debe tener el tamaño adecuado para contener la cadena más el carácter de terminación.

```
char cadena[20];
scanf("%s", cadena);
```

Biblioteca para el manejo de caracteres

- ▶ Archivo de cabecera `<ctype.h>`
- ▶ Incluye funciones para realizar pruebas o verificación y para la manipulación de datos tipo caracteres.

Biblioteca para el manejo de caracteres

- ▶ Archivo de cabecera <ctype.h>
- ▶ Incluye funciones para realizar pruebas o verificación y para la manipulación de datos tipo caracteres.

Algunas funciones son:

- ▶ `int isdigit(int c)`

Biblioteca para el manejo de caracteres

- ▶ Archivo de cabecera `<ctype.h>`
- ▶ Incluye funciones para realizar pruebas o verificación y para la manipulación de datos tipo caracteres.

Algunas funciones son:

- ▶ `int isdigit(int c)`
- ▶ `int isalpha(int c)`

Biblioteca para el manejo de caracteres

- ▶ Archivo de cabecera `<ctype.h>`
- ▶ Incluye funciones para realizar pruebas o verificación y para la manipulación de datos tipo caracteres.

Algunas funciones son:

- ▶ `int isdigit(int c)`
- ▶ `int isalpha(int c)`
- ▶ `int isalnum(int c)`

Biblioteca para el manejo de caracteres

- ▶ Archivo de cabecera `<ctype.h>`
- ▶ Incluye funciones para realizar pruebas o verificación y para la manipulación de datos tipo caracteres.

Algunas funciones son:

- ▶ `int isdigit(int c)`
- ▶ `int isalpha(int c)`
- ▶ `int isalnum(int c)`
- ▶ `int isxdigit(int c)`

Biblioteca para el manejo de caracteres

- ▶ Archivo de cabecera <ctype.h>
- ▶ Incluye funciones para realizar pruebas o verificación y para la manipulación de datos tipo caracteres.

Algunas funciones son:

- ▶ `int isdigit(int c)`
- ▶ `int isalpha(int c)`
- ▶ `int isalnum(int c)`
- ▶ `int isxdigit(int c)`
- ▶ `int islower(int c)`

Biblioteca para el manejo de caracteres

- ▶ Archivo de cabecera `<ctype.h>`
- ▶ Incluye funciones para realizar pruebas o verificación y para la manipulación de datos tipo caracteres.

Algunas funciones son:

- ▶ `int isdigit(int c)`
- ▶ `int isalpha(int c)`
- ▶ `int isalnum(int c)`
- ▶ `int isxdigit(int c)`
- ▶ `int islower(int c)`
- ▶ `int isupper(int c)`

Biblioteca para el manejo de caracteres

- ▶ Archivo de cabecera <ctype.h>
- ▶ Incluye funciones para realizar pruebas o verificación y para la manipulación de datos tipo caracteres.

Algunas funciones son:

- ▶ `int isdigit(int c)`
- ▶ `int isalpha(int c)`
- ▶ `int isalnum(int c)`
- ▶ `int isxdigit(int c)`
- ▶ `int islower(int c)`
- ▶ `int isupper(int c)`
- ▶ `int tolower(int c)`

Biblioteca para el manejo de caracteres

- ▶ Archivo de cabecera <ctype.h>
- ▶ Incluye funciones para realizar pruebas o verificación y para la manipulación de datos tipo caracteres.

Algunas funciones son:

- ▶ `int isdigit(int c)`
- ▶ `int isalpha(int c)`
- ▶ `int isalnum(int c)`
- ▶ `int isxdigit(int c)`
- ▶ `int islower(int c)`
- ▶ `int isupper(int c)`
- ▶ `int tolower(int c)`
- ▶ `int toupper(int c)`

Funciones de conversión de cadenas

- ▶ Archivo de cabecera `<stdlib.h>` (biblioteca general de utilería).
- ▶ Convierte cadenas de dígitos a enteros y valores en punto flotante.

Funciones de conversión de cadenas

- ▶ Archivo de cabecera `<stdlib.h>` (biblioteca general de utilería).
- ▶ Convierte cadenas de dígitos a enteros y valores en punto flotante.

Algunas funciones son:

- ▶ `double atof(const char *nptr)`

Funciones de conversión de cadenas

- ▶ Archivo de cabecera `<stdlib.h>` (biblioteca general de utilería).
- ▶ Convierte cadenas de dígitos a enteros y valores en punto flotante.

Algunas funciones son:

- ▶ `double atof(const char *nptr)`
- ▶ `int atoi(const char *nptr)`

Funciones de conversión de cadenas

- ▶ Archivo de cabecera `<stdlib.h>` (biblioteca general de utilería).
- ▶ Convierte cadenas de dígitos a enteros y valores en punto flotante.

Algunas funciones son:

- ▶ `double atof(const char *nptr)`
- ▶ `int atoi(const char *nptr)`
- ▶ `long atol(const char *nptr)`

Funciones de conversión de cadenas

- ▶ Archivo de cabecera `<stdlib.h>` (biblioteca general de utilería).
- ▶ Convierte cadenas de dígitos a enteros y valores en punto flotante.

Algunas funciones son:

- ▶ `double atof(const char *nptr)`
- ▶ `int atoi(const char *nptr)`
- ▶ `long atol(const char *nptr)`
- ▶ `float strtod(const char *nptr, char **endptr)`

Funciones de conversión de cadenas

- ▶ Archivo de cabecera `<stdlib.h>` (biblioteca general de utilería).
- ▶ Convierte cadenas de dígitos a enteros y valores en punto flotante.

Algunas funciones son:

- ▶ `double atof(const char *nptr)`
- ▶ `int atoi(const char *nptr)`
- ▶ `long atol(const char *nptr)`
- ▶ `float strtod(const char *nptr, char **endptr)`
- ▶ `double strtod(const char *nptr, char **endptr)`

Funciones de conversión de cadenas

- ▶ Archivo de cabecera `<stdlib.h>` (biblioteca general de utilería).
- ▶ Convierte cadenas de dígitos a enteros y valores en punto flotante.

Algunas funciones son:

- ▶ `double atof(const char *nptr)`
- ▶ `int atoi(const char *nptr)`
- ▶ `long atol(const char *nptr)`
- ▶ `float strtod(const char *nptr, char **endptr)`
- ▶ `double strtod(const char *nptr, char **endptr)`

¿Qué significa `const char *nptr`?

Manipulación y comparación de cadenas

- ▶ Archivo de cabecera <string.h>

Manipulación y comparación de cadenas

- ▶ Archivo de cabecera <string.h>

Algunas funciones son:

- ▶ `char *strcpy(char *dest, const char *src)`

Manipulación y comparación de cadenas

► Archivo de cabecera <string.h>

Algunas funciones son:

- `char *strcpy(char *dest, const char *src)`
- `char *strncpy(char *dest, const char *src, size_t n)`

Manipulación y comparación de cadenas

- ▶ Archivo de cabecera <string.h>

Algunas funciones son:

- ▶ `char *strcpy(char *dest, const char *src)`
- ▶ `char *strncpy(char *dest, const char *src, size_t n)`
- ▶ `char *strcat(char *dest, const char *src)`

Manipulación y comparación de cadenas

► Archivo de cabecera <string.h>

Algunas funciones son:

- ▶ `char *strcpy(char *dest, const char *src)`
- ▶ `char *strncpy(char *dest, const char *src, size_t n)`
- ▶ `char *strcat(char *dest, const char *src)`
- ▶ `char *strncat(char *dest, const char *src, size_t n)`

Manipulación y comparación de cadenas

► Archivo de cabecera <string.h>

Algunas funciones son:

- ▶ `char *strcpy(char *dest, const char *src)`
- ▶ `char *strncpy(char *dest, const char *src, size_t n)`
- ▶ `char *strcat(char *dest, const char *src)`
- ▶ `char *strncat(char *dest, const char *src, size_t n)`
- ▶ `char *strcmp(const char *s1, const char *s2)`

Manipulación y comparación de cadenas

- ▶ Archivo de cabecera <string.h>

Algunas funciones son:

- ▶ `char *strcpy(char *dest, const char *src)`
- ▶ `char *strncpy(char *dest, const char *src, size_t n)`
- ▶ `char *strcat(char *dest, const char *src)`
- ▶ `char *strncat(char *dest, const char *src, size_t n)`
- ▶ `char *strcmp(const char *s1, const char *s2)`
- ▶ `char *strncmp(const char *s1, const char *s2, size_t n)`

Manipulación y comparación de cadenas

- ▶ Archivo de cabecera <string.h>

Algunas funciones son:

- ▶ `char *strcpy(char *dest, const char *src)`
- ▶ `char *strncpy(char *dest, const char *src, size_t n)`
- ▶ `char *strcat(char *dest, const char *src)`
- ▶ `char *strncat(char *dest, const char *src, size_t n)`
- ▶ `char *strcmp(const char *s1, const char *s2)`
- ▶ `char *strncmp(const char *s1, const char *s2, size_t n)`
- ▶ `size_t strlen(const char *s)`

Arreglos de punteros – Arreglos de cadenas

- ▶ Los arreglos pueden contener punteros

Arreglos de punteros – Arreglos de cadenas

- ▶ Los arreglos pueden contener punteros
- ▶ Uso común: arreglos de cadenas

Arreglos de punteros – Arreglos de cadenas

- ▶ Los arreglos pueden contener punteros
- ▶ Uso común: arreglos de cadenas
 - ▶ Cada entrada del arreglo es una cadena

Arreglos de punteros – Arreglos de cadenas

- ▶ Los arreglos pueden contener punteros
- ▶ Uso común: arreglos de cadenas
 - ▶ Cada entrada del arreglo es una cadena
 - ▶ Cada entrada es un puntero al primer carácter de la cadena

Arreglos de punteros – Arreglos de cadenas

- ▶ Los arreglos pueden contener punteros
- ▶ Uso común: arreglos de cadenas
 - ▶ Cada entrada del arreglo es una cadena
 - ▶ Cada entrada es un puntero al primer carácter de la cadena

Ejemplo

```
char *key[4] = {"Top", "Down", "Left", "Right"};
```

Arreglos de punteros – Arreglos de cadenas

- ▶ Los arreglos pueden contener punteros
- ▶ Uso común: arreglos de cadenas
 - ▶ Cada entrada del arreglo es una cadena
 - ▶ Cada entrada es un puntero al primer carácter de la cadena

Ejemplo

```
char *key[4] = {"Top", "Down", "Left", "Right"};
```

¿Qué pasa si se almacenan las cadenas en un arreglo bidimensional?

Ejemplo – Argumentos a la función main

```
int main(void)
{
    /* Programa */
    . . .
    return 0;
}
```

Ejemplo – Argumentos a la función main

```
int main(void)
{
    /* Programa */
    . . .
    return 0;
}
```

```
int main(int argc, char *argv[])
{
    /* Programa */
    . . .
    return 0;
}
```

Ejemplo – Argumentos a la función main

```
int main(void)
{
    /* Programa */
    . .
    return 0;
}
```

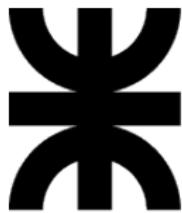
```
int main(int argc, char *argv[])
{
    /* Programa */
    . .
    return 0;
}
```

- ▶ **argc**: cantidad de argumentos en la línea de comandos al ejecutar el programa
- ▶ **argv**: puntero a un vector de cadenas que contiene los argumentos (**argv[argc]** es un puntero **NULL**)

Informática II

Clases de almacenamiento, reglas de alcance,
y calificadores de variables

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2021 –

Clases de almacenamiento

- ▶ Identificadores: para nombres de variables (y funciones)

Clases de almacenamiento

- ▶ **Identificadores:** para nombres de variables (y funciones)
- ▶ Atributos de una variable: **nombre**, **tipo** (tamaño) y **valor**

Clases de almacenamiento

- ▶ **Identificadores:** para nombres de variables (y funciones)
- ▶ Atributos de una variable: **nombre**, **tipo** (tamaño) y **valor**

Cada identificador tiene otro atributo llamados **clase de almacenamiento**, el cual define:

Clases de almacenamiento

- ▶ **Identificadores:** para nombres de variables (y funciones)
- ▶ Atributos de una variable: **nombre**, **tipo** (tamaño) y **valor**

Cada identificador tiene otro atributo llamados **clase de almacenamiento**, el cual define:

1. **Duración de almacenamiento:** período durante el cual el identificador existe en memoria

Clases de almacenamiento

- ▶ **Identificadores:** para nombres de variables (y funciones)
- ▶ Atributos de una variable: **nombre**, **tipo** (tamaño) y **valor**

Cada identificador tiene otro atributo llamados **clase de almacenamiento**, el cual define:

1. **Duración de almacenamiento:** período durante el cual el identificador existe en memoria
2. **Alcance:** desde donde se puede referenciar al identificador (reglas de alcance)

Clases de almacenamiento

- ▶ **Identificadores:** para nombres de variables (y funciones)
- ▶ Atributos de una variable: **nombre**, **tipo** (tamaño) y **valor**

Cada identificador tiene otro atributo llamados **clase de almacenamiento**, el cual define:

1. **Duración de almacenamiento:** período durante el cual el identificador existe en memoria
2. **Alcance:** desde donde se puede referenciar al identificador (reglas de alcance)
3. **Enlace/vinculación:** para programas de varios archivos fuentes

Clases de almacenamiento

- ▶ **Identificadores:** para nombres de variables (y funciones)
- ▶ Atributos de una variable: **nombre**, **tipo** (tamaño) y **valor**

Cada identificador tiene otro atributo llamados **clase de almacenamiento**, el cual define:

1. **Duración de almacenamiento:** período durante el cual el identificador existe en memoria
2. **Alcance:** desde donde se puede referenciar al identificador (reglas de alcance)
3. **Enlace/vinculación:** para programas de varios archivos fuentes

C cuenta con 4 clases de almacenamiento

- ▶ **auto**
- ▶ **register**
- ▶ **static**
- ▶ **extern**

Clases de almacenamiento – Persistencia/duración

Persistencia automática vs. persistencia estática

Clases de almacenamiento – Persistencia/duración

Persistencia automática vs. persistencia estática

Persistencia automática: se crean cuando se entra en el ámbito de un bloque, existen solo en dicho bloque y se destruyen cuando se sale.
(solo para variables)

Clases de almacenamiento – Persistencia/duración

Persistencia automática vs. persistencia estática

Persistencia automática: se crean cuando se entra en el ámbito de un bloque, existen solo en dicho bloque y se destruyen cuando se sale.
(solo para variables)

Las palabras reservadas **auto** y **register** se usan en variables de persistencia automática.

Clases de almacenamiento – Persistencia/duración

Persistencia automática vs. persistencia estática

Persistencia automática: se crean cuando se entra en el ámbito de un bloque, existen solo en dicho bloque y se destruyen cuando se sale.
(solo para variables)

Las palabras reservadas `auto` y `register` se usan en variables de persistencia automática.

Persistencia estática: existen a partir de que el programa inicia su ejecución.
Esto no significa que puedan ser utilizados (alcance).
(también para funciones)

Clases de almacenamiento – Persistencia/duración

Persistencia automática vs. persistencia estática

Persistencia automática: se crean cuando se entra en el ámbito de un bloque, existen solo en dicho bloque y se destruyen cuando se sale.
(solo para variables)

Las palabras reservadas `auto` y `register` se usan en variables de persistencia automática.

Persistencia estática: existen a partir de que el programa inicia su ejecución.
Esto no significa que puedan ser utilizados (alcance).
(también para funciones)

Las palabras reservadas `static` y `extern` se usan en variables de persistencia estática.

Persistencia automática

- ▶ **auto:** Variables locales de una función, ya sean declaradas en la lista de parámetros o en el cuerpo de la función.

```
auto float x, y;
```

Las variables locales tienen persistencia automática por defecto.

Persistencia automática

- ▶ **auto**: Variables locales de una función, ya sean declaradas en la lista de parámetros o en el cuerpo de la función.

```
auto float x, y;
```

Las variables locales tienen persistencia automática por defecto.

- ▶ **register**: Le sugiere al compilador que la variable se guarde en los registros del microprocesador.

```
register int contador = 0;
```

La palabra reservada **register** se puede usar solo en variables automáticas.

(el compilador puede ignorar la sugerencia)

Persistencia estática

- ▶ **static**: Variables locales declaradas con el especificador de clase de almacenamiento **static**. Conocidas solo en la función donde son definidas, pero a diferencia de las variables automáticas, estas conservan su valor cuando se sale de la función.

```
static int contador = 1;
```

Se inicializan a cero si no son inicializadas de forma explícita (o NULL para punteros).

Persistencia estática

- ▶ **static:** Variables locales declaradas con el especificador de clase de almacenamiento **static**. Conocidas solo en la función donde son definidas, pero a diferencia de las variables automáticas, estas conservan su valor cuando se sale de la función.

```
static int contador = 1;
```

Se inicializan a cero si no son inicializadas de forma explícita (o NULL para punteros).

- ▶ **extern:** Se utiliza para programas de varios archivos fuentes. Por defecto, las variables globales y los nombres de funciones son de la clase de almacenamiento **extern**.

Más sobre **extern**

- *externo* en contraste a *interno* –en funciones–

Más sobre `extern`

- ▶ *externo* en contraste a *interno* –en funciones–
- ▶ Una variable es *externa* si se encuentra fuera de cualquier función

Más sobre **extern**

- ▶ *externo* en contraste a *interno* –en funciones–
- ▶ Una variable es *externa* si se encuentra fuera de cualquier función

Para el caso de variables **extern**:

Declaración: expone las propiedades de una variable (principalmente su tipo)

Definición: provoca que se reserve espacio para el almacenamiento

Alcance de variables

Alcance de un identificador: porción del programa donde puede ser referenciado.

Alcance de variables

Alcance de un identificador: porción del programa donde puede ser referenciado.

Ejemplo: variable local declarada en un bloque –puede ser referenciada en el bloque o bloques anidados–

Alcance de variables

Alcance de un identificador: porción del programa donde puede ser referenciado.

Ejemplo: variable local declarada en un bloque –puede ser referenciada en el bloque o bloques anidados–

Tipos de alcances

1. Alcance de archivo: identificador declarado fuera de cualquier función

Alcance de variables

Alcance de un identificador: porción del programa donde puede ser referenciado.

Ejemplo: variable local declarada en un bloque –puede ser referenciada en el bloque o bloques anidados–

Tipos de alcances

1. Alcance de archivo: identificador declarado fuera de cualquier función
2. Alcance de función: etiquetas (identificador seguido de :). p.e.: **start:**
Etiquetas **case** en estructura **switch**

Alcance de variables

Alcance de un identificador: porción del programa donde puede ser referenciado.

Ejemplo: variable local declarada en un bloque –puede ser referenciada en el bloque o bloques anidados–

Tipos de alcances

1. Alcance de archivo: identificador declarado fuera de cualquier función
2. Alcance de función: etiquetas (identificador seguido de :). p.e.: **start:**
Etiquetas **case** en estructura **switch**
3. Alcance de bloque: identificadores declarados dentro de un bloque
Las variables locales **static** declaradas dentro de funciones tienen alcance de bloques aún cuando existen al momento de ejecutarse el programa

Alcance de variables

Alcance de un identificador: porción del programa donde puede ser referenciado.

Ejemplo: variable local declarada en un bloque –puede ser referenciada en el bloque o bloques anidados–

Tipos de alcances

1. Alcance de archivo: identificador declarado fuera de cualquier función
2. Alcance de función: etiquetas (identificador seguido de :). p.e.: **start**:
Etiquetas **case** en estructura **switch**
3. Alcance de bloque: identificadores declarados dentro de un bloque
Las variables locales **static** declaradas dentro de funciones tienen alcance de bloques aún cuando existen al momento de ejecutarse el programa
4. Alcance de prototipo de función: lista de parámetros en los prototipos de funciones

Alcance de variables – Ejemplos

El alcance de una variable o función *externa* va desde el punto en el que se declara hasta el fin del archivo.

Alcance de variables – Ejemplos

El alcance de una variable o función *externa* va desde el punto en el que se declara hasta el fin del archivo.

```
int main(void) { . . . }

int indice = 0;
double vector[MAXVAL];

void f1(double f) { . . . }
double f2(void) { . . .}
```

Alcance de variables – Ejemplos

El alcance de una variable o función *externa* va desde el punto en el que se declara hasta el fin del archivo.

```
int main(void) { . . . }

int indice = 0;
double vector[MAXVAL];

void f1(double f) { . . . }
double f2(void) { . . . }
```

- ▶ Las variables `indice` y `vector` se pueden utilizar en `f1()` y `f2()`, pero no en `main()`.
- ▶ Si se hace referencia a una variable *externa* antes de su definición, o si está definida en un archivo fuente diferente, es obligatorio una declaración `extern`.

Alcance de variables – Ejemplos

Si las líneas

```
int indice;  
double vector[MAXVAL];
```

aparecen fuera de cualquier función, *definen* las variables externas `indice` y `vector`, y reserva espacio para su almacenamiento.

Alcance de variables – Ejemplos

Si las líneas

```
int indice;  
double vector[MAXVAL];
```

aparecen fuera de cualquier función, *definen* las variables externas `indice` y `vector`, y reserva espacio para su almacenamiento.

Las líneas

```
extern int indice;  
extern double vector[];
```

declaran para el resto del archivo que `indice` es un `int` y que `vector` es un arreglo `double` (cuyo tamaño se determina en algún otro lugar), pero no crea las variables ni les reserva espacio.

Alcance de variables – Ejemplos

Si las líneas

```
int indice;  
double vector[MAXVAL];
```

aparecen fuera de cualquier función, *definen* las variables externas `indice` y `vector`, y reserva espacio para su almacenamiento.

Las líneas

```
extern int indice;  
extern double vector[];
```

declaran para el resto del archivo que `indice` es un `int` y que `vector` es un arreglo `double` (cuyo tamaño se determina en algún otro lugar), pero no crea las variables ni les reserva espacio.

- ▶ Debe existir una única *definición* de una variable externa entre todos los archivos fuentes que forman un programa.
- ▶ Los demás archivos pueden hacer *declaraciones extern* de estas variables para tener acceso a ellas.

Alcance de variables – Ejemplos

Archivo 1

```
int indice = 0;
double vector[MAXVAL];
```

Archivo 2

```
extern int indice;
extern double vector;

void f1(double f) { . . . }

double f2(void) { . . . }
```

Alcance de variables – Ejemplos

Archivo 1

```
int indice = 0;  
double vector[MAXVAL];
```

Archivo 2

```
extern int indice;  
extern double vector;  
  
void f1(double f) { . . . }  
  
double f2(void) { . . . }
```

Ver ejemplo de *alcance* D&D 5.12.

Calificadores – **volatile** y **const**

volatile

- ▶ Le indica al compilador no optimizar lo relacionado a dichas variables
- ▶ Utilizado en la mayoría de los casos para el acceso al hardware
- ▶ Le indica que la variable no se guarde en cache

Calificadores – **volatile** y **const**

volatile

- ▶ Le indica al compilador no optimizar lo relacionado a dichas variables
- ▶ Utilizado en la mayoría de los casos para el acceso al hardware
- ▶ Le indica que la variable no se guarde en cache

Ejemplo:

```
salir = 0;
while(!salir)
{
    /* bucle corto completamente
     * visible al compilador */
}
```

Calificadores – **volatile** y **const**

volatile

- ▶ Le indica al compilador no optimizar lo relacionado a dichas variables
- ▶ Utilizado en la mayoría de los casos para el acceso al hardware
- ▶ Le indica que la variable no se guarde en cache

Ejemplo:

```
salir = 0;
while(!salir)
{
    /* bucle corto completamente
     * visible al compilador */
}
```

- ▶ El compilador determina que el cuerpo del bucle no modifica la variable **salir** y convierte el bucle en un bucle **while(1)**. Incluso si la variable de salida se establece en el controlador de señal para SIGINT y SIGTERM.
- ▶ Si la variable **salir** se declara **volatile**, el compilador se ve obligado a cargarla cada vez, ya que puede modificarse en otro lugar.

Calificadores – volatile y const

const

- Le informa al compilador que el valor de una variable no debe modificarse.

Calificadores – **volatile** y **const**

const

- ▶ Le informa al compilador que el valor de una variable no debe modificarse.
- ▶ No existía en las primeras versiones de C. Fue agregado en el ANSI C.

Calificadores – **volatile** y **const**

const

- ▶ Le informa al compilador que el valor de una variable no debe modificarse.
- ▶ No existía en las primeras versiones de C. Fue agregado en el ANSI C.
- ▶ Seis posibilidades del uso y no uso de **const** con parámetros de función.

Calificadores – **volatile** y **const**

const

- ▶ Le informa al compilador que el valor de una variable no debe modificarse.
- ▶ No existía en las primeras versiones de C. Fue agregado en el ANSI C.
- ▶ Seis posibilidades del uso y no uso de **const** con parámetros de función.
 - ▶ dos al pasar parámetros en llamada por valor, y

Calificadores – **volatile** y **const**

const

- ▶ Le informa al compilador que el valor de una variable no debe modificarse.
- ▶ No existía en las primeras versiones de C. Fue agregado en el ANSI C.
- ▶ Seis posibilidades del uso y no uso de **const** con parámetros de función.
 - ▶ dos al pasar parámetros en llamada por valor, y
 - ▶ cuatro al pasar parámetros en llamada por referencia (punteros).

Calificadores – **volatile** y **const**

const

- ▶ Le informa al compilador que el valor de una variable no debe modificarse.
- ▶ No existía en las primeras versiones de C. Fue agregado en el ANSI C.
- ▶ Seis posibilidades del uso y no uso de **const** con parámetros de función.
 - ▶ dos al pasar parámetros en llamada por valor, y
 - ▶ cuatro al pasar parámetros en llamada por referencia (punteros).

Ejemplo:

```
void imprimir_arreglo(const int datos[], const int tam)
{
    . . .
}
```

Calificador **const** con punteros

Existen cuatro formas de pasar punteros a funciones:

Calificador `const` con punteros

Existen cuatro formas de pasar punteros a funciones:

1. Puntero no constante a datos no constantes, p.e.: `int *pi;` (notación) (no incluye `const`)

Calificador `const` con punteros

Existen cuatro formas de pasar punteros a funciones:

1. Puntero no constante a datos no constantes, p.e.: `int *pi;` (notación) (no incluye `const`)
2. Puntero no constante a datos constantes

Calificador `const` con punteros

Existen cuatro formas de pasar punteros a funciones:

1. Puntero no constante a datos no constantes, p.e.: `int *pi;` (notación) (no incluye `const`)
2. Puntero no constante a datos constantes
3. Puntero constante a datos no constantes. Inicializados al declararlos (nombre de arreglo)

Calificador `const` con punteros

Existen cuatro formas de pasar punteros a funciones:

1. Puntero no constante a datos no constantes, p.e.: `int *pi;` (notación) (no incluye `const`)
2. Puntero no constante a datos constantes
3. Puntero constante a datos no constantes. Inicializados al declararlos (nombre de arreglo)
4. Puntero constante a datos constantes

Calificador **const** con punteros –Ejemplos

Puntero a entero constante

```
int * const foo = &var;
```

Calificador **const** con punteros –Ejemplos

Puntero a entero constante

```
int * const foo = &var;
```

Puntero constante a entero

```
int const * foo = &var;  
const int * foo = &var;
```

Calificador **const** con punteros –Ejemplos

Puntero a entero constante

```
int * const foo = &var;
```

Puntero constante a entero

```
int const * foo = &var;  
const int * foo = &var;
```

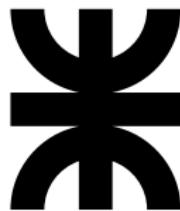
Puntero constante a un entero constante

```
int const * const foo = &var;  
const int * const foo = &var;
```


Informática II

Recursión

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2021 –

Introducción

- ▶ Programas sin recursión: están estructurados como funciones que se llaman unas a otras de una forma disciplinada y jerárquica.

Introducción

- ▶ Programas sin recursión: están estructurados como funciones que se llaman unas a otras de una forma **disciplinada** y **jerárquica**.
- ▶ Para la resolución de algunos problemas es útil contar con funciones que se llaman a sí mismas.

Introducción

- ▶ Programas sin recursión: están estructurados como funciones que se llaman unas a otras de una forma **disciplinada** y **jerárquica**.
- ▶ Para la resolución de algunos problemas es útil contar con funciones que se llaman a sí mismas.

Función recursiva

Es una función que se llama a sí misma, ya sea directa o indirectamente, a través de otra función.

Enfoque recursivo

Los problemas a resolver con enfoque recursivo tienen ciertos aspectos en común:

Enfoque recursivo

Los problemas a resolver con enfoque recursivo tienen ciertos aspectos en común:

- ▶ Existe un **caso base** (la función solo resuelve este caso).

Enfoque recursivo

Los problemas a resolver con enfoque recursivo tienen ciertos aspectos en común:

- ▶ Existe un **caso base** (la función solo resuelve este caso).
- ▶ Para un problema complejo existe una versión del mismo pero más sencillo.

Entonces, se divide el problema:

Enfoque recursivo

Los problemas a resolver con enfoque recursivo tienen ciertos aspectos en común:

- ▶ Existe un **caso base** (la función solo resuelve este caso).
- ▶ Para un problema complejo existe una versión del mismo pero más sencillo.

Entonces, se divide el problema:

1. una parte que se sabe resolver

Enfoque recursivo

Los problemas a resolver con enfoque recursivo tienen ciertos aspectos en común:

- ▶ Existe un **caso base** (la función solo resuelve este caso).
- ▶ Para un problema complejo existe una versión del mismo pero más sencillo.

Entonces, se divide el problema:

1. una parte que se sabe resolver
2. una parte que no se sabe resolver pero parecido al problema original

Enfoque recursivo

Los problemas a resolver con enfoque recursivo tienen ciertos aspectos en común:

- ▶ Existe un **caso base** (la función solo resuelve este caso).
- ▶ Para un problema complejo existe una versión del mismo pero más sencillo.

Entonces, se divide el problema:

1. una parte que se sabe resolver
2. una parte que no se sabe resolver pero parecido al problema original

 - ▶ Terminación de la recursión. Convergencia al caso base.

Ejemplo 1: cálculo de factorial

Ejemplo 1: cálculo de factorial

Factorial de un número entero no negativo ($n > 0$) es

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdots 1$$

con $1! = 0! = 1$.

Ejemplo 1: cálculo de factorial

Factorial de un número entero no negativo ($n > 0$) es

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdots 1$$

con $1! = 0! = 1$.

Ejemplo: $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$.

Ejemplo 1: cálculo de factorial

Factorial de un número entero no negativo ($n > 0$) es

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdots 1$$

con $1! = 0! = 1$.

Ejemplo: $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$.

Factorial de un número entero `num` mayor a cero de *forma iterativa*

```
1  factorial = 1;
2  for(cont = num; cont >= 1; cont--)
3      factorial *= cont;
```

Ejemplo 1: cálculo de factorial

Definición recursiva

$$n! = n \cdot (n - 1)!$$

Ejemplo 1: cálculo de factorial

Definición recursiva

$$n! = n \cdot (n - 1)!$$

Ejemplo: 5!

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$

$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$

$$5! = 5 \cdot (4!)$$

Ejemplo 1: cálculo de factorial

Definición recursiva

$$n! = n \cdot (n - 1)!$$

Ejemplo: 5!

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$

$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$

$$5! = 5 \cdot (4!)$$

El problema se divide en: 1) una parte que se sabe resolver (caso base) y 2) una parte similar al problema original

Ejemplo 1: cálculo de factorial

Definición recursiva

$$n! = n \cdot (n - 1)!$$

Ejemplo: 5!

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$

$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$

$$5! = 5 \cdot (4!)$$

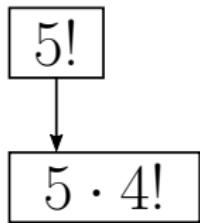
El problema se divide en: 1) una parte que se sabe resolver (caso base) y 2) una parte similar al problema original

👉 Caso base: $0! = 1! = 1$

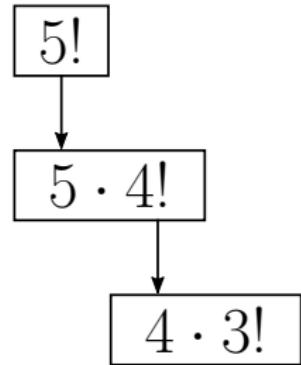
Ejemplo 1: cálculo de factorial

5!

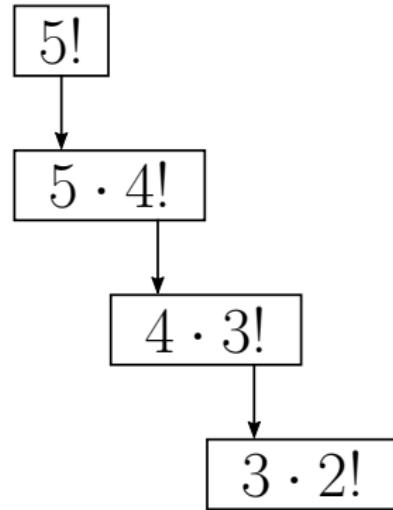
Ejemplo 1: cálculo de factorial



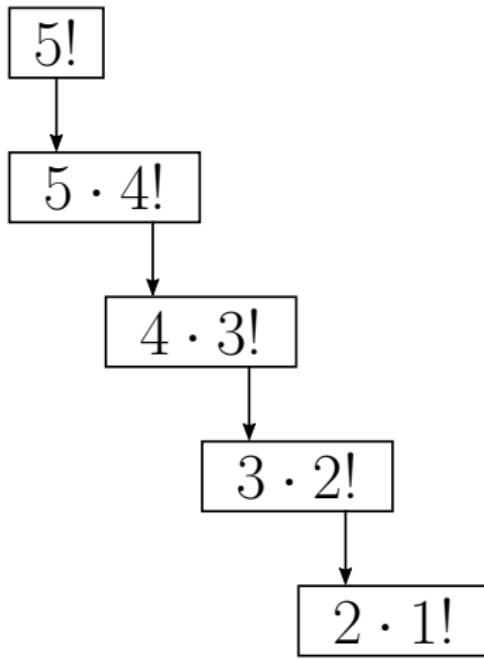
Ejemplo 1: cálculo de factorial



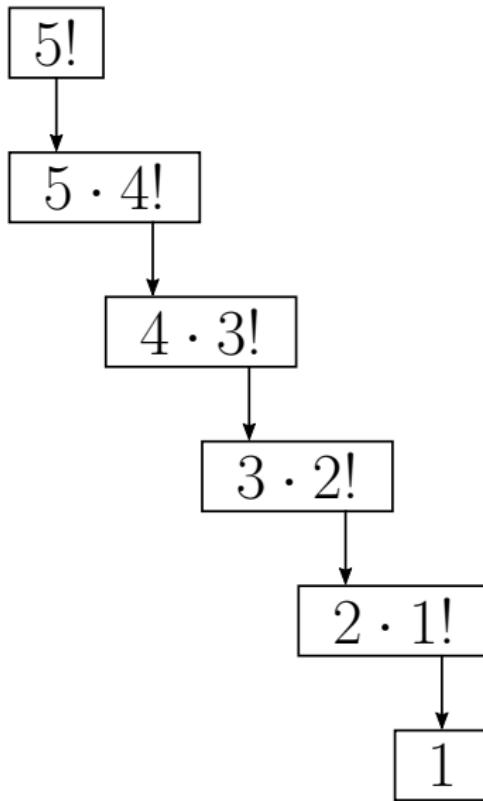
Ejemplo 1: cálculo de factorial



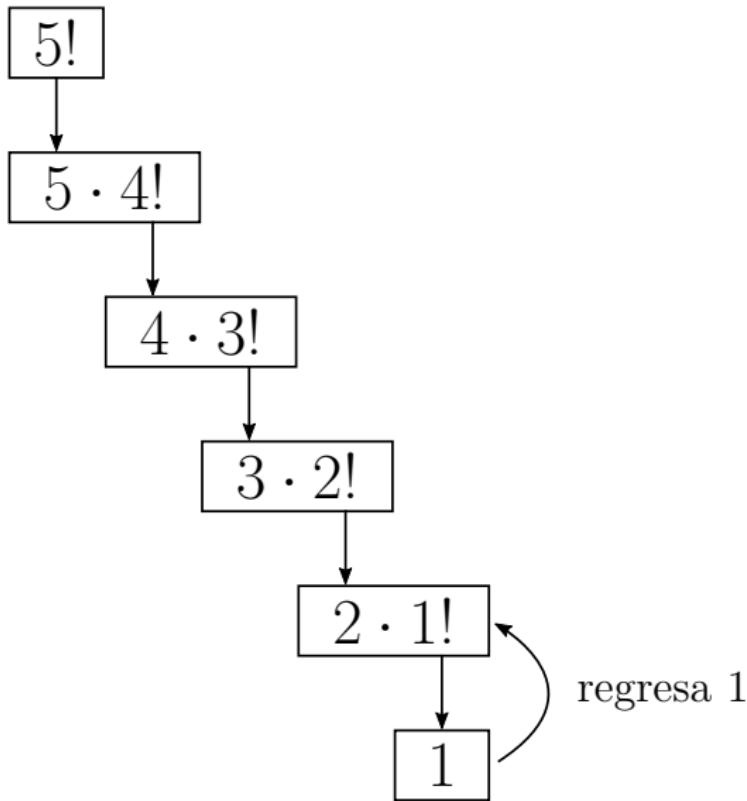
Ejemplo 1: cálculo de factorial



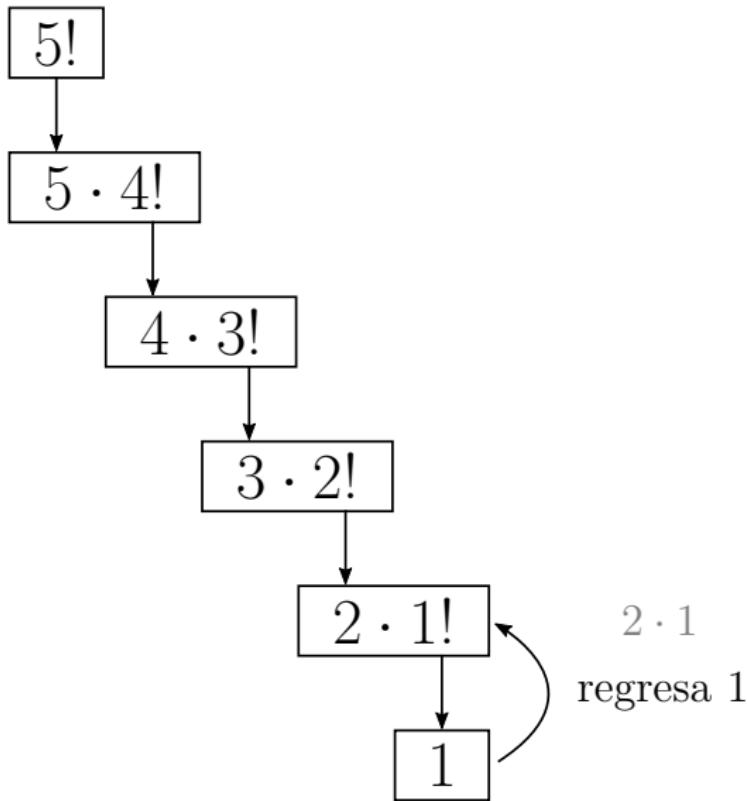
Ejemplo 1: cálculo de factorial



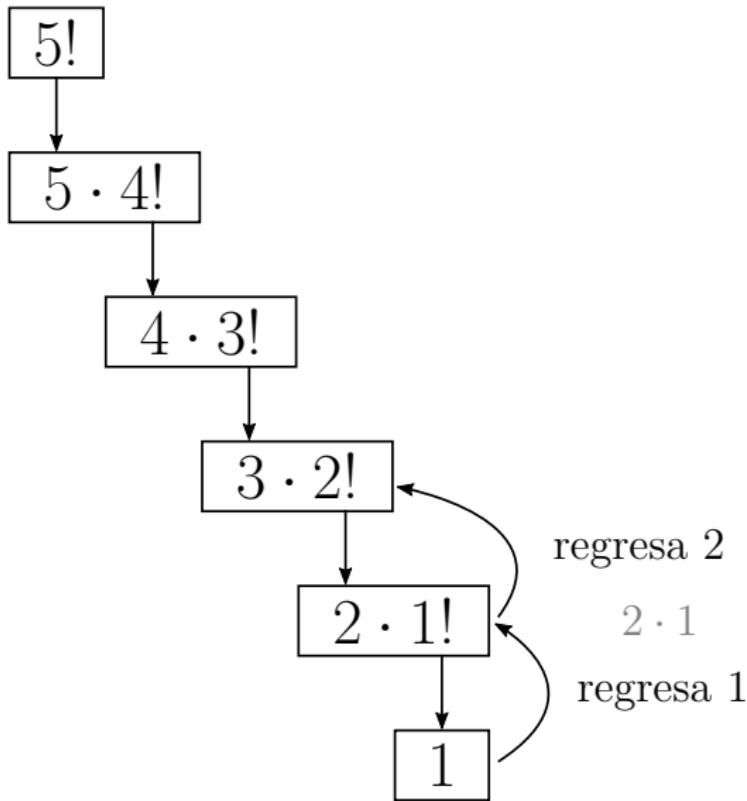
Ejemplo 1: cálculo de factorial



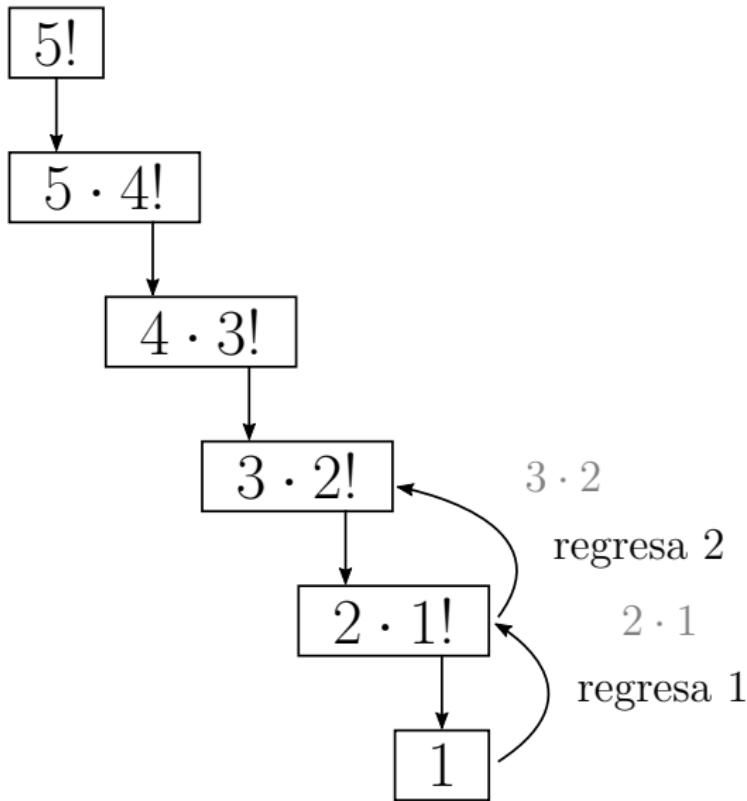
Ejemplo 1: cálculo de factorial



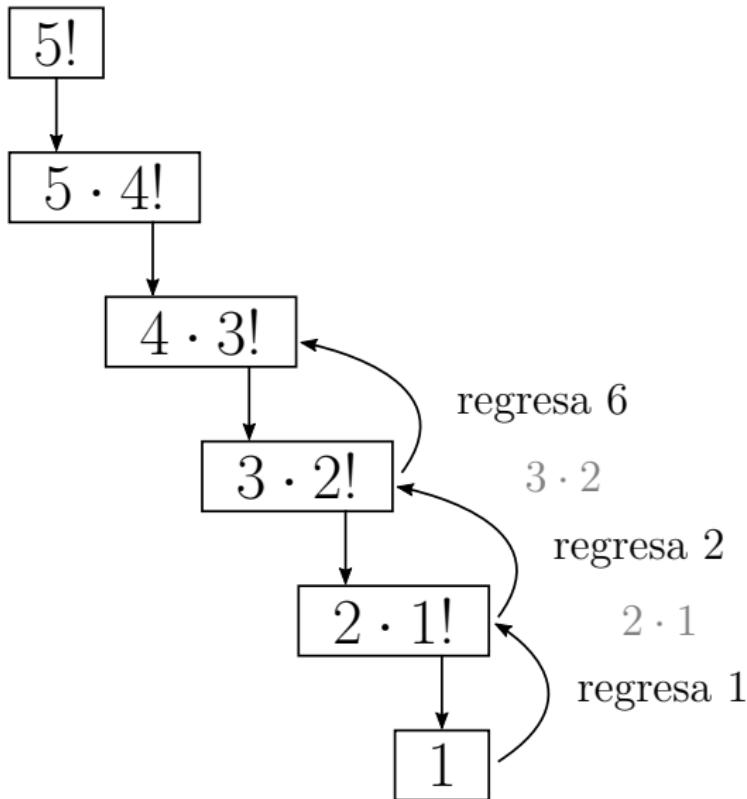
Ejemplo 1: cálculo de factorial



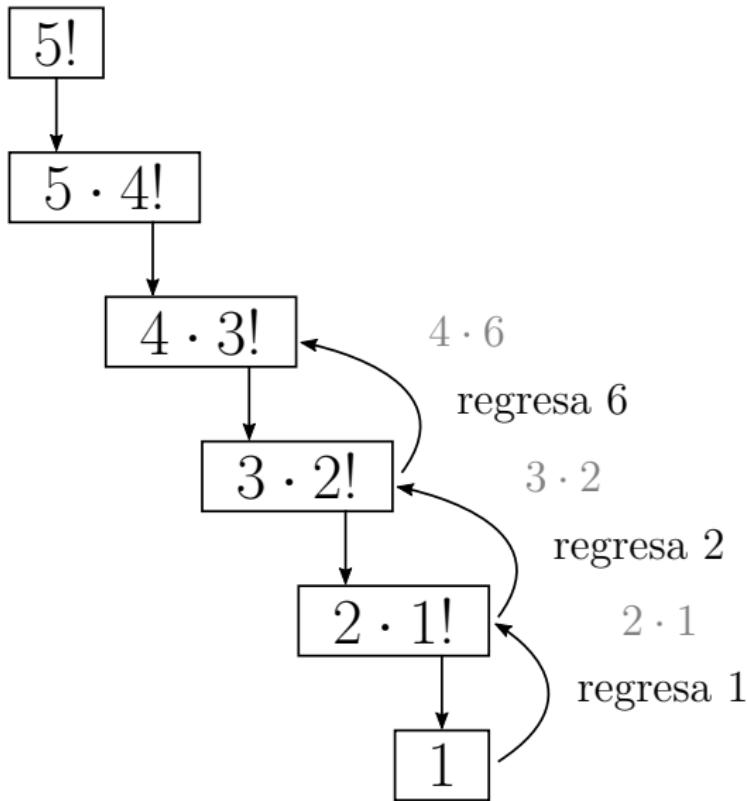
Ejemplo 1: cálculo de factorial



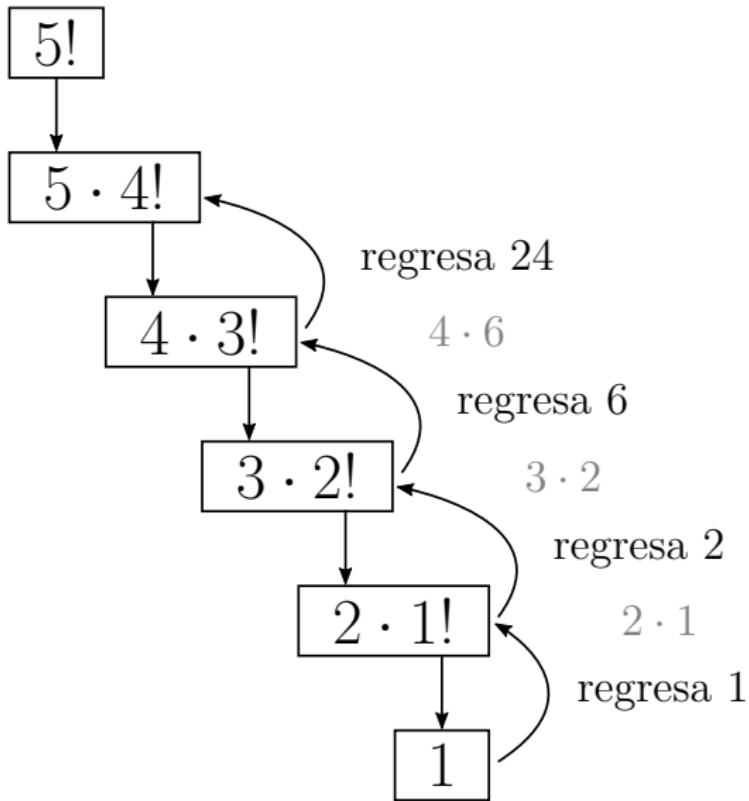
Ejemplo 1: cálculo de factorial



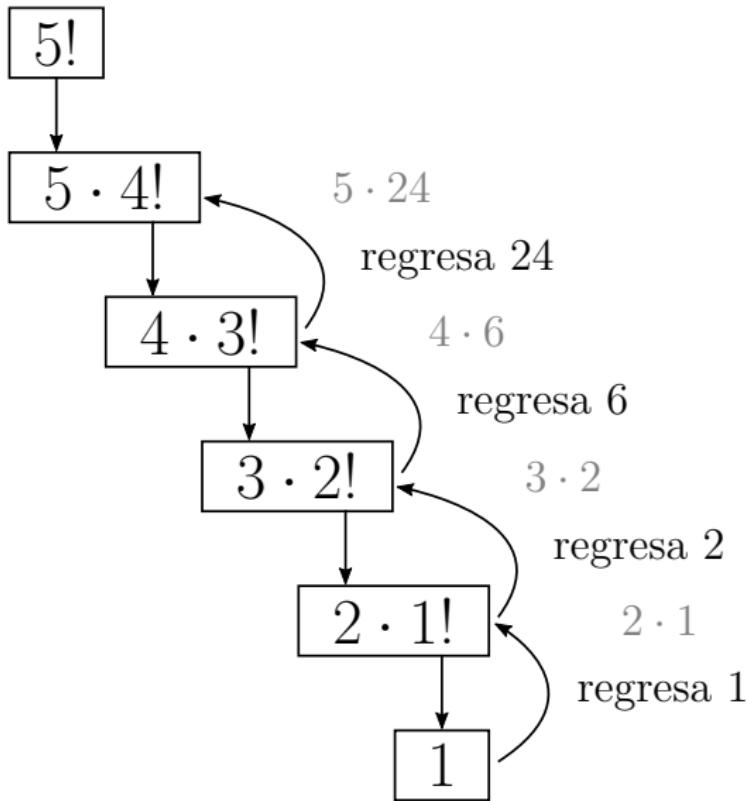
Ejemplo 1: cálculo de factorial



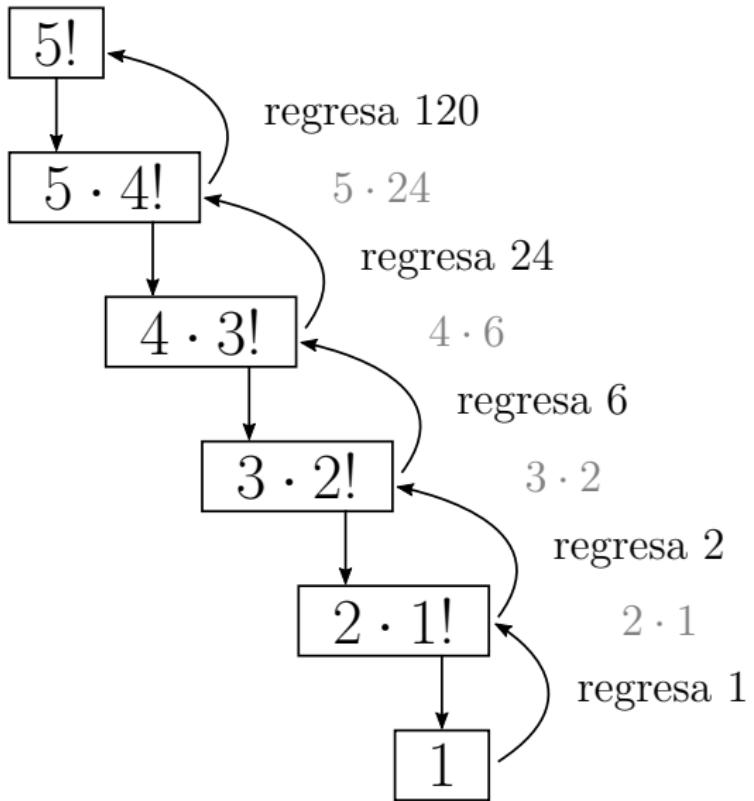
Ejemplo 1: cálculo de factorial



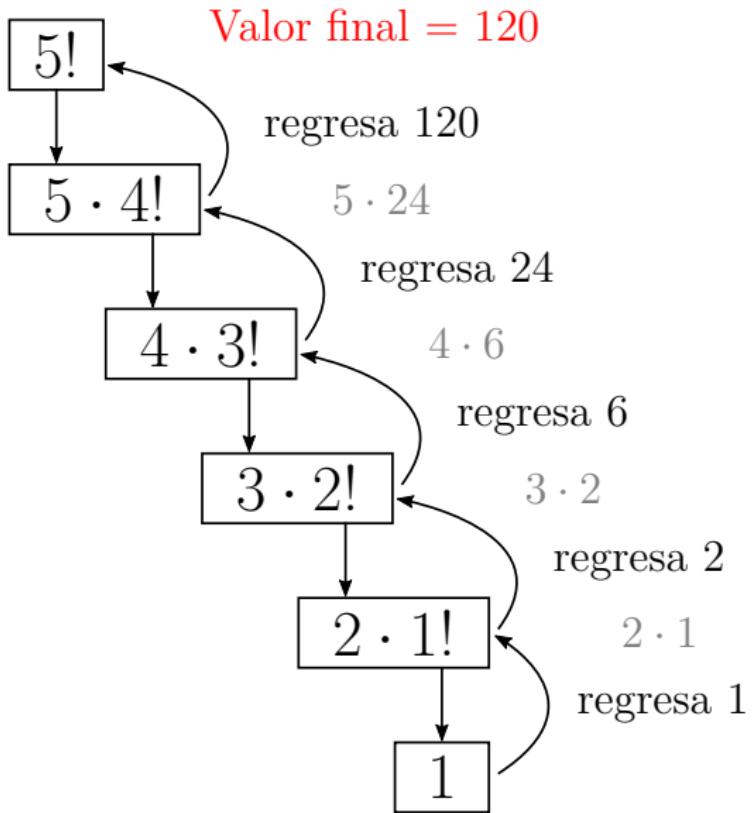
Ejemplo 1: cálculo de factorial



Ejemplo 1: cálculo de factorial



Ejemplo 1: cálculo de factorial



Ejemplo 1: cálculo de factorial

```
1 #include <stdio.h>
2
3 long factorial(long ); /* Prototipo de función */
4
5 int main(void)
6 {
7     int i;
8
9     for(i = 0; i <= 10; i++)
10    printf("%2d! = %ld\n", i, factorial(i));
11
12    return 0;
13 }
14
15 /* Función recursiva 'factorial' */
16 long factorial(long numero)
17 {
18
19
20
21
22 }
```

Ejemplo 1: cálculo de factorial

```
1 #include <stdio.h>
2
3 long factorial(long ); /* Prototipo de función */
4
5 int main(void)
6 {
7     int i;
8
9     for(i = 0; i <= 10; i++)
10    printf("%2d! = %ld\n", i, factorial(i));
11
12    return 0;
13 }
14
15 /* Función recursiva 'factorial' */
16 long factorial(long numero)
17 {
18     if(numero <= 1) /* Caso base */
19         return 1;
20
21
22 }
```

Ejemplo 1: cálculo de factorial

```
1 #include <stdio.h>
2
3 long factorial(long ); /* Prototipo de función */
4
5 int main(void)
6 {
7     int i;
8
9     for(i = 0; i <= 10; i++)
10    printf("%2d! = %ld\n", i, factorial(i));
11
12    return 0;
13 }
14
15 /* Función recursiva 'factorial' */
16 long factorial(long numero)
17 {
18     if(numero <= 1) /* Caso base */
19         return 1;
20     else /* Recursión */
21         return (numero * factorial(numero - 1));
22 }
```

Recursión: ventajas y desventajas

Ventajas

- Algoritmos más claros y sencillos

Recursión: ventajas y desventajas

Ventajas

- ▶ Algoritmos más claros y sencillos
- ▶ Las funciones recursivas son más elegantes

Recursión: ventajas y desventajas

Ventajas

- ▶ Algoritmos más claros y sencillos
- ▶ Las funciones recursivas son más elegantes
- ▶ Sintetizan en poco espacio lo que sería complicado en la forma iterativa

Recursión: ventajas y desventajas

Ventajas

- ▶ Algoritmos más claros y sencillos
- ▶ Las funciones recursivas son más elegantes
- ▶ Sintetizan en poco espacio lo que sería complicado en la forma iterativa
- ▶ Forma natural a los problemas inherentemente recursivos

Recursión: ventajas y desventajas

Ventajas

- ▶ Algoritmos más claros y sencillos
- ▶ Las funciones recursivas son más elegantes
- ▶ Sintetizan en poco espacio lo que sería complicado en la forma iterativa
- ▶ Forma natural a los problemas inherentemente recursivos

Desventajas

- ▶ Consumen más memoria, pudiendo agotarse

Recursión: ventajas y desventajas

Ventajas

- ▶ Algoritmos más claros y sencillos
- ▶ Las funciones recursivas son más elegantes
- ▶ Sintetizan en poco espacio lo que sería complicado en la forma iterativa
- ▶ Forma natural a los problemas inherentemente recursivos

Desventajas

- ▶ Consumen más memoria, pudiendo agotarse
- ▶ Más lentas que las versiones iterativas debido a la cantidad de llamadas a funciones

Recursión: ventajas y desventajas

Ventajas

- ▶ Algoritmos más claros y sencillos
- ▶ Las funciones recursivas son más elegantes
- ▶ Sintetizan en poco espacio lo que sería complicado en la forma iterativa
- ▶ Forma natural a los problemas inherentemente recursivos

Desventajas

- ▶ Consumen más memoria, pudiendo agotarse
- ▶ Más lentas que las versiones iterativas debido a la cantidad de llamadas a funciones
- ▶ Más difíciles de comprender

Ejemplo 2: la serie Fibonacci

La serie de Fibonacci

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \dots$$

empieza con 0 y 1, y tiene la propiedad de que cada número es la suma de los dos números previos.

