Fundamentos de Sistemas Operativos

Introducción al Lenguaje de Programación C II

Universidad de Valladolid

Departamento de Informática

Curso 2023/2024

Contenidos

Librería matemática

- Las funciones matemáticas en C están en la biblioteca `libm.a`, para usarla hay que incluir el fichero `math.h`.

```
#include <math.h>
int main() {
   double a=2, b=3, c;
   c = pow(a,b);
}
```

- Y compilar enlazando la librería, puesto que gcc no la enlaza por defecto.

```
gcc fuente.c -lm -o ejecutable
```

Cadenas de caracteres

- Las cadenas de caracteres (*strings*) en C son *arrays* de `char`.
- Se pueden declarar e inicializar de diferentes maneras.

```
int main()
{
    char v[10];
    char w[] = "Hola"
}
```

- Por convenio, todas las cadenas de caracteres tienen como último elemento el carácter nulo (`'\0'`).
- Siempre que se traten cadenas hay que asegurarse que el último carácter es el nulo.

Cadenas de caracteres

- Este tipo de declaración se encarga de reservar memoria para que contenga la cadena entera y añade el carácter nulo '\0'.

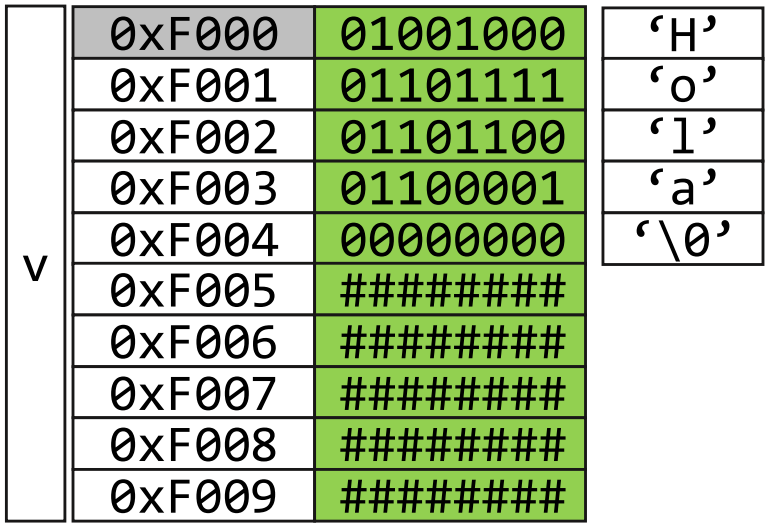
```
char w[] = "Hola"
```

- Luego esa cadena tiene una longitud de 4 caracteres aunque realmente ocupa 5.

w	0xF000	01001000	'H'
	0xF001	01101111	'o'
	0xF002	01101100	'l'
	0xF003	01100001	'a'
	0xF004	00000000	'\0'

Cadenas de caracteres

```
char v[10];
v[0] = 'H';
v[1] = 'o';
v[2] = 'l';
v[3] = 'a';
v[4] = '\0';
```



- En este caso, la cadena de caracteres `v` tiene una longitud de 4 caracteres aunque realmente ocupa 10 y está bien construida porque termina con el carácter nulo.

Cadenas de caracteres

- Muchas funciones que usan *strings* no necesitan conocer su tamaño de manera explícita , puesto que dan por hecho que terminan en el carácter nulo.

```
#include<stdio.h>
void mayusculas(char *v) {
    unsigned int i = 0;
    while(v[i]!='\0')
    {
        if (v[i] >= 'a' && v[i] <= 'z')
            v[i] = v[i] - 32;
        i++;
    }
}
int main() {
    char v[] = "hola";
    mayusculas(v);
    printf("texto en maysuculas: %s\n",v);
    return 0;
}
```

Cadenas de caracteres

`atoi`: Transforma una cadena de caracteres en un número entero. Requiere la inclusión de `stdlib.h`.

```
int atoi(const char *str);
```

Parámetros:

- `str`: puntero que apunta a la primera dirección del *string* a transformar.

Retorno:

- `int`: Esta función devuelve el número convertido como un valor eterno `int`. Si no se ha podido realizar una conversión válida, devuelve `0`.

La función `atoi` omite todos los caracteres con espacios en blanco al principio de la cadena, convierte los caracteres siguientes como parte del número y se detiene cuando encuentra el primer carácter que no sea un número.

Cadenas de caracteres

- **NOTA:** La declaración de un parámetro con `const char*` significa que el contenido al que apunta el puntero (en este caso caracteres) no se puede modificar.

```
#include<stdio.h>
void mayusculas(const char *v) {
    unsigned int i = 0;
    while(v[i]!='\0') {
        if (v[i] >= 'a' && v[i] <= 'z')
            v[i] = v[i] - 32; //ERROR! no se puede
   modificar el contenido apuntado
        i++;
    }
}
```

Cadenas de caracteres

`atof`: Transforma una cadena de caracteres en un número real de doble precisión. Requiere la inclusión de `stdlib.h`.

```
double atof(const char *str);
```

Parámetros:

- `str`: puntero que apunta a la primera dirección del *string* a transformar.

