

Sesión 02: Ejecutando C

Programación para Sistemas

Ángel Herranz

2019-2020


Universidad Politécnica de Madrid

Recordatorio

- ¿Cómo van esas instalaciones de Ubuntu?
- ¿Cómo va Bash? (**pwd**, **ls**, **cd**, **mkdir**, ...)
- ¿Y C? (**gcc**, **./a.out**)
- ¿Cómo va el repaso de las transparencias?
- ¿Cómo van las hojas de ejercicios? (¡hay material extra!)
- ¿Cómo van esos accesos a triqui? (**ssh**)

- *Learning by doing.*

- *Learning by doing.*
- Mientras no tengamos sitio en las aulas informáticas,
- yo seré vuestras manos,

 leed con cuidado cada una de las líneas que aparecen en la pantalla,

cualquier cosa que no se entienda, duda,
sugerencia, prueba que queráis hacer, ...


- *Learning by doing.*
- Mientras no tengamos sitio en las aulas informáticas,
- yo seré vuestras manos,

⚠ leed con cuidado cada una de las líneas que aparecen en la pantalla,

cualquier cosa que no se entienda, duda,
sugerencia, prueba que queráis hacer, ...

¡Paradme!

Hola ángel i

-  Escribir un programa en C (hola.c) que saque por la salida estándar "¡Hola NOMBRE!", donde NOMBRE va a ser un argumento de la línea de comandos. Es decir, el programa tiene que hacer esto:

```
$ ./hola ángel  
¡Hola ángel!
```

Si el programa se invoca sin argumentos:


```
$ ./hola  
¡Hola mundo!
```

Hola ángel ii

```
#include <stdio.h>
int main(int argc,
        char *argv[]) {
    char *quien;
    if (argc == 1) {
        quien = "mundo";
    }
    else {
        quien = argv[1];
    }
    printf("¡Hola %s\n",
        quien);
    return 0;
}
```

- **argc**: Números de argumentos con el que se invoca el programa desde la línea de comandos.
- argc siempre es mayor que 0 porque el nombre del programa se considera un argumento.
- **argv**: Argumentos del programa (*array de strings*).
- argv[0]: **Nombre** con el que se invoca el programa.
- argv[1]: Primer argumento.
- argv[2]: Segundo argumento.

Factorial i

-  Escribir un programa que imprima en la salida estándar el factorial de un número que se le pasa desde la línea de comandos.

Ejemplos de uso:

```
$ ./fact 1
```

```
1
```

```
$ ./fact 2
```

```
2
```

```
$ ./fact 3
```

```
6
```

```
$ ./fact 4
```

```
24
```


Factorial ii

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

int factorial(int n) {
    int f = 1;
    int i;
    for(i = 1; i <= n; i++) {
        f *= i;
    }
    return f;
}

int main(int argc, char *argv[]) {
    int n;
    if (argc == 1)
        n = 0;
    else
        n = atoi(argv[1]);
```

```
    if (n == 0) {
        fprintf(
            stderr,
            "Uso: fact N (N > 0)\n"
        );
        return 1;
    }
    else {
        printf("%i\n",
            factorial(n));
        return 0;
    }
}
```

Factorial ii


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

int factorial(int n) {
    int f = 1;
    int i;
    for(i = 1; i <= n; i++) {
        f *= i;
    }
    return f;
}

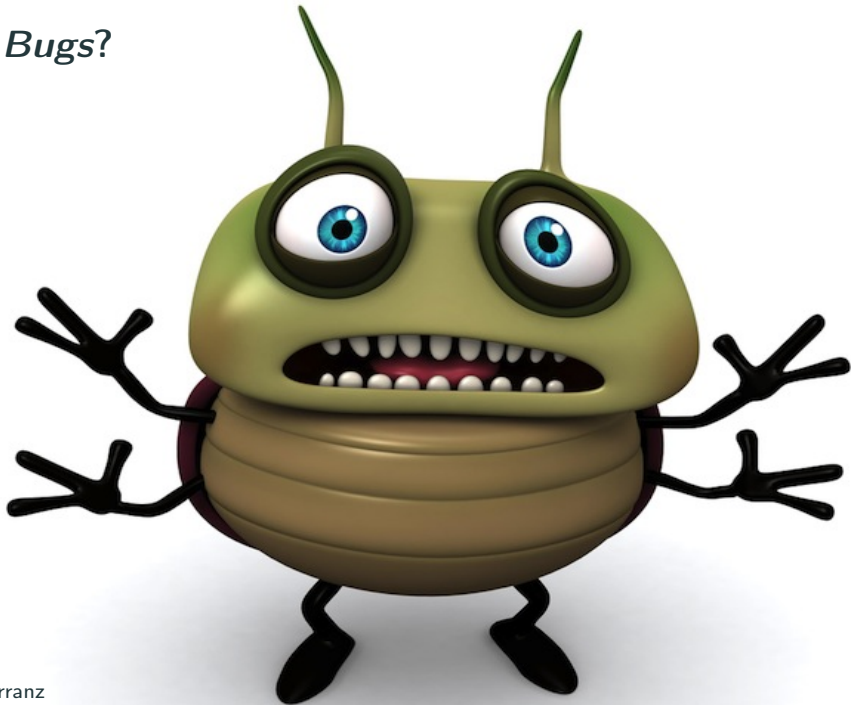
int main(int argc, char *argv[]) {
    int n;
    if (argc == 1)
        n = 0;
    else
        n = atoi(argv[1]);
```

```
if (n = 0) {
    fprintf(
        stderr,
        "Uso: fact N (N > 0)\n"
    );
    return 1;
}
else {
    printf("%i\n",
        factorial(n));
    return 0;
}
}
```

Hay al menos dos *bugs*

 Ejecutar el programa.

