

Estructura de Datos y Algoritmos

ELO 320 - Clase 05

*“Paso de parámetros desde CMD, FILE I/O, y
asignación de memoria Dinámica en C”*

Dr. Ioannis Vourkas

06 de mayo, 2020, Valparaíso, Chile



UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA



DEPARTAMENTO DE
ELECTRONICA

Good Luck!



sobre ELO 320



Evaluación

Evaluaciones del ramo (coordinadas): **2 certámenes** y **2 tareas individuales**. Nota del ramo:

$$Nota = \frac{C1 + C2}{2} * 0.7 + \frac{T1 + T2}{2} * 0.3$$

- Dos certámenes obligatorios según calendario publicado. El promedio de los 2 certámenes debe ser como **mínimo 45** para poder aprobar la asignatura. **En caso de no cumplirse esta regla**, la nota de reprobación será el promedio de los certámenes que el alumno ha tomado.
- Dos tareas obligatorias según calendario publicado. El trabajo realizado en cada tarea debe ser coherente con lo solicitado, demostrando un intento claro hacia una solución, a fin de ser evaluada. **En caso de no cumplirse esta regla**, su nota máxima podrá ser 54.



Cualquier punto no contemplado quedará a criterio del profesor y/o la coordinación, según sea necesario.

Contacto con el profesor

Depto. de Electrónica, oficina (**ZOOM**)

Horario: **a coordinar via email**

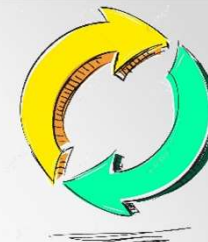
Email: ioannis.vourkas@usm.cl

Libro guía:

"Introduction to Algorithms", Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, MIT Press, 2009

Punteros en C

Puntero a un arreglo... explicaciones



```
/* an array with 5 rows and 2 columns*/  
int arr[5][2] = {{0,2}, {1,4}, {3,6}, {5,8},{7,9}};
```

La matriz $[i][j]$ como la imaginamos

$i \backslash j$	0	1
0	0	2
1	1	4
2	3	6
3	5	8
4	7	9

Direcciones de
memoria
contigua

Recuerden:
 $*(arr + i) + j$
es equivalente a
 $arr[i][j]$

```
/* qué se imprime con:*/  
printf ("%u\n", xxx );  
/* si donde xxx pongo:
```

- | | | | |
|------------|-------------|---------------|--------------|
| 1. arr | 2. *arr | 3. *(arr)+1 | 4. **arr |
| 5. (arr+1) | 6. *(arr+1) | 7. *(arr+1)+1 | 8. **(arr+1) |

La matriz $[i][j]$ en la memoria

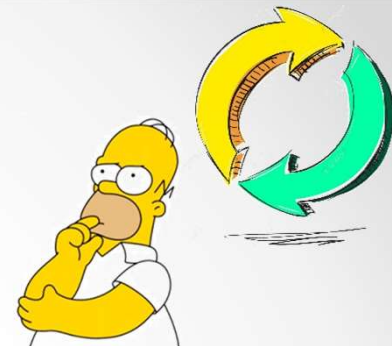
3888	0	4 bytes
3892	2	4 bytes
3896	1	4 bytes
3900	4	:
3904	3	
3908	6	
3912	5	
3916	8	
3920	7	
3924	9	

Ejemplo de
dirección

contenido

Estructuras en C

¿Qué son las estructuras en C?



En la clase 00 nos preguntamos: ¿Qué es una estructura de datos?

- una manera particular para organizar y almacenar **gran cantidad de datos**, a los que se puede acceder y modificar de manera eficiente
- Cuando programamos, a menudo es conveniente tener un nombre único para referirse a un grupo de valores relacionados.
 - Las **estructuras** proporcionan una forma de almacenar muchos valores diferentes en variables de tipos (potencialmente) diferentes, bajo el mismo nombre.
 - Esto hace que un programa sea más modular, que es más fácil de modificar.
- Las estructuras generalmente son útiles cuando se deben **agrupar muchos datos**, e.g:
 - ✓ para guardar registros de una base de datos
 - ✓ para almacenar información sobre contactos en una libreta de direcciones.
- Recuerden que los **arreglos** pueden contener varios elementos de **datos pero del mismo tipo**.
- La **estructura** es otro tipo de datos **definido por el usuario** que permite combinar elementos de datos **de diferentes tipos**.

