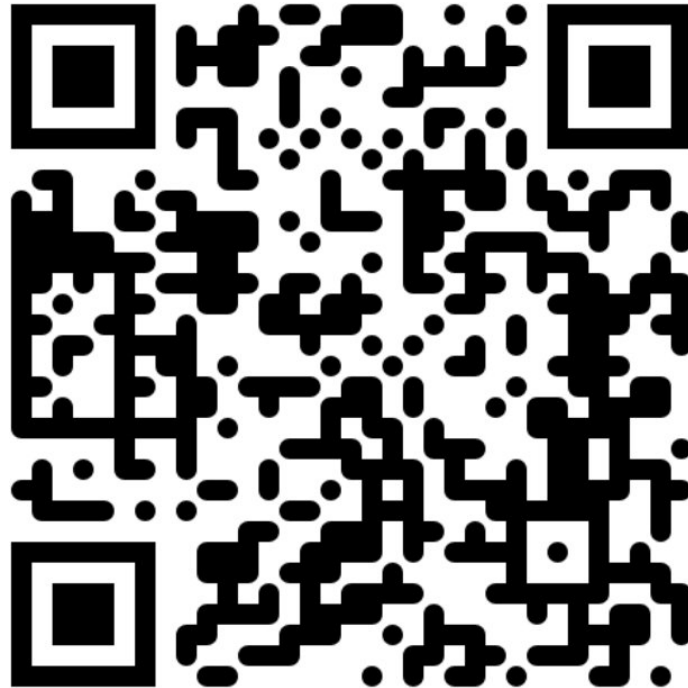


Ampliación de C para “normies”

Jorge Adrian Saghin Dudulea
@Z4na14

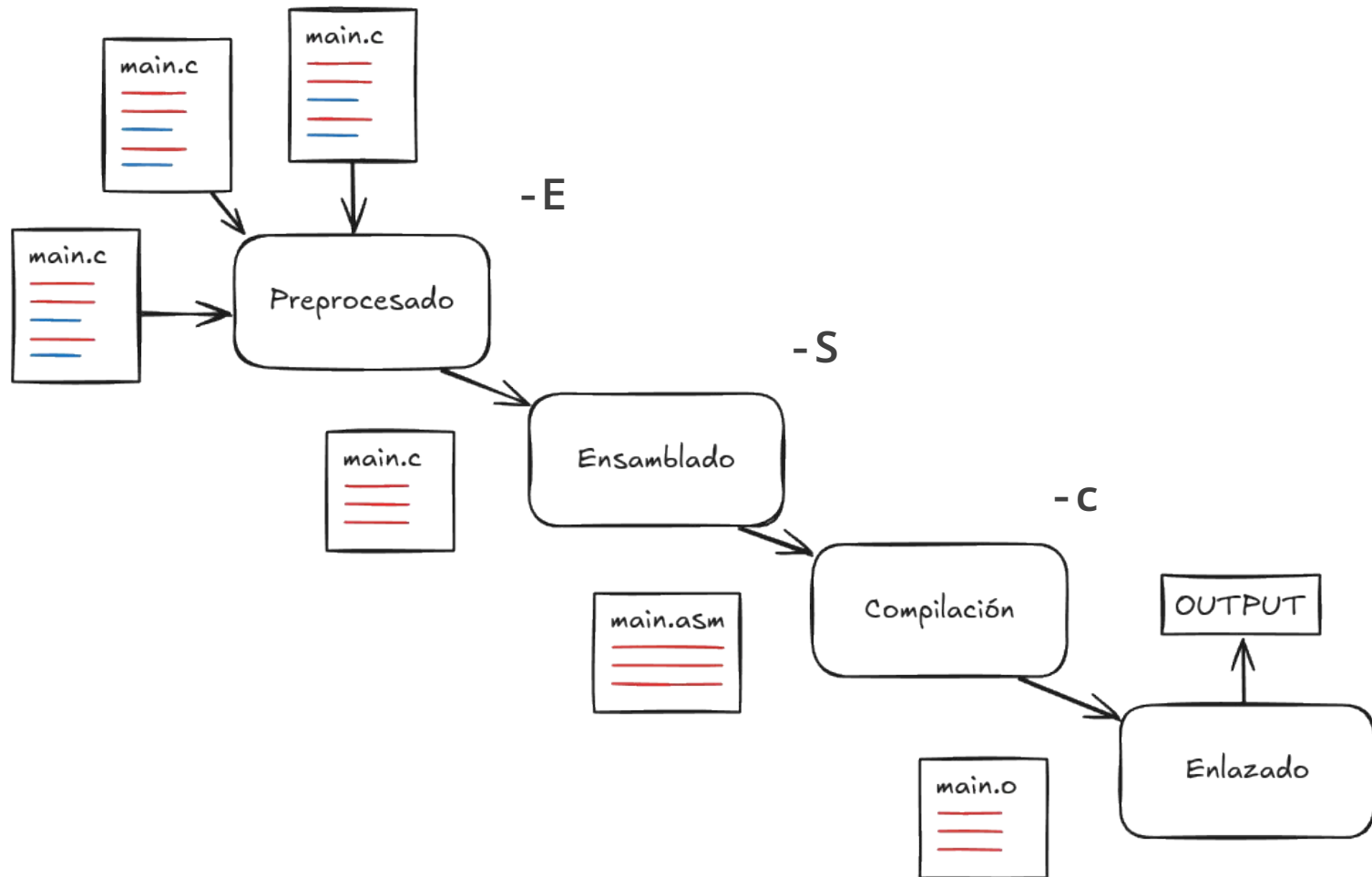
Grupo de Usuarios de Linux
@guluc3m | gul.uc3m.es

