Tema 1: Paso de Parámetros y Proceso de Depuración

Introducción a la programación II

Ana Isabel Sierra de las Heras
Marcos Novalbos
Tiago Manuel Louro Machado de Simas
Rodrigo Alonso Solaguren-Beascoa
Alfonso Castro Escudero

Índice

Índice

Tema 1 – Apartado 1

1. Repaso de Punteros

1. Repaso Punteros

Array de tipo char


```
char Nombre  [3];

Nombre  [0] ='A';
Nombre  [1] ='n';
Nombre  [2] ='a';
```

Array de tipo char


```
char Nombre2  [6];

Nombre2  [0] ='A';
Nombre2  [1] ='n';
Nombre2  [2] ='a';
Nombre2  [3] ='b';
Nombre2  [4] ='e';
Nombre2  [5] ='l';
```

Puntero de tipo char


```
char *pNombre;

*(pNombre)  = 'J';
*(pNombre+1)  = 'o';
*(pNombre+2)  = 's';
*(pNombre+3)  = 'e';
*(pNombre+4)  = '\0';
*(pNombre+5)  = 'L';
*(pNombre+6)  = 'u';
*(pNombre+7)  = 'i';
*(pNombre+8)  = 's';
```

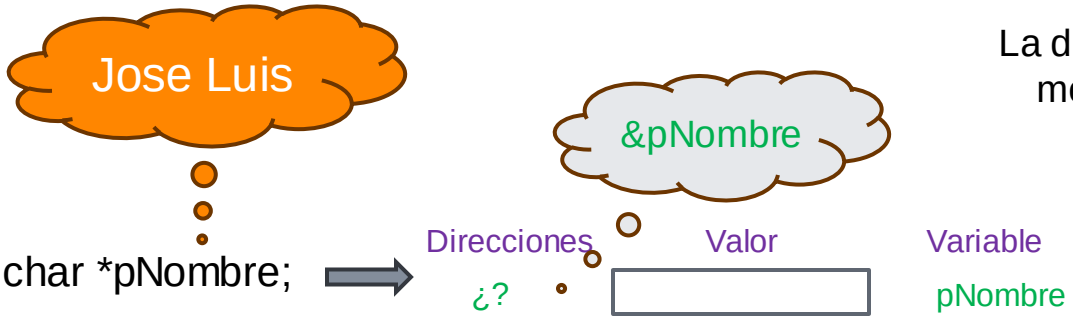
Un array es un tipo de datos con un tamaño estático. ¿Cómo tratar ejemplos en los que no se conoce el tamaño?

1. Repaso Punteros

Puntero de tipo char

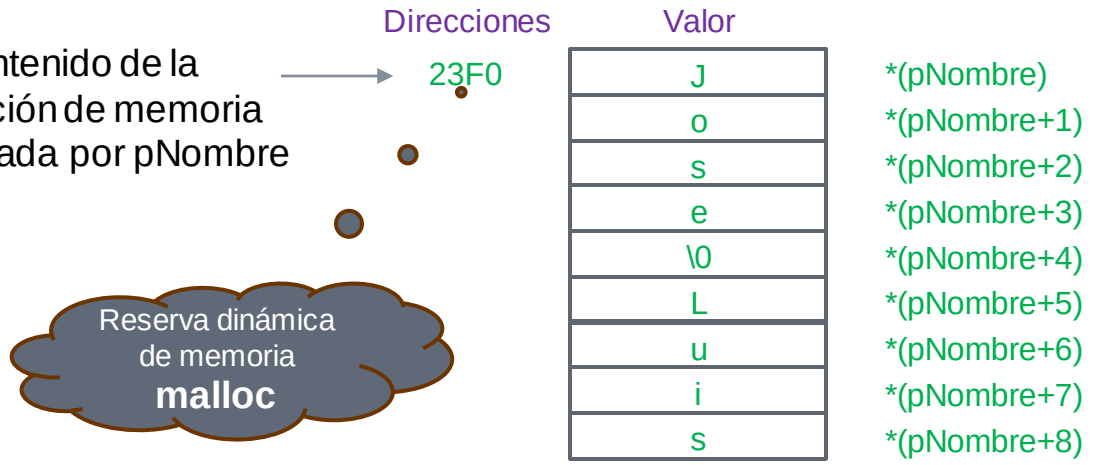
Declarar un puntero es reservar de memoria para guardar una dirección de memoria a partir de la cual va a haber algo

La dirección adecuada se completa en el momento que el puntero se inicializa




```
*(pNombre)  = 'J';
*(pNombre+1)  = 'o';
*(pNombre+2)  = 's';
*(pNombre+3)  = 'e';
*(pNombre+4)  = '\0';
*(pNombre+5)  = 'L';
*(pNombre+6)  = 'u';
*(pNombre+7)  = 'i';
*(pNombre+8)  = 's';
```

El contenido de la dirección de memoria apuntada por pNombre



1. Repaso Punteros

Programa que realice la suma de complejos

float sumaParteReal (float ParteRealNum1, float ParteRealNum2)
float sumaParteImg (float ParteIMGNum1, float ParteImgNum2)

¿Cómo hacerlo con una sola función?