Ejemplo 2: la serie Fibonacci

La serie de Fibonacci

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \dots$$

empieza con 0 y 1, y tiene la propiedad de que cada número es la suma de los dos números previos.

La relación de números sucesivos de Fibonacci converge al valor constante 1,618, conocido como *relación áurea*.

Ejemplo 2: la serie Fibonacci

La serie de Fibonacci

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \dots$$

empieza con 0 y 1, y tiene la propiedad de que cada número es la suma de los dos números previos.

La relación de números sucesivos de Fibonacci converge al valor constante 1,618, conocido como *relación áurea*.

Definición recursiva

$$\text{fibonacci}(0) = 0$$

$$\text{fibonacci}(1) = 1$$

$$\text{fibonacci}(2) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$$

Ejemplo 2: la serie Fibonacci

```
1 #include <stdio.h>
2 long fibonacci(long ); /* Prototipo de función */
3
4 int main(void)
5 {
6     long resultado, numero;
7
8     printf("Ingrese un entero: ");
9     scanf("%ld", &numero);
10
11    resultado = fibonacci(numero);
12    printf("fibonacci(%ld) = %ld\n", numero, resultado);
13    return 0;
14 }
15
16 /* Función recursiva 'fibonacci' */
17 long fibonacci(long n)
18 {
19
20
21
22 }
```

Ejemplo 2: la serie Fibonacci

```
1 #include <stdio.h>
2 long fibonacci(long ); /* Prototipo de función */
3
4 int main(void)
5 {
6     long resultado, numero;
7
8     printf("Ingrese un entero: ");
9     scanf("%ld", &numero);
10
11    resultado = fibonacci(numero);
12    printf("fibonacci(%ld) = %ld\n", numero, resultado);
13    return 0;
14 }
15
16 /* Función recursiva 'fibonacci' */
17 long fibonacci(long n)
18 {
19     if(n == 0 || n == 1) /* Caso base */
20         return n;
21     else /* Recursión */
22         return fibonacci(n-1) + fibonacci(n-2);
23 }
```

Comparación entre recursión e iteración

- Basados en una estructura de control
 - iteración: estructura de repetición
 - recursión: estructura de selección

Comparación entre recursión e iteración

- ▶ Basados en una estructura de control
 - iteración: estructura de repetición
 - recursión: estructura de selección
- ▶ Implican repetición
 - iteración: utiliza la estructura de repetición de forma explícita
 - recursión: repetición con llamadas de función repetidas

Comparación entre recursión e iteración

- ▶ Basados en una estructura de control
 - iteración: estructura de repetición
 - recursión: estructura de selección
- ▶ Implican repetición
 - iteración: utiliza la estructura de repetición de forma explícita
 - recursión: repetición con llamadas de función repetidas
- ▶ Prueba de terminación
 - iterativa: falla la condición de continuación del ciclo
 - recursión: se reconoce un caso base

Comparación entre recursión e iteración

- ▶ Basados en una estructura de control
 - iteración: estructura de repetición
 - recursión: estructura de selección
- ▶ Implican repetición
 - iteración: utiliza la estructura de repetición de forma explícita
 - recursión: repetición con llamadas de función repetidas
- ▶ Prueba de terminación
 - iterativa: falla la condición de continuación del ciclo
 - recursión: se reconoce un caso base
- ▶ Pueden ocurrir de forma indefinida (ciclo infinito)
 - iteración: si la condición de continuación del ciclo nunca es falsa
 - recursión: si la recursión no reduce el problema en cada ocasión

Complejidad computacional

- ▶ En cada llamada recursiva de `fibonacci` se llama dos veces a la misma función
- ▶ La cantidad de llamadas recursivas es por lo tanto 2^n
- ▶ Este número crece muy rápidamente; ejemplos: $2^{20} \approx$ millón, $2^{30} \approx$ mil millones
- ▶ En ciencia de la computación a esto se le llama *complejidad exponencial*

Actividad práctica

1. Escribir un programa que defina una función **factorial()** donde el cálculo se realiza de forma iterativa. Evaluar dicha función de forma similar al programa ejemplo de la función factorial recursiva.
2. Escribir dos programas que imprima la serie de Fibonacci con la siguiente interacción con el usuario:

```
> ./a.out
--- Programa de serie de Fibonacci ---
Ingrese un entero: 12
Serie: 0 1 1 2 3 5 8 13 21 34 55 89 144
```

Uno de ellos debe implementar la función **fibonacci(n)** de forma iterativa y el otro de forma recursiva.

Actividad práctica

3. Modificar el programa ejemplo de la función recursiva del cálculo del factorial para visualizar la recursión. El programa debe tener la siguiente interacción con el usuario

```
> ./a.out  
Ingrese un entero: 4
```

```
4! = 4 * 3!  
    3! = 3 * 2!  
        2! = 2 * 1!
```

```
--> Resultado: 24
```

```
> ./a.out  
Ingrese un entero: 5
```

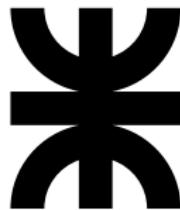
```
5! = 5 * 4!  
    4! = 4 * 3!  
        3! = 3 * 2!  
            2! = 2 * 1!
```

```
--> Resultado: 120
```


Informática II

Uniones y campos de bits

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2021 –

Uniones – Definición

Es un **tipo de dato derivado** —como lo es una estructura— cuyos miembros comparten el mismo espacio de almacenamiento.

Uniones – Definición

Es un **tipo de dato derivado** —como lo es una estructura— cuyos miembros comparten el mismo espacio de almacenamiento.

Para ciertas situaciones en un programa, algunas variables pudieran ser importantes y otras no. Las uniones **comparten el espacio**, en vez de desperdiciar almacenamiento en variables que no están siendo utilizadas.

Uniones – Definición

Es un **tipo de dato derivado** —como lo es una estructura— cuyos miembros comparten el mismo espacio de almacenamiento.

Para ciertas situaciones en un programa, algunas variables pudieran ser importantes y otras no. Las uniones **comparten el espacio**, en vez de desperdiciar almacenamiento en variables que no están siendo utilizadas.

Declaración:

```
union numero {  
    int x;  
    float y;  
};
```

Uniones – Definición

Es un **tipo de dato derivado** —como lo es una estructura— cuyos miembros comparten el mismo espacio de almacenamiento.

Para ciertas situaciones en un programa, algunas variables pudieran ser importantes y otras no. Las uniones **comparten el espacio**, en vez de desperdiciar almacenamiento en variables que no están siendo utilizadas.

Declaración:

```
union numero {  
    int x;  
    float y;  
};
```

- ▶ Ocupa en memoria lo suficiente para contener el miembro más grande

Uniones – Definición

Es un **tipo de dato derivado** —como lo es una estructura— cuyos miembros comparten el mismo espacio de almacenamiento.

Para ciertas situaciones en un programa, algunas variables pudieran ser importantes y otras no. Las uniones **comparten el espacio**, en vez de desperdiciar almacenamiento en variables que no están siendo utilizadas.

Declaración:

```
union numero {  
    int x;  
    float y;  
};
```

- ▶ Ocupa en memoria lo suficiente para contener el miembro más grande
- ▶ En general contienen dos o más tipos de datos

Uniones – Definición

Es un **tipo de dato derivado** —como lo es una estructura— cuyos miembros comparten el mismo espacio de almacenamiento.

Para ciertas situaciones en un programa, algunas variables pudieran ser importantes y otras no. Las uniones **comparten el espacio**, en vez de desperdiciar almacenamiento en variables que no están siendo utilizadas.

Declaración:

```
union numero {  
    int x;  
    float y;  
};
```

- ▶ Ocupa en memoria lo suficiente para contener el miembro más grande
- ▶ En general contienen dos o más tipos de datos
- ▶ En cada momento se puede referenciar un tipo de dato

Uniones – Operaciones permitidas

- ▶ Tener acceso a los miembros de una unión utilizando el operador de miembro de estructura y el operador de apuntador de estructura.
- ▶ Asignar una unión a otra unión del mismo tipo.
- ▶ Tomar la dirección (`&`) de una unión.

Uniones – Operaciones permitidas

- ▶ Tener acceso a los miembros de una unión utilizando el operador de miembro de estructura y el operador de apuntador de estructura.
- ▶ Asignar una unión a otra unión del mismo tipo.
- ▶ Tomar la dirección (`&`) de una unión.

Se pueden comparar las uniones?

Uniones – Operaciones permitidas

- ▶ Tener acceso a los miembros de una unión utilizando el operador de miembro de estructura y el operador de apuntador de estructura.
- ▶ Asignar una unión a otra unión del mismo tipo.
- ▶ Tomar la dirección (`&`) de una unión.

Se pueden comparar las uniones? **NO**

Uniones – Operaciones permitidas

- ▶ Tener acceso a los miembros de una unión utilizando el operador de miembro de estructura y el operador de apuntador de estructura.
- ▶ Asignar una unión a otra unión del mismo tipo.
- ▶ Tomar la dirección (`&`) de una unión.

Se pueden comparar las uniones? **NO** . . . y las estructuras?

Uniones – Operaciones permitidas

- ▶ Tener acceso a los miembros de una unión utilizando el operador de miembro de estructura y el operador de apuntador de estructura.
- ▶ Asignar una unión a otra unión del mismo tipo.
- ▶ Tomar la dirección (`&`) de una unión.

Se pueden comparar las uniones? **NO** . . . y las estructuras? **NO**

Uniones – Operaciones permitidas

- ▶ Tener acceso a los miembros de una unión utilizando el operador de miembro de estructura y el operador de apuntador de estructura.
- ▶ Asignar una unión a otra unión del mismo tipo.
- ▶ Tomar la dirección (`&`) de una unión.

Se pueden comparar las uniones? **NO** ... y las estructuras? **NO**

Inicialización: Se puede inicializar en la definición con un valor del mismo tipo que el primer miembro de la unión

```
union numero {  
    int x;  
    float y;  
};  
  
union numero u1 = {10};  
union numero u1 = {0.02} /* Qué hace? */;
```

Ejemplo de union int y float

```
1 #include <stdio.h>
2
3 union int_float {
4     int entero;
5     float real;
6 };
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
```

Ejemplo de union int y float

```
1 #include <stdio.h>
2
3 union int_float {
4     int entero;
5     float real;
6 };
7
8 void imprimir_union_int_float(union int_float u)
9 {
10    printf("- El miembro INT de la union es: %d\n", u.entero);
11    printf("- El miembro FLOAT de la union es: %f\n", u.real);
12 }
13
14
15
16
17
18
19
20
21
22
```

Ejemplo de union int y float

```
1 #include <stdio.h>
2
3 union int_float {
4     int entero;
5     float real;
6 };
7
8 void imprimir_union_int_float(union int_float u)
9 {
10    printf("- El miembro INT de la union es: %d\n", u.entero);
11    printf("- El miembro FLOAT de la union es: %f\n", u.real);
12 }
13
14 int main(void)
15 {
16     union int_float u;
17
18     /* Solicitar valor 'int' e imprimir union */
19     /* Solicitar varlor 'float' e imprimir union */
20
21     return 0;
22 }
```

Más ejemplos

```
union uint_ucvec {  
    unsigned int uint;  
    unsigned char ucvec[4];  
} reg1;
```

Más ejemplos

```
union uint_ucvec {  
    unsigned int uint;  
    unsigned char ucvec[4];  
} reg1;  
  
for(i = 0; i < 4; i++)  
    printf("%d ", reg1.ucvec[i]);
```

Más ejemplos

```
union uint_ucvec {  
    unsigned int uint;  
    unsigned char ucvec[4];  
} reg1;  
  
for(i = 0; i < 4; i++)  
    printf("%d ", reg1.ucvec[i]);
```

```
union uint_bytes {  
    unsigned int uint;  
    struct {  
        unsigned char byte0;  
        unsigned char byte1;  
        unsigned char byte2;  
        unsigned char byte3;  
    } bytes;  
} reg2;
```

Más ejemplos

```
union uint_ucvec {  
    unsigned int uint;  
    unsigned char ucvec[4];  
} reg1;  
  
for(i = 0; i < 4; i++)  
    printf("%d ", reg1.ucvec[i]);
```

```
union uint_bytes {  
    unsigned int uint;  
    struct {  
        unsigned char byte0;  
        unsigned char byte1;  
        unsigned char byte2;  
        unsigned char byte3;  
    } bytes;  
} reg2;  
  
printf("%d ", reg2.bytes.byte0);  
printf("%d ", reg2.bytes.byte1);  
printf("%d ", reg2.bytes.byte2);  
printf("%d ", reg2.bytes.byte3);
```

Más ejemplos

```
union uint_ucvec {  
    unsigned int uint;  
    unsigned char ucvec[4];  
} reg1;  
  
for(i = 0; i < 4; i++)  
    printf("%d ", reg1.ucvec[i]);
```

```
union uint {  
    unsigned int uint;  
    unsigned char ucvec[4];  
    struct {  
        unsigned char byte0;  
        unsigned char byte1;  
        unsigned char byte2;  
        unsigned char byte3;  
    } bytes;  
};
```

```
union uint_bytes {  
    unsigned int uint;  
    struct {  
        unsigned char byte0;  
        unsigned char byte1;  
        unsigned char byte2;  
        unsigned char byte3;  
    } bytes;  
} reg2;  
  
printf("%d ", reg2.bytes.byte0);  
printf("%d ", reg2.bytes.byte1);  
printf("%d ", reg2.bytes.byte2);  
printf("%d ", reg2.bytes.byte3);
```


Campos de bits

Permite definir el **número de bits** en el cual se almacenan los miembros **unsigned** o **int** de una estructura o de una union.

Campos de bits

Permite definir el **número de bits** en el cual se almacenan los miembros **unsigned** o **int** de una estructura o de una union.

Los miembros de los campos de bits deben ser declarados como **unsigned** o **int**

Campos de bits

Permite definir el **número de bits** en el cual se almacenan los miembros **unsigned** o **int** de una estructura o de una union.

Los miembros de los campos de bits deben ser declarados como **unsigned** o **int**

```
struct bitCard {  
    unsigned face : 4;  
    unsigned suit : 2;  
    unsigned color : 1;  
};
```

- ▶ Nombre del campo seguido de dos puntos : y una constante entera del *ancho del campo*
- ▶ La cantidad de bits se fija según el rango de valores de cada miembro
- ▶ El acceso a los miembros se realiza como en cualquier estructura

Campos de bits – Ejemplos

Byte/registro para manejo de colores

```
struct RGB_color {  
    unsigned char r : 2; /* 2-bits */  
    unsigned char g : 2;  
    unsigned char b : 2;  
    unsigned char : 2; /* padding */  
};
```

Campos de bits – Ejemplos

Byte/registro para manejo de colores

```
struct RGB_color {  
    unsigned char r : 2; /* 2-bits */  
    unsigned char g : 2;  
    unsigned char b : 2;  
    unsigned char : 2; /* padding */  
};
```

Y si se quisiera modificar los bits RGB todos juntos?

Campos de bits – Ejemplos

Byte/registro para manejo de colores

```
struct RGB_color {  
    unsigned char r : 2; /* 2-bits */  
    unsigned char g : 2;  
    unsigned char b : 2;  
    unsigned char : 2; /* padding */  
};
```

Y si se quisiera modificar los bits RGB todos juntos?

```
union RGB_color {  
    struct {  
        unsigned char r:2, g:2, b:2;  
        unsigned char : 2;  
    };  
    struct {  
        unsigned char rgb : 6;  
        unsigned char : 2;  
    };  
};
```


Actividad práctica

1. Escribir un programa que defina una union entre un `float` y un vector de cuatro `unsigned char` (4 bytes) e imprima los campos. La interacción con el usuario debe ser la siguiente:

```
Ingrese un valor real: 3.14
- El valor del FLOAT es: 3.140
- El vector de UCHAR es: 195 245 72 64
```

Actividad práctica

1. Escribir un programa que defina una union entre un `float` y un vector de cuatro `unsigned char` (4 bytes) e imprima los campos. La interacción con el usuario debe ser la siguiente:

```
Ingrese un valor real: 3.14
- El valor del FLOAT es: 3.140
- El vector de UCHAR es: 195 245 72 64
```

2. Modificar la función que imprime un `unsigned char` en binario (`imprimir_binario`) para que no modifique el valor a imprimir. Modificar el prototipo de función para que el parámetro sea `const`.

Actividad práctica

1. Escribir un programa que defina una union entre un `float` y un vector de cuatro `unsigned char` (4 bytes) e imprima los campos. La interacción con el usuario debe ser la siguiente:

```
Ingrese un valor real: 3.14
- El valor del FLOAT es: 3.140
- El vector de UCHAR es: 195 245 72 64
```

2. Modificar la función que imprime un `unsigned char` en binario (`imprimir_binario`) para que no modifique el valor a imprimir. Modificar el prototipo de función para que el parámetro sea `const`.
3. Modificar la función `imprimir_binario` para que imprima cualquier variable de tipo entero utilizando un `typedef`.

Actividad práctica

1. Escribir un programa que defina una union entre un `float` y un vector de cuatro `unsigned char` (4 bytes) e imprima los campos. La interacción con el usuario debe ser la siguiente:

```
Ingrese un valor real: 3.14
- El valor del FLOAT es: 3.140
- El vector de UCHAR es: 195 245 72 64
```

2. Modificar la función que imprime un `unsigned char` en binario (`imprimir_binario`) para que no modifique el valor a imprimir. Modificar el prototipo de función para que el parámetro sea `const`.
3. Modificar la función `imprimir_binario` para que imprima cualquier variable de tipo entero utilizando un `typedef`.
4. Modificar la función `imprimir_binario` para que imprima un `float`.

Actividad práctica

- Escribir un programa que defina una union entre un `float` y un vector de cuatro `unsigned char` (4 bytes) e imprima los campos. La interacción con el usuario debe ser la siguiente:

```
Ingrese un valor real: 3.14
- El valor del FLOAT es: 3.140
- El vector de UCHAR es: 195 245 72 64
```

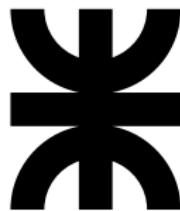
- Modificar la función que imprime un `unsigned char` en binario (`imprimir_binario`) para que no modifique el valor a imprimir. Modificar el prototipo de función para que el parámetro sea `const`.
- Modificar la función `imprimir_binario` para que imprima cualquier variable de tipo entero utilizando un `typedef`.
- Modificar la función `imprimir_binario` para que imprima un `float`.
- Modificar el programa anterior para que reciba el valor `float` por la línea de comandos, para que pueda ser ejecutado y muestre la salida como:

```
./float2bin 123.123
123.123001 = 11111010 00111110 11110110 01000010
```


Informática II

Asignación dinámica de memoria

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2021 –

Asignación dinámica de memoria

- ▶ Asignación dinámica de memoria es la capacidad de obtener más memoria para datos en tiempo de ejecución de un programa.

Asignación dinámica de memoria

- ▶ Asignación dinámica de memoria es la capacidad de obtener más memoria para datos en tiempo de ejecución de un programa.
- ▶ Se puede solicitar y devolver memoria al sistema operativo.

Asignación dinámica de memoria

- ▶ Asignación dinámica de memoria es la capacidad de obtener más memoria para datos en tiempo de ejecución de un programa.
- ▶ Se puede solicitar y devolver memoria al sistema operativo.
- ▶ La asignación dinámica de memoria, o memoria dinámica tiene lugar en un segmento llamado *heap*.

Asignación dinámica de memoria

- ▶ Asignación dinámica de memoria es la capacidad de obtener más memoria para datos en tiempo de ejecución de un programa.
- ▶ Se puede solicitar y devolver memoria al sistema operativo.
- ▶ La asignación dinámica de memoria, o memoria dinámica tiene lugar en un segmento llamado *heap*.
- ▶ Las funciones `malloc` y `free`, junto al operador `sizeof` se utilizan en la asignación dinámica de memoria (archivo de cabecera `stdlib.h`).

Disposición de la memoria en un programas C

- ▶ Cuando se carga un programa en memoria, este queda organizado en bloques llamados *segmentos*.

Disposición de la memoria en un programas C

- ▶ Cuando se carga un programa en memoria, este queda organizado en bloques llamados *segmentos*.
- ▶ Cada segmento tiene permisos de lectura, escritura y ejecución.

Disposición de la memoria en un programas C

- ▶ Cuando se carga un programa en memoria, este queda organizado en bloques llamados *segmentos*.
- ▶ Cada segmento tiene permisos de lectura, escritura y ejecución.

Los segmentos son:

Disposición de la memoria en un programas C

- ▶ Cuando se carga un programa en memoria, este queda organizado en bloques llamados *segmentos*.
- ▶ Cada segmento tiene permisos de lectura, escritura y ejecución.

Los segmentos son:

text: contiene código binario del programa (solo lectura). Una única copia para varias instancias del programa (ver error de -lm).

Disposición de la memoria en un programas C

- ▶ Cuando se carga un programa en memoria, este queda organizado en bloques llamados *segmentos*.
- ▶ Cada segmento tiene permisos de lectura, escritura y ejecución.

Los segmentos son:

text: contiene código binario del programa (solo lectura). Una única copia para varias instancias del programa (ver error de `-lm`).

data: subdividido en dos:

Disposición de la memoria en un programas C

- ▶ Cuando se carga un programa en memoria, este queda organizado en bloques llamados *segmentos*.
- ▶ Cada segmento tiene permisos de lectura, escritura y ejecución.

Los segmentos son:

text: contiene código binario del programa (solo lectura). Una única copia para varias instancias del programa (ver error de `-lm`).

data: subdividido en dos:

- ▶ Datos no inicializados – variables globales y estáticas.

Disposición de la memoria en un programas C

- ▶ Cuando se carga un programa en memoria, este queda organizado en bloques llamados *segmentos*.
- ▶ Cada segmento tiene permisos de lectura, escritura y ejecución.

Los segmentos son:

text: contiene código binario del programa (solo lectura). Una única copia para varias instancias del programa (ver error de `-lm`).

data: subdividido en dos:

- ▶ Datos no inicializados – variables globales y estáticas.
- ▶ Datos inicializados explícitamente – variables globales, estáticas, y datos constantes (solo lectura).

Disposición de la memoria en un programas C

- ▶ Cuando se carga un programa en memoria, este queda organizado en bloques llamados *segmentos*.
- ▶ Cada segmento tiene permisos de lectura, escritura y ejecución.

Los segmentos son:

text: contiene código binario del programa (solo lectura). Una única copia para varias instancias del programa (ver error de `-lm`).

data: subdividido en dos:

- ▶ Datos no inicializados – variables globales y estáticas.
- ▶ Datos inicializados explícitamente – variables globales, estáticas, y datos constantes (solo lectura).

stack: para almacenar variables locales, argumentos pasados a funciones, y dirección de retorno (crece hacia abajo) [stack frames].

Disposición de la memoria en un programas C

- ▶ Cuando se carga un programa en memoria, este queda organizado en bloques llamados *segmentos*.
- ▶ Cada segmento tiene permisos de lectura, escritura y ejecución.

Los segmentos son:

text: contiene código binario del programa (solo lectura). Una única copia para varias instancias del programa (ver error de `-lm`).

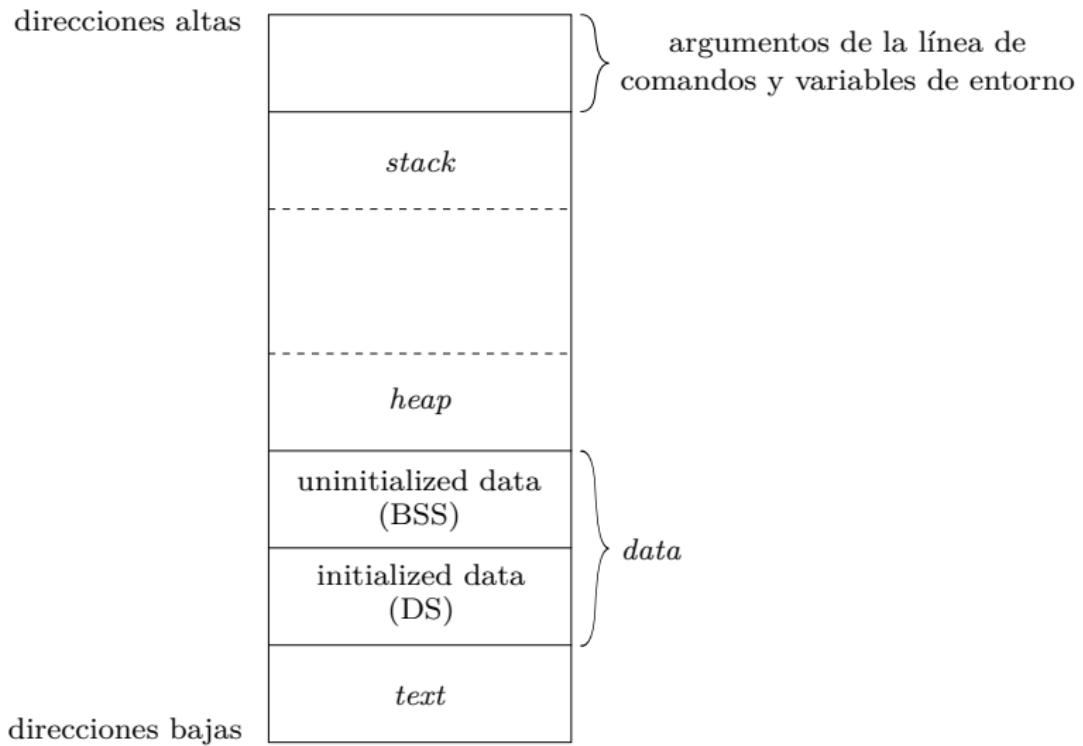
data: subdividido en dos:

- ▶ Datos no inicializados – variables globales y estáticas.
- ▶ Datos inicializados explícitamente – variables globales, estáticas, y datos constantes (solo lectura).

stack: para almacenar variables locales, argumentos pasados a funciones, y dirección de retorno (crece hacia abajo) [stack frames].

heap: cuando se asigna memoria (`malloc`) en tiempo de ejecución la memoria se obtiene del *heap* (crece hacia arriba).

Disposición de la memoria en un programas C



Segmento de datos – Ejemplos

Una cadena definida como

```
char s[] = "Hola mundo";
```

o un enunciado

```
int debug = 1;
```

fueras de `main` (global) almacena las variables en el área de datos inicializados como read-write.

Segmento de datos – Ejemplos

Una cadena definida como

```
char s[] = "Hola mundo";
```

o un enunciado

```
int debug = 1;
```

fueras de `main` (global) almacena las variables en el área de datos inicializados como read-write.

Un enunciado global como

```
const char *string = "Hola mundo";
```

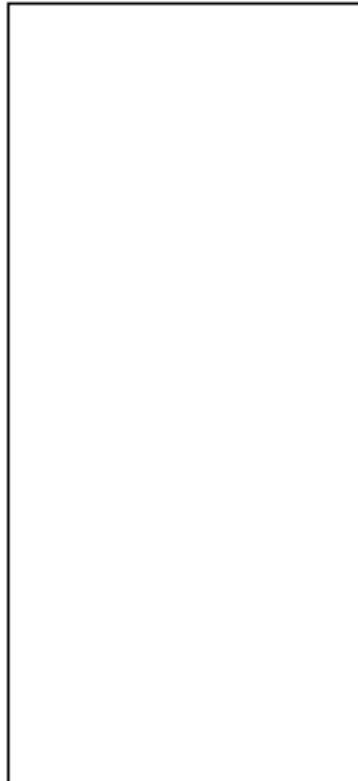
almacena la cadena literal en el área inicializada como read-only, y la variable puntero en el área inicializada como read-write.

Segmento *stack* – Ejemplo

```
1 #include <stdio.h>
2 int res; /* En BSS */
3
4 int cuadrado(int x)
5 {
6     return x*x;
7 }
8
9 int cuadradoDeSuma(int x, int y)
10 {
11     int z = cuadrado(x+y);
12     return z;
13 }
14
15 int main(void)
16 {
17     int a = 2, b = 3;
18     res = cuadradoDeSuma(a, b);
19     printf("Resultado: %d\n", res);
20     return 0;
21 }
```

Segmento *stack* – Ejemplo

```
1 #include <stdio.h>
2 int res; /* En BSS */
3
4 int cuadrado(int x)
5 {
6     return x*x;
7 }
8
9 int cuadradoDeSuma(int x, int y)
10 {
11     int z = cuadrado(x+y);
12     return z;
13 }
14
15 int main(void)
16 {
17     int a = 2, b = 3;
18     res = cuadradoDeSuma(a, b);
19     printf("Resultado: %d\n", res);
20     return 0;
21 }
```



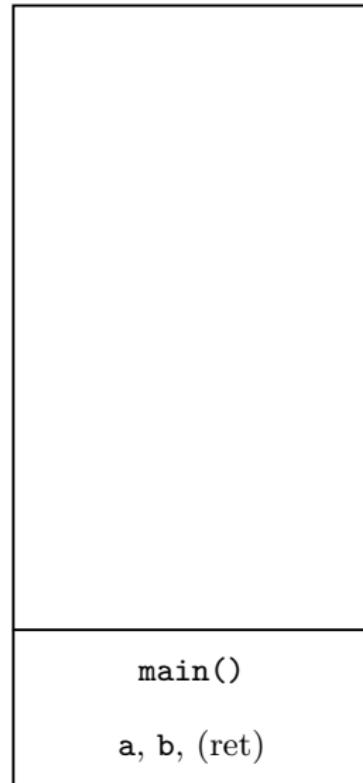
Segmento *stack* – Ejemplo

```
1 #include <stdio.h>
2 int res; /* En BSS */
3
4 int cuadrado(int x)
5 {
6     return x*x;
7 }
8
9 int cuadradoDeSuma(int x, int y)
10 {
11     int z = cuadrado(x+y);
12     return z;
13 }
14
15 int main(void)
16 {
17     int a = 2, b = 3;
18     res = cuadradoDeSuma(a, b);
19     printf("Resultado: %d\n", res);
20     return 0;
21 }
```



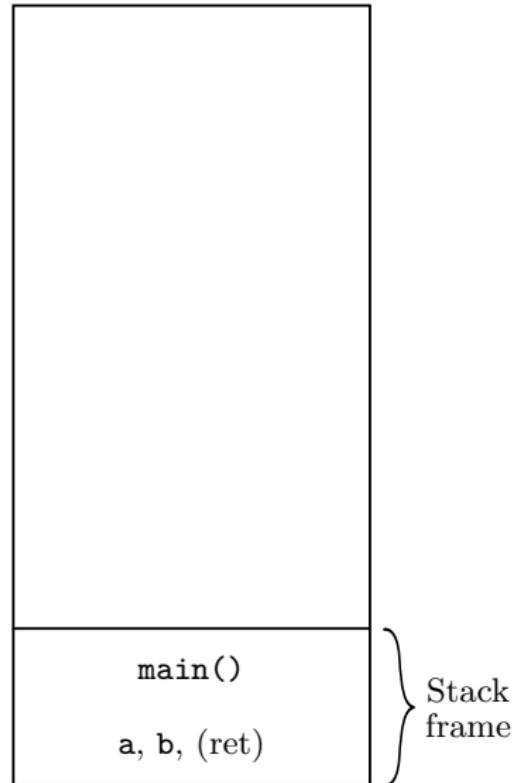
Segmento *stack* – Ejemplo

```
1 #include <stdio.h>
2 int res; /* En BSS */
3
4 int cuadrado(int x)
5 {
6     return x*x;
7 }
8
9 int cuadradoDeSuma(int x, int y)
10 {
11     int z = cuadrado(x+y);
12     return z;
13 }
14
15 int main(void)
16 {
17     int a = 2, b = 3;
18     res = cuadradoDeSuma(a, b);
19     printf("Resultado: %d\n", res);
20     return 0;
21 }
```



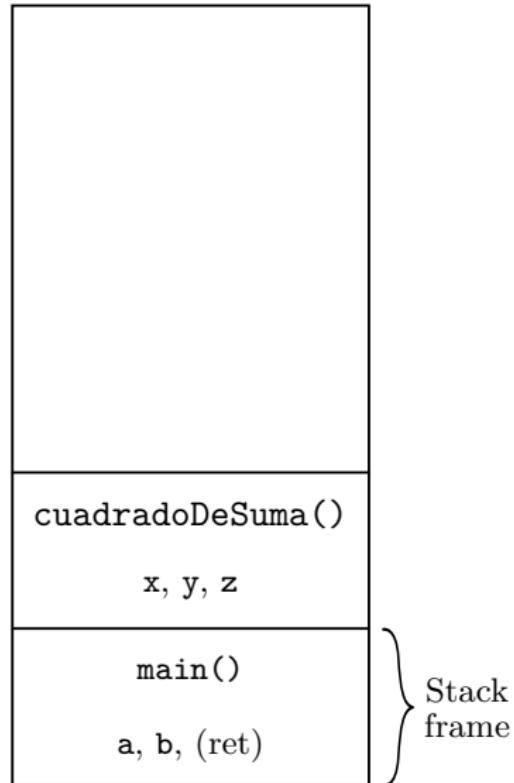
Segmento *stack* – Ejemplo

```
1 #include <stdio.h>
2 int res; /* En BSS */
3
4 int cuadrado(int x)
5 {
6     return x*x;
7 }
8
9 int cuadradoDeSuma(int x, int y)
10 {
11     int z = cuadrado(x+y);
12     return z;
13 }
14
15 int main(void)
16 {
17     int a = 2, b = 3;
18     res = cuadradoDeSuma(a, b);
19     printf("Resultado: %d\n", res);
20     return 0;
21 }
```



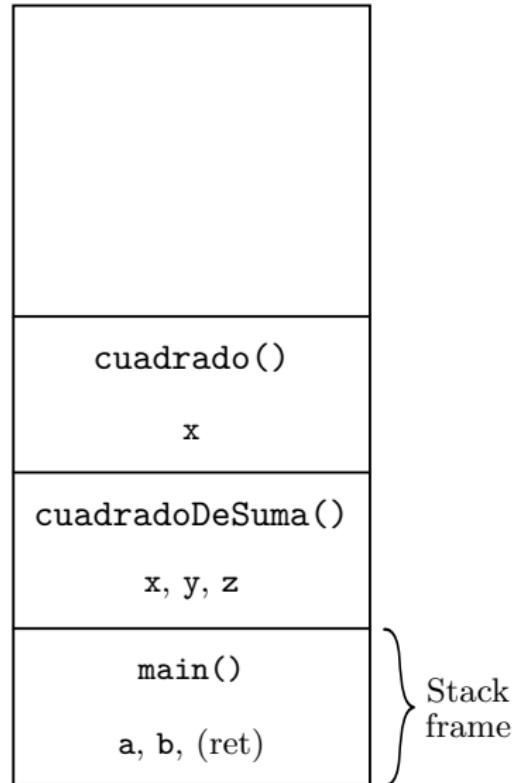
Segmento *stack* – Ejemplo

```
1 #include <stdio.h>
2 int res; /* En BSS */
3
4 int cuadrado(int x)
5 {
6     return x*x;
7 }
8
9 int cuadradoDeSuma(int x, int y)
10 {
11     int z = cuadrado(x+y);
12     return z;
13 }
14
15 int main(void)
16 {
17     int a = 2, b = 3;
18     res = cuadradoDeSuma(a, b);
19     printf("Resultado: %d\n", res);
20     return 0;
21 }
```



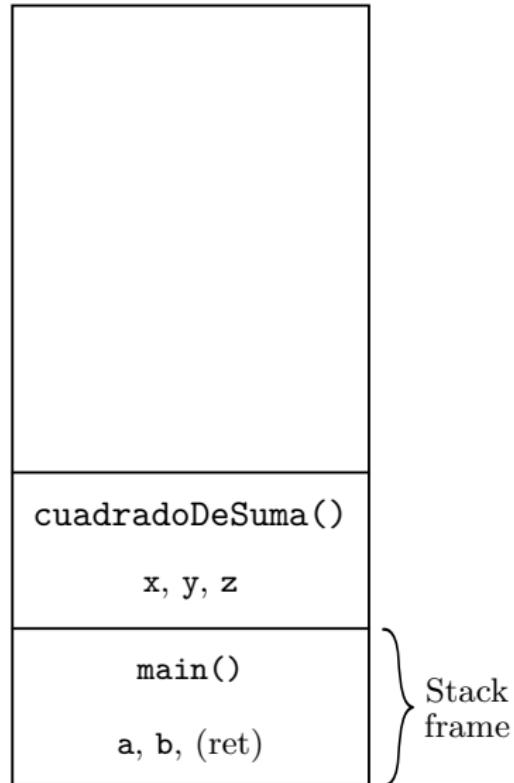
Segmento *stack* – Ejemplo

```
1 #include <stdio.h>
2 int res; /* En BSS */
3
4 int cuadrado(int x)
5 {
6     return x*x;
7 }
8
9 int cuadradoDeSuma(int x, int y)
10 {
11     int z = cuadrado(x+y);
12     return z;
13 }
14
15 int main(void)
16 {
17     int a = 2, b = 3;
18     res = cuadradoDeSuma(a, b);
19     printf("Resultado: %d\n", res);
20     return 0;
21 }
```



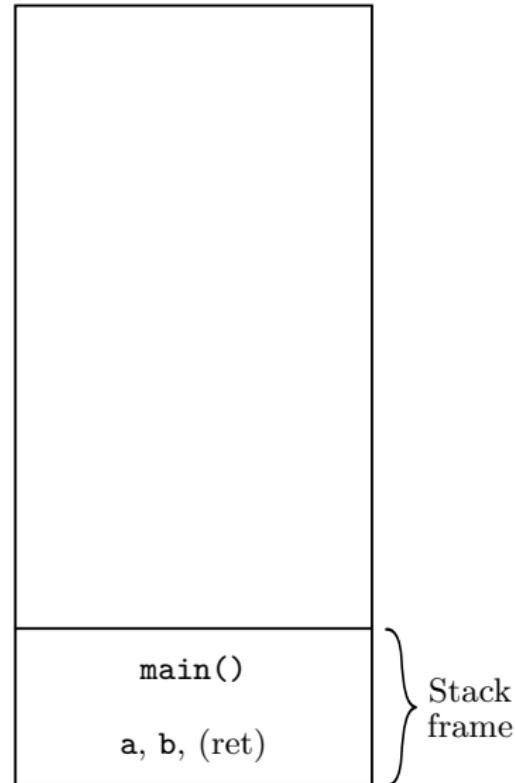
Segmento *stack* – Ejemplo

```
1 #include <stdio.h>
2 int res; /* En BSS */
3
4 int cuadrado(int x)
5 {
6     return x*x;
7 }
8
9 int cuadradoDeSuma(int x, int y)
10 {
11     int z = cuadrado(x+y);
12     return z;
13 }
14
15 int main(void)
16 {
17     int a = 2, b = 3;
18     res = cuadradoDeSuma(a, b);
19     printf("Resultado: %d\n", res);
20     return 0;
21 }
```



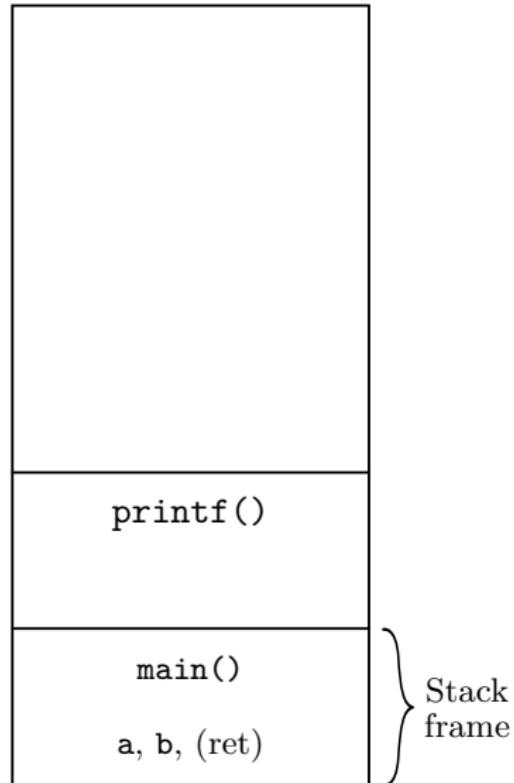
Segmento *stack* – Ejemplo

```
1 #include <stdio.h>
2 int res; /* En BSS */
3
4 int cuadrado(int x)
5 {
6     return x*x;
7 }
8
9 int cuadradoDeSuma(int x, int y)
10 {
11     int z = cuadrado(x+y);
12     return z;
13 }
14
15 int main(void)
16 {
17     int a = 2, b = 3;
18     res = cuadradoDeSuma(a, b);
19     printf("Resultado: %d\n", res);
20     return 0;
21 }
```



Segmento *stack* – Ejemplo

```
1 #include <stdio.h>
2 int res; /* En BSS */
3
4 int cuadrado(int x)
5 {
6     return x*x;
7 }
8
9 int cuadradoDeSuma(int x, int y)
10 {
11     int z = cuadrado(x+y);
12     return z;
13 }
14
15 int main(void)
16 {
17     int a = 2, b = 3;
18     res = cuadradoDeSuma(a, b);
19     printf("Resultado: %d\n", res);
20     return 0;
21 }
```



Asignación dinámica de memoria

Ciclo de vida de las variables en el *heap* son:

1. asignar espacio a la variable usando `malloc()` o `calloc()`

Asignación dinámica de memoria

Ciclo de vida de las variables en el *heap* son:

1. asignar espacio a la variable usando `malloc()` o `calloc()`
2. redimensionar la variable utilizando `realloc()` [Opcional]

Asignación dinámica de memoria

Ciclo de vida de las variables en el *heap* son:

1. asignar espacio a la variable usando `malloc()` o `calloc()`
2. redimensionar la variable utilizando `realloc()` [Opcional]
3. liberar la memoria con `free()`

Asignación dinámica de memoria

Ciclo de vida de las variables en el *heap* son:

1. asignar espacio a la variable usando `malloc()` o `calloc()`
2. redimensionar la variable utilizando `realloc()` [Opcional]
3. liberar la memoria con `free()`

Prototipos de funciones – archivo de cabecera `stdlib.h`

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

Asignación dinámica de memoria

Ciclo de vida de las variables en el *heap* son:

1. asignar espacio a la variable usando `malloc()` o `calloc()`
2. redimensionar la variable utilizando `realloc()` [Opcional]
3. liberar la memoria con `free()`

Prototipos de funciones – archivo de cabecera `stdlib.h`

```
void *malloc(size_t size);  
  
void *calloc(size_t nmemb, size_t size);
```

Asignación dinámica de memoria

Ciclo de vida de las variables en el *heap* son:

1. asignar espacio a la variable usando `malloc()` o `calloc()`
2. redimensionar la variable utilizando `realloc()` [Opcional]
3. liberar la memoria con `free()`

Prototipos de funciones – archivo de cabecera `stdlib.h`

```
void *malloc(size_t size);  
  
void *calloc(size_t nmemb, size_t size);  
  
void *realloc(void *ptr, size_t size);
```

Asignación dinámica de memoria

Ciclo de vida de las variables en el *heap* son:

1. asignar espacio a la variable usando `malloc()` o `calloc()`
2. redimensionar la variable utilizando `realloc()` [Opcional]
3. liberar la memoria con `free()`

Prototipos de funciones – archivo de cabecera `stdlib.h`

```
void *malloc(size_t size);

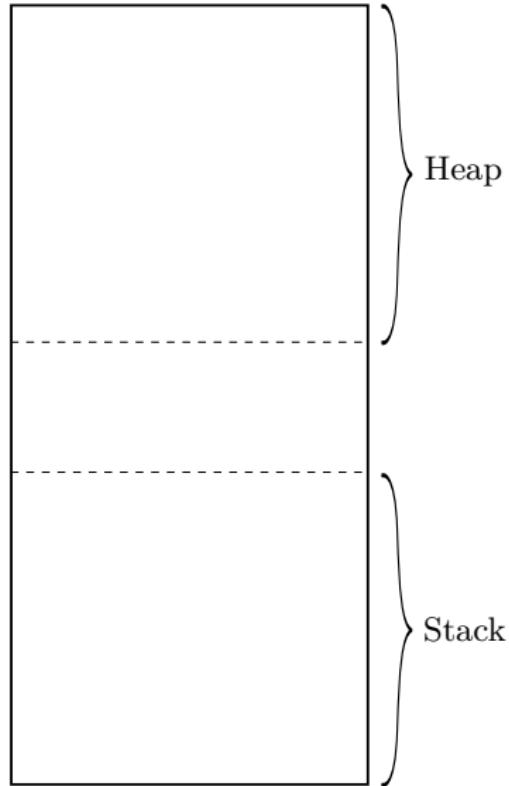
void *calloc(size_t nmemb, size_t size);

void *realloc(void *ptr, size_t size);

void free(void *ptr);
```

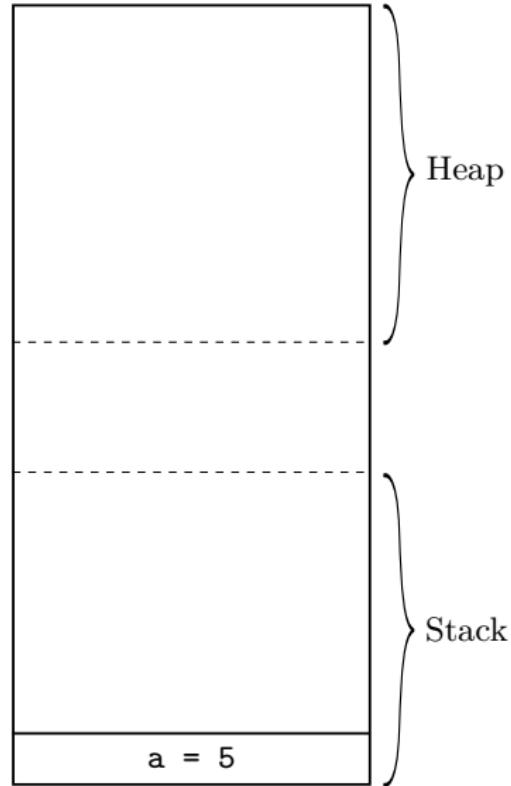
Segmento *heap* – Ejemplo

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int a = 5; /* ¿Dónde está? */
7
8
9
10
11
12
13
14
15
16
17     return 0;
18 }
```



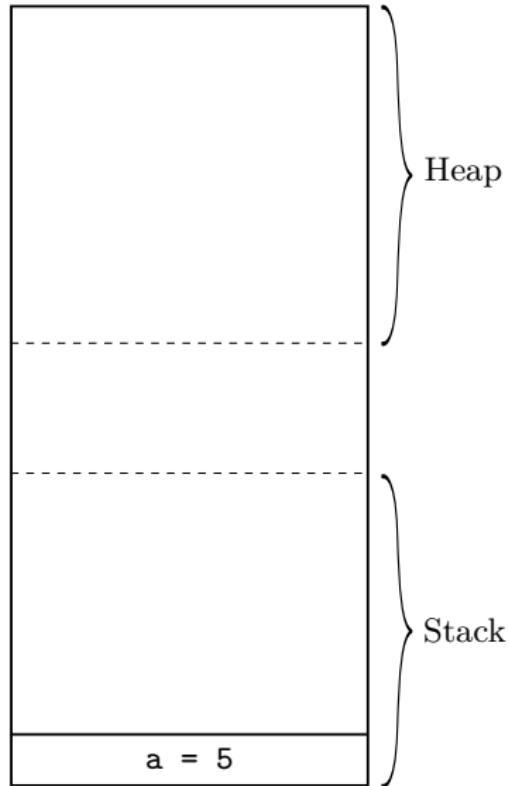
Segmento *heap* – Ejemplo

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int a = 5; /* ¿Dónde está? */
7
8
9
10
11
12
13
14
15
16
17     return 0;
18 }
```



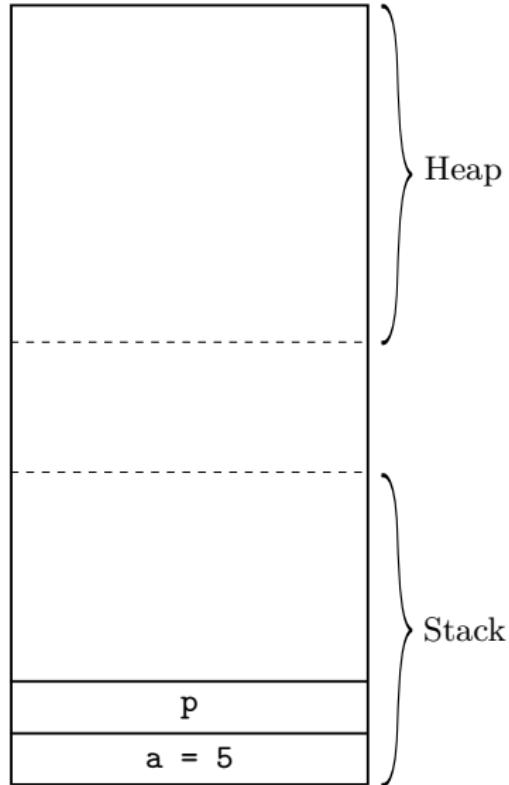
Segmento *heap* – Ejemplo

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int a = 5; /* En stack */
7
8     int *p;
9
10
11
12
13
14
15
16
17     return 0;
18 }
```



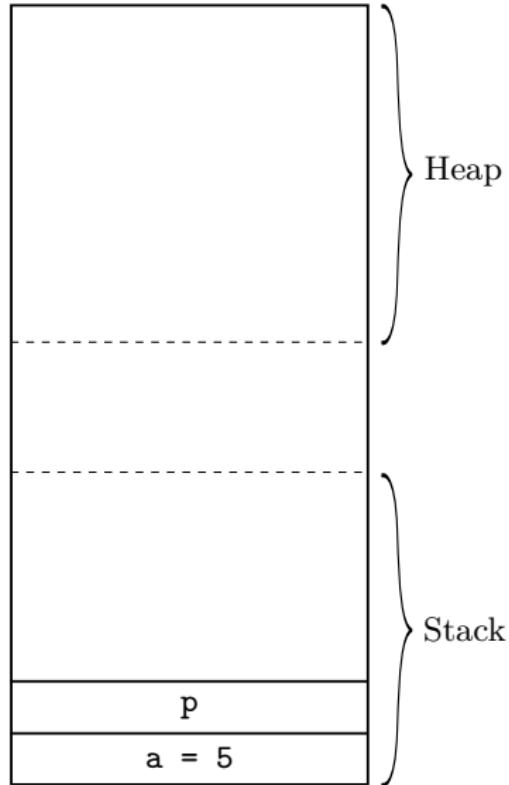
Segmento *heap* – Ejemplo

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int a = 5; /* En stack */
7
8     int *p;
9
10
11
12
13
14
15
16
17     return 0;
18 }
```