¿Bugs?



Depurador GDB

- *GNU Debugger*
- Un depurador sirve para depurar ;)
- Pero para nosotros, en esta asignatura

sirve para ver cómo se ejecuta mi programa.

Modelo Computacional

- Ejecución paso a paso
- Paquete gdb en Ubuntu:

```
$ sudo apt-get install gdb
```

GDB *crash course*

Compilación y arranque de GDB

- Para poder luego usar GDB, compilar con argumento **-g**:

```
$ gcc -g -o fact fact.c
```

```
$ ./fact 5
```

```
1
```

```
$ gdb fact
```

```
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
```

```
...
```

```
Reading symbols from fact...done.
```

```
(gdb)
```

- Depurador en marcha, esperando órdenes.
- El *prompt* ha cambiado a **(gdb)**.

Ejecución desde GDB i

- Para ejecutar el programa cargado (`./fact`) usamos la orden **run**:

```
(gdb) run
```

```
Starting program: /home/angel/...
```

```
1
```

```
[Inferior 1 (process 21938) exited normally]
```

```
(gdb) |
```

- Vemos que el resultado es sacar 1 en la salida estándar.

Ejecución desde GDB ii

- Para ejecutar el programa cargado pasándole 5 como parámetro:

```
(gdb) run 5
```

```
Starting program: /home/angel/...
```

```
1
```

```
[Inferior 1 (process 21938) exited normally]
```

```
(gdb) |
```

- Vemos que el resultado es 1 en vez de 120: algo va mal.

Breakpoints

- Podemos pedir a GDB que ponga *breakpoint* para que se pare cuando la ejecución llegue a ese punto.
- Usamos la orden *break* para que GDB se pare cuando comience a ejecutar la función main:

```
(gdb) break main
```

```
Breakpoint 1 at 0x55555555475b: file fact.c, line 16.
```

```
(gdb) run 5
```

```
Starting program: /home/angel/Asignaturas/pps/02/fact 5
```

```
Breakpoint 1, main (argc=2, argv=0x7fffffffef088) at ...
```

```
16         if (argc == 1) {
```

```
(gdb) |
```

- GDB informa de la *siguiente linea* que va a ejecutar.

Ejecución desde GDB paso a paso i

- En cada paso se puede explorar el valor de las variables y argumentos con la orden **print**:

```
(gdb) print argc
```

```
$1 = 2
```

```
(gdb) print argv[0]
```

```
$2 = 0x7fffffff402 "/home/angel/Asignaturas/pps/02/fact"
```

```
(gdb) print argv[1]
```

```
$3 = 0x7fffffff433 "5"
```

```
(gdb) print n
```

```
$4 = 0
```

Ejecución desde GDB paso a paso ii

- A partir del *breakpoint* podemos pedir a GDB que ejecute el programa paso a paso con la orden **step**:

```
(gdb) step
```

```
20          n = atoi(argv[1]);
```

```
(gdb) step
```

```
atoi (nptr=0x7fffffffef462 "5") at atoi.c:26
```

```
26      atoi.c: No such file or directory.
```

```
(gdb) |
```

- GDB ha intentado entrar en la implementación de `atoi` pero no tiene disponible el código fuente.
- Aún así, no falla, la ejecución paso a paso continúa.

Ejecución desde GDB paso a paso iii

- Con la orden **finish** vamos a pedir a GDB que directamente termine de ejecutar la función (atoi):

```
(gdb) finish
```

```
Run till exit from #0  atoi (nptr=0x7fffffffef462 ...  
0x000055555555477d in main (argc=2, argv=0x7ffffff...  
20          n = atoi(argv[1]);
```

```
Value returned is $5 = 5
```

```
(gdb) |
```

Ejecución desde GDB paso a paso iv

```
(gdb) step
```

```
22         if (n = 0) {
```

```
(gdb) print n
```

```
$6 = 5
```

```
(gdb) step
```

```
27         printf("%i\n", factorial(n));
```

```
(gdb) print n
```

```
$7 = 0
```

- ¿Después de ejecutar “**if** (n = 0) {” el valor de n es 0?

No es lo mismo == que =

Ejecución desde GDB paso a paso v

- Trucos: **r** en vez de **run**, **b** en vez de **break**, etc.
- Truco: **Enter** para ejecutar la última orden de nuevo.

 Continuar con la ejecución hasta encontrar el otro bug.

Algunas referencias a GDB

- Visitar <https://www.gnu.org/software/gdb/>
- *Debugging with GDB. The GNU Source-Level Debugger* (Richard Stallman, Roland Pesch, Stan Shebs, et al.):
 - <https://docs.freebsd.org/info/gdb/gdb.pdf> (v4)
 - <http://tierra.aslab.upm.es/~sanz/old/cursos/C1/gdb.pdf> (v9)
- Introduction to GDB a tutorial - Harvard CS50