Retorno:

- `double`: Esta función devuelve el número convertido como un valor real de doble precisión `double`. Si no se ha podido realizar una conversión válida, devuelve `0.0`.

La función `atof` omite todos los caracteres con espacios en blanco al principio de la cadena, convierte los caracteres siguientes como parte del número y se detiene cuando encuentra el primer carácter que no sea un número.

Cadenas de caracteres

`strlen`: calcula la longitud de la cadena pero sin incluir en la cuenta el carácter nulo final. Requiere la inclusión de `string.h`.

```
size_t strlen(const char *str);
```

Parámetros:

- `str`: puntero que apunta a la primera dirección del string.

Retorno:

- `size_t`: longitud de la cadena o string.
  `size_t` está definido en `stddef.h`, no hace falta incluirlo si se incluye `stdlib.h` o `stdio.h`, y es un entero cuya definición puede depender del compilador, por ejemplo:

```
typedef unsigned long size_t
```

Cadenas de caracteres

Ejemplo de uso:

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

int main(){
    char s[]="Hola";
    int tam;

    tam=strlen(s);
    printf("La longitud de %s es %d\n",s,tam);

    return 0;
}
```

Cadenas de caracteres

`strcmp`: compara dos cadenas de caracteres o *strings*. Se considera que una cadena es inferior a otra si es anterior alfabéticamente (orden `ascii`). Requiere la inclusión de `string.h`.

```
int strcmp(const char *str1, const char *str2);
```

Parámetros:

- `str1`: puntero que apunta a la primera cadena a comparar.
- `str2`: puntero que apunta a la segunda cadena a comparar.

Retorno:

- `int`: Esta función devuelve los siguientes valores:
  - Si el valor de retorno menor que `0` entonces indica que `str1` es menor que `str2`.
  - Si el valor de retorno mayor que `0` entonces indica que `str2` es menor que `str1`.
  - Si el valor de retorno igual a `0` entonces indica que `str1` es igual a `str2`.

Cadenas de caracteres

`strcpy`: Copia una cadena de caracteres o *string*. Requiere la inclusión de `string.h`.

```
char *strcpy(char *dest, const char *src);
```

Parámetros:

- `dest`: puntero a la cadena de destino donde se va a copiar el contenido. ¡Importante! el puntero debe apuntar a una zona de memoria válida donde quepa la cadena.
- `src`: puntero de la cadena a copiar.

Retorno:

- `char *`: devuelve un puntero a la cadena de destino `dest`.

¡Va antes el destino de la copia que el origen! ¯_(ツ)_/¯

Cadenas de caracteres

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

int main(){
    char src[] = "Hola";
    char dest[5];
    /*Alternativa de reserva de memoria
    char *dest = NULL;
    dest = (char *)malloc(sizeof(char) * (strlen(src)
    + 1));
    if (dest == NULL){
        printf("Error reservando memoria\n");
        exit(-1);
    }*/
    strcpy(dest, src);
    printf("Cadena: %s\n",dest);
    /*free(dest);*/
    return 0;
}
```

Cadenas de caracteres

`strcat`: Concatena dos strings. Añade uno al final del otro.

```
char *strcat(char *dest, const char *src);
```

Parámetros:

- `dest`: Es un puntero a la cadena destino, que debe contener un *string*, y debe ser lo suficientemente grande como para contener la cadena resultante concatenada.
- `src`: Cadena que se va a añadir o concatenar.

Retorno:

- `char *`: devuelve un puntero a la cadena de destino `dest`.

Cadenas de caracteres

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

int main (){
   char src[50], dest[50];

   strcpy(src,  "Cadena origen");
   strcpy(dest, "Cadena destino");

   strcat(dest, src);

   printf("Cadena resultante : %s", dest);

   return(0);
}
```

Cadenas de caracteres

`strtok`: rompe la cadena en una serie de *tokens* utilizando un delimitador.

```
char *strtok(char *str, const char *delim);
```

Parámetros:

- `str`: El contenido de este string **se modifica** y se divide en cadenas más pequeñas (*tokens*).
- `delim`: Es el string que contiene el delimitador. Estos pueden variar de una llamada a otra.

Retorno:

- `char *`: Esta función devuelve un puntero al *token* encontrado en el *string*. Devuelve un puntero nulo `NULL` si no quedan *tokens* por recuperar.

Se suele usar en un bucle `while` que irá invocando a `strtok` en cada iteración hasta que la función devuelva `NULL` (no hay más `tokens`). **¡Recuerda!** la cadena `str` es modificada por la función y se puede perder su contenido.

Cadenas de caracteres

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

int main () {
   char cadena[100] = "Esto es un token;esto otro;y
    esto otro mas";
   const char delim[2] = ";";
   char *token; //no requiere reserva de memoria

   //lectura del primer token
   token = strtok(cadena, delim);

   //Bucle de busqueda de tokens
   while( token != NULL ) {
      printf( "%s\n", token );
      token = strtok(NULL, delim);
   }
   printf("%s\n",cadena);

   return(0);
}
```

Cadenas de caracteres

`fgets` es una función que permite leer una línea de texto desde un archivo o desde la entrada estándar. Con scanf no se pueden leer *strings* que contienen espacios.

```
char *fgets(char *str, int num, FILE *stream);
```

Parámetros:

- `str`: puntero al *array* donde se almacenará la línea leída.
- `num`: número máximo de caracteres a leer (incluyendo el carácter nulo de terminación).
- `stream`: puntero al archivo del que se leerá.