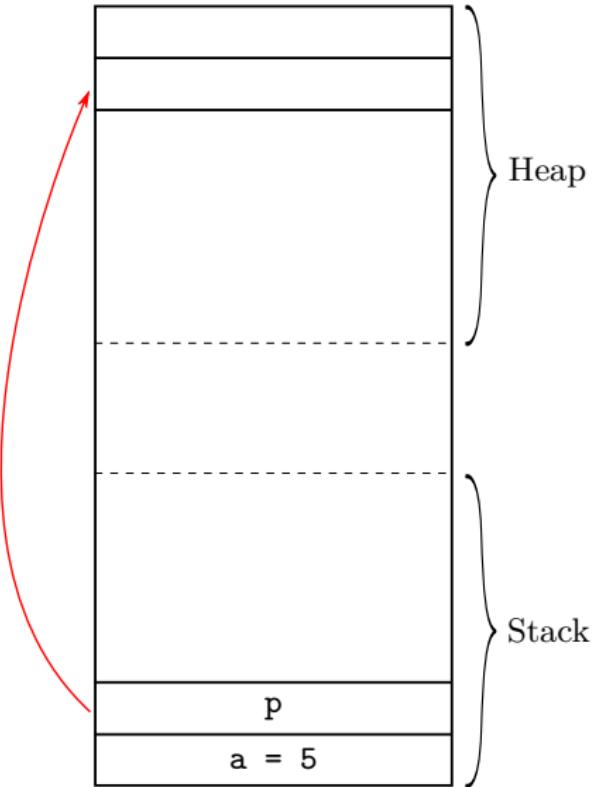
Segmento *heap* – Ejemplo

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int a = 5; /* En stack */
7
8     int *p;
9     p = (int *)malloc(sizeof(int));
10
11
12
13
14
15
16
17     return 0;
18 }
```



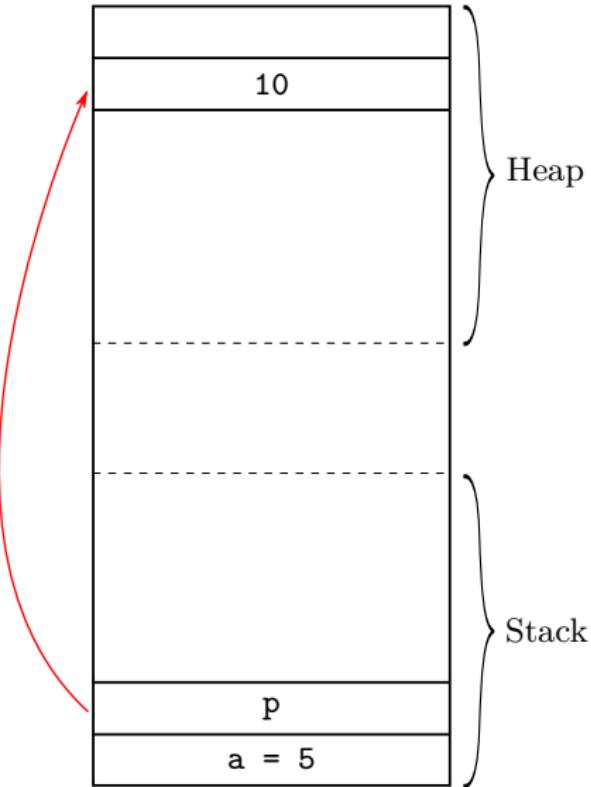
Segmento *heap* – Ejemplo

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int a = 5; /* En stack */
7
8     int *p;
9     p = (int *)malloc(sizeof(int));
10
11
12
13
14
15
16
17     return 0;
18 }
```



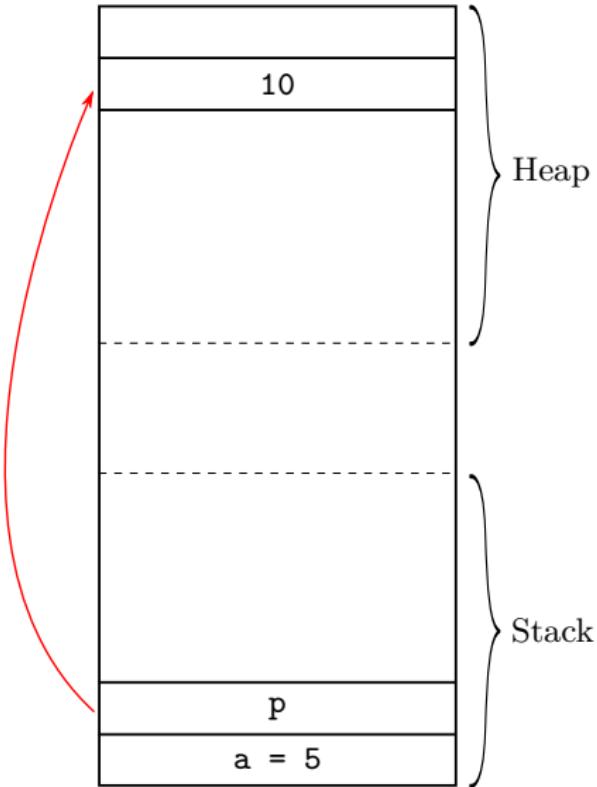
Segmento *heap* – Ejemplo

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int a = 5; /* En stack */
7
8     int *p;
9     p = (int *)malloc(sizeof(int));
10    *p = 10;
11
12
13
14
15
16
17    return 0;
18 }
```



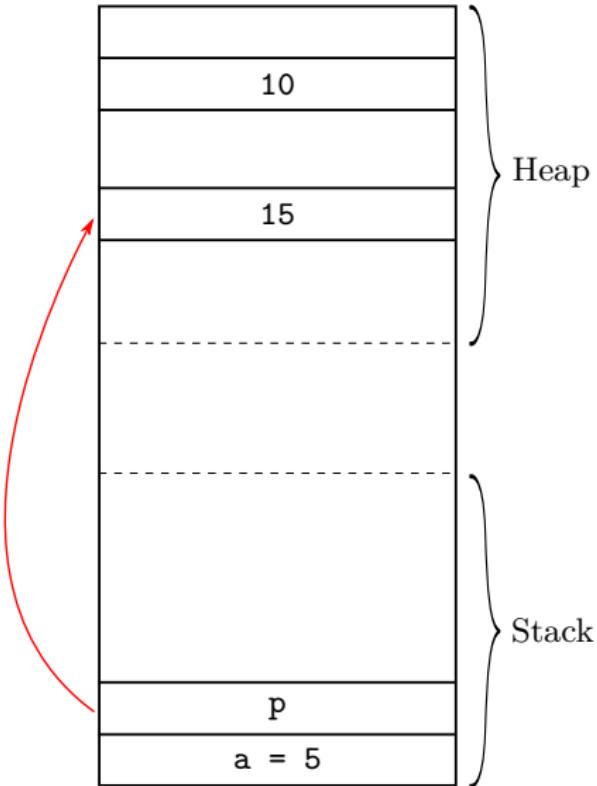
Segmento *heap* – Ejemplo

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int a = 5; /* En stack */
7
8     int *p;
9     p = (int *)malloc(sizeof(int));
10    *p = 10;
11
12
13    p = (int *)malloc(sizeof(int));
14    *p = 15;
15
16
17    return 0;
18 }
```



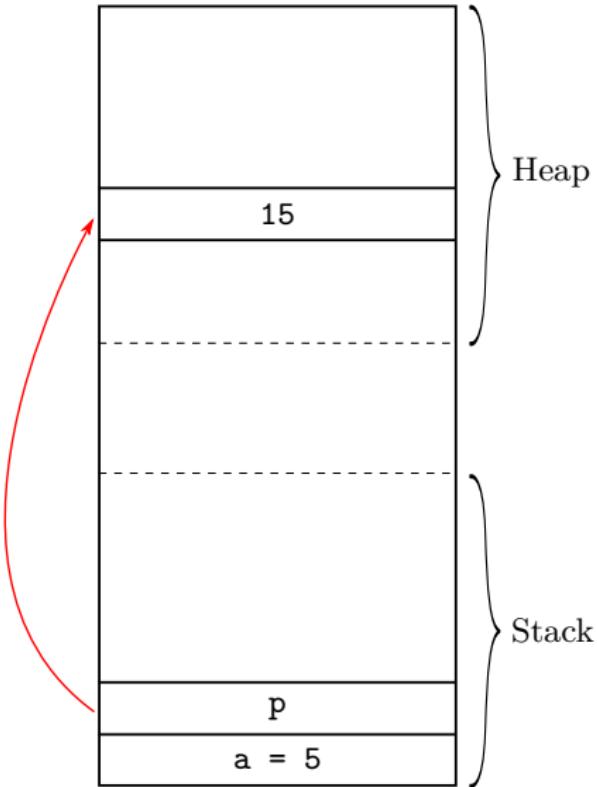
Segmento *heap* – Ejemplo

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int a = 5; /* En stack */
7
8     int *p;
9     p = (int *)malloc(sizeof(int));
10    *p = 10;
11
12
13    p = (int *)malloc(sizeof(int));
14    *p = 15;
15
16
17    return 0;
18 }
```



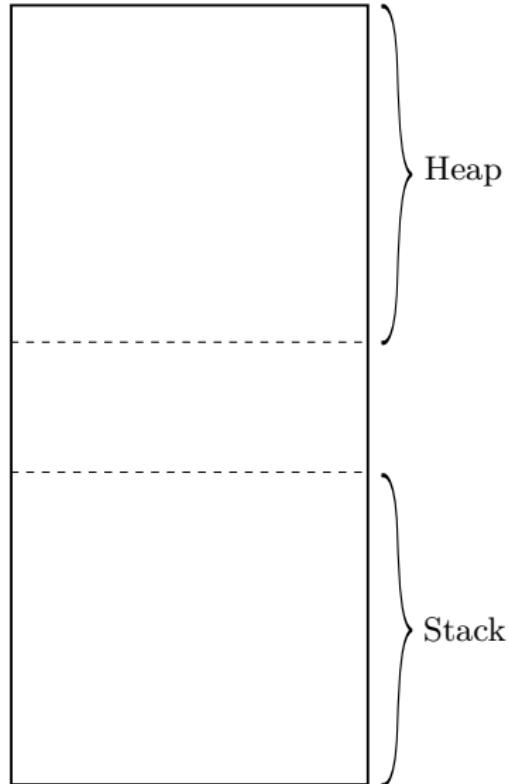
Segmento *heap* – Ejemplo

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int a = 5; /* En stack */
7
8     int *p;
9     p = (int *)malloc(sizeof(int));
10    *p = 10;
11    free(p);
12
13    p = (int *)malloc(sizeof(int));
14    *p = 15;
15
16
17    return 0;
18 }
```



Segmento *heap* – Ejemplo

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int a = 5; /* En stack */
7
8     int *p;
9     p = (int *)malloc(sizeof(int));
10    *p = 10;
11    free(p);
12
13    p = (int *)malloc(sizeof(int));
14    *p = 15;
15    free(p);
16
17    return 0;
18 }
```



Asignación dinámica de memoria – Ejemplos

Tipos básicos y arreglos

```
int *edad = malloc(sizeof(int));
```

Asignación dinámica de memoria – Ejemplos

Tipos básicos y arreglos

```
int *edad = malloc(sizeof(int));
```

```
double *promedio = malloc(sizeof(double));
```

Asignación dinámica de memoria – Ejemplos

Tipos básicos y arreglos

```
int *edad = malloc(sizeof(int));  
  
double *promedio = malloc(sizeof(double));  
  
char *cadena = malloc(20 * sizeof(char));
```

Asignación dinámica de memoria – Ejemplos

Tipos básicos y arreglos

```
int *edad = malloc(sizeof(int));  
  
double *promedio = malloc(sizeof(double));  
  
char *cadena = malloc(20 * sizeof(char));  
  
float *lista = malloc(3 * sizeof(float));
```

Asignación dinámica de memoria – Ejemplos

Tipos básicos y arreglos

```
int *edad = malloc(sizeof(int));  
  
double *promedio = malloc(sizeof(double));  
  
char *cadena = malloc(20 * sizeof(char));  
  
float *lista = malloc(3 * sizeof(float));
```

Estructuras

```
struct paciente {  
    char apellido[20];  
    char nombre[20];  
    int edad;  
    float peso;  
    float altura;  
};  
struct paciente *jperez = malloc(sizeof(struct paciente));
```

Asignación dinámica de memoria – Ejemplos

Tipos básicos y arreglos

```
int *edad = malloc(sizeof(int));  
  
double *promedio = malloc(sizeof(double));  
  
char *cadena = malloc(20 * sizeof(char));  
  
float *lista = malloc(3 * sizeof(float));
```

Estructuras

```
struct paciente {  
    char apellido[20];  
    char nombre[20];  
    int edad;  
    float peso;  
    float altura;  
};  
struct paciente *jperez = malloc(sizeof(struct paciente));
```

No olvidar liberar la memoria con `free()` [Memory leak]

Actividad práctica

1. Escribir un programa que reserve espacio en memoria para las variables de tipos básicos del ejemplo, le asigne valores, imprima los valores, y libere la memoria.
2. Escribir un programa que calcule y muestre en pantalla el promedio de una serie de valores enteros. La interacción con el usuario es la siguiente:
 - ▶ Le solicita la cantidad de valores a promediar
 - ▶ Le solicita los datos
 - ▶ Imprime el promedio calculado

Calcular el promedio con una función que tenga el siguiente prototipo

```
float promedio(int * , int );
```

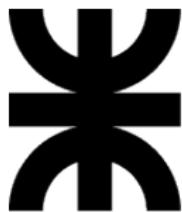
El programa debe reservar memoria para almacenar los datos previo a llamar a la función. Interacción con el usuario (entrada/salida):

```
Ingrese la cantidad de datos a promediar: 4
Ingrese el dato #1: 1
Ingrese el dato #2: 3
Ingrese el dato #3: 7
Ingrese el dato #4: 11
El promedio es: 5.500000
```


Informática II

Operadores a nivel de bits

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2021 –

Operadores a nivel de bits

Para manipular los bits individuales de los tipos de datos básicos (generalmente `unsigned`).

Operadores a nivel de bits

Para manipular los bits individuales de los tipos de datos básicos (generalmente `unsigned`).

- ▶ AND a nivel de bit: `&` (ampersand o et)

Operadores a nivel de bits

Para manipular los bits individuales de los tipos de datos básicos (generalmente `unsigned`).

- ▶ AND a nivel de bit: `&` (ampersand o et)
- ▶ OR inclusivo a nivel de bit: `|` (barra vertical o pleca)

Operadores a nivel de bits

Para manipular los bits individuales de los tipos de datos básicos (generalmente `unsigned`).

- ▶ AND a nivel de bit: `&` (ampersand o et)
- ▶ OR inclusivo a nivel de bit: `|` (barra vertical o pleca)
- ▶ OR exclusivo a nivel de bit: `^` (caret o sombrero)

Operadores a nivel de bits

Para manipular los bits individuales de los tipos de datos básicos (generalmente `unsigned`).

- ▶ AND a nivel de bit: `&` (ampersand o et)
- ▶ OR inclusivo a nivel de bit: `|` (barra vertical o pleca)
- ▶ OR exclusivo a nivel de bit: `^` (caret o sombrero)
- ▶ Desplazamiento a la izquierda: `<<` (menor que)

Operadores a nivel de bits

Para manipular los bits individuales de los tipos de datos básicos (generalmente `unsigned`).

- ▶ AND a nivel de bit: `&` (ampersand o et)
- ▶ OR inclusivo a nivel de bit: `|` (barra vertical o pleca)
- ▶ OR exclusivo a nivel de bit: `^` (caret o sombrero)
- ▶ Desplazamiento a la izquierda: `<<` (menor que)
- ▶ Desplazamiento a la derecha: `>>` (mayor que)

Operadores a nivel de bits

Para manipular los bits individuales de los tipos de datos básicos (generalmente `unsigned`).

- ▶ AND a nivel de bit: `&` (ampersand o et)
- ▶ OR inclusivo a nivel de bit: `|` (barra vertical o pleca)
- ▶ OR exclusivo a nivel de bit: `^` (caret o sombrero)
- ▶ Desplazamiento a la izquierda: `<<` (menor que)
- ▶ Desplazamiento a la derecha: `>>` (mayor que)
- ▶ Complemento: `~` (tilde)

Operadores a nivel de bits

Para manipular los bits individuales de los tipos de datos básicos (generalmente `unsigned`).

- ▶ AND a nivel de bit: `&` (ampersand o et)
- ▶ OR inclusivo a nivel de bit: `|` (barra vertical o pleca)
- ▶ OR exclusivo a nivel de bit: `^` (caret o sombrero)
- ▶ Desplazamiento a la izquierda: `<<` (menor que)
- ▶ Desplazamiento a la derecha: `>>` (mayor que)
- ▶ Complemento: `~` (tilde)

Tienen su equivalente en operadores de asignación:

`&=` `|=` `^=` `<<=` `>>=`

Operadores a nivel de bits

Para manipular los bits individuales de los tipos de datos básicos (generalmente `unsigned`).

- ▶ AND a nivel de bit: `&` (ampersand o et)
- ▶ OR inclusivo a nivel de bit: `|` (barra vertical o pleca)
- ▶ OR exclusivo a nivel de bit: `^` (caret o sombrero)
- ▶ Desplazamiento a la izquierda: `<<` (menor que)
- ▶ Desplazamiento a la derecha: `>>` (mayor que)
- ▶ Complemento: `~` (tilde)

Tienen su equivalente en operadores de asignación:

`&=` `|=` `^=` `<<=` `>>=`

Ejemplos:

```
a = a & b;  
a &= b;
```

```
a = a | b;  
a |= b;
```

```
a = a << 2;  
a <<= 2;
```

Operadores a nivel de bits

AND a nivel de bits

| a | b | a & b |
|---|---|-------|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Operadores a nivel de bits

AND a nivel de bits

| a | b | a & b |
|---|---|-------|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

$$5d = 0x5 = 0101$$

$$9d = 0x9 = 1001$$

$$0001 = 0x1 = 1d$$

Operadores a nivel de bits

AND a nivel de bits

| a | b | a & b |
|---|---|-------|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

$$5d = 0x5 = 0101$$

$$9d = 0x9 = 1001$$

$$0001 = 0x1 = 1d$$

$$185d = 0xB9 = 1011\ 1001$$

$$154d = 0x9A = 1001\ 1010$$

$$1001\ 1000 = 0x98 = 152d$$

Operadores a nivel de bits

AND a nivel de bits

| a | b | a & b |
|---|---|-------|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

$$\begin{array}{l} 5d = 0x5 = 0101 \\ 9d = 0x9 = 1001 \\ \hline 0001 = 0x1 = 1d \end{array}$$

$$\begin{array}{l} 185d = 0xB9 = 1011\ 1001 \\ 154d = 0x9A = 1001\ 1010 \\ \hline \end{array}$$

$$1001\ 1000 = 0x98 = 152d$$

OR a nivel de bits

| a | b | a b |
|---|---|-------|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Operadores a nivel de bits

AND a nivel de bits

| a | b | a & b |
|---|---|-------|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

$$5d = 0x5 = 0101$$

$$9d = 0x9 = 1001$$

$$0001 = 0x1 = 1d$$

$$185d = 0xB9 = 1011\ 1001$$

$$154d = 0x9A = 1001\ 1010$$

$$1001\ 1000 = 0x98 = 152d$$

OR a nivel de bits

| a | b | a b |
|---|---|-------|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

$$5d = 0x5 = 0101$$

$$9d = 0x9 = 1001$$

$$1101 = 0xD = 13d$$

Operadores a nivel de bits

AND a nivel de bits

| a | b | a & b |
|---|---|-------|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

$$\begin{array}{l} 5d = 0x5 = 0101 \\ 9d = 0x9 = 1001 \\ \hline 0001 = 0x1 = 1d \\ \\ 185d = 0xB9 = 1011\ 1001 \\ 154d = 0x9A = 1001\ 1010 \\ \hline 1001\ 1000 = 0x98 = 152d \end{array}$$

OR a nivel de bits

| a | b | a b |
|---|---|-------|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

$$\begin{array}{l} 5d = 0x5 = 0101 \\ 9d = 0x9 = 1001 \\ \hline 1101 = 0xD = 13d \\ \\ 185d = 0xB9 = 1011\ 1001 \\ 154d = 0x9A = 1001\ 1010 \\ \hline 1011\ 1011 = 0xBB = 187d \end{array}$$

Operadores a nivel de bits

OR a nivel de bits

| a | b | $a \wedge b$ | |
|---|---|--------------|----------------------------|
| 0 | 0 | 0 | $185d = 0xB9 = 1011\ 1001$ |
| 0 | 1 | 1 | $154d = 0x9A = 1001\ 1010$ |
| 1 | 0 | 1 | ----- |
| 1 | 1 | 0 | $0010\ 0011 = 0x23 = 35d$ |

Operadores a nivel de bits

OR a nivel de bits

| a | b | a ^ b | |
|---|---|-------|-------------------------|
| 0 | 0 | 0 | 185d = 0xB9 = 1011 1001 |
| 0 | 1 | 1 | 154d = 0x9A = 1001 1010 |
| 1 | 0 | 1 | ----- |
| 1 | 1 | 0 | 0010 0011 = 0x23 = 35d |

Complemento

$$\sim 170d = \sim(1010\ 1010) = 0101\ 0101 = 0x55 = 85d$$

Operadores a nivel de bits

OR a nivel de bits

| a | b | a ^ b | |
|---|---|-------|-------------------------|
| 0 | 0 | 0 | 185d = 0xB9 = 1011 1001 |
| 0 | 1 | 1 | 154d = 0x9A = 1001 1010 |
| 1 | 0 | 1 | ----- |
| 1 | 1 | 0 | 0010 0011 = 0x23 = 35d |

Complemento

$\sim 170d = \sim(1010 1010) = 0101 0101 = 0x55 = 85d$

Desplazamiento a la izquierda

$22d \ll 2$
 $0001\ 0110 \ll 2 = 0101\ 1000 = 0x58 = 88d$

Operadores a nivel de bits

OR a nivel de bits

| a | b | a ^ b | |
|---|---|-------|-------------------------|
| 0 | 0 | 0 | 185d = 0xB9 = 1011 1001 |
| 0 | 1 | 1 | 154d = 0x9A = 1001 1010 |
| 1 | 0 | 1 | ----- |
| 1 | 1 | 0 | 0010 0011 = 0x23 = 35d |

Complemento

$\sim 170d = \sim(1010 1010) = 0101 0101 = 0x55 = 85d$

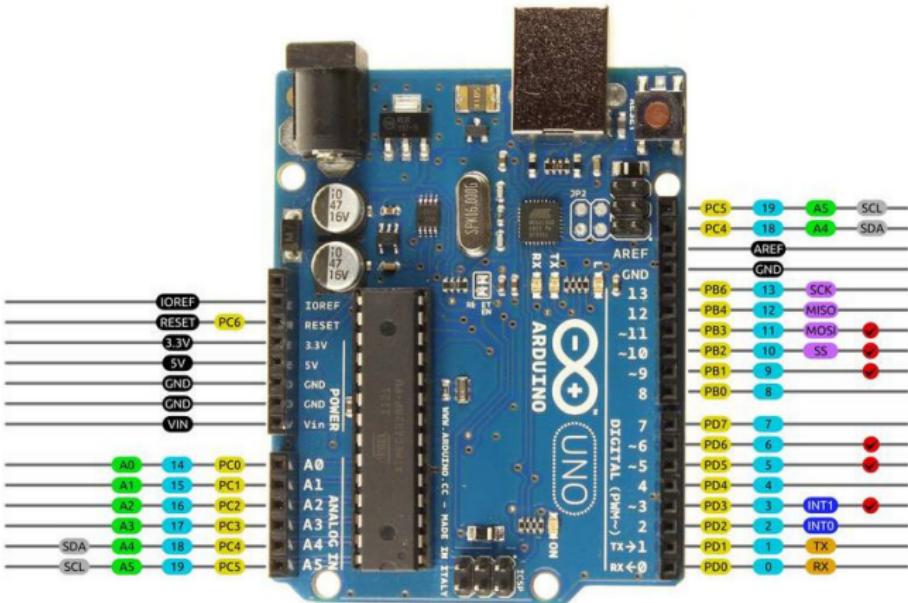
Desplazamiento a la izquierda

$22d \ll 2$
 $0001\ 0110 \ll 2 = 0101\ 1000 = 0x58 = 88d$

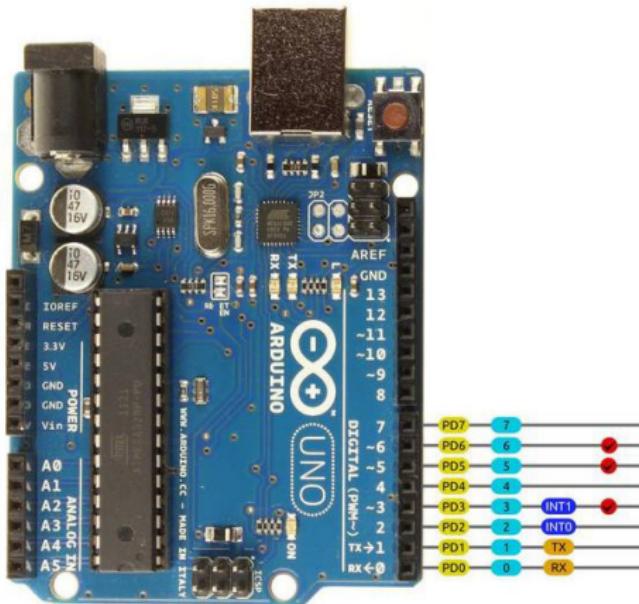
Desplazamiento a la derecha

$154d \gg 3$
 $1001\ 1010 \gg 3 = 0001\ 0011 = 0x13 = 19d$

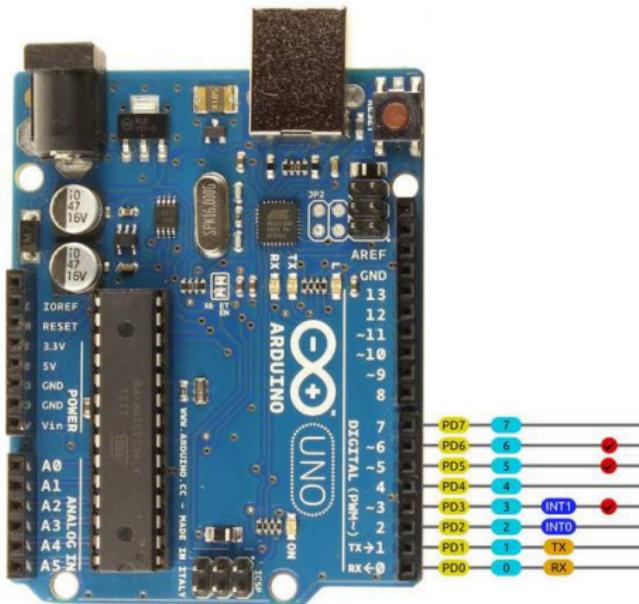
Operadores a nivel de bits



Operadores a nivel de bits

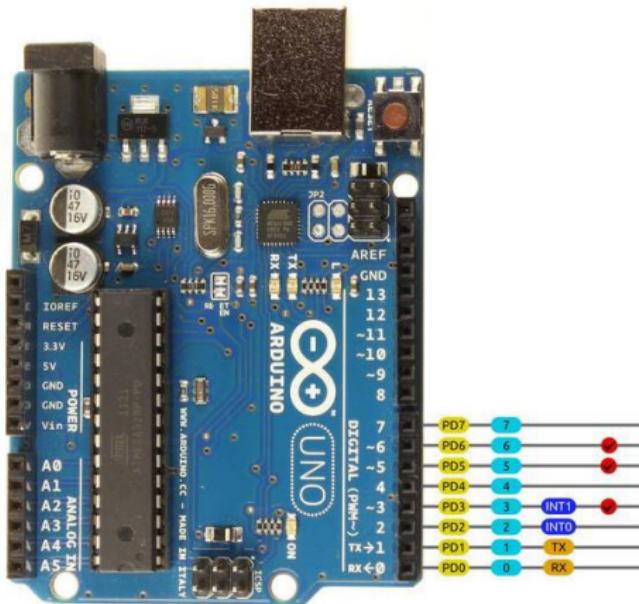


Operadores a nivel de bits



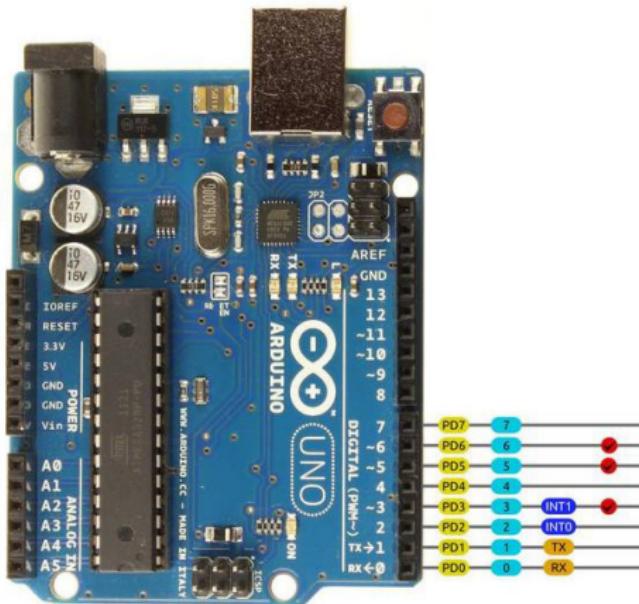
- ▶ ¿Cuántos bits tiene el puerto D?

Operadores a nivel de bits



- ▶ ¿Cuántos bits tiene el puerto D?
- ▶ ¿Qué tipo de dato se puede utilizar para almacenar el valor?

Operadores a nivel de bits



- ▶ ¿Cuántos bits tiene el puerto D?
- ▶ ¿Qué tipo de dato se puede utilizar para almacenar el valor?
`unsigned char puerto;`

Operadores a nivel de bits

Valor actual del puerto D:

| PD7 | PD6 | PD5 | PD4 | PD3 | PD2 | PD1 | PD0 |
|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 |

Operadores a nivel de bits

Valor actual del puerto D:

| PD7 | PD6 | PD5 | PD4 | PD3 | PD2 | PD1 | PD0 |
|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 |

- Valor en hexadecimal:

Operadores a nivel de bits

Valor actual del puerto D:

| PD7 | PD6 | PD5 | PD4 | PD3 | PD2 | PD1 | PD0 |
|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 |

- Valor en hexadecimal: 0x6B

Operadores a nivel de bits

Valor actual del puerto D:

| PD7 | PD6 | PD5 | PD4 | PD3 | PD2 | PD1 | PD0 |
|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 |

- Valor en hexadecimal: 0x6B

```
1 unsigned char puerto;
2
3 puerto = leerPuerto();
4 /* Modificar el valor de un bit */
5 escribirPuerto(puerto);
```

Operadores a nivel de bits

Valor actual del puerto D:

| PD7 | PD6 | PD5 | PD4 | PD3 | PD2 | PD1 | PD0 |
|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 |

- Valor en hexadecimal: 0x6B

```
1 unsigned char puerto;
2
3 puerto = leerPuerto();
4 /* Modificar el valor de un bit */
5 escribirPuerto(puerto);
```

Qué operación utilizar para:

- Poner a 1 un único bit sin alterar el resto

Operadores a nivel de bits

Valor actual del puerto D:

| PD7 | PD6 | PD5 | PD4 | PD3 | PD2 | PD1 | PD0 |
|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 |

- Valor en hexadecimal: 0x6B

```
1 unsigned char puerto;
2
3 puerto = leerPuerto();
4 /* Modificar el valor de un bit */
5 escribirPuerto(puerto);
```

Qué operación utilizar para:

- Poner a 1 un único bit sin alterar el resto
- Poner a 0 un único bit sin alterar el resto

Operadores a nivel de bits

Valor actual del puerto D:

| PD7 | PD6 | PD5 | PD4 | PD3 | PD2 | PD1 | PD0 |
|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 |

- ▶ Valor en hexadecimal: 0x6B

```
1 unsigned char puerto;
2
3 puerto = leerPuerto();
4 /* Modificar el valor de un bit */
5 escribirPuerto(puerto);
```

Qué operación utilizar para:

- ▶ Poner a 1 un único bit sin alterar el resto
- ▶ Poner a 0 un único bit sin alterar el resto
- ▶ Invertir el valor de un único bit sin alterar el resto

Operadores a nivel de bits

Pone a 1 un bit:

```
1 unsigned char puerto = leerPuerto();  
2 puerto |= 0x10; // Pone a 1 PD4  
3 escribirPuerto(puerto);
```

Operadores a nivel de bits

Pone a 1 un bit:

```
1 unsigned char puerto = leerPuerto();  
2 puerto |= 0x10; // Pone a 1 PD4  
3 escribirPuerto(puerto);
```

0x6B = 0 1 1 0 1 0 1 1
0x10 = 0 0 0 1 0 0 0 0
----- (OR)
0 1 1 1 1 0 1 1

Operadores a nivel de bits

Pone a 1 un bit:

```
1 unsigned char puerto = leerPuerto();  
2 puerto |= 0x10; // Pone a 1 PD4  
3 escribirPuerto(puerto);
```

0x6B = 0 1 1 0 1 0 1 1
0x10 = 0 0 0 1 0 0 0 0
----- (OR)
0 1 1 1 1 0 1 1

Pone a 0 un bit:

```
1 unsigned char puerto = leerPuerto();  
2 puerto &= ~(0x10); // Pone a 1 PD5  
3 escribirPuerto(puerto);
```

Operadores a nivel de bits

Pone a 1 un bit:

```
1 unsigned char puerto = leerPuerto();
2 puerto |= 0x10; // Pone a 1 PD4
3 escribirPuerto(puerto);
```

0x6B = 0 1 1 0 1 0 1 1
0x10 = 0 0 0 1 0 0 0 0
----- (OR)
0 1 1 1 1 0 1 1

Pone a 0 un bit:

```
1 unsigned char puerto = leerPuerto();
2 puerto &= ~(0x10); // Pone a 1 PD5
3 escribirPuerto(puerto);
```

0x6B = 0 1 1 0 1 0 1 1
~(0x10) = 1 1 1 0 1 1 1 1
----- (AND)
0 1 1 0 1 0 1 1

Operadores a nivel de bits

Pone a 1 un bit:

```
1 unsigned char puerto = leerPuerto();
2 puerto |= 0x10; // Pone a 1 PD4
3 escribirPuerto(puerto);
```

0x6B = 0 1 1 0 1 0 1 1
0x10 = 0 0 0 1 0 0 0 0
----- (OR)
0 1 1 1 1 0 1 1

Pone a 0 un bit:

```
1 unsigned char puerto = leerPuerto();
2 puerto &= ~(0x10); // Pone a 1 PD5
3 escribirPuerto(puerto);
```

0x6B = 0 1 1 0 1 0 1 1
~(0x10) = 1 1 1 0 1 1 1 1
----- (AND)
0 1 1 0 1 0 1 1

Invierte el valor de un bit:

```
1 unsigned char puerto = leerPuerto();
2 puerto ^= 0x02; // Invierte PD1
3 escribirPuerto(puerto);
```

Operadores a nivel de bits

Pone a 1 un bit:

```
1 unsigned char puerto = leerPuerto();
2 puerto |= 0x10; // Pone a 1 PD4
3 escribirPuerto(puerto);
```

0x6B = 0 1 1 0 1 0 1 1
0x10 = 0 0 0 1 0 0 0 0
----- (OR)
0 1 1 1 1 0 1 1

Pone a 0 un bit:

```
1 unsigned char puerto = leerPuerto();
2 puerto &= ~(0x10); // Pone a 1 PD5
3 escribirPuerto(puerto);
```

0x6B = 0 1 1 0 1 0 1 1
~(0x10) = 1 1 1 0 1 1 1 1
----- (AND)
0 1 1 0 1 0 1 1

Invierte el valor de un bit:

```
1 unsigned char puerto = leerPuerto();
2 puerto ^= 0x02; // Invierte PD1
3 escribirPuerto(puerto);
```

0x6B = 0 1 1 0 1 0 1 1
0x10 = 0 0 0 0 0 0 1 0
----- (exOR)
0 1 1 0 1 0 0 1

Operadores de desplazamiento

Desplazamiento a la izquierda

- ▶ Los valores desplazados se pierden
- ▶ Los valores a la derecha se rellenan con ceros

Operadores de desplazamiento

Desplazamiento a la izquierda

- ▶ Los valores desplazados se pierden
- ▶ Los valores a la derecha se llenan con ceros

Desplazamiento a la derecha

- ▶ Los valores desplazados se pierdes
- ▶ Los valores a la izquierda dependen del tipo de dato (`signed/unsigned`)

Imprimir variables en binario

```
1 void imprimir_binario(unsigned int val)
2 {
3
4
5
6
7
8
9
10
11
12
13
14
15 }
```

Imprimir variables en binario

```
1 void imprimir_binario(unsigned int val)
2 {
3     unsigned int b, mask = 1 << (8*sizeof(unsigned int)-1);
4
5
6
7
8
9
10
11
12
13
14
15 }
```

Imprimir variables en binario

```
1 void imprimir_binario(unsigned int val)
2 {
3     unsigned int b, mask = 1 << (8*sizeof(unsigned int)-1);
4
5
6
7
8
9
10
11
12
13
14
15 }
```

- ▶ ¿Cuanto es `sizeof(unsigned int)`?

Imprimir variables en binario

```
1 void imprimir_binario(unsigned int val)
2 {
3     unsigned int b, mask = 1 << (8*sizeof(unsigned int)-1);
4
5
6
7
8
9
10
11
12
13
14
15 }
```

- ▶ ¿Cuanto es `sizeof(unsigned int)`? 4

Imprimir variables en binario

```
1 void imprimir_binario(unsigned int val)
2 {
3     unsigned int b, mask = 1 << (8*sizeof(unsigned int)-1);
4
5
6
7
8
9
10
11
12
13
14
15 }
```

- ▶ ¿Cuanto es `sizeof(unsigned int)`? 4

```
unsigned int mask = 1 << 31; // 0x80000000
```

Imprimir variables en binario

```
1 void imprimir_binario(unsigned int val)
2 {
3     unsigned int b, mask = 1 << (8*sizeof(unsigned int)-1);
4
5     for(b = 1; b <= 8*sizeof(unsigned int); b++)
6     {
7
8
9
10
11
12 }
13
14
15 }
```

- ▶ ¿Cuanto es `sizeof(unsigned int)`? 4

```
unsigned int mask = 1 << 31; // 0x80000000
```

Imprimir variables en binario

```
1 void imprimir_binario(unsigned int val)
2 {
3     unsigned int b, mask = 1 << (8*sizeof(unsigned int)-1);
4
5     for(b = 1; b <= 8*sizeof(unsigned int); b++)
6     {
7         putchar(val & mask ? '1' : '0');
8         val <= 1;
9
10
11    }
12
13
14
15 }
```

- ▶ ¿Cuanto es `sizeof(unsigned int)`? 4

```
unsigned int mask = 1 << 31; // 0x80000000
```

Imprimir variables en binario

```
1 void imprimir_binario(unsigned int val)
2 {
3     unsigned int b, mask = 1 << (8*sizeof(unsigned int)-1);
4
5     for(b = 1; b <= 8*sizeof(unsigned int); b++)
6     {
7         putchar(val & mask ? '1' : '0');
8         val <= 1;
9
10        if(b % 8 == 0)
11            putchar(' ');
12    }
13
14    putchar('\n');
15 }
```

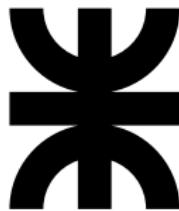
- ▶ ¿Cuanto es `sizeof(unsigned int)`? 4

```
unsigned int mask = 1 << 31; // 0x80000000
```


Informática II

El compilador de C del proyecto GNU (`gcc`, `g++`)

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2021 –

El compilador de C del proyecto GNU

Herramientas de compilación del proyecto GNU ([GNU Compiler Collection](#)):

El compilador de C del proyecto GNU

Herramientas de compilación del proyecto GNU ([GNU Compiler Collection](#)):

- ▶ Maneja varios dialectos de C: ANSI-C, tradicional (Kernighan & Ritchie), extensiones GNU, etc.

El compilador de C del proyecto GNU

Herramientas de compilación del proyecto GNU ([GNU Compiler Collection](#)):

- ▶ Maneja varios dialectos de C: ANSI-C, tradicional (Kernighan & Ritchie), extensiones GNU, etc.
- ▶ Puede compilar C++

El compilador de C del proyecto GNU

Herramientas de compilación del proyecto GNU ([GNU Compiler Collection](#)):

- ▶ Maneja varios dialectos de C: ANSI-C, tradicional (Kernighan & Ritchie), extensiones GNU, etc.
- ▶ Puede compilar C++
- ▶ Realiza la optimización del código

El compilador de C del proyecto GNU

Herramientas de compilación del proyecto GNU ([GNU Compiler Collection](#)):

- ▶ Maneja varios dialectos de C: ANSI-C, tradicional (Kernighan & Ritchie), extensiones GNU, etc.
- ▶ Puede compilar C++
- ▶ Realiza la optimización del código
- ▶ Genera información de depuración (debugging)

El compilador de C del proyecto GNU

Herramientas de compilación del proyecto GNU ([GNU Compiler Collection](#)):

- ▶ Maneja varios dialectos de C: ANSI-C, tradicional (Kernighan & Ritchie), extensiones GNU, etc.
- ▶ Puede compilar C++
- ▶ Realiza la optimización del código
- ▶ Genera información de depuración (debugging)
- ▶ Es un compilador cruzado (cross-compiler)
`$ gcc --version`
`$ gcc -v`
(`--build`, `--host`, `--target`)

Compilando – Algunos ejemplos

hola.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hola, mundo!\n");
6     return 0;
7 }
```

Compilando – Algunos ejemplos

hola.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hola, mundo!\n");
6     return 0;
7 }
```

Compilación

```
> gcc hola.c
```

Compilando – Algunos ejemplos

hola.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hola, mundo!\n");
6     return 0;
7 }
```

Compilación

```
> gcc hola.c
```

(salida a.out)

Compilando – Algunos ejemplos

hola.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hola, mundo!\n");
6     return 0;
7 }
```

Compilación

```
> gcc hola.c
```

(salida a.out) Cómo se ejecuta?

Compilando – Algunos ejemplos

hola.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hola, mundo!\n");
6     return 0;
7 }
```

Compilación

```
> gcc hola.c
```

(salida a.out) Cómo se ejecuta?

Cómo cambiar el nombre a binario?

Compilando – Algunos ejemplos

hola.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hola, mundo!\n");
6     return 0;
7 }
```

Compilación

```
> gcc hola.c
```

(salida a.out) Cómo se ejecuta?

Cómo cambiar el nombre a binario?

```
> gcc hola.c -o hola
```

Compilando – Algunos ejemplos

hola.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hola, mundo!\n");
6     return 0;
7 }
```

Compilación

```
> gcc hola.c
```

(salida a.out) Cómo se ejecuta?

Cómo cambiar el nombre a binario?

```
> gcc hola.c -o hola
```

```
> gcc -Wall hola.c -o hola
```

Compilando – Algunos ejemplos

hola.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hola, mundo!\n");
6     return 0;
7 }
```

Compilación

```
> gcc hola.c
```

(salida a.out) Cómo se ejecuta?

Cómo cambiar el nombre a binario?

```
> gcc hola.c -o hola
```

```
> gcc -Wall hola.c -o hola
```

(habilita todas las advertencias)

Compilando – Algunos ejemplos

hola.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hola, mundo!\n");
6     return 0;
7 }
```

mal.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Dos y dos son %f\n", 4);
6     return 0;
7 }
```

Compilación

```
> gcc hola.c
```

(salida a.out) Cómo se ejecuta?

Cómo cambiar el nombre a binario?

```
> gcc hola.c -o hola
```

```
> gcc -Wall hola.c -o hola
```

(habilita todas las advertencias)

Compilando – Algunos ejemplos

hola.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hola, mundo!\n");
6     return 0;
7 }
```

mal.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Dos y dos son %f\n", 4);
6     return 0;
7 }
```

Compilación

```
> gcc hola.c
```

(salida a.out) Cómo se ejecuta?

Cómo cambiar el nombre a binario?

```
> gcc hola.c -o hola
```

```
> gcc -Wall hola.c -o hola
```

(habilita todas las advertencias)

Compilar y ejecutar

```
> gcc mal.c -o mal
> ./mal
Dos y dos son 0.000000
```

Compilando – Algunos ejemplos

hola.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hola, mundo!\n");
6     return 0;
7 }
```

mal.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Dos y dos son %f\n", 4);
6     return 0;
7 }
```

Compilación

```
> gcc hola.c
```

(salida a.out) Cómo se ejecuta?

Cómo cambiar el nombre a binario?

```
> gcc hola.c -o hola
```

```
> gcc -Wall hola.c -o hola
```

(habilita todas las advertencias)

Compilar y ejecutar

```
> gcc mal.c -o mal
> ./mal
Dos y dos son 0.000000
```

Error de compilación (-Wall)

```
mal.c:5:10: warning: format '%f'
expects argument of type 'double'
but argument 2 has type 'int'
[-Wformat=]
```

Compilando – Ejemplos con varios archivos fuentes

main.c

```
1 #include "hola.h"
2
3 int main(void)
4 {
5     hola("mundo");
6     return 0;
7 }
```

Compilando – Ejemplos con varios archivos fuentes

hola.c

main.c

```
1 #include "hola.h"
2
3 int main(void)
4 {
5     hola("mundo");
6     return 0;
7 }
```

```
1 #include <stdio.h>
2 #include "hola.h"
3
4 void hola(const char * nombre)
5 {
6     printf("Hola, %s!\n", nombre);
7 }
```

Compilando – Ejemplos con varios archivos fuentes

hola.c

main.c

```
1 #include "hola.h"
2
3 int main(void)
4 {
5     hola("mundo");
6     return 0;
7 }
```

```
1 #include <stdio.h>
2 #include "hola.h"
3
4 void hola(const char * nombre)
5 {
6     printf("Hola, %s!\n", nombre);
7 }
```

hola.h

```
1 void hola(const char * nombre);
```

Compilando – Ejemplos con varios archivos fuentes

hola.c

main.c

```
1 #include "hola.h"
2
3 int main(void)
4 {
5     hola("mundo");
6     return 0;
7 }
```

```
1 #include <stdio.h>
2 #include "hola.h"
3
4 void hola(const char * nombre)
5 {
6     printf("Hola, %s!\n", nombre);
7 }
```

hola.h

```
1 void hola(const char * nombre);
```

Compilación

```
> gcc -Wall main.c hola.c -o hola
```

Compilando – Ejemplos con varios archivos fuentes

hola.c

main.c

```
1 #include "hola.h"
2
3 int main(void)
4 {
5     hola("mundo");
6     return 0;
7 }
```

```
1 #include <stdio.h>
2 #include "hola.h"
3
4 void hola(const char * nombre)
5 {
6     printf("Hola, %s!\n", nombre);
7 }
```

hola.h

```
1 void hola(const char * nombre);
```

Compilación

```
> gcc -Wall main.c hola.c -o hola
```

Se puede compilar separadamente cada archivo fuente

```
> gcc -Wall -c main.c
> gcc -Wall -c hola.c
```

Compilando – Ejemplos con varios archivos fuentes

hola.c

main.c

```
1 #include "hola.h"
2
3 int main(void)
4 {
5     hola("mundo");
6     return 0;
7 }
```

```
1 #include <stdio.h>
2 #include "hola.h"
3
4 void hola(const char * nombre)
5 {
6     printf("Hola, %s!\n", nombre);
7 }
```

hola.h

```
1 void hola(const char * nombre);
```

Compilación

```
> gcc -Wall main.c hola.c -o hola
```

Se puede compilar separadamente cada archivo fuente

```
> gcc -Wall -c main.c
> gcc -Wall -c hola.c
```

y unirlo con el linker (enlazador)

```
> gcc main.o hola.o -o hola
```

Compilando – Ejemplos con varios archivos fuentes

hola.c

main.c

```
1 #include "hola.h"
2
3 int main(void)
4 {
5     hola("mundo");
6     return 0;
7 }
```

```
1 #include <stdio.h>
2 #include "hola.h"
3
4 void hola(const char * nombre)
5 {
6     printf("Hola, %s!\n", nombre);
7 }
```

hola.h

```
1 void hola(const char * nombre);
```

Compilación

```
> gcc -Wall main.c hola.c -o hola
```

Se puede compilar separadamente cada archivo fuente

```
> gcc -Wall -c main.c
> gcc -Wall -c hola.c
```

y unirlo con el linker (enlazador)