Pasándole las direcciones donde va a dejar las partes reales e imaginarias de la suma y modificando en la función directamente los contenidos de las direcciones

UTILIZANDO PUNTEROS

void sumaComplejosI (float ParteRealNum1, float ParteRealNum2, float ParteImgNum1, float ParteImgNum2 float, *SumaParteReal, float * SumaParteImg)

1. Repaso Punteros. Variables puntero. Declaración

- Los punteros se declaran igual que las variables normales, pero con un asterisco (*) delante del nombre de las variables:

```
<tipo> * <identificador de puntero>;
```

Donde `tipo` hace referencia al tipo de datos al que "apuntará" nuestro puntero.

- Es muy importante respetar el tipo de datos al que apunta un puntero. Al igual que en el caso de las variables simples, un descuido en el tipo de datos generará errores.
- Se recomienda, para el nombre de la variable de tipo puntero, poner una 'p' que nos "recuerde" que se trata de un puntero.

```
double *pdoble;   /* Variable pdoble que es de tipo puntero a doble */
int *pentero; /* Variable pentero que es de tipo puntero a entero */
```

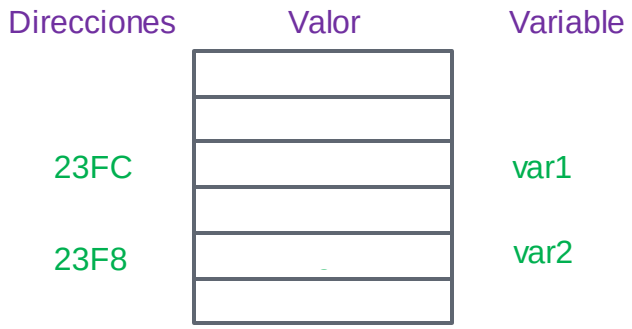
1. Repaso Punteros. Operadores de puntero

- Ejemplo Operador * :

…

```
int var1, var2;
```

…



1. Repaso Punteros. Operadores de puntero

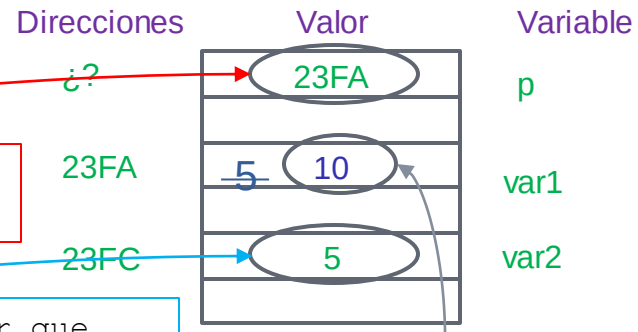
- Ejemplo Operador * :

```
…
int var1, var2;
int *p;
var1 = 5;
p = &var1;
var2 = *p;
*p =10;
…
```

El puntero p se inicializa a la dirección de la variable var1

La variable var2 toma el valor que contiene la dirección apuntada por p y que inicialmente valía 5.

El contenido de la dirección apuntada por el puntero p toma el valor 10

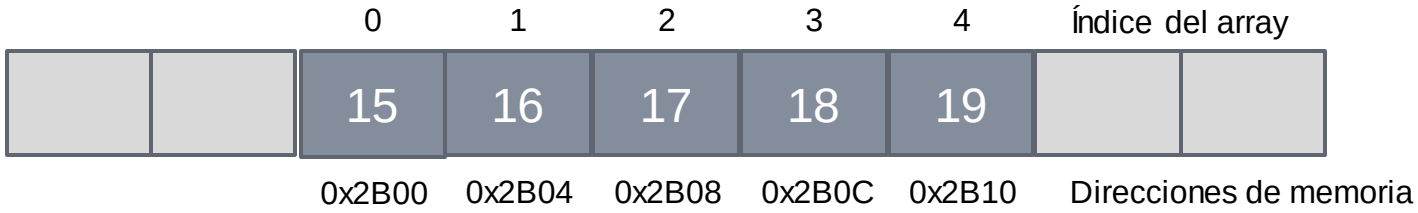


ATENCIÓN: Después de la asignación p = &var1; tenemos dos maneras de manipular los valores enteros almacenados en la variable var1: directamente mediante var1 o indirectamente mediante el puntero p.