Estructuras en C

¿Cómo definimos estructuras en C?



Para definir una estructura, debe usar el keyword **struct**, que define un nuevo tipo de datos, compuesto por más de un miembro

Ejemplo(s)

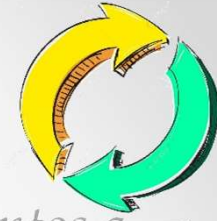
Prototipo

```
struct structure_tag {  
  
    member definition;  
    member definition;  
    ...  
    member definition;  
} one or more structure variables;
```

- *structure_tag* es opcional y es el **nombre de este nuevo tipo** de datos
- **member definition**, puede ser de cualquier tipo de variable
- **Antes del último ';'** podemos instanciar una o más variables, pero esto también es opcional

```
struct Book {  
    char title [50];  
    char author [50];  
    char subject [100];  
    int book_id;  
} book_1;  
  
struct point {  
    int value;  
};  
// definir una estructura  
struct point p1;  
  
/* OJO!!! así es como  
accedemos a las variables  
internas */  
p1.value = 10;
```

Estructuras en C



Estructuras como argumentos a funciones, y punteros a estructuras

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

struct Book {
    char* title; // equivalente a un string en C
    char author [50];
};

void printBook (struct Book);

int main( ) {
    struct Book B1; /* structure of type Book */

    B1.title = "C Programming";
    strcpy( B1.author, "Nuha Ali");

    printBook (B1);
    return 0;
}

void printBook ( struct Book b ) {
    printf ( "Book title : %s\n", b.title);
    printf ( "Book author : %s\n", b.author);
}
```

Igual que para cada otro tipo de datos, se pueden definir **punteros a estructuras**

```
struct Book *struct_pointer;
```

La **dirección** de una estructura la obtenemos con el **&**

```
struct_pointer = &B1;
```

Ahora podemos usar el puntero para acceder a los datos miembros de la estructura

```
(*struct_pointer).title = "ELO 320";
```

Con punteros a estructuras, es más conveniente usar el **->** en lugar de **(*)**

```
struct_pointer->title = "EDA";
```


Estructuras en C



El mismo ejemplo pero con punteros a estructuras

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

struct Book {
    char* title; // equivalente a un string en C
    char author [50];
};

void printBook (struct Book *);

int main( ) {
    struct Book B1; /* structure of type Book */

    B1.title = "C Programming";
    strcpy( B1.author, "Nuha Ali");

    printBook (&B1);
    return 0;
}

void printBook ( struct Book *b ) {
    printf ( "Book title : %s\n", b->title);
    printf ( "Book author : %s\n", b->author);
}
```

Igual que para cada otro tipo de datos, se pueden definir **punteros a estructuras**

```
struct Book *struct_pointer;
```

La **dirección** de una estructura la obtenemos con el **&**

```
struct_pointer = &B1;
```

Ahora podemos usar el puntero para acceder a los datos miembros de la estructura

```
(*struct_pointer).title = "ELO 320";
```

Con punteros a estructuras, es más conveniente usar el **->** en lugar de **(*)**

```
struct_pointer->title = "EDA";
```

Estructuras en C



*El mismo ejemplo pero **con** **typedef**
y punteros a estructuras*

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

typedef struct Record {
    char* title; // equivalente a un string en C
    char author [50];
} Book;

void printBook (Book *);

int main( ) {
    Book* B1; // pointer to structure of type
              // Book

    B1->title = "C Programming";
    strcpy( B1->author, "Nuha Ali");

    printBook (B1);
    return 0;
}

void printBook (Book * b) {
    printf ( "Book title : %s\n", b->title);
    printf ( "Book author : %s\n", b->author);
}
```

Igual que para cada otro tipo de datos, se pueden definir **punteros a estructuras**