```
> gcc main.o hola.o -o hola
```

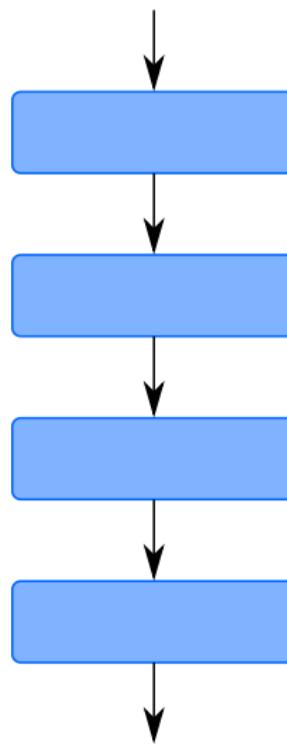
Esto permite modificar un archivo fuente y recompilar solo ese archivo.

Etapas de compilación/construcción

El proceso de compilación/construcción involucra **4** etapas (preprocesamiento, compilación, ensamblado, y enlazado).

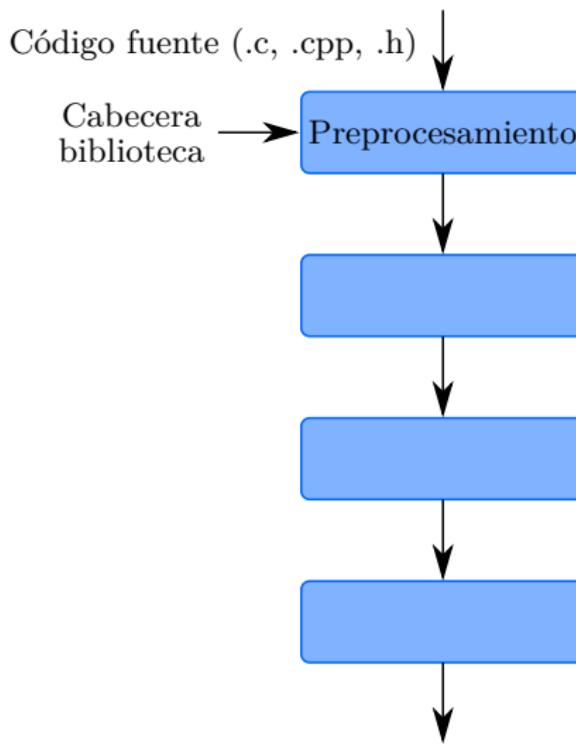
Etapas de compilación/construcción

El proceso de compilación/construcción involucra **4** etapas (preprocesamiento, compilación, ensamblado, y enlazado).



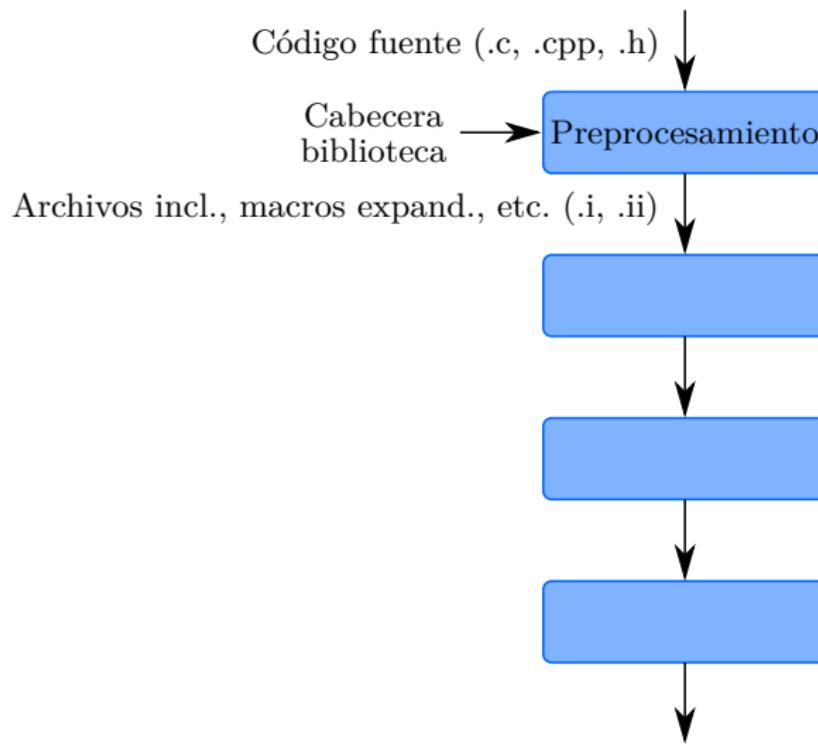
Etapas de compilación/construcción

El proceso de compilación/construcción involucra **4** etapas (preprocesamiento, compilación, ensamblado, y enlazado).



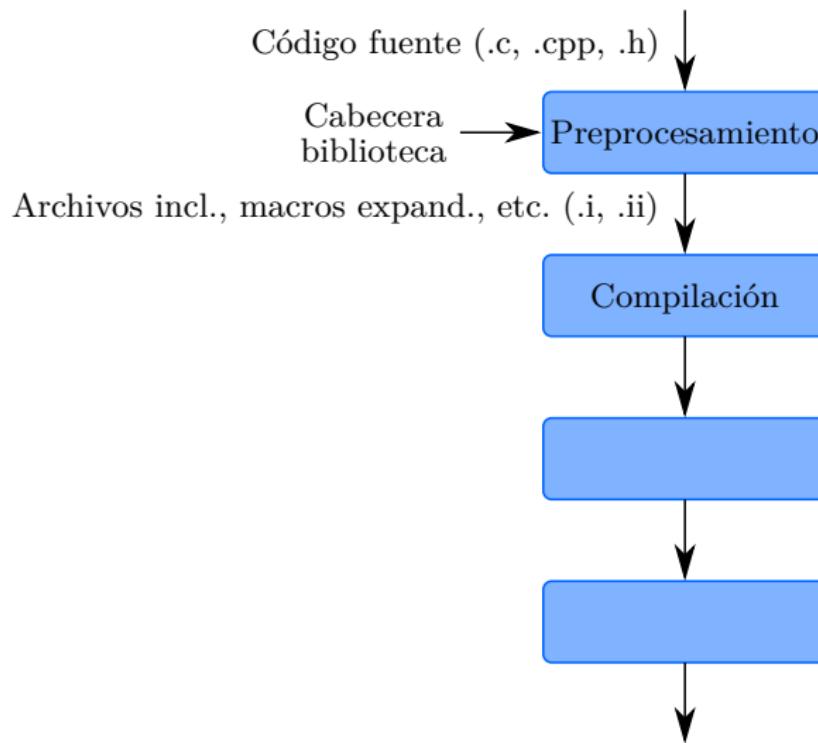
Etapas de compilación/construcción

El proceso de compilación/construcción involucra **4** etapas (preprocesamiento, compilación, ensamblado, y enlazado).



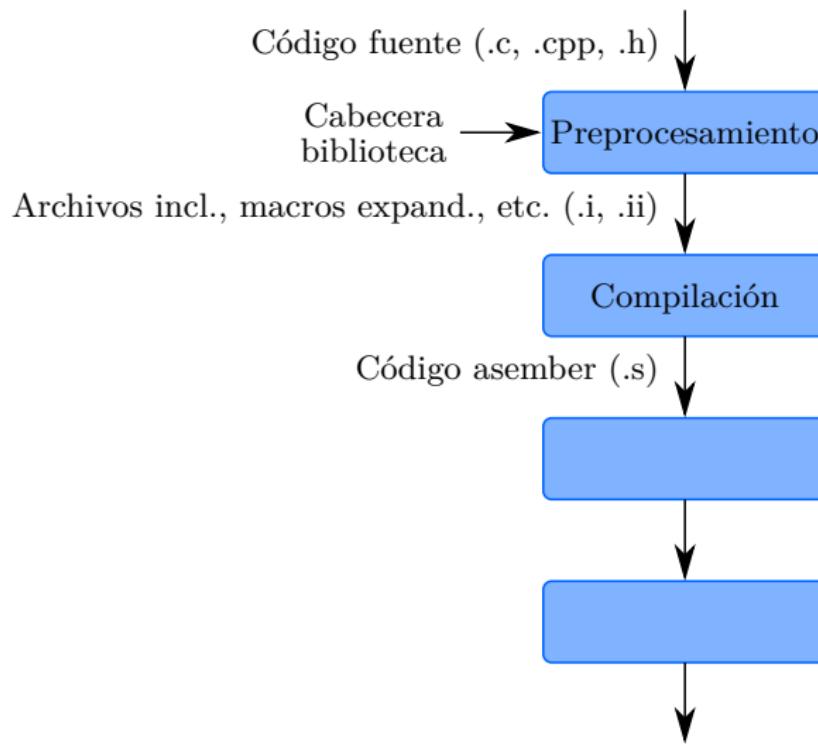
Etapas de compilación/construcción

El proceso de compilación/construcción involucra **4** etapas (preprocesamiento, compilación, ensamblado, y enlazado).



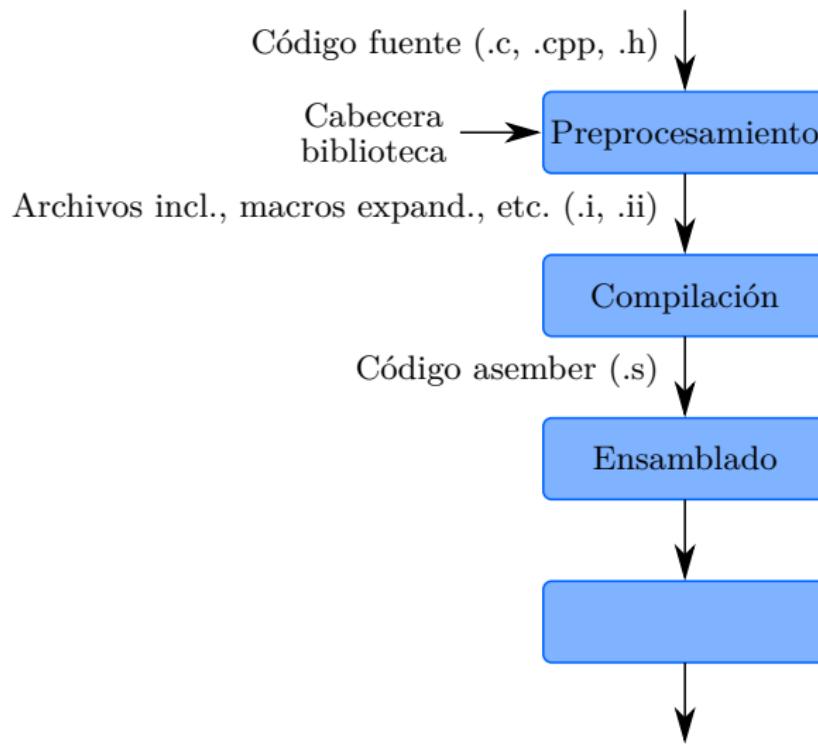
Etapas de compilación/construcción

El proceso de compilación/construcción involucra **4** etapas (preprocesamiento, compilación, ensamblado, y enlazado).



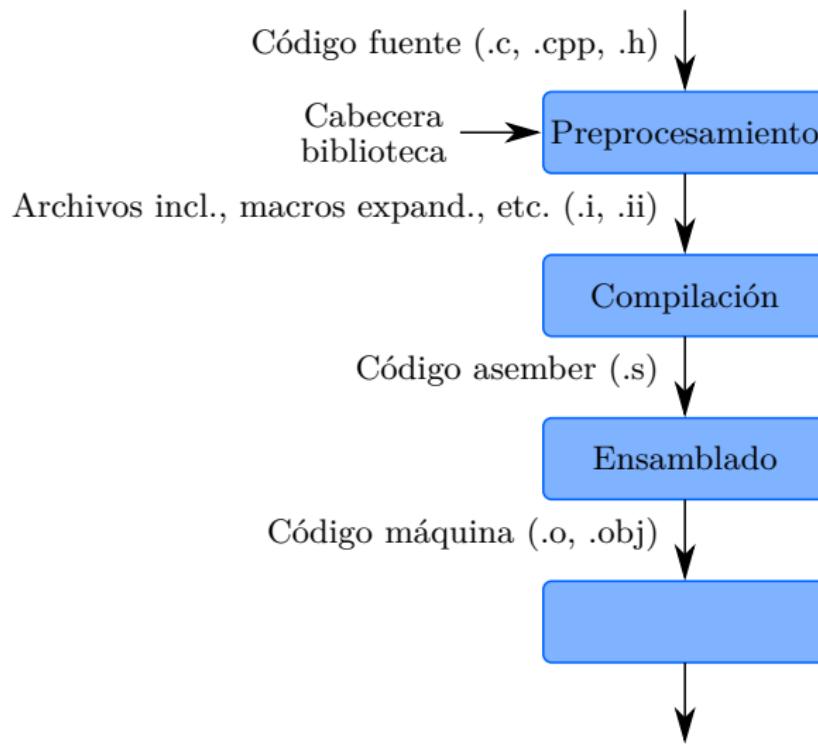
Etapas de compilación/construcción

El proceso de compilación/construcción involucra **4** etapas (preprocesamiento, compilación, ensamblado, y enlazado).



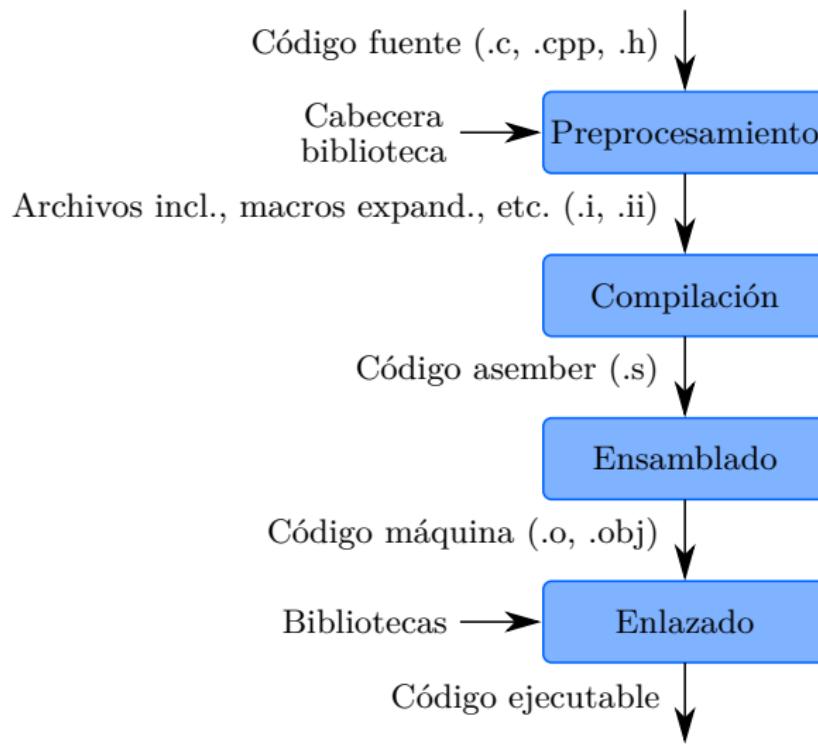
Etapas de compilación/construcción

El proceso de compilación/construcción involucra **4** etapas (preprocesamiento, compilación, ensamblado, y enlazado).



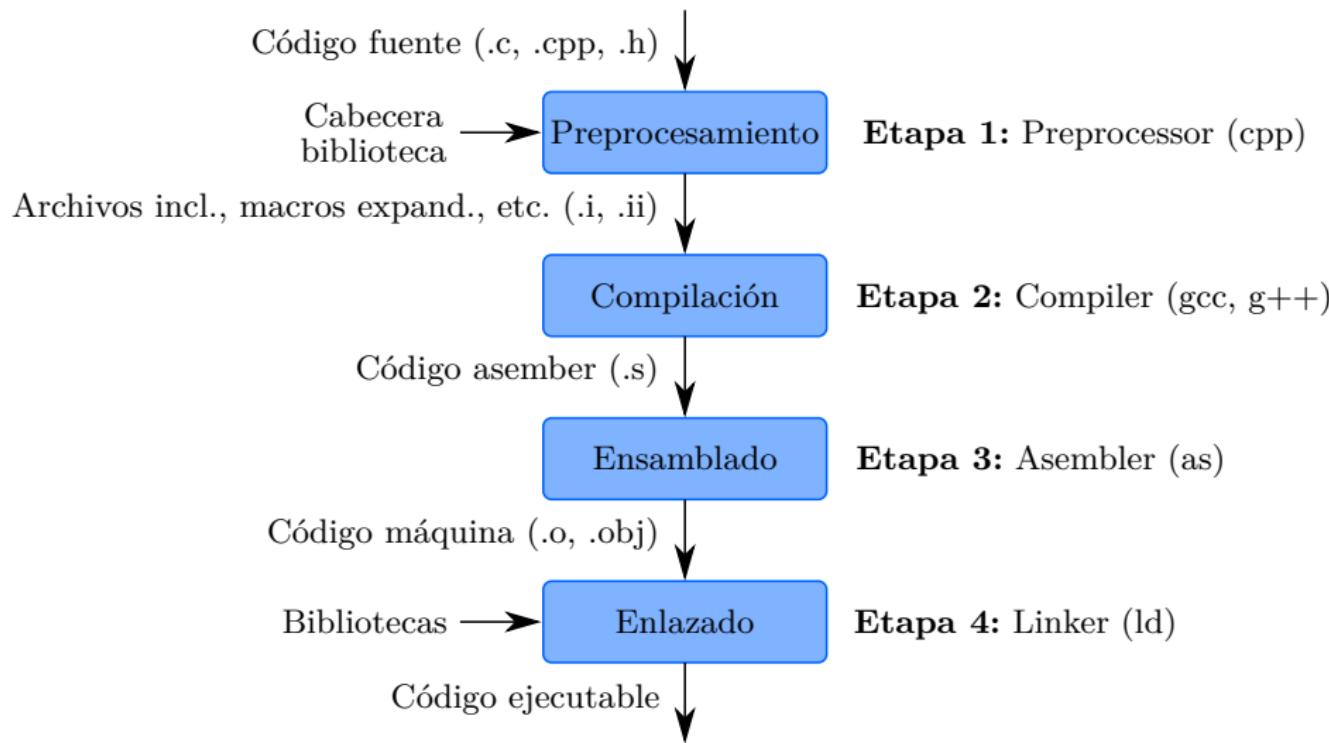
Etapas de compilación/construcción

El proceso de compilación/construcción involucra **4 etapas** (preprocesamiento, compilación, ensamblado, y enlazado).



Etapas de compilación/construcción

El proceso de compilación/construcción involucra **4 etapas** (preprocesamiento, compilación, ensamblado, y enlazado). Conjunto de herramientas: *toolchain*.



Preprocesado

test.c

```
1 #define TEST "Hola mundo!"  
2 const char str[] = TEST;
```

Preprocesado

test.c

```
1 #define TEST "Hola mundo!"  
2 const char str[] = TEST;
```

```
> gcc -E test.c
```

(O bien: `cpp test.c`)

Preprocesado

test.c

```
1 #define TEST "Hola mundo!"  
2 const char str[] = TEST;
```

```
> gcc -E test.c
```

(O bien: `cpp test.c`)

Salida:

```
# 1 "test.c"  
# 1 "<built-in>"  
# 1 "<command-line>"  
# 1 "/usr/include/stdc-predef.h" 1 3 4  
# 1 "<command-line>" 2  
# 1 "test.c"  
  
const char str[] = "Hola, Mundo!";
```

Preprocesado

test.c

```
1 #define TEST "Hola mundo!"  
2 const char str[] = TEST;
```

```
> gcc -E test.c
```

(O bien: `cpp test.c`)

Salida:

```
# 1 "test.c"  
# 1 "<built-in>"  
# 1 "<command-line>"  
# 1 "/usr/include/stdc-predef.h" 1 3 4  
# 1 "<command-line>" 2  
# 1 "test.c"  
  
const char str[] = "Hola, Mundo!";
```

(el preprocesador inserta líneas de registro de archivo fuente y el nro. de línea en la forma `#num-de-línea "archivo fuente"`)

Preprocesado

hola1.c

```
1 #include <stdio.h>
2
3 /* Función main */
4 int main(void)
5 {
6     /* Uso de cadena literal */
7     printf("Hola mundo!\n");
8     return 0;
9 }
```

hola2.c

```
1 #include <stdio.h>
2 #define MENSAJE "Hola mundo!\n"
3
4 /* Función main */
5 int main(void)
6 {
7     /* Uso de constante simbólica */
8     printf(MENSAJE);
9     return 0;
10 }
```

Preprocesado

hola1.c

```
1 #include <stdio.h>
2
3 /* Función main */
4 int main(void)
5 {
6     /* Uso de cadena literal */
7     printf("Hola mundo!\n");
8     return 0;
9 }
```

hola2.c

```
1 #include <stdio.h>
2 #define MENSAJE "Hola mundo!\n"
3
4 /* Función main */
5 int main(void)
6 {
7     /* Uso de constante simbólica */
8     printf(MENSAJE);
9     return 0;
10 }
```

Preprocesado:

```
> gcc -E hola1.c > hola1.i
```

Preprocesado:

```
> gcc -E hola2.c > hola2.i
```

Preprocesado

hola1.c

```
1 #include <stdio.h>
2
3 /* Función main */
4 int main(void)
5 {
6     /* Uso de cadena literal */
7     printf("Hola mundo!\n");
8     return 0;
9 }
```

hola2.c

```
1 #include <stdio.h>
2 #define MENSAJE "Hola mundo!\n"
3
4 /* Función main */
5 int main(void)
6 {
7     /* Uso de constante simbólica */
8     printf(MENSAJE);
9     return 0;
10 }
```

Preprocesado:

```
> gcc -E hola1.c > hola1.i
```

Comparar los archivos de salida.

Preprocesado:

```
> gcc -E hola2.c > hola2.i
```

Preprocesado

hola1.c

```
1 #include <stdio.h>
2
3 /* Función main */
4 int main(void)
5 {
6     /* Uso de cadena literal */
7     printf("Hola mundo!\n");
8     return 0;
9 }
```

hola2.c

```
1 #include <stdio.h>
2 #define MENSAJE "Hola mundo!\n"
3
4 /* Función main */
5 int main(void)
6 {
7     /* Uso de constante simbólica */
8     printf(MENSAJE);
9     return 0;
10 }
```

Preprocesado:

```
> gcc -E hola1.c > hola1.i
```

Preprocesado:

```
> gcc -E hola2.c > hola2.i
```

Comparar los archivos de salida.

Observar: constantes simbólicas, comentarios, archivos cabecera (includes).

Compilado, ensamblado y enlazado

Compilar el archivo de salida del preprocesador:

```
> gcc -Wall -S hola.i
```

[entrada: **hola.i** – salida: **hola.s**]

Compilado, ensamblado y enlazado

Compilar el archivo de salida del preprocesador:

```
> gcc -Wall -S hola.i
```

[entrada: **hola.i** – salida: **hola.s**]

Ensamblado:

```
> gcc -c hola.s -o hola.o
```

[entrada: **hola.s** – salida: **hola.o**]

(O bien: **as hola.s -o hola.o**)

Compilado, ensamblado y enlazado

Compilar el archivo de salida del preprocesador:

```
> gcc -Wall -S hola.i
```

[entrada: **hola.i** – salida: **hola.s**]

Ensamblado:

```
> gcc -c hola.s -o hola.o
```

[entrada: **hola.s** – salida: **hola.o**]

(O bien: **as hola.s -o hola.o**)

Enlazado:

```
>gcc hola.o -o hola
```

(Se puede utilizar también **ld**)

Construcción paso a paso – Resumen

Ejecutar diferentes etapas de construcción:

- ▶ `gcc -E`: Preprocesamiento sin compilación
- ▶ `gcc -S`: Compilación sin ensamblado
- ▶ `gcc -c`: Preprocesamiento, compilación y ensamblado sin enlazado

Construcción paso a paso – Resumen

Ejecutar diferentes etapas de construcción:

- ▶ `gcc -E`: Preprocesamiento sin compilación
- ▶ `gcc -S`: Compilación sin ensamblado
- ▶ `gcc -c`: Preprocesamiento, compilación y ensamblado sin enlazado

Ver página de manual de `gcc` (> `man gcc`)

Construcción paso a paso – Resumen

Ejecutar diferentes etapas de construcción:

- ▶ `gcc -E`: Preprocesamiento sin compilación
- ▶ `gcc -S`: Compilación sin ensamblado
- ▶ `gcc -c`: Preprocesamiento, compilación y ensamblado sin enlazado

Ver página de manual de `gcc` (`> man gcc`)

Flag `--save-temp`s genera archivos intermedios

```
> gcc --save-temp hola.c -o hola
```

(Ver flag `-v`)

Construcción paso a paso – Resumen

Ejecutar diferentes etapas de construcción:

- ▶ `gcc -E`: Preprocesamiento sin compilación
- ▶ `gcc -S`: Compilación sin ensamblado
- ▶ `gcc -c`: Preprocesamiento, compilación y ensamblado sin enlazado

Ver página de manual de `gcc` (`> man gcc`)

Flag `--save-temp`s genera archivos intermedios

```
> gcc --save-temp hola.c -o hola
```

(Ver flag `-v`)

Otros flags: `-std=c90`, `-Wall`, `-Werror`

Construcción paso a paso – Archivos de salida

```
> file hola.s  
hola.s: assembler source, ASCII text
```

[file hola.c, file hola.i (?)]

Construcción paso a paso – Archivos de salida

```
> file hola.s  
hola.s: assembler source, ASCII text
```

[file hola.c, file hola.i (?)]

```
> file hola.o  
hola.o: ELF 64-bit LSB relocatable, x86-64,  
version 1 (SYSV), not stripped
```

Construcción paso a paso – Archivos de salida

```
> file hola.s
hola.s: assembler source, ASCII text
```

[file hola.c, file hola.i (?)]

```
> file hola.o
hola.o: ELF 64-bit LSB relocatable, x86-64,
version 1 (SYSV), not stripped
```

```
> file hola
hola ELF 64-bit LSB executable, x86-64, version 1
(SYSV), dynamically linked, interpreter /lib64/ld-
linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]
=620d3c9fadabd53755c0a647c0a43a172481a6f8, not stripped
```

Construcción paso a paso – Archivos de salida

```
> file hola.s
hola.s: assembler source, ASCII text
```

[file hola.c, file hola.i (?)]

```
> file hola.o
hola.o: ELF 64-bit LSB relocatable, x86-64,
version 1 (SYSV), not stripped
```

```
> file hola
hola ELF 64-bit LSB executable, x86-64, version 1
(SYSV), dynamically linked, interpreter /lib64/ld-
linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]
=620d3c9fadabd53755c0a647c0a43a172481a6f8, not stripped
```

```
> ldd hola
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
```

Construcción paso a paso – Archivos de salida

```
> file hola.s
hola.s: assembler source, ASCII text
```

[file hola.c, file hola.i (?)]

```
> file hola.o
hola.o: ELF 64-bit LSB relocatable, x86-64,
version 1 (SYSV), not stripped
```

```
> file hola
hola ELF 64-bit LSB executable, x86-64, version 1
(SYSV), dynamically linked, interpreter /lib64/ld-
linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]
=620d3c9fadabd53755c0a647c0a43a172481a6f8, not stripped
```

```
> ldd hola
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
```

ELF: Executable and Linkable Format

Construcción paso a paso – Archivos de salida

```
> file hola.s  
hola.s: assembler source, ASCII text
```

[file hola.c, file hola.i (?)]

```
> file hola.o  
hola.o: ELF 64-bit LSB relocatable, x86-64,  
version 1 (SYSV), not stripped
```

```
> file hola  
hola ELF 64-bit LSB executable, x86-64, version 1  
(SYSV), dynamically linked, interpreter /lib64/ld-  
linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]  
=620d3c9fadabd53755c0a647c0a43a172481a6f8, not stripped
```

```
> ldd hola  
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
```

ELF: Executable and Linkable Format

Otros comandos a probar: `hexdump -C`, `objdump -x`, `readelf -d`

El preprocesador

El preprocesamiento ocurre antes de la compilación de un programa.

El preprocesador

El preprocesamiento ocurre antes de la compilación de un programa.

Algunas de las acciones son:

1. inclusión de otros archivos dentro del archivo a compilar
2. definición de constantes simbólicas y macros
3. compilación condicional del código del programa

El preprocesador – Inclusión

La directiva del preprocesador `#include` provoca la inclusión de una copia del archivo especificado en el lugar de la directiva.

El preprocesador – Inclusión

La directiva del preprocesador `#include` provoca la inclusión de una copia del archivo especificado en el lugar de la directiva.

Tiene dos formas:

```
1 #include <nombre de archivo>
2 #include "nombre de archivo"
```

El preprocesador – Inclusión

La directiva del preprocesador `#include` provoca la inclusión de una copia del archivo especificado en el lugar de la directiva.

Tiene dos formas:

```
1 #include <nombre de archivo>
2 #include "nombre de archivo"
```

Difieren en la ubicación en la que el preprocesador busca el archivo a incluir:

1. Nombre del archivo entre llaves angulares (< y >) se utiliza por los *encabezados de la biblioteca estándar*.
2. Nombre del archivo entre comillas: busca el archivo en el mismo directorio en donde se encuentra el archivo que va a compilarse.

El preprocesador – Inclusión

La directiva del preprocesador `#include` provoca la inclusión de una copia del archivo especificado en el lugar de la directiva.

Tiene dos formas:

```
1 #include <nombre de archivo>
2 #include "nombre de archivo"
```

Difieren en la ubicación en la que el preprocesador busca el archivo a incluir:

1. Nombre del archivo entre llaves angulares (< y >) se utiliza por los *encabezados de la biblioteca estándar*.
2. Nombre del archivo entre comillas: busca el archivo en el mismo directorio en donde se encuentra el archivo que va a compilarse.

Las declaraciones que se incluyen en archivos de cabecera son de estructuras y uniones, enumeraciones y prototipos de funciones.

El preprocesador – Contantes simbólicas

La directiva `#define` crea **constantes simbólicas** (constantes representadas por símbolos) y **macros** (operaciones definidas como símbolos). El formato es:

```
#define identificador texto de reemplazo
```

El preprocesador – Contantes simbólicas

La directiva `#define` crea **constantes simbólicas** (constantes representadas por símbolos) y **macros** (operaciones definidas como símbolos). El formato es:

```
#define identificador texto de reemplazo
```

Hace que las ocurrencias subsecuentes del identificador sean reempazadas con el **texto de reemplazo**. Por ejemplo:

```
#define PI 3.14159
```

reemplaza todas las ocurrencias de la constante simbólica PI con la constante numérica 3.14159.

El preprocesador – Macros

- ▶ Una **macro** es un identificador definido dentro de una directiva de preprocesador **#define**.
- ▶ Como en las constantes simbólicas, el identificador de la macro se reemplaza en el programa con el **texto de reemplazo**.

El preprocesador – Macros

- ▶ Una **macro** es un identificador definido dentro de una directiva de preprocesador **#define**.
- ▶ Como en las constantes simbólicas, el identificador de la macro se reemplaza en el programa con el **texto de reemplazo**.

Las macros se puede definir con o sin argumentos.

- ▶ Una macro sin argumentos se procesa como una constante simbólica.
- ▶ En una **macro** con argumentos, los argumentos se sustituyen dentro del texto de reemplazo, y después se desarrolla la macro.

El preprocesador – Macros

- ▶ Una **macro** es un identificador definido dentro de una directiva de preprocesador **#define**.
- ▶ Como en las constantes simbólicas, el identificador de la macro se reemplaza en el programa con el **texto de reemplazo**.

Las macros se puede definir con o sin argumentos.

- ▶ Una macro sin argumentos se procesa como una constante simbólica.
- ▶ En una **macro** con argumentos, los argumentos se sustituyen dentro del texto de reemplazo, y después se desarrolla la macro.

Por ejemplo:

```
#define AREA_CIRCULO( x ) ( (PI) * (x) * (x) )
```

El preprocesador – Macros

- ▶ Una **macro** es un identificador definido dentro de una directiva de preprocesador **#define**.
- ▶ Como en las constantes simbólicas, el identificador de la macro se reemplaza en el programa con el **texto de reemplazo**.

Las macros se puede definir con o sin argumentos.

- ▶ Una macro sin argumentos se procesa como una constante simbólica.
- ▶ En una **macro** con argumentos, los argumentos se sustituyen dentro del texto de reemplazo, y después se desarrolla la macro.

Por ejemplo:

```
#define AREA_CIRCULO( x ) ( (PI) * (x) * (x) )
```

Cuando aparezca **AREA_CIRCULO(y)** en el archivo, el valor de **x** se sustituirá por **y** dentro del texto de reemplazo.

El preprocesador – Macros

Por ejemplo, la línea:

```
area = AREA_CIRCULO(4);
```

se desarrolla como:

```
area = ( ( 3.14159 ) * (4) * (4) );
```

y el valor de la expresión se evalúa y se asigna a la variable **area**.

El preprocesador – Macros

Por ejemplo, la línea:

```
area = AREA_CIRCULO(4);
```

se desarrolla como:

```
area = ( ( 3.14159 ) * (4) * (4) );
```

y el valor de la expresión se evalúa y se asigna a la variable `area`.

Los paréntesis alrededor de cada `x` dentro del texto de reemplazo fuerza el orden apropiado de evaluación, cuando el argumento de la macro es una expresión:

```
area = AREA_CIRCULO(c + 2);
```

se desarrolla como:

```
area = AREA_CIRCULO( (3.14159) * (c + 2) * (c + 2) );
```

El preprocesador – Macros

Por ejemplo, la línea:

```
area = AREA_CIRCULO(4);
```

se desarrolla como:

```
area = ( ( 3.14159 ) * (4) * (4) );
```

y el valor de la expresión se evalúa y se asigna a la variable `area`.

Los paréntesis alrededor de cada `x` dentro del texto de reemplazo fuerza el orden apropiado de evaluación, cuando el argumento de la macro es una expresión:

```
area = AREA_CIRCULO(c + 2);
```

se desarrolla como:

```
area = AREA_CIRCULO( (3.14159) * (c + 2) * (c + 2) );
```

Las macros pueden tener más de un argumento. Por ejemplo:

```
#define AREA_RECTANGULO(x, y) ( (x) * (y) )
```

El preprocesador – Macros con valor

dtestval.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("El valor de NUM es %d\n", NUM);
6     return 0;
7 }
```

El preprocesador – Macros con valor

dtestval.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("El valor de NUM es %d\n", NUM);
6     return 0;
7 }
```

```
> gcc -Wall -DNUM=100 dtestval.c
> ./a.out
El valor de NUM es 100
```

(-DNAME=VALUE)

El preprocesador – Macros con valor

dtestval.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("El valor de NUM es %d\n", NUM);
6     return 0;
7 }
```

```
> gcc -Wall -DNUM=100 dtestval.c
> ./a.out
El valor de NUM es 100
```

(-DNAME=VALUE)

```
> gcc -Wall -DNUM="2+2" dtestval.c
> ./a.out
El valor de NUM es 4
```

El preprocesador – Macros con valor

dtestval.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("El valor de NUM es %d\n", NUM);
6     return 0;
7 }
```

```
> gcc -Wall -DNUM=100 dtestval.c
> ./a.out
El valor de NUM es 100
```

(-DNAME=VALUE)

```
> gcc -Wall -DNUM="2+2" dtestval.c
> ./a.out
El valor de NUM es 4
```

(Macros predefinidas: `cpp -dM /dev/null`, p.e.: GNUC)

El preprocesador – Macros con valor

dtestval1.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Diez veces NUM es %d\n", 10 * (NUM));
6     return 0;
7 }
```

El preprocesador – Macros con valor

dtestval1.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Diez veces NUM es %d\n", 10 * (NUM));
6     return 0;
7 }
```

```
> gcc -Wall -DNUM="2+2" dtestval1.c
> ./a.out
El valor de NUM es 40
```

El preprocesador – Macros con valor

dtestval1.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Diez veces NUM es %d\n", 10 * (NUM));
6     return 0;
7 }
```

```
> gcc -Wall -DNUM="2+2" dtestval1.c
> ./a.out
El valor de NUM es 40
```

¿Qué valor se imprime si no se ponen los paréntesis en la macro?

El preprocesador – Macros con valor

dtestval1.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Diez veces NUM es %d\n", 10 * (NUM));
6     return 0;
7 }
```

```
> gcc -Wall -DNUM="2+2" dtestval1.c
> ./a.out
El valor de NUM es 40
```

¿Qué valor se imprime si no se ponen los paréntesis en la macro?

Valor por defecto de una macro

```
> gcc -Wall -DNUM dtestval1.c
> ./a.out
El valor de NUM es 1
```

El preprocesador – Macros con valor

dtestval1.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Diez veces NUM es %d\n", 10 * (NUM));
6     return 0;
7 }
```

```
> gcc -Wall -DNUM="2+2" dtestval1.c
> ./a.out
El valor de NUM es 40
```

¿Qué valor se imprime si no se ponen los paréntesis en la macro?

Valor por defecto de una macro

```
> gcc -Wall -DNUM dtestval1.c
> ./a.out
El valor de NUM es 1
```

Una macro vacía `-DNUM=""` queda definida (`#ifdef`) pero no se expande a nada.

El preprocesador – Compilación condicional

dtest.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     #ifdef TEST
6         printf("Modo test\n");
7     #endif
8     printf("Ejecutando... \n");
9     return 0;
10 }
```

El preprocesador – Compilación condicional

dtest.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     #ifdef TEST
6         printf("Modo test\n");
7     #endif
8     printf("Ejecutando... \n");
9     return 0;
10 }
```

```
> gcc -Wall dtest.c
> ./a.out
Ejecutando...
```

El preprocesador – Compilación condicional

dtest.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     #ifdef TEST
6         printf("Modo test\n");
7     #endif
8     printf("Ejecutando... \n");
9     return 0;
10 }
```

```
> gcc -Wall dtest.c
> ./a.out
Ejecutando...
```

Macro definida desde la línea de comandos

```
> gcc -Wall -DTEST dtest.c
> ./a.out
Modo test
Ejecutando...
```

El preprocesador – Compilación condicional

dtest.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     #ifdef TEST
6         printf("Modo test\n");
7     #endif
8     printf("Ejecutando... \n");
9     return 0;
10 }
```

```
> gcc -Wall dtest.c
> ./a.out
Ejecutando...
```

Macro definida desde la línea de comandos

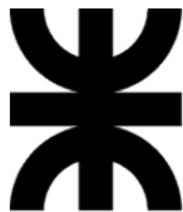
```
> gcc -Wall -DTEST dtest.c
> ./a.out
Modo test
Ejecutando...
```

(probar utilizando `gcc -E`)

Informática II

El compilador de C del proyecto GNU (gcc, g++) – Bibliotecas

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2021 –

Enlazando con bibliotecas externas

Biblioteca

Colección de archivos **objetos** precompilados que pueden ser enlazados dentro de un programas.

Enlazando con bibliotecas externas

Biblioteca

Colección de archivos **objetos** precompilados que pueden ser enlazados dentro de un programas.

Estática: Archivos especiales con la extensión **.a** (ejemplo: **libm.a**)

Dinámica: Archivos especiales con la extensión **.so** (ejemplo: **libm.so**)

Enlazando con bibliotecas externas

Biblioteca

Colección de archivos **objetos** precompilados que pueden ser enlazados dentro de un programas.

Estática: Archivos especiales con la extensión **.a** (ejemplo: **libm.a**)

Dinámica: Archivos especiales con la extensión **.so** (ejemplo: **libm.so**)

- ▶ Se encuentran normalmente en directorios
 - ▶ **/usr/lib** o **/lib**
 - ▶ **/usr/lib64** o **/lib64**
- ▶ o en directorios específicos de la arquitectura
 - ▶ **/usr/lib/i386-linux-gnu/**
 - ▶ **/usr/lib/x86_64-linux-gnu**

Enlazando con bibliotecas externas

Biblioteca

Colección de archivos **objetos** precompilados que pueden ser enlazados dentro de un programas.

Estática: Archivos especiales con la extensión **.a** (ejemplo: **libm.a**)

Dinámica: Archivos especiales con la extensión **.so** (ejemplo: **libm.so**)

- ▶ Se encuentran normalmente en directorios
 - ▶ **/usr/lib** o **/lib**
 - ▶ **/usr/lib64** o **/lib64**
- ▶ o en directorios específicos de la arquitectura
 - ▶ **/usr/lib/i386-linux-gnu/**
 - ▶ **/usr/lib/x86_64-linux-gnu**

Las declaraciones de los **prototipos de funciones** de las funciones de biblioteca se encuentran en archivos de cabecera (archivo con extensión **.h**)

Enlazando con bibliotecas externas – Ejemplo

calc.c

```
1 #include <stdio.h>
2 #include <math.h>
3
4 int main(void)
5 {
6     double x = 2.0;
7     double y = sqrt(x);
8     printf("La raíz cuadrada de %f es %f\n", x, y);
9     return 0;
10 }
```

Enlazando con bibliotecas externas – Ejemplo

calc.c

```
1 #include <stdio.h>
2 #include <math.h>
3
4 int main(void)
5 {
6     double x = 2.0;
7     double y = sqrt(x);
8     printf("La raíz cuadrada de %f es %f\n", x, y);
9     return 0;
10 }
```

```
> gcc -Wall -std=c90 calc.c -o calc
/tmp/cceIc3rZ.o: In function `main':
calc.c:(.text+0x23): undefined reference to `sqrt'
collect2: error: ld returned 1 exit status
```

Enlazando con bibliotecas externas – Ejemplo

calc.c

```
1 #include <stdio.h>
2 #include <math.h>
3
4 int main(void)
5 {
6     double x = 2.0;
7     double y = sqrt(x);
8     printf("La raíz cuadrada de %f es %f\n", x, y);
9     return 0;
10 }
```

```
> gcc -Wall -std=c90 calc.c -o calc
/tmp/cceIc3rZ.o: In function `main':
calc.c:(.text+0x23): undefined reference to `sqrt'
collect2: error: ld returned 1 exit status
```

- ▶ /tmp/cceIc3rZ.o

Enlazando con bibliotecas externas – Ejemplo

calc.c

```
1 #include <stdio.h>
2 #include <math.h>
3
4 int main(void)
5 {
6     double x = 2.0;
7     double y = sqrt(x);
8     printf("La raíz cuadrada de %f es %f\n", x, y);
9     return 0;
10 }
```

```
> gcc -Wall -std=c90 calc.c -o calc
/tmp/cceIc3rZ.o: In function `main':
calc.c:(.text+0x23): undefined reference to `sqrt'
collect2: error: ld returned 1 exit status
```

- ▶ /tmp/cceIc3rZ.o
- ▶ undefined reference (.text+0x23)

Enlazando con bibliotecas externas – Ejemplo

calc.c

```
1 #include <stdio.h>
2 #include <math.h>
3
4 int main(void)
5 {
6     double x = 2.0;
7     double y = sqrt(x);
8     printf("La raíz cuadrada de %f es %f\n", x, y);
9     return 0;
10 }
```

```
> gcc -Wall -std=c90 calc.c -o calc
/tmp/cceIc3rZ.o: In function `main':
calc.c:(.text+0x23): undefined reference to `sqrt'
collect2: error: ld returned 1 exit status
```

- ▶ /tmp/cceIc3rZ.o
- ▶ undefined reference (.text+0x23)
- ▶ ld returned 1

Enlazando con bibliotecas externas – Ejemplo

calc.c

```
1 #include <stdio.h>
2 #include <math.h>
3
4 int main(void)
5 {
6     double x = 2.0;
7     double y = sqrt(x);
8     printf("La raíz cuadrada de %f es %f\n", x, y);
9     return 0;
10 }
```

```
> gcc -Wall -std=c90 calc.c -o calc
/tmp/cceIc3rZ.o: In function `main':
calc.c:(.text+0x23): undefined reference to `sqrt'
collect2: error: ld returned 1 exit status
```

- ▶ /tmp/cceIc3rZ.o
- ▶ undefined reference (.text+0x23)
- ▶ ld returned 1
- ▶ Borrar el include a math.h y re-compilar (gcc -c)

Enlazando con bibliotecas externas – Ejemplo

- ✓ Enlazado con biblioteca estática (buscar `libm.a`)

Enlazando con bibliotecas externas – Ejemplo

- ✓ Enlazado con biblioteca estática (buscar `libm.a`)

```
> gcc -static -Wall calc.c /usr/lib/x86_64-linux-gnu/libm.a\  
-o calc_static
```

Enlazando con bibliotecas externas – Ejemplo

- ✓ Enlazado con biblioteca estática (buscar `libm.a`)

```
> gcc -static -Wall calc.c /usr/lib/x86_64-linux-gnu/libm.a\  
-o calc_static
```

(Notar el flag `-static`)

Enlazando con bibliotecas externas – Ejemplo

- ✓ Enlazado con biblioteca estática (buscar `libm.a`)

```
> gcc -static -Wall calc.c /usr/lib/x86_64-linux-gnu/libm.a\  
-o calc_static
```

(Notar el flag `-static`)

- ✓ Enlazado con biblioteca dinámica

```
> gcc -Wall calc.c -lm -o calc_dynamic
```

Enlazando con bibliotecas externas – Ejemplo

- ✓ Enlazado con biblioteca estática (buscar `libm.a`)

```
> gcc -static -Wall calc.c /usr/lib/x86_64-linux-gnu/libm.a\  
      -o calc_static
```

(Notar el flag `-static`)

- ✓ Enlazado con biblioteca dinámica

```
> gcc -Wall calc.c -lm -o calc_dynamic
```

`-lNAME` enlaza contra la biblioteca `libNAME.so` (p.e. `libm.so`).

Enlazando con bibliotecas externas – Ejemplo

✓ Enlazado con biblioteca estática (buscar `libm.a`)

```
> gcc -static -Wall calc.c /usr/lib/x86_64-linux-gnu/libm.a\  
-o calc_static
```

(Notar el flag `-static`)

✓ Enlazado con biblioteca dinámica

```
> gcc -Wall calc.c -lm -o calc_dynamic
```

`-lNAME` enlaza contra la biblioteca `libNAME.so` (p.e. `libm.so`).

- ▶ Probar comando `ldd` y `file` con ambas aplicaciones.
- ▶ Ver tamaños de los binario (`ls -lh`).

Archivos de cabecera de bibliotecas

badconv.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     double x = strtod("123", NULL);
6     printf("El valor es %f\n", x);
7     return 0;
8 }
```

Archivos de cabecera de bibliotecas

badconv.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     double x = strtod("123", NULL);
6     printf("El valor es %f\n", x);
7     return 0;
8 }
```

Compilar

```
gcc -Wall badconv.c -o badconv
badconv.c: In function 'main':
badconv.c:5:14: warning: implicit declaration of function
'strtod' [-Wimplicit-function-declaration]
    double x = strtod("123", NULL);
               ^
```

Archivos de cabecera de bibliotecas

badconv.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     double x = strtod("123", NULL);
6     printf("El valor es %f\n", x);
7     return 0;
8 }
```

Compilar

```
gcc -Wall badconv.c -o badconv
badconv.c: In function 'main':
badconv.c:5:14: warning: implicit declaration of function
  'strtod' [-Wimplicit-function-declaration]
    double x = strtod("123", NULL);
               ^
```

Ejecutar

```
> ./badconv
El valor es 0.000000
```

Archivos de cabecera de bibliotecas

badconv.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     double x = strtod("123", NULL);
6     printf("El valor es %f\n", x);
7     return 0;
8 }
```

Compilar

```
gcc -Wall badconv.c -o badconv
badconv.c: In function 'main':
badconv.c:5:14: warning: implicit declaration of function
'strtod' [-Wimplicit-function-declaration]
    double x = strtod("123", NULL);
               ^
```

Ejecutar

```
> ./badconv
El valor es 0.000000
```

Agregar archivo de cabecera y probar nuevamente.

Otras opciones de compilación

ERROR: FILE.h: No such file or directory

Otras opciones de compilación

ERROR: FILE.h: No such file or directory

Qué significa?

Otras opciones de compilación

ERROR: FILE.h: No such file or directory

Qué significa? No se encuentra un archivo de cabecera de una biblioteca

Otras opciones de compilación

ERROR: FILE.h: No such file or directory

Qué significa? No se encuentra un archivo de cabecera de una biblioteca

ERROR: /usr/bin/ld: cannot find library

Otras opciones de compilación

ERROR: FILE.h: No such file or directory

Qué significa? No se encuentra un archivo de cabecera de una biblioteca

ERROR: /usr/bin/ld: cannot find library

Qué significa? No se encuentra la biblioteca compartida al enlazar

Otras opciones de compilación

ERROR: FILE.h: No such file or directory

Qué significa? No se encuentra un archivo de cabecera de una biblioteca

ERROR: /usr/bin/ld: cannot find library

Qué significa? No se encuentra la biblioteca compartida al enlazar

Por defecto, gcc busca los archivos en los siguientes directorios:

Archivos de cabecera:

- ▶ /usr/include
- ▶ /usr/local/include

Otras opciones de compilación

ERROR: FILE.h: No such file or directory

Qué significa? No se encuentra un archivo de cabecera de una biblioteca

ERROR: /usr/bin/ld: cannot find library

Qué significa? No se encuentra la biblioteca compartida al enlazar

Por defecto, gcc busca los archivos en los siguientes directorios:

Archivos de cabecera:

- ▶ /usr/include
- ▶ /usr/local/include

Bibliotecas:

- ▶ /usr/bib
- ▶ /usr/local/lib

Otras opciones de compilación

ERROR: FILE.h: No such file or directory

Qué significa? No se encuentra un archivo de cabecera de una biblioteca

ERROR: /usr/bin/ld: cannot find library

Qué significa? No se encuentra la biblioteca compartida al enlazar

Por defecto, gcc busca los archivos en los siguientes directorios:

Archivos de cabecera:

- ▶ /usr/include
- ▶ /usr/local/include

Bibliotecas:

- ▶ /usr/bib
- ▶ /usr/local/lib

Opciones de compilación para agregar rutas: -I y -L. Ejemplos:

```
> gcc -I/usr/include/newlib source.c
```

```
> gcc -L/usr/X11/lib -lX11 source.c
```

Otras opciones de compilación

ERROR: FILE.h: No such file or directory

Qué significa? No se encuentra un archivo de cabecera de una biblioteca

ERROR: /usr/bin/ld: cannot find library

Qué significa? No se encuentra la biblioteca compartida al enlazar

Por defecto, gcc busca los archivos en los siguientes directorios:

Archivos de cabecera:

- ▶ /usr/include
- ▶ /usr/local/include

Bibliotecas:

- ▶ /usr/bib
- ▶ /usr/local/lib

Opciones de compilación para agregar rutas: -I y -L. Ejemplos:

```
> gcc -I/usr/include/newlib source.c
```

```
> gcc -L/usr/X11/lib -lX11 source.c
```

- ▶ -IPATH: -I/usr/include/libusb, -I\$HOME/milib/include/, etc.
- ▶ -LPATH: -L/usr/lib/graphviz, -L\$HOME/milib/lib, etc.

Estándares del lenguaje C

gcc compila por defecto el dialecto de GNU del lenguaje C, llamado GNU C.

Estándares del lenguaje C

gcc compila por defecto el dialecto de GNU del lenguaje C, llamado GNU C.

Este dialecto incorpora el estándar oficial ANSI/ISO con varias extensiones¹ útiles para sistemas GNU, por ejemplo:

¹<https://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html>

Estándares del lenguaje C

gcc compila por defecto el dialecto de GNU del lenguaje C, llamado GNU C.

Este dialecto incorpora el estándar oficial ANSI/ISO con varias extensiones¹ útiles para sistemas GNU, por ejemplo:

- ▶ funciones definidas dentro de funciones

¹<https://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html>

Estándares del lenguaje C

gcc compila por defecto el dialecto de GNU del lenguaje C, llamado GNU C.

Este dialecto incorpora el estándar oficial ANSI/ISO con varias extensiones¹ útiles para sistemas GNU, por ejemplo:

- ▶ funciones definidas dentro de funciones
- ▶ vectores de tamaño variable

¹<https://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html>

Estándares del lenguaje C

gcc compila por defecto el dialecto de GNU del lenguaje C, llamado GNU C.

Este dialecto incorpora el estándar oficial ANSI/ISO con varias extensiones¹ útiles para sistemas GNU, por ejemplo:

- ▶ funciones definidas dentro de funciones
- ▶ vectores de tamaño variable
- ▶ estructuras vacías, etc.

¹<https://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html>

Estándares del lenguaje C

gcc compila por defecto el dialecto de GNU del lenguaje C, llamado GNU C.

Este dialecto incorpora el estándar oficial ANSI/ISO con varias extensiones¹ útiles para sistemas GNU, por ejemplo:

- ▶ funciones definidas dentro de funciones
- ▶ vectores de tamaño variable
- ▶ estructuras vacías, etc.

Opciones más comunes:

- ▶ **-ansi**: en C es equivalente a **-std=c90**

¹<https://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html>

Estándares del lenguaje C

gcc compila por defecto el dialecto de GNU del lenguaje C, llamado GNU C.

Este dialecto incorpora el estándar oficial ANSI/ISO con varias extensiones¹ útiles para sistemas GNU, por ejemplo:

- ▶ funciones definidas dentro de funciones
- ▶ vectores de tamaño variable
- ▶ estructuras vacías, etc.

Opciones más comunes:

- ▶ **-ansi**: en C es equivalente a **-std=c90**
- ▶ **-Wall**: habilita todos las advertencias (warnings)

¹<https://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html>

Estándares del lenguaje C

gcc compila por defecto el dialecto de GNU del lenguaje C, llamado GNU C.

Este dialecto incorpora el estándar oficial ANSI/ISO con varias extensiones¹ útiles para sistemas GNU, por ejemplo:

- ▶ funciones definidas dentro de funciones
- ▶ vectores de tamaño variable
- ▶ estructuras vacías, etc.

Opciones más comunes:

- ▶ **-ansi**: en C es equivalente a **-std=c90**
- ▶ **-Wall**: habilita todos las advertencias (warnings)
- ▶ **-Werror**: convierte las advertencias en errores

¹<https://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html>

Estándares del lenguaje C

gcc compila por defecto el dialecto de GNU del lenguaje C, llamado GNU C.

Este dialecto incorpora el estándar oficial ANSI/ISO con varias extensiones¹ útiles para sistemas GNU, por ejemplo:

- ▶ funciones definidas dentro de funciones
- ▶ vectores de tamaño variable
- ▶ estructuras vacías, etc.

Opciones más comunes:

- ▶ **-ansi**: en C es equivalente a **-std=c90**
- ▶ **-Wall**: habilita todos las advertencias (warnings)
- ▶ **-Werror**: convierte las advertencias en errores
- ▶ **-pedantic**: genera advertencias si se utiliza alguna extensión de GNU

¹<https://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html>

Estándares del lenguaje C

gcc compila por defecto el dialecto de GNU del lenguaje C, llamado GNU C.

Este dialecto incorpora el estándar oficial ANSI/ISO con varias extensiones¹ útiles para sistemas GNU, por ejemplo:

- ▶ funciones definidas dentro de funciones
- ▶ vectores de tamaño variable
- ▶ estructuras vacías, etc.

Opciones más comunes:

- ▶ **-ansi**: en C es equivalente a **-std=c90**
- ▶ **-Wall**: habilita todos las advertencias (warnings)
- ▶ **-Werror**: convierte las advertencias en errores
- ▶ **-pedantic**: genera advertencias si se utiliza alguna extensión de GNU
- ▶ **-std**: puede ser **-std=c90**, **-std=c99**, etc.

¹<https://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html>

Estándares del lenguaje C – ANSI/ISO estricto

gnuarray.c

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[])
4 {
5     int i, n = argc;
6     double x[n];
7
8     for(i = 0; i < n; i++)
9         x[i] = i;
10
11    return 0;
12 }
```

Ejemplo de programa con array de tamaño variable de la extensión de GNU C.

Estándares del lenguaje C – ANSI/ISO estricto

gnuarray.c

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[])
4 {
5     int i, n = argc;
6     double x[n];
7
8     for(i = 0; i < n; i++)
9         x[i] = i;
10
11    return 0;
12 }
```

Ejemplo de programa con array de tamaño variable de la extensión de GNU C.

```
> gcc -Wall -ansi gnuarray.c
```

Estándares del lenguaje C – ANSI/ISO estricto

gnuarray.c

```
1 #include <stdio.h>
2
3 int main(int argc, char *argv[])
4 {
5     int i, n = argc;
6     double x[n];
7
8     for(i = 0; i < n; i++)
9         x[i] = i;
10
11    return 0;
12 }
```

Ejemplo de programa con array de tamaño variable de la extensión de GNU C.

```
> gcc -Wall -ansi gnuarray.c
```

```
> gcc -Wall -ansi -pedantic gnuarray.c
gnuarray.c: In function 'main':
gnuarray.c:6:3: warning: ISO C90 forbids variable length
array 'x' [-Wvla]
```


Construcción de bibliotecas

Se creará una biblioteca pequeña `libsaludo` que contiene dos funciones `hola` y `adios`.

Construcción de bibliotecas

Se creará una biblioteca pequeña `libsaludo` que contiene dos funciones `hola` y `adios`.