1. Repaso Punteros. Arrays y punteros

- Un array se almacena en memoria de manera secuencial

```
int edades [5] = {15,16,17,18,19}
```



- Si tenemos un puntero `pedades` que apunte al array `edades[5]` se puede aplicar fácilmente la aritmética de punteros para hacer referencia al siguiente elemento del array.

`&edades[1]` sería equivalente a: `pedades+1` o `++pedeades`

`edades[1]` es equivalente a `*(pedades+1)` o `*(++pedades)`

1. Repaso Punteros. . Ejercicio

Crear un programa que copie una cadena en otra utilizando punteros.

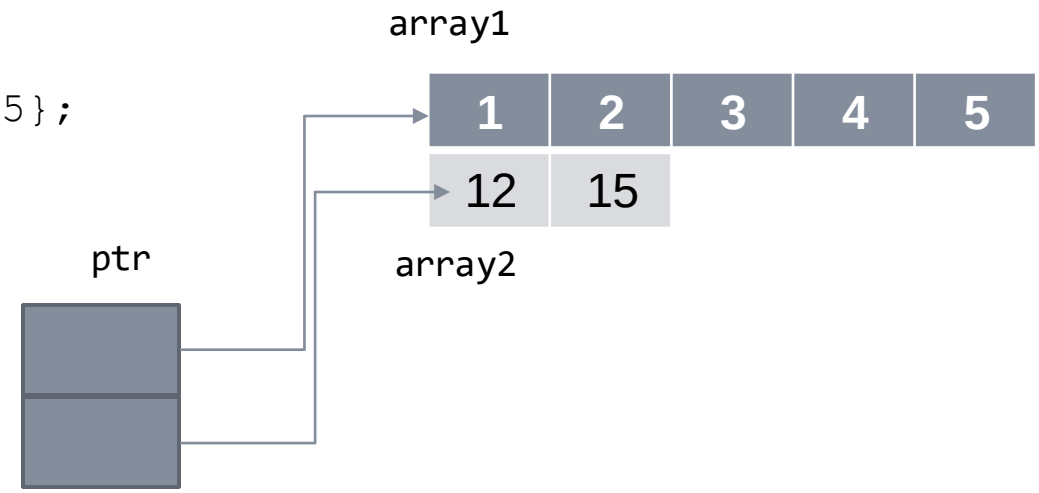
1. Repaso Punteros. Ejercicio

```
#include <stdio.h>
#define MAX_SIZE 20 // Maximum size of the string

void leeLinea(char texto[], int Tamanio);
void main()
{
    char text1[MAX_SIZE], text2[MAX_SIZE];
    char * str1 = text1; //cadena origen
    char * str2 = text2; //cadena destino
             int i = 0;

    //Entrada de la cadena
    printf("Introduce la cadena:\n");
    leeLinea(str1,MAX_SIZE);

    /* Copy text1 to text2 character by character */
             //while(*(str2++) = *(str1++));
             while (*(str1+i)!='\0'){
                          *(str2+i) = *(str1+i);
                          i++;
             }
    printf("Cadena origen = %s\n", text1);
    printf("Cadena destino = %s\n", text2);
}
```

```
void leeLinea(char texto[], int Tamanio){
     char newChar;
     int i=0;
     while (((newChar = getchar())!='\n')&&(i<Tamanio)){
          texto[i]=newChar; //cada caracter que ha leido se asigna a
las posiciones de la cadena
          i++;
          if (i==Tamanio){
               printf ("Has superado el tamanio\nVuelve a introducir
la cadena\n");
               while (getchar()!='\n'); //limpia Buffer
               i=0;
          }
     }
     texto[i]='\0'; //Con esto lo dejamos con un \0 al final
}
```

1. Arrays y punteros. Array de punteros

```
int array1 [5] = {1,2,3,4,5};
int array2 [2] = {12,15};
int *ptr [2];
ptr [0] = array1;
ptr [1] = array2;
```