Retorno:

- `char *`: retorna un puntero a la cadena de caracteres leída si tiene éxito, o `NULL` en caso de error o si se alcanza el final del archivo.

Cadenas de caracteres

```
char buffer[100];
fgets(buffer, 100, stdin);
```

Aunque todavía no se han estudiado archivos, se pueden usar tres archivos especiales (flujos de datos) para leer de la entrada estándar y escribir en las salidas estándar y en la salida estándar de error:

- **stdin** (*Standard Input*): Es el flujo de entrada estándar. Por defecto, es el teclado.
- **stdout** (*Standard Output*): Es el flujo de salida estándar. Por defecto, es el terminal.
- **stderr** (*Standard Error*): Es el flujo de salida de errores estándar. Por defecto también es el terminal.

Cadenas de caracteres

Con la función `fprintf`, de uso equivalente a `printf`, se puede escribir en la salida estándar o en la salida estándar de error.

```
#include<stdio.h>
int main() {
    int num;
    printf("Escribe un numero positivo: ");
    scanf("%d",&num);
    if (num>=0)
        fprintf(stdout,"%d es positivo\n",num);
        //Equilalente a
        //printf("%d es positivo\n",num);
    else
        fprintf(stderr,"%d no es positivo\n",num);
    return 0;
}
```

También existe la función `fscanf`, de uso equivalente a `scanf`.

Parámetros el `main`

- Un programa desarrollado en C puede recibir parámetros de entradas como lo que recibe un comando del shell.
- Para poder procesar los parámetros de entrada hay que declarar la función `main` de la siguiente manera:

```
#include<stdio.h>
int main(int argc, char* argv[]) {
//int main(int argc, char** argv) {
    return 0;
}
```

- `argc` entero que representa el número de parámetros incluido el nombre del ejecutable.
- `argv` es un *array* de tamaño `argc` que consiente punteros a `char`, es decir, punteros a cadena de caracteres o *strings*. Esos *strings* son los parametros que se han introducido, incluido el nombre del ejecutable.