`saludo.h`

```
1 #ifndef SALUDO_H
2 #define SALUDO_H
3
4 void hola(const char * );
5 void adios(void);
6
7 #endif
```

Construcción de bibliotecas

Se creará una biblioteca pequeña `libsaludo` que contiene dos funciones `hola` y `adios`.

saludo.h

```
1 #ifndef SALUDO_H
2 #define SALUDO_H
3
4 void hola(const char * );
5 void adios(void);
6
7 #endif
```

hola.c

```
1 #include <stdio.h>
2 #include "saludo.h"
3
4 void hola(const char * nombre)
5 {
6     printf("Hola, %s!\n", nombre);
7 }
```

Construcción de bibliotecas

Se creará una biblioteca pequeña `libsaludo` que contiene dos funciones `hola` y `adios`.

saludo.h

```
1 #ifndef SALUDO_H
2 #define SALUDO_H
3
4 void hola(const char * );
5 void adios(void);
6
7#endif
```

hola.c

```
1 #include <stdio.h>
2 #include "saludo.h"
3
4 void hola(const char * nombre)
5 {
6     printf("Hola, %s!\n", nombre);
7 }
```

adios.c

```
1 #include <stdio.h>
2 #include "saludo.h"
3
4 void adios(void)
5 {
6     printf("Adios!\n");
7 }
```

Construcción de bibliotecas – Estática

Compilar

```
> gcc -Wall -c hola.c  
> gcc -Wall -c adios.c
```

Construcción de bibliotecas – Estática

Compilar

```
> gcc -Wall -c hola.c  
> gcc -Wall -c adios.c
```

Combinar los archivos y generar la biblioteca

```
> ar cr libsaludo.a hola.o adios.o
```

Construcción de bibliotecas – Estática

Compilar

```
> gcc -Wall -c hola.c  
> gcc -Wall -c adios.c
```

Combinar los archivos y generar la biblioteca

```
> ar cr libsaludo.a hola.o adios.o
```

- ▶ **c**: crear el archivo (*.a)
- ▶ **r**: insertar un miembro (reemplazándolo si existe)

Construcción de bibliotecas – Estática

Compilar

```
> gcc -Wall -c hola.c  
> gcc -Wall -c adios.c
```

Combinar los archivos y generar la biblioteca

```
> ar cr libsaludo.a hola.o adios.o
```

- ▶ **c**: crear el archivo (*.a)
- ▶ **r**: insertar un miembro (reemplazándolo si existe)

Listar los archivos objetos de la biblioteca

```
> ar t libsaludo.a  
hola.o  
adios.o
```

Construcción de bibliotecas – Estática

Compilar

```
> gcc -Wall -c hola.c  
> gcc -Wall -c adios.c
```

Combinar los archivos y generar la biblioteca

```
> ar cr libsaludo.a hola.o adios.o
```

- ▶ **c**: crear el archivo (*.a)
- ▶ **r**: insertar un miembro (reemplazándolo si existe)

Listar los archivos objetos de la biblioteca

```
> ar t libsaludo.a  
hola.o  
adios.o
```

(Ver comando **nm –man**)

Construcción de bibliotecas – Dinámica

Compilar

```
> gcc -Wall -c -fPIC hola.c  
> gcc -Wall -c -fPIC adios.c
```

Construcción de bibliotecas – Dinámica

Compilar

```
> gcc -Wall -c -fPIC hola.c  
> gcc -Wall -c -fPIC adios.c
```

- ▶ **-fPIC**: genera código independiente de la posición (position independent code)

Construcción de bibliotecas – Dinámica

Compilar

```
> gcc -Wall -c -fPIC hola.c  
> gcc -Wall -c -fPIC adios.c
```

- ▶ **-fPIC**: genera código independiente de la posición (position independent code)

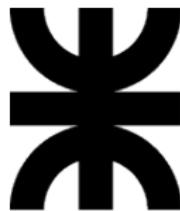
Combinar los archivos y generar la biblioteca

```
> gcc -shared hola.o adios.o -o libsaludo.so
```


Informática II

Construcción de proyectos con make

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2021 –

Construcción de proyectos con make – Necesidad

hola.h

```
1 #ifndef HOLA_H
2 #define HOLA_H
3
4 void hola(const char *nombre);
5
6 #endif
```

main.c

```
1 #include "hola.h"
2
3 int main(void)
4 {
5     hola("mundo");
6     return 0;
7 }
```

hola.c

```
1 #include <stdio.h>
2 #include "hola.h"
3
4 void hola(const char *nombre)
5 {
6     printf("Hola, %s!\n", nombre);
7 }
```

Construcción de proyectos con make – Necesidad

hola.h

```
1 #ifndef HOLA_H
2 #define HOLA_H
3
4 void hola(const char *nombre);
5
6 #endif
```

main.c

```
1 #include "hola.h"
2
3 int main(void)
4 {
5     hola("mundo");
6     return 0;
7 }
```

hola.c

```
1 #include <stdio.h>
2 #include "hola.h"
3
4 void hola(const char *nombre)
5 {
6     printf("Hola, %s!\n", nombre);
7 }
```

Compilación

```
gcc -Wall main.c hola.c -o hola
```

Construcción de proyectos con make – Necesidad

hola.h

```
1 #ifndef HOLA_H
2 #define HOLA_H
3
4 void hola(const char *nombre);
5
6 #endif
```

main.c

```
1 #include "hola.h"
2
3 int main(void)
4 {
5     hola("mundo");
6     return 0;
7 }
```

hola.c

```
1 #include <stdio.h>
2 #include "hola.h"
3
4 void hola(const char *nombre)
5 {
6     printf("Hola, %s!\n", nombre);
7 }
```

Compilación

```
gcc -Wall main.c hola.c -o hola
```

Se puede compilar separadamente cada archivo fuente

```
gcc -Wall -c main.c
gcc -Wall -c hola.c
```

Construcción de proyectos con make – Necesidad

hola.h

```
1 #ifndef HOLA_H
2 #define HOLA_H
3
4 void hola(const char *nombre);
5
6 #endif
```

main.c

```
1 #include "hola.h"
2
3 int main(void)
4 {
5     hola("mundo");
6     return 0;
7 }
```

hola.c

```
1 #include <stdio.h>
2 #include "hola.h"
3
4 void hola(const char *nombre)
5 {
6     printf("Hola, %s!\n", nombre);
7 }
```

Compilación

```
gcc -Wall main.c hola.c -o hola
```

Se puede compilar separadamente cada archivo fuente

```
gcc -Wall -c main.c
gcc -Wall -c hola.c
```

Luego unirlo con el linker

```
gcc main.o hola.o -o hola
```

Construcción de proyectos con make – Necesidad

hola.h

```
1 #ifndef HOLA_H
2 #define HOLA_H
3
4 void hola(const char *nombre);
5
6 #endif
```

main.c

```
1 #include "hola.h"
2
3 int main(void)
4 {
5     hola("mundo");
6     return 0;
7 }
```

hola.c

```
1 #include <stdio.h>
2 #include "hola.h"
3
4 void hola(const char *nombre)
5 {
6     printf("Hola, %s!\n", nombre);
7 }
```

Compilación

```
gcc -Wall main.c hola.c -o hola
```

Se puede compilar separadamente cada archivo fuente

```
gcc -Wall -c main.c
gcc -Wall -c hola.c
```

Luego unirlo con el linker

```
gcc main.o hola.o -o hola
```

Permite modificar un archivo fuente y recompilar solo el archivo modificado.

Construcción de proyectos con `make` – Ejemplo

Construcción con `make`

```
> make
gcc -c -o hola.o hola.c
gcc -c -o main.o main.c
gcc hola.o main.o -o hola
```

Construcción de proyectos con `make` – Ejemplo

Construcción con `make`

```
> make
gcc -c -o hola.o hola.c
gcc -c -o main.o main.c
gcc hola.o main.o -o hola
```

Ejecutar

```
Hola, mundo!
```

Construcción de proyectos con `make` – Ejemplo

Construcción con `make`

```
> make
gcc -c -o hola.o hola.c
gcc -c -o main.o main.c
gcc hola.o main.o -o hola
```

Ejecutar

```
Hola, mundo!
```

Modificar el archivo fuente `main.c` y reconstruir

```
> make
gcc -c -o main.o main.c
gcc hola.o main.o -o hola
```

Construcción de proyectos con `make` – Ejemplo

Construcción con `make`

```
> make
gcc -c -o hola.o hola.c
gcc -c -o main.o main.c
gcc hola.o main.o -o hola
```

Ejecutar

```
Hola, mundo!
```

Modificar el archivo fuente `main.c` y reconstruir

```
> make
gcc -c -o main.o main.c
gcc hola.o main.o -o hola
```

Ejecutar

```
Hola, InfoII!
```


Herramienta make

Herramienta para la construcción (re-construcción) de software.

Herramienta make

Herramienta para la construcción (re-construcción) de software.

- ▶ `make` simplifica el proceso de construcción de proyectos de múltiples archivos fuentes, que generalmente requieren varias llamadas al compilador.

Herramienta make

Herramienta para la construcción (re-construcción) de software.

- ▶ `make` simplifica el proceso de construcción de proyectos de múltiples archivos fuentes, que generalmente requieren varias llamadas al compilador.
- ▶ Automatiza: qué partes construir, cómo construirlas, y cuando.

Herramienta make

Herramienta para la construcción (re-construcción) de software.

- ▶ `make` simplifica el proceso de construcción de proyectos de múltiples archivos fuentes, que generalmente requieren varias llamadas al compilador.
- ▶ Automatiza: qué partes construir, cómo construirlas, y cuando.
- ▶ Le permite al programador poder concentrarse en el código.

Herramienta `make`

Herramienta para la construcción (re-construcción) de software.

- ▶ `make` simplifica el proceso de construcción de proyectos de múltiples archivos fuentes, que generalmente requieren varias llamadas al compilador.
- ▶ Automatiza: qué partes construir, cómo construirlas, y cuando.
- ▶ Le permite al programador poder concentrarse en el código.
- ▶ `make` minimiza el tiempo de construcción (determina qué archivos cambiaron), además trabaja con `dependencias`.

Herramienta `make`

Herramienta para la construcción (re-construcción) de software.

- ▶ `make` simplifica el proceso de construcción de proyectos de múltiples archivos fuentes, que generalmente requieren varias llamadas al compilador.
- ▶ Automatiza: qué partes construir, cómo construirlas, y cuando.
- ▶ Le permite al programador poder concentrarse en el código.
- ▶ `make` minimiza el tiempo de construcción (determina qué archivos cambiaron), además trabaja con `dependencias`.

Optimiza el tiempo del ciclo `editar-compilar-verificar`

Ejemplo de dependencias

```
1 /* main.c */
2 #include "a.h"
3
4
5 . . .
```

```
1 /* 2.c */
2 #include "a.h"
3 #include "b.h"
4
5 . . .
```

```
1 /* 3.c */
2 #include "b.h"
3 #include "c.h"
4
5 . . .
```

Ejemplo de dependencias

```
1 /* main.c */
2 #include "a.h"
3
4
5 . . .
```

```
1 /* 2.c */
2 #include "a.h"
3 #include "b.h"
4
5 . . .
```

```
1 /* 3.c */
2 #include "b.h"
3 #include "c.h"
4
5 . . .
```

- ▶ Si se modifica `c.h`, los archivos `main.c` y `2.c` no necesitan ser recompilados

Ejemplo de dependencias

```
1 /* main.c */
2 #include "a.h"
3
4
5 . . .
```

```
1 /* 2.c */
2 #include "a.h"
3 #include "b.h"
4
5 . . .
```

```
1 /* 3.c */
2 #include "b.h"
3 #include "c.h"
4
5 . . .
```

- ▶ Si se modifica `c.h`, los archivos `main.c` y `2.c` no necesitan ser recompilados
- ▶ El archivo `3.c` depende del archivo `c.h`

Ejemplo de dependencias

```
1 /* main.c */
2 #include "a.h"
3
4
5 . . .
```

```
1 /* 2.c */
2 #include "a.h"
3 #include "b.h"
4
5 . . .
```

```
1 /* 3.c */
2 #include "b.h"
3 #include "c.h"
4
5 . . .
```

- ▶ Si se modifica `c.h`, los archivos `main.c` y `2.c` no necesitan ser recompilados
- ▶ El archivo `3.c` depende del archivo `c.h`
- ▶ Qué pasa si se modifica `b.h` y no se recompila `2.c`

Ejemplo de dependencias

```
1 /* main.c */
2 #include "a.h"
3
4
5 . . .
```

```
1 /* 2.c */
2 #include "a.h"
3 #include "b.h"
4
5 . . .
```

```
1 /* 3.c */
2 #include "b.h"
3 #include "c.h"
4
5 . . .
```

- ▶ Si se modifica `c.h`, los archivos `main.c` y `2.c` no necesitan ser recompilados
- ▶ El archivo `3.c` depende del archivo `c.h`
- ▶ Qué pasa si se modifica `b.h` y no se recompila `2.c`

Dependencias

```
myapp: main.o 2.o 3.o
```

Ejemplo de dependencias

```
1 /* main.c */
2 #include "a.h"
3
4
5 . . .
```

```
1 /* 2.c */
2 #include "a.h"
3 #include "b.h"
4
5 . . .
```

```
1 /* 3.c */
2 #include "b.h"
3 #include "c.h"
4
5 . . .
```

- ▶ Si se modifica `c.h`, los archivos `main.c` y `2.c` no necesitan ser recompilados
- ▶ El archivo `3.c` depende del archivo `c.h`
- ▶ Qué pasa si se modifica `b.h` y no se recompila `2.c`

Dependencias

```
myapp: main.o 2.o 3.o
main.o: main.c a.h
```

Ejemplo de dependencias

```
1 /* main.c */
2 #include "a.h"
3
4
5 . . .
```

```
1 /* 2.c */
2 #include "a.h"
3 #include "b.h"
4
5 . . .
```

```
1 /* 3.c */
2 #include "b.h"
3 #include "c.h"
4
5 . . .
```

- ▶ Si se modifica `c.h`, los archivos `main.c` y `2.c` no necesitan ser recompilados
- ▶ El archivo `3.c` depende del archivo `c.h`
- ▶ Qué pasa si se modifica `b.h` y no se recompila `2.c`

Dependencias

```
myapp: main.o 2.o 3.o
main.o: main.c a.h
2.o: 2.c a.h b.h
```

Ejemplo de dependencias

```
1 /* main.c */
2 #include "a.h"
3
4
5 . . .
```

```
1 /* 2.c */
2 #include "a.h"
3 #include "b.h"
4
5 . . .
```

```
1 /* 3.c */
2 #include "b.h"
3 #include "c.h"
4
5 . . .
```

- ▶ Si se modifica `c.h`, los archivos `main.c` y `2.c` no necesitan ser recompilados
- ▶ El archivo `3.c` depende del archivo `c.h`
- ▶ Qué pasa si se modifica `b.h` y no se recompila `2.c`

Dependencias

```
myapp: main.o 2.o 3.o
main.o: main.c a.h
2.o: 2.c a.h b.h
3.o: 3.c b.h c.h
```

Ejemplo de dependencias

```
1 /* main.c */
2 #include "a.h"
3
4
5 . . .
```

```
1 /* 2.c */
2 #include "a.h"
3 #include "b.h"
4
5 . . .
```

```
1 /* 3.c */
2 #include "b.h"
3 #include "c.h"
4
5 . . .
```

- ▶ Si se modifica `c.h`, los archivos `main.c` y `2.c` no necesitan ser recompilados
- ▶ El archivo `3.c` depende del archivo `c.h`
- ▶ Qué pasa si se modifica `b.h` y no se recompila `2.c`

Dependencias

```
myapp: main.o 2.o 3.o
main.o: main.c a.h
2.o: 2.c a.h b.h
3.o: 3.c b.h c.h
```

Ver [Makefile](#) del ejemplo del comienzo

Archivo Makefile

Un archivo **Makefile** es un archivo de texto que contiene *reglas* que le indican a **make** qué construir y cómo. Una *regla* consiste en:

Archivo Makefile

Un archivo **Makefile** es un archivo de texto que contiene *reglas* que le indican a **make** qué construir y cómo. Una *regla* consiste en:

- ▶ Un *target*: lo que se debe construir (objetivo)
- ▶ Una lista de una o más *dependencias*: archivos necesarios para construir el *target* (prerrequisito)
- ▶ Una lista de *comandos* a ejecutar para construir el objetivo (receta)

Archivo Makefile

Un archivo **Makefile** es un archivo de texto que contiene *reglas* que le indican a **make** qué construir y cómo. Una *regla* consiste en:

- ▶ Un *target*: lo que se debe construir (objetivo)
- ▶ Una lista de una o más *dependencias*: archivos necesarios para construir el *target* (prerrequisito)
- ▶ Una lista de *comandos* a ejecutar para construir el objetivo (receta)

```
target: dependency dependency [...]
      command
      command
      [...]
```

Archivo Makefile

Un archivo **Makefile** es un archivo de texto que contiene *reglas* que le indican a **make** qué construir y cómo. Una *regla* consiste en:

- ▶ Un *target*: lo que se debe construir (objetivo)
- ▶ Una lista de una o más *dependencias*: archivos necesarios para construir el *target* (prerrequisito)
- ▶ Una lista de *comandos* a ejecutar para construir el objetivo (receta)

```
target: dependency dependency [...]
      command
      command
      [...]
```

Cuando se ejecuta, **make** busca los archivos **GNUmakefile**, **makefile**, y **Makefile**, en ese orden.

Archivo Makefile

Un archivo **Makefile** es un archivo de texto que contiene *reglas* que le indican a **make** qué construir y cómo. Una *regla* consiste en:

- ▶ Un *target*: lo que se debe construir (objetivo)
- ▶ Una lista de una o más *dependencias*: archivos necesarios para construir el *target* (prerrequisito)
- ▶ Una lista de *comandos* a ejecutar para construir el objetivo (receta)

```
target: dependency dependency [...]
      command
      command
      [...]
```

Cuando se ejecuta, **make** busca los archivos **GNUmakefile**, **makefile**, y **Makefile**, en ese orden.

(ver Makefiles paso-a-paso)

Archivo **Makefile** – reglas implícitas

Makefile para el ejemplo del comienzo

```
1 CC=gcc
2 CFLAGS=-Wall
3
4 hola: main.o hola.o
```

Archivo **Makefile** – reglas implícitas

Makefile para el ejemplo del comienzo

```
1 CC=gcc
2 CFLAGS=-Wall
3
4 hola: main.o hola.o
```

- ▶ make tiene muchas reglas por defecto llamadas *reglas implícitas*

Archivo **Makefile** – reglas implícitas

Makefile para el ejemplo del comienzo

```
1 CC=gcc
2 CFLAGS=-Wall
3
4 hola: main.o hola.o
```

- ▶ make tiene muchas reglas por defecto llamadas *reglas implícitas*
- ▶ Por ejemplo: indicar que los archivos .o sean construidos desde archivos .c, y que el binario sea creado enlazando los archivos .o juntos

Archivo **Makefile** – reglas implícitas

Makefile para el ejemplo del comienzo

```
1 CC=gcc
2 CFLAGS=-Wall
3
4 hola: main.o hola.o
```

- ▶ `make` tiene muchas reglas por defecto llamadas *reglas implícitas*
- ▶ Por ejemplo: indicar que los archivos `.o` sean construidos desde archivos `.c`, y que el binario sea creado enlazando los archivos `.o` juntos
- ▶ Se definen mediante las variables de `make` (`CC` y `CFLAGS`)

Archivo **Makefile** – reglas implícitas

Makefile para el ejemplo del comienzo

```
1 CC=gcc
2 CFLAGS=-Wall
3
4 hola: main.o hola.o
```

- ▶ make tiene muchas reglas por defecto llamadas *reglas implícitas*
- ▶ Por ejemplo: indicar que los archivos .o sean construidos desde archivos .c, y que el binario sea creado enlazando los archivos .o juntos
- ▶ Se definen mediante las variables de make (CC y CFLAGS)
- ▶ Para el lenguaje C
 - ▶ CC es el compilador
 - ▶ CFLAGS son opciones del compilador

Archivo **Makefile** – reglas implícitas

Makefile para el ejemplo del comienzo

```
1 CC=gcc
2 CFLAGS=-Wall
3
4 hola: main.o hola.o
```

- ▶ make tiene muchas reglas por defecto llamadas *reglas implícitas*
- ▶ Por ejemplo: indicar que los archivos .o sean construidos desde archivos .c, y que el binario sea creado enlazando los archivos .o juntos
- ▶ Se definen mediante las variables de make (CC y CFLAGS)
- ▶ Para el lenguaje C
 - ▶ CC es el compilador
 - ▶ CFLAGS son opciones del compilador
- ▶ Para el lenguaje C++
 - ▶ CXX es el compilador
 - ▶ CXXFLAGS son opciones del compilador

Archivo Makefile – Otro ejemplo

```
1 editor : editor.o screen.o keyboard.o
2     gcc -o editor editor.o screen.o keyboard.o
3
4 editor.o : editor.c editor.h keyboard.h screen.h
5     gcc -c editor.c
6
7 screen.o : screen.c screen.h
8     gcc -c screen.c
9
10 keyboard.o : keyboard.c keyboard.h
11    gcc -c keyboard.c
12
13 clean :
14     rm editor *.o
```

Archivo Makefile – Otro ejemplo

```
1 editor : editor.o screen.o keyboard.o
2     gcc -o editor editor.o screen.o keyboard.o
3
4 editor.o : editor.c editor.h keyboard.h screen.h
5     gcc -c editor.c
6
7 screen.o : screen.c screen.h
8     gcc -c screen.c
9
10 keyboard.o : keyboard.c keyboard.h
11    gcc -c keyboard.c
12
13 clean :
14     rm editor *.o
```

(Tiene 5 targets. Una por defecto.)

Archivo Makefile – Otro ejemplo

```
1 editor : editor.o screen.o keyboard.o
2     gcc -o editor editor.o screen.o keyboard.o
3
4 editor.o : editor.c editor.h keyboard.h screen.h
5     gcc -c editor.c
6
7 screen.o : screen.c screen.h
8     gcc -c screen.c
9
10 keyboard.o : keyboard.c keyboard.h
11    gcc -c keyboard.c
12
13 clean :
14     rm editor *.o
```

(Tiene 5 targets. Una por defecto.)

Construir/compilar el proyecto **editor**

```
$ make
```

Archivo Makefile – Otro ejemplo

```
1 editor : editor.o screen.o keyboard.o
2     gcc -o editor editor.o screen.o keyboard.o
3
4 editor.o : editor.c editor.h keyboard.h screen.h
5     gcc -c editor.c
6
7 screen.o : screen.c screen.h
8     gcc -c screen.c
9
10 keyboard.o : keyboard.c keyboard.h
11    gcc -c keyboard.c
12
13 clean :
14     rm editor *.o
```

(Tiene 5 targets. Una por defecto.)

Construir/compilar el proyecto **editor**

```
$ make
```

(\$ make clean)

Variables y comentarios

```
1 all: myapp
2
3 # Which compiler
4 CC = gcc
5
6 # Where are include file kept
7 INCLUDE = .
8
9 # Options for development
10 CFLAGS = -g -Wall -std=c90
11 # Options for release
12 # CFLAGS = -O -Wall -std=c90
13
14 myapp: main.o 2.o 3.o
15 $(CC) -o myapp main.o 2.o 3.o
16
17 main.o: main.c a.h
18 $(CC) -I$(INCLUDE) $(CFLAGS) -c main.c
19
20 2.o: 2.c a.h b.h
21 $(CC) -I$(INCLUDE) $(CFLAGS) -c 2.c
22
23 3.o: 3.c b.h c.h
24 $(CC) -I$(INCLUDE) $(CFLAGS) -c 3.c
```


Actividad práctica

Descargar los archivos **Makefile** de ejemplo y recuerde usar la opción **-f** de la aplicación **make** para indicarle el archivo **Makefile** a utilizar. Realizar los siguientes pasos:

1. Copiar el archivo **Makefile1** a **Makefile**, ejecutar **make** y analizar el error

Actividad práctica

Descargar los archivos **Makefile** de ejemplo y recuerde usar la opción **-f** de la aplicación **make** para indicarle el archivo **Makefile** a utilizar. Realizar los siguientes pasos:

1. Copiar el archivo **Makefile1** a **Makefile**, ejecutar **make** y analizar el error
2. Editar **main.c**, **2.c** y **3.c**, como se muestra a continuación, volver a ejecutar **make** y analizar el error

Actividad práctica

Descargar los archivos **Makefile** de ejemplo y recuerde usar la opción **-f** de la aplicación **make** para indicarle el archivo **Makefile** a utilizar. Realizar los siguientes pasos:

1. Copiar el archivo **Makefile1** a **Makefile**, ejecutar **make** y analizar el error
2. Editar **main.c**, **2.c** y **3.c**, como se muestra a continuación, volver a ejecutar **make** y analizar el error
3. Crear archivos header haciendo
 > **touch {a.h,b.h,c.h}**
y volver a ejecutar **make**

Actividad práctica

Descargar los archivos **Makefile** de ejemplo y recuerde usar la opción **-f** de la aplicación **make** para indicarle el archivo **Makefile** a utilizar. Realizar los siguientes pasos:

1. Copiar el archivo **Makefile1** a **Makefile**, ejecutar **make** y analizar el error
2. Editar **main.c**, **2.c** y **3.c**, como se muestra a continuación, volver a ejecutar **make** y analizar el error
3. Crear archivos header haciendo
 > **touch {a.h,b.h,c.h}**
y volver a ejecutar **make**
4. > **make -f Makefile1**

Actividad práctica

Descargar los archivos **Makefile** de ejemplo y recuerde usar la opción **-f** de la aplicación **make** para indicarle el archivo **Makefile** a utilizar. Realizar los siguientes pasos:

1. Copiar el archivo **Makefile1** a **Makefile**, ejecutar **make** y analizar el error
2. Editar **main.c**, **2.c** y **3.c**, como se muestra a continuación, volver a ejecutar **make** y analizar el error
3. Crear archivos header haciendo
 > **touch {a.h,b.h,c.h}**
y volver a ejecutar **make**
4. > **make -f Makefile1**
5. Hacer
 > **touch b.h**
y volver a ejecutar **make**

Actividad práctica

Descargar los archivos **Makefile** de ejemplo y recuerde usar la opción **-f** de la aplicación **make** para indicarle el archivo **Makefile** a utilizar. Realizar los siguientes pasos:

1. Copiar el archivo **Makefile1** a **Makefile**, ejecutar **make** y analizar el error
2. Editar **main.c**, **2.c** y **3.c**, como se muestra a continuación, volver a ejecutar **make** y analizar el error
3. Crear archivos header haciendo
> **touch {a.h,b.h,c.h}**
y volver a ejecutar **make**
4. > **make -f Makefile1**
5. Hacer
> **touch b.h**
y volver a ejecutar **make**
6. Eliminar el archivo **2.o**, y probar nuevamente

Actividad práctica

```
/* main.c */
#include <stdio.h>
#include "a.h"

extern void function_two();
extern void function_three();

int main()
{
    function_two();
    function_three();
    return 0;
}
```

```
/* 2.c */
#include "a.h"
#include "b.h"

void function_two()
{
}
```

```
/* 3.c */
#include "b.h"
#include "c.h"

void function_three()
{
}
```

Actividad práctica

Ejemplo de archivo Makefile básico

```
1 myapp: main.o 2.o 3.o
2   gcc -o myapp main.o 2.o 3.o
3
4 main.o: main.c a.h
5   gcc -c main.c
6
7 2.o: 2.c a.h b.h
8   gcc -c 2.c
9
10 3.o: 3.c b.h c.h
11   gcc -c 3.c
```

Actividad práctica

Ejemplo de archivo **Makefile** básico

```
1 myapp: main.o 2.o 3.o
2   gcc -o myapp main.o 2.o 3.o
3
4 main.o: main.c a.h
5   gcc -c main.c
6
7 2.o: 2.c a.h b.h
8   gcc -c 2.c
9
10 3.o: 3.c b.h c.h
11   gcc -c 3.c
```

Los demás archivos **Makefile** van agregando:

- ▶ Variables CC, CFLAGS, INCLUDE
- ▶ Comentarios y target **clean**
- ▶ Variables @ y <

Informática II

Manejo de archivos en lenguaje C (GNU/Linux)

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2021 –

Manejo de archivos en C

Estándar de la jerarquía del sistema de archivo (FHS: Filesystem Hierarchy Standard) define los directorios principales y su contenido en los sistemas Linux.

Manejo de archivos en C

Estándar de la jerarquía del sistema de archivo (FHS: Filesystem Hierarchy Standard) define los directorios principales y su contenido en los sistemas Linux.

```
/--  
|-- bin  
|-- boot  
|-- dev  
|-- etc  
|-- home  
|-- lib  
|-- media  
|-- mnt  
|-- opt  
|-- proc  
|-- root  
|-- sys  
|-- usr  
|-- var
```

Manejo de archivos en C

Estándar de la jerarquía del sistema de archivo (FHS: Filesystem Hierarchy Standard) define los directorios principales y su contenido en los sistemas Linux.

```
/--          ► /bin: ejecutables (binarios) de los comandos básicos
|-- bin
|-- boot
|-- dev
|-- etc
|-- home
|-- lib
|-- media
|-- mnt
|-- opt
|-- proc
|-- root
|-- sys
|-- usr
|-- var
```

Manejo de archivos en C

Estándar de la jerarquía del sistema de archivo (FHS: Filesystem Hierarchy Standard) define los directorios principales y su contenido en los sistemas Linux.

- /--
 - |-- bin
 - |-- boot
 - |-- dev
 - |-- etc
 - |-- home
 - |-- lib
 - |-- media
 - |-- mnt
 - |-- opt
 - |-- proc
 - |-- root
 - |-- sys
 - |-- usr
 - |-- var
- ▶ /bin: ejecutables (binarios) de los comandos básicos
- ▶ /boot: cargador de arranque (boot loader), Kernel, archivos **initrd**

Manejo de archivos en C

Estándar de la jerarquía del sistema de archivo (FHS: Filesystem Hierarchy Standard) define los directorios principales y su contenido en los sistemas Linux.

- /--
 - |-- bin
 - |-- boot
 - |-- dev
 - |-- etc
 - |-- home
 - |-- lib
 - |-- media
 - |-- mnt
 - |-- opt
 - |-- proc
 - |-- root
 - |-- sys
 - |-- usr
 - |-- var
- ▶ /bin: ejecutables (binarios) de los comandos básicos
- ▶ /boot: cargador de arranque (boot loader), Kernel, archivos **initrd**
- ▶ /etc: archivos de configuración del sistema

Manejo de archivos en C

Estándar de la jerarquía del sistema de archivo (FHS: Filesystem Hierarchy Standard) define los directorios principales y su contenido en los sistemas Linux.

- /--
 - |-- bin
 - |-- boot
 - |-- dev
 - |-- etc
 - |-- home
 - |-- lib
 - |-- media
 - |-- mnt
 - |-- opt
 - |-- proc
 - |-- root
 - |-- sys
 - |-- usr
 - |-- var
- ▶ /bin: ejecutables (binarios) de los comandos básicos
- ▶ /boot: cargador de arranque (boot loader), Kernel, archivos **initrd**
- ▶ /etc: archivos de configuración del sistema
- ▶ /home: directorios de usuarios (/home/gonzalo)

Manejo de archivos en C

Estándar de la jerarquía del sistema de archivo (FHS: Filesystem Hierarchy Standard) define los directorios principales y su contenido en los sistemas Linux.

- /--
 - |-- bin
 - |-- boot
 - |-- dev
 - |-- etc
 - |-- home
 - |-- lib
 - |-- media
 - |-- mnt
 - |-- opt
 - |-- proc
 - |-- root
 - |-- sys
 - |-- usr
 - |-- var
- ▶ /bin: ejecutables (binarios) de los comandos básicos
- ▶ /boot: cargador de arranque (boot loader), Kernel, archivos **initrd**
- ▶ /etc: archivos de configuración del sistema
- ▶ /home: directorios de usuarios (**/home/gonzalo**)
- ▶ /mnt: sistemas de archivos montados temporalmente

Manejo de archivos en C

Estándar de la jerarquía del sistema de archivo (FHS: Filesystem Hierarchy Standard) define los directorios principales y su contenido en los sistemas Linux.

- /--
 - |-- bin
 - |-- boot
 - |-- dev
 - |-- etc
 - |-- home
 - |-- lib
 - |-- media
 - |-- mnt
 - |-- opt
 - |-- proc
 - |-- root
 - |-- sys
 - |-- usr
 - |-- var
- ▶ /bin: ejecutables (binarios) de los comandos básicos
- ▶ /boot: cargador de arranque (boot loader), Kernel, archivos **initrd**
- ▶ /etc: archivos de configuración del sistema
- ▶ /home: directorios de usuarios (**/home/gonzalo**)
- ▶ /mnt: sistemas de archivos montados temporalmente
- ▶ /usr: jerarquía secundaria para datos compartidos (read-only)

Manejo de archivos en C

Estándar de la jerarquía del sistema de archivo (FHS: Filesystem Hierarchy Standard) define los directorios principales y su contenido en los sistemas Linux.

- /--
 - |-- bin ► /bin: ejecutables (binarios) de los comandos básicos
 - |-- boot ► /boot: cargador de arranque (boot loader), Kernel, archivos **initrd**
 - |-- dev
 - |-- etc
 - |-- home
 - |-- lib
 - |-- media
 - |-- mnt
 - |-- opt
 - |-- proc
 - |-- root
 - |-- sys
 - |-- usr
 - |-- var

Manejo de archivos en C

Estándar de la jerarquía del sistema de archivo (FHS: Filesystem Hierarchy Standard) define los directorios principales y su contenido en los sistemas Linux.

- ▶ /bin: ejecutables (binarios) de los comandos básicos
 - ▶ /boot: cargador de arranque (boot loader), Kernel, archivos **initrd**
 - ▶ /etc: archivos de configuración del sistema
 - ▶ /home: directorios de usuarios (**/home/gonzalo**)
 - ▶ /mnt: sistemas de archivos montados temporalmente
 - ▶ /usr: jerarquía secundaria para datos compartidos (read-only)
 - ▶ /opt: complementos de paquetes de software de aplicación
 - ▶ /proc: sistema de archivo virtual, estado de procesos
- |-- bin
 - |-- boot
 - |-- dev
 - |-- etc
 - |-- home
 - |-- lib
 - |-- media
 - |-- mnt
 - |-- opt
 - |-- proc
 - |-- root
 - |-- sys
 - |-- usr
 - |-- var

Manejo de archivos en C

Estándar de la jerarquía del sistema de archivo (FHS: Filesystem Hierarchy Standard) define los directorios principales y su contenido en los sistemas Linux.

- ▶ /bin: ejecutables (binarios) de los comandos básicos
- ▶ /boot: cargador de arranque (boot loader), Kernel, archivos **initrd**
- ▶ /etc: archivos de configuración del sistema
- ▶ /home: directorios de usuarios (**/home/gonzalo**)
- ▶ /mnt: sistemas de archivos montados temporalmente
- ▶ /usr: jerarquía secundaria para datos compartidos (read-only)
- ▶ /opt: complementos de paquetes de software de aplicación
- ▶ /proc: sistema de archivo virtual, estado de procesos
- ▶ /sys: sistema de archivos para objetos exportados del Kernel

Manejo de archivos en C

Estándar de la jerarquía del sistema de archivo (FHS: Filesystem Hierarchy Standard) define los directorios principales y su contenido en los sistemas Linux.

- - |-- bin
 - ▶ /bin: ejecutables (binarios) de los comandos básicos
 - |-- boot
 - ▶ /boot: cargador de arranque (boot loader), Kernel, archivos **initrd**
 - |-- dev
 - ▶ /etc: archivos de configuración del sistema
 - |-- etc
 - ▶ /home: directorios de usuarios (**/home/gonzalo**)
 - |-- home
 - ▶ /mnt: sistemas de archivos montados temporalmente
 - |-- lib
 - ▶ /usr: jerarquía secundaria para datos compartidos (read-only)
 - |-- media
 - ▶ /opt: complementos de paquetes de software de aplicación
 - |-- mnt
 - ▶ /proc: sistema de archivo virtual, estado de procesos
 - |-- opt
 - ▶ /sys: sistema de archivos para objetos exportados del Kernel
 - |-- proc
 - ▶ /dev: archivos de dispositivos (caracteres o bloques)
 - |-- root
 - ▶ /sys: sistema de archivos para objetos exportados del Kernel
 - |-- sys
 - ▶ /dev: archivos de dispositivos (caracteres o bloques)
 - |-- usr
 - ▶ /dev: archivos de dispositivos (caracteres o bloques)
 - |-- var
 - ▶ /dev: archivos de dispositivos (caracteres o bloques)

Manejo de archivos en C

Cada archivo tiene un nombre y algunas propiedades como la fecha de creación/modificación, permisos, etc.

Manejo de archivos en C

Cada archivo tiene un nombre y algunas propiedades como la fecha de creación/modificación, permisos, etc.

```
crw-rw---- 1 root dialout 188, 0 jul 25 10:46 /dev/ttyUSB0
-----^ ^ ^ ^ ^ ^ ^ |----- nombre del archivo
-----| | | | | |----- minutos : Fecha y
-----| | | | |----- hora : hora de la
-----| | | |----- día del mes : última
-----| | |----- mes : modificación

-----| |----- Tamaño en bytes
-----| |----- Nombre del grupo
-----| |----- Nombre del propietario
-----| |----- nro. de enlace rígido (hard link)

-----| |----- 001 permiso de ejecución : Para
-----| |----- 002 permiso de escritura : un usuario
-----| |----- 004 permiso de lectura : cualquiera
-----| |----- 010 permiso de ejecución : Para usuario
-----| |----- 020 permiso de escritura : perteneciente
-----| |----- 040 permiso de lectura : al grupo
-----| |----- 100 permiso de ejecución : Para usuario
-----| |----- 200 permiso de escritura : propietario
-----| |----- 400 permiso de lectura :

-----|----- Tipo de archivo
```

Manejo de archivos en C

- ▶ Cada programa en ejecución (proceso) tiene una cantidad de archivos asociados (descriptor de archivo).

Manejo de archivos en C

- ▶ Cada programa en ejecución (proceso) tiene una cantidad de archivos asociados (descriptor de archivo).
- ▶ **Descriptores de archivos:** son números enteros pequeños que se pueden utilizar para acceder a estos archivos o dispositivos.

Manejo de archivos en C

- ▶ Cada programa en ejecución (proceso) tiene una cantidad de archivos asociados (descriptor de archivo).
- ▶ **Descriptores de archivos:** son números enteros pequeños que se pueden utilizar para acceder a estos archivos o dispositivos.

Todo programa tiene abierto tres descriptores de archivos:

- ▶ 0: entrada estándar (`stdin`)
- ▶ 1: salida estándar (`stdout`)
- ▶ 2: error estándar (`stderr`)

Manejo de archivos en C – Ejemplos

escritura.c

```
1 #include <unistd.h>
2
3 int main(void)
4 {
5     if((write(1, "Hola mundo!\n", 12)) != 12)
6         write(2, "ERROR\n", 6);
7
8     return 0;
9 }
```

Manejo de archivos en C – Ejemplos

lectura.c

```
1 #include <unistd.h>
2
3 int main(void) {
4     char buffer[128];
5     int nread;
6
7     nread = read(0, buffer, 128);
8     if(nread == -1)
9         write(2, "Error de lectura\n", 17);
10
11    if((write(1, buffer, nread)) != nread)
12        write(2, "Error de escritura\n", 19);
13
14    return 0;
15 }
```

Manejo de archivos en C – Ejemplos

lectura.c

```
1 #include <unistd.h>
2
3 int main(void) {
4     char buffer[128];
5     int nread;
6
7     nread = read(0, buffer, 128);
8     if(nread == -1)
9         write(2, "Error de lectura\n", 17);
10
11    if((write(1, buffer, nread)) != nread)
12        write(2, "Error de escritura\n", 19);
13
14    return 0;
15 }
```

- ▶ Compilar y ejecutar (con/sin redirección)

Manejo de archivos en C – Ejemplos

lectura.c

```
1 #include <unistd.h>
2
3 int main(void) {
4     char buffer[128];
5     int nread;
6
7     nread = read(0, buffer, 128);
8     if(nread == -1)
9         write(2, "Error de lectura\n", 17);
10
11    if((write(1, buffer, nread)) != nread)
12        write(2, "Error de escritura\n", 19);
13
14    return 0;
15 }
```

- ▶ Compilar y ejecutar (con/sin redirección)
- ▶ Ejecutar con redirección
➤ ./a.out < prueba.txt

Manejo de archivos en C – Ejemplos

lectura.c

```
1 #include <unistd.h>
2
3 int main(void) {
4     char buffer[128];
5     int nread;
6
7     nread = read(0, buffer, 128);
8     if(nread == -1)
9         write(2, "Error de lectura\n", 17);
10
11    if((write(1, buffer, nread)) != nread)
12        write(2, "Error de escritura\n", 19);
13
14    return 0;
15 }
```

- ▶ Compilar y ejecutar (con/sin redirección)
- ▶ Ejecutar con redirección
 - > ./a.out < prueba.txt
- ▶ Ver número de proceso
 - > ps aux | grep a.out

Manejo de archivos en C – Ejemplos

lectura.c

```
1 #include <unistd.h>
2
3 int main(void) {
4     char buffer[128];
5     int nread;
6
7     nread = read(0, buffer, 128);
8     if(nread == -1)
9         write(2, "Error de lectura\n", 17);
10
11    if((write(1, buffer, nread)) != nread)
12        write(2, "Error de escritura\n", 19);
13
14    return 0;
15 }
```

- ▶ Compilar y ejecutar (con/sin redirección)
- ▶ Ejecutar con redirección
 - > ./a.out < prueba.txt
- ▶ Ver número de proceso
 - > ps aux | grep a.out
- ▶ Ver directorio /proc/<PID>

Manejo de archivos en C – Ejemplos

lectura.c

```
1 #include <unistd.h>
2
3 int main(void) {
4     char buffer[128];
5     int nread;
6
7     nread = read(0, buffer, 128);
8     if(nread == -1)
9         write(2, "Error de lectura\n", 17);
10
11    if((write(1, buffer, nread)) != nread)
12        write(2, "Error de escritura\n", 19);
13
14    return 0;
15 }
```

- ▶ Compilar y ejecutar (con/sin redirección)
- ▶ Ejecutar con redirección
 - > ./a.out < prueba.txt
- ▶ Ver número de proceso
 - > ps aux | grep a.out
- ▶ Ver directorio /proc/<PID>
- ▶ Ver /proc/<PID>/fd

Manejo de archivos en C – Ejemplos

lectura.c

```
1 #include <unistd.h>
2
3 int main(void) {
4     char buffer[128];
5     int nread;
6
7     nread = read(0, buffer, 128);
8     if(nread == -1)
9         write(2, "Error de lectura\n", 17);
10
11    if((write(1, buffer, nread)) != nread)
12        write(2, "Error de escritura\n", 19);
13
14    return 0;
15 }
```

- ▶ Compilar y ejecutar (con/sin redirección)
- ▶ Ejecutar con redirección
 - > ./a.out < prueba.txt
- ▶ Ver número de proceso
 - > ps aux | grep a.out
- ▶ Ver directorio /proc/<PID>
- ▶ Ver /proc/<PID>/fd
- ▶ Desde otra terminal
 - > echo "Hola" >
 - /proc/<PID>/fd/0

Manejo de archivos en C

Los programas pueden manejar archivos de disco, puerto serie, y otros dispositivos (excepto conexiones de red), todos de la misma forma.

Manejo de archivos en C

Los programas pueden manejar archivos de disco, puerto serie, y otros dispositivos (excepto conexiones de red), todos de la misma forma.

Archivos de dispositivos

Manejo de archivos en C

Los programas pueden manejar archivos de disco, puerto serie, y otros dispositivos (excepto conexiones de red), todos de la misma forma.

Archivos de dispositivos

- ▶ Los dispositivos de hardware se representan por archivos (mapean).

Manejo de archivos en C

Los programas pueden manejar archivos de disco, puerto serie, y otros dispositivos (excepto conexiones de red), todos de la misma forma.

Archivos de dispositivos

- ▶ Los dispositivos de hardware se representan por archivos (mapean).
- ▶ Los dispositivos se clasifican en: dispositivos de **caracteres** o de **bloques**.

Manejo de archivos en C

Los programas pueden manejar archivos de disco, puerto serie, y otros dispositivos (excepto conexiones de red), todos de la misma forma.

Archivos de dispositivos

- ▶ Los dispositivos de hardware se representan por archivos (mapean).
- ▶ Los dispositivos se clasifican en: dispositivos de **caracteres** o de **bloques**.

```
> ls -l /dev
crw----- 1 root root      5,   1 jul 10 13:55 console
crw-rw-rw- 1 root root    1,   3 jul 10 13:54 null
```

Manejo de archivos en C

Los programas pueden manejar archivos de disco, puerto serie, y otros dispositivos (excepto conexiones de red), todos de la misma forma.

Archivos de dispositivos

- ▶ Los dispositivos de hardware se representan por archivos (mapean).
- ▶ Los dispositivos se clasifican en: dispositivos de **caracteres** o de **bloques**.

```
> ls -l /dev
crw----- 1 root root      5,   1 jul 10 13:55 console
crw-rw-rw- 1 root root      1,   3 jul 10 13:54 null
brw-rw---- 1 root disk     8,   0 jul 19 13:51 sda
brw-rw---- 1 root disk     8,   1 jul 19 13:51 sda1
brw-rw---- 1 root disk     8,   2 jul 19 13:51 sda2
brw-rw---- 1 root disk     8,   3 jul 19 13:51 sda3
```

Manejo de archivos en C

Los programas pueden manejar archivos de disco, puerto serie, y otros dispositivos (excepto conexiones de red), todos de la misma forma.

Archivos de dispositivos

- ▶ Los dispositivos de hardware se representan por archivos (mapean).
- ▶ Los dispositivos se clasifican en: dispositivos de **caracteres** o de **bloques**.

```
> ls -l /dev
crw----- 1 root root      5,   1 jul 10 13:55 console
crw-rw-rw- 1 root root      1,   3 jul 10 13:54 null
brw-rw---- 1 root disk     8,   0 jul 19 13:51 sda
brw-rw---- 1 root disk     8,   1 jul 19 13:51 sda1
brw-rw---- 1 root disk     8,   2 jul 19 13:51 sda2
brw-rw---- 1 root disk     8,   3 jul 19 13:51 sda3
lrwxrwxrwx 1 root root    15 jul 10 13:53 stderr -> /proc/self/fd/2
lrwxrwxrwx 1 root root    15 jul 10 13:53 stdin -> /proc/self/fd/0
lrwxrwxrwx 1 root root    15 jul 10 13:53 stdout -> /proc/self/fd/1
```

Manejo de archivos en C

Los programas pueden manejar archivos de disco, puerto serie, y otros dispositivos (excepto conexiones de red), todos de la misma forma.

Archivos de dispositivos

- ▶ Los dispositivos de hardware se representan por archivos (mapean).
- ▶ Los dispositivos se clasifican en: dispositivos de **caracteres** o de **bloques**.

```
> ls -l /dev
crw----- 1 root root      5,   1 jul 10 13:55 console
crw-rw-rw- 1 root root      1,   3 jul 10 13:54 null
brw-rw---- 1 root disk     8,   0 jul 19 13:51 sda
brw-rw---- 1 root disk     8,   1 jul 19 13:51 sda1
brw-rw---- 1 root disk     8,   2 jul 19 13:51 sda2
brw-rw---- 1 root disk     8,   3 jul 19 13:51 sda3
lrwxrwxrwx 1 root root    15 jul 10 13:53 stderr -> /proc/self/fd/2
lrwxrwxrwx 1 root root    15 jul 10 13:53 stdin -> /proc/self/fd/0
lrwxrwxrwx 1 root root    15 jul 10 13:53 stdout -> /proc/self/fd/1
crw-rw---- 1 root dialout   4,  64 jul 10 13:53 ttyS0
crw-rw---- 1 root dialout  188,   0 jul 19 19:10 ttyUSB0
```

Manejo de archivos en C

Acceso a archivos

- ▶ Funciones de bajo nivel (llamada al sistema y drives de dispositivos)
- ▶ Funciones de alto nivel (biblioteca de entrada/salida) [buffers]

Manejo de archivos en C

Acceso a archivos

- ▶ Funciones de bajo nivel (llamada al sistema y drives de dispositivos)
- ▶ Funciones de alto nivel (biblioteca de entrada/salida) [buffers]

- ▶ **Bajo nivel:** Archivo de cabecera
`unistd.h` – En los lenguajes C y C++
este archivo de cabecera define la API
que brinda acceso al sistema operativo
POSIX (descriptor de archivo).

Manejo de archivos en C

Acceso a archivos

- ▶ Funciones de bajo nivel (llamada al sistema y drives de dispositivos)
- ▶ Funciones de alto nivel (biblioteca de entrada/salida) [buffers]

- ▶ **Bajo nivel:** Archivo de cabecera
unistd.h – En los lenguajes C y C++
este archivo de cabecera define la API
que brinda acceso al sistema operativo
POSIX (descriptor de archivo).
(open, close, write, read, etc.)

Manejo de archivos en C

Acceso a archivos

- ▶ Funciones de bajo nivel (llamada al sistema y drives de dispositivos)
 - ▶ Funciones de alto nivel (biblioteca de entrada/salida) [buffers]
-
- ▶ **Bajo nivel:** Archivo de cabecera
`unistd.h` – En los lenguajes C y C++ este archivo de cabecera define la API que brinda acceso al sistema operativo POSIX (descriptor de archivo).
(`open`, `close`, `write`, `read`, etc.)
 - ▶ **Alto nivel:** Archivo de cabecera
`stdio.h` (ANSI C) – Manejo de *stream*, se implementa como puntero a estructura de tipo `FILE*`.

Manejo de archivos en C

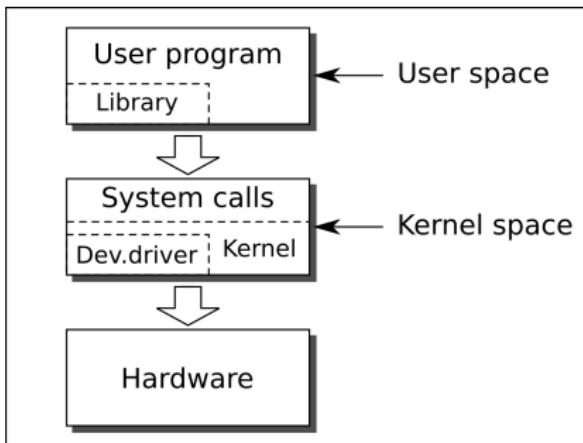
Acceso a archivos

- ▶ Funciones de bajo nivel (llamada al sistema y drives de dispositivos)
 - ▶ Funciones de alto nivel (biblioteca de entrada/salida) [buffers]
-
- ▶ **Bajo nivel:** Archivo de cabecera
`unistd.h` – En los lenguajes C y C++ este archivo de cabecera define la API que brinda acceso al sistema operativo POSIX (descriptor de archivo).
(`open`, `close`, `write`, `read`, etc.)
 - ▶ **Alto nivel:** Archivo de cabecera
`stdio.h` (ANSI C) – Manejo de *stream*, se implementa como puntero a estructura de tipo `FILE*`.
(`fopen`, `fclose`, `fwrite`, `fread`, etc.)

Manejo de archivos en C

Acceso a archivos

- ▶ Funciones de bajo nivel (llamada al sistema y drives de dispositivos)
 - ▶ Funciones de alto nivel (biblioteca de entrada/salida) [buffers]
-
- ▶ **Bajo nivel:** Archivo de cabecera `unistd.h` – En los lenguajes C y C++ este archivo de cabecera define la API que brinda acceso al sistema operativo POSIX (descriptor de archivo). (`open`, `close`, `write`, `read`, etc.)
 - ▶ **Alto nivel:** Archivo de cabecera `stdio.h` (ANSI C) – Manejo de *stream*, se implementa como puntero a estructura de tipo `FILE*`. (`fopen`, `fclose`, `fwrite`, `fread`, etc.)



Manejo de archivos en C – Funciones de bajo nivel

Se utilizan cinco funciones: `open`, `close`, `write`, `read`, e `ioctl`.

Manejo de archivos en C – Funciones de bajo nivel

Se utilizan cinco funciones: `open`, `close`, `write`, `read`, e `ioctl`.

- ▶ `open()`: Abrir archivo o dispositivo

Manejo de archivos en C – Funciones de bajo nivel

Se utilizan cinco funciones: `open`, `close`, `write`, `read`, e `ioctl`.

- ▶ `open()`: Abrir archivo o dispositivo
- ▶ `close()`: Cerrar archivo o dispositivo

Manejo de archivos en C – Funciones de bajo nivel

Se utilizan cinco funciones: `open`, `close`, `write`, `read`, e `ioctl`.

- ▶ `open()`: Abrir archivo o dispositivo
- ▶ `close()`: Cerrar archivo o dispositivo
- ▶ `read()`: Leer archivo o dispositivo

Manejo de archivos en C – Funciones de bajo nivel

Se utilizan cinco funciones: `open`, `close`, `write`, `read`, e `ioctl`.

- ▶ `open()`: Abrir archivo o dispositivo
- ▶ `close()`: Cerrar archivo o dispositivo
- ▶ `read()`: Leer archivo o dispositivo
- ▶ `write()`: Escribir archivo o dispositivo

Manejo de archivos en C – Funciones de bajo nivel

Se utilizan cinco funciones: `open`, `close`, `write`, `read`, e `ioctl`.

- ▶ `open()`: Abrir archivo o dispositivo
- ▶ `close()`: Cerrar archivo o dispositivo
- ▶ `read()`: Leer archivo o dispositivo
- ▶ `write()`: Escribir archivo o dispositivo
- ▶ `ioctl()`: Intercambiar información de control con el driver

Manejo de archivos en C – Funciones de bajo nivel

Se utilizan cinco funciones: `open`, `close`, `write`, `read`, e `ioctrl`.

- ▶ `open()`: Abrir archivo o dispositivo
- ▶ `close()`: Cerrar archivo o dispositivo
- ▶ `read()`: Leer archivo o dispositivo
- ▶ `write()`: Escribir archivo o dispositivo
- ▶ `ioctrl()`: Intercambiar información de control con el driver

En el Kernel están los *drivers de dispositivos* (device drivers): interfaz de bajo nivel para el control de hardware.

Manejo de archivos en C – Funciones de bajo nivel

Se utilizan cinco funciones: `open`, `close`, `write`, `read`, e `ioctrl`.

- ▶ `open()`: Abrir archivo o dispositivo
- ▶ `close()`: Cerrar archivo o dispositivo
- ▶ `read()`: Leer archivo o dispositivo
- ▶ `write()`: Escribir archivo o dispositivo
- ▶ `ioctrl()`: Intercambiar información de control con el driver

En el Kernel están los *drivers de dispositivos* (device drivers): interfaz de bajo nivel para el control de hardware.

Descriptores de archivos abiertos en cualquier programa: 0, 1 y 2

Manejo de archivos en C – Funciones de bajo nivel

Abrir archivo

```
#include <fcntl.h> /* File control definitions */
#include <unistd.h> /* UNIX standard function definitions */

int open(const char* path, int oflags);
```

- ▶ **path:** nombre del archivo o dispositivo
- ▶ **oflags:** indica acciones al abrir el archivo (`O_RDONLY`, `O_WRONLY`, `O_RDWR`)

Devuelve el *descriptor de archivo* (entero no negativo) si tuvo éxito, o -1 si falló.

Manejo de archivos en C – Funciones de bajo nivel

Abrir archivo

```
#include <fcntl.h> /* File control definitions */  
#include <unistd.h> /* UNIX standard function definitions */  
  
int open(const char* path, int oflags);
```

- ▶ **path:** nombre del archivo o dispositivo
- ▶ **oflags:** indica acciones al abrir el archivo (`O_RDONLY`, `O_WRONLY`, `O_RDWR`)

Devuelve el *descriptor de archivo* (entero no negativo) si tuvo éxito, o -1 si falló.

Cerrar archivo

```
#include <fcntl.h> /* File control definitions */  
#include <unistd.h> /* UNIX standard function definitions */  
  
int close(int fildes);
```

- ▶ **fildes:** descriptor de archivo

Manejo de archivos en C – Funciones de bajo nivel

Leer archivo

```
size_t read(int fildes, void *buf, size_t nbytes);
```

Escribir archivo

```
size_t write(int fildes, const void *buf, size_t nbytes);
```

I/O control

```
int ioctl(int fildes, int cmd, ...);
```

- ▶ **buf**: Buffer para la lectura/escritura
- ▶ **nbytes**: Cantidad de bytes a leer/escribir
- ▶ **cmd**: Acción a realizar sobre el archivo

Manejo de archivos en C – Funciones de alto nivel

Algunas de las funciones de alto nivel son:

Manejo de archivos en C – Funciones de alto nivel

Algunas de las funciones de alto nivel son:

- ▶ `fopen`, `fclose`

Manejo de archivos en C – Funciones de alto nivel

Algunas de las funciones de alto nivel son:

- ▶ `fopen`, `fclose`
- ▶ `fwrite`, `fread`

Manejo de archivos en C – Funciones de alto nivel

Algunas de las funciones de alto nivel son:

- ▶ `fopen`, `fclose`
- ▶ `fwrite`, `fread`
- ▶ `fflush`, `fseek`

Manejo de archivos en C – Funciones de alto nivel

Algunas de las funciones de alto nivel son:

- ▶ `fopen, fclose`
- ▶ `fwrite, fread`
- ▶ `fflush, fseek`
- ▶ `fputs, fgets`

Manejo de archivos en C – Funciones de alto nivel

Algunas de las funciones de alto nivel son:

- ▶ `fopen, fclose`
- ▶ `fwrite, fread`
- ▶ `fflush, fseek`
- ▶ `fputs, fgets`
- ▶ `fscanf, fprintf`

Manejo de archivos en C – Funciones de alto nivel

Algunas de las funciones de alto nivel son:

- ▶ `fopen, fclose`
- ▶ `fwrite, fread`
- ▶ `fflush, fseek`
- ▶ `fputs, fgets`
- ▶ `fscanf, fprintf`
- ▶ `getline, getdelim`

Manejo de archivos en C – Funciones de alto nivel

Algunas de las funciones de alto nivel son:

- ▶ `fopen, fclose`
- ▶ `fwrite, fread`
- ▶ `fflush, fseek`
- ▶ `fputs, fgets`
- ▶ `fscanf, fprintf`
- ▶ `getline, getdelim`

Flujos de datos abiertos en cualquier programa: `stdin, stdout y stderr`

Manejo de archivos en C – Funciones de alto nivel

Abrir archivo

```
#include <stdio.h>

FILE *fopen(const char *filename, const char *mode);
```

- ▶ **filename**: nombre del archivo o dispositivo
- ▶ **mode**: indica abrir el archivo ("w", "r", "w+", "r+", "a", "a+b", etc.)