```
struct Book *struct_pointer;
```

La **dirección** de una estructura la obtenemos con el **&**

```
struct_pointer = &B1;
```

Ahora podemos usar el puntero para acceder a los datos miembros de la estructura

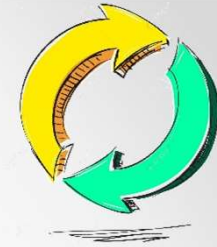
```
(*struct_pointer).title = "ELO 320";
```

Con punteros a estructuras, es más conveniente usar el **->** en lugar de **(*)**

```
struct_pointer->title = "EDA";
```


C-like Strings

Arreglo de caracteres (`char []`) vs. Puntero `char *`



- En C, para operar con **strings** usaremos siempre **arreglos uni-dimensionales de caracteres**. Por ejemplo:
`char msg[] = "ELO320";`
- Pero en algunos ejemplos hemos visto strings definidos de esta forma:
`char* title = "CProgramming";`
- Sin embargo, **NO PUEDO** hacer lo mismo con **title**. Esto es porque:
`char* title = "CProgramming";`
hace que el puntero **title** apunte a un **string constante** que no se puede modificar.
- Si bien puedo hacer que el puntero apunte a un string diferente:

```
char* saludo = "Hola!";  
puts(saludo);  
saludo = "Ciao!";  
puts(saludo);
```

Explicaciones

- msg** es un **arreglo** de caracteres
- title** es un **puntero** a **char**
- Cualquier cosa entre “doble comillas” en C es un **string constante** (string literal)
- al ser un arreglo el **msg**, significa que podemos modificar el valor de cualquiera de sus elementos. Ejemplo: `msg[5] = '1';`
- NO se puede tratar como un arreglo de caracteres**, y por lo tanto no puedo hacerle cambios como este: `saludo[0] = 'c';`
- Por último, recuerden que el **nombre de un arreglo es puntero CONSTANTE** a su primer elemento → **no se puede modificar**

Parámetros por línea de comandos



argc y argv[]

- ¿Sabían que pueden **pasar parámetros desde la línea de comandos** a sus programas en C?
Muchas veces esto es importante, especialmente cuando desea controlar el programa desde afuera.
→ Ejemplo: un uso común es *pasar el nombre de un archivo* para abrir en el programa.
- Para utilizar los argumentos de línea de comando en su programa, primero debe comprender la **declaración completa de la función main**, que previamente en todos los ejemplos no ha aceptado ningún argumento.
- De hecho, **main () acepta dos argumentos !!!** :
 - uno representa el **número de los argumentos** pasados desde la línea de comando,
 - el otro es la **lista de todos los argumentos** pasados desde la línea de comando.

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

Parámetros por *línea de comandos*


argc y argv[]

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

- **argc**: ARGument Count (por lo tanto, argc) es el número de argumentos pasados al programa, **¡incluido el nombre del programa!**
 - (**argv [0]** es el **nombre** del programa, **argv [1]** es un puntero al **primer argumento** pasado por la línea de comandos, etc.).
- El arreglo de punteros a caracteres **argv []** es la lista de todos los argumentos pasados.
 - Si NO se proporcionan argumentos, **argc será igual a uno**

¿y cómo paso los argumentos a mi programa?

- Todos los argumentos pasados desde línea de comando **se ponen después del nombre del programa en cmd separados entre si por un espacio**
 - **Ejemplo:** `program data.txt output.txt`


argv[0] argv[1] argv[2]

Parámetros por *línea de comandos*

argc y argv[]

A continuación se muestra un ejemplo simple en el cual se pueden pasar para controlar el paso correcto de los argumentos esperados por sus programas desde la línea de comando ...