Transparencias



<https://github.com/Z4na14/C-para-normies>

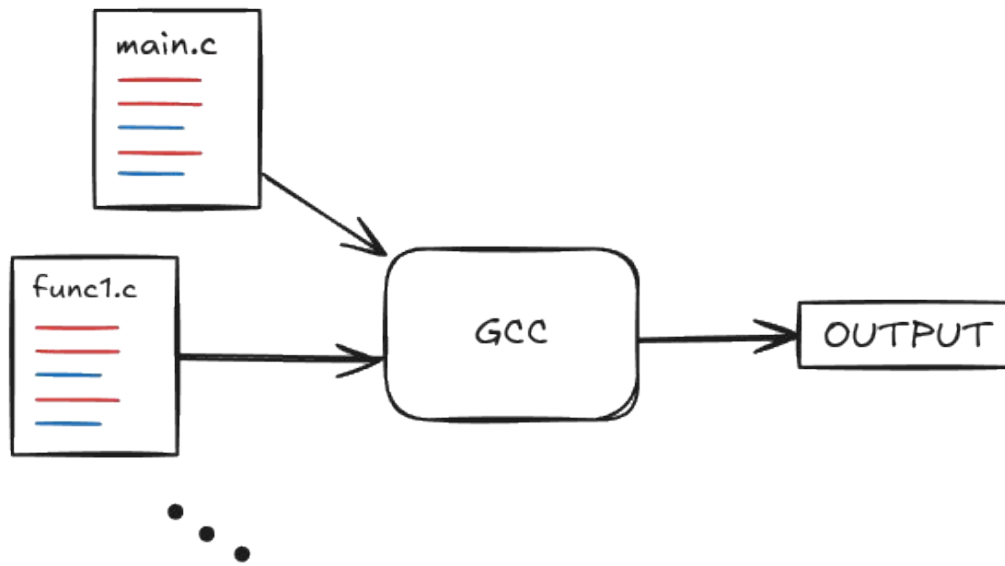
Compilación de archivos





Compilación de archivos

`gcc -o OUTPUT main.c func1.c ...`



Juntar todo el código fuente directamente en un ejecutable.

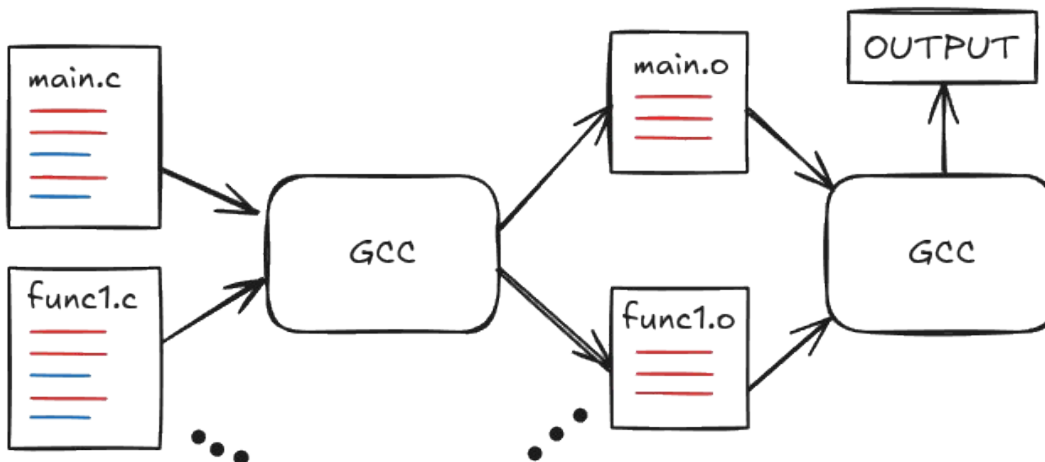
- Más fácil
- Mayor tiempo de compilación para proyectos grandes



Compilación de archivos

```
gcc -c main.c func1.c ...
```

```
gcc -o OUTPUT main.o func1.o ...
```



Compilar todos los archivos individualmente y enlazarlos luego en un segundo paso.

- Menor tiempo de compilación, solo se compilan los que han cambiado
- Útil solo si se compila lo que cambia



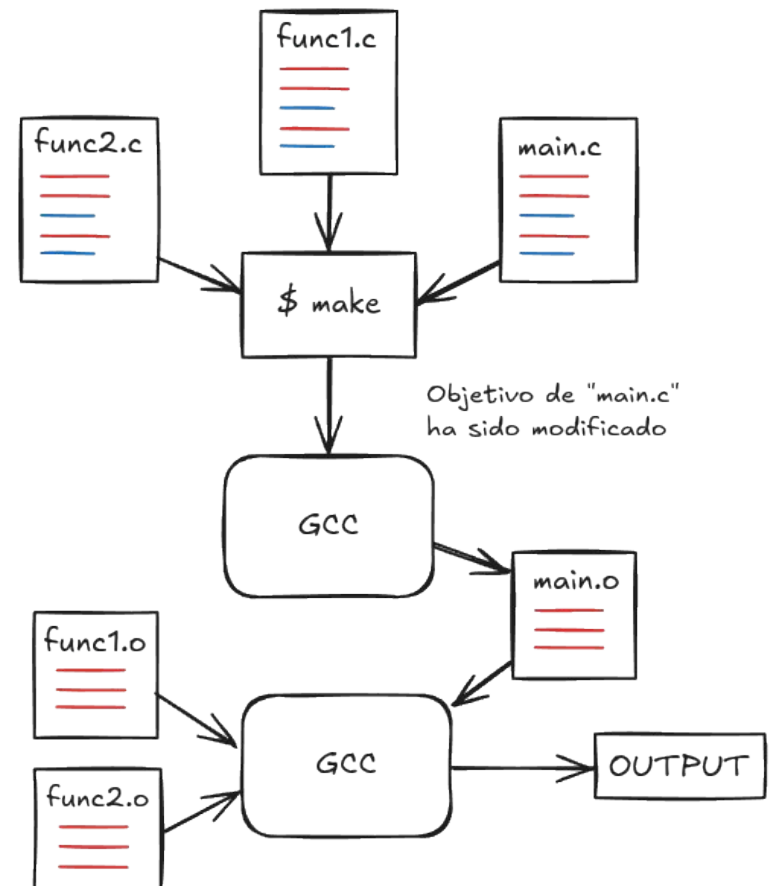
Compilación de archivos

¿Solución? -> Makefiles

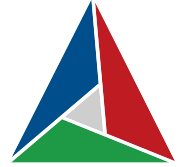
```
all : main.o func1.o func2.o
    cc -o program main.o func1.o func2.o
main.o : main.c library1.h
    cc -c main.c
func1.o : func1.c library1.h library2.h
    cc -c func1.c
func2.o : func2.c
    cc -c func2.c
clean :
    rm program main.o func1.o func2.o
```