Devuelve un puntero (no nulo) *FILE si tuvo éxito, o NULL si falló.

Manejo de archivos en C – Funciones de alto nivel

Abrir archivo

```
#include <stdio.h>

FILE *fopen(const char *filename, const char *mode);
```

- ▶ **filename**: nombre del archivo o dispositivo
- ▶ **mode**: indica abrir el archivo ("w", "r", "w+", "r+", "a", "a+b", etc.)

Devuelve un puntero (no nulo) *FILE si tuvo éxito, o NULL si falló.

Cerrar archivo

```
#include <stdio.h>

int fclose(FILE *stream);
```

- ▶ **sream**: flujo de datos a cerrar

Manejo de archivos en C – Funciones de alto nivel

Leer archivo

```
size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream);
```

Escribir archivo

```
size_t fwrite(const void *ptr, size_t size, size_t nitems, FILE *stream);
```

- ▶ **ptr:** Puntero al buffer de datos para la lectura/escritura
 - ▶ **size:** Tamaño del registro a transferir
 - ▶ **nitems:** Cantidad de registros a transferir
-

Manejo de archivos en C – Funciones de alto nivel

Leer archivo

```
size_t fread(void *ptr, size_t size, size_t nitems, FILE *stream);
```

Escribir archivo

```
size_t fwrite(const void *ptr, size_t size, size_t nitems, FILE *stream);
```

- ▶ **ptr**: Puntero al buffer de datos para la lectura/escritura
 - ▶ **size**: Tamaño del registro a transferir
 - ▶ **nitems**: Cantidad de registros a transferir
-

Leer cadena desde un archivo

```
ssize_t getline(char **lineptr, size_t *n, FILE *stream);
```

- ▶ **lineptr**: Buffer para almacenar la cadena
- ▶ **n**: Longitud de la cadena leída

Manejo de archivos en C – Funciones de alto nivel

Flush

```
int fflush (FILE *stream);
```

Fuerza que los datos del buffer sean efectivamente enviados.

Seek

```
int fseek (FILE *stream, long int offset, int whence);
```

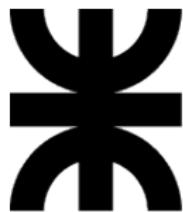
Fija el puntero del **stream** para la próxima transferencia.

- ▶ **offset**: posición medida en bytes
- ▶ **whence**: puede ser
 - 1. SEEK_SET: posición. absoluta
 - 2. SEEK_CUR: posición. actual
 - 3. SEEK_END: relativo al final

Informática II

Comunicación serie y estándar RS-232

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2021 –

Introducción

En informática, la comunicación o transmisión de datos entre dispositivos se puede realizar de forma paralela o serie.

Introducción

En informática, la comunicación o transmisión de datos entre dispositivos se puede realizar de forma paralela o serie.

1. Comun. paralela: permite transmitir varios bits de forma simultanea.

Introducción

En informática, la comunicación o transmisión de datos entre dispositivos se puede realizar de forma paralela o serie.

1. Comun. paralela: permite transmitir varios bits de forma simultanea.
2. Comun. serie: se transmite un único bit a la vez de forma secuencial.

Introducción

En informática, la comunicación o transmisión de datos entre dispositivos se puede realizar de forma paralela o serie.

1. Comun. paralela: permite transmitir varios bits de forma simultanea.
2. Comun. serie: se transmite un único bit a la vez de forma secuencial.

La transmisión de datos permite comunicar equipos electrónicos tales como:

- ▶ PC (Computadoras Personales), PLC (Controladores Lógicos Programables), instrumentos de laboratorios (multímetros, oscilloscopios, etc.).
- ▶ Placas de desarrollo de sistemas embebidos (Arduino, Raspberry Pi, etc.).
- ▶ Componentes internos de una PC.

Comunicación serie y paralela

- ▶ Interfaces o buses de comunicación entre comp. internos de una PC son:
 - ISA:** Industry Standard Architecture.
 - ATA/IDE:** Advanced Technology Attachment/Integrated Drive Electronics.
 - PCI:** Peripheral Component Interconnect.

Comunicación serie y paralela

- ▶ Interfaces o buses de comunicación entre comp. internos de una PC son:
 - ISA:** Industry Standard Architecture.
 - ATA/IDE:** Advanced Technology Attachment/Integrated Drive Electronics.
 - PCI:** Peripheral Component Interconnect.
- ▶ Interfaces de comunicación externa en paralelo:
 - IEEE-1284:** Incluido en las primeras PC de IBM y luego estandarizado por el Institute of Electrical and Electronics Engineers (IEEE).
 - IEEE-488:** Desarrollado originalmente por Hewlett-Packard para conectar dispositivos de testeo y medición con una PC. Luego estandarizado por el IEEE.

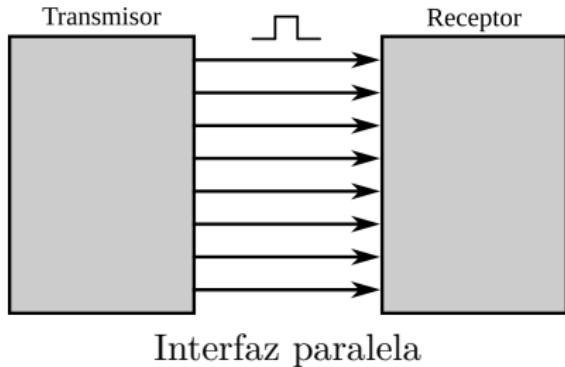
Comunicación serie y paralela

- ▶ Interfaces o buses de comunicación entre comp. internos de una PC son:
 - ISA:** Industry Standard Architecture.
 - ATA/IDE:** Advanced Technology Attachment/Integrated Drive Electronics.
 - PCI:** Peripheral Component Interconnect.
- ▶ Interfaces de comunicación externa en paralelo:
 - IEEE-1284:** Incluido en las primeras PC de IBM y luego estandarizado por el Institute of Electrical and Electronics Engineers (IEEE).
 - IEEE-488:** Desarrollado originalmente por Hewlett-Packard para conectar dispositivos de testeo y medición con una PC. Luego estandarizado por el IEEE.
- ▶ Interfaces de comunicación externa en serie: **RS-232/EIA-232** (a estudiar en detalle) y **RS-485/EIA-485** estándar de comunicaciones en bus diferencial de la capa física (modelo OSI –ISO/IEC 7498-1).

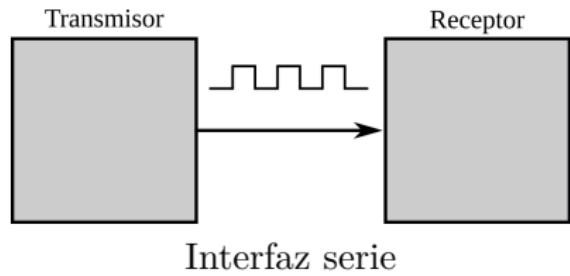
Comunicación serie y paralela

- ▶ Interfaces o buses de comunicación entre comp. internos de una PC son:
 - ISA:** Industry Standard Architecture.
 - ATA/IDE:** Advanced Technology Attachment/Integrated Drive Electronics.
 - PCI:** Peripheral Component Interconnect.
- ▶ Interfaces de comunicación externa en paralelo:
 - IEEE-1284:** Incluido en las primeras PC de IBM y luego estandarizado por el Institute of Electrical and Electronics Engineers (IEEE).
 - IEEE-488:** Desarrollado originalmente por Hewlett-Packard para conectar dispositivos de testeo y medición con una PC. Luego estandarizado por el IEEE.
- ▶ Interfaces de comunicación externa en serie: **RS-232/EIA-232** (a estudiar en detalle) y **RS-485/EIA-485** estándar de comunicaciones en bus diferencial de la capa física (modelo OSI –ISO/IEC 7498-1).
- ▶ Interfaces de com. en Sistemas Embebidos (SE): **SPI**, **I²C**, **1-Wire**, etc.

Comunicación serie y paralela



Interfaz paralela

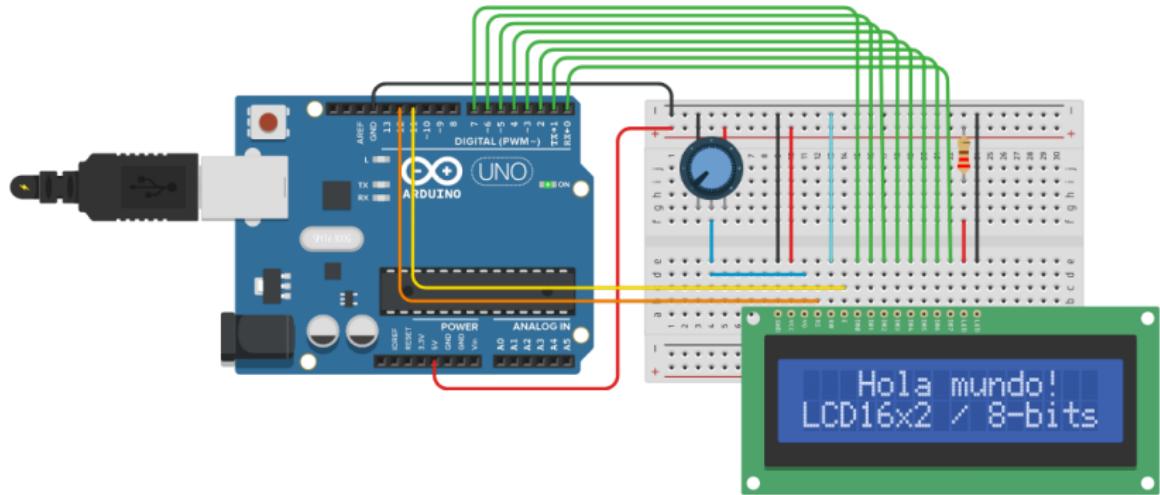


Interfaz serie

- ▶ En la comunicación paralela se necesitan tantas conexiones como bits se quieran transmitir.
- ▶ En la comunicación serie en un solo sentido se necesitan solo una conexión.

Comunicación serie y paralela

Ejemplo de comunicación paralela en SE

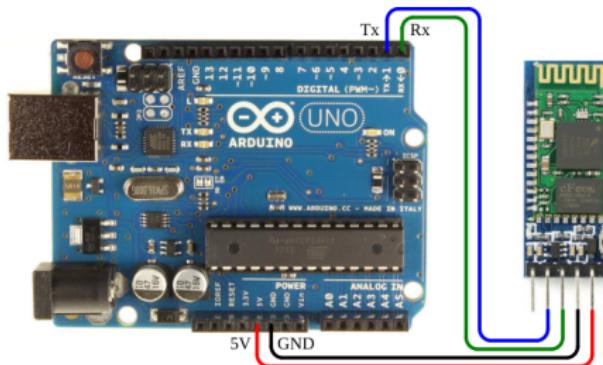


Conexión entre placa Arduino UNO y LCD de 16×2

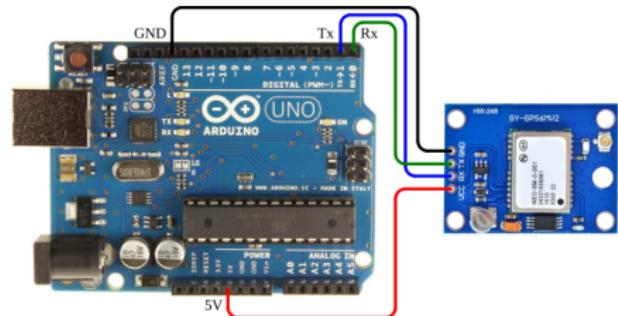
- ▶ Utiliza 8 bits de datos en paralelo.
- ▶ Bits de control: RS (Register Select), E (Enable) y R/W (Read/Write).

Comunicación serie y paralela

Ejemplo de comunicación serie en SE



Módulo Bluetooth HC-05



Módulo GPS NEO-6MV2

Conexión entre placa Arduino UNO con módulos Bluetooth y GPS a través de interfaz serie UART-TTL.

Transmisión serie síncrona vs. asíncrona

- ▶ Cuando dos dispositivos se comunican intercambian algún tipo de señal detectable que representa los datos. Sin comunicación no se enviará ninguna señal a través del canal de comunicación.

Transmisión serie síncrona vs. asíncrona

- ▶ Cuando dos dispositivos se comunican intercambian algún tipo de señal detectable que representa los datos. Sin comunicación no se enviará ninguna señal a través del canal de comunicación.
- ▶ Cuando haya datos para enviar, el dispositivo de envío comenzará a enviar señales. Debe entonces haber una forma para que el dispositivo de destino sepa cuándo comenzar a leer datos.

Transmisión serie síncrona vs. asíncrona

- ▶ Cuando dos dispositivos se comunican intercambian algún tipo de señal detectable que representa los datos. Sin comunicación no se enviará ninguna señal a través del canal de comunicación.
- ▶ Cuando haya datos para enviar, el dispositivo de envío comenzará a enviar señales. Debe entonces haber una forma para que el dispositivo de destino sepa cuándo comenzar a leer datos.
- ▶ Se debe establecer y mantener algún tipo de sincronización entre los dispositivos para que las señales se produzcan y detecten con precisión.

Transmisión serie síncrona vs. asíncrona

- ▶ Cuando dos dispositivos se comunican intercambian algún tipo de señal detectable que representa los datos. Sin comunicación no se enviará ninguna señal a través del canal de comunicación.
- ▶ Cuando haya datos para enviar, el dispositivo de envío comenzará a enviar señales. Debe entonces haber una forma para que el dispositivo de destino sepa cuándo comenzar a leer datos.
- ▶ Se debe establecer y mantener algún tipo de sincronización entre los dispositivos para que las señales se produzcan y detecten con precisión.

Comun. asíncrona: la sincronización se restablece con la transmisión de cada carácter mediante el uso de bits de inicio y parada.

Transmisión serie síncrona vs. asíncrona

- ▶ Cuando dos dispositivos se comunican intercambian algún tipo de señal detectable que representa los datos. Sin comunicación no se enviará ninguna señal a través del canal de comunicación.
- ▶ Cuando haya datos para enviar, el dispositivo de envío comenzará a enviar señales. Debe entonces haber una forma para que el dispositivo de destino sepa cuándo comenzar a leer datos.
- ▶ Se debe establecer y mantener algún tipo de sincronización entre los dispositivos para que las señales se produzcan y detecten con precisión.

Comun. asíncrona: la sincronización se restablece con la transmisión de cada carácter mediante el uso de bits de inicio y parada.

Comun. síncrona: la sincronización se mantiene mediante un bit especial en cada bloque de datos proporcionado por una señal adicional de reloj.

Transmisión serie síncrona vs. asíncrona

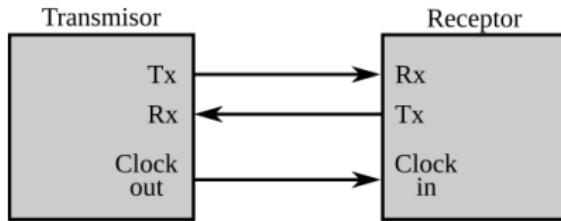
- ▶ Cuando dos dispositivos se comunican intercambian algún tipo de señal detectable que representa los datos. Sin comunicación no se enviará ninguna señal a través del canal de comunicación.
- ▶ Cuando haya datos para enviar, el dispositivo de envío comenzará a enviar señales. Debe entonces haber una forma para que el dispositivo de destino sepa cuándo comenzar a leer datos.
- ▶ Se debe establecer y mantener algún tipo de sincronización entre los dispositivos para que las señales se produzcan y detecten con precisión.

Comun. asíncrona: la sincronización se restablece con la transmisión de cada carácter mediante el uso de bits de inicio y parada.

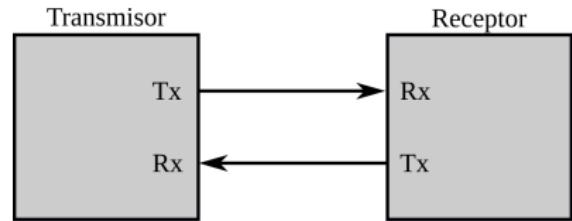
Comun. síncrona: la sincronización se mantiene mediante un bit especial en cada bloque de datos proporcionado por una señal adicional de reloj.

Debido a la menor cantidad de bits utilizados para mantener la sincronización, la comunicación síncrona resulta más eficiente.

Transmisión serie síncrona vs. asíncrona

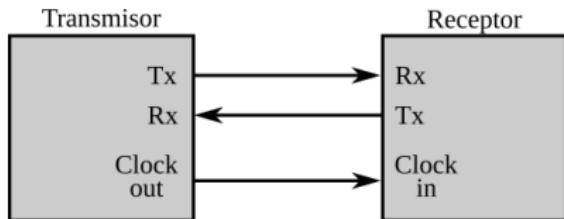


Comunicación síncrona

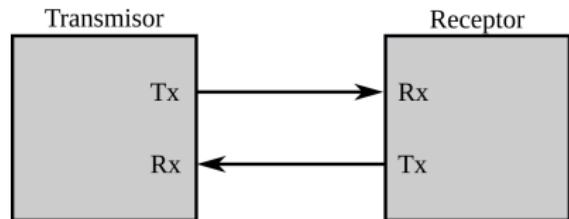


Comunicación asíncrona

Transmisión serie síncrona vs. asíncrona



Comunicación síncrona



Comunicación asíncrona

- ▶ En ambos casos, los dos dispositivos pueden transmitir y recibir información.
- ▶ En la comunicación síncrona además de las señales de transmisión y recepción de datos se incluye también una señal de reloj o clock.
- ▶ En la comunicación asíncrona la señal de reloj no es necesaria.

Transmisión serie Half-Duplex y Full-Duplex

- ▶ Las sesiones de comunicación de datos son en general de naturaleza bidireccional.
- ▶ Incluso si el objetivo de la comunicación es enviar un archivo desde el remitente al destino.
- ▶ Generalmente alguna comunicación debe ir desde el destino de regreso al remitente.

Transmisión serie Half-Duplex y Full-Duplex

- ▶ Las sesiones de comunicación de datos son en general de naturaleza bidireccional.
- ▶ Incluso si el objetivo de la comunicación es enviar un archivo desde el remitente al destino.
- ▶ Generalmente alguna comunicación debe ir desde el destino de regreso al remitente.

Comunicación Full-Duplex

Ambos dispositivos pueden transmitir al mismo tiempo (análogo a una conversación en persona).

Transmisión serie Half-Duplex y Full-Duplex

- ▶ Las sesiones de comunicación de datos son en general de naturaleza bidireccional.
- ▶ Incluso si el objetivo de la comunicación es enviar un archivo desde el remitente al destino.
- ▶ Generalmente alguna comunicación debe ir desde el destino de regreso al remitente.

Comunicación Full-Duplex

Ambos dispositivos pueden transmitir al mismo tiempo (análogo a una conversación en persona).

Comunicación Half-Duplex

Uno de los dispositivos puede oír o hablar en un momento determinado (similar a una conversación con walkie-talkies donde solo puede hablar aquel que presione el botón).

Transmisión serie Half-Duplex y Full-Duplex

- ▶ Las sesiones de comunicación de datos son en general de naturaleza bidireccional.
- ▶ Incluso si el objetivo de la comunicación es enviar un archivo desde el remitente al destino.
- ▶ Generalmente alguna comunicación debe ir desde el destino de regreso al remitente.

Comunicación Full-Duplex

Ambos dispositivos pueden transmitir al mismo tiempo (análogo a una conversación en persona).

Comunicación Half-Duplex

Uno de los dispositivos puede oír o hablar en un momento determinado (similar a una conversación con walkie-talkies donde solo puede hablar aquel que presione el botón).

Cuando la comunicación es unidireccional, se conoce con el nombre de Simplex.

El estándar RS-232

- La comunicación serie se utiliza ampliamente en la industria debido principalmente a su relativa simplicidad en el hardware.

El estándar RS-232

- ▶ La comunicación serie se utiliza ampliamente en la industria debido principalmente a su relativa simplicidad en el hardware.
- ▶ Uno de los estándares de comunicación en serie más utilizado es el EIA/TIA-232-E.

El estándar RS-232

- ▶ La comunicación serie se utiliza ampliamente en la industria debido principalmente a su relativa simplicidad en el hardware.
- ▶ Uno de los estándares de comunicación en serie más utilizado es el EIA/TIA-232-E.
- ▶ El estándar se refiere a la comunicación de datos entre un sistema host (DTE, Data Terminal Equipment) y un sistema periférico (DCE, Data Circuit-Terminating Equipment o Data Communication Equipment).

El estándar RS-232

- ▶ La comunicación serie se utiliza ampliamente en la industria debido principalmente a su relativa simplicidad en el hardware.
 - ▶ Uno de los estándares de comunicación en serie más utilizado es el EIA/TIA-232-E.
-
- ▶ El estándar se refiere a la comunicación de datos entre un sistema host (DTE, Data Terminal Equipment) y un sistema periférico (DCE, Data Circuit-Terminating Equipment o Data Communication Equipment).
 - ▶ Gran parte de la terminología RS-232 refleja su origen como estándar para las comun. entre un terminal de computadora (PC) y un módem externo.

El estándar RS-232

- ▶ La comunicación serie se utiliza ampliamente en la industria debido principalmente a su relativa simplicidad en el hardware.
 - ▶ Uno de los estándares de comunicación en serie más utilizado es el EIA/TIA-232-E.
-
- ▶ El estándar se refiere a la comunicación de datos entre un sistema host (DTE, Data Terminal Equipment) y un sistema periférico (DCE, Data Circuit-Terminating Equipment o Data Communication Equipment).
 - ▶ Gran parte de la terminología RS-232 refleja su origen como estándar para las comun. entre un terminal de computadora (PC) y un módem externo.
 - ▶ En la actualidad resulta más frecuente el uso del puerto RS-232 para conectar una PC a un sistema embebido o bien dos sistemas embebidos entre sí (UART-TTL).

El estándar RS-232

El EIA-TIA-232 es un estándar “completo” que garantiza la compatibilidad entre el host y los sistemas periféricos mediante la especificación de:

El estándar RS-232

El EIA-TIA-232 es un estándar “completo” que garantiza la compatibilidad entre el host y los sistemas periféricos mediante la especificación de:

1. señales y niveles comunes de voltaje (caract. eléctricas),

El estándar RS-232

El EIA-TIA-232 es un estándar “completo” que garantiza la compatibilidad entre el host y los sistemas periféricos mediante la especificación de:

1. señales y niveles comunes de voltaje (caract. eléctricas),
2. configuraciones de pines y cableado (caract. mecánicas) e

El estándar RS-232

El EIA-TIA-232 es un estándar “completo” que garantiza la compatibilidad entre el host y los sistemas periféricos mediante la especificación de:

1. señales y niveles comunes de voltaje (caract. eléctricas),
2. configuraciones de pines y cableado (caract. mecánicas) e
3. información de control mínima entre el host y los sistemas periféricos (caract. funcionales).

El estándar RS-232

El EIA-TIA-232 es un estándar “completo” que garantiza la compatibilidad entre el host y los sistemas periféricos mediante la especificación de:

1. señales y niveles comunes de voltaje (caract. eléctricas),
2. configuraciones de pines y cableado (caract. mecánicas) e
3. información de control mínima entre el host y los sistemas periféricos (caract. funcionales).

El EIA/TIA-232-E fue desarrollado por:

Electronic Industry Association y la *Telecommunications Industry Association*, se conoce más popularmente simplemente como **RS-232**, donde “RS” significa *Recommended Standard*.

El estándar RS-232 – Características eléctricas

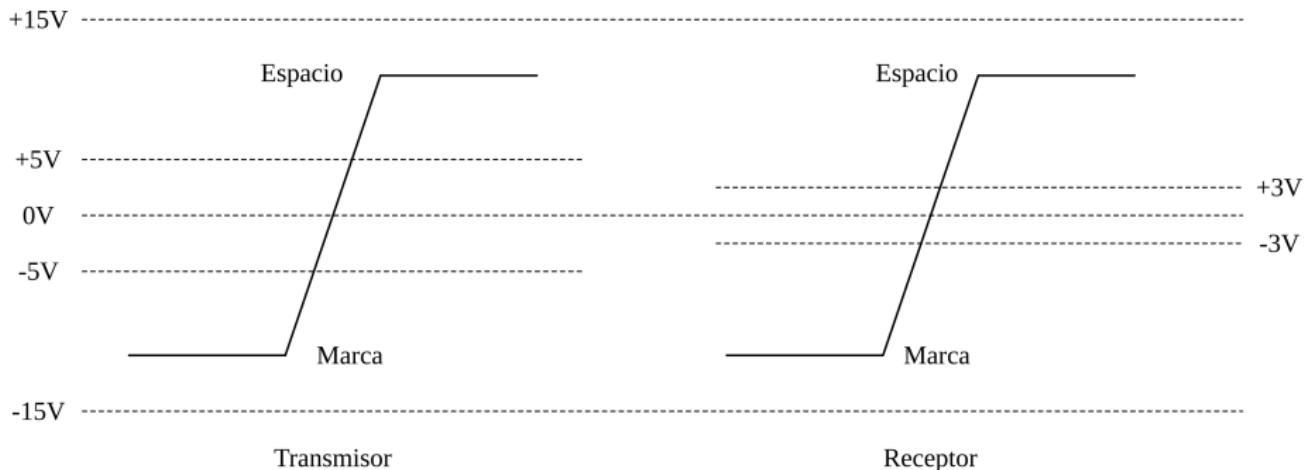
Especificaciones eléctricas del puerto serie o RS-232:

- ▶ Un cero lógico o *espacio* entre +5 y +15V (transmisor).
- ▶ Un uno lógico o *marca* entre -5 y -15V (transmisor).
- ▶ Los niveles de tensión de entrada tienen un margen de ruido de 2V.
- ▶ Los valores entre +3V y -3V representan estados indefinidos.

El estándar RS-232 – Características eléctricas

Especificaciones eléctricas del puerto serie o RS-232:

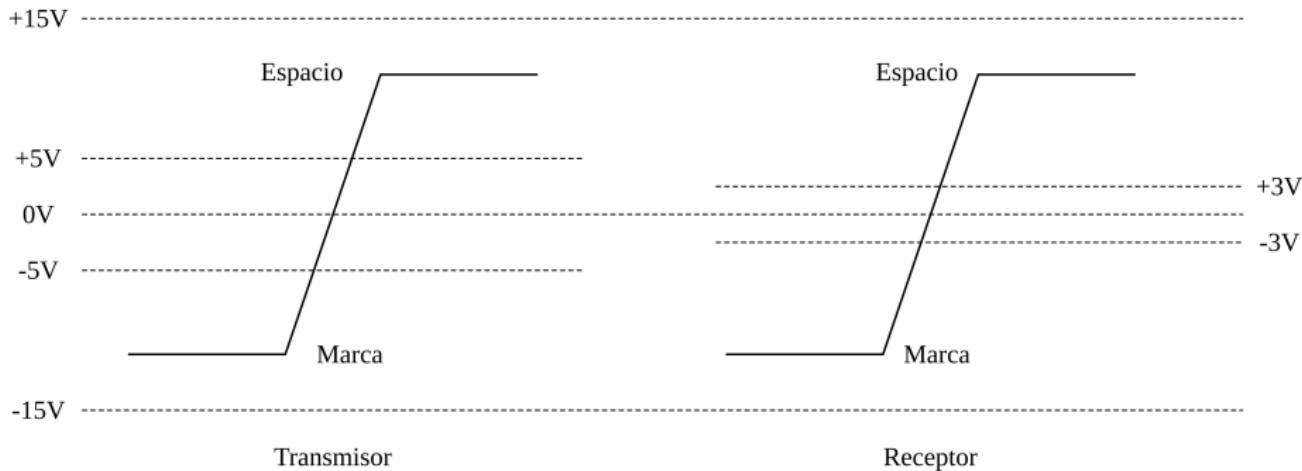
- ▶ Un cero lógico o *espacio* entre +5 y +15V (transmisor).
- ▶ Un uno lógico o *marca* entre -5 y -15V (transmisor).
- ▶ Los niveles de tensión de entrada tienen un margen de ruido de 2V.
- ▶ Los valores entre +3V y -3V representan estados indefinidos.



El estándar RS-232 – Características eléctricas

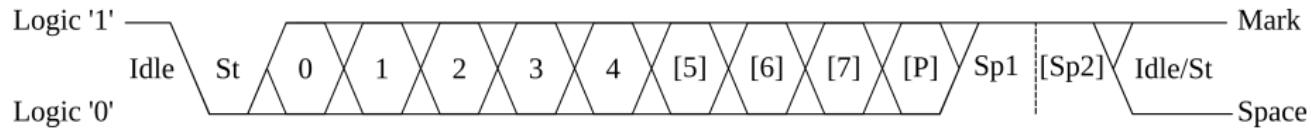
Especificaciones eléctricas del puerto serie o RS-232:

- ▶ Un cero lógico o *espacio* entre +5 y +15V (transmisor).
- ▶ Un uno lógico o *marca* entre -5 y -15V (transmisor).
- ▶ Los niveles de tensión de entrada tienen un margen de ruido de 2V.
- ▶ Los valores entre +3V y -3V representan estados indefinidos.

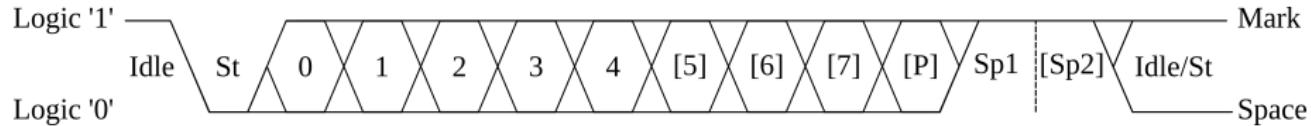


(Otros parámetros son: corriente de cortocircuito, capacitancia máxima de línea, tasa de cambio del niveles de las señales, impedancia de la línea, etc.)

El estándar RS-232 – Trama de comunicación

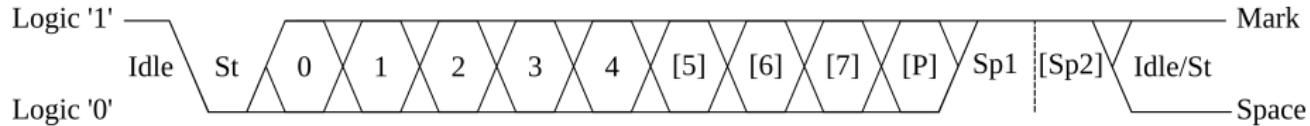


El estándar RS-232 – Trama de comunicación



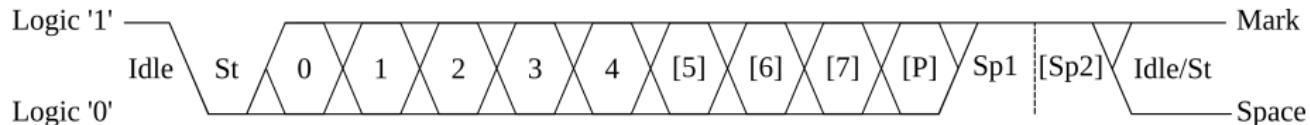
- ▶ Idle: Ocioso, no hay transferencia de datos (TxD y RxD) [Logic 1]

El estándar RS-232 – Trama de comunicación



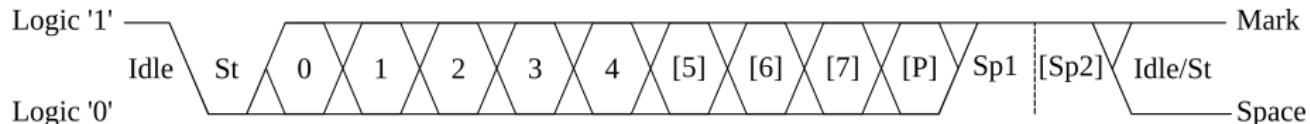
- ▶ Idle: Ocioso, no hay transferencia de datos (TxD y RxD) [Logic 1]
- ▶ St: Start bit (bit de arranque) [Logic 0]

El estándar RS-232 – Trama de comunicación



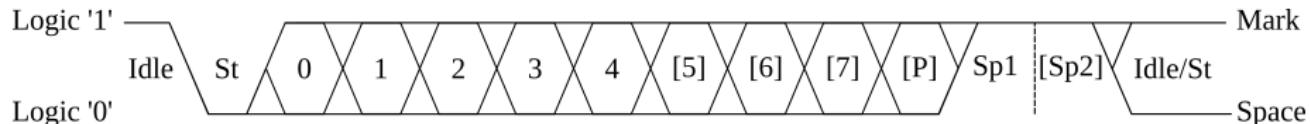
- ▶ Idle: Ocioso, no hay transferencia de datos (TxD y RxD) [Logic 1]
- ▶ St: Start bit (bit de arranque) [Logic 0]
- ▶ (n): Bits de datos

El estándar RS-232 – Trama de comunicación



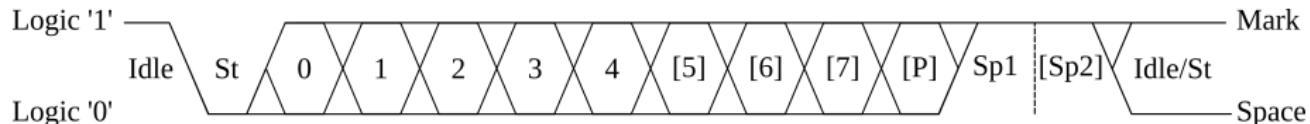
- ▶ Idle: Ocioso, no hay transferencia de datos (TxD y RxD) [Logic 1]
- ▶ St: Start bit (bit de arranque) [Logic 0]
- ▶ (n): Bits de datos
- ▶ P: Parity bit (bit de paridad)

El estándar RS-232 – Trama de comunicación



- ▶ Idle: Ocioso, no hay transferencia de datos (TxD y RxD) [Logic 1]
- ▶ St: Start bit (bit de arranque) [Logic 0]
- ▶ (n): Bits de datos
- ▶ P: Parity bit (bit de paridad)
- ▶ Sp: Stop bit (bit de parada) [Logic 1]

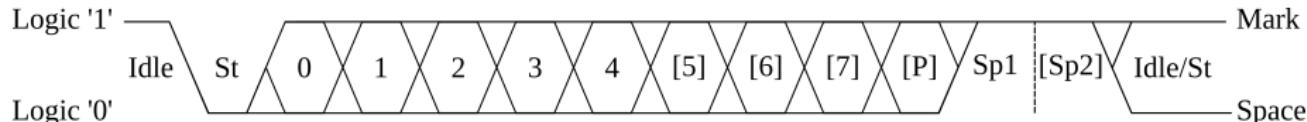
El estándar RS-232 – Trama de comunicación



- ▶ Idle: Ocioso, no hay transferencia de datos (TxD y RxD) [Logic 1]
- ▶ St: Start bit (bit de arranque) [Logic 0]
- ▶ (n): Bits de datos
- ▶ P: Parity bit (bit de paridad)
- ▶ Sp: Stop bit (bit de parada) [Logic 1]

El bit menos significativo (LSb) se envía primero.

El estándar RS-232 – Trama de comunicación



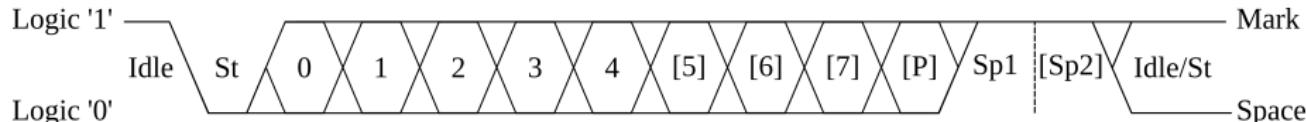
- ▶ Idle: Ocioso, no hay transferencia de datos (TxD y RxD) [Logic 1]
- ▶ St: Start bit (bit de arranque) [Logic 0]
- ▶ (n): Bits de datos
- ▶ P: Parity bit (bit de paridad)
- ▶ Sp: Stop bit (bit de parada) [Logic 1]

El bit menos significativo (LSb) se envía primero.

Algunas alternativas de tramas

- ▶ 8N1: 8 bits de datos, sin (N) bit de paridad, y 1 bit de stop.
- ▶ 5N2: 5 bits de datos, sin bit de paridad, y 2 bits de stop.

El estándar RS-232 – Trama de comunicación



- ▶ Idle: Ocioso, no hay transferencia de datos (TxD y RxD) [Logic 1]
- ▶ St: Start bit (bit de arranque) [Logic 0]
- ▶ (n): Bits de datos
- ▶ P: Parity bit (bit de paridad)
- ▶ Sp: Stop bit (bit de parada) [Logic 1]

El bit menos significativo (LSb) se envía primero.

Algunas alternativas de tramas

- ▶ 8N1: 8 bits de datos, sin (N) bit de paridad, y 1 bit de stop.
- ▶ 5N2: 5 bits de datos, sin bit de paridad, y 2 bits de stop.

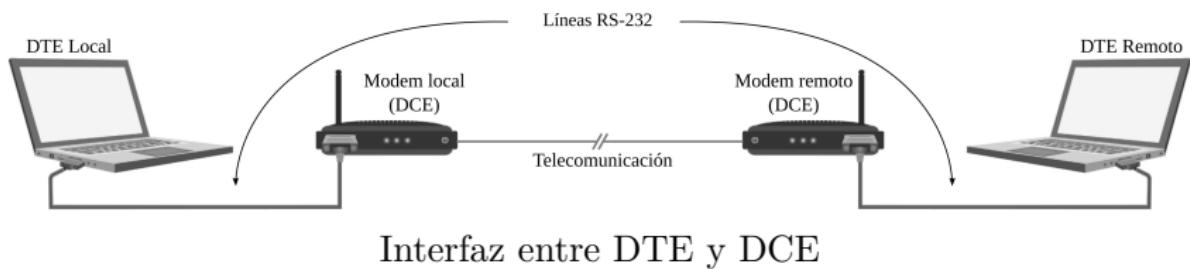
¿Qué duración tiene cada bit?

El estándar RS-232 – Características funcionales

- ▶ El RS-232 define la función de las diferentes señales que se utilizan en la interfaz.
- ▶ Pocas aplicaciones requieren todas estas señales definidas en el estándar.
- ▶ De hecho, aplicaciones como la comunicación por módem requiere solo nueve señales (dos señales de datos, seis señales de control y referencia).

El estándar RS-232 – Características funcionales

- ▶ El RS-232 define la función de las diferentes señales que se utilizan en la interfaz.
- ▶ Pocas aplicaciones requieren todas estas señales definidas en el estándar.
- ▶ De hecho, aplicaciones como la comunicación por módem requiere solo nueve señales (dos señales de datos, seis señales de control y referencia).



El estándar RS-232 – Características funcionales

| DB-9 (DB-25) Pin No. | Abbrev. | Full name | |
|-----------------------------|----------------|---------------------|--|
| 3 (2) | TD | Transmit Data | |
| 2 (3) | RD | Receive Data | |
| 7 (4) | RTS | Request To Send | |
| 8 (5) | CTS | Clear To Send | |
| 6 (6) | DSR | Data Set Ready | |
| 5 (7) | SG | Signal Ground | |
| 1 (8) | DCD | Data Carrier Detect | |
| 4 (20) | DTR | Data Terminal Ready | |
| 9 (22) | RI | Ring Indicator | |

El estándar RS-232 – Características funcionales

| DB-9 (DB-25) Pin No. | Abbrev. | Full name | PC — Perif. |
|----------------------|---------|---------------------|-------------|
| 3 (2) | TD | Transmit Data | DTE → DCE |
| 2 (3) | RD | Receive Data | |
| 7 (4) | RTS | Request To Send | |
| 8 (5) | CTS | Clear To Send | |
| 6 (6) | DSR | Data Set Ready | |
| 5 (7) | SG | Signal Ground | |
| 1 (8) | DCD | Data Carrier Detect | |
| 4 (20) | DTR | Data Terminal Ready | |
| 9 (22) | RI | Ring Indicator | |

TD: Salida de datos seriales, TxD

El estándar RS-232 – Características funcionales

| DB-9 (DB-25) Pin No. | Abbrev. | Full name | PC — Perif. |
|----------------------|---------|---------------------|-------------|
| 3 (2) | TD | Transmit Data | DTE → DCE |
| 2 (3) | RD | Receive Data | DTE ← DCE |
| 7 (4) | RTS | Request To Send | |
| 8 (5) | CTS | Clear To Send | |
| 6 (6) | DSR | Data Set Ready | |
| 5 (7) | SG | Signal Ground | |
| 1 (8) | DCD | Data Carrier Detect | |
| 4 (20) | DTR | Data Terminal Ready | |
| 9 (22) | RI | Ring Indicator | |

RD: Entrada de datos seriales, RxD

El estándar RS-232 – Características funcionales

| DB-9 (DB-25) Pin No. | Abbrev. | Full name | PC — Perif. |
|----------------------|---------|---------------------|-------------|
| 3 (2) | TD | Transmit Data | DTE → DCE |
| 2 (3) | RD | Receive Data | DTE ← DCE |
| 7 (4) | RTS | Request To Send | DTE → DCE |
| 8 (5) | CTS | Clear To Send | |
| 6 (6) | DSR | Data Set Ready | |
| 5 (7) | SG | Signal Ground | |
| 1 (8) | DCD | Data Carrier Detect | |
| 4 (20) | DTR | Data Terminal Ready | |
| 9 (22) | RI | Ring Indicator | |

RTS: Le indica al módem que la UART está lista para comunicarse

El estándar RS-232 – Características funcionales

| DB-9 (DB-25) Pin No. | Abbrev. | Full name | PC — Perif. |
|-----------------------------|----------------|---------------------|--------------------|
| 3 (2) | TD | Transmit Data | DTE → DCE |
| 2 (3) | RD | Receive Data | DTE ← DCE |
| 7 (4) | RTS | Request To Send | DTE → DCE |
| 8 (5) | CTS | Clear To Send | DTE ← DCE |
| 6 (6) | DSR | Data Set Ready | |
| 5 (7) | SG | Signal Ground | |
| 1 (8) | DCD | Data Carrier Detect | |
| 4 (20) | DTR | Data Terminal Ready | |
| 9 (22) | RI | Ring Indicator | |

CTS: Indica que el módem está listo para comunicarse

El estándar RS-232 – Características funcionales

| DB-9 (DB-25) Pin No. | Abbrev. | Full name | PC — Perif. |
|----------------------|---------|---------------------|-------------|
| 3 (2) | TD | Transmit Data | DTE → DCE |
| 2 (3) | RD | Receive Data | DTE ← DCE |
| 7 (4) | RTS | Request To Send | DTE → DCE |
| 8 (5) | CTS | Clear To Send | DTE ← DCE |
| 6 (6) | DSR | Data Set Ready | DTE ← DCE |
| 5 (7) | SG | Signal Ground | |
| 1 (8) | DCD | Data Carrier Detect | |
| 4 (20) | DTR | Data Terminal Ready | |
| 9 (22) | RI | Ring Indicator | |

DSR: Le indica a la UART que el módem está listo para establecer una conexión

El estándar RS-232 – Características funcionales

| DB-9 (DB-25) Pin No. | Abbrev. | Full name | PC — Perif. |
|----------------------|---------|---------------------|-------------|
| 3 (2) | TD | Transmit Data | DTE → DCE |
| 2 (3) | RD | Receive Data | DTE ← DCE |
| 7 (4) | RTS | Request To Send | DTE → DCE |
| 8 (5) | CTS | Clear To Send | DTE ← DCE |
| 6 (6) | DSR | Data Set Ready | DTE ← DCE |
| 5 (7) | SG | Signal Ground | DTE — DCE |
| 1 (8) | DCD | Data Carrier Detect | DTE ← DCE |
| 4 (20) | DTR | Data Terminal Ready | |
| 9 (22) | RI | Ring Indicator | |

DCD: Indica que el módem detecta “portadora”

El estándar RS-232 – Características funcionales

| DB-9 (DB-25) Pin No. | Abbrev. | Full name | PC — Perif. |
|----------------------|---------|---------------------|-------------|
| 3 (2) | TD | Transmit Data | DTE → DCE |
| 2 (3) | RD | Receive Data | DTE ← DCE |
| 7 (4) | RTS | Request To Send | DTE → DCE |
| 8 (5) | CTS | Clear To Send | DTE ← DCE |
| 6 (6) | DSR | Data Set Ready | DTE ← DCE |
| 5 (7) | SG | Signal Ground | DTE — DCE |
| 1 (8) | DCD | Data Carrier Detect | DTE ← DCE |
| 4 (20) | DTR | Data Terminal Ready | DTE → DCE |
| 9 (22) | RI | Ring Indicator | |

DTR: Opuesto a DSR. Le indica al módem que la UART está lista para establecer conexión

El estándar RS-232 – Características funcionales

| DB-9 (DB-25) Pin No. | Abbrev. | Full name | PC — Perif. |
|----------------------|---------|---------------------|-------------|
| 3 (2) | TD | Transmit Data | DTE → DCE |
| 2 (3) | RD | Receive Data | DTE ← DCE |
| 7 (4) | RTS | Request To Send | DTE → DCE |
| 8 (5) | CTS | Clear To Send | DTE ← DCE |
| 6 (6) | DSR | Data Set Ready | DTE ← DCE |
| 5 (7) | SG | Signal Ground | DTE — DCE |
| 1 (8) | DCD | Data Carrier Detect | DTE ← DCE |
| 4 (20) | DTR | Data Terminal Ready | DTE → DCE |
| 9 (22) | RI | Ring Indicator | DTE ← DCE |

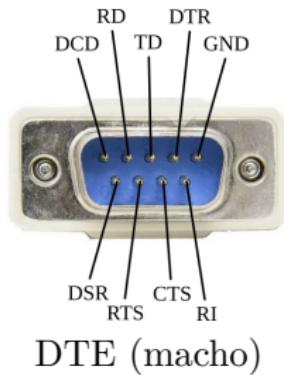
RI: Se activa ante la presencia de llamada

El estándar RS-232 – Características mecánicas

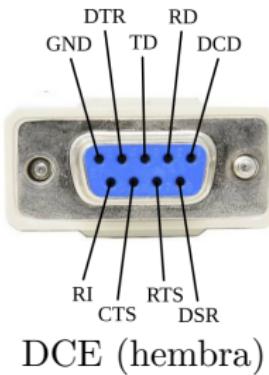
- ▶ El RS-232 especifica un conector de 25 pines (tamaño mínimo para albergar todas las señales definidas en la parte funcional del estándar).
- ▶ El conector para el **DTE** tiene una carcasa hembra con pines de conexión macho, y para el **DCE** es macho para la carcasa y hembra para los pines.
- ▶ El conector más popular es el DB9 (9 pines).

El estándar RS-232 – Características mecánicas

- ▶ El RS-232 especifica un conector de 25 pines (tamaño mínimo para albergar todas las señales definidas en la parte funcional del estándar).
- ▶ El conector para el **DTE** tiene una carcasa hembra con pines de conexión macho, y para el **DCE** es macho para la carcasa y hembra para los pines.
- ▶ El conector más popular es el DB9 (9 pines).



DTE (macho)



DCE (hembra)

La UART

- ▶ Las señales necesarias para la comunicación en serie son generadas y recibidas por un circuito integrado (CI) conocido como UART (Universal Asynchronous Receiver/Transmitter).
- ▶ Este circuito integrado hace de conversor paralelo/serie para “serializar/des-serializar” los datos.

La UART

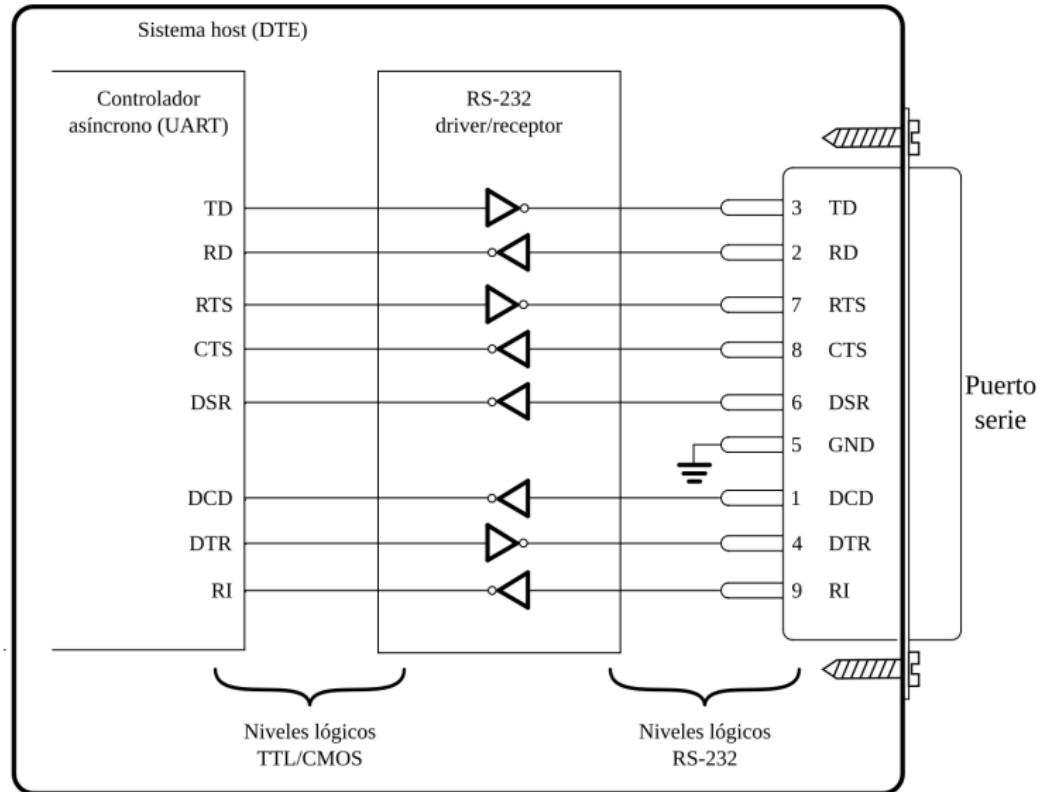
- ▶ Las señales necesarias para la comunicación en serie son generadas y recibidas por un circuito integrado (CI) conocido como UART (Universal Asynchronous Receiver/Transmitter).
 - ▶ Este circuito integrado hace de conversor paralelo/serie para “serializar/des-serializar” los datos.
-
- ▶ La UART del sistema host genera los bits de inicio y parada para indicar al sistema periférico cuándo se inicia y termina la comunicación.
 - ▶ Hay otro CI que hace de driver de entrada/salida del RS-232 y adapta los nivel de tensión necesarios entre la UART y el RS-232.

La UART

- ▶ Las señales necesarias para la comunicación en serie son generadas y recibidas por un circuito integrado (CI) conocido como UART (Universal Asynchronous Receiver/Transmitter).
- ▶ Este circuito integrado hace de conversor paralelo/serie para “serializar/des-serializar” los datos.
- ▶ La UART del sistema host genera los bits de inicio y parada para indicar al sistema periférico cuándo se inicia y termina la comunicación.
- ▶ Hay otro CI que hace de driver de entrada/salida del RS-232 y adapta los nivel de tensión necesarios entre la UART y el RS-232.



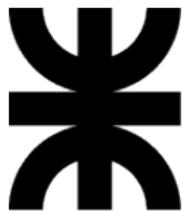
La UART



Informática II

Puerto serie en la PC

Gonzalo F. Perez Paina



Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2021 –

Introducción

- ▶ El USB (Universal Serial Bus) es el puerto de comunicaciones más utilizado para la conexión de periféricos en la actualidad.
- ▶ Existen diferentes modelos de adaptadores USB-serie.
- ▶ Cuando se habla de puerto serie se hace referencia la [comunicación serial asíncrona](#) tipo RS-232 o UART-TTL.

Introducción

- ▶ El USB (Universal Serial Bus) es el puerto de comunicaciones más utilizado para la conexión de periféricos en la actualidad.
- ▶ Existen diferentes modelos de adaptadores USB-serie.
- ▶ Cuando se habla de puerto serie se hace referencia la [comunicación serial asíncrona](#) tipo RS-232 o UART-TTL.



- ▶ Adaptador USB-RS232.
- ▶ Adaptador USB-UART con/sin cable y con/sin bits de control (handshake).

Puerto serie en la PC – Acceso a dispositivo

En GNU/Linux se tiene acceso a los dispositivos (puertos series) mediante archivos de dispositivos, los cuales se encuentran en el directorio `/dev`.

Puerto serie en la PC – Acceso a dispositivo

En GNU/Linux se tiene acceso a los dispositivos (puertos series) mediante archivos de dispositivos, los cuales se encuentran en el directorio `/dev`.

Ejemplo de archivo de dispositivos

```
crw-rw---- 1 root dialout 188,  0 jul 20 14:02 /dev/ttyUSB0
crw-rw-rw- 1 root dialout 166,  0 jul 20 15:57 /dev/ttyACM0
brw-rw---- 1 root disk      8,  0 jul 20 14:02 /dev/sda
```

('c': dispositivos de caracteres – 'b' dispositivos de bloques)

Puerto serie en la PC – Acceso a dispositivo

En GNU/Linux se tiene acceso a los dispositivos (puertos series) mediante archivos de dispositivos, los cuales se encuentran en el directorio `/dev`.

Ejemplo de archivo de dispositivos

```
crw-rw---- 1 root dialout 188,  0 jul 20 14:02 /dev/ttyUSB0
crw-rw-rw- 1 root dialout 166,  0 jul 20 15:57 /dev/ttyACM0
brw-rw---- 1 root disk      8,   0 jul 20 14:02 /dev/sda
```

('c': dispositivos de caracteres – 'b' dispositivos de bloques)

- ▶ Los archivos de dispositivos para los puertos serie suelen ser: `/dev/ttys0`, `/dev/ttys1`, `/dev/ttyUSB0`, `/dev/ttyACM0`, etc.

Puerto serie en la PC – Acceso a dispositivo

En GNU/Linux se tiene acceso a los dispositivos (puertos series) mediante archivos de dispositivos, los cuales se encuentran en el directorio `/dev`.