```
#include <stdio.h>

int main ( int argc, char *argv[] ) {
    /*Control de paso correcto de argumentos*/
    if ( argc == 2 ) { //existe un solo parámetro pasado
        printf ("The argument supplied is %s\n", argv[1]);
    }
    else if ( argc > 2 ) { //hay mas de un parámetro
        printf ("Too many arguments supplied.\n");
    }
    else { //caso de ningún parámetro pasado
        printf ("One argument expected.\n");
    }
    return 0;
}
```



Parámetros por *línea de comandos*

argc y argv[]

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

/*Imprime la suma de 2 enteros ingresados por línea de comando*/
int main (int argc, char* argv[]){
    /*Control de paso correcto de argumentos*/
    if ( argc != 3 ) {
        fprintf ( stderr, "Ejecutar: %s <numero1> <numero2>\n", argv[0]);
        exit (EXIT_FAILURE);
    }

    int temp1 = atoi(argv[1]);
    int temp2 = atoi(argv[2]);

    printf("Resultado: %d + %d = %d\n", temp1, temp2, (temp1+temp2));

    return EXIT_SUCCESS;
}
```

File I/O en C

Veamos el concepto de los streams



- La librería `<stdio.h>` en el lenguaje C utiliza lo que se llama "**streams**" para operar con los dispositivos físicos como el *teclado*, *impresoras*, *pantallas* o con cualquier otro tipo de *archivos* admitidos por el sistema.



¿Qué son los streams?

- En realidad, los **streams** son una **abstracción** para interactuar con los dispositivos de manera uniforme, y todos tienen propiedades similares.

¿Cómo se manejan los streams?

- Se manejan en la librería `<stdio.h>` como "**punteros a objetos de tipo FILE**", que identifican un stream y contienen la información necesaria para operar con ellos.
 - Un puntero a un objeto FILE **identifica de forma única un stream** y se usa como parámetro en las operaciones que involucran ese stream (como lectura, escritura, etc.).

File I/O en C

stdin, stdout, stderr



Los streams estándar

- También existen tres streams estándar: **stdin**, **stdout** y **stderr**, que se crean y se abren automáticamente para todos los programas que usan la librería `<stdio.h>`.
- Los **streams estándar de I/O** (**stdin** y **stdout**) le permiten comunicarse con su "consola", la pantalla y el teclado que forman por defecto la interfaz de usuario de la computadora.
- El **stream estándar de error** (**stderr**) es un stream que se comporta como stdout, excepto que su contenido no pasa por un búfer: todos los caracteres enviados a **stderr** se envían inmediatamente a la pantalla.
- La generalidad del concepto de los streams se muestra en la facilidad de **escribir/leer de/a archivos de su computador**
 - Esencialmente, usted hace lo mismo que con la entrada/salida de la consola, pero ya lee/escribe información desde/hacia un archivo.
 - C provee una serie de funciones para trabajar con archivos en la **librería `<stdio.h>`**. Veamos algunos ejemplos:

File I/O en C



Cómo trabajar con archivos en C usando punteros a FILE

FILE *

Hay que usar un puntero a **FILE**, que permitirá que el programa realice un seguimiento del archivo al que se accede.

FILE *fp;

Para abrir un archivo (o crear uno nuevo), se usa la función **fopen()**, que retorna un puntero a **FILE**. Una vez que haya abierto un archivo, puede usar este puntero para realizar operaciones de lectura y/o escritura sobre este.

```
FILE *fopen (const char *filename, const char *mode);
```

```
r - open for reading  
w - open for writing (file need not exist)  
a - open for appending (file need not exist)  
r+ - open for reading and writing, start at beginning  
w+ - open for reading and writing (overwrite file)  
a+ - open for reading and writing (append if file exists)
```

```
FILE *fp = fopen ("c:\\test.txt", "r");
```

OJO con el \\ doble...



File I/O en C



Cómo trabajar con archivos en C usando punteros a FILE

Tenga en cuenta que es posible que **fopen** falle incluso si su programa es perfectamente correcto: puede tratar de abrir un archivo especificado por el usuario, mientras dicho archivo no exista (o esta protegido por el OS). En estos casos, **fopen** devolverá 0, es decir NULL.

Para cerrar un archivo, se usa **fclose**, cuyo prototipo es: **int fclose (FILE *fp);**
Esta función retorna 0 si la operación es exitosa, o **EOF** si hubo algún error.