gcc -MM *.c

-M : Todos los headers
-MM : Excluye los headers del sistema



Compilación de archivos



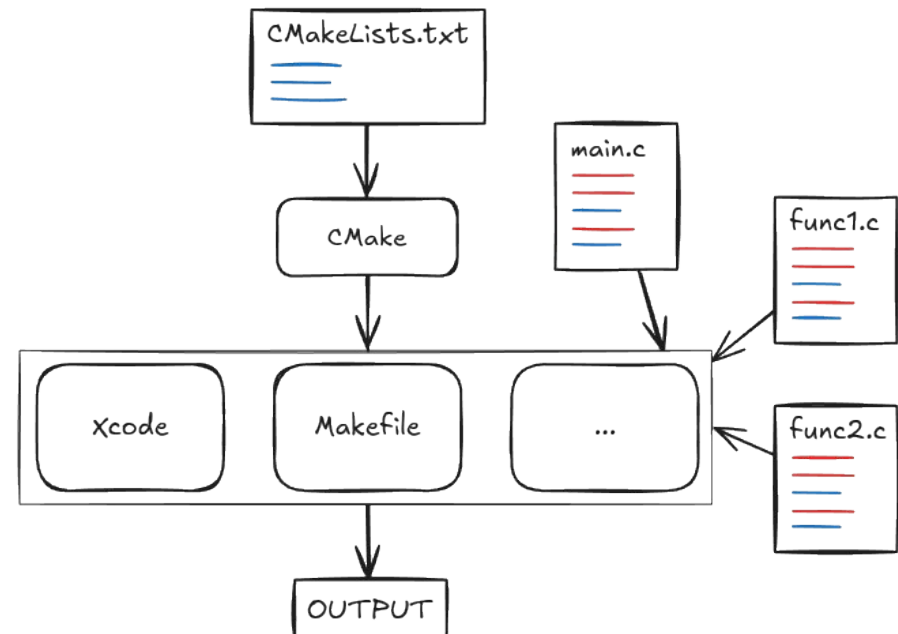
¿Solución a la solución? -> CMake

```
cmake_minimum_required(VERSION X.XX)
project(Skibidi-Toilet)

set(SRC_DIR ${PROJECT_SOURCE_DIR}/src)
set(INC_DIR ${PROJECT_SOURCE_DIR}/inc)

add_executable(${PROJECT_NAME}
    ${SRC_DIR}/main.c
    ${SRC_DIR}/func1.c
    ${SRC_DIR}/func2.c)

include_directories(${INC_DIR})
```



Tipos de datos primitivos

- Void
- Naturales
 - Byte: (8 BIT)
 - Short: (2 Bytes)
 - Int: (4 bytes)
 - Long: (6 bytes)
- Punto flotante
 - Float: (4 bytes)
 - Double: (8 bytes)
- Char: (1 byte)

Tipos de variables

- Locales: Dentro del scope en el que se declara.
- Globales: Fuera de cualquier bloque, solo llamables dentro del mismo archivo.
- Estáticas: Preserva su ubicación en memoria junto con su valor durante toda la duración del programa. ``static``
- Constantes: variables inmodificables ``const``
- Externa: la variable o la función esta definida en otro archivo. ``extern``

Cuidado con las
definiciones tentativas



Casting implícito

Ocurre durante el uso de operadores con operandos de distinto tipo.

Norma general: El operando de menor precisión toma el tipo del operador de mayor precisión.

Ej.

5 * 3.1416 -> Se opera con el 5 pasado a decimal

Casting explícito

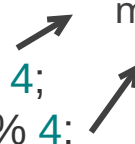
(tipo de dato) expresión

Se puede transformar manualmente una expresión al tipo de dato correspondiente.

Ej.

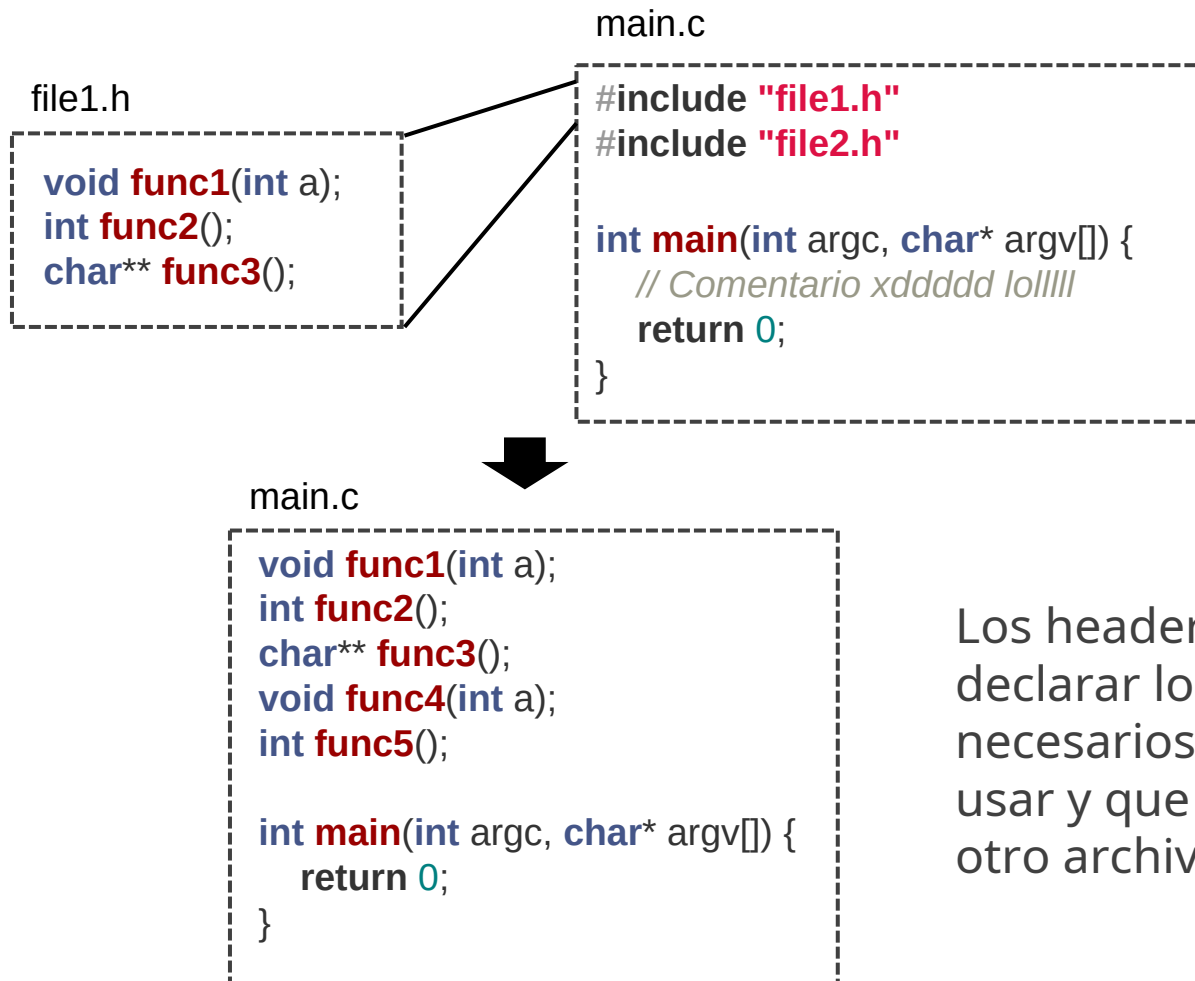
```
int a = (int) 5.5 % 4;  
int b = ((int) 5.5) % 4;  
printf("%d\n", (int) 'a');
```

Cuidado con los
paréntesis, no son lo
mismo



Directivas de preprocesador

#include



Los headers sirven para declarar los elementos necesarios que se vayan a usar y que estén definidas en otro archivo.