Ejemplo de archivo de dispositivos

```
crw-rw---- 1 root dialout 188,  0 jul 20 14:02 /dev/ttyUSB0
crw-rw-rw- 1 root dialout 166,  0 jul 20 15:57 /dev/ttyACM0
brw-rw---- 1 root disk      8,  0 jul 20 14:02 /dev/sda
```

('c': dispositivos de caracteres – 'b' dispositivos de bloques)

- ▶ Los archivos de dispositivos para los puertos serie suelen ser: `/dev/ttys0`, `/dev/ttys1`, `/dev/ttyUSB0`, `/dev/ttyACM0`, etc.
- ▶ Para ver el archivo de dispositivo creado al conectar el hardware (Arduino o adaptador USB-serie) utilizando el comando `dmesg`.

Puerto serie en la PC – Acceso a dispositivo

Ejemplo de salida de `dmesg` con placa Arduino UNO

```
usb 3-1: new full-speed USB device number 15 using xhci_hcd
usb 3-1: New USB device found, idVendor=1a86, idProduct=7523
usb 3-1: New USB device strings: Mfr=0, Product=2, SerialNumber=0
usb 3-1: Product: USB2.0-Serial
ch341 3-1:1.0: ch341-uart converter detected
usb 3-1: ch341-uart converter now attached to ttyUSB0
```

Puerto serie en la PC – Acceso a dispositivo

Ejemplo de salida de `dmesg` con placa Arduino UNO

```
usb 3-1: new full-speed USB device number 15 using xhci_hcd
usb 3-1: New USB device found, idVendor=1a86, idProduct=7523
usb 3-1: New USB device strings: Mfr=0, Product=2, SerialNumber=0
usb 3-1: Product: USB2.0-Serial
ch341 3-1:1.0: ch341-uart converter detected
usb 3-1: ch341-uart converter now attached to ttyUSB0
```

Ejemplo de salida de `dmesg` con adaptador USB-serie

```
usb 3-2: new full-speed USB device number 17 using xhci_hcd
usb 3-2: New USB device found, idVendor=067b, idProduct=2303
usb 3-2: New USB device strings: Mfr=1, Product=2, SerialNumber=0
usb 3-2: Product: USB-Serial Controller
usb 3-2: Manufacturer: Prolific Technology Inc.
pl2303 3-2:1.0: pl2303 converter detected
usb 3-2: pl2303 converter now attached to ttyUSB1
```

Puerto serie en la PC – Acceso a dispositivo

- ▶ Para modificar y ver la configuración actual de un puerto serie (archivo de dispositivo en `/dev`) se puede utilizar el comando `stty`.
 - ▶ Para ver la velocidad de comunicación actual

```
> stty -F /dev/ttyUSB0  
speed 9600 baud; line = 0;
```

- ▶ Para modificar la velocidad de comunicación

```
> stty -F /dev/ttyUSB1 115200  
> stty -F /dev/ttyUSB1  
speed 115200 baud; line = 0;
```

Puerto serie en la PC – Acceso a dispositivo

- ▶ Para modificar y ver la configuración actual de un puerto serie (archivo de dispositivo en `/dev`) se puede utilizar el comando `stty`.
 - ▶ Para ver la velocidad de comunicación actual

```
> stty -F /dev/ttyUSB0  
speed 9600 baud; line = 0;
```

- ▶ Para modificar la velocidad de comunicación

```
> stty -F /dev/ttyUSB1 115200  
> stty -F /dev/ttyUSB1  
speed 115200 baud; line = 0;
```

- ▶ Ejemplo con Arduino:
 - ▶ Abrir IDE Arduino y grabar el ejemplo `AnalogReadSerial`
 - ▶ Ver valores enviados desde el Monitor Serial (IDE)
 - ▶ Ver valores enviados desde `cutecom`

Puerto serie en la PC – Acceso a dispositivo

- ▶ Para modificar y ver la configuración actual de un puerto serie (archivo de dispositivo en `/dev`) se puede utilizar el comando `stty`.
 - ▶ Para ver la velocidad de comunicación actual

```
> stty -F /dev/ttyUSB0  
speed 9600 baud; line = 0;
```

- ▶ Para modificar la velocidad de comunicación

```
> stty -F /dev/ttyUSB1 115200  
> stty -F /dev/ttyUSB1  
speed 115200 baud; line = 0;
```

- ▶ Ejemplo con Arduino:
 - ▶ Abrir IDE Arduino y grabar el ejemplo `AnalogReadSerial`
 - ▶ Ver valores enviados desde el Monitor Serial (IDE)
 - ▶ Ver valores enviados desde `cutecom`
- ▶ Algunas terminales: `screen`, `cutecom`, `minicom`, `PuTTY`, etc.

Puerto serie en la PC – Acceso a dispositivo

Puerto serie en la PC – Acceso a dispositivo

- ▶ Uso de **socat** para generación de puertos “virtuales”
 > `socat PTY,link=/tmp/ttyS0 PTY,link=/tmp/ttyS1`
- ▶ Mostrar en terminal lo recibido por el puerto serie
 > `cat /tmp/ttyS1`
- ▶ Enviar una cadeba a un puerto serie
 > `cat /tmp/ttyS0`
- ▶ Enviar una cadena desde **cutecom**

Programación – Abrir y cerrar el puerto serie

```
1 #include <stdio.h> /* Standard input/output definitions */
2 #include <fcntl.h> /* File control definitions */
3 #include <unistd.h> /* UNIX standard function definitions */
4
5 int main(void) {
6     int fd; /* File descriptor for the port */
7
8     fd = open("/dev/ttyUSBO", O_RDWR | O_NOCTTY | O_NDELAY);
9
10    if(fd == -1)
11        /* ERROR */
12
13    close(fd);
14    return 0;
15 }
```

Flags:

- ▶ O_RDWR: Lectura/Escritura
- ▶ O_NOCTTY: Para que no sea una terminal de control (ps -a)
- ▶ O_NDELAY: Ignora la señal DCD (o el proceso duerme hasta activarse DCD)

Programación – Escribir y leer datos

```
1 #include <stdio.h> /* Standard input/output definitions */
2 #include <fcntl.h> /* File control definitions */
3 #include <unistd.h> /* UNIX standard function definitions */
4
5 int main(void) {
6     int n, fd;
7     char* data;
8
9     fd = open("/dev/ttyUSB0", O_RDWR | O_NOCTTY | O_NDELAY);
10    if(fd == -1) {} /* ERROR */
11
12    /* Writing data to the port */
13    n = write(fd, "Hello\n", 6);
14    if(n < 0)
15        /* ERROR */
16
17    /* Reading data from the port */
18    /* n = read(fd, data, 4); */
19    /* if(n < 4) */
20
21    close(fd);
22    return 0;
23 }
```

Configuración del puerto serie (termios)

Terminales en sistemas POSIX (Portable Operating System Interface (UNIX))

- ▶ `termios.h`: estructura de control de terminal y funciones de control.
- ▶ Cambiar parámetros tales como velocidad de comunicación, tamaño trama, etc.

Las dos funciones más importantes son: `tcgetattr()`, `tcsetattr()`

Miembros de la `estructura termios`:

- ▶ `c_cflags`: Opciones de control
- ▶ `c_lflags`: Opciones de línea
- ▶ `c_iflags`: Opciones de entrada
- ▶ `c_oflags`: Opciones de salida
- ▶ `c_cc`: Caracteres de control
- ▶ `c_ispeed`: Baudrate de entrada (interfaz nueva)
- ▶ `c_ospeed`: Baudrate de salida (interfaz nueva)

Configuración del puerto serie (termios)

```
struct termios options;
/* Get the current options for the port */
tcgetattr(fd, &options);

/* Set the baud rates to 19200 */
cfsetispeed(&options, B19200);
cfsetospeed(&options, B19200);

/* Enable the receiver and set local mode */
options.c_cflag |= (CLOCAL | CREAD);

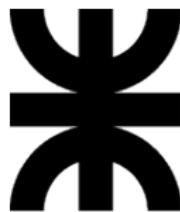
/* No parity 8N1 */
options.c_cflag &= ~PARENB; /* No parity */
options.c_cflag &= ~CSTOPB; /* 1 stop bit */
options.c_cflag &= ~CSIZE; /* Mask the character size bits */
options.c_cflag |= CS8;

/* Set the new options for the port */
tcsetattr(fd, TCSANOW, &options);
```

Informática II

Programación del microcontrolador ATmega328

Gonzalo F. Perez Paina

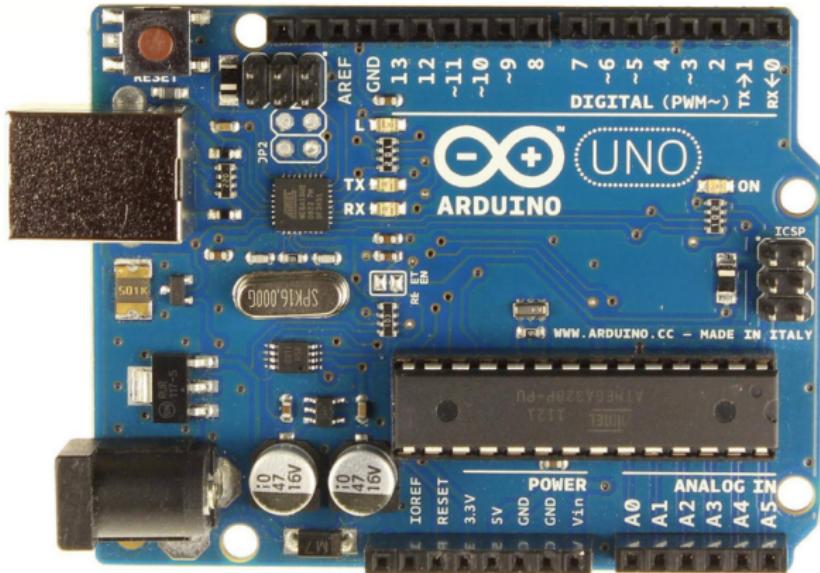


Universidad Tecnológica Nacional
Facultad Regional Córdoba
UTN-FRC

– 2021 –

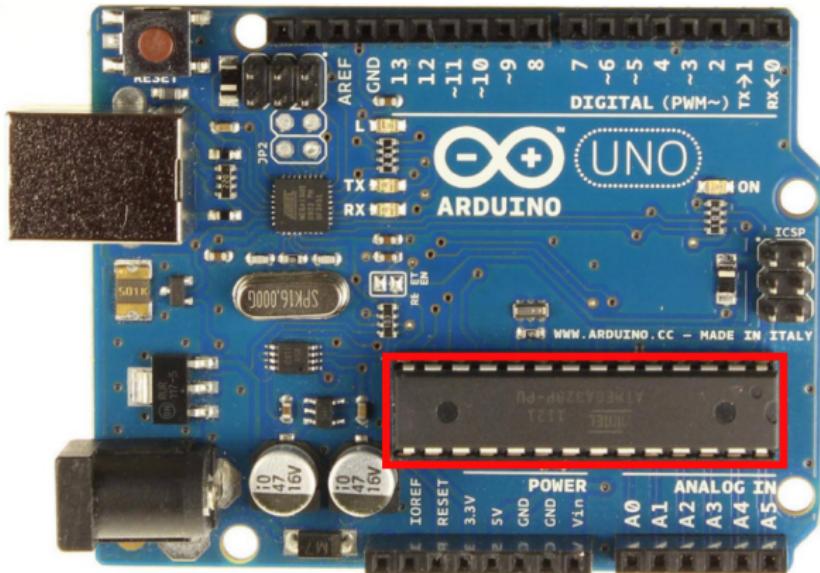
Arduino UNO Rev3

La placa Arduino UNO tiene un microcontrolador (μ C) marca Microchip® (anteriormente Atmel®) modelo ATmega328.



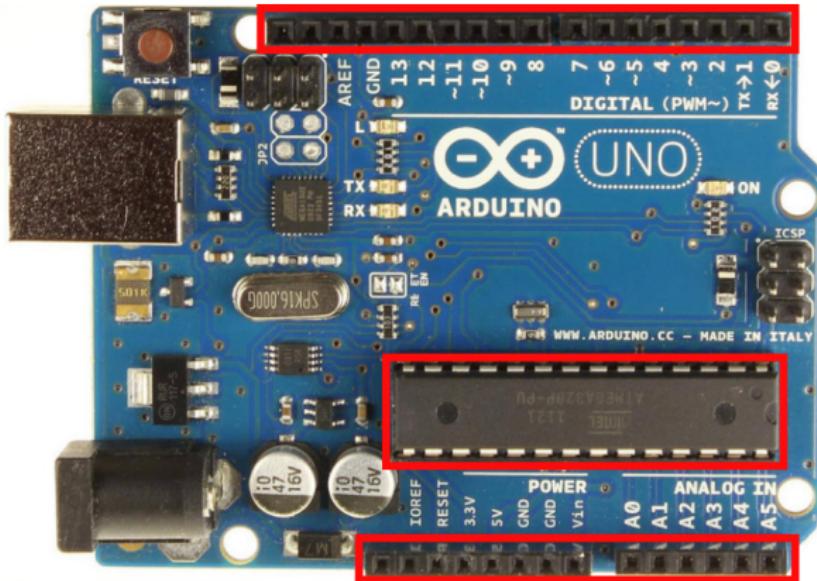
Arduino UNO Rev3

La placa Arduino UNO tiene un microcontrolador (μ C) marca Microchip® (anteriormente Atmel®) modelo ATmega328.



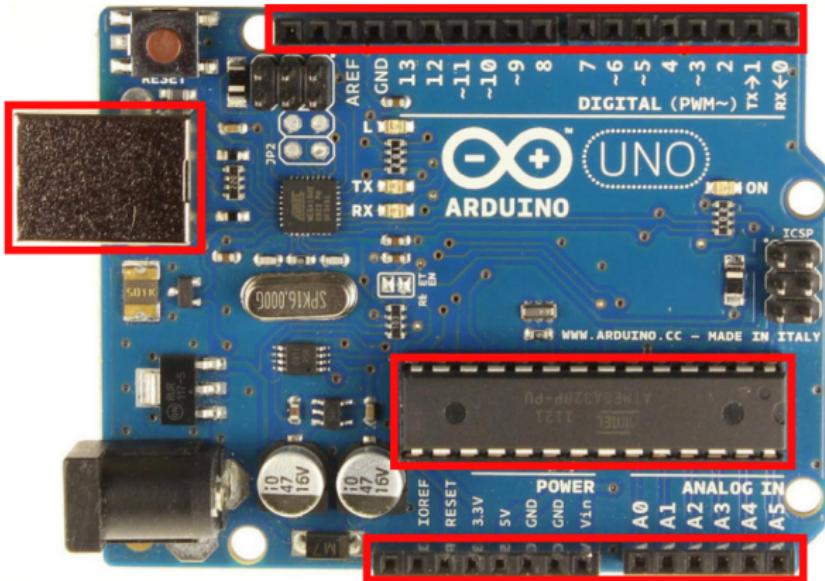
Arduino UNO Rev3

La placa Arduino UNO tiene un microcontrolador (μ C) marca Microchip® (anteriormente Atmel®) modelo ATmega328.



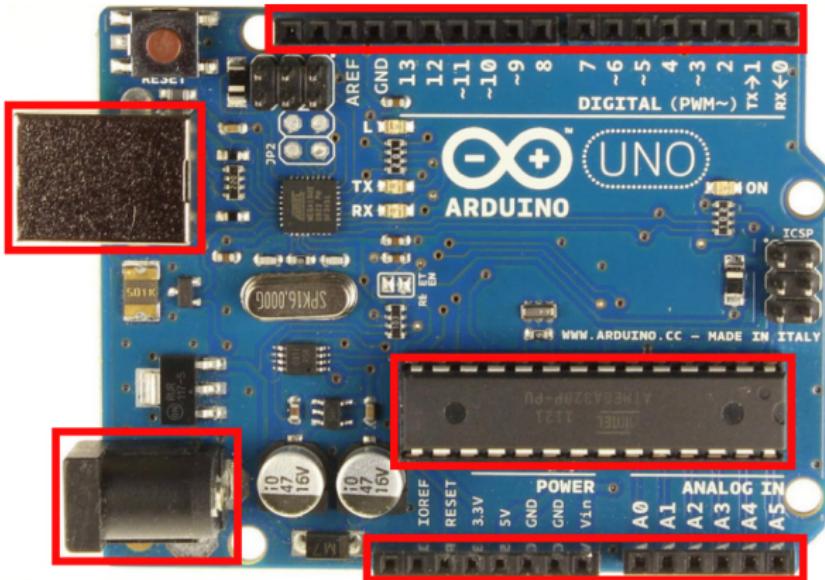
Arduino UNO Rev3

La placa Arduino UNO tiene un microcontrolador (μ C) marca Microchip® (anteriormente Atmel®) modelo ATmega328.



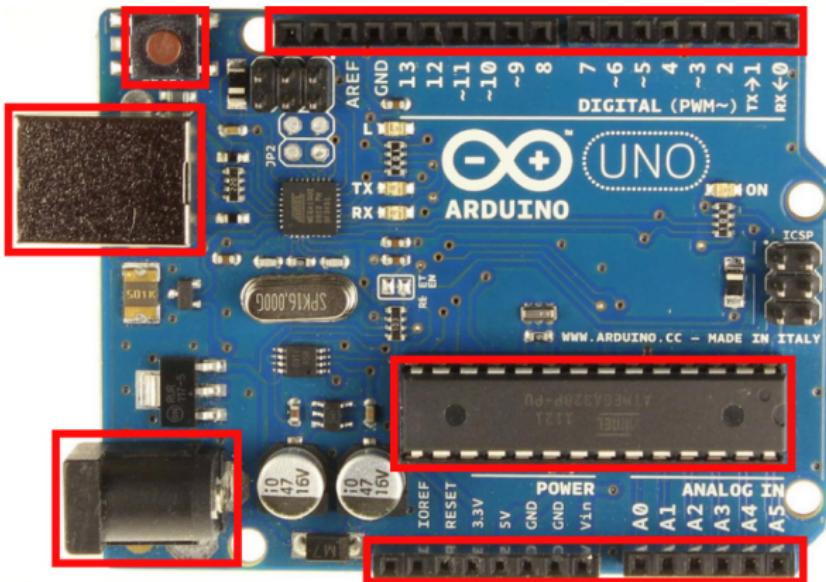
Arduino UNO Rev3

La placa Arduino UNO tiene un microcontrolador (μ C) marca Microchip® (anteriormente Atmel®) modelo ATmega328.



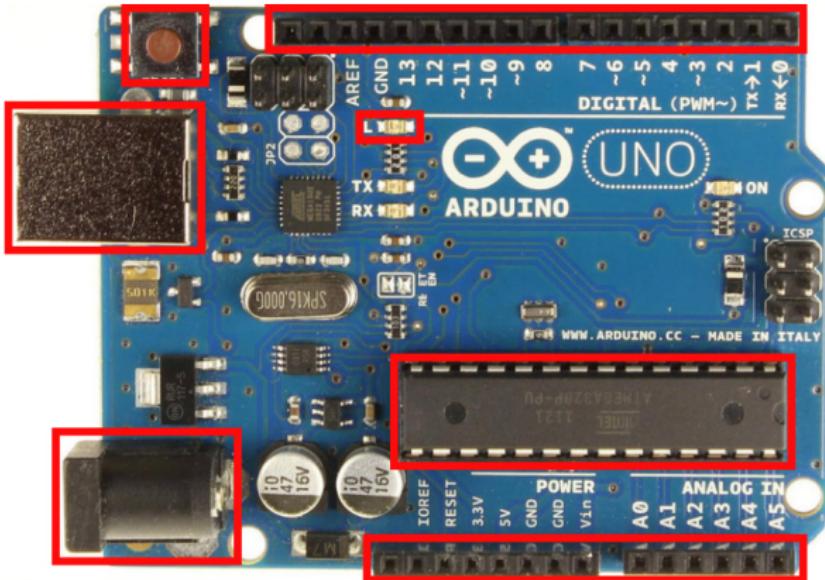
Arduino UNO Rev3

La placa Arduino UNO tiene un microcontrolador (μ C) marca Microchip® (anteriormente Atmel®) modelo ATmega328.



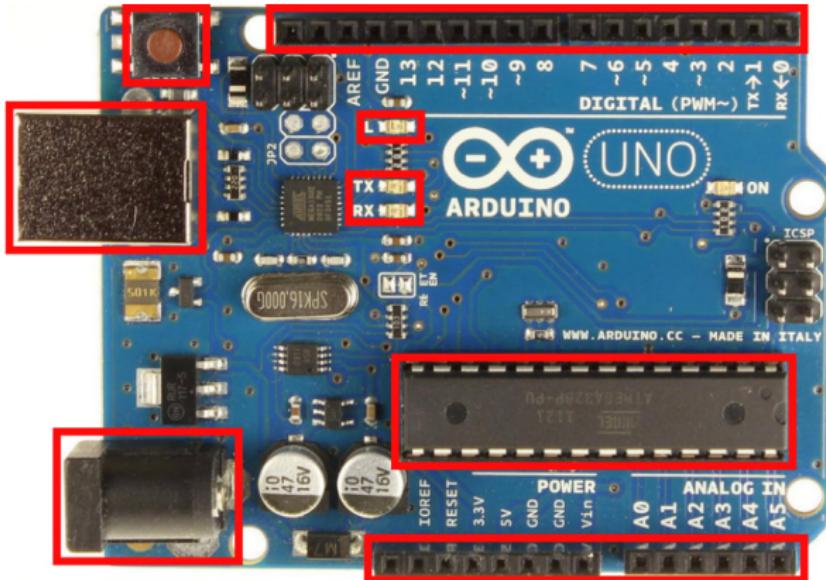
Arduino UNO Rev3

La placa Arduino UNO tiene un microcontrolador (μ C) marca Microchip® (anteriormente Atmel®) modelo ATmega328.



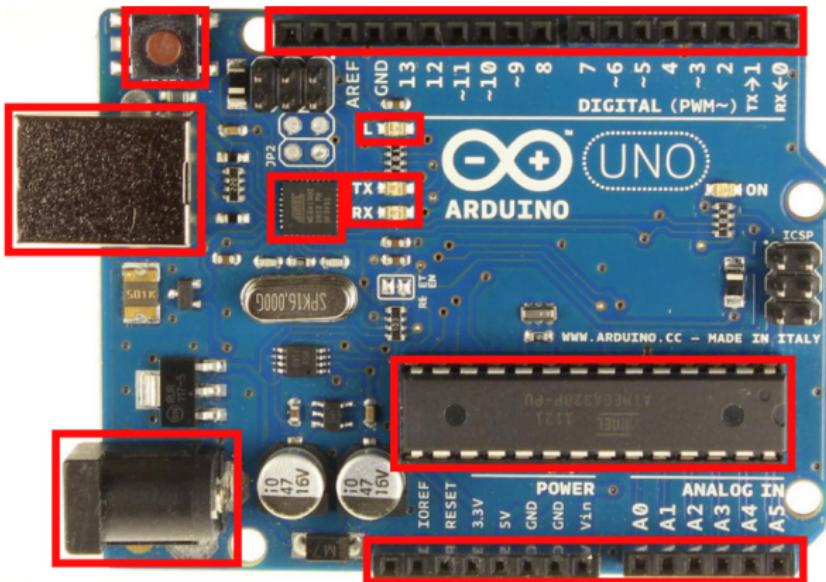
Arduino UNO Rev3

La placa Arduino UNO tiene un microcontrolador (μ C) marca Microchip® (anteriormente Atmel®) modelo ATmega328.



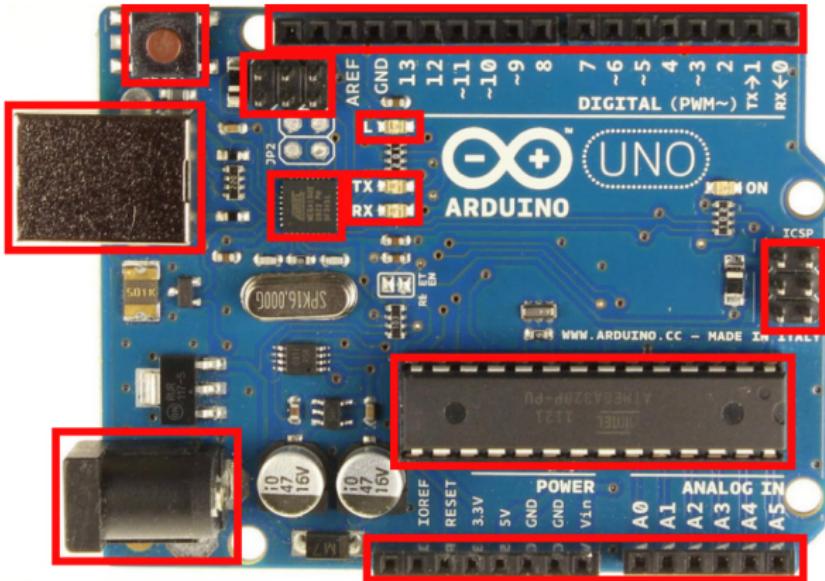
Arduino UNO Rev3

La placa Arduino UNO tiene un microcontrolador (μ C) marca Microchip® (anteriormente Atmel®) modelo ATmega328.



Arduino UNO Rev3

La placa Arduino UNO tiene un microcontrolador (μ C) marca Microchip® (anteriormente Atmel®) modelo ATmega328.

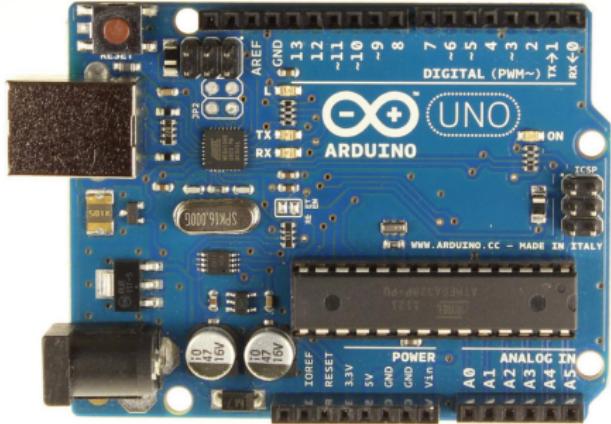


Arduino UNO Rev3 – Algunas características

Algunas de las características de esta placa son:

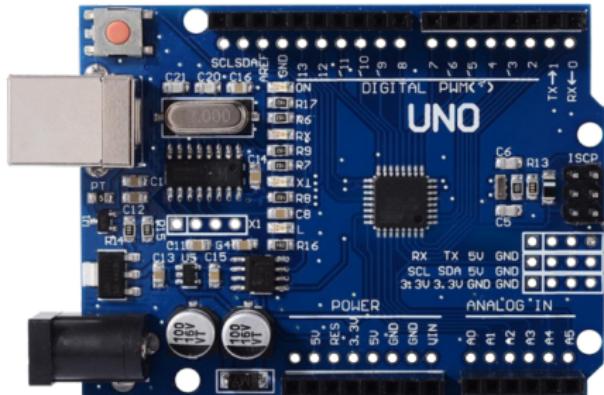
- ▶ Tensión de funcionamiento de 5V
- ▶ Tensión de entrada de 7 a 12V (de 6 a 20V como rango máximo)
- ▶ 14 pines digitales de entrada/salida
- ▶ 6 pines digitales con función PWM
- ▶ 6 pines de entrada analógica

Arduino UNO – Versiones



Arduino UNO Rev3:

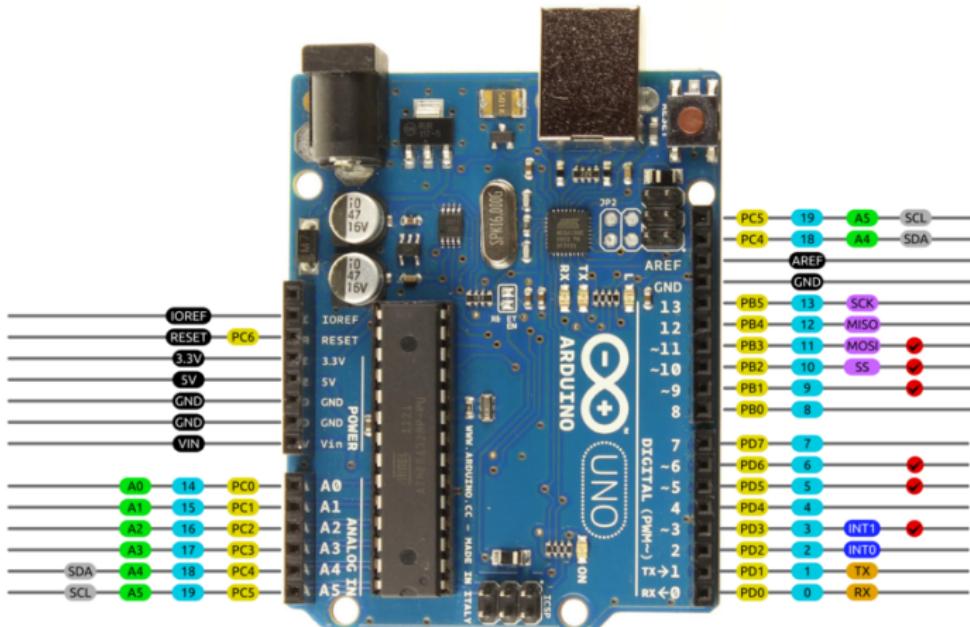
- ▶ ATmega328P DIP (Dual In-line Package)
- ▶ ATmega16U2 μ C con controlador USB, QFN (Quad-flat No-leads)



Arduino UNO CH340:

- ▶ ATmega328P TQFP (Thin Quad Flat Package)
- ▶ CH340, Chip USB a serial

Entrada/salida (pinout) y periféricos



AVR DIGITAL ANALOG POWER SERIAL SPI I2C PWM INTERRUPT



2014 by Bouni
Photo by Arduino.cc

El ATmega328

Es un microcontrolador de 8-bits de arquitectura AVR RISC mejorado.
Algunas de las características de este μ C son:

- ▶ Arquitectura RISC avanzada:
 - ▶ 131 instrucciones (la mayoría de las cuales se ejecutan en un ciclo de reloj)
 - ▶ 32 registros de propósitos generales de 8-bits
 - ▶ Máximo de 20MIPS (million instructions per second) @ 20MHz de frecuencia de reloj
 - ▶ Multiplicación por hardware

El ATmega328

Es un microcontrolador de 8-bits de arquitectura AVR RISC mejorado.
Algunas de las características de este μ C son:

- ▶ Arquitectura RISC avanzada:
 - ▶ 131 instrucciones (la mayoría de las cuales se ejecutan en un ciclo de reloj)
 - ▶ 32 registros de propósitos generales de 8-bits
 - ▶ Máximo de 20MIPS (million instructions per second) @ 20MHz de frecuencia de reloj
 - ▶ Multiplicación por hardware
- ▶ Memoria:
 - ▶ 32KB de memoria FLASH
 - ▶ 2KB de memoria SRAM (Static Random Access Memory)
 - ▶ 1KB de memoria EEPROM (Electrically Erasable Programmable Read-Only Memory)

El ATmega328

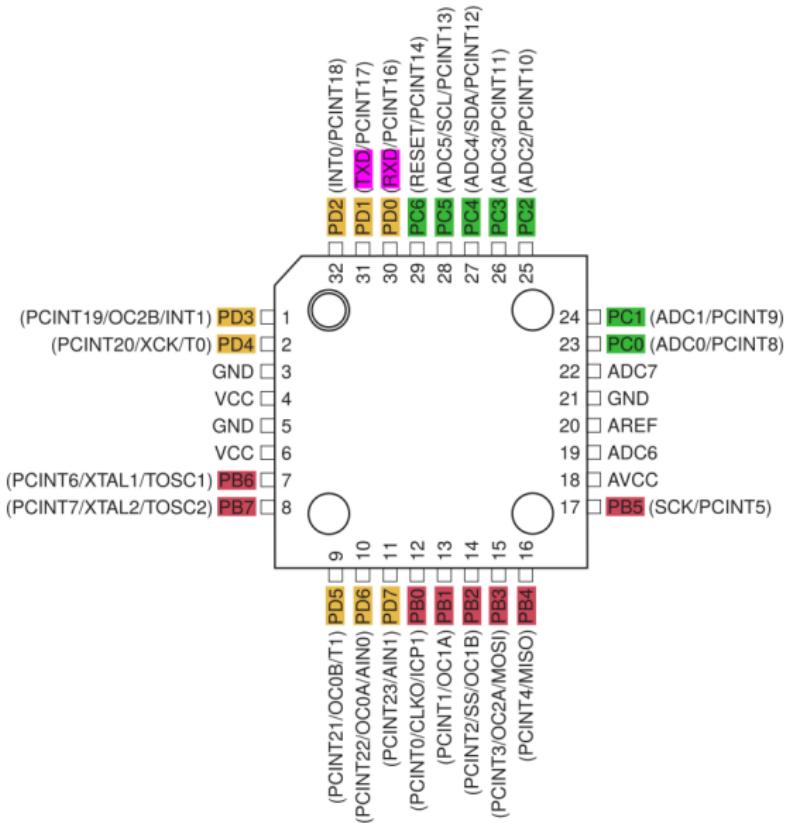
- ▶ Periféricos:
 - ▶ 23 entrada-salida de propósito general (GPIO: General Purpose Input-Output)
 - ▶ 6 canales de PWM (Pulse Width Modulation)
 - ▶ 6/8 canales de ADC (Analog to Digital Converter) de 10-bits
 - ▶ Puerto serial programable (USART: Universal Synchronous/Asynchronous Receiver/Transmitter)
 - ▶ Interfaz serial SPI (Serial Peripheral Interface) maestro-esclavo
 - ▶ Interfaz serial de 2-cables (compatible con el I²C de Philips)
 - ▶ Interrupción externa por cambio de nivel en entrada digital

El ATmega328

| 28 PDIP | |
|-----------------------------------|----|
| (PCINT14/RESET) PC6 | 1 |
| (PCINT16/ RXD) PD0 | 2 |
| (PCINT17/ TXD) PD1 | 3 |
| (PCINT18/INT0) PD2 | 4 |
| (PCINT19/OC2B/INT1) PD3 | 5 |
| (PCINT20/XCK/T0) PD4 | 6 |
| VCC | 7 |
| GND | 8 |
| (PCINT6/XTAL1/TOSC1) PB6 | 9 |
| (PCINT7/XTAL2/TOSC2) PB7 | 10 |
| (PCINT21/OC0B/T1) PD5 | 11 |
| (PCINT22/OC0A/AIN0) PD6 | 12 |
| (PCINT23/AIN1) PD7 | 13 |
| (PCINT0/CLKO/ICP1) PB0 | 14 |
| | 15 |
| | 16 |
| | 17 |
| | 18 |
| | 19 |
| | 20 |
| | 21 |
| | 22 |
| | 23 |
| | 24 |
| | 25 |
| | 26 |
| | 27 |
| | 28 |
| PC5 (ADC5/SCL/PCINT13) | |
| PC4 (ADC4/SDA/PCINT12) | |
| PC3 (ADC3/PCINT11) | |
| PC2 (ADC2/PCINT10) | |
| PC1 (ADC1/PCINT9) | |
| PC0 (ADC0/PCINT8) | |
| GND | |
| AREF | |
| AVCC | |
| PB5 (SCK/PCINT5) | |
| PB4 (MISO/PCINT4) | |
| PB3 (MOSI/OC2A/PCINT3) | |
| PB2 (SS/OC1B/PCINT2) | |
| PB1 (OC1A/PCINT1) | |

El ATmega328

32 TQFP Top View



El ATmega328 – Memoria

Tiene dos espacios de memorias diferentes (arquitectura Harvard):

1. una es la memoria de programa y
2. otra la memoria de datos.

El ATmega328 – Memoria

Tiene dos espacios de memorias diferentes (arquitectura Harvard):

1. una es la memoria de programa y
2. otra la memoria de datos.

Memoria de programa (Flash)

- ▶ La memoria Flash, para almacenar el programa, es de $16K \times 16\text{-bits}$ haciendo un total de 32KBytes ($32K \times 8\text{-bits}$).
- ▶ El contador de programa (PC: Program Counter) es de 14-bits lo que permite direccionar 16K ubicaciones de la memoria de programa.

El ATmega328 – Memoria

Tiene dos espacios de memorias diferentes (arquitectura Harvard):

1. una es la memoria de programa y
2. otra la memoria de datos.

Memoria de programa (Flash)

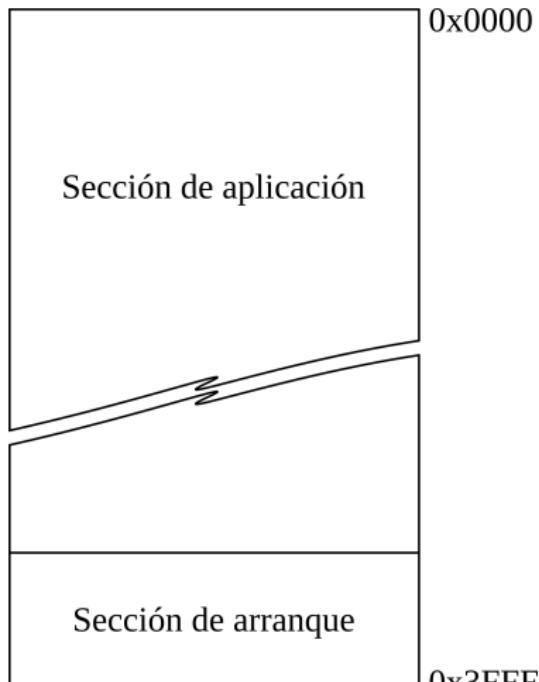
- ▶ La memoria Flash, para almacenar el programa, es de $16K \times 16\text{-bits}$ haciendo un total de 32KBytes ($32K \times 8\text{-bits}$).
- ▶ El contador de programa (PC: Program Counter) es de 14-bits lo que permite direccionar 16K ubicaciones de la memoria de programa.

Memoria de programa (RAM)

La memoria RAM se organiza en diferentes secciones.

- ▶ Las primeras direcciones para acceder a los registros de propósito general y los de entrada-salida,
- ▶ las siguientes para almacenar las variables del programa.

El ATmega328 – Memoria



Mapa de mem. Flash

Mapa de mem. RAM

Puerto digital de entrada/salida

Para el manejo de los puertos digitales de entrada/salida se utilizan 3 registros: DDRx, PORTx y PINx (x: B, C o D).

Puerto digital de entrada/salida

Para el manejo de los puertos digitales de entrada/salida se utilizan 3 registros: DDRx, PORTx y PINx (x: B, C o D).

- ▶ Los puertos digitales de entrada/salida suelen denominarse también como GPIO de General Purpose Input/Output.

Puerto digital de entrada/salida

Para el manejo de los puertos digitales de entrada/salida se utilizan 3 registros: DDRx, PORTx y PINx (x: B, C o D).

- ▶ Los puertos digitales de entrada/salida suelen denominarse también como GPIO de General Purpose Input/Output.

Registros

- ▶ DDRx: registro de dirección (Data Direction Register) del puerto x
- ▶ PORTx: registro de datos (Data Register) del puerto x
- ▶ PINx: registro de lectura (Input Pin Register) del puerto x

Puerto digital de entrada/salida

Para el manejo de los puertos digitales de entrada/salida se utilizan 3 registros: DDRx, PORTx y PINx (x: B, C o D).

- ▶ Los puertos digitales de entrada/salida suelen denominarse también como GPIO de General Purpose Input/Output.

Registros

- ▶ DDRx: registro de dirección (Data Direction Register) del puerto x
- ▶ PORTx: registro de datos (Data Register) del puerto x
- ▶ PINx: registro de lectura (Input Pin Register) del puerto x

Los registros de manejo de los puertos digitales de entrada/salida se encuentran mapeados en la memoria RAM de μ C, o sea que tienen direcciones de memorias fijas.

Puerto digital de entrada/salida

Registros para el manejo de puertos digitales de E/S

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---------------|--------|--------|--------|--------|--------|--------|--------|--------|-------------------------|
| | DDx7 | DDx6 | DDx5 | DDx4 | DDx3 | DDx2 | DDx1 | DDx0 | DDR_x |
| Read/Write | R/W | |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | PORTx7 | PORTx6 | PORTx5 | PORTx4 | PORTx3 | PORTx2 | PORTx1 | PORTx0 | PORT_x |
| Read/Write | R/W | |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | PINx7 | PINx6 | PINx5 | PINx4 | PINx3 | PINx2 | PINx1 | PINx0 | PIN_x |
| Read/Write | R/W | |
| Initial value | N/A | |

Puerto digital de entrada/salida

Registros para el manejo de puertos digitales de E/S

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---------------|--------|--------|--------|--------|--------|--------|--------|--------|-------------------------|
| | DDx7 | DDx6 | DDx5 | DDx4 | DDx3 | DDx2 | DDx1 | DDx0 | DDR_x |
| Read/Write | R/W | |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | PORTx7 | PORTx6 | PORTx5 | PORTx4 | PORTx3 | PORTx2 | PORTx1 | PORTx0 | PORT_x |
| Read/Write | R/W | |
| Initial value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | PINx7 | PINx6 | PINx5 | PINx4 | PINx3 | PINx2 | PINx1 | PINx0 | PIN_x |
| Read/Write | R/W | |
| Initial value | N/A | |

Los bits DD_{xn} del registro DD_x seleccionan si el pin correspondiente actuará como entrada o salida:

- ▶ Si se escribe un 1 lógico a DD_{xn} el pin Px_n se configura como salida.
- ▶ Si se escribe un 0 lógico a DD_{xn}, el pin Px_n se configura como entrada.

Puerto digital de entrada/salida

| | PINx | DDRx | PORTx |
|-----------------|------|------|-------|
| Puerto B | 0x23 | 0x24 | 0x25 |
| Puerto C | 0x26 | 0x27 | 0x28 |
| Puerto D | 0x29 | 0x2A | 0x2B |

Dirección de los registros de manejo de puertos digitales de E/S

Puerto serie USART

El μ C ATmega328 posee un periférico denominado USART (Universal Synchronous and Asynchronous Receiver and Transmitter) que permite la comunicación serie con otro dispositivo (p.e. una PC).

Puerto serie USART

El μ C ATmega328 posee un periférico denominado USART (Universal Synchronous and Asynchronous Receiver and Transmitter) que permite la comunicación serie con otro dispositivo (p.e. una PC).

- ▶ La USART del ATmega328 se denomina USART0.

Puerto serie USART

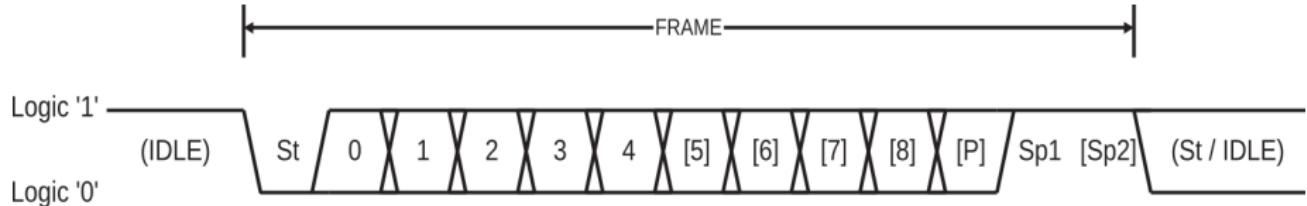
El μ C ATmega328 posee un periférico denominado USART (Universal Synchronous and Asynchronous Receiver and Transmitter) que permite la comunicación serie con otro dispositivo (p.e. una PC).

- ▶ La USART del ATmega328 se denomina USART0.

Algunas características

- ▶ Permite operación Full-duplex (tiene registros independientes de transmisión y recepción)
- ▶ Permite operación síncrona (modo SPI) y asíncrona (modo UART)
- ▶ Soporta tramas seriales con 5, 6, 7, 8 y 9 bits de datos, y 1 o 2 bits de parada
- ▶ Generación de paridad impar o par y verificación de paridad por hardware

Puerto serie USART – Trama

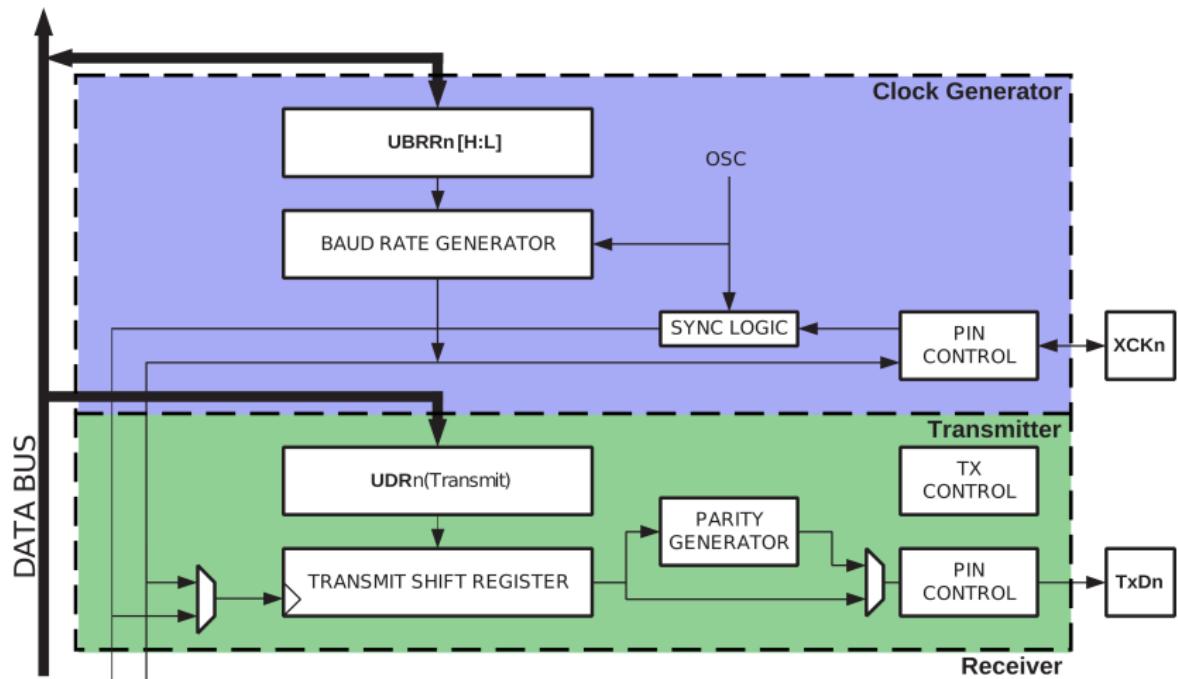


Donde:

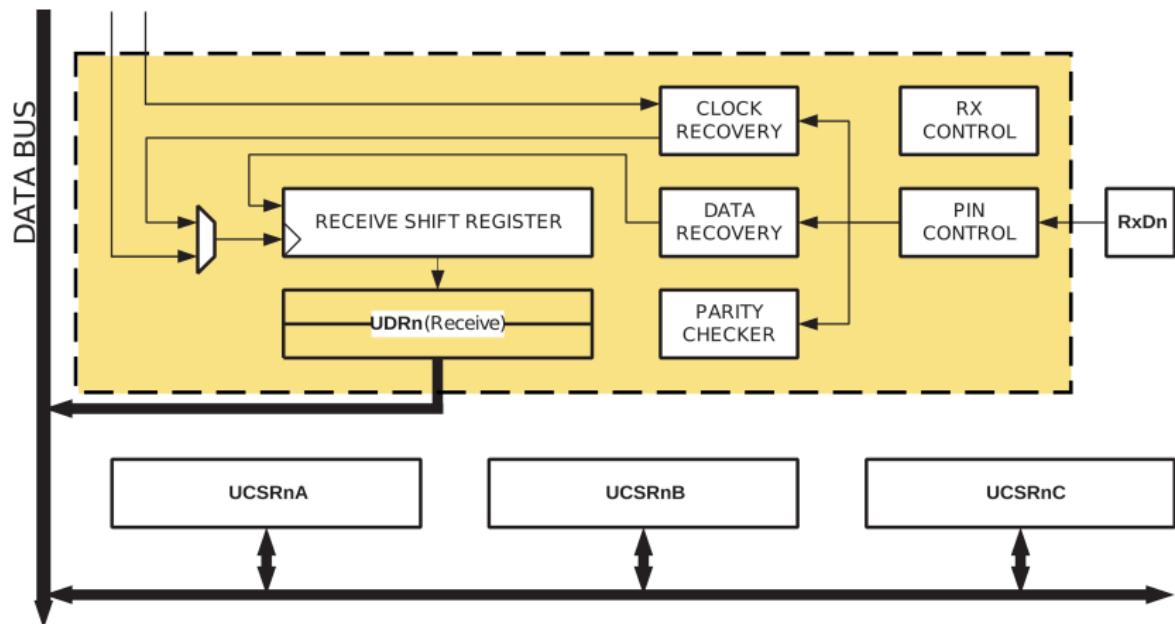
- ▶ **St:** Bit de inicio de trama (nivel bajo)
- ▶ **(n):** Bits de datos (0 a 8)
- ▶ **P:** Bit de paridad, que puede ser impar o par
- ▶ **Sp:** Bit de parada (nivel alto)
- ▶ **IDLE:** No hay transferencia de datos sobre las líneas de comunicación (RXD o TXD) (nivel alto)

Se envía primer el bit menos significativo (LSb: Least Significant bit)

Puerto serie USART – Diagrama en bloques (1/2)



Puerto serie USART – Diagrama en bloques (2/2)



Puerto serie USART – Velocidad de comunic.

- ▶ El registro Baud Rate (UBRRn) en conjunto con un contador descendente (down-counter) funcionan como un prescaler o generador de baud-rate programable.
- ▶ El contador descendente (corriendo a f_{osc}) se carga con el valor del registro UBRRn cada vez que alcanza el valor cero y genera un pulso de reloj.
- ▶ El reloj del generador de baud-rate tiene una frecuencia $f_{osc}/(UBRRn + 1)$ la cual se divide luego por 2, 8 y 16 dependiendo del modo de la USART.

Puerto serie USART – Velocidad de comunic.

- ▶ El registro Baud Rate (UBRRn) en conjunto con un contador descendente (down-counter) funcionan como un prescaler o generador de baud-rate programable.
- ▶ El contador descendente (corriendo a f_{osc}) se carga con el valor del registro UBRRn cada vez que alcanza el valor cero y genera un pulso de reloj.
- ▶ El reloj del generador de baud-rate tiene una frecuencia $f_{osc}/(UBRRn + 1)$ la cual se divide luego por 2, 8 y 16 dependiendo del modo de la USART.

En el modo asíncrono normal la tasa de transmisión de datos (BR) es

$$BR = \frac{f_{osc}}{16 \cdot (UBRRn + 1)},$$

por lo tanto

$$UBRRn = \frac{f_{osc}}{16 \cdot BR} - 1.$$

donde UBRRn es el valor del registro de velocidad.

Puerto serie USART – Velocidad de comunic.

| Baud rate | UBRRn | Error (%) | Baud rate | UBRRn | Error (%) |
|-----------|-------|-----------|-----------|-------|-----------|
| 2400 | 416 | -0.1 | 28.8K | 34 | -0.8 |
| 4800 | 207 | 0.2 | 38.4K | 25 | 0.2 |
| 9600 | 103 | 0.2 | 57.6K | 16 | 2.1 |
| 14.4K | 68 | 0.6 | 76.8K | 12 | 0.2 |
| 19.2K | 51 | 0.2 | 115.2K | 8 | -3.5 |

Velocidades de comunicación y valor del registro UBRRn
para $f_{osc} = 16MHz$

Puerto serie USART – Inicialización

- ▶ Para poder utilizar la USART en una comunicación es necesario su inicialización.
- ▶ En general, el proceso de inicialización consiste en configurar el baud-rate, el formato de la trama y habilitar el transmisor y/o el receptor.

Puerto serie USART – Inicialización

- ▶ Para poder utilizar la USART en una comunicación es necesario su inicialización.
- ▶ En general, el proceso de inicialización consiste en configurar el baud-rate, el formato de la trama y habilitar el transmisor y/o el receptor.

Configuración:

1. Configuración del baud-rate: se realiza escribiendo los registros **UBRR0[H:L]**

Puerto serie USART – Inicialización

- ▶ Para poder utilizar la USART en una comunicación es necesario su inicialización.
- ▶ En general, el proceso de inicialización consiste en configurar el baud-rate, el formato de la trama y habilitar el transmisor y/o el receptor.

Configuración:

1. Configuración del baud-rate: se realiza escribiendo los registros **UBRR0[H:L]**
2. El formato de la trama de comunicación se configura utilizando el registro **UCSR0C**

Puerto serie USART – Inicialización

- ▶ Para poder utilizar la USART en una comunicación es necesario su inicialización.
- ▶ En general, el proceso de inicialización consiste en configurar el baud-rate, el formato de la trama y habilitar el transmisor y/o el receptor.

Configuración:

1. Configuración del baud-rate: se realiza escribiendo los registros **UBRR0[H:L]**
2. El formato de la trama de comunicación se configura utilizando el registro **UCSR0C**
3. La habilitación de la transmisión y recepción se realiza a través del registro **UCSR0B**

Puerto serie USART – Registros

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|-----------------------------|-------------------|----------------|---------------|---------------|--------------------|-----------------|-----------------|-----------------|---------------------|
| 0xC6 | RXB[7:0] | | | | | | | | UDR0 (Read) |
| | TXB[7:0] | | | | | | | | UDR0 (Write) |
| Read/Write Initial value | R/W 0 | R/W 0 | R/W 0 | R/W 0 | R/W 0 | R/W 0 | R/W 0 | R/W 0 | |
| 0xC0 | RXC0 | TXC0 | UDRE0 | FE0 | DOR0 | UPE0 | U2X0 | MPCM0 | UCSR0A |
| Read/Write Initial value | R 0 | R/W 0 | R 1 | R 0 | R 0 | R 0 | R/W 0 | R/W 0 | |
| 0xC1 | RXCIE0 | TXCIE0 | UDRIE0 | RXEN0 | TXEN0 | UCSZ02 | RXB80 | TXB80 | UCSR0B |
| Read/Write Initial value | R/W 0 | R/W 0 | R/W 0 | R/W 0 | R/W 0 | R/W 0 | R 0 | R/W 0 | |
| 0xC2 | UMSEL01 | UMSEL00 | UPM01 | UPM00 | USBS0 | UCSZ01 | UCSZ00 | UCPOL0 | UCSR0C |
| Read/Write Initial value | R/W 0 | R/W 0 | R/W 0 | R/W 0 | R/W 0 | R/W 1 | R 1 | R/W 0 | |
| 0xC5 | - | - | - | - | UBRR0[11:8] | | | | |
| 0xC4 | UBRR0[7:0] | | | | | | | | UBRR0H |
| Read/Write Initial value | R R/W 0 | R R/W 0 | R R/W 0 | R R/W 0 | R/W R/W 0 | R/W R/W 0 | R/W R/W 0 | R/W R/W 0 | UBRR0L |

USART0 – Registro de datos

Los registros búfer de transmisión y de recepción de datos de la USART comparten la misma dirección de entrada/salida como registro UDR0 (USART Data Register).