Se pueden usar las funciones **fprintf** y **fscanf** que, comparándolas con las que conocen, ellas reciben además el puntero **FILE** como primer argumento. Por ejemplo:

```
FILE *fp = fopen ("c:\\test.txt", "w");  
fprintf (fp, "Testing...\n");
```

También se puede usar la función **fputs** cuyo prototipo es: **int fputs(const char *s, FILE *fp);**
Esta función escribe el **string s** al archivo y retorna un valor no negativo si la operación fue exitosa, o **EOF** si hubo algún error.

También **se puede leer/escribir un carácter a la vez**. La función más simple para escribir caracteres es la siguiente:

int fputc (int c, FILE *fp);

que escribe el carácter indicado y además lo devuelve si la operación fue exitosa, o devuelve **EOF** si hubo algún error

File I/O en C

Cómo trabajar con archivos en C usando punteros a FILE



```
/* fputs example */
/* This program allows to append
 * a line to a file called mylog.txt
 * each time it is run.*/
#include <stdio.h>

int main ()
{
    FILE * pFile;
    char sentence [256];

    printf ("Enter sentence to append: ");
    fgets (sentence, 256, stdin);

    pFile = fopen ("mylog.txt","a");
    fputs (sentence, pFile);
    fclose (pFile);
    return 0;
}
```



Standard input stream

`FILE * stdin;`

Se refiere al stream estandar **de entrada** para aplicaciones que en la mayoría de los sistemas es por defecto asociado al teclado.

stdin puede usarse como argumento en cualquiera de las funciones que reciben como parámetro un stream (`FILE*`) para recibir datos, como por ejemplo `fgets` o `fscanf`.

Más ejemplos en:

<http://www.cplusplus.com/reference/cstdio/>

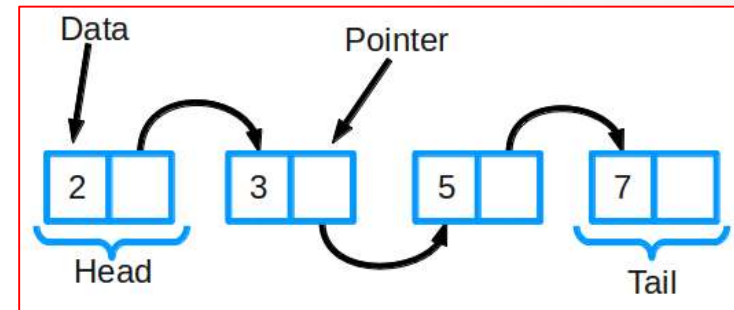


Anticipando las *Listas enlazadas*

estructuras que apuntan a otras estructuras

La lista enlazada es la estructura de datos *más utilizada* después del arreglo.

Una **lista enlazada** es una **secuencia** de **estructuras de datos**, que están conectadas entre sí a través de enlaces (punteros).



Iremos agregando y sacando nodos de la lista **de manera dinámica**, según las necesidades del problema que estamos atacando.

El lenguaje C proporciona varias funciones para la **asignación y gestión** de memoria. Estas funciones se pueden encontrar en `<stdlib.h>`

```
// asigna un conjunto de num bytes  
void *malloc (int num);
```

```
struct node {  
    int data;  
    struct node *next;  
};  
  
struct node *root;  
root = (struct node *) malloc (sizeof (struct node));  
root->data = 5;  
root->next = NULL;  
free (root);
```

```
/* libera un bloque de memoria especificado  
por address. */  
void free (void *address);
```

¿Por qué reservar memoria de forma dinámica?

En el lenguaje C, un **arreglo** es una **colección de elementos** almacenados en ubicaciones de **memoria contigua**.