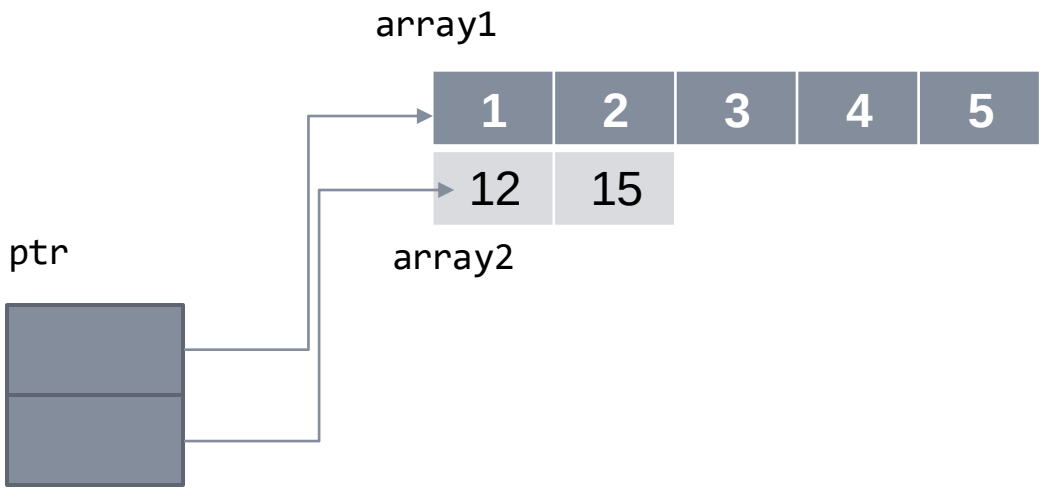
Que devolvería *ptr

Que devolvería *(*ptr)

1. Arrays y punteros. Array de punteros

Ilustremos un ejemplo:

```
int array1 [5] = {1,2,3,4,5};
int array2 [2] = {12,15};
int *ptr [2];
int *q;
int q1;

ptr [0] = array1;
ptr [1] = array2;
```



Que devolvería *ptr

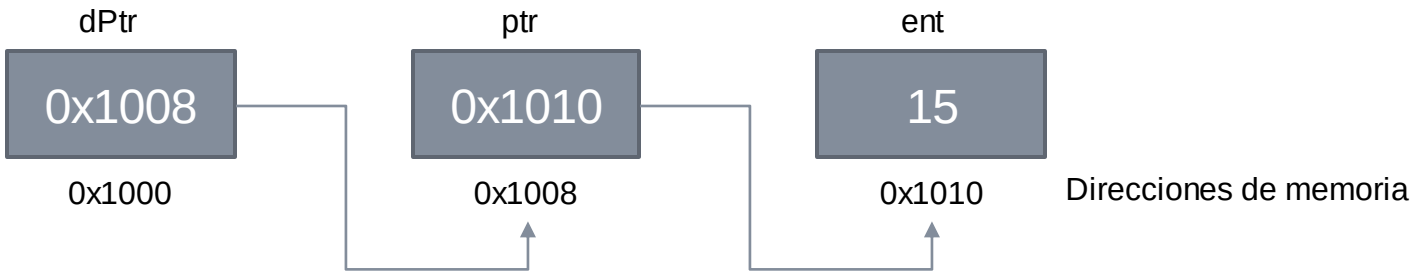
Que devolvería *(*ptr)

```
q=*ptr;          // array1
q1=*(*ptr);      // 1
```

1. Arrays y punteros. Punteros dobles

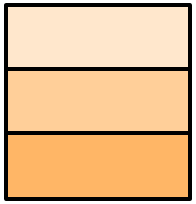
- Un puntero doble es aquel que apunta a otro u otros punteros.

```
int **dPtr;
```



Repaso puntero a puntero

```
1.  #include <stdio.h>
2.  int main()
3.  {
4.      int i;
5.      int *ptrToi;            /* Puntero a entero */
6.      int **ptrToPtrToi;      /* Puntero a puntero a entero */
7.
8.      ptrToPtrToi = &ptrToi; /* Puntero contiene dirección de puntero */
9.      ptrToi = &i;           /* Puntero contiene dirección de entero */
10.
11.     i = 10;                /* Asignación directa */
12.     *ptrToi = 20;          /* Asignación indirecta */
13.     **ptrToPtrToi = 30;    /* Asignación con doble indirección */
14.
15.     return 0;
16. }
```



Índice

Tema 1 – Apartado 2

2. Paso de Parámetros a través de la línea de comandos: argc y argv

1. Paso de parámetros a través de la línea de comandos

- Algunas veces resulta útil pasar información al programa cuando se ejecuta.
- El método general es pasar información a la función main() mediante el uso de argumentos de línea de comandos
- Un argumento de línea de comandos es la información que sigue al nombre del programa cuando lo ejecutamos.
- El fichero media_alturas.c invoca al main utilizando los siguientes argumentos:

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

- Estos argumentos deben pasarse cuando se ejecuta el fichero media_alturas.exe:

```
$ ./media_alturas.exe 157 180
```

1. Paso de parámetros a través de la línea de comandos

- Hay dos argumentos, **argc y argv**, que se utilizan para recibir los argumentos de línea de órdenes.