Directivas de preprocesador

#define

main.c

```
#include <stdio.h>
#define PI 3.141518
#define DOUBLE(x) x*x

int main(int argc, char* argv[]) {
    printf("Pi al cuadrado es: %d\n", DOUBLE(PI));
    return 0;
}
```

#define != constantes

main.c

```
...
//Write formatted output to stdout.
//This function is a possible cancellation point and
therefore not marked with __THROW.
extern int printf (const char *__restrict __format, ...);
...

int main(int argc, char* argv[]) {
    printf("Pi al cuadrado es: %d\n", 3.141518*3.141518);
    return 0;
}
```

Se reemplazan las
ocurrencias por la
declaración definida



Directivas de preprocesador

#ifdef/endif

main.c

```
#include <stdio.h>
#define DEBUG // DEBUG = 1

void testFunction() {
    printf("This is a normal function.\n");

    #ifdef DEBUG
        printf("Debug Mode: Extra debugging
information.\n");
    #endif
}

int main() {
    testFunction();
    return 0;
}
```

Definir:

-D<NOMBRE>

Eliminar:

-U<NOMBRE>

Estructura del archivo principal

main.c

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

} Directivas como
headers o definiciones

```
int main(int argc, char *argv[]) {  
    printf("Hello, World!");  
    return 0;  
}
```

Código de ejecución de la
función

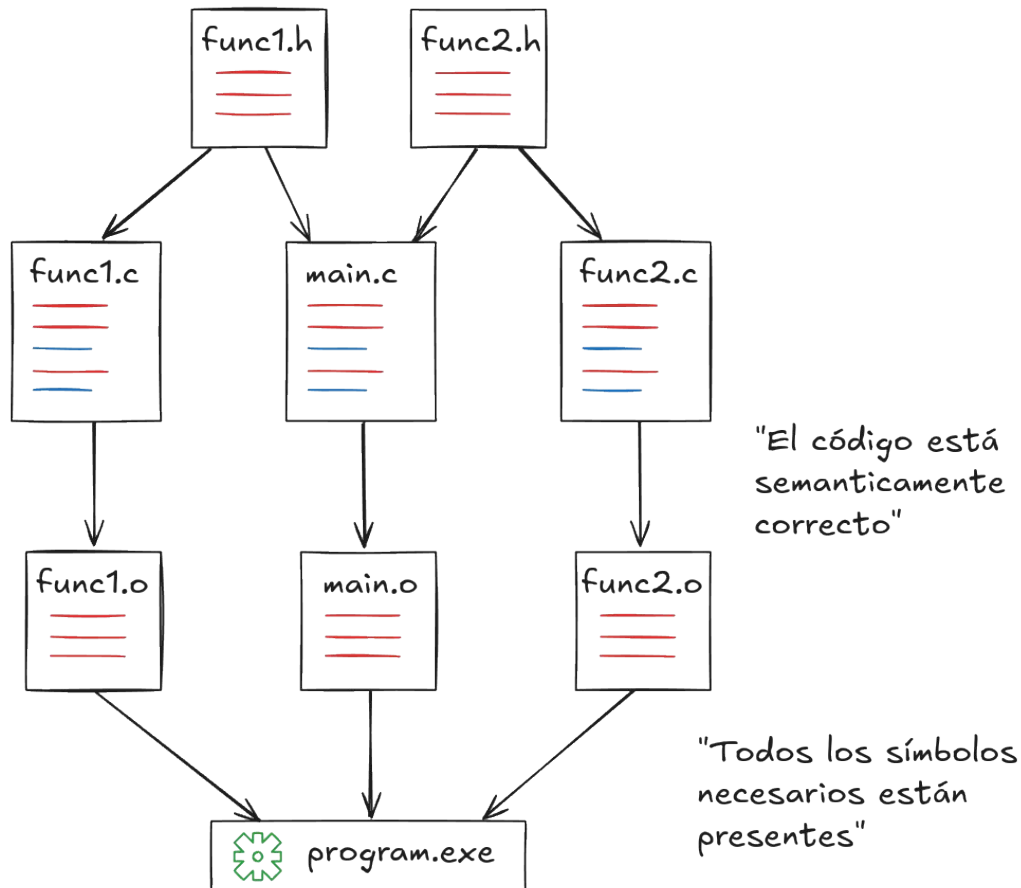
- 0: Ejecución satisfactoria
- !0: Código de error de la
aplicación

En el archivo principal debe ir la
mínima implementación. Todas
las funciones deben ser
completamente abstractas.

Estructura del archivo principal

Librerías

Declaración != Definición



Las librerías locales se ponen "entrecomilladas", y las globales del sistema entre <flechas>.

Estructura del archivo principal

Librerías

myheader.h

```
#ifndef MYHEADER_H
#define MYHEADER_H

void myFunction();

#endif
```

El uso de "include guards" es necesario para evitar la redeclaración y/o redifinición en el código fuente.

Otra forma es usar '#pragma once', pero este no está en el estandar de C

```
#include "myheader.h"

int myAnotherFunc()
{
    printf("Hello world\n");
    return 0;
}
```

func1.c

```
#include "myheader.h"

int main(int argc, char *argv[])
{
    myfunction();
    return 0;
}
```