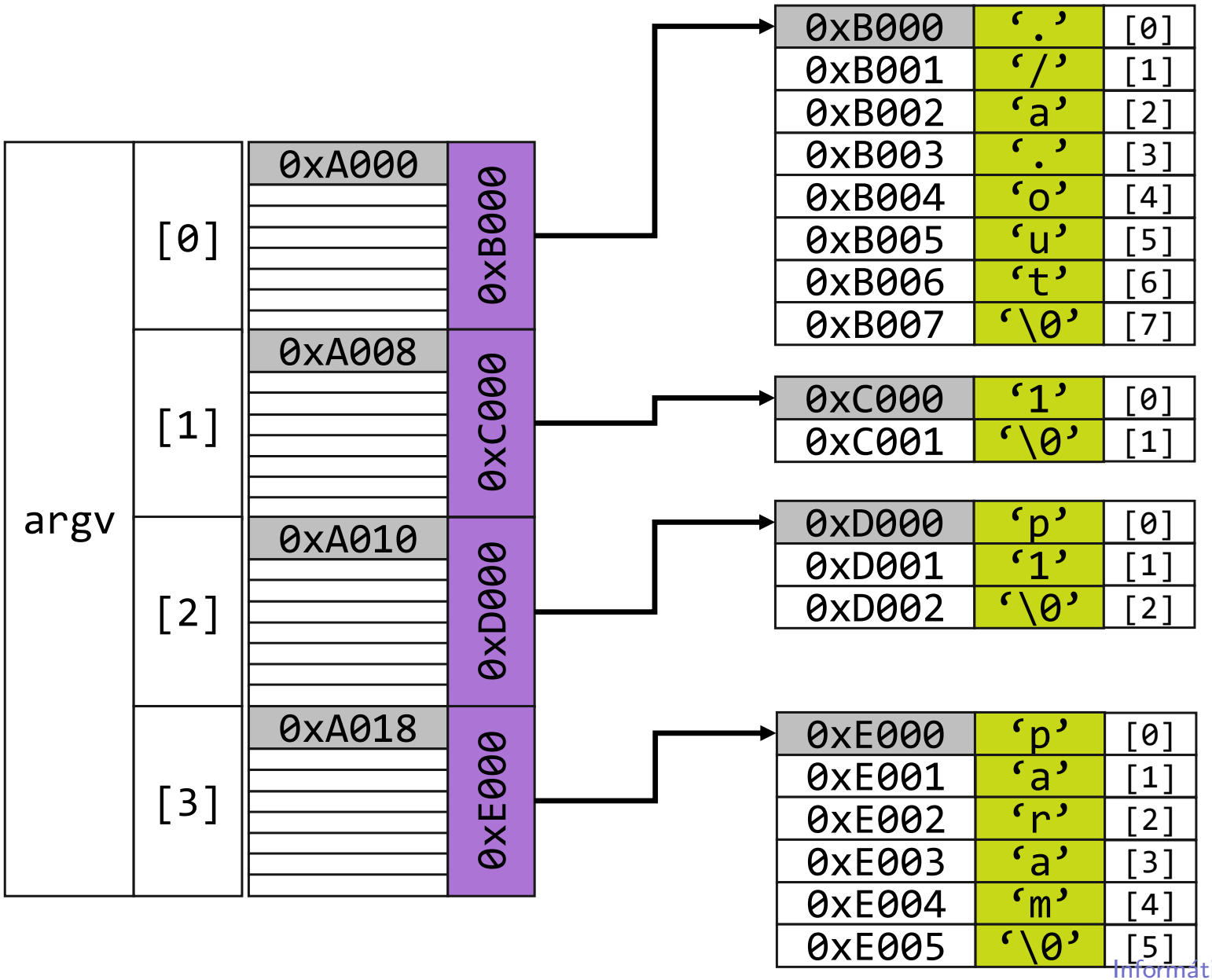
Parámetros el `main`

- Si se ejecuta el fichero ejecutable `a.out` con los siguientes parámetros.

```
./a.out 1 p1 param
```

- `argc = 4`. El valor de `argc` es 4, puesto que hay 4 parámetros incluido el nombre del ejecutrable (`./a.out`, `1`, `p1` y `param`)
- El array de 4 posiciones `argv` contiene:
  - argv[0]: contiene la primera dirección de memoria del *string* `"./a.out"`.
  - argv[1]: contiene la primera dirección de memoria del *string* `"1"`.
  - argv[2]: contiene la primera dirección de memoria del *string* `"p1"`.
  - argv[3]: contiene la primera dirección de memoria del *string* `"param"`.

Parámetros el `main`



Parámetros el `main`

```
int main(int argc, char *argv[]) {
   int a, b;
   if (argc!=3){
      fprintf(stderr,"Numero de parametros incorrecto.\n");
      fprintf(stderr,"Uso: ejecutable param1 param2\n");
      exit(-1);
   }
   a = atoi(argv[1]);
   b = atoi(argv[2]);

   printf("Suma de %s + %s es %d\n",argv[1], argv[2], a+b);
}
```

Prueba a ejecutar ese programa con los parámetros:

- 1 2
- 1a 2a
- a1 a2

¿Con cuáles funciona? Para validar la entrada también se pueden usar las funciones `isdigit` o `isalpha`.

Parámetros el `main`

- `return` se utiliza dentro de una función para devolver un valor y terminar la ejecución de la función.
- `exit` se utiliza para terminar el programa (proceso) y devolver un valor de salida al sistema operativo.
- `return` devuelve un valor a la llamada de la función o al sistema operativo en el `main`.
- `exit` puede ser usado en cualquier parte del programa y termina todo el programa (proceso), devolviendo un valor de salida al sistema operativo.
- Después de usar `return`, el programa sigue ejecutando a menos que se llegue al final del `main`.
- Después de usar `exit`, el programa se detiene inmediatamente y se devuelve el control al sistema operativo.

Reserva dinámica de memoria

- Cuando no se conoce *a priori* el tamaño de un *array*, se puede declarar sobredimensionado:

```
int main() {
    int a[1000];
}
```

- O hacer la declaración una vez conocido el tamaño:

```
int main() {
    int tam=0;
    printf("Tamano del array: ");
    scanf("%d",&tam);
    int a[tam];
}
```

En ambos casos, esas variables se almacenan en la pila. Si sus tamaños son muy elevados, pueden desbordar la pila (*stack overflow*) y provocar una violación de segmento.
Además, hasta que no termina la ejecución de la función la memoria sigue ocupada aunque ya no se use.

Reserva dinámica de memoria

- C permite la reserva dinámica (en tiempo de ejecución) de memoria con la función *malloc* (*Memory ALLOCation*). La familia de funciones de reserva y liberación de memoria requiere la inclusión del fichero de cabecera `<stdlib.h>`

```
void* malloc(size_t size);
```

Parámetros:

- `size`: especifica el tamaño en bytes que la memoria que se quiere reservar o alojar.

Retorno:

- `void *`: Si la reserva de memoria se pudo realizar, retorna un puntero al inicio del bloque de memoria; si falla, retorna `NULL`.