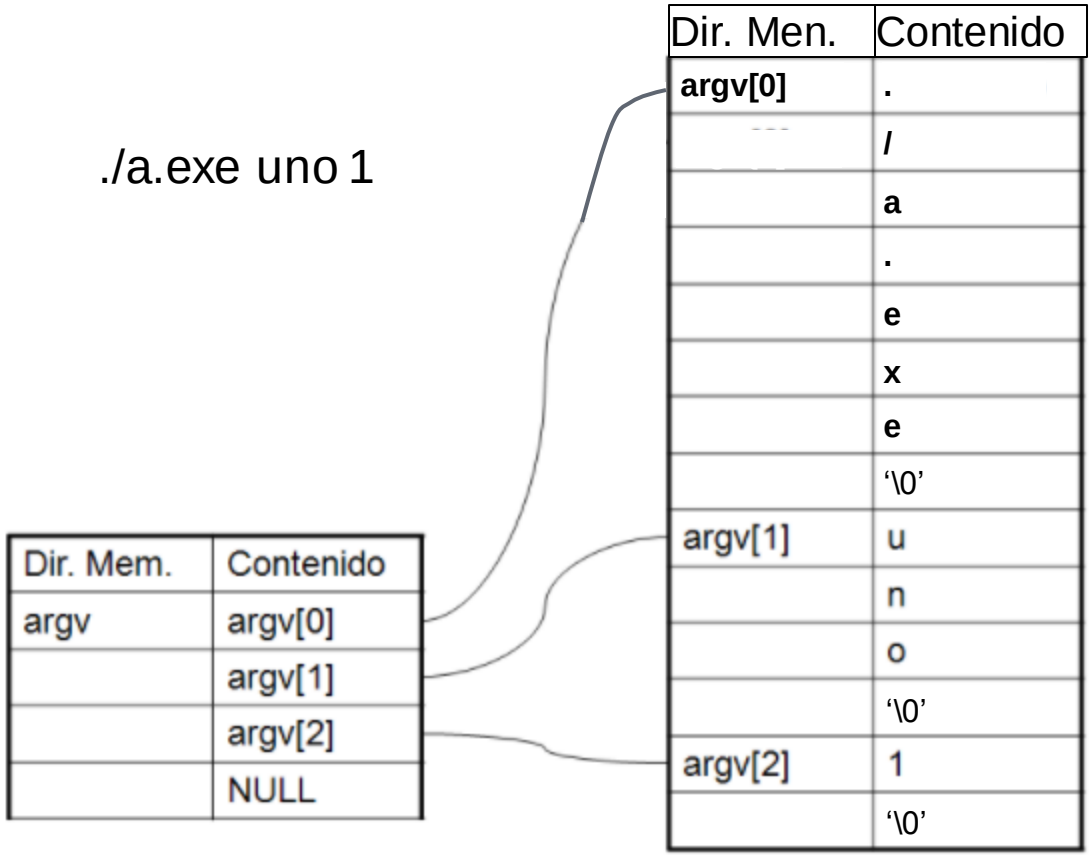
- El parámetro **argc**
  - Contiene el número de argumentos de línea de comandos y es un entero.
  - Al menos, o como mínimo, siempre vale 1, ya que el nombre del programa cuenta como primer argumento

- El parámetro **argv**
  - Es un puntero a un array de punteros a caracteres.
  - Cada elemento del array apunta a un argumento de la línea de órdenes

- Todos los argumentos de línea de órdenes son cadenas. Cualquier número tendrá que ser convertido manualmente por el programa al formato deseado

char *argv[ ] o char **argv

En la figura se puede ver un ejemplo gráfico simplificado de la ubicación en memoria de los argumentos pasados a un programa: argv será la dirección de memoria del primer elemento del array de punteros argv[], es decir, la dirección de memoria del elemento argv[0]; a su vez argv[0] apunta al primer elemento (carácter) de la cadena compuesta por el nombre del programa, es decir, argv[0] es a su vez un puntero a una cadena de caracteres. De igual manera se razona para argv[1] y argv[2] con respecto a las cadenas introducidas como primer y segundo argumento. Una conclusión de lo expuesto es que argv es un puntero a un puntero, y como tal puede ser tratado



Ejemplo 1: argc y argv

Crear un programa que muestre todos los argumentos introducidos a través de la línea de órdenes.

```
#include<stdio.h>
int main (int argc, char * argv[]) {

  int i;
  printf("Argumentos de la linea de ordenes\n\n");
  for(i=0; i<argc; i++)
    printf("El argumento %d es %s\n", i, argv[i]);

   printf("\n\nTerminación normal del programa.\n");
  return 0;
}
```

Resultado de ejecutar el programa

```
$ ./a.out 1 2 3 4
Argumentos de la línea de órdenes
El argumento 0 es ./a.out
El argumento 1 es 1
El argumento 2 es 2
El argumento 3 es 3
El argumento 4 es 4
```

Ejemplo 2: argc y argv

Escribir un programa que calcule la media de una lista de enteros. El programa recibirá el número de elementos de los que consta la lista como argumento desde la línea de comandos, seguido de los números que se quieren calcular.