main.c

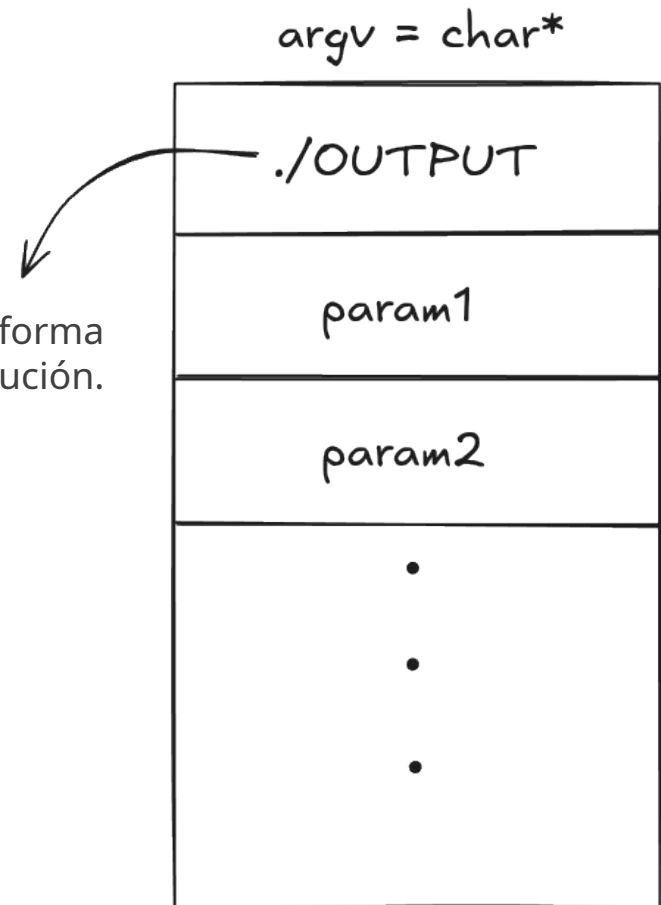
Estructura del archivo principal

argc y argv

0 1 2 = argc
./OUTPUT param1 param2 ...

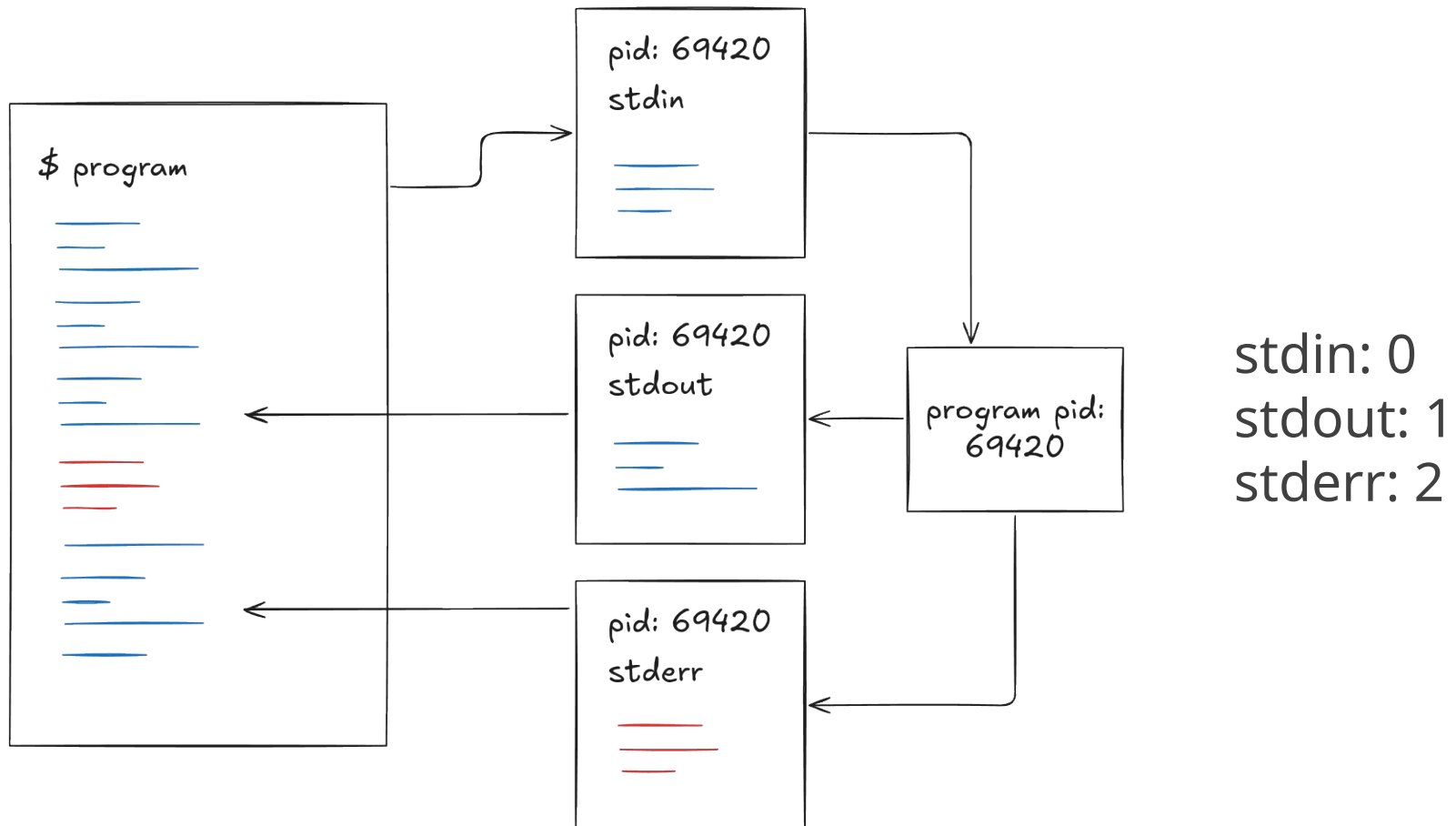
Se almacena de forma
literal su ejecución.

- argc almacena cuantos
argumentos se proporciona.
- argv almacena esos argumentos.



I/O

Streams



I/O

Streams

```
> cat io-test.c
File: io-test.c
1  #include <stdio.h>
2  #include <unistd.h>
3
4
5  int main(int argc, char *argv[])
6  {
7      printf("Hello world\n");
8
9      printf("%d\n", getpid());
10
11     int num = 0;
12     scanf("%d", &num);
13     printf("%d\n", num);
14
15     return 0;
16 }

> cat input.txt
File: input.txt
1  5

> ./io-test < input.txt > output.txt
> cat output.txt
File: output.txt
1  Hello world
2  6543
3  5
```

./programa > salida.txt
./programa >> salida.txt

./programa 1> salida.txt
./programa 2> error.txt

./programa < input.txt

./programa | grep "patrón"