40	55	63	17	22	68	89	97	89
0	1	2	3	4	5	6	7	8

<- Array Indices

En este ejemplo, el **largo** (tamaño) del arreglo mostrado es 9 y esto **no se puede modificar**. Pero, ¿qué pasa **si hay un requisito para cambiar este tamaño** durante la ejecución de un programa?

```
...  
int size;  
if (tiene_que_cambiar_tamaño ()) {  
    // preguntemos al usuario  
    printf("Qué tamaño quieres que tenga el arreglo?\n");  
    scanf("%d",&size);  
    float miArreglo [size];  
}
```



type arrayName [**arraySize**];

El **arraySize** debe ser una **constante entera mayor que cero**

No puede ser una variable

Gestión *dinámica* de memoria

Definición y manera de hacerlo en mis programas

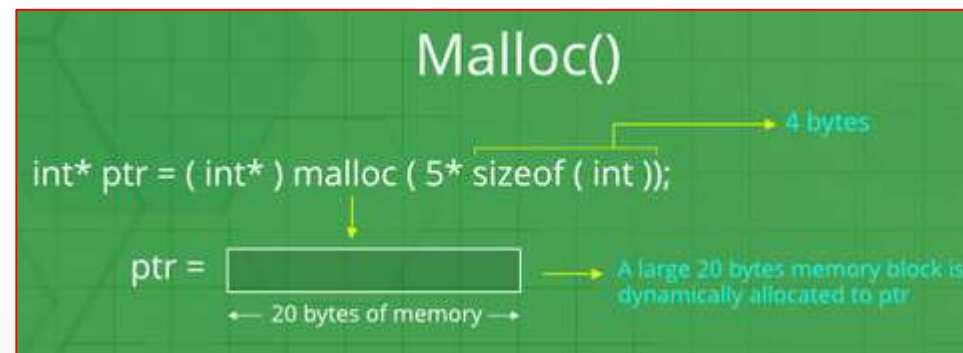
La **gestión de memoria dinámica** se puede definir como el procedimiento en el que el **tamaño** de una estructura de datos (como un arreglo) **se cambia durante el tiempo de ejecución del programa**.

La función "**malloc**" o "memory allocation" (asignación de memoria) se utiliza para asignar dinámicamente un solo bloque de memoria con el tamaño especificado. **Devuelve un puntero de tipo void** que se puede convertir (**type casting**) en un puntero de cualquier tipo.

```
type * ptr = (type*) malloc (num_of_elements * sizeof (type));
```

type casting

Ejemplo:



Gestión *dinámica* de memoria

uso de funciones malloc () y free ()

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

```
int main() {
```

```
    char nombre[20], apellido[20];
    char *info = NULL;
```

```
    strcpy (nombre, "Ioannis");
    strcpy (apellido, "Vourkas");
```

```
    /*allocate memory dynamically for 200 char*/
```

```
    info = (char *) malloc ( 200 * sizeof (char) );
```

```
    if ( info != NULL ) {
        strcpy ( info, "es el nuevo profesor de EDA");
    }
    printf ("Nombre = %s Apellido = %s\n", nombre, apellido );
    printf ("Info: %s\n", info );
```

```
    return 0;
}
```

¿qué contiene
info aquí?

```
// asigna un arreglo de num bytes
void *malloc (int num);

/* libera un bloque de memoria
especificado por address */
void free (void *address);
```

¿Por qué es necesario gestionar la memoria de manera dinámica?

Imaginemos el caso donde *queremos guardar un nombre* en un arreglo **char** nombre [xxx];

- Si sabemos el largo, fijamos **xxx** según sea necesario, e.g. **char** nombre [25];

→ Pero ¿qué tal si se trata de un texto del cual no tenemos idea de su largo?

→ Definimos un **puntero a char** (**char*** es un **C string**) y luego reservamos memoria según sea necesario


Gestión *dinámica* de memoria

Otro ejemplo de uso de `malloc ()` y `free ()`

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

```
int main ()
{
    int largo, n;
    char * buffer;
```

```
printf ("How long do you want the string? ");
scanf ("%d", &largo);
```



```
buffer = (char*) malloc ((largo+1) * sizeof (char));
//if (buffer==NULL) return (-1);
```

```
for (n=0; n<largo; n++)
    buffer[n] = rand()%26 + 'a';
buffer[largo] = '\0';
```



```
printf ("Random string: %s\n", buffer);
free (buffer);
```

```
return 0;
}
```

```
// asigna un arreglo de num bytes
void *malloc (int num);

/* libera un bloque de memoria
especificado por address */
void free (void *address);
```

- Recuerden que el tamaño de un arreglo **NO se puede cambiar** dentro de un programa.
- En este ejemplo de código, se genera un *string* de un *largo* especificado por el usuario, el cual se llena con caracteres.
- Cuando la memoria reservada ya no es necesaria, hay que liberarla con ***free ()***!