Usar la función `strtol` que convierte un string en long int

```
long int strtol(const char *str, char **endptr, int base)
```

Ejemplo:

```
long int valor=strtol("5",NULL,10);
```

$./media 4 2 3 5 4

Ejemplo 2: argc y argv

Escribir un programa que calcule la media de una lista de enteros. El programa recibirá el número de elementos de los que consta la lista como argumento desde la línea de comandos, seguido de los números que se quieren calcular.

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

int main (int argc, char * argv[]) {
            long Numero = 0;
            float Suma = 0;
            float Media = 0;
            int NoNumeros =0;
            int i;
            printf("Argumentos de la linea de ordenes\n");
            for(i=1; i<argc; i++){
                        printf("El argumento %d es %s\n", i, argv[i]);
                        Numero = strtol (argv[i], NULL, 10); //strtol retorna 0 si error
                        if ((Numero==0) && !(*argv[i]=='0'))
                                    NoNumeros ++;
                        Suma = Suma + Numero;
            }
            Media = Suma / (float) (argc -1-NoNumeros);
            printf("La media es %f \n", Media );
  return 0;
}
```

Ejemplo 3: argc y argv

Escribir un programa que reciba una matriz 3x3 de números en float a través de la línea de comandos y nos la presente por pantalla

Usar strtof() (string to float)

Ejemplo:

```
float valor=strtof("3.5",NULL);
```

Ejemplo 3: argc y argv

Escribir un programa que reciba una matriz 3x3 de números en float a través de la línea de comandos y nos la presente por pantalla

```
/*
  Escribir un programa que admita una matriz
  3x3 de float a través de la línea de órdenes.
*/

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

#define FILAS 3
#define COLUMNAS 3

int main(int argc, char * argv[])
 {
  float matriz[FILAS][COLUMNAS];
  int num_datos = FILAS*COLUMNAS+1;
  int i, fila, columna;
  if (argc != num_datos)
   {
    printf("\n\nUtilización: ./a.out 9
números\n\n");
   }
else
   {
    printf("\n\n");
    i=1;
    for(fila=0;fila<FILAS;fila++)
    for(columna=0;columna<COLUMNAS;columna++)
     {
      matriz[fila][columna] =
strtof(argv[i++],NULL);
     }
    printf("\n\nLa matriz proporcionada
era:\n\n");
    for(fila=0;fila<FILAS;fila++)
     {
      printf("|");

for(columna=0;columna<COLUMNAS;columna++)
       printf("%6.2f", matriz[fila][columna]);
      printf("|\n");
     }
   }
  printf("\n\nTerminación normal\n\n");
  return 0;
 }
```

Ejemplo 3: argc y argv

Escribir un programa que reciba una matriz 3x3 de números en float a través de la línea de comandos y nos la presente por pantalla

```
#include<stdio.h>
#include<stdlib.h>
#define FILAS 3
#define COLUMNAS 3
int main(int argc, char * argv[]){
   float matriz[FILAS][COLUMNAS];
   //valor de elementos introducidos en linea comandos
   int num_datos = FILAS*COLUMNAS+1;
   int i, fila, columna,error = 0;