Nota: ">" modifica y ">>"
agrega al final del archivo

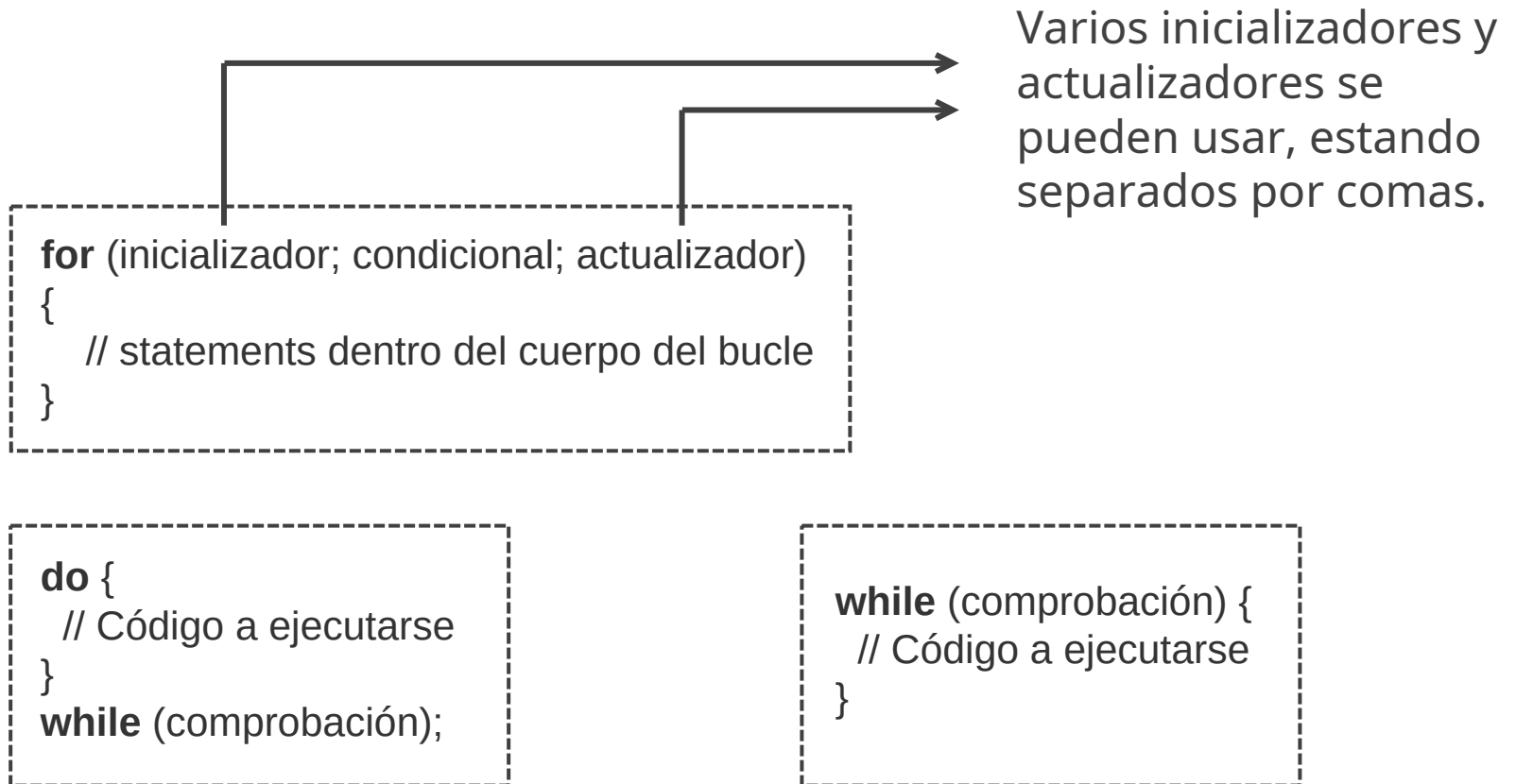
Control de flujo

Condiciones básicas

```
if (condition) {  
    // Código a ejecutarse si  
    // la condición es verdadera  
}  
else if (another condition) {  
    // Código a ejecutarse si  
    // otra condición es  
verdadera  
}  
else {  
    // Código a ejecutarse en  
    // caso de que no lo sea  
}
```

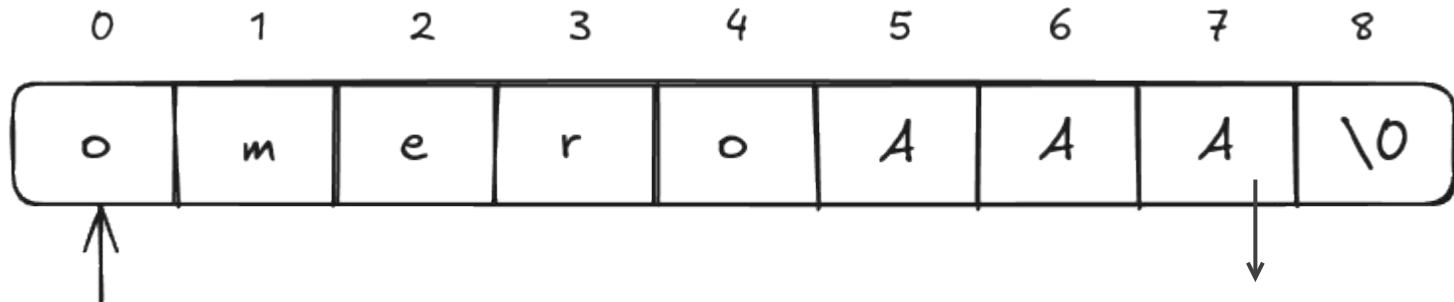
Declaraciones
anidadas
válidas

```
switch (expresión)  
{  
    case constante1:  
        // código  
        break;  
  
    case constante2:  
        // código  
        break;  
    .  
    .  
    default:  
        // Ninguno de los casos  
        // anteriores era verdadero  
}
```



Estructuras de datos

Arrays / Strings



char str[8]
0x69420A

El "\n" se usa para poner el salto de línea ya que hay emuladores que no lo ponen automáticamente. Si no se usa para imprimir, no hace falta ponerlo.

Strings = Arrays de caracteres con el terminador \0

<string.h>

char *strcat(char *dest, const char *src)
int sprintf(char *str, const char *format, argument-list)
int strcmp(const char *str1, const char *str2)

Concatenar strings
Introducir un nuevo valor
Comparar strings

Como asignarle nuevos valores a un string

`int sprintf(char *str, size_t n const char *format, argument-list)`

└──────────> Los strings siguen siendo arrays, hay que ir elemento por elemento

fgets() vs scanf()

`char *fgets(char *str, int n, FILE *stream)`

Lee hasta el `\n` o `n-1` caracteres

`int scanf(const char *format-string, argument-list);`

Busca las cadenas especificadas

Estructuras de datos

Estructuras

Las "clases" de C

```
struct nombre {  
    int contador;  
    char titulo[40];  
    float horas;  
};
```

La estructura se aloca completamente en memoria, por lo que podemos usar punteros.