- ▶ Cuando se realiza una escritura en UDR0, los datos se almacenan en el registro búfer de transmisión (TXB).

USART0 – Registro de datos

Los registros búfer de transmisión y de recepción de datos de la USART comparten la misma dirección de entrada/salida como registro UDR0 (USART Data Register).

- ▶ Cuando se realiza una escritura en UDR0, los datos se almacenan en el registro búfer de transmisión (TXB).
- ▶ Cuando se lee UDR0 se obtiene el valor del registro búfer de recepción de datos (RXB).

USART0 – Registro de datos

Los registros búfer de transmisión y de recepción de datos de la USART comparten la misma dirección de entrada/salida como registro UDR0 (USART Data Register).

- ▶ Cuando se realiza una escritura en UDR0, los datos se almacenan en el registro búfer de transmisión (TXB).
- ▶ Cuando se lee UDR0 se obtiene el valor del registro búfer de recepción de datos (RXB).
- ▶ El búfer de transmisión puede escribirse cuando el bit UDRE0 del registro UCSROA este a 1.

USART0 – Registro de datos

Los registros búfer de transmisión y de recepción de datos de la USART comparten la misma dirección de entrada/salida como registro UDR0 (USART Data Register).

- ▶ Cuando se realiza una escritura en UDR0, los datos se almacenan en el registro búfer de transmisión (TXB).
- ▶ Cuando se lee UDR0 se obtiene el valor del registro búfer de recepción de datos (RXB).
- ▶ El búfer de transmisión puede escribirse cuando el bit UDRE0 del registro UCSROA este a 1.
- ▶ El búfer de recepción consta de una FIFO de dos niveles, cuyo estado cambia cada vez que se acceda al búfer de recepción.

USART0 – Registro de control y estado A

Funciones de los bits más utilizados del registro UCSR0A:

- ▶ Bit 7 – RXC0 (USART Receive Complete): bit que actúa como bandera (flag) para indicar que hay un dato no leído en el búfer de recepción y se borra cuando el búfer de recepción está vacío.

USART0 – Registro de control y estado A

Funciones de los bits más utilizados del registro UCSR0A:

- ▶ Bit 7 – RXC0 (USART Receive Complete): bit que actúa como bandera (flag) para indicar que hay un dato no leído en el búfer de recepción y se borra cuando el búfer de recepción está vacío.
- ▶ Bit 6 – TXC0 (USART Transmit Complete): este bit de bandera se pone a 1 cuando la trama completa en el registro de desplazamiento de transmisión ha sido serializado y no hay datos nuevos en el búfer de transmisión.

USART0 – Registro de control y estado A

Funciones de los bits más utilizados del registro UCSR0A:

- ▶ Bit 7 – RXC0 (USART Receive Complete): bit que actúa como bandera (flag) para indicar que hay un dato no leído en el búfer de recepción y se borra cuando el búfer de recepción está vacío.
- ▶ Bit 6 – TXC0 (USART Transmit Complete): este bit de bandera se pone a 1 cuando la trama completa en el registro de desplazamiento de transmisión ha sido serializado y no hay datos nuevos en el búfer de transmisión.
- ▶ Bit 5 – UDRE0 (USART Data Register Empty): este flag indica si el búfer de transmisión está listo para recibir un dato nuevo. Si está a 1 el búfer está vacío y listo para ser escrito.

USART0 – Registro de control y estado A

Funciones de los bits más utilizados del registro UCSR0A:

- ▶ Bit 7 – RXC0 (USART Receive Complete): bit que actúa como bandera (flag) para indicar que hay un dato no leído en el búfer de recepción y se borra cuando el búfer de recepción está vacío.
- ▶ Bit 6 – TXC0 (USART Transmit Complete): este bit de bandera se pone a 1 cuando la trama completa en el registro de desplazamiento de transmisión ha sido serializado y no hay datos nuevos en el búfer de transmisión.
- ▶ Bit 5 – UDRE0 (USART Data Register Empty): este flag indica si el búfer de transmisión está listo para recibir un dato nuevo. Si está a 1 el búfer está vacío y listo para ser escrito.
- ▶ Bit 4 – FEO (Frame Error):

USART0 – Registro de control y estado A

Funciones de los bits más utilizados del registro UCSR0A:

- ▶ Bit 7 – RXC0 (USART Receive Complete): bit que actúa como bandera (flag) para indicar que hay un dato no leído en el búfer de recepción y se borra cuando el búfer de recepción está vacío.
- ▶ Bit 6 – TXC0 (USART Transmit Complete): este bit de bandera se pone a 1 cuando la trama completa en el registro de desplazamiento de transmisión ha sido serializado y no hay datos nuevos en el búfer de transmisión.
- ▶ Bit 5 – UDRE0 (USART Data Register Empty): este flag indica si el búfer de transmisión está listo para recibir un dato nuevo. Si está a 1 el búfer está vacío y listo para ser escrito.
- ▶ Bit 4 – FEO (Frame Error):
- ▶ Bit 3 – DOR0 (Data OverRun):

USART0 – Registro de control y estado A

Funciones de los bits más utilizados del registro UCSR0A:

- ▶ Bit 7 – RXC0 (USART Receive Complete): bit que actúa como bandera (flag) para indicar que hay un dato no leído en el búfer de recepción y se borra cuando el búfer de recepción está vacío.
- ▶ Bit 6 – TXC0 (USART Transmit Complete): este bit de bandera se pone a 1 cuando la trama completa en el registro de desplazamiento de transmisión ha sido serializado y no hay datos nuevos en el búfer de transmisión.
- ▶ Bit 5 – UDRE0 (USART Data Register Empty): este flag indica si el búfer de transmisión está listo para recibir un dato nuevo. Si está a 1 el búfer está vacío y listo para ser escrito.
- ▶ Bit 4 – FEO (Frame Error):
- ▶ Bit 3 – DOR0 (Data OverRun):
- ▶ Bit 2 – UPE0 (USART Parity Error):

USART0 – Registro de control y estado B

Funciones de los bits más utilizados del registro UCSR0B:

- ▶ Bit 4 – RXENO (Receiver Enable): al escribir un 1 se habilita la recepción de datos y se configura el pin RxD0.

USART0 – Registro de control y estado B

Funciones de los bits más utilizados del registro UCSR0B:

- ▶ Bit 4 – RXENO (Receiver Enable): al escribir un 1 se habilita la recepción de datos y se configura el pin RxD0.
- ▶ Bit 3 – TXENO (Transmitter Enable): al escribir un 1 se habilita la transmisión de datos y se configura el pin TxD0.

USART0 – Registro de control y estado B

Funciones de los bits más utilizados del registro UCSR0B:

- ▶ Bit 4 – RXENO (Receiver Enable): al escribir un 1 se habilita la recepción de datos y se configura el pin RxD0.
- ▶ Bit 3 – TXENO (Transmitter Enable): al escribir un 1 se habilita la transmisión de datos y se configura el pin TxD0.
- ▶ Bit 2 – UCSZ02 (Character Size): este bit, en combinación con los bits UCSZ01:0, configuran la cantidad de bits de datos (Character SiZe) de la trama de transmisión y recepción.

USART0 – Registro de control y estado C

Funciones de los bits más utilizados del registro UCSR0C:

- ▶ Bits 7:6 – UMSEL_{01:0} (USART Mode Select): estos bits seleccionan el modo de operación de la USART0, donde para el modo asíncrono los valores deben ser 00.

USART0 – Registro de control y estado C

Funciones de los bits más utilizados del registro UCSR0C:

- ▶ Bits 7:6 – UMSEL_{1:0} (USART Mode Select): estos bits seleccionan el modo de operación de la USART0, donde para el modo asíncrono los valores deben ser 00.
- ▶ Bits 5:4 – UPM_{1:0} (Parity Mode): fijan el tipo de paridad utilizada. Estos son: 00, paridad desactivada; 01, reservado; 10, paridad par; y 11, paridad impar.

USART0 – Registro de control y estado C

Funciones de los bits más utilizados del registro UCSR0C:

- ▶ Bits 7:6 – UMSEL01:0 (USART Mode Select): estos bits seleccionan el modo de operación de la USART0, donde para el modo asíncrono los valores deben ser 00.
- ▶ Bits 5:4 – UPM01:0 (Parity Mode): fijan el tipo de paridad utilizada. Estos son: 00, paridad desactivada; 01, reservado; 10, paridad par; y 11, paridad impar.
- ▶ Bit 3 – USBS0 (Stop Bit Select): seleccionan la cantidad de bits de stop agregados en la transmisión (el receptor ignora esta configuración). Las opciones son 0: 1-bit de parada, o 1: 2-bits de parada.

USART0 – Registro de control y estado C

Funciones de los bits más utilizados del registro UCSR0C:

- ▶ Bits 7:6 – UMSEL01:0 (USART Mode Select): estos bits seleccionan el modo de operación de la USART0, donde para el modo asíncrono los valores deben ser 00.
- ▶ Bits 5:4 – UPM01:0 (Parity Mode): fijan el tipo de paridad utilizada. Estos son: 00, paridad desactivada; 01, reservado; 10, paridad par; y 11, paridad impar.
- ▶ Bit 3 – USBS0 (Stop Bit Select): seleccionan la cantidad de bits de stop agregados en la transmisión (el receptor ignora esta configuración). Las opciones son 0: 1-bit de parada, o 1: 2-bits de parada.
- ▶ Bit 2:1 – UCSZ01:0 (Character Size): estos bits, en conjunto con el bit UCSZ02 en UCSR0B, configuran la cantidad de bits de datos (Character SiZe) utilizada en la trama de transmisión y recepción.

USART0 – Tamaño de carácter (UCSR0B/C)

| UCSZ02 | UCSZ01 | UCSZ00 | Tamaño de carácter |
|--------|--------|--------|--------------------|
| 0 | 0 | 0 | 5 bits |
| 0 | 0 | 1 | 6 bits |
| 0 | 1 | 0 | 7 bits |
| 0 | 1 | 1 | 8 bits |
| 1 | 0 | 0 | Reservado |
| 1 | 0 | 1 | Reservado |
| 1 | 1 | 0 | Reservado |
| 1 | 1 | 1 | 9 bits |

Programación con IDE Arduino



Arduino IDE 1.8.13

The open-source Arduino Software (IDE) makes it easy to write code and upload it to the board. This software can be used with any Arduino board.

Refer to the [Getting Started](#) page for installation instructions.

SOURCE CODE

Active development of the Arduino software is [hosted by GitHub](#). See the instructions for [building the code](#). Latest release source code archives are available [here](#). The archives are PGP-signed so they can be verified using [this](#) gpg key.

DOWNLOAD OPTIONS

Windows Win 7 and newer

Windows ZIP file

Windows app Win 8.1 or 10



Linux 32 bits

Linux 64 bits

Linux ARM 32 bits

Linux ARM 64 bits

Mac OS X 10.10 or newer

[Release Notes](#) [Checksums \(sha512\)](#)

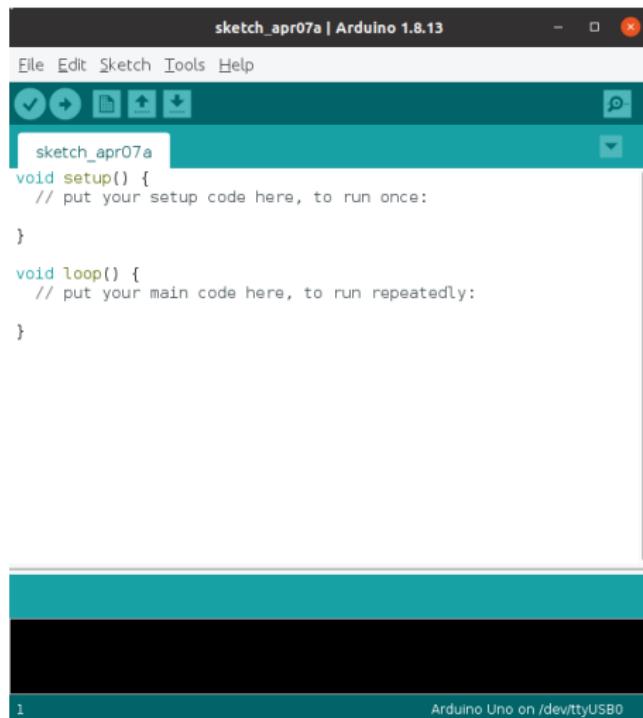
- ▶ Descargar y descomprimir el archivo (`arduino-1.8.13-linux64.tar.xz`)
- ▶ Ejecutar el script de instalación (`./install.sh`)

Programación con IDE Arduino

IDE: Conjunto de herramientas de Sw que permite a los programadores desarrollar (básicamente escribir y probar) sus propios programas.

Programación con IDE Arduino

IDE: Conjunto de herramientas de Sw que permite a los programadores desarrollar (básicamente escribir y probar) sus propios programas.



The screenshot shows the Arduino IDE interface. The title bar reads "sketch_apr07a | Arduino 1.8.13". The menu bar includes File, Edit, Sketch, Tools, and Help. Below the menu is a toolbar with icons for save, upload, and search. A dropdown menu shows "sketch_apr07a". The main code editor contains the following code:

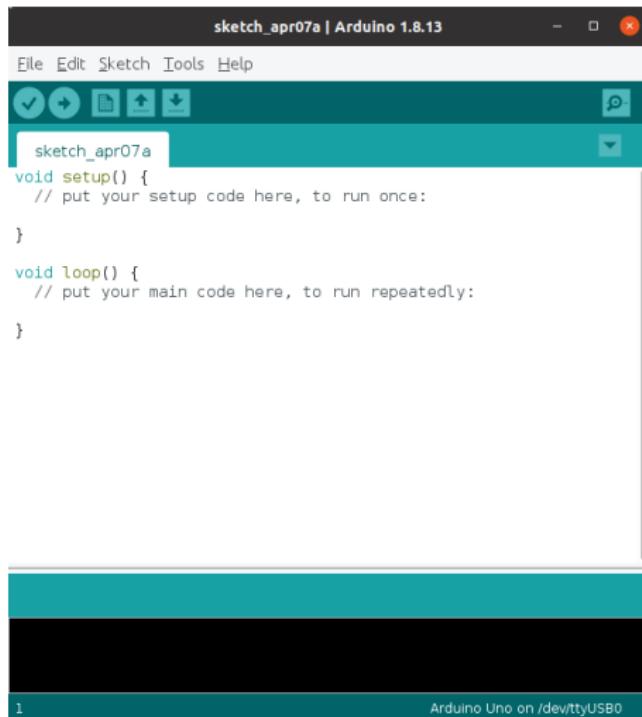
```
void setup() {
  // put your setup code here, to run once:
}

void loop() {
  // put your main code here, to run repeatedly:
}
```

The status bar at the bottom indicates "1" and "Arduino Uno on /dev/ttyUSB0".

Programación con IDE Arduino

IDE: Conjunto de herramientas de Sw que permite a los programadores desarrollar (básicamente escribir y probar) sus propios programas.

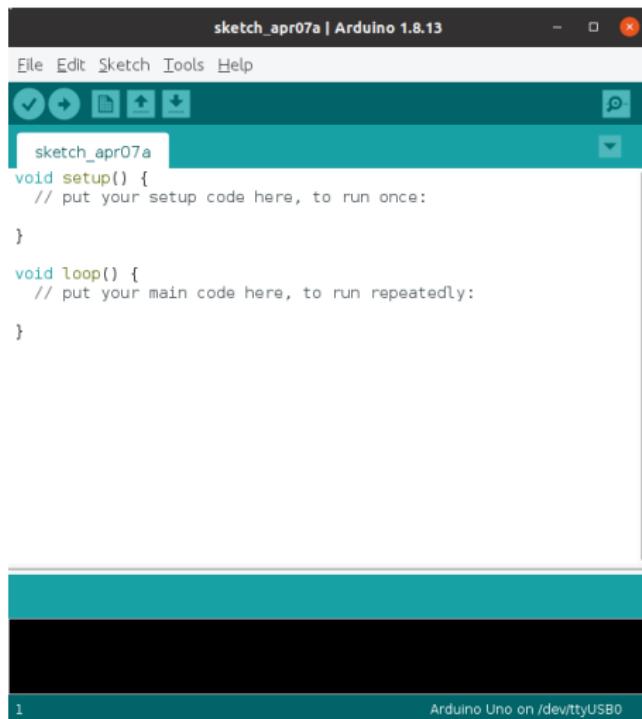


Se divide en 4 áreas:

1. Barra de menús
2. Barra de botones
3. Editor
4. Ventana de mensajes

Programación con IDE Arduino

IDE: Conjunto de herramientas de Sw que permite a los programadores desarrollar (básicamente escribir y probar) sus propios programas.



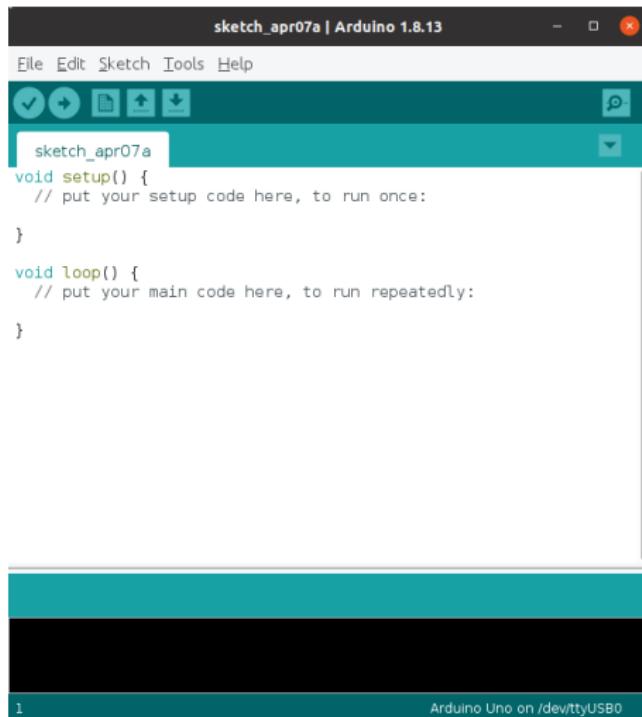
Se divide en 4 áreas:

1. Barra de menús
2. Barra de botones
3. Editor
4. Ventana de mensajes

sketch Arduino – 2 bloques:

Programación con IDE Arduino

IDE: Conjunto de herramientas de Sw que permite a los programadores desarrollar (básicamente escribir y probar) sus propios programas.



Se divide en 4 áreas:

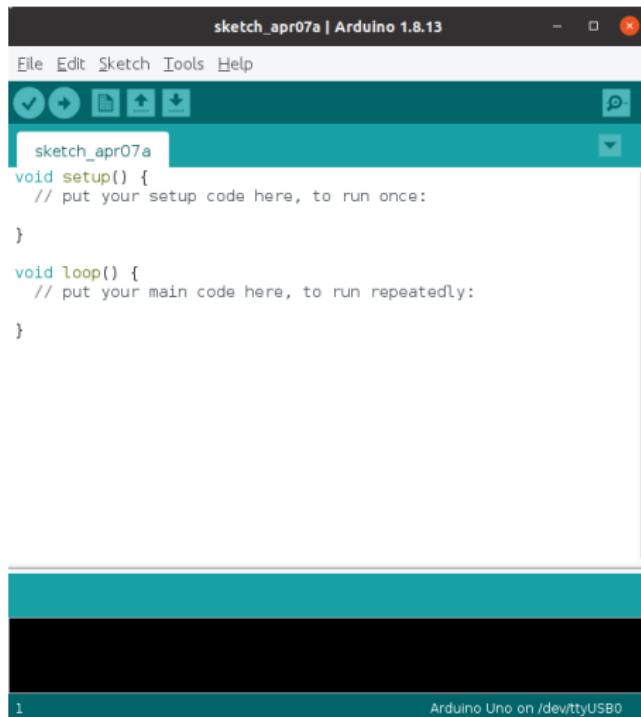
1. Barra de menús
2. Barra de botones
3. Editor
4. Ventana de mensajes

sketch Arduino – 2 bloques:

- ▶ **setup()**: se ejecuta una única vez cuando se enciende o resetea la placa

Programación con IDE Arduino

IDE: Conjunto de herramientas de Sw que permite a los programadores desarrollar (básicamente escribir y probar) sus propios programas.



Se divide en 4 áreas:

1. Barra de menús
2. Barra de botones
3. Editor
4. Ventana de mensajes

sketch Arduino – 2 bloques:

- ▶ **setup()**: se ejecuta una única vez cuando se enciende o resetea la placa
- ▶ **loop()**: se ejecuta de forma constante (bucle)

Sketch ejemplo: blink

File->Examples->01.Basics->Blink

```
1 // the setup function runs once when you press reset or
2 // power the board
3 void setup() {
4     // initialize digital pin LED_BUILTIN as an output.
5     pinMode(LED_BUILTIN, OUTPUT);
6 }
7
8 // the loop function runs over and over again forever
9 void loop() {
10    digitalWrite(LED_BUILTIN, HIGH);      // turn the LED on
11    delay(1000);                      // wait for a second
12    digitalWrite(LED_BUILTIN, LOW);     // turn the LED off
13    delay(1000);                      // wait for a second
14 }
```

Sketch ejemplo: puerto serie

```
1 #define MENSAJE "Hola mundo"
2
3 // La función 'setup' se ejecuta una única vez al
4 // presionar reset o encender la placa
5 void setup() {
6     // Inicializa el puerto serie (UART) a 9600 bps.
7     Serial.begin(9600);
8 }
9
10 // La función 'loop' corre indefinidamente una y otra vez
11 void loop() {
12     // Imprime (envía) la cadena MENSAJE por puerto serie.
13     Serial.println(MENSAJE);
14     delay(500);
15 }
```

Programación con Toolchain GCC para AVR

El Toolchain de GCC para arquitectura AVR está compuesto por:

`gcc`: el compilador de C y C++

Programación con Toolchain GCC para AVR

El Toolchain de GCC para arquitectura AVR está compuesto por:

`gcc`: el compilador de C y C++

`binutils`: colección de herramientas para la manipulación de archivos binarios

Programación con Toolchain GCC para AVR

El Toolchain de GCC para arquitectura AVR está compuesto por:

`gcc`: el compilador de C y C++

`binutils`: colección de herramientas para la manipulación de archivos binarios

`libc-avr`: subconjunto de la biblioteca estándar de C con funciones específicas para arquitectura AVR

Programación con Toolchain GCC para AVR

El Toolchain de GCC para arquitectura AVR está compuesto por:

`gcc`: el compilador de C y C++

`binutils`: colección de herramientas para la manipulación de archivos binarios

`libc-avr`: subconjunto de la biblioteca estándar de C con funciones específicas para arquitectura AVR

`avrdude`: programa para escribir y leer la memoria Flash del μ C

Programación con Toolchain GCC para AVR

El Toolchain de GCC para arquitectura AVR está compuesto por:

`gcc`: el compilador de C y C++

`binutils`: colección de herramientas para la manipulación de archivos binarios

`libc-avr`: subconjunto de la biblioteca estándar de C con funciones específicas para arquitectura AVR

`avrdude`: programa para escribir y leer la memoria Flash del μ C

La instalación del Toolchain en Ubuntu se realiza instalando los siguientes paquetes: `gcc-avr`, `binutils-avr`, `avr-libc` y `avrdude`.

Programación con Toolchain GCC para AVR

El Toolchain de GCC para arquitectura AVR está compuesto por:

`gcc`: el compilador de C y C++

`binutils`: colección de herramientas para la manipulación de archivos binarios

`libc-avr`: subconjunto de la biblioteca estándar de C con funciones específicas para arquitectura AVR

`avrdude`: programa para escribir y leer la memoria Flash del μ C

La instalación del Toolchain en Ubuntu se realiza instalando los siguientes paquetes: `gcc-avr`, `binutils-avr`, `avr-libc` y `avrdude`.

- ▶ El compilador GCC para AVR (`avr-gcc`) es en realidad es un compilador cruzado (cross-compiler).
- ▶ Un compilador cruzado es aquel que genera código máquina para una arquitectura diferente a aquella en la cual se ejecuta el compilador.

Programación con Toolchain GCC para AVR

Archivo 'blink.c'

```
1 #include <avr/io.h>
2 #include <util/delay.h>
3
4 #define BLINK_DELAY_MS 1000
5
6 int main (void)
7 {
8     /* Inicializa el pin digital como salida */
9     DDRB |= _BV(DDB5);
10
11    while(1)
12    {
13        PORTB |= _BV(PORTB5); /* Enciende LED */
14        _delay_ms(BLINK_DELAY_MS);
15
16        PORTB &= ~_BV(PORTB5); /* Apaga LED */
17        _delay_ms(BLINK_DELAY_MS);
18    }
19    return 0;
20 }
```

Programación con Toolchain GCC para AVR

Construcción del programa y grabación en la memoria del μ C ATmega328 de la placa Arduino UNO:

Programación con Toolchain GCC para AVR

Construcción del programa y grabación en la memoria del μ C ATmega328 de la placa Arduino UNO:

1. Construcción del código fuente (.c) hasta la etapa de ensamblado, de la cual se obtiene un archivo objeto (.o):

```
avr-gcc -Os -DF_CPU=16000000UL -mmcu=atmega328p -c -o  
blink.o blink.c
```

Programación con Toolchain GCC para AVR

Construcción del programa y grabación en la memoria del μ C ATmega328 de la placa Arduino UNO:

1. Construcción del código fuente (.c) hasta la etapa de ensamblado, de la cual se obtiene un archivo objeto (.o):

```
avr-gcc -Os -DF_CPU=16000000UL -mmcu=atmega328p -c -o  
blink.o blink.c
```

2. Enlazado con bibliotecas, del cual se obtiene un archivo binario (.elf):
`avr-gcc -mmcu=atmega328p blink.o -o blink.elf`

Programación con Toolchain GCC para AVR

Construcción del programa y grabación en la memoria del μ C ATmega328 de la placa Arduino UNO:

1. Construcción del código fuente (.c) hasta la etapa de ensamblado, de la cual se obtiene un archivo objeto (.o):

```
avr-gcc -Os -DF_CPU=16000000UL -mmcu=atmega328p -c -o  
blink.o blink.c
```

2. Enlazado con bibliotecas, del cual se obtiene un archivo binario (.elf):
`avr-gcc -mmcu=atmega328p blink.o -o blink.elf`

3. Conversión del código binario (.elf) a formato objeto hexadecimal Intel (.hex)

```
avr-objcopy -O ihex -R .eeprom blink.elf blink.hex
```

Programación con Toolchain GCC para AVR

Construcción del programa y grabación en la memoria del μ C ATmega328 de la placa Arduino UNO:

1. Construcción del código fuente (.c) hasta la etapa de ensamblado, de la cual se obtiene un archivo objeto (.o):

```
avr-gcc -Os -DF_CPU=16000000UL -mmcu=atmega328p -c -o  
blink.o blink.c
```

2. Enlazado con bibliotecas, del cual se obtiene un archivo binario (.elf):
`avr-gcc -mmcu=atmega328p blink.o -o blink.elf`

3. Conversión del código binario (.elf) a formato objeto hexadecimal Intel (.hex)
`avr-objcopy -O ihex -R .eeprom blink.elf blink.hex`

4. Grabación en la memoria de programa del μ C

```
avrdude -F -V -c arduino -p ATMEGA328P -P /dev/ttyACM0  
-b 115200 -U flash:w:blink.hex
```


Programación de los puertos digitales – salidas (1)

- ▶ El código fuente `blink.c` es un ejemplo de manejo de un puerto digital de entrada/salida.
- ▶ Se utiliza el bit 5 del puerto B (`PB5`) configurado como salida para encender y apagar un LED conectado al pin correspondiente.
- ▶ La placa Arduino UNO incluye un LED conectado a dicho pin

Programación de los puertos digitales – salidas (1)

- ▶ El código fuente `blink.c` es un ejemplo de manejo de un puerto digital de entrada/salida.
- ▶ Se utiliza el bit 5 del puerto B (`PB5`) configurado como salida para encender y apagar un LED conectado al pin correspondiente.
- ▶ La placa Arduino UNO incluye un LED conectado a dicho pin

Configuración del pin `PB5` como salida:

```
DDRB |= _BV(DDB5); /* Equiv. a DDRB = DDRB | _BV(DDB5); */
```

Programación de los puertos digitales – salidas (1)

- ▶ El código fuente `blink.c` es un ejemplo de manejo de un puerto digital de entrada/salida.
- ▶ Se utiliza el bit 5 del puerto B (PB5) configurado como salida para encender y apagar un LED conectado al pin correspondiente.
- ▶ La placa Arduino UNO incluye un LED conectado a dicho pin

Configuración del pin PB5 como salida:

```
DDRB |= _BV(DDB5); /* Equiv. a DDRB = DDRB | _BV(DDB5); */
```

que usa la macro `_BV()` definida como:

```
#define _BV(bit) (1 << (bit))
```

y la constante simbólica DDB5 que vale 5.

Programación de los puertos digitales – salidas (1)

- ▶ El código fuente `blink.c` es un ejemplo de manejo de un puerto digital de entrada/salida.
- ▶ Se utiliza el bit 5 del puerto B (PB5) configurado como salida para encender y apagar un LED conectado al pin correspondiente.
- ▶ La placa Arduino UNO incluye un LED conectado a dicho pin

Configuración del pin PB5 como salida:

```
DDRB |= _BV(DDB5); /* Equiv. a DDRB = DDRB | _BV(DDB5); */
```

que usa la macro `_BV()` definida como:

```
#define _BV(bit) (1 << (bit))
```

y la constante simbólica DDB5 que vale 5. Por lo que la operación queda:

```
DDRB = DDRB | (1 << (5)); /* DDRB = DDRB | 0x40 */
```

Programación de los puertos digitales – salidas (1)

- ▶ El código fuente `blink.c` es un ejemplo de manejo de un puerto digital de entrada/salida.
- ▶ Se utiliza el bit 5 del puerto B (PB5) configurado como salida para encender y apagar un LED conectado al pin correspondiente.
- ▶ La placa Arduino UNO incluye un LED conectado a dicho pin

Configuración del pin PB5 como salida:

```
DDRB |= _BV(DDB5); /* Equiv. a DDRB = DDRB | _BV(DDB5); */
```

que usa la macro `_BV()` definida como:

```
#define _BV(bit) (1 << (bit))
```

y la constante simbólica DDB5 que vale 5. Por lo que la operación queda:

```
DDRB = DDRB | (1 << (5)); /* DDRB = DDRB | 0x40 */
```

que pone a 1 el bit 5 del registro DDRB.

Programación de los puertos digitales – salidas (1)

Luego en el bucle principal se enciende y apaga el LED cada 1000ms, poniendo a 1 y a 0 respectivamente el bit 5 del registro de datos del puerto B (**PORTB5**).

Programación de los puertos digitales – salidas (1)

Luego en el bucle principal se enciende y apaga el LED cada 1000ms, poniendo a 1 y a 0 respectivamente el bit 5 del registro de datos del puerto B (**PORTB5**).

El bit se pone a 1 con la línea:

```
PORTB |= _BV(PORTB5);
```

Programación de los puertos digitales – salidas (1)

Luego en el bucle principal se enciende y apaga el LED cada 1000ms, poniendo a 1 y a 0 respectivamente el bit 5 del registro de datos del puerto B (**PORTB5**).

El bit se pone a 1 con la línea:

```
PORTB |= _BV(PORTB5);
```

y se pone a 0 con la línea:

```
PORTB = PORTB & ~(1 << (5)); /* PORTB = PORTB & 0xBF; */
```

la cual utiliza el operador AND y NOT a nivel de bit. La constante simbólica **PORTB5** vale 5.

Programación de los puertos digitales – salidas (1)

Luego en el bucle principal se enciende y apaga el LED cada 1000ms, poniendo a 1 y a 0 respectivamente el bit 5 del registro de datos del puerto B (**PORTB5**).

El bit se pone a 1 con la línea:

```
PORTB |= _BV(PORTB5);
```

y se pone a 0 con la línea:

```
PORTB = PORTB & ~(1 << (5)); /* PORTB = PORTB & 0xBF; */
```

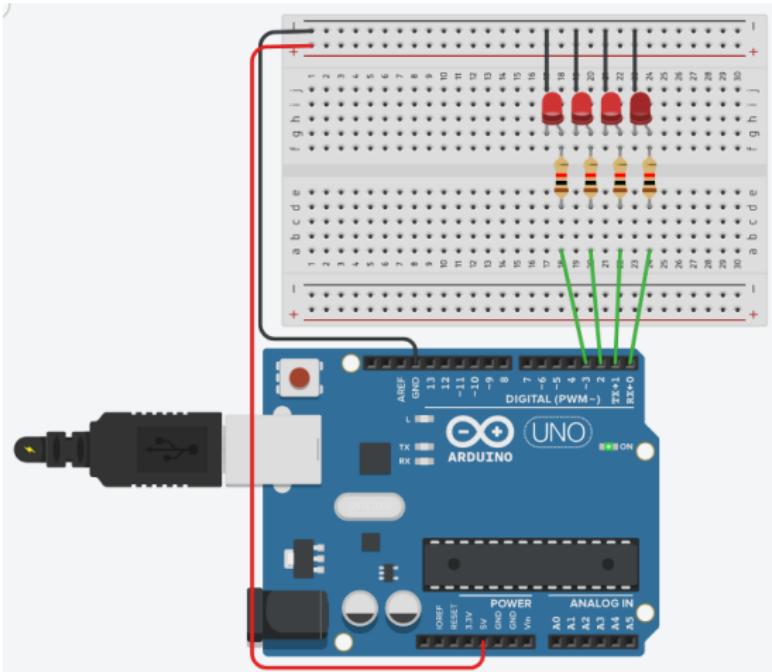
la cual utiliza el operador AND y NOT a nivel de bit. La constante simbólica **PORTB5** vale 5.

Otra forma de hacer lo mismo es usando el operador XOR, reemplazando el bucle principal por:

```
while(1)
{
    PORTB ^= _BV(PORTB5); /* Toggle LED */
    _delay_ms(BLINK_DELAY_MS);
}
```

Programación de los puertos digitales – salidas (2)

Circuito contador de 4 bits



Utiliza los 4 bits menos significativos del puerto D (PB0 a PB3).

Programación de los puertos digitales – salidas (2)

Archivo 'led_hex_counter.c'

```
1 #include <avr/io.h>
2 #include <util/delay.h>
3
4 #define LED_DDR DDRD
5 #define LED_PORT PORTD
6 #define LED_MASK 0x0F
7
8 #define DELAY_MS 500
9
10 int main (void)
11 {
12     unsigned char hex = 0;
13
14     /* Inicializa salidas digitales */
15     LED_DDR |= LED_MASK;
16
17     while(1)
18     {
19         LED_PORT &= ~(LED_MASK); // Apaga los LEDs
20         LED_PORT |= hex; // Muestra valor
21 }
```

Programación de los puertos digitales – salidas (2)

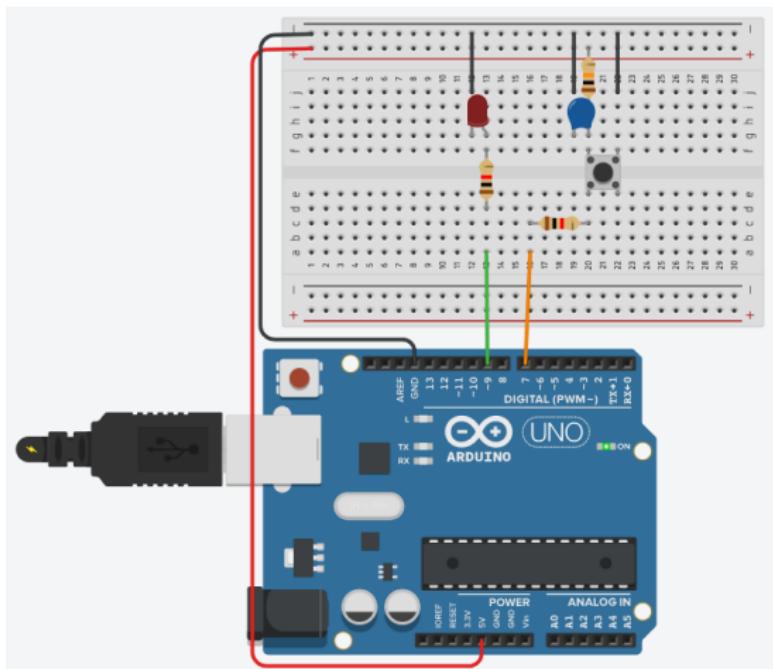
Archivo 'led_hex_counter.c' (cont.)

```
22     if(++hex > 16) // Controla rango máximo
23         hex = 0;
24
25     _delay_ms(DELAY_MS);
26 }
27 return 0;
28 }
```

Los LEDs muestran una cuenta binario de 4 bits, o sea del rango de valores que va desde $0d = 0000b = 0x00$ hasta $15d = 1111b = 0x0F$.

Programación de los puertos digitales – entradas

Circuito con pulsador y LED



El pulsador se conecta al pin 7 que es el bit 7 del puerto D (PD7) el cual debe configurarse como entrada y el LED se conecta al pin 9 que es el bit 1 del puerto B (PB1) que debe configurarse como salida.

Programación de los puertos digitales – entradas

Archivo 'led_sw.c'

```
1 #include <avr/io.h>
2 #include <util/delay.h>
3
4 /*
5  * LED: pin 9 --> PB1 (Salida)
6  * Sw: pin 7 --> PD7 (Entrada)
7  */
8 #define SW_MASK _BV(PIND7)
9 #define SW_PRES 0
10
11 int main(void)
12 {
13     uint8_t sw;
14
15     /* Configuración de entrada/salida digitales */
16     DDRB |= _BV(DDB1); /* LED como salida */
17     DDRD &= ~_BV(DDD7); /* Sw como entrada */
18     PORTB &= ~_BV(PORTB1); /* Apaga el LED */
19
```

Programación de los puertos digitales – entradas

Archivo 'led_sw.c' (cont.)

```
20 while(1)
21 {
22     /* Lee el valor del pulsador */
23     sw = (PIND & SW_MASK) >> PIND7;
24
25     if(sw == SW_PRES)
26     {
27         PORTB |= _BV(PORTB1); /* Enciende LED */
28         _delay_ms(1000);
29         PORTB &= ~_BV(PORTB1); /* Apaga LED */
30     }
31     _delay_ms(1);
32 }
33 return 0;
34 }
```

El programa lee el valor de la entrada digital conectada al pulsador y, en caso de estar presionado, encender el LED durante 1 seg. y luego apagarlo.

Programación de los puertos digitales – entradas

La constante simbólica `SW_MASK` se utiliza como máscara para la lectura de la entrada digital (línea 23).

Programación de los puertos digitales – entradas

La constante simbólica `SW_MASK` se utiliza como máscara para la lectura de la entrada digital (línea 23).

La línea que realiza la lectura de la entrada es:

```
sw = (PIND & SW_MASK) >> PIND7;
```

donde se utiliza el valor del registro `PIND` para la lectura del Puerto D, la que se enmascara con `SW_MASK`, o sea que se ponen a cero todos los bits menos aquel cuyo valor interesa (a esto se le denomina máscara).

Programación de los puertos digitales – entradas

La constante simbólica `SW_MASK` se utiliza como máscara para la lectura de la entrada digital (línea 23).

La línea que realiza la lectura de la entrada es:

```
sw = (PIND & SW_MASK) >> PIND7;
```

donde se utiliza el valor del registro `PIND` para la lectura del Puerto D, la que se enmascara con `SW_MASK`, o sea que se ponen a cero todos los bits menos aquel cuyo valor interesa (a esto se le denomina máscara).

Este valor luego se desplaza hacia la derecha para ocupar el bit menos significativo de la variable de 8 bits de nombre `sw`, que se utiliza luego en la estructura de selección `if` para determinar si hay que encender el LED.

Programación de los puertos digitales – entradas

La constante simbólica **SW_MASK** se utiliza como máscara para la lectura de la entrada digital (línea 23).

La línea que realiza la lectura de la entrada es:

```
sw = (PIND & SW_MASK) >> PIND7;
```

donde se utiliza el valor del registro **PIND** para la lectura del Puerto D, la que se enmascara con **SW_MASK**, o sea que se ponen a cero todos los bits menos aquel cuyo valor interesa (a esto se le denomina máscara).

Este valor luego se desplaza hacia la derecha para ocupar el bit menos significativo de la variable de 8 bits de nombre **sw**, que se utiliza luego en la estructura de selección **if** para determinar si hay que encender el LED.

La constante simbólica **SW_PRES** fija el valor que tendrá la entrada digital si el pulsador se encuentra presionado, lo cual dependerá de cómo este armado el circuito.

Programación de la UART – Configuración

La configuración de la USART consiste en:

- ▶ seleccionar el modo de funcionamiento asíncrono (UART),
- ▶ definir el tipo de trama,
- ▶ velocidad de comunicación (bps) y
- ▶ habilitar el transmisor y/o el receptor.

Programación de la UART – Configuración

La configuración de la USART consiste en:

- ▶ seleccionar el modo de funcionamiento asíncrono (UART),
 - ▶ definir el tipo de trama,
 - ▶ velocidad de comunicación (bps) y
 - ▶ habilitar el transmisor y/o el receptor.
-
- ▶ El tipo de trama queda determinada por la cantidad de bits de datos, y de parada y si se habilita o no la detección de errores mediante paridad y el tipo de paridad (par o impar).

Programación de la UART – Configuración

La configuración de la USART consiste en:

- ▶ seleccionar el modo de funcionamiento asíncrono (UART),
 - ▶ definir el tipo de trama,
 - ▶ velocidad de comunicación (bps) y
 - ▶ habilitar el transmisor y/o el receptor.
-
- ▶ El tipo de trama queda determinada por la cantidad de bits de datos, y de parada y si se habilita o no la detección de errores mediante paridad y el tipo de paridad (par o impar).
 - ▶ La configuración de la USART0 del μ C ATmega328 se utilizan mediante los registros de control y estado B y C, UCSR0B y UCSR0C.

Programación de la UART – Configuración

La configuración de la USART consiste en:

- ▶ seleccionar el modo de funcionamiento asíncrono (UART),
 - ▶ definir el tipo de trama,
 - ▶ velocidad de comunicación (bps) y
 - ▶ habilitar el transmisor y/o el receptor.
-
- ▶ El tipo de trama queda determinada por la cantidad de bits de datos, y de parada y si se habilita o no la detección de errores mediante paridad y el tipo de paridad (par o impar).
 - ▶ La configuración de la USART0 del μ C ATmega328 se utilizan mediante los registros de control y estado B y C, UCSR0B y UCSR0C.
 - ▶ El formato de trama suele indicarse por una combinación de dos números y una letra, p.e.: 8N1, 5E2, 7O1, etc.

Programación de la UART – Configuración

Definición de constantes simbólicas (antes de la función `main()`):

```
#define BAUD_RATE_4800 207
#define BAUD_RATE_9600 103
#define BAUD_RATE_38400 25
#define BAUD_RATE_57600 16
#define BAUD_RATE_115200 8
```

En la función `main`:

```
uint16_t ubrr = BAUD_RATE_115200;

/* Configuración el baud-rate */
UBRROH = (ubrr >> 8) & 0xFF; /* Byte más significativo */
UBRROL = ubrr & 0xFF; /* Byte menos significativo */

UCSROC = _BV(UCSZ00) | _BV(UCSZ01); /* UART con trama 8N1 */
```

La última línea equivale a escribir el valor $6d = 00000110b = 0x06$ en el registro.

Programación de la UART – Transmisión

Archivo 'hola_mundo.c'

```
1 #include <avr/io.h>
2 #include <util/delay.h>
3 #include <stdint.h>
4
5 #define BAUD_RATE_115200 8
6
7 int main()
8 {
9     int i;
10    uint16_t ubrr = BAUD_RATE_115200;
11    uint8_t dato[] = "Hola mundo\n";
12
13    /* Configuración el baud-rate */
14    UBRROH = (ubrr >> 8) & 0xFF; /* Byte más significativo */
15    UBRROL = ubrr & 0xFF; /* Byte menos significativo */
16
17    UCSROC = _BV(UCSZ00) | _BV(UCSZ01); /* UART con trama 8N1 */
18    UCSROB |= _BV(TXENO); /* Habilita el transmisor */
19
```

Programación de la UART – Transmisión

Archivo 'hola_mundo.c' (cont.)

```
20 while(1)
21 {
22     for(i = 0; dato[i] != '\0'; i++)
23     {
24         /* Espera a que el búfer de transmisión esté vacío */
25         while( !(UCSROA & _BV(UDRE0)) ) ;
26         UDR0 = dato[i]; /* Escribe el búfer de transmisión */
27     }
28     _delay_ms(500);
29 }
30 return 0;
31 }
```

Programación de la UART – Transmisión

Archivo 'hola_mundo.c' (cont.)

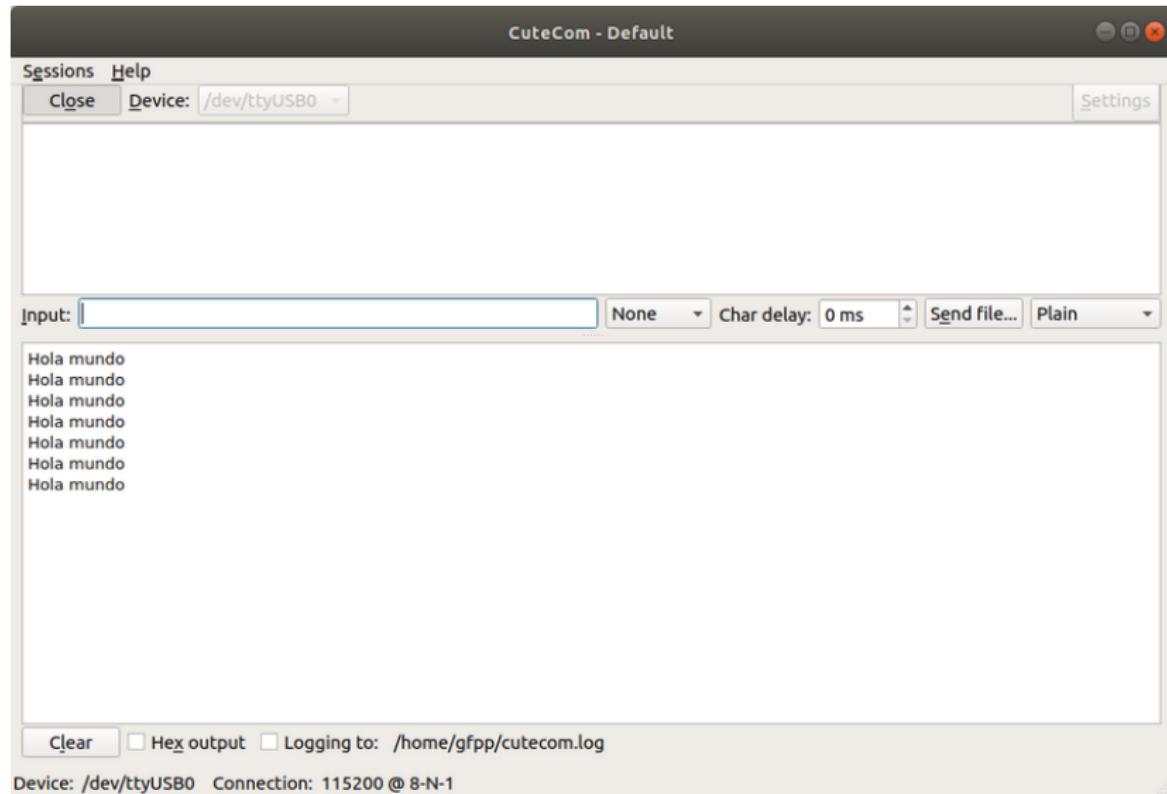
```
20 while(1)
21 {
22     for(i = 0; dato[i] != '\0'; i++)
23     {
24         /* Espera a que el búfer de transmisión esté vacío */
25         while( !(UCSROA & _BV(UDRE0)) ) ;
26         UDR0 = dato[i]; /* Escribe el búfer de transmisión */
27     }
28     _delay_ms(500);
29 }
30 return 0;
31 }
```

Se escribe el registro de transmisión de datos, UDR0, habiendo verificado que el búfer de transmisión está vacío. La línea:

```
while( !(UCSROA & _BV(UDRE0)) ) ;
```

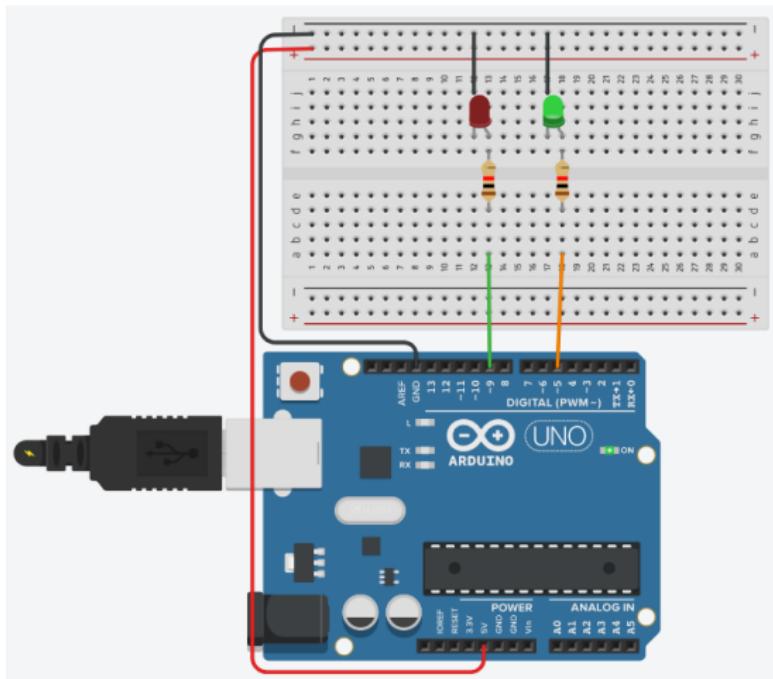
espera a que el búfer de transmisión este vacío.

Programación de la UART – Transmisión



Programación de la UART – Recepción

Circuito para la recepción de la UART



- ▶ El LED rojo, en PB1, cambia de estado al recibir el carácter 'r'
- ▶ El LED verde, en PD5, cambia de estado al recibir el carácter 'v'

Programación de la UART – Recepción

Archivo 'uart_read.c'

```
1 #include <avr/io.h>
2 #include <stdint.h>
3
4 #define BAUD_RATE_115200 8
5
6 int main()
7 {
8     uint16_t ubrr = BAUD_RATE_115200;
9
10    DDRB |= _BV(DDB1); /* Led rojo, salida */
11    DDRD |= _BV(DDD5); /* Led verde, salida */
12
13    /* Configuración el baud-rate */
14    UBRROH = (ubrr >> 8) & 0xFF; /* Byte más significativo */
15    UBRROL = ubrr & 0xFF; /* Byte menos significativo */
16
17    UCSROC = _BV(UCSZ00) | _BV(UCSZ01); /* UART con trama 8N1 */
18    UCSROB |= _BV(RXENO); /* Habilita el receptor */
19
```

Programación de la UART – Recepción

Archivo 'uart_read.c' (cont.)

```
20 while(1)
21 {
22     /* Espera a recibir un dato */
23     while( !(UCSROA & _BV(RXC0)) ) ;
24
25     switch(UDR0) {
26         case 'r':
27         case 'R':
28             PORTB ^= _BV(PORTB1); /* Cambia estado de LED rojo */
29             break;
30
31         case 'v':
32         case 'V':
33             PORTD ^= _BV(PORTD5); /* Cambia estado de LED verde */
34             break;
35     }
36 }
37 return 0;
38 }
```

Programación de la UART – Recepción

En el bucle principal la línea:

```
while( !(UCSROA & _BV(RXCO)) ) ;
```

espera hasta recibir un carácter.

Programación de la UART – Recepción

En el bucle principal la línea:

```
while( !(UCSROA & _BV(RXCO)) ) ;
```

espera hasta recibir un carácter.

Luego se utiliza la estructura de selección **switch-case** para actuar sobre el LED adecuado dependiendo del carácter recibido.

Programación de la UART – Trans. y recep.

Archivo 'to_upper.c'

```
1 #include <avr/io.h>
2 #include <stdint.h>
3 #define BAUD_RATE_57600 16
4
5 int main()
6 {
7     uint16_t ubrr = BAUD_RATE_57600;
8     uint8_t car;
9
10    /* Configuración el baud-rate */
11    UBRROH = (ubrr >> 8) & 0xFF; /* Byte más significativo */
12    UBRROL = ubrr & 0xFF; /* Byte menos significativo */
13
14    UCSROC = _BV(UCSZ00) | _BV(UCSZ01); /* UART con trama 8N1 */
15    UCSROB |= _BV(TXENO); /* Habilita el transmisor */
16    UCSROB |= _BV(RXENO); /* Habilita el receptor */
17
```

Programación de la UART – Trans. y recep.

Archivo 'to_upper.c' (cont.)

```
18  while(1)
19  {
20      /* Espera a recibir un dato */
21      while( !(UCSROA & _BV(RXCO)) ) ;
22      car = UDR0; /* Lee carácter */
23
24      if( (car >= 'a') && (car <= 'z') )
25          car -= 32;
26
27      /* Espera a que el búfer de transmisión esté vacío */
28      while( !(UCSROA & _BV(UDRE0)) ) ;
29      UDR0 = car; /* Escribe el búfer de transmisión */
30  }
31  return 0;
32 }
```

Programación de la UART – Trans. y recep.

Archivo 'to_upper.c' (cont.)

```
18  while(1)
19  {
20      /* Espera a recibir un dato */
21      while( !(UCSROA & _BV(RXCO)) ) ;
22      car = UDR0; /* Lee carácter */
23
24      if( (car >= 'a') && (car <= 'z') )
25          car -= 32;
26
27      /* Espera a que el búfer de transmisión esté vacío */
28      while( !(UCSROA & _BV(UDRE0)) ) ;
29      UDR0 = car; /* Escribe el búfer de transmisión */
30  }
31  return 0;
32 }
```

O bien utilizando funciones de la biblioteca estándar de C, como:

```
if( isalpha(car) )
    car = toupper(car);
```

para lo cual es necesario incluir el archivo de cabecera `ctype.h`.