La memoria reservada no está en la pila de la función, sino en el montículo (*heap*) y su ciclo de vida no se limita al ciclo de vida de ejecución de la función como ocurre con la pila (*stack*).

Reserva dinámica de memoria

- La macro `sizeof` en C se utiliza para determinar el tamaño en bytes de un tipo de dato.
- Sintaxis: `sizeof(tipo)`
- Retorna el tamaño del tipo de dato (de tipo `size_t`, puede depender de la arquitectura).
- Puede ser usada con cualquier tipo de dato, incluyendo tipos de datos definidos por el usuario.

```
typedef struct {
   double x;
   double y;
} Punto;

int main() {
    printf ("Un double ocupa %lu bytes\n",sizeof(double));
    printf ("Un Punto ocupa %lu bytes\n",sizeof(Punto));
    return 0;
}
```

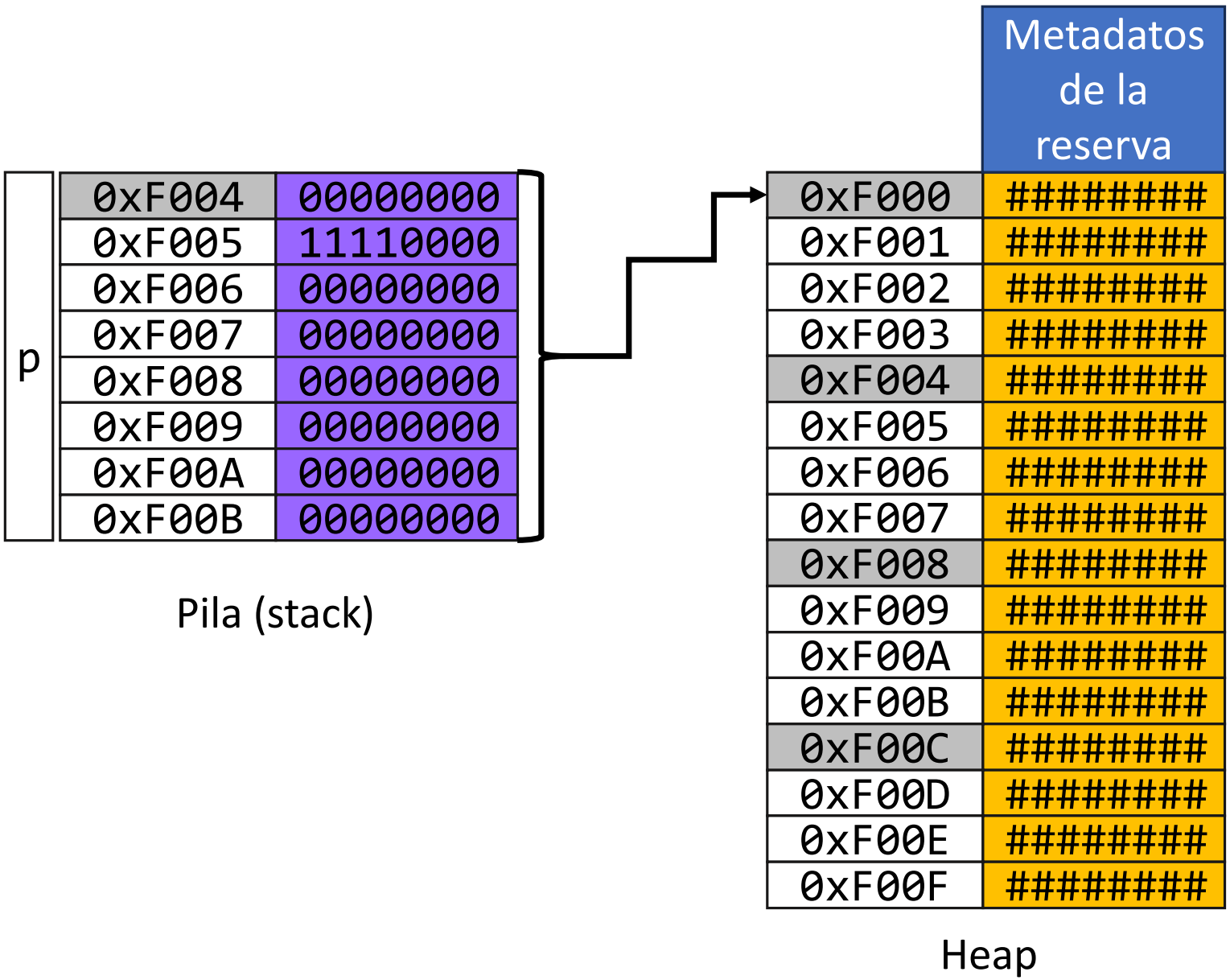
Reserva dinámica de memoria

- Usando `malloc` y `sizeof` se puede hacer reserva de memoria para un *array* de enteros (`int`), por ejemplo se puede reservar memoria para un array de enteros.

```
#define TAM 4
int main() {
  int *v = NULL;
  v = (int *)malloc(sizeof(int)*TAM);
  if (v!=NULL) {
    for (int i=0; i<TAM; i++) {
      v[i] = i;
    }
    free(v); //libera
  }
  else {
    fprintf(stderr,"Error asignando memoria.\n");
  }
  return 0;
}
```

Una vez que se tiene una dirección de memoria válida en una variable de tipo puntero (`v`), se puede acceder a sus datos con los operadores `*`, `[]` o `->`.

