Curso de programación en C moderno

(II Edición)

Neira Ayuso, Pablo Falgueras García, Carlos

Tema 5

C modular

Índice

Funciones

En C las funciones:

- Retornan un solo valor o nada (*void*)
- De cero a N parámetros
- Cada parámetro es de un tipo específico
- Todos los parámetros se pasan **por copia**
- Tienen una **declaración** y una **definición**
- Una función ha de estar declarada antes de ser llamada
- Una función no tiene por que estar definida a la hora de ser llamada

```c
#include <stdio.h>

/* Declaracion */
int f(int a, int b);

int main()
{
  /* Llamada */
  printf("%d\n", f(2, 3));

  return 0;
}

/* Definicion */
int f(int a, int b)
{
  return a + b;
}
```

Paso por copia

- **Siempre se copia** el parámetro (variable o constante) que se le pasa a la función al llamarla
- Dentro de la función se trabaja con la copia
- Las variables originales no se ven afectadas

```c
#include <stdio.h>

void f(int a)
{
   a = 33;
}

int main()
{
   int a = 3;
   f(a);
   printf("%d\n", a);

   return 0;
}
```

Paso por referencia

- Para poder modificar las variables originales dentro de una función, esta ha de trabajar con la **referencia** a la variable, no con la original
- Esto se consigue con **punteros**

```c
#include <stdio.h>

void f(int *a)
{
   *a = 33;
}

int main()
{
   int a = 3;
   f(&a);
   printf("%d\n", a);

   return 0;
}
```

Cabeceras

- Podemos crear nuestros propios ficheros **.h*
- Un fichero de cabecera suele tener únicamente declaraciones y no definiciones
- En el fichero de código (**.c*) se definen las funciones declaradas en la cabecera
- Podemos tener una cabecera y varios tipos de definiciones
- Nos permite crear una interfaz que aisle al usuario de la definición de las funciones

Ejemplo

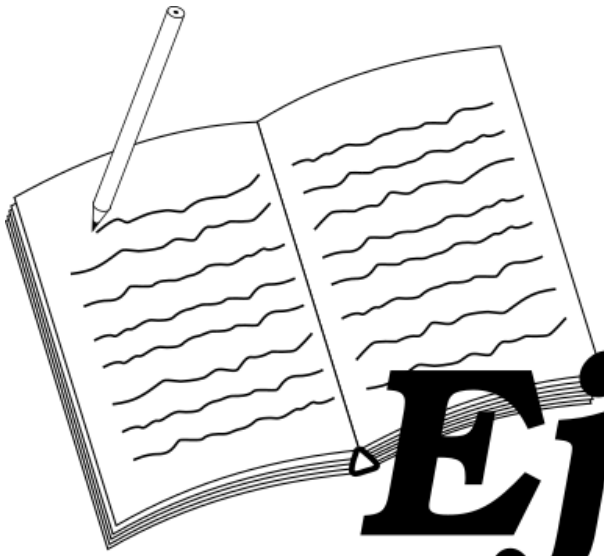
person.h

```
1  #ifndef _PERSON_H_
2  #define _PERSON_H_
3
4  #define MAX_NAME 256
5
6  struct person {
7    char name[MAX_NAME];
8    char surname[MAX_NAME];
9    unsigned int age;
10   int phone;
11 };
12
13 void print_person(const struct person *p);
14
15 #endif
```

person.c

```
1  #include <stdio.h>
2  #include "person.h"
3
4  void print_person(const struct person *p)
5  {
6    printf("%s, %s\n", p->surname, p->name);
7    printf("\t- age:   %d\n", p->age);
8    printf("\t- phone: %d\n", p->phone);
9  }
```