¿podemos usar `free ()` con cualquier puntero? NO. Tiene que ser puntero que devolvió antes `malloc` o `realloc`.

Gestión *dinámica* de memoria

uso de funciones `malloc ()` , *realloc* () , y `free ()`

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

```
int main() {
```

```
    char nombre[20], apellido[20], *info = NULL;
    strcpy (nombre, "Ioannis");
    strcpy (apellido, "Vourkas");
```

```
    info = (char *) malloc ( 30 * sizeof (char) );
    if ( info != NULL ) {
        strcpy( info, "es el nuevo profesor de EDA");
    }
```

```
    // aumentar la memoria reservada con realloc ()
```

```
    info = realloc (info, 80 * sizeof (char) );
    if ( info != NULL ) {
        strcat( info, " y es griego");
    }
    printf ("Nombre = %s Apellido = %s\n", nombre, apellido );
    printf ("Info: %s\n", info );
    return 0;
}
```

```
// asigna un arreglo de num bytes
void *malloc (int num);

/* libera un bloque de memoria
especificado por address */
void free (void *address);
```

¿Por qué es necesario gestionar la memoria de manera dinámica?

Imaginemos el caso donde *queremos guardar un nombre* en un arreglo `char nombre [xxx]`;

- Si sabemos el largo, fijamos `xxx` según sea necesario, e.g. `char nombre [25]`;

→ Pero ¿qué tal si se trata de un texto del cual no tenemos idea de su largo?

→ Definimos un **puntero a char** (`char*` es un **C string**) y luego reservamos memoria según sea necesario

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

```
int main (void) {
    int num_of_elements, i, sum = 0;

    printf ("Enter number of elements: ");
    scanf ("%d", &num_of_elements);

    int * ptr = (int*) malloc (num_of_elements*sizeof(int));
    if (ptr == NULL) {
        fprintf(stderr, "Error! memory not allocated.");
        exit(-1);
    }

    printf ("Enter elements: ");
    for (i = 0; i < num_of_elements; i++) {
        scanf ("%d", (ptr + i)); // (ptr + i) es &ptr[i]
        sum += *(ptr + i); // igual a: sum = sum + ptr[i];
    }

    printf ("Sum = %d", sum);
    free (ptr);
    return 0;
}
```

Más ejemplos

Aquí se ve cómo lo que no podría hacer con arreglos convencionales, **lo puedo hacer** asignando a mi programa **memoria** de manera **dinámica**.

```
/* ¿dónde se almacena la memoria
solicitada de manera dinámica? */
```

```
float pi = 3.14;
float * ppi = & pi;
printf ("Address of pi: %u\n", ppi);
printf ("Address of ppi: %u\n", &ppi);
```

```
float *mpi = (float*) malloc (sizeof (float));
*mpi = pi;
printf ("Address of mpi: %u\n", mpi);
/* ¿Qué diferencias observan en las
direcciones que imprimen? */
free(mpi);
```



Introducción a C

Tutoriales y referencias útiles en www



<http://www.cplusplus.com/reference/clibrary/>



- <https://www.cprogramming.com/tutorial/c-tutorial.html>



- <https://www.tutorialspoint.com/cprogramming/index.htm>
- https://www.tutorialspoint.com/data_structures_algorithms/index.htm

programming notes

- https://www3.ntu.edu.sg/home/ehchua/programming/cpp/c0_Introduction.html

Fin de la Clase

Gracias por
su asistencia y atención

...

¿ Preguntas ?