Reserva dinámica de memoria



Reserva dinámica de memoria

Otro ejemplo

```
#define TAM 4
typedef struct {
    double x;
    double y;
} Punto;
int main() {
  Punto  *v = NULL;
    v = (Punto *)malloc(sizeof(Punto)*TAM);
    if (v!=NULL) {
        for (int i=0; i<TAM; i++) {
            v[i].x = i;
            v[i].y = i + 1;
        }
        free(v);
    }
    else {
        fprintf(stderr,"Error asignando memoria.\n");
    }
    return 0;
}
```

Reserva dinámica de memoria

- El *heap* es una región de memoria que se utiliza para almacenar datos que persisten más allá del alcance de la función que los creó.
- La asignación y liberación de memoria en el *heap* se realiza explícitamente por el programador.
- Se debe liberar la memoria utilizando `free` cuando ya no se necesite (no tiene que ser obligatoriamente al final del programa). Se pasa como argumento el puntero con primera dirección de la zona a liberar (la que devolvió `malloc`, o `calloc`, o `realloc`).
- La memoria asignada no se libera automáticamente y debe ser liberada explícitamente utilizando la función `free` cuando ya no se necesite. Si no se libera, puede resultar en una fuga de memoria (*memory leak*).

Reserva dinámica de memoria

- `calloc`: se utiliza para asignar bloques de memoria inicializados a cero, es muy parecida a `malloc`.

```
void* calloc(size_t nmemb, size_t size);
```

Parámetros:

- `nmemb`: número de elementos a reservar
- `size`: tamaño de cada elemento. En `malloc` es un único parámetro `nmemb * size`.

Retorno:

- `void *`: retorna un puntero al inicio del bloque de memoria asignada.

Es útil cuando se trabaja con *arrays* y se desea inicializarlos a cero.

Reserva dinámica de memoria

- `realloc`: se utiliza para cambiar el tamaño de un bloque de memoria previamente asignado.

```
void* realloc(void* ptr, size_t size);
```

Parámetros:

- `ptr`: un puntero al bloque de memoria existente.
- `size`: el nuevo tamaño deseado.

Retorno:

- `void *`: retorna un puntero al inicio del bloque de memoria, que puede ser diferente al original si la asignación se mueve.

Puede ser usado para expandir o reducir un bloque de memoria previamente asignado.

Reserva dinámica de memoria

`memcpy`: copia `n` bytes del área de memoria apuntada por `src` al área de memoria apuntada por `dest`.

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

Parámetros:

- `dest`: puntero al array de destino donde se copiará el contenido, convertido a un puntero de tipo `void*`.
- `src`: puntero al array origen de los datos a copiar, convertido a un puntero de tipo `void*`.
- `n`: es el número de bytes a copiar.

Retorno:

- `void *`: retorna un puntero al inicio del bloque de memoria de destino (`dest`).

Reserva dinámica de memoria

- Cuando se reserva memoria con `malloc` o `calloc` en C, aunque se trate de estructuras bidimensionales (o de orden superior), los datos se agrupan en la memoria de manera unidimensional o lineal y sólo se pude usar un índice.
- Si se quiere recorrer la matriz por filas y columnas con dos índices, uno para indexar la fila y otro la columna, y por lo tanto, con dos bucles anidados, se puede usar de una expresión que operando con los índices de fila y columna obtenga el índice del recorrido lineal (depende del número de columnas).