s1.variable // Acceder directamente

// Dos formas de acceder con sus punteros

s1 -> variable
(*s1).variable

Sigue siendo un array

// Primera opción

```
struct nombre s1;  
s1.contador = 30;  
sprintf(s1.titulo, "OMERO");  
s1.horas = 3.45;
```

// Segunda opción

```
struct nombre s1 = {123, "say yes to affirm", 10.5};
```

// Tercera opción

```
struct nombre s1;  
s1 = (struct nombre) {12332423, "Ñ", 10.5};
```

Se puede hacer el casting explícito en más sitios

Estructuras de datos

Typedef

*Creación de tipos
más complejos*

```
typedef tipos_existentes nuevo_tipo;
```

Aliases que se usan principalmente en los headers para definir tipos mas complejos y mejorar la legibilidad.

```
typedef unsigned long int ULONG;  
typedef short int SHORT;
```

*// Los structs se benefician enormemente
// del typedef*

```
typedef struct mystruct {  
    ULONG a;  
    SHORT b;  
} STR;
```

Funciones:

```
typedef int (*operación)(int, int);
```

Arrays:

```
typedef int IntArray[5];
```

El nombre se usa como tipo.

Funciones

Estructuras como argumentos

Todo esto es también válido dentro de structs

Otras funciones

```
typedef <return type> (*<def name>)(<parameters>);  
void func(<def name> func1)
```

Las funciones en muchos statements acaban usandose implícitamente como punteros.

Arrays

```
void print(int m, int n, int arr[][n])
```

La keyword `static` también se puede usar en los parametros de tipo array:

```
void func(int foo[static 42]);
```

Número mínimo de elementos que el array debe tener.

Cuidado con la *degradación de arrays*. Al pasarlo dentro de una función acaba tratándose como un puntero a integers en vez de un array. El tamaño se pierde.

Funciones

Elipsis como argumentos

<stdarg.h>

```
int sum(int count, ...) {
    va_list args;
    int tot = 0;

    // El segundo argumento es
    // el último argumento antes
    // de la elipsis
    va_start(args, count);

    for (int i = 0; i < count; i++) {
        tot = tot + va_arg(args, int);
    }
    va_end(args);

    return tot;
}
```

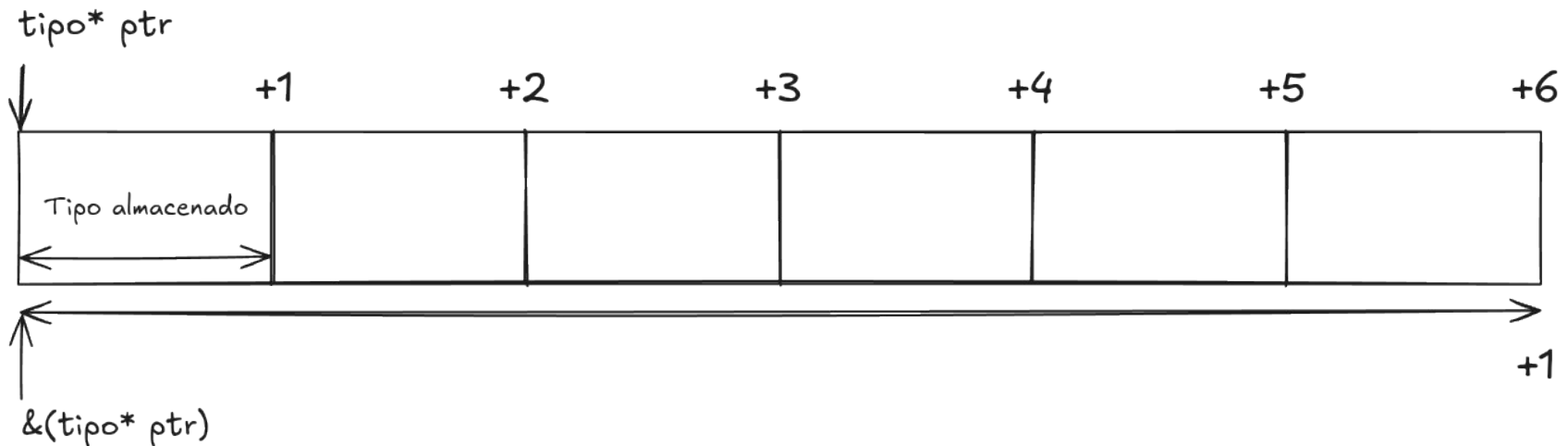
```
void va_start(va_list ap, parmN);
type va_arg(va_list ap, type);
void va_end(va_list ap);
void va_copy(va_list dest, va_list src);
```

Elipsis: Operador de parametros variables en una función (lo que usa *printf()*).

Primero se declara una lista variable, la inicializamos, y por cada llamada de función nos devuelve el siguiente argumento.

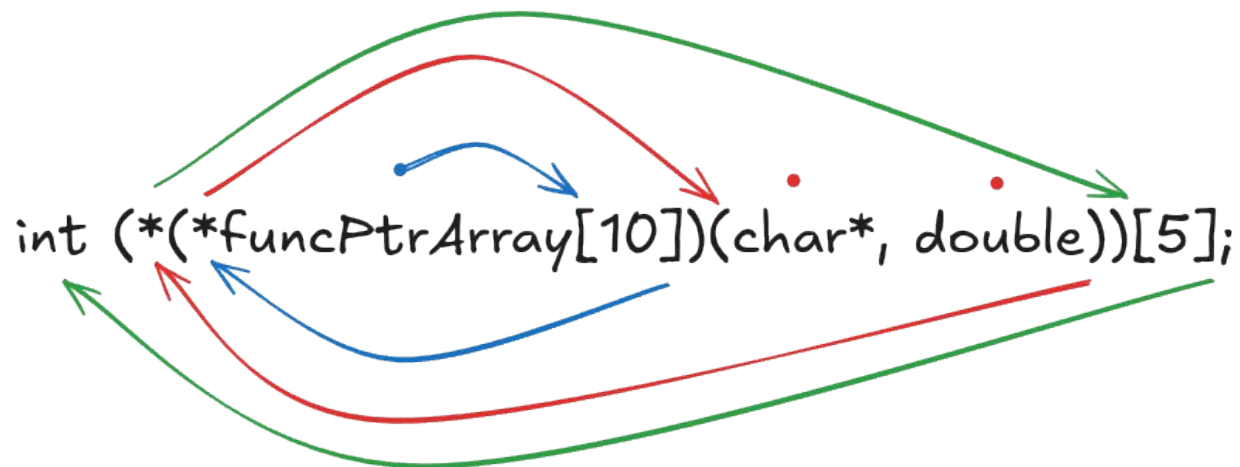
Punteros

Aritmética de punteros



Principalmente útil para sacar la longitud del array o iterar posiciones.