   if (argc != num_datos){
      printf("\n\nUtilización: ./a.out 9 números\n\n");
      error = 1;
   }
   else{
      printf("\n\n");
      i=1;
      for(fila=0;fila<FILAS;fila++)
         for(columna=0;columna<COLUMNAS;columna++){
            matriz[fila][columna] = strtof(argv[i++],NULL);
            if (matriz[fila][columna] == 0){
               error = 1;
               printf("\n\nUtilización: ./a.out 9 números (no letras\n\n");
            }
         }
if (!error){
            printf("\n\nLa matriz proporcionada era:\n\n");
               for(fila=0;fila<FILAS;fila++){
                  printf("|");
                  for(columna=0;columna<COLUMNAS;columna++)
                     printf("%6.2f", matriz[fila][columna]);
                  printf("|\n");
               }
         }
   }
   if (!error)
      printf("\n\nTerminación normal");
   else
      printf("Terminación con error\n\n");
   return error;
}
```

Ejercicio. Calculadora

- Implementar un programa tipo "calculadora" que reciba por parámetros "argc/argv" los operandos y la operación a realizar. **Está prohibido** usar funciones de lectura de parámetros por STDIN (getchar, scanf, etc...) y las librerías stdlib.h y string.h

- Modo de paso de parámetros:
  - El programa recibirá 3 parámetros especificados de la siguiente manera:

    - Se usarán 3 palabras para identificar los operandos y la operación : OP1, OP2 , OPERACION
    - Estas palabras estarán seguidas (sin espacios) por el separador "=",seguido a su vez por el valor que se quiera dar a esa variable.
      - Ej: OP1=5
    - Los valores válidos para OP1 y OP2 son números racionales
    - Los valores válidos para OPERACION son las palabras "suma", "resta" y "multiplicación", sin comillas

      - Ej: OPERACION=suma

    - Un ejemplo de uso sería el siguiente:
      - ./calculadora OP1=5 OP2=7 OPERACION=suma
      - El resultado obtenido será: 12

Ejercicio. Nombre y Apellido

- Escribir por pantalla el nombre completo de un alumno en formato <nombre> <primer apellido> <segundo apellido>
- El nombre completo en formato apellidos <primer apellido> <segundo apellido>, <nombre> se pasa como parámetros del main.
- Ejemplos de uso:

```
$ ./nombreyapellidos Castro Escudero, Alfonso
$ El nombre del alumno es Alfonso Castro Escudero
$ ./nombreyapellidos Garcia-Romero de la Rosa, Juan Luis
$ El nombre del alumno es Juan Luis Garcia-Romero de la Rosa
```

Ejercicio.DNI

- Crear el DNI de un ciudadano concatenando un número de 8 cifras que se el pasa como parámetro del main y la letra correspondiente.
- Para obtener la letra, es necesario dividir el número del DNI entre 23 y en lugar de calcular los decimales, nos fijaremos en el resto que ofrece la solución. Luego comparar el resto con el **código TRWAGMYFPDXBNJZSQVHLCKE**. El número que obtenido marcará la posición de la letra en el código. Es decir, que si era el resto era 3, la letra del DNI será la W.
- Ejemplo de uso:

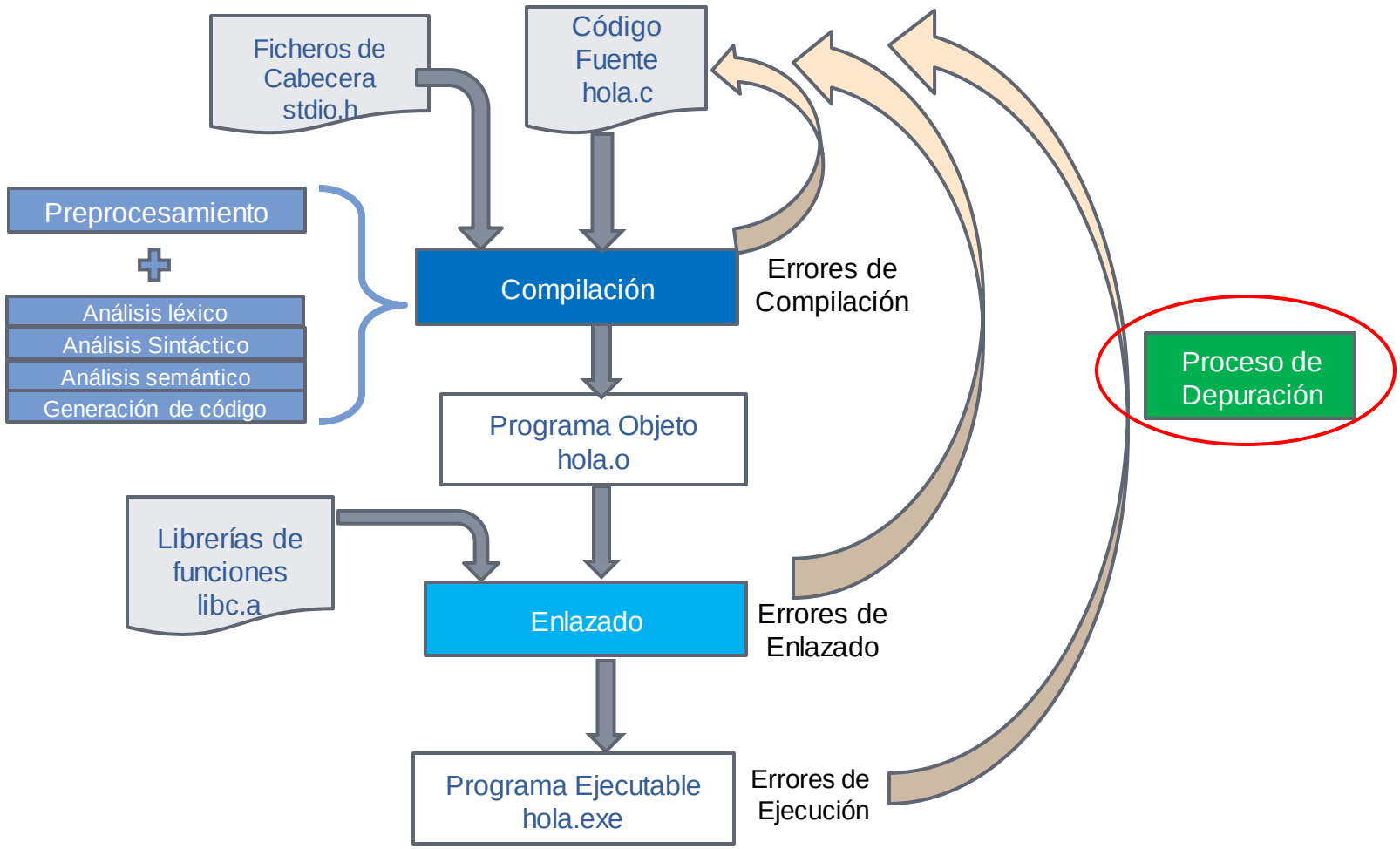
```
$ ./crearDNI 12345678
$ El número de DNI es 12345678Z
```

Índice

Tema 1 – Apartado 3

3. Depuración: GDB

5. Proceso de Depuración y verificación



3. Proceso de Depuración y verificación

- La depuración de un programa es el proceso de encontrar los errores de ejecución de un programa y corregir o eliminar dichos errores.

- Depuración manual:
  - Se proporciona al programa entradas válidas que conducen a una solución conocida.
  - También deben incluirse datos válidos para comprobar la capacidad de detección y generación de errores del programa
  - Se incluyen “trazas” en el programa para ir comprobando los valores intermedios que se van obteniendo son los esperados.

3. Proceso de Depuración y verificación

- Depuración a partir de herramientas:
  - Dada la complejidad y el tamaño de los programas y, para ahorrar tiempo y recursos, existen lo que conocemos como herramientas de depuración (debugger).
  - Todos los IDE tienen asociados su herramienta de depuración.
  - Las acciones más habituales que realizan estas herramientas son:
    - Ejecutar paso a paso un programa (stepping).
    - Marcar puntos de parada (breakpoints).
    - Examinar el contenido de las variables y objetos.
    - Conocer el encadenamiento de llamadas de procedimientos.
    - Retomar la ejecución hasta un nuevo punto de detención.

3. Proceso de Depuración y verificación

- La verificación es la acción de comprobar que el programa está de acuerdo con su especificación o definición de requisitos.
- Es decir, se comprueba que el sistema cumple con los requerimientos funcionales y no funcionales que se han especificado.
- Verificación Software: Asignatura Optativa de INSO4

"La mejor herramienta de depuración es evitar los errores desde el principio"

3. Proceso de Depuración y verificación

- Herramienta de depuración gdb
  - Compilar con opciones de depuración:
    - **gcc –g archivo.c -o archivo.exe**
  - Cargar el ejecutable en el depurador:
    - **gdb archivo.exe**
  - Poner breakpoints en algún punto del programa:
    - Opción 1:
      - **break num_linea**
      - **break fichero:num_linea**
    - Opción 2:
      - **break nombreFunción**
    - Borrar un breakpoint:
      - **delete n (borra el breakpoint n)**
      - **delete (borra todos los breakpoints)**
    - Informacion sobre breakpoint
      - **info break**

3. Proceso de Depuración y verificación

- Ejecutar el programa:
  - **run**
  - **run param1 param2... ParamN**
- Consultar los argumentos del programa:
  - **show args**
- Ejecutar una instrucción (paso a paso):
  - **step**
- Continuar con la ejecución:
  - **continue**
- Inspeccionar variables:
  - **print nombreVariable**
  - **print *nombreVariable**
  - **print *nombreArray@tamanio**
- Salir de la depuración:
  - **quit**

3. Proceso de Depuración y Verificación

- Mostrar el estado de la pila de llamadas:
  - Primero parar el programa: Ctrl+c
  - Mostrar el estado del programa:
    - **backtrace**
  - Moverse a una sección de la pila de llamadas
    - **frame "número"**

Asignatura/Tema