Reserva dinámica de memoria

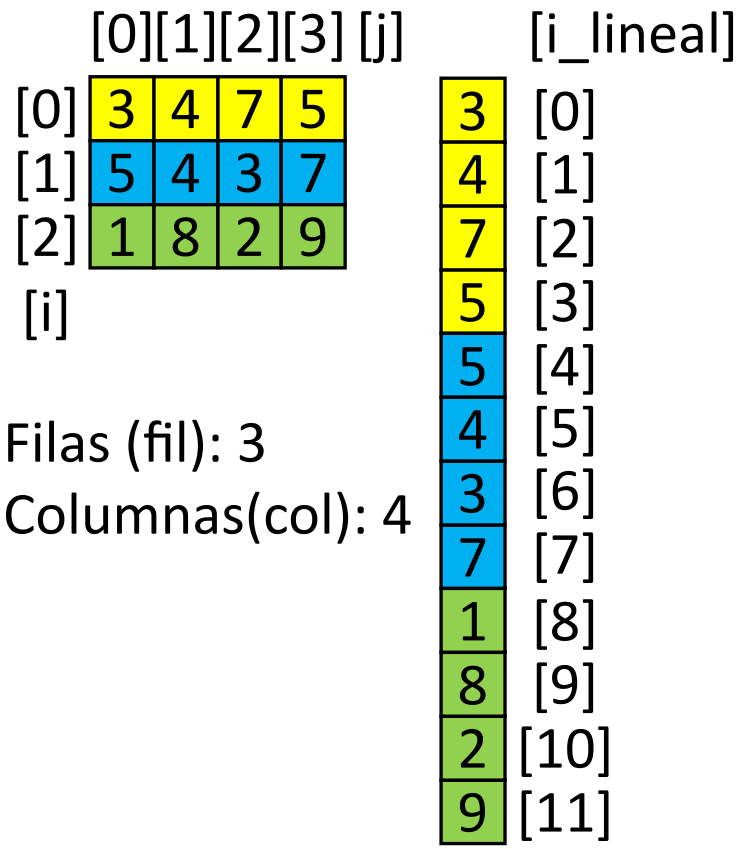
```
int main()
{
    int fil=3, col=4, i;
    int *matriz = NULL;

    //Reserva de la matriz
    matriz = (int *)malloc(fil*col*sizeof(int));
    if (matriz == NULL) {
        fprintf(stderr,"Error alojando memoria\n");
        exit(-1);
    }

    //Solo se puede usar un indice para acceder a las
    posiciones de la matriz
    for (i=0; i<fil*col; i++) {
        matriz[i] = rand() % 10;
        printf("matriz[%d]: %d\n",i,matriz[i]);
    }
    free(matriz);
}
```

Reserva dinámica de memoria

i	j	i_lineal = (i*col + j)
0	0	0
0	1	1
0	2	2
0	3	3
1	0	4
1	1	5
1	2	6
1	3	7
2	0	8
2	1	9
2	2	10
2	3	11



Reserva dinámica de memoria

```c
int main()
{
  int fil=3, col=4, i, j;
  int *matriz = NULL;
  //Reserva de la matriz
  matriz = (int *)malloc(fil*col*sizeof(int));
  if (matriz == NULL)     {
    fprintf(stderr,"Error alojando memoria\n");
    exit(-1);
  }
  for (i=0; i<fil; i++) { //recorrido por filas
    for (j=0; j<col; j++) { //recorrido por columnas
      matriz[i*col + j] = rand() % 10;
      printf("matriz[%d, %d]: %d\n",i,j,matriz[i*col + j]);
    }
  }
  free(matriz);
}
```

Reserva dinámica de memoria

Puede ser útil cuando se pasa una matriz como parámetro a una función.

```
void func(int *m, int fil, int col)
{
   for (int i=0; i<fil; i++) { //recorrido por filas
      for (int j=0; j<col; j++) { //recorrido por columnas
         m[i*col + j] = rand() % 10;
         printf("matriz[%d, %d]: %d\n",i,j,m[i*col + j]);
      }
   }
}
int main()
{
  int fil=3, col=4;
  int matriz[3][4];
  //Reserva de la matriz

  func((int *)matriz,fil,col);
}
```

Ficheros

- Los ficheros se declaran con un tipo especial: `FILE *`
- Se trata de un puntero a una estructura de datos que describe el fichero (nombre, estado, permisos...).
- Cada vez que se opera sobre un fichero se modifica su estado, con lo que las funciones que lo manipulan siempre modifican la variable (por eso es un puntero).
- Antes de usar un fichero hay que abrirlo.
- Y se debe cerrar cuando ya no se vaya a usar.

Ficheros

- `fopen`: permite abrir un fichero.

```
FILE * fopen(const char *nombre_fichero, const char *modo)
```

Parámetros:

- `nombre_fichero`: nombre del fichero que se va a abrir.
- `modo`: modo de apertura.

Retorno:

- `FILE *`: puntero a la estructura del archivo cargada en memoria. `NULL` si el archivo no se pudo abrir.

En esta sesión se estudian los ficheros de texto.

Ficheros

Modos de apertura

Modo	Descripción
`r`	Apertura para lectura. El fichero debe existir.
`w`	Apertura para escritura. Si el fichero no existe, se crea. Si existe, se sobrescribe.
`a`	Apertura para añadir al final. Si el fichero no existe, se crea.
`r+`	Apertura para lectura y escritura. El fichero debe existir.
`w+`	Apertura para lectura y escritura. Si el fichero no existe, se crea. Si existe, se sobrescribe.
`a+`	Apertura para lectura y añadir al final. Si el fichero no existe, se crea.

Ficheros

- `fclose:` cierra un fichero previamente abierto.

```
int fclose(FILE *fp);
```

Parámetros:

- `fp`: puntero al fichero a cerrar.

Retorno:

- `int *`: devuelve 0 (cero) si se pudo cerrar. Si ocurrió algún error al intentar cerrar el archivo, devuelve un valor diferente de cero.