`*(&ptr+1) - ptr = sizeof(ptr) / sizeof(ptr[0])`



“funcPtrArray es un array de 10 elementos de tipo puntero a funciones que aceptan un puntero a un carácter y un double, que devuelven un puntero a un array de 5 elementos de tipo integer.”

- Primero a la derecha, leyendo la declaración en forma de espiral, de forma literal. Los paréntesis han de ser leídos completamente antes de pasar a una capa exterior.

Punteros

void* ptr

```
#include <stdio.h>

int main() {
    int num = 10;
    void *ptr = &num;

    printf("Value of num: %d\n", *(int *)ptr);
    return 0;
}
```

Hay que castear el puntero antes de dereferenciarlo

Sirve para almacenar el puntero de un tipo que no tienes claro que va a ocuparlo.

- En funciones también se suele usar para programación genérica.

```
void func(void *ptr, char type);
```

Memoria

Malloc family / Free

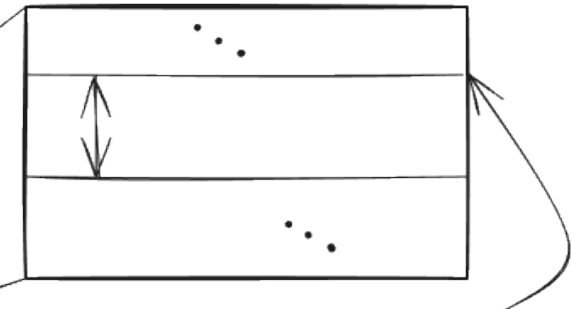
<stdlib.h>

Código a ejecutarse

Constantes y
variables globales

Memoria allocada
dinámicamente

Variables locales



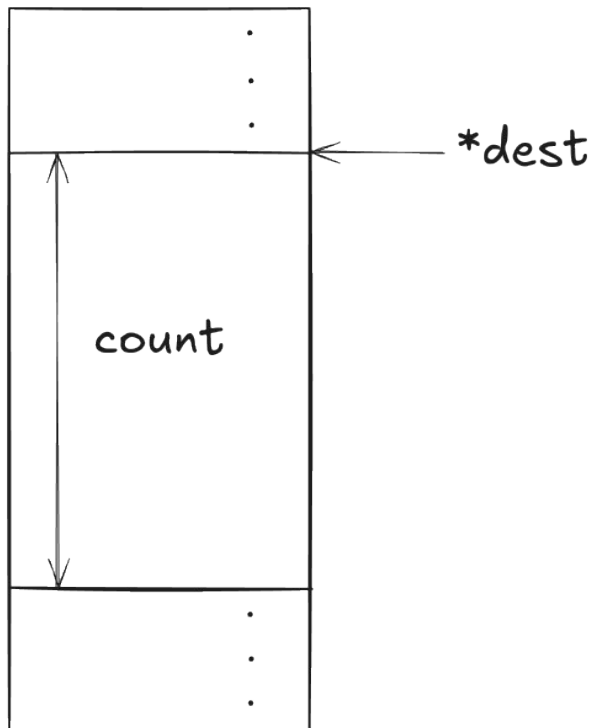
`malloc(TAMAÑO EN BYTES);`

El segmento DATA se maneja automáticamente, lo que se aloca manualmente **NO**
=> `free(puntero devuelto por malloc)`

Este también reserva el espacio ->
Puede haber contenido residual dentro

<stdlib.h>

```
void *memset(void *dest, int c, size_t count);
```

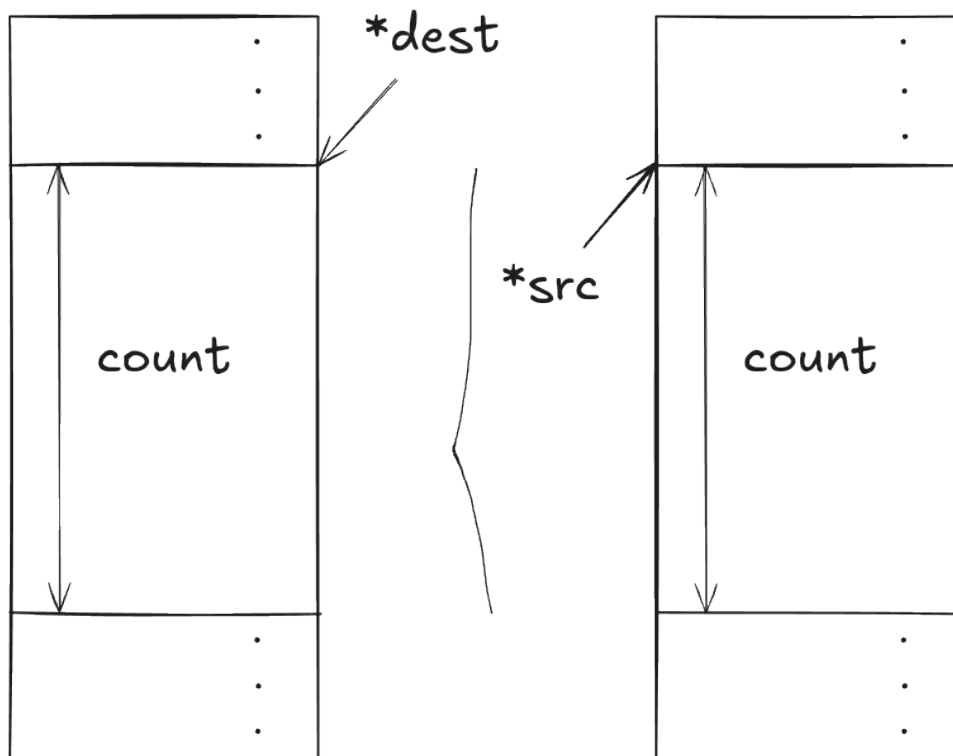


Se usa para rellenar la memoria con el argumento "c".

Útil para inicializar espacios de memoria alocados con malloc o para vaciar un array.

<stdlib.h>

```
void *memcpy(void *dest, const void *src, size_t count);
```



Copia literalmente el trozo de memoria al que se apunta, al destino que pasamos como argumento

Enlaces de interés

- [IBM](#)
- [acaldero/uc3m_c](#)
- [GCC reference](#)
- [cppreference.com](#)
- [GNU C reference manual](#)
- [The Open Group](#)



**C no es complicado,
solo juguetón,**

Gracias!

Grupo de Usuarios de Linux
@guluc3m | gul.uc3m.es

Cualquier duda le
espameáis a estos