Ejercicios: Funciones y cabeceras



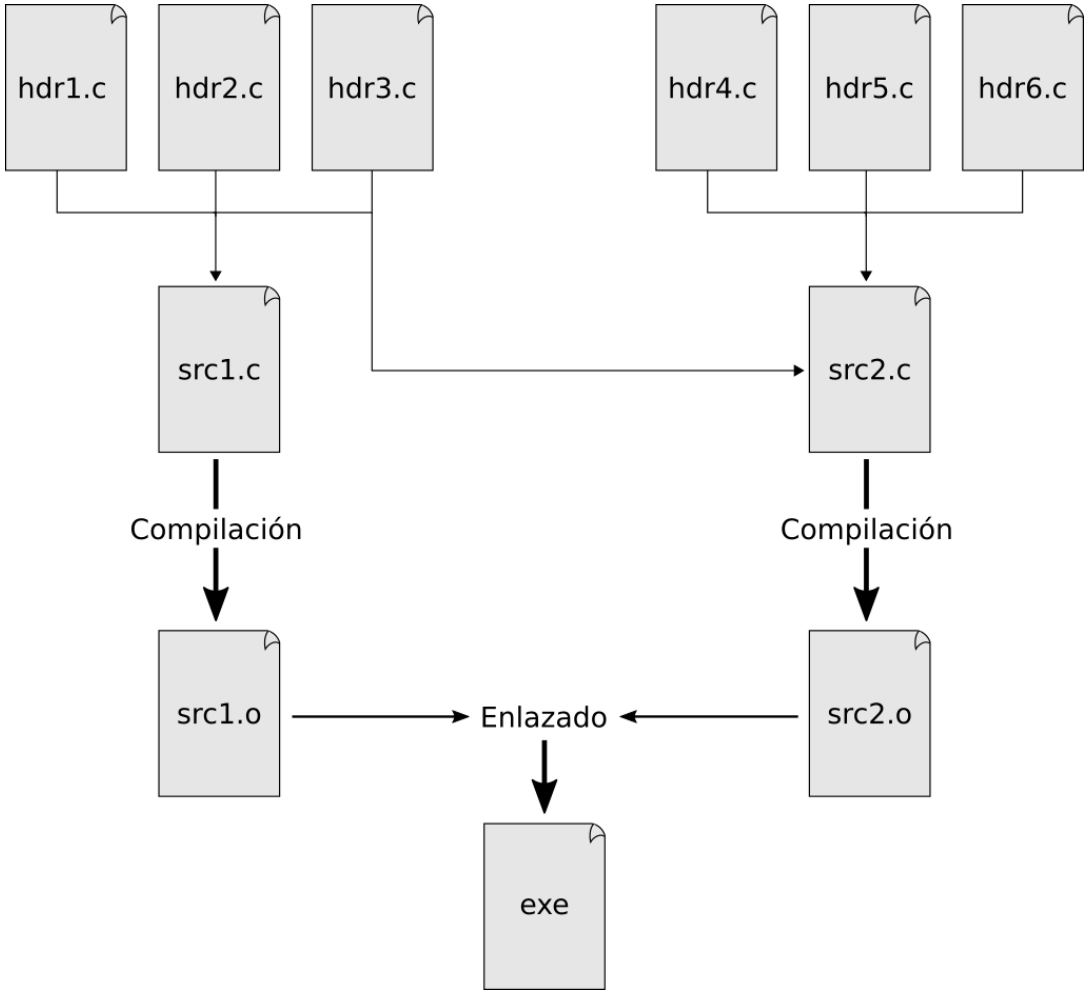
Ejercicios

Compilación por bloques

Razones:

- Tiempo:
  - La compilación es un proceso complejo y costoso
  - Proyectos muy grandes pueden tardar mucho tiempo en compilar
  - Cuando se está desarrollando esto se vuelve prohibitivo
- Organización:
  - Dividir el código en bloques lógicos es una buena práctica
  - Mejora:
    - El mantenimiento
    - La legibilidad
    - La portabilidad
    - La escalabilidad
    - etc

Compilación por bloques



Compilación por bloques

Solución:

- Cada fichero de código puede compilarse de manera independiente, creando un **fichero objeto** (**.o*)
- El **linker** se encarga de enlazar todos los ficheros objetos para crear el ejecutable
- Un módulo objeto puede tener referencias a símbolos definidos en otro módulo

Cómo se genera: `gcc -c persona.c`

Make y Makefile

Herramienta para automatizar la compilación del código (y mucho más)

- Gestiona dependencias para no compilar innecesariamente
- Facilita enormemente el trabajo
- También se suele utilizar para instalación y desinstalación

Estructura


```
objetivo: prerequisito1 prerequisito2 ...
  ordenes para generar "objetivo"
```

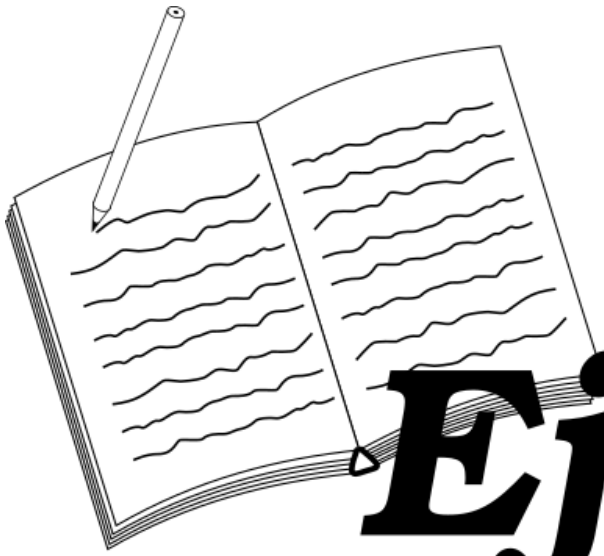
- **Objetivos**: Un objetivo suele ser un archivo que se desea generar (por ejemplo, un ejecutable)
- **Prerequisitos**: Lista de objetivos
- **Órdenes**: Instrucciones a realizar para generar el objetivo. Siempre van precedidas de una tabulación

Ejemplo

makefile

```
1  all: mi_prog
2
3  mi_prog: main.o persona.o
4    gcc main.o persona.o -o mi_prog
5
6  main.o: main.c
7    gcc -c main.c
8
9  persona.o: persona.h persona.c
10   gcc -c persona.c
```

Ejercicios: Makefile



Ejercicios