Ficheros

Puede ser útil cuando se paso una matriz como parámetro.

```
FILE *fp;
char nombre[] = "./fichero.dat";
fp = fopen(nombre, "r");
//apertura en modo lectura

if (fp == NULL) {
   fprintf(stderr,"Error abriendo el fichero: %s\n",nombre);
   exit(-1);
}
//acciones usando el fichero:
fclose(fp); //cierre
```

Ficheros

```
int fprintf(FILE *fp, const char *format, ...)
```

- Uso similar a `printf`.

```
int fputs(const char *str, FILE *fp)
```

- Escribe una cadena de caracteres en el archivo.

```
int fputs(const char *str, FILE *fp)
```

- Escribe un carácter en el archivo.

Retornan un valor no negativo si la escritura tuvo éxito, o un valor negativo en caso de error.

Ficheros

```
int fscanf(FILE *fp, const char *format, ...)
```

- Retorna el número de elementos leídos y asignados correctamente. Si llega al final del fichero o encuentra un error, retorna `EOF`.

Ficheros

```c
int main() {
    FILE *fp;
    int num;

    // Abre el archivo para lectura
    fp = fopen("datos.txt", "r");

    if (fp != NULL) {
        while (fscanf(fp, "%d", &num) != EOF) {
            printf("%d\n", num);
        }

        fclose(fp);
    } else {
        printf("No se pudo abrir el archivo.\n");
    }

    return 0;
}
```

Ficheros

fgets

```
char *fgets(char *str, int num, FILE *fp)
```

- Lee una línea de un archivo y la almacena en el puntero `str`.
- Lee hasta `num-1` caracteres o hasta encontrar un salto de línea (que también lee) o el final del archivo.
- Retorna `NULL` al llegar al final del archivo.

fgetc

```
int fgetc(FILE *fp)
```

- Lee un carácter del archivo.
- Retorna el carácter leído o `EOF` si se llega al final del archivo o si hay un error.

Ficheros

```c
int main() {
    FILE *archivo;
    char linea[100]; // Tamaño máximo de una línea

    archivo = fopen("text.txt", "r");

    if (archivo == NULL) {
        perror("Error al abrir el archivo\n");
        exit(-1);
    }

    // Lectura línea por línea y salida estándar
    while (fgets(linea, 100, archivo) != NULL) {
        printf("%s", linea);
    }

    fclose(archivo);
    return 0;
}
```

Ficheros

`feof`: la función que determina si se ha alcanzado el final de un archivo.

```
int feof(FILE *fp)
```

- `fp`: puntero al archivo que se va a verificar.
- devuelve un valor distinto de cero si se ha alcanzado el final del archivo, y devuelve cero si no se ha alcanzado.

```
FILE *fp;
fp = fopen("archivo.txt", "r");
if (fp != NULL) {
    while (!feof(fp)) {
        char c = fgetc(fp);
        if (c != EOF) {
            printf("%c", c);
        }
    }
    fclose(fp);
}
```

Aritmética de punteros

La aritmética de punteros permite operar sobre la dirección de memoria contenida en una variable de tipo puntero.

- Al sumar o restar un valor a un puntero, se desplaza por la memoria en unidades del tamaño del tipo apuntado.
- Por ejemplo, dado un puntero a un entero (`int *ptr`), `ptr + 1` apuntará al siguiente entero en memoria. Realmente la dirección de memoria se incrementa en 4 (o al tamaño que tenga un entero en esa arquitecutra).
- Un puntero a un `char` (`char *ptr`), `ptr + 1` apuntará al siguiente `char` en memoria.

Aritmética de punteros

```
1 int arr[5] = {10, 20, 30, 40, 50};
2 int *ptr = arr; // ptr apunta al primer elemento de arr
3
4 for(int i = 0; i < 5; i++) {
5     printf("Elemento %d: %d\n", i, *(ptr++));
6 }
```

Prueba el mismo ejemplo cambiando la línea 5 por:

```
printf("Elemento %d: %d\n", i, *(++ptr));
```

y compruebe la diferencia ¿Se ha desbordado el vector?

Aritmética de punteros

malloc y la aritmética de punteros

```
int main()
{
    int * p = (int *)malloc(sizeof(int)*10);
    if (malloc!=NULL) {
      for (int i=0; i<10; i++)
          p[i] = i*10;
      for (int i=0; i<10; i++)
          printf("Elemento %d: %d\n", i, *(p++));

      free(p); //Esto genera un ERROR para
      //hacer un free el puntero debe conservar
      //la dirección que retornó malloc
    }
}
```

Aritmética de punteros

Solución

```
int main()
{
    int *p = (int *)malloc(sizeof(int)*10);
    if (malloc!=NULL) {
        int *f;
        f=p;
        for (int i=0; i<10; i++)
            p[i] = i*10;
        for (int i=0; i<10; i++)
            printf("Elemento %d: %d\n", i, *(f++));

        free(p);
    }
}
```